PROGRAM TRANSFORMATION OF HARDWARE DESCRIPTIONS BY MEANS OF ILP.

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

IMEC†Kapeldreef 75, B-3030 Leuven, Belgium, Phone +32/16/281216

ABSTRACT

In this paper a new technique is presented for application of correctness preserving transformations to designs, described in an existing hardware description language. A new program transformation technique is used, based on Integer Linear Programming (ILP), to calculate automatically the structure description of the transformed design given the original designs' description and a transformation description. This technique is used for refining designs towards efficient implementations. It has been applied to the design of a systolic FIR filter and of a parameterized mutliplier-accumulator module used in the Cathedral-II silicon compilation system.

INTRODUCTION

The CATHEDRAL-II silicon compiler [1] is organized according to the meet-in-the-middle principle [2]. An automatic synthesis is done from the high level specification language SILAGE [3] into a number of controllers and execution units which are predesigned as parameterized modules. Examples of parameters are: the wordlength of the inputs, the number of busses, ... The set of required module generators is designed by circuit specialists as a set of LISP procedures [4], automatically generating the circuit layout for the module when instanciated with the appropriate parameter values.

While the high level synthesis is assumed to be 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 module generators 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. This paper describes a new technique that can be used to guarantee the functional correctness of such parameterized hardware modules.

For formal verification of *parameterized* hardware modules general theorem provers can be used. Researchers like Hunt [5] and Gordon [6] resort to well known formalisms of predicate calculus and higher order logic to describe hardware. The Boyer-Moore theorem prover [7] has been succesfully applied by Hunt for the correctness proof of a microprocessor [5] and by German for the verification of a parameterized comparator [8]. The HOL theorem proving environment [9] has been used by Cohn [10] and by Joyce [11] to conduct proofs concerning microprocessors. A common drawback [11] of these general purpose theorem provers is that the construction of the proof is a tedious task. Also the proof has to be well understood in advance.

This drawback can be avoided by adopting a transformational design style: correctness preserving transformations [12] are used to refine designs towards efficient implementations. Sheeran developed μFP [13] and more recently Ruby [14] to support such a design style. μFP and Ruby are two formal languages specially developeResearchers for the purpose of formal verification. However, Borrione [15] showed that it is possible to use an existing Hardware Description Language (HDL) for the purpose of formal verification. The motivation for using existing HDLs stems from the fact that they are more familiar to hardware designers than the formalisms mentioned above. The parameterized hardware modules to be verified, are described in an existing HDL and the technique presented is used in a transformational design strategy [16]; the designer specifies the transformations he wants to apply to the design by means of transformation descriptions. Both the transformation description language and the HDL will be described in the following section. The technique is based on a geometrical interpretation of the structure and transformation description as will be explained later on. The implementation of the technique is based on the ILP algorithm, which will be discussed in the last section.

HARDWARE AND TRANSFORMATION DESCRIPTION

The HDL is called HILARICS [18]. HILARICS starts from the concept that the network *structure* should be described completely independent from the other aspects of a design. It is an applicative language in which designs can be described hierarchically and in a parameterized way. HILARICS has been used for the description of several hierarchical and parameterized VLSI designs. As a working example the structure description of the piece of hardware of figure 1 is given. The minimum value of the parameter p is three. The drawing in figure 1 is made for two specific values of the parameter (p=3 and p=7). The HILARICS description is given in figure 2.

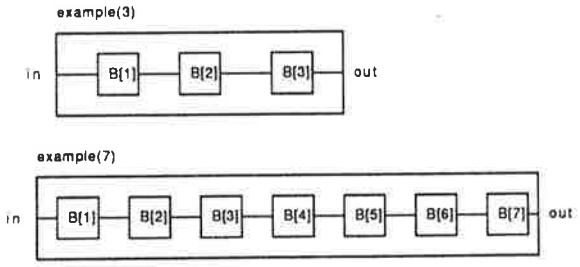

Figure 1: Working example: a simple row of building blocks B

*Professor at K.U.Leuven

†Interuniversity Micro Electronics Center

ISCAS '89

CH2692-2/89/0000-1174 $1.00 © 1989 IEEE

```
CELL example (p)

 TERMINALS in out

 COMPONENTS
  IF p≥3 THEN FOR x=1 TO p DO B[i] END
         ELSE NOCOMP
  END

 CONNECTIONS
  IF p≥3 THEN FOR x=1 TO p DO
                IF x≥1 THEN in = B[x].i
                       ELSE IF x≥p-1 THEN B[x-1].o = B[x].i
                                     ELSE B[x-1].o = B[x].i
                                          B[x].o = out
                            END
               END
              END
         ELSE NONET
  END
 END
```

Figure 2: The structure description of the working example

A *transformation* description indicates two things:

- which primitive equivalence transformation is used,
- and to which part(s) of the design this primitive transformation applies.

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, if and only if the original description fulfils certain conditions depending on the primitive transformation being used. In the DSP-like applications considered so far, the following classes of primitive equivalence transformations can be considered:

- primitive arithmetic transformations can be applied to arithmetic building blocks as e.g. full adders.
- primitive boolean transformations are used to replace boolean logic. The correctness of these transformations is checked by a tautology checker.
- primitive flowgraph transformations like delay management and retiming operations.

More details on the transformations and the transformational design method can be found in [17].

As an example of a transformation we will substitute some blocks B of figure 1 by different, but functionally equivalent, blocks NEWB. This transformation will only have to be executed if the parameter 'p' is larger than four. The transformation description to achieve this is given below and the corresponding transformed structure is given in figure 3 (the NOP operation indicates that no transformation has to be executed).

```
TRANSFORMATION example (p)

 IF p≥5 THEN IF x≥3 THEN IF x≤p-1 THEN SUBST(B[x]NEWB[x])
                                  ELSE NOP
                        END
                   ELSE NOP
          END
     ELSE NOP
 END
```

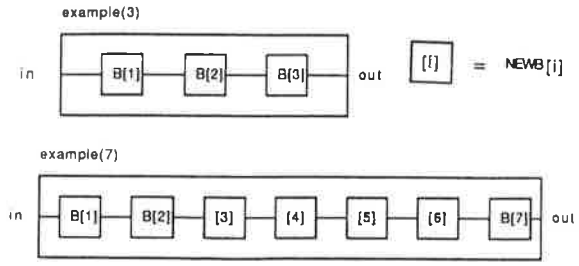

Figure 3: The working example after transformation

GEOMETRICAL INTERPRETATION

As a shorthand notation for the structure and transformation description we use a binary tree. The leafs of the tree correspond to the net definitions resp. primitive transformations. The vertices contain the tests of the `IF THEN ELSE` constructs. The left edge of a vertex corresponds to the `THEN` case of the `IF THEN ELSE` construct in that vertex and the right edge corresponds to the `ELSE` case.

The conditions found in the vertices can be seen as equalities (or inequalities) defining hyperplanes (or hyperspaces) in an N-dimensional space; N is the number of indices and parameters concerned in the structure description. By following a path from the root to a leaf L of such a tree, the vertices (conditions) encountered, define a polytope whose boundary planes are given by the conditions in the vertices. For all integer points within this polytope, the net definition of the leaf L is active, i.e. the nets defined by the definition in the leaf are present in the structure. In the case of the transformation description, the primitive transformation defined in the leaf L has to be executed for all integer points within this polytope.

Figures 4 and 5 show the tree representation and the geometrical interpretation of resp. the structure description and the transformation description of the `example` design. To calculate the

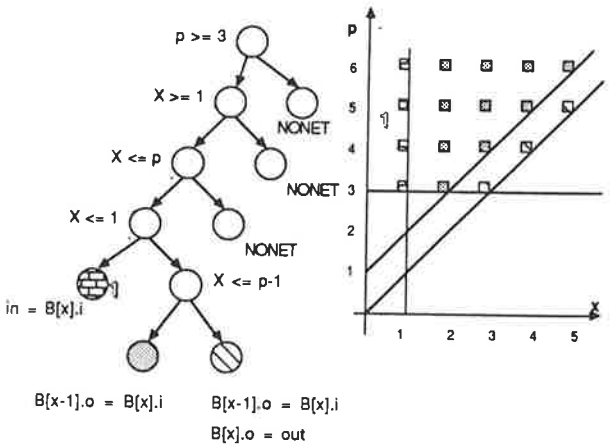

Figure 4: The structure tree and its geometrical interpretation

description of the transformed structure we will have to find out if a particular net definition of the structure tree has to be transformed by a particular transformation definition of the transformation tree. This can be done by looking for intersections of the polytopes in which the respective definitions are active. For all integer points in such an intersection both the net and the transformation definition are active, meaning that the particular net has to be transformed

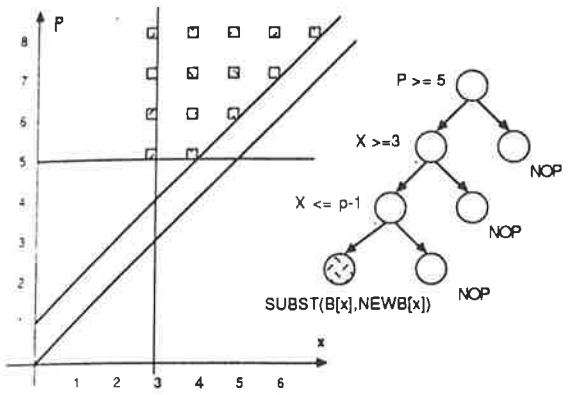

Figure 5: The geometrical interpretation of the transformation tree

by the particular transformation. These intersections can be determined by inserting the complete transformation tree in front of each leaf of the structure tree. The tree obtained in that way is called the *merged* tree. Each root-leaf path RL of the merged tree describes exactly one intersection (see figure 6). Three cases can

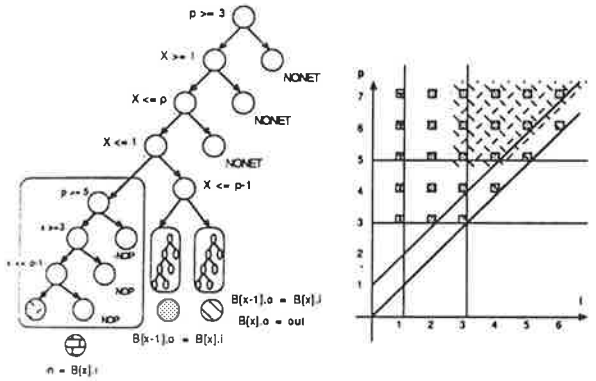

Figure 6: The merged tree and its geometrical interpretation

be distinguished:

1. RL defines a region in which both net definition and transformation definition are active: the intersection of the two polytopes. In this region the new net definition can be calculated from the old net definition and the transformation definition.
2. RL defines a region in which the net definition is active but the transformation definition is not: there is no intersection. The old net definition doesn't have to be transformed.
3. RL defines a region in which the net definition is not active.

The tree representing the transformed design can be derived directly from this merged tree, using the techniques described in the next section. This transformed tree is shown in figure 7 and it corresponds to the drawings of figure 4.

ILP TECHNIQUES

The problem is now reduced to:

- determine wether the two polytopes have an intersection.
- if they have an intersection determine the minimal set of (in)equalities describing that intersection.

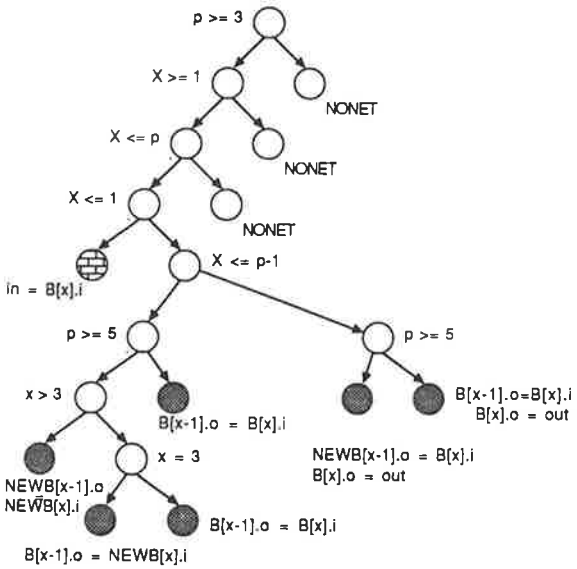

Figure 7: The structure tree of the transformed design

Intersection of polytopes

In the first stage we are given a number of (in)equalities that describe an intersection and we want to determine if that intersection is non-empty, i.e. if the (in)equalities contain a feasible solution. Since we are dealing with hardware descriptions the solution will be made up of integers (parameters and indices). This problem can be mapped on the ILP problem (pp.307 of [19]). An algorithm to solve this consists of three parts:

1. determine an initial feasible solution by solving a slightly different LP problem by means of the *simplex* algorithm.
2. solve the relaxed LP problem (i.e. identical to the ILP problem but without the constraint for an integer solution) by means of *simplex*.
3. iterate to an optimal and integer solution by means of the *cutting plane* algorithm.

For our problem we can leave out the second step: we only want to know whether the set of constraints allows for a feasible solution - meaning the two polytopes have an intersection - and we don't need an optimal solution (for that matter we don't have a goal function to optimize either).

After this first stage we know whether there is an intersection in the root-leaf path considered. If there is no intersection we check the following root-leaf path of the merged tree. If there is an intersection we proceed with the second stage to eliminate redundant (in)equalities to obtain a minimal set describing the intersection.

Elimination of redundant equations

We have a sequence of (in)equalities determining an intersection and we want to eliminate all superfluous ones. We suppose that there are no redundant (in)equalities among those of the structure tree and then we check one by one the (in)equalities of the transformation tree. A redundant equation is an (in)equality which is always false or always true given the previous (in)equalities. We use the ILP algorithm to decide about the redundancy of the last

(in)equality in a sequence of (in)equalities by means of the following technique (see figure 8):

- if the ILP finishes without a feasible solution then the last (in)equality is always false and thus can be deleted. However, since we first checked that there was indeed a solution, this situation will not occur.
- if there exists a feasible solution we restart the ILP, but with the last (in)equality inverted. If there is still a feasible solution then the last (in)equality is not redundant (figure 8.a). If there is no feasible solution then the last (in)equality is always true and can be deleted (figure 8.b).

By proceeding in this way we can eliminate all the redundant (in) equalities and prune the merged tree to obtain the tree representing the transformed structure (see figure 7).

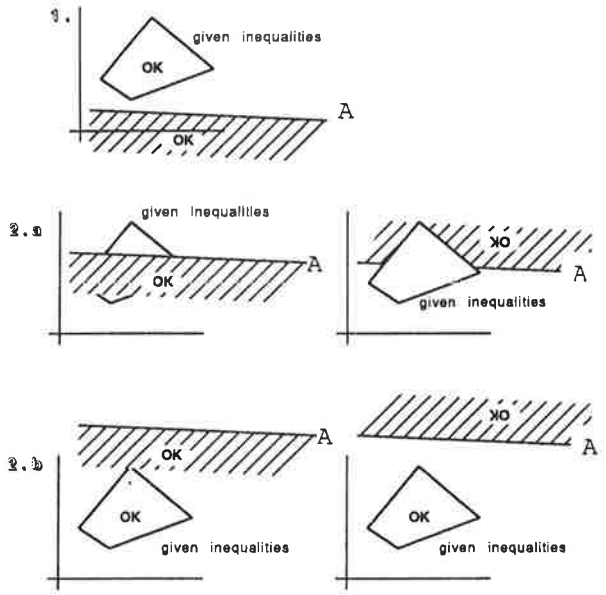

Figure 8: Deciding an inequality A using ILP

Extensions

A serious restriction of ILP is that it can only deal with linear (in)equalities. This means that expressions involving e.g. i j or mod(i,j) can not be dealt with (i and j are variables). Since this sort of constructs is allowed in most HDLs and is often used when describing for example systolic arrays, the algorithms will have to be adapted to work with *non-linear programming* techniques.

CONCLUSIONS

It has been shown that it is feasible to do program transformation of hardware descriptions using an existing HDL. A new technique has been presented therefore. Given the hardware description of a design and a transformation description, this program transformation technique calculates the hardware description corresponding with the transformed design.

The method is based on the simplex and cutting-plane algorithm for solving LP and ILP problems. Hence it is limited to hardware descriptions which involve only linear expressions though it could be extended by adding techniques to solve non-linear programming problems.

The technique has been illustrated on the basis of a simple example. It has also been applied for the design of a systolic FIR filter and a parameterized multiplier accumulator module used in the Cathedral-II silicon compiler.

REFERENCES

[1] H.De Man, "Cathedral-II: A Silicon Compiler for Digital Signal Processing", *IEEE Design & Test of Computers*, Dec 1986, Vol.3, N0.6, pp.73-85

[2] H.De Man, "Evolution of CAD tools towards third generation custom VLSI design" *Revue Phys. Appl. 22*, Vol.22, Jan 1988, pp.31-45

[3] P.N.Hilfinger, "A High Level Language and Silicon Compiler for Digital Signal Processing", *Proc CICC-85*, pp.213-216

[4] P.Six, "An Intelligent Module Generation Environment", *Proc 23rd DAC*, Las Vegas June 29 - July 2, 1986, pp.730-735

[5] 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

[6] 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. (North-Holland), 1985 pp.153-178

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

[8] S.M.German, Y.Wang, "Formal Verification of Parameterized Hardware Designs", *Proc ICCD-85*, October 1985, pp.549-552

[9] M.Gordon, "HOL: A Proof Generating System for Higher-Order Logic", *in VLSI Specification, Verification and Synthesis*, editors: G.Birtwistle, P.Subrahmanyam, Kluwer Academic Publishers, 1988, pp.73-128

[10] A.Cohn, "A Proof of Correctness of the Viper Microprocessor: The First Level", *in VLSI Specification, Verification and Synthesis*, editors: G.Birtwistle, P.Subrahmanyam, Kluwer Academic Publishers, 1988, pp.28-71

[11] J.J.Joyce, "Formal Verification and Implementation of a Microprocessor", *in VLSI Specification, Verification and Synthesis*, editors: G.Birtwistle, P.Subrahmanyam, Kluwer Academic Publishers, 1988, pp.129-157

[12] H.Eveking, "Verification, Synthesis and Correctnes-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

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

[14] M.Sheeran, "Describing and Reasoning about Circuits using Relations", *Proc. Leeds workshop on Theoretical Aspects of VLSI Design*, 1986

[15] D.Borrione, "An Approach to the Formal Verification of VHDL Descriptions", *Rapport de Recherche, Institut National Polytechnique de Grenoble* Grenoble (France), November 1987

[16] L.Claesen, "Guided Synthesis and Formal Verification Techniques for Parameterized Hardware Modules", *Proc. Compeuro 88*, pp.90-99

[17] D.Verkest, "Formal Techniques for Proving Correctness of Parameterized Hardware using Correctness Preserving Transformations", *International working Conference on "The Fusion of Hardware Design and Verification"*, Glasgow, July 3-6, 1988, pp.75-96

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

[19] C.H.Papadimitriou, K.Steiglitz, "Combinatorial Optimization: Algorithms and Complexity", *Prentice Hall*, 1982

proc. ISCAS-85

PROGRAM TRANSFORMATION OF HARDWARE DESCRIPTIONS BY MEANS OF ILP.

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

IMEC†Kapeldreef 75, B-3030 Leuven, Belgium, Phone +32/16/281216

ABSTRACT

In this paper a new technique is presented for application of correctness preserving transformations to designs, described in an existing hardware description language. A new program transformation technique is used, based on Integer Linear Programming (ILP), to calculate automatically the structure description of the transformed design given the original designs' description and a transformation description. This technique is used for refining designs towards efficient implementations. It has been applied to the design of a systolic FIR filter and of a parameterized mutliplier-accumulator module used in the Cathedral-II silicon compilation system.

INTRODUCTION

The CATHEDRAL-II silicon compiler [1] is organized according to the meet-in-the-middle principle [2]. An automatic synthesis is done from the high level specification language SILAGE [3] into a number of controllers and execution units which are predesigned as parameterized modules. Examples of parameters are: the wordlength of the inputs, the number of busses, ... The set of required module generators is designed by circuit specialists as a set of LISP procedures [4], automatically generating the circuit layout for the module when instanciated with the appropriate parameter values.

While the high level synthesis is assumed to be 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 module generators 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. This paper describes a new technique that can be used to guarantee the functional correctness of such parameterized hardware modules.

For formal verification of *parameterized* hardware modules general theorem provers can be used. Researchers like Hunt [5] and Gordon [6] resort to well known formalisms of predicate calculus and higher order logic to describe hardware. The Boyer-Moore theorem prover [7] has been succesfully applied by Hunt for the correctness proof of a microprocessor [5] and by German for the verification of a parameterized comparator [8]. The HOL theorem proving environment [9] has been used by Cohn [10] and by Joyce [11] to conduct proofs concerning microprocessors. A common drawback [11] of these general purpose theorem provers is that the construction of the proof is a tedious task. Also the proof has to be well understood in advance.

This drawback can be avoided by adopting a transformational design style: correctness preserving transformations [12] are used to refine designs towards efficient implementations. Sheeran developed μFP [13] and more recently Ruby [14] to support such a design style. μFP and Ruby are two formal languages specially developeResearchers for the purpose of formal verification. However, Borrione [15] showed that it is possible to use an existing Hardware Description Language (HDL) for the purpose of formal verification. The motivation for using existing HDLs stems from the fact that they are more familiar to hardware designers than the formalisms mentioned above. The parameterized hardware modules to be verified, are described in an existing HDL and the technique presented is used in a transformational design strategy [16]; the designer specifies the transformations he wants to apply to the design by means of transformation descriptions. Both the transformation description language and the HDL will be described in the following section. The technique is based on a geometrical interpretation of the structure and transformation description as will be explained later on. The implementation of the technique is based on the ILP algorithm, which will be discussed in the last section.

HARDWARE AND TRANSFORMATION DESCRIPTION

The HDL is called HILARICS [18]. HILARICS starts from the concept that the network *structure* should be described completely independent from the other aspects of a design. It is an applicative language in which designs can be described hierarchically and in a parameterized way. HILARICS has been used for the description of several hierarchical and parameterized VLSI designs. As a working example the structure description of the piece of hardware of figure 1 is given. The minimum value of the parameter p is three. The drawing in figure 1 is made for two specific values of the parameter (p=3 and p=7). The HILARICS description is given in figure 2.

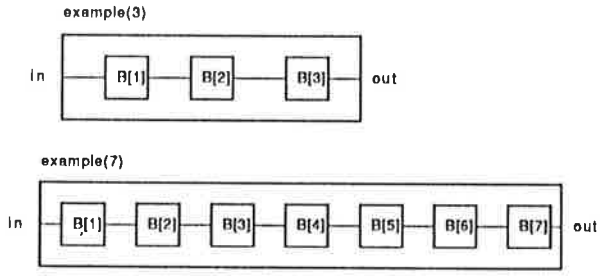

Figure 1: Working example: a simple row of building blocks B

*Professor at K.U.Leuven

†Interuniversity Micro Electronics Center

```
CELL example (p)

 TERMINALS in out

 COMPONENTS
  IF p≥3 THEN FOR x=1 TO p DO B[i] END
         ELSE NOCOMP
  END

 CONNECTIONS
  IF p≥3 THEN FOR x=1 TO p DO
                 IF x≤1 THEN in = B[x].i
                        ELSE IF x≤p-1 THEN B[x-1].o = B[x].i
                                      ELSE B[x-1].o = B[x].i
                                           B[x].o = out
                              END
                 END
                END
         ELSE NONET
  END
 END
```

Figure 2: The structure description of the working example

A *transformation* description indicates two things:

- which primitive equivalence transformation is used,
- and to which part(s) of the design this primitive transformation applies.

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, if and only if the original description fulfils certain conditions depending on the primitive transformation being used. In the DSP-like applications considered so far, the following classes of primitive equivalence transformations can be considered:

- primitive arithmetic transformations can be applied to arithmetic building blocks as e.g. full adders.
- primitive boolean transformations are used to replace boolean logic. The correctness of these transformations is checked by a tautology checker.
- primitive flowgraph transformations like delay management and retiming operations.

More details on the transformations and the transformational design method can be found in [17].

As an example of a transformation we will substitute some blocks B of figure 1 by different, but functionally equivalent, blocks NEWB. This transformation will only have to be executed if the parameter 'p' is larger than four. The transformation description to achieve this is given below and the corresponding transformed structure is given in figure 3 (the NOP operation indicates that no transformation has to be executed).

```
TRANSFORMATION example (p)

 IF p≥5 THEN IF x≥3 THEN IF x≤p-1 THEN SUBST(B[x]NEWB[x])
                                  ELSE NOP
                         END
                ELSE NOP
           END
      ELSE NOP
 END
```

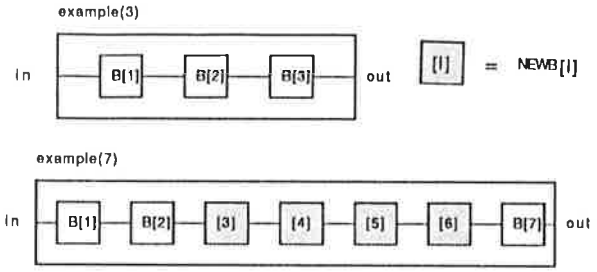

Figure 3: The working example after transformation

GEOMETRICAL INTERPRETATION

As a shorthand notation for the structure and transformation description we use a binary tree. The leafs of the tree correspond to the net definitions resp. primitive transformations. The vertices contain the tests of the IF THEN ELSE constructs. The left edge of a vertex corresponds to the THEN case of the IF THEN ELSE construct in that vertex and the right edge corresponds to the ELSE case.

The conditions found in the vertices can be seen as equalities (or inequalities) defining hyperplanes (or hyperspaces) in an N-dimensional space; N is the number of indices and parameters concerned in the structure description. By following a path from the root to a leaf L of such a tree, the vertices (conditions) encountered, define a polytope whose boundary planes are given by the conditions in the vertices. For all integer points within this polytope, the net definition of the leaf L is active, i.e. the nets defined by the definition in the leaf are present in the structure. In the case of the transformation description, the primitive transformation defined in the leaf L has to be executed for all integer points within this polytope.

Figures 4 and 5 show the tree representation and the geometrical interpretation of resp. the structure description and the transformation description of the example design. To calculate the

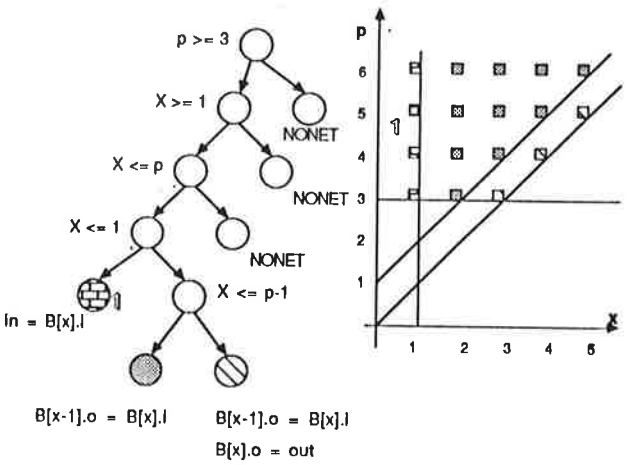

Figure 4: The structure tree and its geometrical interpretation

description of the transformed structure we will have to find out if a particular net definition of the structure tree has to be transformed by a particular transformation definition of the transformation tree. This can be done by looking for intersections of the polytopes in which the respective definitions are active. For all integer points in such an intersection both the net and the transformation definition are active, meaning that the particular net has to be transformed

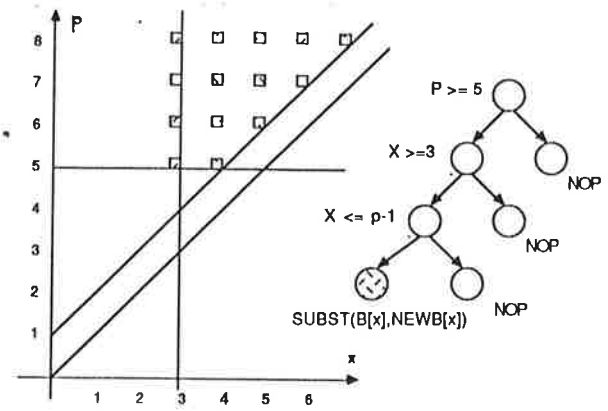

Figure 5: The geometrical interpretation of the transformation tree

by the particular transformation. These intersections can be determined by inserting the complete transformation tree in front of each leaf of the structure tree. The tree obtained in that way is called the *merged* tree. Each root-leaf path RL of the merged tree describes exactly one intersection (see figure 6). Three cases can

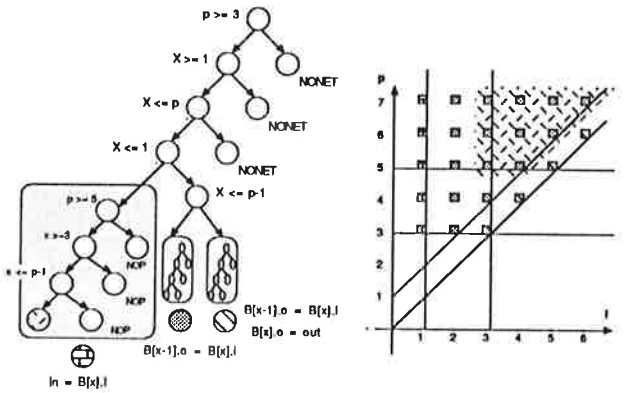

Figure 6: The merged tree and its geometrical interpretation

be distinguished:

1. RL defines a region in which both net definition and transformation definition are active: the intersection of the two polytopes. In this region the new net definition can be calculated from the old net definition and the transformation definition.
2. RL defines a region in which the net definition is active but the transformation definition is not: there is no intersection. The old net definition doesn't have to be transformed.
3. RL defines a region in which the net definition is not active.

The tree representing the transformed design can be derived directly from this merged tree, using the techniques described in the next section. This transformed tree is shown in figure 7 and it corresponds to the drawings of figure 4.

ILP TECHNIQUES

The problem is now reduced to:

- determine wether the two polytopes have an intersection.
- if they have an intersection determine the minimal set of (in)equalities describing that intersection.

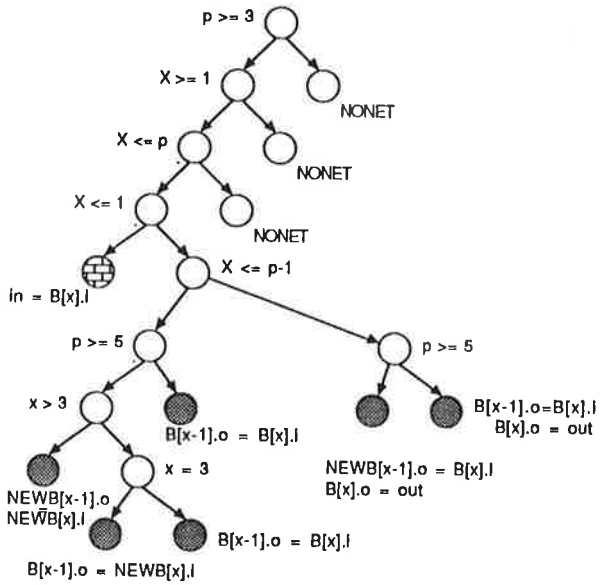

Figure 7: The structure tree of the transformed design

Intersection of polytopes

In the first stage we are given a number of (In)equalities that describe an intersection and we want to determine if that intersection is non-empty, i.e. if the (in)equalities contain a feasible solution. Since we are dealing with hardware descriptions the solution will be made up of integers (parameters and indices). This problem can be mapped on the ILP problem (pp.307 of [19]). An algorithm to solve this consists of three parts:

1. determine an initial feasible solution by solving a slightly different LP problem by means of the *simplex* algorithm.
2. solve the relaxed LP problem (i.e. identical to the ILP problem but without the constraint for an integer solution) by means of *simplex*.
3. iterate to an optimal and integer solution by means of the *cutting plane* algorithm.

For our problem we can leave out the second step: we only want to know whether the set of constraints allows for a feasible solution - meaning the two polytopes have an intersection - and we don't need an optimal solution (for that matter we don't have a goal function to optimize either).

After this first stage we know whether there is an intersection in the root-leaf path considered. If there is no intersection we check the following root-leaf path of the merged tree. If there is an intersection we proceed with the second stage to eliminate redundant (in)equalities to obtain a minimal set describing the intersection.

Elimination of redundant equations

We have a sequence of (in)equalities determining an intersection and we want to eliminate all superfluous ones. We suppose that there are no redundant (in)equalities among those of the structure tree and then we check one by one the (in)equalities of the transformation tree. A redundant equation is an (in)equality which is always false or always true given the previous (in)equalities. We use the ILP algorithm to decide about the redundancy of the last

(in)equality in a sequence of (in)equalities by means of the following technique (see figure 8):

- if the ILP finishes without a feasible solution then the last (in)equality is always false and thus can be deleted. However, since we first checked that there was indeed a solution, this situation will not occur.
- if there exists a feasible solution we restart the ILP, but with the last (in)equality inverted. If there is still a feasible solution then the last (in)equality is not redundant (figure 8.a). If there is no feasible solution then the last (in)equality is always true and can be deleted (figure 8.b).

By proceeding in this way we can eliminate all the redundant (in) equalities and prune the merged tree to obtain the tree representing the transformed structure (see figure 7).

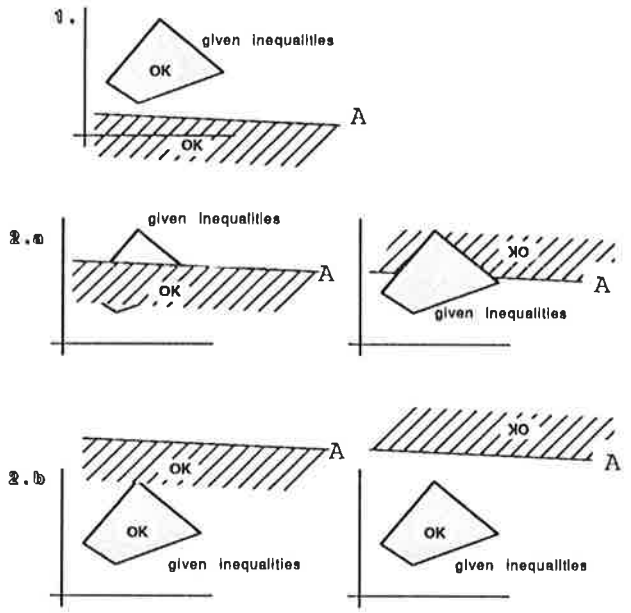

Figure 8: Deciding an inequality A using ILP

Extensions

A serious restriction of ILP is that it can only deal with linear (in)equalities. This means that expressions involving e.g. i j or mod(i,j) can not be dealt with (i and j are variables). Since this sort of constructs is allowed in most HDLs and is often used when describing for example systolic arrays, the algorithms will have to be adapted to work with *non-linear programming* techniques.

CONCLUSIONS

It has been shown that it is feasible to do program transformation of hardware descriptions using an existing HDL. A new technique has been presented therefore. Given the hardware description of a design and a transformation description, this program transformation technique calculates the hardware description corresponding with the transformed design.

The method is based on the simplex and cutting-plane algorithm for solving LP and ILP problems. Hence it is limited to hardware descriptions which involve only linear expressions though it could be extended by adding techniques to solve non-linear programming problems.

The technique has been illustrated on the basis of a simple example. It has also been applied for the design of a systolic FIR filter and a parameterized multiplier accumulator module used in the Cathedral-II silicon compiler.

REFERENCES

[1] H.De Man, "Cathedral-II: A Silicon Compiler for Digital Signal Processing", *IEEE Design & Test of Computers*, Dec 1986, Vol.3, N0.6, pp.73-85

[2] H.De Man, "Evolution of CAD tools towards third generation custom VLSI design" *Revue Phys. Appl. 22*, Vol.22, Jan 1988, pp.31-45

[3] P.N.Hilfinger, "A High Level Language and Silicon Compiler for Digital Signal Processing", *Proc CICC-85*, pp.213-216

[4] P.Six, "An Intelligent Module Generation Environment", *Proc 23rd DAC*, Las Vegas June 29 - July 2, 1986, pp.730-735

[5] 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

[6] 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. (North-Holland), 1985 pp.153-178

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

[8] S.M.German, Y.Wang, "Formal Verification of Parameterized Hardware Designs", *Proc ICCD-85*, October 1985, pp.549-552

[9] M.Gordon, "HOL: A Proof Generating System for Higher-Order Logic", *in VLSI Specification, Verification and Synthesis*, editors: G.Birtwistle, P.Subrahmanyam, Kluwer Academic Publishers, 1988, pp.73-128

[10] A.Cohn, "A Proof of Correctness of the Viper Microprocessor: The First Level", *in VLSI Specification, Verification and Synthesis*, editors: G.Birtwistle, P.Subrahmanyam, Kluwer Academic Publishers, 1988, pp.28-71

[11] J.J.Joyce, "Formal Verification and Implementation of a Microprocessor", *in VLSI Specification, Verification and Synthesis*, editors: G.Birtwistle, P.Subrahmanyam, Kluwer Academic Publishers, 1988, pp.129-157

[12] H.Eveking, "Verification, Synthesis and Correctnes-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

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

[14] M.Sheeran, "Describing and Reasoning about Circuits using Relations", *Proc. Leeds workshop on Theoretical Aspects of VLSI Design*, 1986

[15] D.Borrione, "An Approach to the Formal Verification of VHDL Descriptions", *Rapport de Recherche, Institut National Polytechnique de Grenoble* Grenoble (France), November 1987

[16] L.Claesen, "Guided Synthesis and Formal Verification Techniques for Parameterized Hardware Modules",*Proc. Compeuro 88*, pp.90-99

[17] D.Verkest, "Formal Techniques for Proving Correctness of Parameterized Hardware using Correctness Preserving Transformations", *International working Conference on "The Fusion of Hardware Design and Verification"*, Glasgow, July 3-6, 1988, pp.75-96

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

[19] C.H.Papadimitriou, K.Steiglitz, "Combinatorial Optimization: Algorithms and Complexity", *Prentice Hall*, 1982