System Level Verification of Video and Image Processing Specifications [*]

H.Samsom, F.Franssen, F.Catthoor, H.De Man[†]
IMEC, Kapeldreef 75, B-3001 Leuven, Belgium

Abstract

A formal verification method is presented to verify the loop ordering of a high level transformed description against its original specification. The verification is done in an automatic way and its complexity is independent on the sizes of the loops bounds. Any practical structure of loop nests can be handled. The method is especially suited for applications in the area of speech, image and video processing, front-end telecom and numerical computing systems which exhibit many loops and complex multi-dimensional signals. The efficiency of the approach is demonstrated on several realistic examples.

1 Introduction

Practice shows that more than half of the total design effort can be taken up by verification or simulation at present [1]. Application studies in the area of speech, image and video processing, front-end telecom and numerical computing systems indicate that many algorithms operate on multi dimensional signals and exhibit a large amount of related control flow, especially expressed in terms of loops. Memory is a severe bottleneck in these applications. Recent research in optimizing the background memory and global communication shows that loop and index manipulations are crucial [2, 3]. Also in the compiler community, manual or automatic loop manipulation plays an important role. There, loop manipulation is usually performed in order to improve parallelism or communication cost [4, 5].

Nowadays, designers use simulation to proof the equivalence of two system level descriptions. In case of many Multi-Dimensional(M-D) signals and complex nested loops, verification by means of pseudo-exhaustive simulation on a workstation or (parallel) DSP board can be impossible due to the large amount of memory necessary [6]. In addition, verification by simulation is impossible when either the loops in initial specification or the intermediate description can not be executed in a procedural way. Also synthesis is then infeasible with the current methods.

[*]Research funded by ESPRIT Project SPRITE (2260)
[†]Professor at K.U.Leuven

The formal techniques used in theorem provers are capable of verification at a high abstraction level but need too much designer knowledge/interaction to come to an automated proof. SFG Tracing, can automatically verify the equivalence between a high level specification and an implementation [7]. However, they start from the assumption that ordering of the operations in the specification is not changed during synthesis. Unrolling of the loops is a possibility in this case, but becomes prohibitive for most practical applications. In the array synthesis community, loop reorderings play a very important role [8, 9]. A method that can handle the verification of simple stepwise loop reorderings is given in [10] but it is restricted by the model used. Several researchers [11, 12, 13] use a methodology which guarantees correctness by construction. They use a formal model to prove their transformations. They restrict the designer to a limited set of transformations which cover most often only small steps in the synthesis process. Moreover, the sequence of transformation steps has to be known.

In this paper we present a system verification method which plays a crucial role in the formal verification of complex loop transformations like loop folding, loop distribution, typically applied in the design trajectory of data dominated signal and data processing applications. A novel formal model for a general applicative language with sequential constructs is defined (section 3). It is the basis for the verification approach and it is not intended as a new user specification model. In case of a fully procedural specification, translation to an applicative description must be done first [4, 14]. With the presented technique, any pair of arbitrarily ordered descriptions can be proven to be behaviorally equivalent if they exhibit the same I/O behavior in terms of signal definitions, as long as the primitive operations performed on the signals are not modified.

In section 2, our novel verification method is described. The formal model, which forms the core of the formal verification is defined in section 3. In section 4, the method is briefly demonstrated on some examples. Finally, section 5 gives the conclusions.

2 The Formal Verification Procedure

The formal model used during the verification, is related to the models used in the array synthesis domain [8, 9]. There equations are valid under a set of constraints that are often represented in a geometrical way. In our model each statement in the description is defined by a predicate. The predicate consists out of a relation (R) which holds for a set of constraints on its input arguments, defined by the precondition (P) ($P.R$). In addition to the models used in the array synthesis domain the presented model can also encapsulate indefinite iterations that are executed sequentially, a data-dependent number of iterations. This is described by a complex predicate whose relation (R) includes a set of predicates as well:

$P.\,DO\;\;(P1.\,R1\;\wedge\;P2.\,R2\;............Pn.\,Rn)\;UNTIL\;R$

The complete description is modeled by the *AND* of all the predicates within the description [15]. The precondition (P), defines the manifest constraints on the indices of the loop bounds surrounding the statement and the index domains for the signals in the statement. The relation (R) is a unique representation of the non-manifest parts in the description. The relation is made independent from the index names from the original description. The modeling in predicates is illustrated by a small example specified in the mixed procedural/applicative language DFL [16]. In the example (Fig. 1), the original specification is shown on the left, the optimized description derived after loop folding is shown on the right. The modeling in terms of predicates is shown in Figures 2 and 3.

```
                                a[0] = in[0];
(i:0..N-1)::                    (i:1..N-1)::
begin                           begin
    a[i] = in[i];       =>          a[i] = in[i];
    out[i] = a[i];                  out[i-1] = a[i-1];
end;                            end;
                                out[N-1] = a[N-1];
```

Figure 1: Original and Optimized Specification

$(\forall i_1, i_2\; \exists i\; (0 \le i \le N-1) \wedge (i_1 = i) \wedge (i_2 = i)).a(i_1) \equiv in(i_2)$

$\wedge$

$(\forall i_1, i_2\; \exists i\; (0 \le i \le N-1) \wedge (i_1 = i) \wedge (i_2 = i)).out(i_1) \equiv a(i_2)$

Figure 2: Modeling of the Original description

$(\forall i_1, i_2\; (i_1 = 0) \wedge (i_2 = 0)).\; a(i_1) \equiv in(i_2)$

$\wedge$

$(\forall i_1, i_2\; \exists i\; (1 \le i \le N-1) \wedge (i_1 = i) \wedge (i_2 = i)).\; a(i_1) \equiv in(i_2)$

$\wedge$

$(\forall i_1, i_2\; \exists i\; (1 \le i \le N-1) \wedge (i_1 = i-1) \wedge (i_2 = i-1)).\; out(i_1) \equiv a(i_2)$

$\wedge$

$(\forall i_1, i_2\; (i_1 = N-1) \wedge (i_2 = N-1)).\; out(i_1) \equiv a(i_2)$

Figure 3: Modeling of the Optimized description

In order to prove their behavioral equivalence, the sets of predicates of two descriptions have to be proven equal. In case of simple predicates, this is done by first identifying the relations (R) that are the same in the original description. The modeling of the relations is done in a unique way. Under the obvious restriction that during design, input/output signal names may not change (index expressions and index names are allowed to change), finding the equal relations and equal indefinite loops is reduced to a direct pattern matching on signal names and operations. In case of equal relations, the two predicates are joined into one new predicate by merging the constraints of the two relations, potentially after some re-substitution steps of intermediate signals.

$P1.\ R\;\;\wedge\;\;P2.\ R\;\;\;\Rightarrow\;\;\;P1 \vee P2.\ R$

In the optimized description of Fig. 3, two pairs of relations are equal and their constraints have to be joined such that finally two predicates will result.

In case of indefinite loop constructs (like WHILE loops), the body of the *indefinite* loop and the rest of the description are separated. The domains of the equal relations in the indefinite body are merged, and the domains of the equal relations outside the *indefinite* body are merged.

Then, for each relation R in the specification, its definition in the implementation is found. In case of indefinite loop constructs, again a separation between the rest of the description and the body of the indefinite loop is made. Given the constraints from the surrounding loops, the bodies of equal indefinite loops in both descriptions must contain the same set of predicates. Otherwise the sequential restrictions imposed by the indefinite loop are violated during loop transformations. Again, equal relations can be easily found by syntactical pattern matching on the signal names and operations. In the example of Fig.1-3, it can be seen that the relations in the two descriptions can be matched by a simple syntactical pattern matching. To prove that the relations are defined under the same preconditions, it has to be proven that any two related domains must define the same index space. This can be done by taking the difference of the two domains. Although the pattern matching step has a quadratic dependency on the size of the code, the most difficult and also most time consuming job is taking the difference of the domains. The complexity of this step is linear with the size of the specification. During the domain operations, extensive use is made of the OMEGA Test [17] toolbox.

3 Formal Model Definition

In this section, a formal definition of the meaning of an applicative language is given. The formalization will be defined using: the syntactic domains which describe the elements in the language, the semantic domains which define the different elements in the formal model, and the semantic functions which define the denotation of the syntactic domains in terms of the semantic domains. The result will be a formal definition in terms of the predicates used in the formal verification method (section 2). Note that we do not include the detailed (RT level) timing behavior yet at this abstraction level.

3.1 Syntactic Domains

Table 1 defines the syntactic domains, in particular function(f), statements(s), expressions(e), and manifest expressions(n), that can be defined in the formal model. The syntactic domains of the language given in Table 1 are based on the syntactic domains of the DFL [16] language. It includes all elements necessary to map in combination with data dependency analysis, most applicative and procedural data flow languages like behavioral VHDL, DFL and SIGNAL [18], into the formal model in an analogue way to the modeling for this abstract syntax.

f	::=	(id s)	(function)
s	::=	(e1 = e2)	(definition)
	|	((id: n1..n2):: s)	(iteration)
	|	(s1 ; s2)	(composition)
	|	(if (n $\geq$ 0) $\rightarrow$ e1 = e2)	(manifest if)
	|	(if (e1) $\rightarrow$ e2 = e3)	(non manifest if)
	|	(do s until e1)	(indefinite iteration)
e	::=	c	(constant)
	|	x	(variable)
	|	e@n	(delayed signal)
	|	e@@n	(initialize delayed signal)
	|	e[n]	(array)
	|	e1 $\otimes$ e2	($\otimes$ arithmetic operator)
n	::=	c_m	(manifest constant)
	|	m	(manifest value)
	|	n1 $\oplus$ n2	($\oplus$ arith. manifest op.)

Table 1: Abstract Syntax

The basic data elements in the language are signals. Each signal is an infinite stream (ordered sequence) of values indexed by discrete sequentially ordered values. In each *frame*, one sample of the stream of values defining a signal is computed. All input samples arrive at the same time at a certain *rate*: the *frame rate*. Given the input samples, the definitions in the program specify the value of the signals in the description. The program only specifies what is to be computed and does not fix an ordering (except for the indefinite iteration which has a fixed sequential ordering).

An indefinite iteration repeats part of a program in a sequential way, until a certain condition is fulfilled. Although the iterations of the indefinite loop are executed sequentially, all signals for one execution of the indefinite loop are defined at the same time (in any order) and are called a *sub-frame*. During loop manipulations, the indefinite iterator defines a fixed sequential ordering and should therefore remain unchanged. Signals defined in the main *frame* (outside the indefinite loop), have an equal value for all *sub-frames*. The value of signals defined in a *sub-frame* (inside an indefinite loop), is equal to the value of the signal in the last *sub-frame* for which such a value has been defined.

A delay construct (@) is used to refer to samples in previous frames. A @ within an indefinite loop refers to the previous *sub-frame*. Within an indefinite iteration, no reference is possible to previous samples in the main *frame*. The initialization construct (@@) initializes the signals within a (sub)*frame*. Loop bound expressions, and index expressions are restricted to affine expressions on the loop indices.

3.2 Semantic Domains

In the previous section, we defined the different syntactic domains of the language whose formal denotation will be given. This section gives the Semantic Domains used for the formal definition. The definition of the semantic domains consists of the following domain equations (only the most important ones are explained).

$$\begin{aligned}
\mathbf{PredSet} &= \mathbf{Predicate} + \\
&\quad (\mathbf{Predicate} \times \mathbf{PredSet} \rightarrow \mathbf{Bool}) && (1)\\
\mathbf{Predicate} &= (\mathbf{Precond} \times \mathbf{Relation} \rightarrow \mathbf{Bool}) + \\
&\quad (\mathbf{Precond} \times \mathbf{Indef} \rightarrow \mathbf{Bool}) && (2)\\
\mathbf{Precond} &= \mathbf{Bool} + (\mathbf{Matrix} \times \mathbf{Vector} \rightarrow \mathbf{Expr}) && (3)\\
\mathbf{Relation} &= (\mathbf{Expr} \times \mathbf{Expr} \rightarrow \mathbf{Bool}) && (4)\\
\mathbf{Indef} &= (\mathbf{PredSet} \times \mathbf{Relation} \rightarrow \mathbf{Bool}) && (5)\\
\mathbf{Expr} &= \mathbf{Value} + (\mathbf{Value} \times \mathbf{Expr} \rightarrow \mathbf{Value}) && (6)\\
\mathbf{Value} &= \mathbf{Num} + \mathbf{Bool} + \mathbf{Ide} + \mathbf{Sym} && (7)\\
\mathbf{Vector[n]} &= \mathbf{Value}_1 \times ...\mathbf{Value}_n && (8)\\
\mathbf{Matrix[n,m]} &= \mathbf{Value}_{11} \times ...\mathbf{Value}_{nm} && (9)
\end{aligned}$$

A **Predicate** is a function from tuples (p,r) to **Bool**. The **Precond** p, models the input arguments for the function r (ϵ **Relation** or ϵ **Indef**). A **Precond** is constructed from affine constraints on integer variables with the logical operators $\neg$, $\wedge$ and $\vee$, and the quantifications $\forall$ and $\exists$. The constraints can be either equality or inequality constraints and are defined as **Matrix-Vector** tuples. The **Precond** models the manifest loop bounds and arrays indices in a description. A **Relation** is a function formed by (in)equality constraints between two **Expr**'s. A **Relation** models the non-manifest parts in a description. An **Indef** is a function used to model the indefinite iterator construct.

3.3 Semantic Functions

The semantic functions defined in this section give a procedure to translate the abstract syntax elements of Table 1 in terms of the semantic domains of the formal model. Especially loop constructs and indexed signals which are our main concern are separated of the other parts of the description. The semantic functions needed for the denotation of the abstract syntax elements are given in Table 2. If x is a phrase of the abstract syntax, then $[\![x]\!]_{y,z}$ is the function defining the meaning of x in terms of the formal model, with as inputs the syntactic phrase x, and the semantic arguments y and z.

Other functions needed in the definition of the semantic functions are: R(p,e) which takes the expression of a **Precond-Expr** tuple, P(p,e) which takes the precondition of a **Precond-Expr** tuple, and # which appends a new row to a **Vector** or **Matrix**.

$[\![f]\!]$	:	$\rightarrow$ **PredSet**
$[\![s]\!]$	:	**Vector** $\times$ **Matrix** $\rightarrow$ **PredSet**
$[\![e]\!]$	:	**Vector** $\times$ **Matrix** $\rightarrow$ **Precond** $\times$ **Exp**
$[\![n]\!]$	:	**Vector** $\rightarrow$ **Matrix**

Table 2: Semantic Functions

3.3.1 manifest expressions $[\![$n $]\!]$

A manifest expression is a combination of constants, arithmetic operators and iterator indices and is computable at compile time. It is used in the definition of array indices and loop bounds and restricted to unimodular affine expressions on the iterator indices. Each manifest expression is modeled by a constraint **Matrix** consisting out of one row. The product of this **Matrix** with the input **Vector** $\bar{i}$ modeling its index vector, gives the initial syntactical phrase. The first element of each row of the constraint **Matrix** is a reference to the time, the last element to a constant, the other values refer to the surrounding loop indices.

e.g. $$[\![i+3j+2]\!]_{\begin{pmatrix} t \\ j \\ i \end{pmatrix}} \overset{def}{=} [0\ 3\ 1\ 2]$$

- A **manifest constant** results in a **Matrix** consisting out of a row vector with all index elements zero except for the constant place. This element is equal to the value of the constant.
$$[\![c_m]\!]_{\bar{i}} \overset{def}{=} [0_1.......0_n\ c_m] \quad | \ n = length(\bar{i})$$
- A **manifest variable**. When the variable is not one of the indices of the surrounding loops, then the variable is a symbolic constant filled in at the last column of the **Matrix** defining **manifest variable**. Otherwise, a one is filled in at the place according to the index element.
$$[\![m]\!]_{\bar{i}} \overset{def}{=} \begin{cases} [0_1...1_x....0_n\ 0] & | \ \bar{i}[x] = m, n = length(\bar{i}) \\ [0_1...0_n\ m] & | \ \bar{i}[x] \neq m, n = length(\bar{i}) \end{cases}$$

3.3.2 expressions $[\![$e $]\!]$

Expressions are represented by **Precond-Expr** pairs. An expression is defined by its syntactical phrase, a **Vector** $\bar{i}$ which is a vector modeling the surrounding time and loop indices, and a **Matrix** I_M which models the relative time and index values.

e.g. $$[\![a[2i]@1]\!]_{\begin{pmatrix} t \\ i \end{pmatrix}, \begin{bmatrix} 1 & 0 & 0 \end{bmatrix}} \overset{def}{=}$$
$$[\![a[2i]]\!]_{\begin{pmatrix} t \\ i \end{pmatrix}, \begin{bmatrix} 1 & 0 & -1 \end{bmatrix}} \overset{def}{=}$$
$$[\![a]\!]_{\begin{pmatrix} t \\ i \end{pmatrix}, \begin{bmatrix} 1 & 0 & -1 \\ 0 & 2 & 0 \end{bmatrix}} \overset{def}{=}$$
$$\forall t_1, i_1 \ \ (t_1 = t-1 \wedge i_1 = 2*i),\ a(t_1, i_1)$$

- A **variable** is defined by a **Precond-Expr** pair. The **Precond** is derived from the given constraint **Matrix** I_M, and the index **Vector** $\bar{i}$. The **Expr** is a function, defined by its variable name. The arguments for the function are variables $\bar{k}$.
$$[\![x]\!]_{\bar{i},I_M} \overset{def}{=} \forall \bar{k}\ \ \bar{k} = I_M * \bar{i},\ x(k_1, ..., k_m) \ | \ m = length(\bar{i})$$
- A **delayed signal**(@) is defined by its expression with as input argument the index **Matrix** (I_M) shifted in the time dimension by the factor of the manifest delay value.
$$[\![e@n]\!]_{\bar{i},I_M} \overset{def}{=} [\![e]\!]_{\bar{i},I_M - [\![n]\!]_{\bar{i}}}$$
- An **array** is defined by its expression with the denotation of the manifest index appended as a new row to the input argument (I_M).
$$[\![e[n]]\!]_{\bar{i},I_M} \overset{def}{=} [\![e]\!]_{\bar{i},I_M \# [\![n]\!]_{\bar{i}}}$$

3.3.3 statements $[\![$s $]\!]$

Each statement is defined by a predicate set (**PredSet**). Each statement has three inputs: a **Vector** $\bar{i}$ which defines a vector with all iterator names surrounding the statement, a constraint **Matrix** C_M which models the constraints defined by the surrounding loop indices and the time dimension, and a **Vector** $\bar{j}$ which models the bound iterator names of $\bar{i}$.

- A **definition** is defined by a **Precond-Relation** tuple. The **Precond** is defined by the constraint **Matrix** for the loop bounds C_M, and the constraint matrices generated by the left and right operands of the **definition**. The **Precond** forms the constraints for the arguments of the function defined by the **Relation** between the expressions of the left and right operands.
$$[\![e_1 = e_2]\!]_{\bar{i},C_M,\bar{j}} \overset{def}{=} \ \exists \bar{j}\ \ C_M * \bar{i} \geq 0 \wedge$$
$$\mathcal{P}([\![e_1]\!]_{\bar{i},I_M}) \wedge \mathcal{P}([\![e_2]\!]_{\bar{i},I_M}).\ \mathcal{R}([\![e_1]\!]_{\bar{i},I_M}) \equiv \mathcal{R}([\![e_2]\!]_{\bar{i},I_M})$$
$$where:\ m = length(\bar{i}),\ I_M = [1\ 0_1.......0_m]$$
$$e.g.\ [\![a[2i] = a[2i]@1]\!]_{\begin{pmatrix} t \\ i \end{pmatrix}, \begin{bmatrix} 0 & 1 & 0 \\ 0 & -1 & N \end{bmatrix}, (i)} \overset{def}{=}$$
$$\forall t_1, i_1, t_2, i_2\ \ \exists i\ \ (0 \leq i \leq N)\ \wedge\ (t_1 = t \wedge i_1 = 2*i)$$
$$\wedge\ (t_2 = t-1 \wedge i_2 = 2*i).\ \ a(t_1, i_1) \equiv a(t_2, i_2)$$

- The **composition** of two statements form a **PredSet** defined by the logical conjunction of the two statements.

$$[\![s_1 ; s_2]\!]_{\overline{i}, C_M, \overline{j}} \overset{def}{=} [\![s_1]\!]_{\overline{i}, C_M, \overline{j}} \wedge [\![s_2]\!]_{\overline{i}, C_M, \overline{j}}$$

- An **iteration** is defined as the **PredSet** defined by its body ($[\![s]\!]$) with as inputs the new index **Vector**, the new constraint **Matrix**, and the new bounded index **Vector**.

$$[\![(i : n_1 \,..\, n_2) :: s]\!]_{\overline{i}, C_M, \overline{j}} \overset{def}{=}$$
$$[\![s]\!]_{\overline{i}\#i,\; C_M\#([\![i - n_1]\!]_{\overline{i}\#i}\#[\![n_2 - i]\!]_{\overline{i}\#i}), \overline{j}\#i}$$

- An **indefinite iteration** ($[\![do\ s\ until\ e1]\!]$) is a loop which fixes a sequential ordering. It is defined by a **Precond-Indef** tuple. It is modeled by the *DO ... UNTIL* function in the **Indef** with constraints on its inputs (**Precond**) formed by the outer loops of the indefinite iteration (C_M), and the constraints imposed by the *until* condition ($\mathcal{P}([\![e_1]\!]_{\overline{k}, I_M})$). The body of the indefinite iteration defines a **PredSet**. The sequential interpretation of the *DO ... UNTIL* construct, introduces a hierarchy in the set of predicates.

$$[\![do\ s\ until\ e_1]\!]_{\overline{i}, C_M, \overline{j}} \overset{def}{=}$$
$$\exists \overline{j}, t_{sub}\ \ C_M * \overline{i} \geq 0 \wedge t_{sub} \geq 0 \wedge \mathcal{P}([\![e_1]\!]_{\overline{k}, I_M}).$$
$$DO\ [\![s]\!]_{\overline{k}, [], ()}\ UNTIL\,(\mathcal{R}([\![e_1]\!]_{\overline{k}, I_M}))$$
$$m = length(\overline{i}),\ \overline{k} = t_{sub}\#tail(\overline{i}),\ I_M = [1\ 0_1 ......0_m]$$

4 CAD Implementation and Results

The proposed approach has been implemented and tested on several examples. In our test-vehicles, the descriptions are given in applicative DFL [16].

Key results of applications taken from several domains are shown in Table 3. In the left column the different alternative descriptions are cited which are validated against their original specification. The second column gives the values of parameters used in the example. The last column gives the CPU times needed to do the verification. The verification has been done for the example of Fig. 1 for both symbolic values of N and M, and instances of N and M. For none of the values of N and M, a difference in value for the CPU time could be measured. This result is expected since the complexity of the method and its CAD implementation are independent on the sizes of the loop bounds.

Another example includes an indefinite loop (Fig. 4). The test vehicle is a LU decomposition algorithm, extended with a construct that does a check on the processed result of the LU decomposition. If the final result does not satisfy the *until* condition, the LU decomposition has to be done again on an updated matrix until a final value is derived. The delay in the body of the indefinite loop is local to the indefinite loop and refers to the previous execution of the loop. The translation step into the domain based model, of the current CAD implementation can not yet handle the indefinite iterator. Therefore a minor manual intervention in the front-end was currently necessary.

spec. vs. altern. descr.	parameters	CPU(s)
example1	N=N,M=M	0.2
	N=10,M=10	0.2
	N=10000,M=10000	0.2
LU	optimized	0.3
	interch.L1,L2,L3	0.3
	split.L3,L1,S2	0.3
CRD	optimized	0.3
	non-procedural	0.3
Int.Video	optimized	1.3
	non-procedural	1.7

Table 3: Results on HP715 workstation

```
    do(
       a[][][0] = G(a_new[][]@1);
L1:    (k : 1 .. N)::
        begin
L2:      (i : k + 1 .. N)::
          begin
S1:        a[i][k][k] = a[i][k][k + -1] / a[k][k][k-1];
L3:        (j : k + 1 .. N)::
            begin
S2:          a[i][j][k] = a[i][j][k + -1] -
                          a[k][j][k-1] * a[i][k][k-1];
            end;
          end;
        end;
     a_new[][]= H(a[][][N],b);
  until (F(a_new[N][N]) > 0)
```

Figure 4: LU decomposition specification

Using transformations such as imperfect loop interchange, loop distribution and loop split, a new loop ordering for the LU description can be derived [17]. New orderings can result in descriptions that are very different from the original description. The optimized description has been validated for the separate steps with the current CAD tools (Table 3). As indicated in the column with the CPU times, the complexity of our verification approach is independent on the number or type of transformation steps. Moreover, it has to be emphasized again that the CPU times are independent on the size of the loop parameter N.

A third demonstrator is a Contour Regularity Detector (CRD) application. This algorithm is used in a robot vision application where robust contour tracing has to be performed on complex images corrupted by the presence of noise. The example contains data dependent array expressions.

The fourth example describes an interlaced video reformatting where indices of the form [a∗i+b] are present.

The last two examples have been validated for two alternative descriptions: an optimized description in terms of memory usage, and a non-procedural executable initial description. Because of the non-procedural code, verification by means of simulation would be impossible in these cases, even if large CPU times would be acceptable.

5 Conclusion

In this paper, a formal verification method of system level transformations for arbitrary loop nests is presented. The method is based on a formal model related to regular array synthesis models but extended with sequential constructs. The method is especially suited for applications which exhibit many complex loops and arrays, like image and video applications. The results obtained on many realistic test-vehicles show that our approach is independent on the size of the loops, or the number of transformations that have been done. Moreover the CPU times are negligible. Two descriptions are proven behaviorally equivalent by a combination of pattern matching and domain operations (partly implemented with the OMEGA Test[17] toolbox).

The verification method can be efficiently combined with SFG tracing [7]. The latter can prove the equivalence over different abstraction levels, as long as index and loop ordering at each of these levels is equivalent. In combination, the two complementary approaches provide a very powerful formal verification approach.

Acknowledgments Thanks go to our colleges at IMEC and the partners in the SPRITE (ESPRIT 2260) and NANA2 (BRA632) projects for their cooperation.

References

[1] B.J.S. De Loore, P. Crombez, A. Delaruelle, P. Sheridan, R. Woudsma, C. Niessen, J. Biesterbos, W. Gubbels, and W. Repko. The design of a competitive asic for the consumer market using the PIRAMID design system. In *Proc. IEEE ASIC 92 Conference*, pages 520–524, 1992.

[2] P. Lippens, J. van Meerbergen, A. van der Werf, W. Verhaegh, B. McSweeney, J. Huisken, and O. McArdle. Phideo: A silicon compiler for high speed algorithms. In *Proc. EDAC*, pages 436–441, Amsterdam, Feb. 1991.

[3] F. Franssen, F. Balasa, M. van Swaaij, F. Catthoor, and H. De Man. Modeling multi-dimensional data and control flow. *IEEE Trans. on VLSI systems*, 1(3):319–327, Sept. 1993.

[4] U. Banerjee. *Loop transformations for restructuring compilers : The foundation.* Kluwer, 1993.

[5] M.E. Wolf and M.S. Lam. A loop transformation theory and an algorithm to maximize parallelism. *IEEE Trans. on Parallel and Distributed Systems*, 2(4):452–471, Oct. 1991.

[6] F.Franssen, L.Nachtergaele, H.Samsom, F.Catthoor, and H.De Man. Control flow optimization for fast system simulation and storage minimization. In *Proc. EDAC*, pages 20–24, Paris, March 1994.

[7] L. Claesen, F. Proesmans, E. Verlind, and H. De Man. SFG-tracing: A methodology for the automatic verification of MOS transistor level implementations from high level behavioral specifications. In *Int. Workshop on Formal Methods in VLSI Design*, 9 - 11 Jan. 1991.

[8] P. Quinton and Y. Robert (eds.). *Algorithms and parallel VLSI architectures II.* Elsevier, 1992.

[9] F. Catthoor and L. Svensson (eds.). *Application-driven architecture synthesis.* Kluwer, 1993.

[10] Z. Chamski, H. Le Verge, C. Mauras, and P. Quinton. Interactive design of parallel algorithms using the ALPHA du Centaur environment. In *Int. Workshop on Compilers for Parallel Computers*, pages 399–410, Paris, Dec. 1990.

[11] Simon Finn, Michael P. Fourman, Michael Francis, and Robert Harris. Formal system design - interactive synthesis based on computer-assisted formal reasoning. In *Int. Workshop on Applied Formal Methods for Correct VLSI Design*, pages 97–110, Nov. 1989.

[12] M.C.Mc. Farland. Formal analysis of correctness of behavioral transformations. *Formal Methods in System Design*, 2:231–257, 1993.

[13] P.F.A. Middelhoek. Transformational design of digital circuits. In *Proc. of the Seventh Computersystems Workshop*, pages 57–69, Eindhoven, Nov. 1993.

[14] P. Feautrier. Dataflow analysis of array and scalar references. *Int. Journal of Parallel Programming*, 20:23–53, 1991.

[15] H. Samsom, F. Franssen, F. Catthoor, and H. De Man. Verification of loop transformations for real time signal processing applications. In *VLSI Signal Processing VII*, pages 208–217. IEEE, Oct. 1994.

[16] Mentor Graphics Corp. *DSP Architect DFL User's and Reference Manual, Software Version 8.2_5*, 1993.

[17] W. Pugh. The Omega Test: a fast and practical integer programming algorithm for dependence analysis. In *Proc. Supercomputing'91*, Nov.

[18] P. Le Guernic. The SIGNAL programming environment. *Algorithms and parallel VLSI architectures II*, pages 347–358, 1992.