Guided Synthesis and Formal Verification Techniques for Parameterized Hardware Modules.

L.Claesen, P.Johannes, D.Verkest, H.De Man*

Interuniversity Micro Electronics Center IMEC, Kapeldreef 75, B-3030
Leuven, Belgium, Phone +32/16/281203

Abstract

In this paper a new method is presented to be used for either guided synthesis or formal correctness verification of *parameterized* digital hardware modules. It starts from a high level *parameterized* description of the module, which is used as the specification. The method is based on the concept of *correctness preserving transformations*. These transformations are described in a formal way by means of *transformation descriptions*. It ends at a lower level *parameterized* structure description of the implementation. Instead of operating on a derived formalization as done in other approaches, direct manipulations are done on an existing HDL, that emphasizes a strict separation between parameterized structure description and behavior description. The concepts have been applied to real VLSI design vehicles such as a pipelined and parameterized multiplier accumulator module and a systolic implementation of an FIR filter. The methods presented here are amenable for implementation in CAD tools.

1 The Problem of Design Correctness.

Current capabilities of VLSI technology allow that larger and larger systems are being integrated on one chip. Due to this fact the design of these large systems has become very complex. This has given rise to new design methodologies and research to new CAD design tools.

Formerly used methodologies such as top-down or bottom-up as such are not affordable anymore, due to the fact that there is only a limited amount of people in the world that are still able to manage the complete design trajectory from high level system specifications down to MOS transistor circuit techniques. The currently emerging methodology of custom VLSI design is more like a *meet-in-the-middle* [4] approach as shown in figure 1, where two different design groups: (1) system level designers and (2) transistor circuit level designers *meet* each other at the level of *functional modules* such as ALU's, multipliers etc. In order that these modules, designed by silicon oriented experts, can often be used by system designers, it is required that these modules are *flexible*. This is achieved by the ability to generate modules in a parameterized way [5].

During this design activity, there is the problem of guaranteeing or verifying the correctness of the design. The CAD support for this activity is currently very poor. Only for specific classes of applications, automatic synthesis (silicon compilation) from high level behavioral specifications to chip layout can be done, resulting in *correctness by construction*. For the class of digital filters

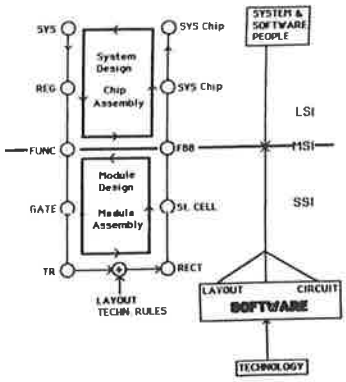


Figure 1: The Meet-in-the-Middle design methodology provides a separation of the design activity at the level of parameterized hardware modules

and general multi-processor digital signal processing systems the feasibility of this approach has successfully been demonstrated by operational systems [2,3].

For several other classes of designs, automatic synthesis is not (yet) feasible, and "manual" system design is still used for most of the applications. This is the case for full custom design as well as for highly optimized circuits such as for example in the video and radar applications. The only tools available for correctness verification are simulators. It is well known that this approach is very error-prone in the discovery of all design errors due to the problem of the choice and interpretation of the appropriate input and output signals. Methods to *formally prove* the correctness would be very useful here.

The CATHEDRAL-II silicon compiler [3] is organized according to the *meet in the middle* principle. An automatic synthesis is done from the high level specification language SILAGE [33] into a number of controllers and execution units which are predesigned as *parameterized modules* such as ALU's, multipliers, dividers, ACU's. The set of required *parameterized module generators* are designed by circuit designers, as a set of LISP procedures [5] automatically generating the circuit layout for the module. The MULTACC (multiplier accumulator) module has the following parameters and *allowable parameter domain* :

- p1 : number of bits in multiplicand X (6, 8, ... 22, 24)
- p2 : number of bits in multiplier Y (6, 8, ... 18, 20)
- p3 : number of overflow bits in accumulator (0, 2, ... 6, 8)
- p4 : presence of a pipeline section (T, NIL)
- p5 : presence of accumulator. (T, NIL)

*Professor at Kath. Univ. Leuven

While the high level synthesis is *correct by construction*, the correctness of the module generators is still being verified using classical verification techniques like logic and circuit level simulation. A further complication in the case of modules is that the correctness has to be guaranteed not only for one instance but for the *whole allowable parameter domain* of the generator procedures. Due to the multiplicity of possible instances that can be generated, verification techniques that act on instances are not appropriate anymore.

The underlying paper addresses new solutions that can be used for guaranteeing the correctness of such parameterized hardware modules.

Section 2 gives an overview of the current approaches for formal correctness verification. In section 3 an outline is given for the method proposed in this paper. The concepts of the HDL used is described in section 4 followed by the description of the equivalence preserving transformations in section 5. Thereafter in section 6 a practical design example of a multiplier accumulator module is considered.

2 Approaches for Formal Correctness Verification.

In software engineering much research has been addressed to the problem of correctness verification [8,9]. Due to the inherent *sequential* character of software programs, these techniques are built on the concepts of pre- and post conditions and the termination properties of software modules. Due to the inherent *parallellism* in hardware design these software verification techniques can not be used as such.

Correctness verification of VLSI designs [22] involves several aspects such as: behavioral, timing, electrical and layout-rule verification. Rule based approaches can be used for electrical correctness verification [18]. For the verification of the correct timing characteristics of MOSVLSI systems, accurate timing verification CAD tools exist [23,24,25]. The layout-rule correctness is guaranteed by using symbolic layout [5] for the construction of the leaf cells. In this paper we only address the problem of behavioral correctness verification of modules as they are used in the CATHEDRAL-II system.

A good overview of the current approaches for behavioral correctness verification of VLSI hardware can be found in [1].

Most of the existing approaches either work out new models for the hardware according to a *functional* description skeleton [16] or instead of the usual HDL descriptions use logical formulas [15] in order to resort on the well-known formal system of the predicate calculus. Besides the use of first order logic, also higher order logic has been investigated [14].

The Boyer-Moore mechanical theorem prover [31] has been successfully applied for the correctness proof of a microprocessor design [30].

For the logic comparison of the functionality of hardware blocks methods have been proposed in references [10,11,12,13,19,20]. These methods are mostly based on heuristics to compare two boolean functions, which is a well known np-complete problem. These approaches have been demonstrated and are very useful on small combinatorial blocks with a few levels of logic. For complicated combinatorial hardware modules such as multipliers, these approaches become impractical.

The methods described above concentrate on hardware that is not parameterized. For the correctness verification of parameterized hardware modules very little has been done yet. In the prolog-based VERIFY system [29] by Barrow, limited parameterizability is possible in the hardware descriptions. Before the start of the proof procedures, the parameters are instanciated with the actual parameter values. This means that for parameterized modules, the proof has to be done for all instances of the allowable parameter domain, which can become very large if not impossible to do. A first approach for the correctness verification of parameterized designs has been reported in [32], relying on the Boyer-Moore theorem prover. However no *actual design example* has been considered yet.

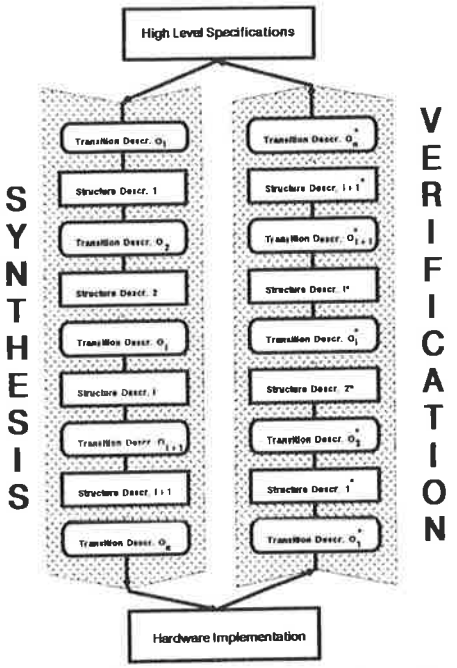


Figure 2: Method of correctness preserving transformations during synthesis and/or verification.

3 Outline of the Synthesis and Formal Verification Method.

In this section an outline is given of the approach presented in this paper. As indicated in fig.2, the method is a step-by-step method based on the concept of *correctness preserving transformations*. The transformations are formalized by means of *transformation descriptions* as described in section 5. The method can as well be used during the synthesis phase as for verification afterwards. Our experience is that it is easier to maintain correctness during the synthesis phase. This is what we will focus further on.

The method is mainly based on correctness preserving manipulations on pure *parameterized structure descriptions* in the HILARICS language as described in section 4.

The method starts from a high level specification in terms of an interconnection of predefined cells, together with the *don't care behavior*. The primitives are *defined* by a *naive* implementation in terms of well known lower level primitives. An example of such a higher level primitive is a *multiplication* that can be defined in

terms of a lower level *naive* implementation based on the shift add algorithm as learned in primary school. The don't care behavior could specify that only positive numbers have to be multiplied. The definition of the naive implementation of a higher level cell is easy to comprehend, but due to hardware and implementation considerations not used as such. Instead a more suited implementation is used, taking into account the allowable hardware, the modularity, the speed considerations, the implementability in terms of the circuit primitives etc..

Starting from that initial specification the designer manipulates the hardware description step by step by means of formalized *transformation descriptions*, until he ends up with the desired implementation in terms of primitives that can directly be realized as circuits in hardware. These are usually small blocks with up to 20 transistors, such as gates or full adders. The correctness of these basic primitives with respect to their layout realization can be verified by using circuit extraction from the (symbolic) layout. From these transistor netlists, the logic equations can be extracted [18,21] and compared [20,19] to the logic equations that come from the behavioral specification of the primitive cell. Starting from the *parameterized structure description* of the implementation, the actual module generator can be created by the

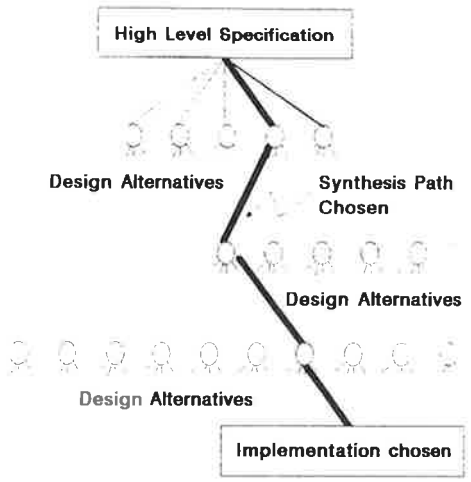


Figure 3: Design history and equivalent design alternatives.

module designer [5]. The formal method presented in this paper ranges from the high level specification until the level of primitive cells.

The method is a step by step correctness preserving method, under the full guidance of the designer who takes all of the *design decisions*. This is in contrast to for example the approach by Barrow in the VERIFY system, where the equivalence between consecutive levels of behavioral specification and implementation is *automatically* verified, without making use of any other information by the designer. It is our belief that by letting the designer *formally* express his elementary design transformations, that the verification problem becomes more manageable, especially for more complex design applications. Automatical systems suffer from the drawback that they have to figure out all of the design transformations on their own. This is especially difficult (possibly unsolvable?) for the general case of parameterized hardware designs.

By keeping track of the elementary design transformations, a designer can maintain a history of the evolution of his design. This has the advantage that a design can be altered from intermediate descriptions as is indicated in figure 3.

4 The hardware description language.

4.1 Considerations on HDL's with respect to formal verification.

Several approaches in formal verification stress the functional modeling aspect, because of the problem that a number of existing HDL's have been derived directly from *imperative* and *procedural* languages. The inappropriate unique modeling of hardware necessitates the use of the funtional model. However, several of the useful HDL's [33,34,35] are *applicative* and are therefore more suitable for direct formal reasoning. This is also the case for pure structure description languages [35] like HILARICS [34] and RTS-1a [36] like register transfer languages and a system level description language like SILAGE [33].

Several of the more sophisticated RTL's provide constructs for accurately modeling the timing behavior for the simulation of the design. This is especially desired for asynchronous circuits. As reported in [15], *these* RTL's can not be used for reasoning. In synchronous VLSI designs, the inclusion of these timing constructs are however not required in behavioral descriptions. We advocate a good separation between the behavior and the timing, such that these aspects can be designed and verified quasi independent from one another. For the timing verification of synchronous designs, timing verification tools are available [23,24,25]. For formal verification only RTL's like RTS1.a [36,37] not cluttered with too much low level timing constructs are appropriate.

The semantic models of the existing HDL's are very familiar to designers, they are a formal way to specify the system under design. In the current design practice they are most often used for simulation purposes.

In contrast to the above mentioned methods, that try to model the hardware based on functional models or by first order logic, we have chosen to directly use the existing HILARICS [34] parameterized structure description language for performing the manipulations. This choice is motivated by the fact that such a language is much closer to a hardware designer than formalisms used by the currently existing proof techniques.

4.2 Semantic model of HILARICS.

We start from the concept that the network structure (composition) should be described completely *independent* from the other aspects (or views) of a design. It is formalized in a structure description language [35,34]. HILARICS [34] is a *parameterized* structure description language. This means that parameterized modules such as an n x m multiplier structure can be described with as parameters n and m.

Parameterizable structure descriptions could be generated from any computer language, but then there would be *no concise semantic model* to rely on for formal verification.

The semantic model of HILARICS is purely applicative. HILARICS describes the following concepts:

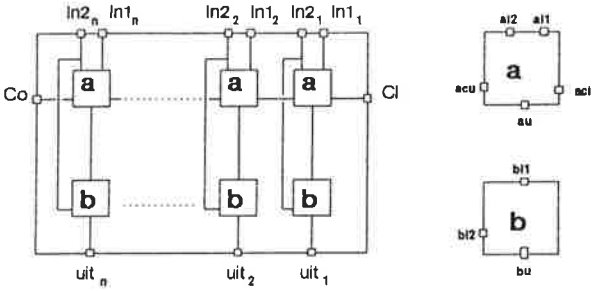
- *cells* : these are the generic building blocks in the design.
- *terminals* : this is the interface of the internal nets of the cell with the higher level cells to which it is connected.
- *components* : these are instances of cells.
- *nets* : these are the sets of interconnected terminals of the

defining cell and the terminals of the components used in the cell.

- *parameters* : The formal parameters are used to define the *parameterized* structure description of the cells. Actual values for components of cells define the specific instances of the components.
- *internal cell variables* : These are *manifest* variables that are calculated at compile time of the structure description for a specific component. They are completely determined by the values of the actual parameters.

HILARICS allows as well for net-oriented as component-oriented descriptions. These are dual descriptions and only one is necessary. They can be described in an applicative way. Besides the generating constructs, the order of the statements is irrelevant. Around most of the statement groups, FOR- and IF ... THEN constructs can be used. FOR-loops define new *index variables* that are known within the FOR construct. The termination conditions for these constructs are defined by the *internal cell variables* and by the *index variables* of outer constructs. Besides the *internal cell variables* and the *index variables* of the FOR-loops no other assignment to variables is possible in HILARICS as it is the case in a normal programming language like for example PASCAL or C.

Notice that HILARICS does not describe any form of behavior. View specific information (not immediately related to the




```
FOR j = 1 TO n DO
   IF j = 1 THEN aci_j = c_i
      ELSE aci_j = acu_{j-1}
   IF j = n THEN acu_j = co
END
FOR j = 1 TO n DO
   ai1_j = in1_j
   ai2_j = in2_j
   au_j = bi1_j
   bi2_j = in2_j
   bu_j = uit_j
END
```

Figure 4: Example of a parameterized cell.

structure) is described separately for the appropriate cells. As such the language is currently in use for the description of the structure part for register transfer descriptions [37] for circuit level simulation, for timing verification, for switch- and logic level simulation and as a definition input for the module generation environment [5]. HILARICS has been used for the description of several hierarchical and parameterized VLSI designs and parts thereof.

Figure 4 depicts an example of a parameterized cell with a parameter n. The cell contains components of two cells a and b. The parameterized structure description looks as follows:

```
CELL Example(n : Integer);
TERMINALS
     ci, in1[1..n], in2[1..n] END : INPUT;
     co, uit[1..n] END : OUTPUT;
END;
COMPONENTS
     a[1..n] : a;
     b[1..n] : a;
CONNECTIONS
     FOR j = 1 TO n DO
          IF j = 1 THEN NET a[j].aci ci END;
               ELSE NET a[j].aci a[j-1].acu END;
          IF j = n THEN NET a[j].acu co END;
     END;
     FOR j = 1 TO n DO
          NET a[j].ai1 in1[j] END;
          NET a[j].ai2 b[j].bi2 in2[j] END;
          NET a[j].au b[j].bi1 END;
          NET b[j].bu uit[j] END;
     END;
END;
```

These kinds of structure descriptions are used to describe the *intermediate* structures in the transformation steps. For a number of usual primitives, naive implementations in terms of HILARICS descriptions should be available. For the specific application of signal processing applications (also includes arithmetic applications), the following additional view information is required per net:

- *arithmetic weight factor* : This information contains the scale factor to be used for certain scalar signal lines. This information is required in order to be able to perform certain equivalence transformations. Two consecutive bits in a binary number have a different scale factor (by a factor of 2 ratio). They may not be interchanged.
- *delay potential* : The delay potential, indicates for synchronous systems, that signals can be represented either in *space* or in *time*. This concept allows to perform retiming transformations [26].

In figure 4 a more concise description with the same meaning as the above one is included. It is on this kind of structure description that equivalence preserving transformations are performed. In section 5 a description of a number of these transformations are given while in section 6 some examples of transformations on a multiplier design will be given.

5 Equivalence Preserving Transformations.

This section describes the basics of the elementary steps for the equivalence preserving transformations. Several aspects are described using parts of the properties and transformations as used for the formal proof of the multiplier module as described in sec-

tion 6. All of the aspects discussed in this section are described in more detail in [28].

In a first subsection the *formalization* of the transformation descriptions and the basic meaning in an example parameterized hardware module is outlined. Hereafter an overview is given of the major classes of equivalence transformations. The last subsection deals with the manipulation rules that act on the parameterized structure descriptions as discussed in section 4.

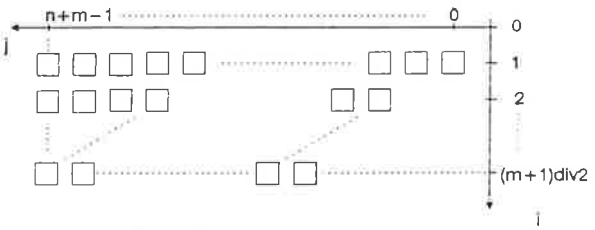
5.1 Transformation descriptions

For automation purposes, the parameterized structure description described in section 4 should be put in a *standard form* according to the principles as described in [28]. For clarity purposes, we do not further elaborate this topic in this paper.

Suppose that figure 5 represents a two-dimensional representation for a module with as parameters m and n. The structure description than has the "standard" form given in the figure.

A transformation description describes:

- which property is to be applied.
- to what part of the hardware module the property is to be applied.

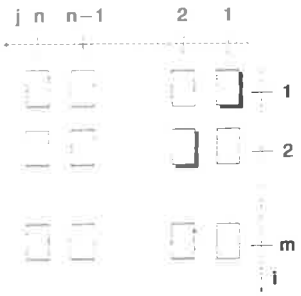



```
FOR i = 1 TO (m+1)div2 DO
  FOR j = 0 TO n+m-1 DO
    IF i = 1 THEN C_{i,j} = 0
      ELSE IF j ≥ 2(i-1) THEN C_{i,j} = K_{i-1,j-1}
        ELSE nil
      END
    END
    IF i = 1 THEN M_{i,j} = ExternalNets
      ELSE IF j ≥ 2(i-1) THEN M_{i,j} = P_{i-1,j}
        ELSE nil
      END
    END
  END
END
```

Figure 5: Two dimensional structure and standard description.

The properties are *equivalence transformations*. If they are applied to a specific structure, a new structure is obtained that is functionally equivalent with the original one.

Only *one property* may be specified per transformation description. Such property may act on more components of the structure description. The above restriction stems from the fact that each elementary transformation description is *applicative*. This allows us to do the code manipulation in an easy way. Introducing more properties in the same transformation description would make the transformation description *procedural* and would result in much more complicated proof techniques.




```
FOR i = 1 TO m DO
  FOR j = 1 TO n DO
    IF i = j THEN property(i, j, k, t, ...)
      ELSE nil
    END
  END
END
```

Figure 6: Example of the form of a formal transformation description on a two-dimensional parameterized structure. The shadowed rectangles indicate the components that are influenced by the transformation description.

The form of an example *formal* transformation description and the structure on which it acts is given in figure 6.

This transformation description manipulates the diagonal blocks with property *property*. *Transformation descriptions* can have the same FOR- and IF- constructs as the *structure descriptions* in section 4. The basic items are now *properties* instead of *nets*.

5.2 Classes of equivalence preserving transformations.

Depending on the hardware functionality of the components, properties can be applied in *equivalence transformations*. For each of the properties, there are a number of *conditions* that must be fulfilled by the structure description and its components before

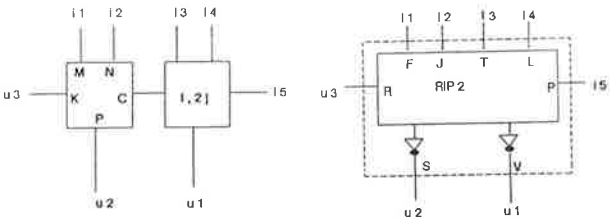


Figure 7: Two Boolean equivalent cells that can be checked by a tautology checker.

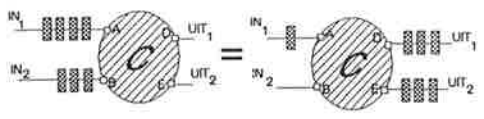


Figure 8: Retiming equivalence transformation

a property may be applied. For the purposes of formal verification, *cells* require properties, that allow to determine that particular equivalence transformations may be applied to components of that cell. In the DSP-like applications considered up to now the following classes of properties can be considered: arithmetic properties, boolean properties and flow graph properties. They are introduced in the next paragraphs.

5.2.1 Arithmetic properties.

These can be applied for arithmetic building blocks like e.g. full adders. An example is *commutativity*. The conditions for this property are that the components must allow commutativity, the signals must have the same arithmetic weight factor and the same time potential. This property is used several times in the formal verification of the multiplier module: for example in the step from carry propagate to carry save structure.

5.2.2 Boolean properties.

These properties act on *Boolean operators*, as are most of the basic components in VLSI hardware. Here the classical rules of Boolean algebra can be used. In an automated CAD environment several of the Boolean properties can be automatically verified by using a tautology checker [19,20]. This will most often be required for the equivalence proof of lower level hardware modules.

Figure 7 shows a good example for the use of a tautology checker that occurs in the case of the multiplier module described in section 6. Here the equivalence transform from two full adders into one dedicated two-bit adder cell is proven by a tautology checker. The implementation of two-bit adder cells has been motived by hardware efficiency considerations in the multiplier module. The condition here is that the logic functionality of the new components and their interconnection must be equivalent to that of the original components and their interconnection. Notice that for the equivalence proof of these smaller building blocks no parameterized descriptions are required.

5.2.3 Flowgraph properties.

Examples of these properties are delay management [27] and retiming operations [26] shown in figure 8. These properties are used in the organization of the internal pipeline of the multiplier discussed in section 6.

A trivial operation is the *remove* operation, that is schematically depicted in figure 9. Here a component b of cell f is removed, because it occurs twice. The conditions for this operation are that there have to be two components of the same cell, and that the inputs for the two components have to be identical. This is a property that is used in the multiplier design in the transformation from the *naive* implementation to a more optimized implementation. The dual operation is the *add* property.

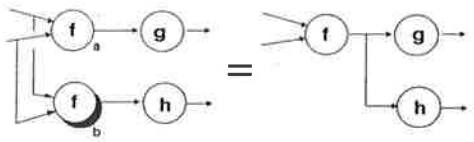


Figure 9: Remove equivalence transformation property.

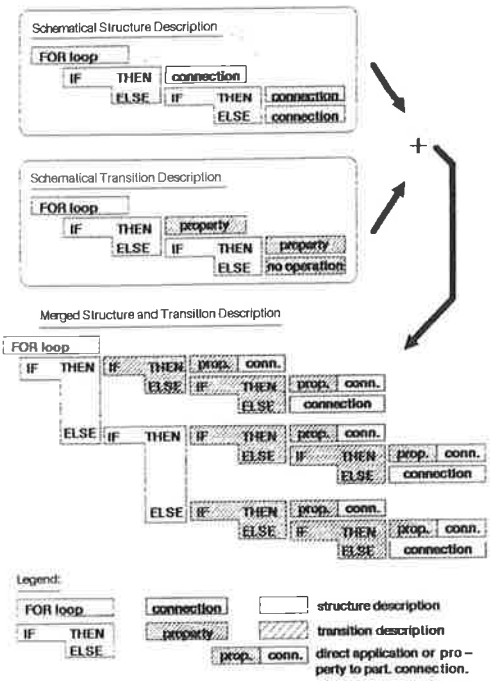


Figure 10: Schematical representation of the merging of a structure description together with a transformation description.

Other general properties are *reindexing operations* that do not change the functionality, but only the way in which components and interconnections are indexed (*by i and j for example*) in the parameterized structure description.

5.3 Manipulation rules.

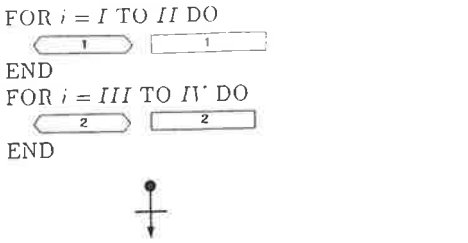
The indices of components and nets in parameterized structure and transformation descriptions indicate particular structural items. The structure description indicates the static composition of a cell. The transformation description indicates how a cell is recomposed in an equivalent way. From this follows that a new structure description of a functionally equivalent cell can be *calculated* by *merging* both the structure description and the transformation description in a new description. In the method we propose, this merging is done by applying the transformation descriptions to *each* individual net description in the structure description. This is schematically indicated in figure 10.

Hereafter a manipulation of the new structure description is needed in order to obtain a much more concise structure description. In order to make the transformation description applyable to specific net definitions in the structure description, often a recalculation of the indices is necessary.

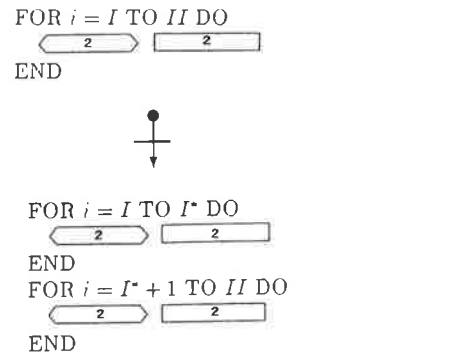
In order to be more suited for CAD automation, the structure and transformation descriptions have to be reorganized according to standard descriptions as described in [28]. Some of the *code manipulation* rules are given below[1] as an example. A more extensive description is given in [28].

[1]The polygons in the description indicate the deeper nested FOR and IF constructs. The rectangles indicate the primitive net descriptions

5.3.1 Joining of FOR loops.

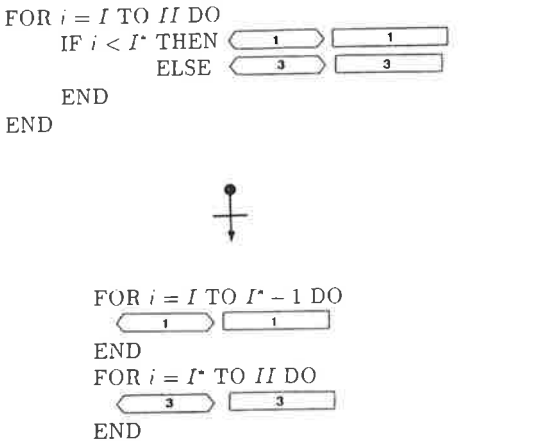


5.3.2 Splitting of FOR loops.



The requirement here is that $I \leq I^* \leq II$.

5.3.3 Mixed FOR and IF construct splitting



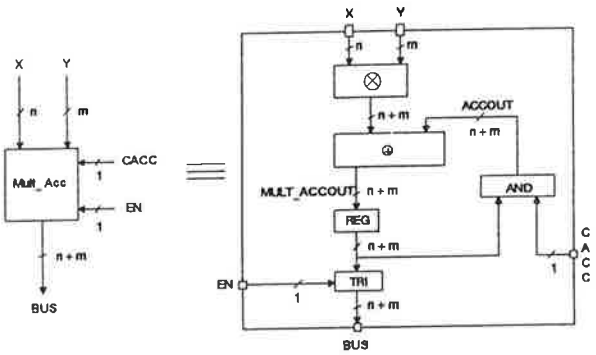
6 Application Example: Correctness of a Module Generator for a Parameterized Multiplier Module.

A *parameterized* multiplier module [7] is a basic building block used in the CATHEDRAL-II system [3]. The concepts described in this paper have been applied for the formal correctness proof of the *parameterized structure description* of the multiplier [28]. The *high level behavioral specification* for the multiplier is given schematically in figure 11, together with its RTL description [37].

This high level specification consists of a basic multiplication of an n-bit multiplicand X and an m-bit multiplier Y to form an n+m bit result that is conditionally (depending on CACC) accumulated.

Notice that this high level specification makes lots of abstractions of the final implementation. The implementation contains much more detail of information for the generation of the parameterized structure, which is motivated by hardware and implementation considerations. The operations MPY and ADD in the RTL [37] description are *defined* in terms of naive implementations, that do not necessarily correspond to the actual implementation.

For reasons of hardware efficiency, it has been decided that *Booth multiplication* is used for the MOS implementation of the MPY operation in figure 11. This results in the structure of figure




```
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
```

Figure 11: High level behavior of multiplier module.

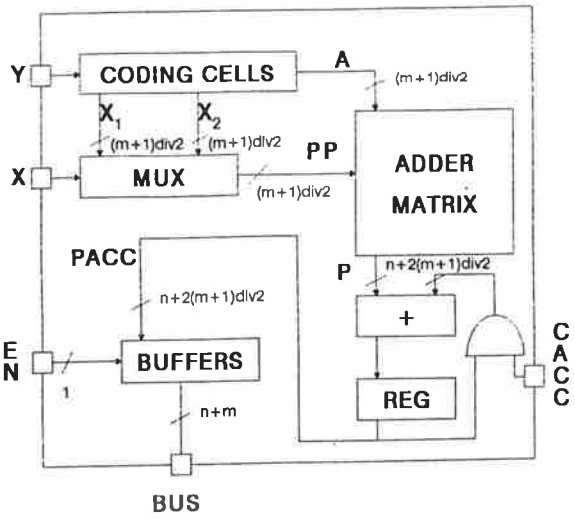


Figure 12: Refined implementation for the multiplier accumulator module.

13. The correctness of the Booth multiplication algorithm with respect to the straight forward definition of MPY is proven in reference [6]. In the further discussion we will concentrate mainly on the Booth multiplier array and the accumulator structure. The *naive implementation* of the Booth algorithm [6] is schematically indicated in figure 13 for an 8 x 8 multiplier, together with the leaf cells used therein. The signals A and PP correspond to the signals from the Booth decoder in figure 12. To this initial structure corresponds a *parameterized* structure description given in detail in [28].

By using 15 elementary transformation descriptions the initial *parameterized* structure description is transformed into the final *parameterized* structure that corresponds to the final implemen-

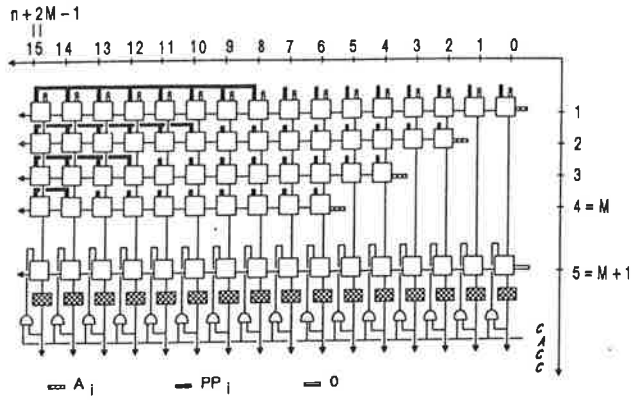


Figure 13: Initial structure naive implementation of a Booth multiplier

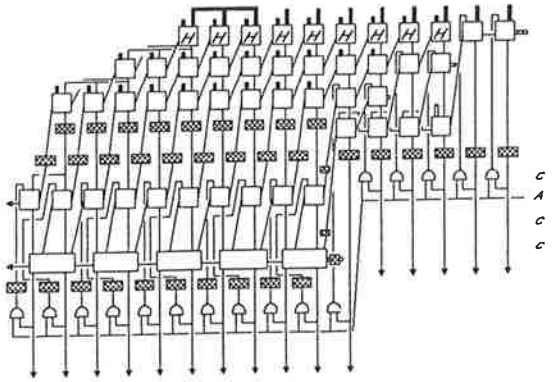
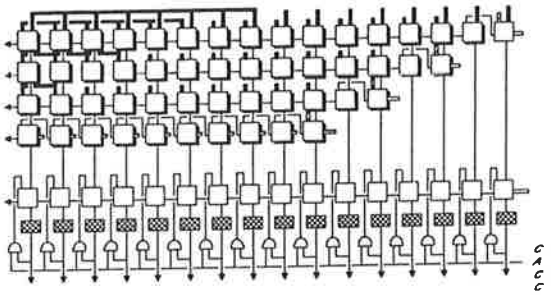


Figure 14: Final implementation structure for multiplier.




```
FOR i = 1 TO M DO
  FOR j = 2(i − 1) TO n + 2M − 1 DO
    IF i ≠ 1 THEN COMM(C_{i,j}; C_{i−1,j})
      IF j < 2i THEN COMM(C_{1,j}; C_{i,j})
      END
    END
    IF j ≥ 2M THEN COMM(C_{1,j}; C_{M,j})
    END
  END
END
```

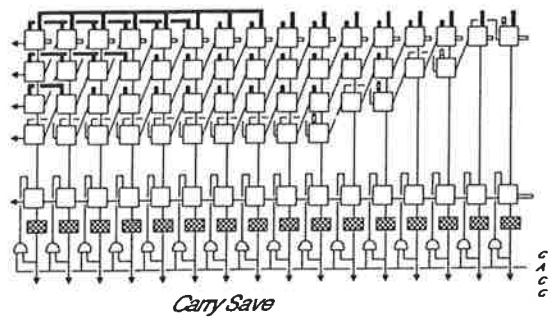


Figure 15: Equivalence transform from carry ripple to carry save addition.

tation. An 8 x 8 instance of the final implemenation is shown in figure 14.

These transformation descriptions consist a.o. of *remove*, *reindex*, *commutativity*, *exchange* operations. In figure 15 the transformation from *carry-ripple* to *carry-save* addition is illustrated. The upper part of the figure indicates the structure before the operation. The components with shadows are effected by the transformation description in the middle of the figure. The lower part of the figure shows the structure after the equivalence transformation.

Due to the *parameterized* nature of the *structure* and *transformation* descriptions at certain equivalence transforms a proof by *induction* on the parameters is required. This is for example the case in the transformation from the original carry save structure to the partially eliminated carry save structure.

A *parameterized* layout module generator for the multiplier accumulator, corresponding to the final *parameterized* structure description in figure 14 has been designed in the MGE environment [5]. Figure 16 shows the layout of a multiplier instance of 8 x 8 bits.

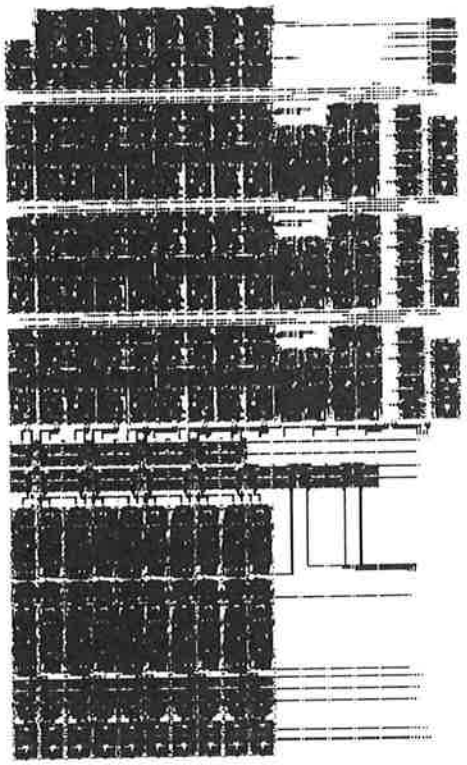


Figure 16: Automatically generated module layout of a 8 x 8 multiplier.

7 Conclusions and Future Research

In this paper we have presented a synthesis and/or verification method that is built on the concept of *correctness preserving transformations*. Transformations are performed on *parameterized* structure descriptions in order to come from a specification up to an actual implementation. In the method the correctness preserving transformations are formalized as *transformation descriptions*. The transformations are applied directly on an existing *parameterized structure description language*. The method has been applied [28] for the formal correctness verification of a parameterized booth-multiplier module and a systolic implementation of an FIR filter.

Future research will concentrate on applying the same principles as outlined above to include manipulations on RTL constructs [36,37] and on high level system specifications in the SILAGE language [33]. In the latter case it will be very useful for the *guided* synthesis of high speed video type signal processing circuits.

Acknowledgements

We hereby would like to thank our colleages Dirk Lanneer, Ivo Vandeweerd and Johan Dries who actually designed the module generator for the multiplier accumulator module that has been discussed in this paper.

References

[1] P.Camurati, P.Prinetto, "Formal verification of hardware correctness: an introduction", *Proceedings IFIP 8th International Symposium on Computer Hardware Description Languages and their Applications*, CHDL-87, April 27-29, 1987, pp. 255-247.

[2] R.Jain, F.Catthoor, J.Vanhoof, B.De Loore, G.Goossens, N.Goncalvez, L.Claesen, J.Van Ginderdeuren, J.Vandewalle, H.De Man, "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*, Volume SC-21, Number 1, February 1986, pp. 73-85, also published in *IEEE Transactions on Circuits and Systems*, Vol. CAS-33 Number 2., February 1986, pp. 73-85.

[3] H.De Man, J.Rabaey, P.Six, L.Claesen, "CATHEDRAL-II: A Silicon Compiler for Digital Signal Processing", *IEEE Design & Test of Computers*, December 1986, Vol.3, No.6, pp. 13-25.

[4] H.De Man, "Evolution of CAD tools towards third generation custom VLSI design", *Revue Phys. Appl. 22*, Vol.22, January 1987, pp. 31-45.

[5] P.Six, L.Claesen, J.Rabaey, H.De Man, "An Inteligent Module Gernator Environment", *Proceedings of the 23rd Design Automation Conference*, Las Vegas, June 29-July 2, 1986, pp. 730-735.

[6] L.P.Rubinsfield, "A Proof of the Modified Booth's Algorithm for Multiplication", *IEEE Trans. on Computers*, October 1975, pp. 1014-1015.

[7] J.Dries, "Ontwerp van een geparameterizeerde vermenigvuldiger akkumulator module", *Eindwerk K.U.Leuven, Departement Electrotechniek* Belgium, July 1986.

[8] C.A.R. Hoare, "An axiomatic approach to computer programming", *Communications of the ACM*, 12(10), 1969, pp. 576-583.

[9] R.C. Backhouse, "Program Construction and Verification", *Series in Computer Science, Prentice Hall Int.*, ed. by C.A.R. Hoare, 1986, ISBN 0-13-729153-1.

[10] K.J. Supowit, S.J. Friedman, "A New Method for Verifying Sequential Circuits", *Proceedings 23th Design Automation Conference*, Las Vegas, 1986, pp. 200-207.

[11] G.Odawara, M.Tomita, O.Okuzawa, T.Ohta, Z-Q.Zhuang, "A Logic Verifier Based on Boolean Comparson", *Proceedings 23th Design Automation Conference*, Las Vegas, 1986, pp. 208-214.

[12] W.Grass, N.Schielow, "VERENA, a program for automatic verification of the refinement of a register transfer description into a logic description", *Proceedings Computer Hardware Description Languages and their Applications*, CHDL-85, ed. C.J.Koomen, T.Moto-oka (eds.) IFIP, Elsevier Science Publishers B.V. (North-Holland).

[13] N.C. Srinivas, "PROVE : PROLOG based verifier", *IEEE International Conference on Computer Aided Design*, ICCAD-86, Santa Clara, Nov. 11-13 1986, pp.306-309.

[14] M.Gordon, "Why higher-order logic is a good formalism for specifying and verifying hardware", *in Formal Aspects of VLSI Design*, editors: G.Milne, P.Subrahmanyam, Elsevier Science Publishers B.V., 1985, pp. 153-178.

[15] H.Eveking, "Formal Verification of Synchronous Systems", *in Formal Aspects of VLSI Design*, editors: G.Milne, P.Subrahmanyam, Elsevier Science Publishers B.V., 1985, pp. 137-152.

[16] R.T.Boute, "Functional Description of Digital Systems", in *Methodologies for Computer System Design*, edited by: W.K. Giloi and B.D. Shriver, Elsevier Science Publishers B.V. (North-Holland), IFIP, 1985, pp. 291-306.

[17] R.L.Spickelmier, A.R.Newton, "WOMBAT: A new netlist comparison program", *Proceedings International Conference on Computer Aided Design*, ICCAD-83, Santa Clara, 1983, pp. 170-171.

[18] H.De Man, I.Bolsens, E.Vanden Meersch, J.Van Cleynenbreughel, "DIALOG: An Expert Debugging System for MOSVLSI Design", *IEEE Transactions on Computer Aided Design*, CAD-4, No.3, June 1985, pp. 303-311.

[19] G.D.Hachtel, 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 7-18, 1986.

[20] P.Lammens, "TC: a tautology checking module", Internal report IMEC/1058/B.a.1.6/2, IMEC Leuven Belgium.

[21] R.E. Bryant, "Papers on a Symbolic Analyzer for MOS Circuits", *Research Report No. CMUCAD-86-20*, SRC-CMU Research Center for Computer-Aided Design, Dept. of Electr. and Comp. Eng. Carnegie Mellon University, PA 15213, September 1986.

[22] L.Claesen, H.De Man, I.Bolsens, W.De Rammelaere, D.Dumlugol, P.Lammens, P.Odent, R.Severyns, E.Vanden Meersch, "Electrical, Timing and Behavioral Verification in the Meet-in-the-Middle MOSVLSI Design Environment of CATHEDRAL-II", *Proceedings International Conference on Computer Design: VLSI in computers & processors*, ICCD-87, Port Chester, New York, Oct.5-Oct.8, 1987.

[23] J.K. Ousterhout "A Switch-Level Timing Verifier for Digital MOS VLSI", *IEEE Transactions on Computer-Aided Design*, July 1985.

[24] E.Vanden Meersch, L.Claesen, H.De Man, "SLOCOP: A timing verification tool for synchronous CMOS logic", *Proceedings European Solid State Circuits Conference*, ESSCIRC-86, Delft, September 16-18, 1986.

[25] J.Benkoski, E.Vanden Meersch, L.Claesen, H.De Man, "Efficient Algorithms for Solving the False Path Problem in Timing Verification", *Proceedings IEEE International Conference on Computer-Aided Design*, ICCAD-87, Santa Clara CA, November 9-12, 1987.

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

[27] A. Fettweis, "A General Theorem for Signal Flow Networks with Applications", *Arch. Elek. Ubertragung.*, Vol. 25, 1971, pp.557-561.

[28] P.Johannes, D.Verkest, "Computer aided synthesis of parameterized VLSI hardware modules", (in Dutch) Engeneering thesis Kath. Univ. Leuven, *IMEC Leuven Belgium*, July 1987.

[29] H.Barrow, "VERIFY: A Program for Proving Correctness of Digital Hardware Designs", *Journal of Artificial Intelligence*, Vol.24, 1984, pp. 437-491.

[30] 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.

[31] R.S.Boyer, J.S.Moore, "A Computational Logic", *Academic Press*, New York, 1979, ISBN 0-12-122950-5.

[32] S.M.German, Y.Wang, "Formal Verification of Parameterized Hardware Designs", *Proceedings IEEE International Conference on Computer Design: VLSI in Computers*, ICCD-85, October 1985, pp. 549-552.

[33] P.N.Hilfinger, "A High-Level Language and Silicon Compiler for Digital Signal Processing", *Proceedings IEEE 1985 Custom Integrated Circuits Conference*, CICC-85, pp. 213-216.

[34] E.Vanden Meersch, R.Severyns, "HILARICS: User's Manual, 2nd edition", *Internal report IMEC, MR03-KUL-?-R?-2*, January 1986.

[35] W.M. VanCleemput, "A hierarchical language for the structural description of digital hardware", *ACM IEEE 14th Design Automation Conference*, Albuquerque, NM (USA), July 1984

[36] H.J. Knobloch, "Description and simulation of complex digital filters by means of the register transfer language RTS 1a.", in *Computer Design Aids for VLSI Circuits*, edited by P.Antognetti, D.O.Pederson, H.De Man, published by: Sijthoff and Noordhof 1981, Aphen aan den Rijn, The Netherlands.

[37] R.Severyns, E.Marien, "LOGMOS user's guide", *Internal report IMEC*, 1986.

BRUSSELS
April 11-14, 1988
CAMPUS V.U.B.
(Vrije Universiteit Brussel)

SYSTEM DESIGN: CONCEPTS METHODS AND TOOLS

Computer Society Order Number 834
Library of Congress Number 87-83546
IEEE Catalog Number 88CH2548-6
ISBN 0-8186-0834-X

Sponsored by
Computer Society of the IEEE
IEEE Region 8
IEEE Benelux
Euromicro

Supported by
Vrije Universiteit Brussel
Universite Catholique de Louvain
Philips Research Laboratory Brussel
EUREL

THE COMPUTER SOCIETY OF THE IEEE
IEEE THE INSTITUTE OF ELECTRICAL AND ELECTRONICS ENGINEERS, INC.
COMPUTER SOCIETY PRESS

pp 30-35