Using Modified Interval Analysis in System Verification

I. Ugarte, P. Sanchez
Microelectronics Engineering Group, TEISA Department, ETSIIT, University of Cantabria
Avda. los Castros s/n, 39005 Santander, Cantabria, Spain
{ugarte, sanchez}@teisa.unican.es

Abstract—Interval arithmetic was originally developed to
determine rounding errors in floating-point computations but it is
now used in a wide variety of applications from constraint
solvers and global optimizers to power and timing analysis of
software processes. The objective of interval analysis (IA) is to
determine the output ranges (or intervals) of a computation set.
The main problem of classical interval analysis is the
overestimation of the output ranges and its dependency on the
coding of the system behavior. In this paper, a modified interval
analysis method is presented. The method reduces the interval
overestimation and it is independent of the coding. This modified
interval analysis is the kernel of a new verification technique that
enables the verification of functional properties in system level
descriptions and obtains functional test vectors.

Index Terms—Interval Arithmetic, System Level Verification.

I. INTRODUCTION

As system complexity grows, designers describe it at higher
abstraction levels and spend more effort on verification. In
order to confront this growing complexity, it is essential to
define verification methodologies that allow the validation of
the design during the specification step, at system level.
Simulation is the most widely used verification technique, but
even if coverage metrics are used, it has several problems (test
bench definition, completeness, etc). Another possibility is to
use formal verification techniques. Some of these are based on
transforming the system description into a functionally
canonical form, such as BDDs [11], and deriving the solution
from this structure. Other verification techniques test the
satisfiability of properties (e.g. SAT-based verification [2, 7,
8, 9, 10]) or combine both approaches. However, these
previously commented approaches generally suffer from
exponential worst-case complexity [2], because they use
Boolean representations of the system that increase the
number of signals and operators during the verification
processes.

This paper proposes a methodology that uses a Finite State
Machine (FSM) technique to identify execution paths that
reach assertions. This technique is combined with a modified
non-linear solver to find counterexamples. Our approach
assumes that the data path has only arithmetic/relational
operations so all Boolean operators are converted into integer
arithmetic expressions and control constructions. Thus, the
main differences from the previously commented approaches
are that the proposed technique uses a higher abstraction level
(integer/real equations instead of Boolean equations) and
handles multiplications efficiently (non-linear equations). This
implies an important cost reduction when designs with a
complex data path are verified. The main drawbacks of the
proposed approach are that it can not handle efficiently
designs with a lot of word-level logic operators or execution
paths.

The kernel of the proposed technique is a MODified
Interval Analysis (MODIA). The objective of classical IA is to
determine the output ranges (or intervals) of a computation set
[12]. Although this technique has been used previously in
verification (e.g. timing and power analysis in software [13]),
it has important drawbacks: overestimation of the output range
and expression dependency. For example, if the behavior of
the ‘y’ output is represented by the equation ‘y=(x*x)-x’ and
the ‘x’ input is defined in the range [-2,3], it is possible to
derive that the output ‘y’ will be defined in the range [-9,11]
using interval arithmetic [12]. However, if the output is
defined by the equivalent equation ‘y=x(x-1)’, the derived
range will be [-9,6]. Both approaches are overestimations of
the correct range, [-1/4,6].

The main objective of the proposed technique is to find an
input range that verifies a set of properties and/or constraints,
reducing the classical interval analysis overestimation and
being independent of the coding [14]. Additionally, the
proposed algorithm computes points instead of classical
intervals, allowing the reduction of the number of
computations.

II. SYSTEM MODELING

In our approach, we assume that the system is described at
behavioral level as a set of statements that operate with integer
data. The addition, subtraction and multiplication operators
are directly supported as well as `if` control statements.
Concerning loop statements, a predefined number of iterations
are unrolled in a similar way as the time frame expansion in
Automatic Test Pattern Generation (ATPG) and Bounded
Model Checking [15]. Boolean operations are converted into
integer expressions as Fig. 1 shows.

The left column VHDL description (a) contains a single bit
(o1) and word-level (o2) Boolean operations. The first
operation is directly converted (b column) into an integer
expression but the second conversion needs 24 if statements.
These control statements increase the number of execution
paths and this reduces the algorithm efficiency.

---

1 This work has been partially supported by the Spanish MCYT through the
TIC-2002-00660 project.
Variable a,b,c: std_logic;
variable m,n,t: std_logic_vector (7 downto 0);
... 
a:= b and c; - - o1
m:= n or l; - - o2 
...

Variable a,b,c: integer range 0 to 1;
Variable m,n: integer range 0 to 255;
Variable t1,t2,t3: integer range 0 to 255;
... 
a:= b*c; - - o1
t1:=n; t2:=l; t3:=0;
if(t1>127 or t2>127) then 
t3:=t3+128;
end if;
if(t2>127) then t2:=t2-128; end if;
end if; -- bit 7
... -- bit ...
m:=t3; - - o2

a) Boolean operators  b) Converted integer operators
Fig. 1. Conversion from Boolean to integer expressions

Other operators (division, etc) are currently supported with the same restrictions as commercial RTL synthesis tools (e.g. Synopsys VHDL Compiler).

With the previously commented consideration, every execution path in the behavioral description can be modeled with polynomials (that model the system behavior) and a set of constraints that model the if-statement conditions and properties/assumptions [16].

The main goal of the modified interval analysis (MODIA) is to find the bounds of non-linear polynomial equations. The algorithm is presented in section III, which is divided into five parts. The first part explains the basic MODIA idea for one dimension, the second shows the formal demonstrations of the bounds, the third is an example of interval evaluation, the fourth presents the proposed interval-oriented arithmetic and the last part, the extension to N dimensions. The verification algorithm (that is presented in section IV) uses these bounds to find input integer intervals that satisfy the predefined set of constraints, defined with non-linear polynomial inequalities. Section IV also presents the execution-path extraction algorithm. Finally, section V presents some experimental results.

III. MODIFIED INTERVAL ANALYSIS

A. Basic Idea

At behavioral level, the functionality can be represented by polynomial functions. Let P(x₁, x₂, ..., xₙ) be the polynomial that describes the behavior of a system ‘S’ with an input space of N integer variables that take positive values: x₁ ∈ [x₁₁,x₁ₛ₁] (0 ≤ x₁₁ ≤ x₁ ≤ xₛ₁₁), ..., xₙ ∈ [xₙ₁,xₙₛₙ] (0 ≤ xₙ₁ ≤ xₙ ≤ xₛₙₙ). The algorithm can also handle systems in which some variables take positive values and others negative values. If the same variable takes positive and negative values, its input interval is split into two parts (positive and negative subinterval) and considered independently by the algorithm. The analysis of intervals with negative values is totally equivalent to the analysis with positive values and it has been omitted to simplify the algorithm.

In order to find a maximum and minimum bound of a polynomial, P(X), its expression is separated into two parts: the positive (P⁺(X)) and negative part (P⁻(X)). A very important property of the positive part (polynomial in which all the monomials take positive values) is that it is an increasing function. Similarly, the negative part (polynomial in which all the monomials take negative values) is a decreasing function. The expression of these polynomials could be very complex (see equation 8) and, in order to bound them, two simple functions are defined: P_R(X) and P_M(X). In the case of a design with only one input, x, the expression of these functions is presented in (1):

\[ P_R(x) = mx + c \]
\[ P_M(x) = \alpha x^n + \beta \]

The ‘n’ parameter is equal to the order of the polynomial that is being bounded (positive or negative part). The ‘m’, ‘c’, ‘α’ and ‘β’ are calculated by evaluating the positive and negative polynomials at the interval extremes (x₁ and xₛ). The demonstrates are shown in subsection III.B.

It is easy to see that the original polynomial, P(X), is now bounded by these simple functions (2):

\[ P^+_R(X) \geq P^+(X) \geq P^+_M(X) \]
\[ P^-_M(X) \leq P^-(X) \leq P^-_R(X) \]
\[ P(X) = P^+(X) + P^-(X) \]
\[ P^+_R(X) + P^-_M(X) \geq P(X) \geq P^+_M(X) + P^-_R(X) \]

The maximum and minimum value of the bound functions (P_R + P_M) can be calculated easily with classical derivative-based algorithms.

B. Formal Demonstrations

Theorem 1. There is a upper bound function (P_R(x)) for any increasing polynomial (the positive part) in an interval of input [xₛ, xₛ].

Proof. Let the positive polynomial, P⁺(x), be

\[ P^+(x) = a_N x^N + a_{N-1} x^{N-1} + \ldots + a_1 x + a_0 \]

where \( a_N > 0, a_i \geq 0 \ \forall i \in [0, N-1] \). In order to simplify the demonstration, the following variable change is carried out.

\[ 0 \leq t \leq 1 \]
\[ x = (x_S - x_I)^n t + x_I \]

The polynomial with the new variable is

\[ P^+(t) = b_N t^N + b_{N-1} t^{N-1} + \ldots + b_1 t + b_0 \]

with \( b_N > 0, b_i \geq 0 \ \forall i \in [0, N-1] \). The upper bound function

\[ P_R(t) = mt + c \]

is the same at the extremities.
\[ P^+(t) = b_0 = P_k(0) = c \]
\[ P^+(t) = \sum_{j=0}^{N} b_j = P_N(t) = m + c \] (3)

Any \( t \) must comply with
\[ P_k(t) - P^+(t) \geq 0 \] (4)

Substituting the expressions (3) into (4) and simplifying,
\[ P_k(t) - P^+(t) = \sum_{j=0}^{N} b_j t^j + b_0 - \sum_{j=0}^{N} b_j t^j = \sum_{j=1}^{N} b_j (t^j - t^j) + (b_0 - b_0) = \]
\[ = \sum_{j=1}^{N} \sum_{i=0}^{j-1} b_j (t^i - t^i) \geq 0 \quad \forall t \in [0, 1] \]

In the case of the decreasing polynomial (the negative part), the demonstration that \( P_M(x) \) is a lower bound function of \( P(x) \), is similar.

Theorem 2. There is a lower bound function \( P_M(x) \) for any increasing positive polynomial in an interval of input \([x_l, x_u]\).

Proof. As the above demonstration, take \( P^-(t) \) to simplify the proof.
\[ P^-(t) = b_N t^N + b_{N-1} t^{N-1} + \ldots + b_1 t + b_0 \]
with \( b_N > 0, b_i \geq 0 \) \( \forall i \in [0, N-1] \). The lower bound function
\[ P_M(x) = ax^r + \beta \]
is the same at the extremities.
\[ P^-(0) = b_0 = P_M(0) = \beta \]
\[ P^-(t) = \sum_{j=0}^{N} b_j = P_M(t) = \alpha + \beta \] (5)

Any \( t \) must comply with
\[ P^-(t) - P^+(t) \geq 0 \] (6)

Substituting the expressions (5) into (6) and simplifying,
\[ P^-(t) - P_M(t) = \sum_{j=0}^{N} b_j t^j - \sum_{j=0}^{N} b_j t^j = \sum_{j=1}^{N} \left( b_j t^j - b_j t^j \right) + (b_0 - b_0) = \]
\[ = \sum_{j=1}^{N} b_j (t^j - t^j) \geq 0 \quad \forall t \in [0, 1] \]

In the case of the decreasing polynomial (the negative part), the demonstrations that \( P_M(x) \) is an upper bound function of \( P(x) \) is similar.

C. Example
The goal is to bound a one-dimensional polynomial (7). The input space is \( x \in [0, 6] \).
\[ P(x) = \frac{1}{50} \left( \frac{1}{4} x^6 - 1.9 x^5 + \frac{1}{4} x^4 + 20 x^3 - 17 x^2 - 55.6 x + 48 \right) \] (7)

As the inputs are always positive, the monomials with positive coefficient constitute the positive part and the rest the negative part. Their equations are shown in (8).
\[ P^+(x) = \frac{1}{50} \left( \frac{1}{4} x^6 + \frac{1}{4} x^4 + 20 x^3 + 48 \right) \]
\[ P^-(x) = \frac{1}{50} \left( -1.9 x^5 - 17 x^2 - 55.6 x \right) \] (8)

In order to calculate the coefficients of \( P_M \) and \( P_k \) functions, it is only necessary to evaluate the interval extremes \( (X_l, X_u) \). In the next section, an efficient algorithm to evaluate these values is presented. In this example, the extreme values are:

<table>
<thead>
<tr>
<th>x value</th>
<th>( X_l = 0 )</th>
<th>( X_u = 6 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Positive polynomial</td>
<td>0.96</td>
<td>327.12</td>
</tr>
<tr>
<td>Negative polynomial</td>
<td>0</td>
<td>314.4</td>
</tr>
</tbody>
</table>

Table I. Values at the extremities of (4)

Fig. 2 presents the bound of polynomial \( P(x) \). Its maximum bound is (9) and its minimum bound is (10).
\[ M(x) = -0.044 x^5 + 54.36 x + 0.96 \] (9)
\[ m(x) = 0.00699 x^6 - 52.4 x + 0.96 \] (10)

In order to calculate the maximum and minimum values, elementary derivative calculus is applied. The result is that the function \( P(x) \) is bounded by the interval \([-180.80, 177.06]\). This result overestimates the real range, \([-1.07, 12.7]\), but it is better than the classical IA results, \([-313.44, 327.12]\).

A classical way to improve the results is to split the original interval into two subintervals and apply the algorithm to both. MODIA improves this process thanks to an important property: only the extreme values are needed to calculate the bound functions. Thus, these values can be reused to calculate the subinterval bounds. For example, if we want to calculate the bounds of the \([0, 3]\) interval after we have calculated the \([0, 6]\) interval, we will only have to evaluate the polynomials \( P^+ \) and \( P^- \) at the point ‘3’, because the value at the other extremity (point ‘0’) has been calculated during the estimation of the \([0, 6]\) interval.

D. Data Flow Graph Evaluation
In order to avoid the extraction of the analytical expression of the polynomials \( P^+ \) and \( P^- \) and/or symbolic computations, the values that the polynomials take, are calculated directly from the data flow graph. In order to do this, a new arithmetic is defined (Fig. 3).

The idea is to replace the classical single-value operands and/or symbolic computations, the values that the polynomials take, are calculated directly from the data flow graph. In order to do this, a new arithmetic is defined (Fig. 3).
verify the following properties:

\[ P(X) = P^+(x) + P^-(x) \]

Fig. 2. Bounds of the polynomial.

The diagonal line plays an important role in our technique and, in order to identify it easily, a variable change is
made. Let \( T \equiv (t_1, \ldots, t_N) \) verify the following properties:

\[ 0 \leq t_1 \leq 1, \ldots, 0 \leq t_N \leq 1 \]

\[ x_1 = (x_{t_1} - x_{t_1}) \cdot t_1 + x_{t_1} \]

... \[ x_N = (x_{t_N} - x_{t_N}) \cdot t_N + x_{t_N} \]

It is very easy to characterize the diagonal points with the new vector:

\[ \text{Diagonal Point} = (t, t, \ldots, t, t) \quad 0 \leq t \leq 1 \]

Additionally, the definition of the subspaces is also very simple. For example, the subspace \( 'i' \) is defined as the set of points \( (t_1, \ldots, t_i, \ldots, t_N) \) in which the value of the \( 'i' \) dimension coordinate is less than the other values, that is, the set that verifies that \( t_i \leq t_j \) \( \forall \neq i, j \in [1, N] \) and \( t_i \in [0, 1] \).

In order to calculate the bound of a subspace, the algorithm makes use of the increasing and decreasing character of the polynomials \( P^+ \) and \( P^- \). Let \( T = (t_1, \ldots, t_N) \) be an \( N \) dimensional input vector and \( \Delta T = (\Delta t_1, \ldots, \Delta t_N) \) be a positive shift \( (\Delta t_i \geq 0, \ldots, \Delta t_N \geq 0) \). Thus, the positive (increasing) and negative (decreasing) polynomials verify:

\[ P^+(T) \leq P^+(T + \Delta T) \]

\[ P^-(T) \geq P^-(T + \Delta T) \]

Let us focus on the polynomial \( P^+ \) (the discussion for the polynomial \( P^- \) is similar). As a consequence of the previously commented property, in the subspace \( 'i' \), the value of the polynomial at a point, \( T \), verifies that:

\[ P^+(T_D) \leq P^+(T) \leq P^+(T_E) \]

Expression (13) allows transforming the N-dimension problem into a 1-dimensional problem. The basic idea is that \( T_D \) and \( T_E \) are functions of only one variable, \( t_i = t \). Thus, we could re-write (13) as:

\[ P^+(T_D) \leq P^+(T) \leq P^+(T_E) \]

Where:

\[ T = (t_1, \ldots, t_i, \ldots, t_N), t_i \leq t_j \forall \neq i \]

\[ T_D = (t_1, \ldots, t_i, \ldots, t_N), \text{Diagonal point} \]

\[ T_E = (t_1, \ldots, t_i, \ldots, t_N), \text{Extreme point} \]

\[ \text{Diagonal Point} = (t, t, \ldots, t, t) \quad 0 \leq t \leq 1 \]

P.D and \( P_E \) are 1-dimensional polynomials and they can be bounded with the section III approach (equation (2)). Thus, equation (14) can be re-written and completed with equation (2) as:

\[ P^+(T_D) \leq P^+(T) \leq P^+(T_E) \]

\[ P^-(T_D) \leq P^-(T) \leq P^-(T_E) \]

\[ P(T) = P^+(T) + P^-(T) \]

\[ P^+(T_D) + P^-(T_D) \leq P(T) \leq P^+(T_E) + P^-(T_E) \]

Table II. Input order calculation

<table>
<thead>
<tr>
<th>Operation</th>
<th>Operand 1 order</th>
<th>Operand 2 order</th>
<th>Result order</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>n</td>
<td>m</td>
<td>( \max(n, m) )</td>
</tr>
<tr>
<td>-</td>
<td>n</td>
<td>m</td>
<td>( \max(n, m) )</td>
</tr>
<tr>
<td>*</td>
<td>n</td>
<td>m</td>
<td>( n \cdot m )</td>
</tr>
</tbody>
</table>

Expression (14) allows the polynomial values, the polynomial order is also needed to calculate the bound functions. This order is also calculated with an operation redefinition, in a similar way to polynomial values. Table II presents the new operation definitions.

E. Extension to N dimensions

The basic idea of the extension to N dimensions is to split the problem into N sub-problems and apply the one-dimension approach (commented in subsection III.A) to each sub-problem.

All these subspaces will have a common edge: the straight line (diagonal line) that joins the \( X_i = (x_{t_1}, \ldots, x_{t_N}) \) and \( X_S = (x_{t_1}, \ldots, x_{t_N}) \) extremities.

The diagonal line plays an important role in our technique and, in order to identify it easily, a variable change is performed. The elements of the new input vector \( T = (t_1, \ldots, t_N) \) verify the following properties:
Where:

\[ T = \{ t_1, \ldots, t_N \}, t_j \leq t_i, \forall j \neq i \]
\[ P_R(t) = mt + c \]
\[ P_M(t) = \alpha t^n + \beta \]

In order to calculate the coefficient of the polynomials \( P_R \) and \( P_M \) for the 'i' subspace, the value of polynomial \( P \) has to be evaluated at 3 points: \( X_I = (0, \ldots, 0) \), \( X_S = (1, \ldots, 1) \) and \( X_i = (1, \ldots, 1, 0, 1, \ldots, 1) \). All the subspaces need the value of the \( X_I \) and \( X_S \) points, thus, only \( N+2 \) points will be needed to bound \( P^+ \) in the \( N \) subspaces.

It is very important to highlight that every evaluated point is shared by \( 2^N \) \( N \)-dimensional intervals, thus the average number of points evaluated per interval is close to 1 in our verification algorithm.

Using equations (11), the MODIA algorithm estimates the \( P(T) \) bounds in every subspace. After the \( N \) subspaces have been bounded, the algorithm determines the bounds of \( P(T) \) in the complete input space.

IV. SYSTEM-LEVEL VERIFICATION BASED ON INTERVALS

The verification process begins from the behavioral description of the system. This specification is compiled in a control and data flow graph (CDFG) that contains only arithmetic (+, -, *) and relational operations, assertions and conditional statements ("if" constructions). At behavioral level, scheduling has not been performed, thus there is no state information in the graph.

For every path, there will be 'm' conditions that have to be satisfied and 'n' assertions that have to be checked. These constraints are modeled as the polynomial \( C_i \) (condition i) and \( A_j \) (assertion j), thus:

\[ C_i(X) > 0 \quad 1 \leq i \leq m \]  
\[ A_j(X) > 0 \quad 1 \leq j \leq n \]

As we can see, all the constraints are converted into inequalities of the type "greater than".

The main goal of the verification algorithm is to find an input functional vector, \( X \in [X_I, X_S] \), which satisfies the 'm' (16) inequalities and violates at least one of the (17) assertions. A point \( U \) violates the assertions polynomial \( A_j(X) \) if \( A_j(U) > 0 \). In order to calculate the bounds of the polynomials \( C_i \) and \( A_j \), the MODIA technique is used.

Additionally, during interval analysis the MODIA technique evaluates every polynomial in \( N+2 \) points. Let \( X_k \) (1 \( \leq k \leq N+2 \)) be the set of points that MODIA has evaluated. When MODIA bounds a path for a particular input interval and the interval and path have not been previously removed, a heuristic cost function (18) is defined:

\[ \text{Cost} = 0.5 \times R + 0.5 \times P \]  

Parameter ‘R’ measures the probability that the conditional constraints are verified. Parameter ‘P’ measures the probability that an assertion is violated.

The complete pseudo-code of the verification algorithm is presented in the next section.

A. Pseudo-code of the Algorithm

The main goal of the algorithm is to find an input vector that violates an assertion. In order to do this the algorithm executes the next main loop for every assertion/property in the description.

Enumerate all paths that reach the property
Select the complete input space as input interval
For every path
   Evaluate the cost function (eq. 18)
End for;
Loop (while there is a path to explore)
   Select the path with maximum cost
   Call path_evaluation function
End loop;

The path_evaluation function tries to find an input interval that violates the assertion/properties. Its pseudo-code is:

Initialize the interval list with the complete input interval
Loop(while there is an input interval in the interval list)
   Select the input interval with maximum cost
   Split the selected interval into 2 subintervals
   Loop (for each subinterval)
      Loop (for every condition)
         Bound the polynomial with MODIA
         If(lower bound of the conditional > 0) then
         Eliminate the condition
         Elseif(upper bound of the conditional \( \leq 0 \)) then
         Remove the input interval from the list
         Next subinterval iteration
         Else
         Calculate the parameter R
         Endif;
      End loop;
   End loop;
   Loop(for every assertion of the path)
      Bound the polynomial with MODIA
      If(a point violates one assertion) then
         Countereexample detected
      Elseif(upper bound of the assertion \( \leq 0 \)) then
         Remove the input interval from the list
         Next subinterval iteration
      Else
      Calculate the parameter P
      Endif;
   End loop;
   Evaluate the cost function of equation 18
   Insert the new interval in the interval list
End loop;
End loop;
A counterexample cannot be found.
V. EXPERIMENTAL RESULTS

In order to validate the proposed technique, four behavioral system descriptions have been proposed. The test benches are data-dominated and they include non-linear behaviors and from 8-bit to 36-bit wide data path operations. Table III shows some properties of the selected test bench.

The performance of the proposed methodology has been compared with two classical model checking tools (SPIN [4] and SMV[3]) and two commercial tools: a demo version of [6] (Commercial1) and a full version of [5] (Commercial2). The results are presented in the Table IV. The table shows the CPU time that the tools need to generate a correct answer. The term OFL (Out of Limit) normally identifies situations in which the program was aborted because the computer does not have enough memory resources. The CPU times in Table IV correspond to seconds on a Pentium III with 256 MB of RAM at 300 Mhz under Windows 2000. In the case of the “Commercial2” tool, the CPU times correspond to Sun Fire V120 Ultra-Sparc IIi with 512 MB of RAM running at 550 Mhz.

The tool ‘SPIN’ is not able to automatically explore the input space, thus an exhaustive search procedure has been included in the system description. SMV and SPIN are able to check simple designs but they run “out of limits” (OFL) when the size of the input space grows. The “Commercial 1” tool (Verity-Check) is able to verify nearly all the proposed examples. Only in one case, the tool was not able to provide a solution after 12 hours of CPU time. The “Commercial 2” tool (FormalCheck) runs on a different platform (Sun Workstations) thus, its execution times are not totally equivalent to the other tools. This tool has problems with the “Space4” test bench because it is not able to handle relational operators with more than 30 bits.

The proposed methodology spends less than a second to verify the proposed test bench. These results are better than other tools and only SPIN is able to obtain similar results in two examples.

VI. CONCLUSIONS AND FUTURE WORK

A method to check properties (or assertions) in behavioral-level system specifications has been presented. The method is based on a modified interval analysis (MODIA) that can be directly computed over the CDFG, requiring (on average) the computation of about one new point per interval, reducing the classical overestimation problem of the IA techniques. Moreover, it is independent of the coding style of the algorithm. Using MODIA, the proposed verification algorithm explores the input space, looking for input intervals in which the violation of the assertion is more probable. The algorithm can also be used to automatically generate functional vectors that exercise predefined paths or properties. These vectors could be used to increase functional coverage metrics or replace random test generators.

The proposed technique has been evaluated with different examples and it produces better results than commercial and classical research model checkers.

The future work in this line is focused on two areas. The goal of the first is to extend the proposed methodology to handle RT-level descriptions. The second is to try to improve the methodology in order to avoid unrolling the loops and support more operators, concurrency, temporal expressions, and modular descriptions.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>#input vectors</th>
<th>#solutions</th>
<th>#constraints</th>
<th>Polynomial order</th>
</tr>
</thead>
<tbody>
<tr>
<td>Simple</td>
<td>256</td>
<td>2509</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>Conditional</td>
<td>256</td>
<td>1</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>Space3</td>
<td>256</td>
<td>1</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>Space4</td>
<td>256</td>
<td>2</td>
<td>6</td>
<td>4</td>
</tr>
</tbody>
</table>

Table III. Test bench properties

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>SPIN</th>
<th>SMV</th>
<th>Comm. 1</th>
<th>Comm. 2</th>
<th>MODIA</th>
</tr>
</thead>
<tbody>
<tr>
<td>Simple</td>
<td>1 s</td>
<td>84 s</td>
<td>9 s</td>
<td>7 s</td>
<td>1 s</td>
</tr>
<tr>
<td>Conditional</td>
<td>1 s</td>
<td>OFL</td>
<td>120 s</td>
<td>100 s</td>
<td>1 s</td>
</tr>
<tr>
<td>Space3</td>
<td>OFL</td>
<td>OFL</td>
<td>22796 s</td>
<td>1637 s</td>
<td>1 s</td>
</tr>
<tr>
<td>Space4</td>
<td>OFL</td>
<td>OFL</td>
<td>&gt; 12 hours</td>
<td>OFL</td>
<td>1 s</td>
</tr>
</tbody>
</table>

Table IV: Comparison with property checkers.

REFERENCES