Correctness Preserving Transformations on the Hough algorithm. *

J.G. Samsom, L.J.M. Claesen, H.J. De Man

IMEC vzw. Kapeldreef 75, B-3001 Leuven, Belgium
Phone +32-16-281525, Fax +32-16-281515, email: samsom@imec.be

Abstract

A formal method for the optimization of a specification in a guaranteed correct way is presented. As an example, the transformation from a behavioral specification of the Hough transform in an image space, towards an optimized specification will be presented. The transformations are meant to be used in an interactive environment. The main result presented is that, by using a limited set of transformations, an optimized description in terms of a silicon compiler can be derived in a guaranteed correct way.

Keywords: *transformational design, guided synthesis, formal verification, correctness preserving transformations, VLSI design.*

1 Introduction

Guaranteed correct VLSI design is an increasing area of research. Thanks to the improvements in VLSI processing technology, circuits become more and more complex and error prone. Post-hoc verification of design steps is still a time-consuming job and often hints are needed to help a theorem prover in his proof of correctness. The hints that are given are often conform to design steps that are taken during design. Because of this, more natural constructive approaches are proposed.

A possible solution can be achieved with fully automatic synthesis systems, yielding correctness by construction. However, automatic synthesis will not always be possible due to high performance requirements and will always be restricted to a limited set of applications. The unbeatable effectiveness of human intelligence in making design transformations is often necessary. Especially at a high description level, complexity can be that high that interaction with an (experienced) designer is needed. The transformation of an algorithm into an optimised version of the algorithm, the partitioning of an algorithm, or the transformation of an algorithm into a set of general recursion equations implementable in a regular array are complex design steps that are often done manually (before entering a silicon compiler).

A guided correctness preserving design system capable of doing all the transformations a user needs, would guarantee the correctness while giving the opportunity to look in a fast and flexible way at several alternatives. In this way, optimisation in terms of memory usage, I/O bandwidth, number of operators, parallelism and timing would be possible in an early design stage (the result will be dependent on the experience of the designer).

In this paper, an example of the Hough transform will be worked out. Starting from a naive description of the behavior of the algorithm (specification) an optimised description in terms of a silicon compiler will be derived. Transformations will all be on a behavioral level on a hardware description language (SILAGE). Other approaches [5] make use of graphical interfaces especially suited for applications with a lot of regularity.

Section 2 defines a subset of Silage. On this subset, transformation laws are defined in section 3. Section 4 describes the algorithm of the studied example (the Hough transform). Then in section 5 some transformation steps applied on this example will be shown. Finally, section 6 draws some conclusions based on the results obtained from the example.

2 A subset of Silage

Silage is an applicative (functional) behavioral specification language especially suited for DSP applications [4]. During the first stage of the specification of a DSP algorithm into a Silage description, the user should not be bothered by efficiency crite-

*Research funded by ESPRIT project SPRITE(3216)

ria, but should have an as simple and clear as possible description of the specification. The only concern of the designer at this stage should be the correctness of the specification. In the next design phase, the designer will optimise the specification using a sequence of correctness preserving transformations. In this way the specification is optimised in terms of the architecture of the automatic silicon compiler used in the next stage. (highly multiplexed, lowly multiplexed, regular array) [1].

In order to ensure the correctness of the transformations, an unambiguous meaning of all constructs in Silage should be given. Work is going on in defining the semantics of Silage in terms of a proof assistant HOL [2][3]. With the semantics of the Silage constructs correctness of transformations can be proven. In a first step, a subset of Silage is chosen. The main difference between the subset and Silage is the explicit appearance of dependencies in the subset, while Silage leaves dependencies implicit. The explicit appearance of dependencies in the subset gives the user a better insight in the efficiency of the description and transformation laws are more easily derived. The dependencies are made explicit by introducing two combinators "seq" and "par". "Seq" gives a strict (sequential) order to the following declarations. A "par" combinator is used when the declarations do not depend on each other and can be executed in any order.

e.g.

```
seq (
    x = 3;
    y = x + 5);

par (
    x = 3;
    y = 4 + 5);
```

Loops are just a way of grouping several statements. A loop is a shorthand notation for the expansion of the loop.

e.g.

```
seq (i : 0..3)::
    x[i] = x[i−1] + S[i];
```

is equivalent to:

```
seq (
    x[0] = x[−1] + S[0];
    x[1] = x[0] + S[1];
    x[2] = x[1] + S[2];
    x[3] = x[2] + S[3]);
```

The abstract syntax of the subset is given in fig 1.

d	::=		(declaration)
	|	(s = e)	(equation)
	|	(seq d1 d2)	(sequential composition)
	|	(par d1 d2)	(parallel composition)
e	::=		(expression)
	|	s	(selector)
	|	s@n	(delayed selector)
	|	e1 * e2	(* arithmetic operator)
s	::=		(selector)
	|	x	(variable)
	|	s[i]	(subscript)

Figure 1: abstract syntax

3 Transformations

Based on the syntax given in the previous section, transformation laws can be defined. Future work will be the integration of these transformations in the proof assistant HOL to prove the correctness of the transformations and to make them available in a library with correctness preserving transformations. In this paper, some examples of simple transformations are given whose correctness constraints are easily found. Future work will be the integration of these transformations in an interactive tool capable of executing the transformations and the connection to HOL. First some definitions will be given. Then a list with a set of transformations will be presented.

- Transformations are described by inference rules. A term above the line can be rewritten in the term below the line when the constraints (between square brackets) are fulfilled.

- For each declaration d, left d is defined as the set of all selectors (fig. 1) occurring at the left-side in all declarations d, and right d is defined as the set of all selectors occurring at the right-side in d.

  e.g.

  ```
  s = x + y;
  left d = s
  right d = x,y
  ```

- d[i] is a declaration d where i is an index used within d.

(1) the order of declarations in a par construct is unimportant. A par construct is a commutative constructor.

$$\frac{par\ d1\ d2}{par\ d2\ d1}$$

(2) a par construct is an associative constructor.

$$\frac{par\ (par\ d1\ d2)\ d3}{par\ d1\ (par\ d2\ d3)}$$

(3) if no selector that is consumed in d2 is produced in d1, then the order of d1 and d2 is independent and the seq construct can be replaced by a par construct.

$$\frac{seq\ d}{par\ d}\ [left(d) \cap right(d) = \{\}]$$

(4) a seq construct is an associative constructor.

$$\frac{seq\ (seq\ d1\ d2)\ d3}{seq\ d1\ (seq\ d2\ d3)}$$

(5) loop folding
With loop folding, pipelining can be introduced in the execution of loop bodies. One declaration is shifted to the previous iteration while the other iteration stays in the same iteration.

$$\frac{\begin{array}{l} seq\ (i : x..y) :: \\ \quad seq\ d1[i]\ d2[i] \end{array}}{\begin{array}{l} seq\ (\ d1[x]; \\ \quad seq\ (i : x+1..y) :: \\ \quad\quad seq\ d2[i-1]\ d1[i]; \\ \quad d2[y]) \end{array}}$$

(6) associativity 1
The algebraic property of associativity gives the possibility to reverse the dependencies. f and g are functions dependent on the iterator i and resulting in an expression. Initialisation of s has changed but is not shown. Intermediate values differ from original values of s and should not be necessary at other places in the program. The final result is s[0] instead of s[N].

$$\frac{\begin{array}{l} seq\ (i : 0..N) :: \\ \quad s[i] = f(i)*g(i) \end{array}}{\begin{array}{l} seq\ (i : 0..N) :: \\ \quad s[N-i] = f(N-i)*g(N-i) \end{array}}\ [*\ is\ assoc]$$

(7) associativity 2
This is a transformation like transformation 6. The difference is in the two (instead of one) nested loops. f and g are functions dependent of both iterators i and j and result in an expression. The same remarks can be made concerning initialisation, use of s at other places and the final result.

$$\frac{\begin{array}{l} seq\ (i : 0..N) :: \\ \quad seq\ (j : 0..M) :: \\ \quad\quad s[i][j] = f(i,j)*g(i,j); \end{array}}{\begin{array}{l} seq\ (i : 0..N) :: \\ \quad seq\ (j : 0..M) :: \\ \quad\quad s[N\text{-}i][M\text{-}j] = f(N\text{-}i, M\text{-}j)*g(N\text{-}i, M\text{-}j); \end{array}}$$

$$[*\ is\ assoc]$$

(8) A par loop can be executed in any order. Because of this the description of the loop can be reversed.

$$\frac{par\ (i : 0..N) :: d[i]}{par\ (i : 0..N) :: d[N-i]}$$

(9) loop merging
Two loops with the same iterator bounds can be taken together in one loop when the production in the second loop does not depend on productions in the first loop that are not yet produced.

$$\frac{\begin{array}{l} seq\ (\ seq\ (i : x..y) :: d1[i]; \\ \quad\quad seq\ (j : x..y) :: d2[j];) \end{array}}{\begin{array}{l} seq\ (i : x..y) :: \\ \quad seq\ d1[i]\ d2[i] \end{array}}$$

$$[\forall j\ (j > i)\ right\ d2[i] \cap left\ d1[j] = \{\}\ ]$$

(10) iterator splitting
An iterator can be split into a small loop that is within another loop.

$$\frac{seq\ (i : x..y) :: d[i]}{\begin{array}{l} seq\ (j : 0..n-1) :: \\ \quad seq\ (i : x+b*j..x+b-1+b*j) :: d[i]; \end{array}}$$

$$[1 + y - x = b * n]$$

(11) loop interchanging
Dependent on the dependencies in the body of a loop, the order of two nested loops can be interchanged.

$$\frac{\begin{array}{l} seq\ (i : 0..N) :: \\ \quad seq\ (j : 0..M) :: d[i,j]; \end{array}}{\begin{array}{l} seq\ (j : 0..M) :: \\ \quad seq\ (i : 0..N) :: d[i,j]; \end{array}}$$

$$[\forall k, j\ (k < i) \wedge (l > j) :\ (rightd[\mathrm{i,j}] \cap left\ d[\mathrm{k,l}] = \{\}\ )]$$

Some transformation rules can be derived from other rules. An example is dependency reversal.

(12) dependency reversal
In this transformation the order of dataflow in a loop of associative operations is reversed. This rule can be derived by executing laws 8,3,7,11,8 and 3. Conditions for this transformation are derived from the transformation laws defining this transformation.

$$\frac{\begin{array}{l} seq\,(i : 0..N) :: \\ \quad seq\,(j : 0..M) :: \\ \quad\quad s[i][j] = f(i\text{-}a, j\text{-}b)*g(i,j); \end{array}}{\begin{array}{c} law3 \\ law8 \\ law11 \\ law7 \\ law3 \\ law8 \end{array}}$$

$$\begin{array}{l} seq\,(j : 0..M) :: \\ \quad seq\,(i : 0..N) :: \\ \quad\quad s[i][j] = f(i+a, j+b)*g(i,j); \end{array}$$

4 The Hough transform

The method is demonstrated by the optimisation of the Hough algorithm in terms of a regular array compiler (Cathedral 4 [6]). Although not all steps are shown, they are all described in transformation laws. The Hough transform is a projection based technique for curve detection in images [7]. An important use of the Hough transform is in applications for straight line detection. In the Hough transform straight lines are identified by determining the total number of image feature points on discrete projection lines covering the image. The following equation describes this process:

$$P_\theta(\rho) = \sum_{(x_s, y_s) \in L_d(\theta,\rho)} f(x_s, y_s) \qquad (1)$$

The function f returns a 1 if the pixel at coordinate (x_s, y_s) has a value which indicates an image feature point and a 0 otherwise. The line $L_d(\theta, \rho)$ represents an approximation of a straight line. The resulting projection value $P_\theta(\rho)$ equals the number of image feature points on line $L_d(\theta, \rho)$. $P_\theta(\rho)$ gives a measure of the probability that a line $L_d(\theta, \rho)$ is present on the image. This information can be used to identify objects.

4.1 Specification

The initial specification of the Hough transform is made according to the regularity exhibited in the image space (fig 2). In the first two loops initialisation is done, while the last two loops give the final result. In the third loop the calculations are done.
Specification:

```
par
    par (i : 0..N−1)::
        x[i+1][−1] = 0;

    par (p : 1..N−1)::
        x[N][p−1] = 0;

    seq (p : 0..N−1)::
        par (i : 0..N−1)::
            x[i][p] = x[i+1][p−1] + I[i][p];

    par (p : 0..N−1)::
        line[p] = x[0][p];

    par (i : 1..N−1)::
        line[i+N−1] = x[i][N−1];
```

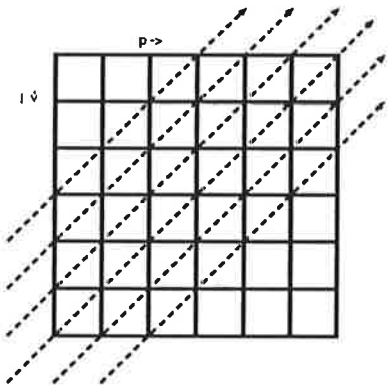

Figure 2: lines in i,p image space

Like many algorithms and implementation proposals for the Hough transform this specification depends on the regularity the transformation exhibits in the image space. As a consequence, it requires as many as N to N processing elements (PE's) (for an image of NxN pixels) detecting lines of 45 degrees and an exorbitant I/O bandwidth. A possible optimisation, is the mapping of the p axis on the time axis, resulting in N PE's and still a high I/O bandwidth.

5 Transformation Steps

In this section some transformation steps that will lead to an optimized description are described. Optimisation is done in terms of memory requirements, I/O requirements and complexity of the processing elements (PE's). For readability and simplicity reasons neither the whole description nor all steps will be

shown fully.

Purpose is to obtain a description in which input pixels are consumed in scan bands, resulting in a lower I/O bandwidth, less memory requirements and less PE's.

5.1 dependency reversal

As a first step, the order of the dependencies is reversed. Transformation law 12 is being used. The associativity of the summator in the loop body is used to reverse the direction in which the image is traversed. Result is the description shown below (initialisations and finalisation are skipped). The order in which the loops are traversed is interchanged, and the direction of the dependencies is reversed. Fig 3 shows the resulting data flow.

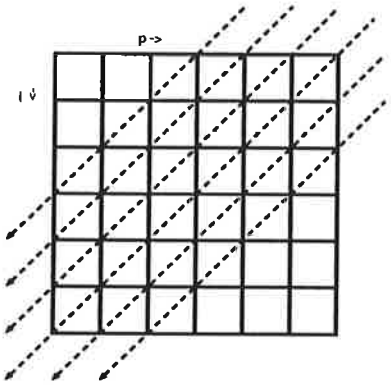

Figure 3: dataflow in i,p image space

```
seq (i : 0..N−1)::
    par (p : 0..N−1)::
        x[i][p] = x[i−1][p+1] + I[i][p];
```

5.2 iterator splitting

In this step the scan bands are introduced. The i axis is split into (N/n) bands "bc" of n pixels high according to transformation law 10. In the resulting description the image is scanned in bands that are traversed in n rows (fig 4).

```
seq (bc : 0..(N/n)−1)::
    seq (i : 0+bc*n..n−1+bc*n)::
        par (p : 0..N−1)::
            x[i][p] = x[i−1][p+1] + I[i][p];
```

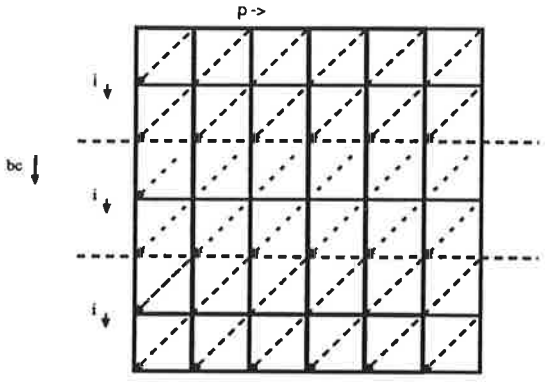

Figure 4: dataflow in i,p image space

5.3 dependency reversal 2

To diminish the number of necessary process elements (PE's), the row oriented traversal (N PE's) will be transformed in a column oriented traversal of the bands. Again a dependency reversal transformation (12) is applied, but this time without affecting the ordering of the scanbands (bc), only the ordering within the bands is reversed. This results in a traversal of the image in bands that are traversed columnwise (fig 5).

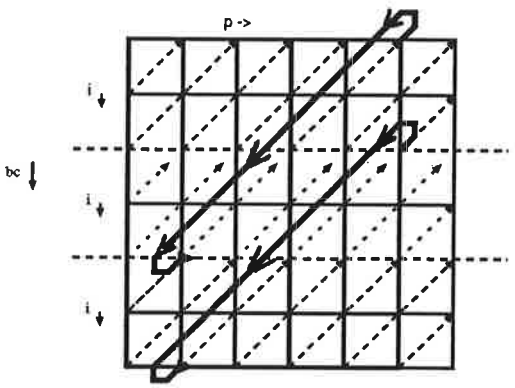

Figure 5: dataflow in i,p image space

```
seq (bc : 0..(N/n)−1)::
    seq (p : 0..N−1)::
        par (i : 0..n−1)::
            x[bc][i+bc*n][p] =
                x[bc][i+bc*n+1][p−1] + I[i+bc*n][p];
```

5.4 final result

Other transformations like loop-folding (5) and loop-merging (9) have been applied in other steps. Finally a description could be derived consisting of an array of process elements "PE", a register "inter" storing intermediate values of the computations, and an output array "out" storing the results of scanning the image. Instead of N combined process/memory elements now only n process elements are necessary. I/O requirements did improve by the same factor by introducing the scan bands.

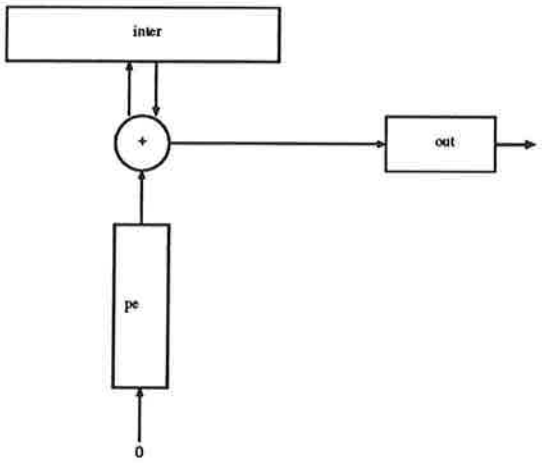

Figure 6: resulting implementation

6 Conclusions

In this paper a gradual improvement of an algorithm is shown. The improvements are done in small transformation steps, described in transformation laws that are based on the abstract syntax of the chosen subset of Silage. Intermediate results very often result in large descriptions most of the time caused by initialisation or finalisation of new variables or by unfolding of loops. In this report these parts of the description are skipped for readability reasons. An interactive system should be able to make the same abstraction to prevent the designer for getting to much unnecessary details. Using an interactive tool to do these transformations could prevent a lot of work (and faults) while still being able to use directly the ideas and experience of the designer. The example of the optimisation of the Hough transform shows the usability of an interactive tool giving the designer the possibility to do his 'own' transformations and make a trade off in I/O, memory requirements and datapaths at an early stage in a consistent way. Future work will be in the description of the semantics of Silage, the description of more transformations that will be imbedded in an interactive design system.

References

[1] H.De Man, F.Catthoor, G.Goossens, J.Van Meerbergen, J.Rabaey, J.Huisken, "Architecture-driven synthesis techniques for mapping digital signal processing algorithms into silicon", *special issue on comp. -aided design of Proc. of the IEEE, Vol.78, No.2, pp.319-335, Feb. 1990*

[2] A. Gordon, "The definition of Tiny–Silage", *Cheops Period progress Report PPR-2, 1991*

[3] M. Gordon, "HOL: A Proof Generating System for Higher Order Logic", *VLSI Specifiaction, Verification and Synthesis, Ed. G.Birtwistle and P.A.Subrahmanyam (Academic Pres, Boston, 1988), pp 73-127*

[4] P.N.Hilfinger, J.Rabaey, D.Genin, C.Scheers, H.De Man, "DSP specification using the Silage language", *Proc. Int. Conf. on Acoustics, Speech and Signal Processing, Albuquerque, NM, April 1990*

[5] A.A.J.de Lange, A.J. van der Hoeven, E.F. Deprettere, P.M. Dewilde, "HIFI: An Object Oriented System for the Structural Synthesis of Signal Processing Algorihms and the VLSI Compilation of Signal Flow Graphs", *IMEC-IFIP International Workshop on Applied Formal Methods for Correct VLSI Design, Nov 1989, pp 462-481*

[6] M.F.X.B. van Swaaij, J. Rosseel, F.V.M. Catthoor, H.J. De Man, "Synthesis of ASIC Regular Arrays for real-time image processing systems", *to be published in Journal of VLSI Signal Processing*

[7] M.F.X.B. van Swaaij, F.V.M. Catthoor, H.J. De Man, "Deriving ASIC architectures for the Hough transform", *Parallel Computing 16 (1990) 113–121 North Holland*