Use of Non-linear Solver to Check Assertions of Behavioral Descriptions

I. Ugarte, P. Sanchez
Microelectronics Engineering Group. TEISA Deparment. ETSIIT. University of Cantabria
{ugarte, sanchez}@teisa.unican.es

***Abstract*— Verification has become an essential aspect of design flow because of the increasing design complexity. According to the latest report of the International Technology Roadmap for Semiconductor, the challenge will be to develop new design-for-verifiability techniques and verification methods for higher levels of abstraction. Several Design-for-Verifiability methodologies (DFV) have been proposed and Assertion-based Verification (ABV) is one of the most promising. In order to automatically verify assertions at the higher abstraction levels, it is necessary to improve the performance and capabilities of current constraint solvers.**

This paper presents a new technique based on non-linear solvers that automatically checks assertions in behavioral descriptions of hardware systems.

The main contribution of this work is the definition of a methodology that allows using continuous non-linear solvers to verify behavioral descriptions. These descriptions are modeled with a set of integer polynomial inequalities. The technique provides better results than integer solvers and it is applied to real designs, such as Viterbi decoders or vocoder digital filters.

***Keywords*—Assertion Checker, Design for Verifiability, Non-linear solvers.**

I. INTRODUCTION

Verification has become the main bottleneck of the design flow as a result of two processes. Firstly, the functional complexity of modern designs is continuously growing. Secondly, the greater emphasis on other aspects of the design process has produced important progress (automated tools for logic synthesis, place-and-route, etc), leaving verification as the main bottleneck that will be a barrier to further progress in the semiconductor industry if there is not a major breakthrough (2004 report of the International Technology Roadmap for Semiconductors [1]).

Formal verification techniques are beginning to gain acceptance and they sometimes complement simulation methods in the process of verification. The main goal of formal hardware verification is to prove the functional correctness of a design instead of simulating some vectors. Traditionally, formal techniques are classified into three groups: equivalence checking, model checking and theorem proving technique. Most of these methodologies use Boolean equations to model some aspects of the design.

Popular techniques to solve these Boolean equation systems (or satisfiability problems) are based on Binary Decision Diagrams (BDD) [2]. BDDs are used to represent binary output value constraints in a canonical form. The main disadvantage of the use of BDDs is the "memory explosion" problem because of the huge size of the diagram even for medium complexity design. Several optimizations have been proposed to compress the diagram (OBDD, ROBDD, etc).

Another way to solve Boolean equations is to use a SAT solver. This technique avoids the exponential space blow-up of BDD [3]. The main drawback is the handing of arithmetic operators. These operators are transformed into a large number of Boolean formulas which reduce the SAT efficiency and limit its application domain. To overcome these disadvantages, hybrid satisfiability approaches, such as HSAT [4], have been proposed. The goal is to combine a SAT and a linear programming solver. The SAT checker is used to solve the logic equations and the linear programming solver is used to check the feasibility of the arithmetic equations. These two engines operate in separate domains. The performance of HSAT is limited by the heuristics that choose the set of assignments to Boolean variables. Other similar approaches (e.g. LPSAT [5]) are based on mixed integer linear programming (MILP) techniques [5]. However, general ILP solvers tend to be inefficient in solving real satisfiability problems. Firstly, they do not directly handle nonlinear operators (multipliers). Secondly, they have numerical convergence problems, and they are sensitive to a number of internal parameters. Other tools are based on Constraint Logic Programming (CLP) techniques [6]. The CLP works at Boolean level and/or Integer domain and it has similar problems to MILP techniques.

This paper presents a different approach to the verification of designs that are modeled with Boolean equations and/or non-linear expressions. The technique is based on a commercial global non-linear solver. The goal of this type of solver is maximizing a non-linear equation (the assertion equation in this paper) while satisfying a set of non-linear constraints that model conditional statements and discontinuous functions in this work. The commercial solver LINDO [7] is an example of a global non-linear solver and it has been used in this work. It has an Application Programming Interface (LINDO API) that has been designed to solve a wide range of optimization problems, including linear programs, mixed integer programs and general nonlinear non-convex programs. The global optimizer of LINDO API uses branching to split the feasible region into subregions and bounding to obtain a valid bound on the optimal objective value in each region. This type of solvers has problems with integer equation systems because integer variables introduce non-smooth problems, thus memory and solution time may rise exponentially with them.

In order to avoid this problem, the proposed verification methodology uses the non-linear solver in the real domain (maximum efficiency and minimum CPU time and memory requirements) and defines a new technique to find integer solutions from the real-domain solver results. The main advantages of the proposed verification techniques are the efficient handling of non-linear systems and the relatively low CPU requirements.

II. System Modeling

In this paper the hardware system is described at behavioral level as a set of concurrent processes. The proposed verification technique is focused on individual process validation, thus only one process will be considered. This process is suspended in an initial wait statement until the input values change. After this, the process body is executed until the initial statement (wait statement) is reached. Figure 1 shows this behavior. The straight arrows model the external inputs (X_i) and the outputs (Z_i). The gray box represents the 'wait' statement and the dashed line the memories or state variables (I_i). The dotted lines represent the execution paths (functionality) of the process.

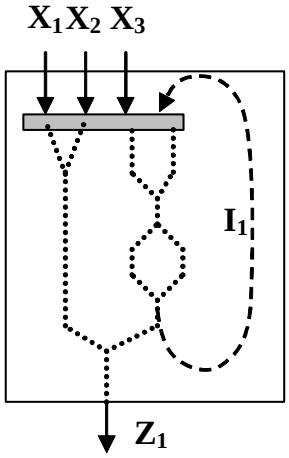

Fig. 1. System Model.

The model includes integer variables and the directly supported operators are addition, subtraction, multiplication, and relational. Other operators have to be transformed into equivalent polynomial equation systems. For example, the modulus operation (D%N) is transformed into the following integer equation system:

$$A = D\%N \Rightarrow \begin{cases} D = D\_Quotient \times N + A \\ A * D\_Quotient >= 0 \\ A \in [-N+1, N-1] \end{cases} \quad (1)$$

It is assumed that all the previous equations take integer values.

Other operators (e.g. bit selection, bit-wise logic operator, etc) are transformed in a similar way.

Word-level logic operators (e.g. "or reduce") and bit-level logic operations are transformed into integer polynomials. For example, the logic equation "a = b or c" is transformed into "a = b + c - (b*c).

Concerning control statements, conditional 'if' statements are totally supported. Every conditional statement is transformed into a two-equation system (see figure 2). A new variable (K) identifies the selected path. This integer control variable takes values in the range [0, 1]. When the variable takes the value '1', the 'true' path is activated (C=f1). When the variable takes the value '0', the 'false' path is chosen (C=f2).

if(A > B) then
C = f1;
else
C = f2;
end if;
	K *(A – B) + (1-K)*(B – A + 1) > 0
	
	C = K*f1+ (1 – K)*f2;
	K ∈ [0, 1]

Fig. 2. Model of the conditional statement.

Finally, the loop operators are handled with restrictions. The 'for' loops are totally unrolled when the number of iterations can be statically determined. Figure 3 shows an example. During the unrolling process, several variable assignations are modified. For example, the 'x' and 'y' variables of figure 3 change in every iteration, thus new variables (x1,…,x4, y1,…,y4) are defined to model the intermediate values. Additionally, only the last iteration output assignments are translated (r variable in figure 3).

for(i = 0; i < 4; i++)
x = y + 5;
y = x + z;
r = y*z;
end for;
	x1 = y + 5;
	y1 = x1 + z;
	x2 = y1 + 5;
	y2 = x2 + z;
	x3 = y2 + 5;
	y3 = x3 + z;
	x4 = y3 + 5;
	y4 = x4 + z;
	r = y4*z;

Fig. 3. Model of the for statement.

The 'while' loops cannot normally be totally unrolled because it is not possible to statically determine the number of iterations. In this case, the algorithm will unroll a new iteration in every step. This means that the algorithm will unroll one iteration in the first step, two in the second and it will repeat the process up to a user-defined maximum number of iterations. If several 'while' loops are nested, the number of unrolled statements will grow exponentially.

With the previously commented transformation, the process body (dotted line in figure 1) will be modeled with polynomials whose input space will change in every process execution. The assertion to be checked and the conditional statements will be modeled with polynomial inequalities.

III. System Modeling Example

In this section, the generation of the polynomial model of a typical communication system component (a Viterbi decoder) is presented. This set of polynomial equations can

be solved by the LINDO API package. The Viterbi decoder is a common component of forward-error-correction (FEC) modules.

Typically, a Viterbi decoder algorithm has four steps [8]: determine branch metrics, accumulate path distances, normalize path distances and determine the survivor path with a trace back algorithm that extracts the decoded symbols.

A simple Viterbi decoder will be presented in this section. It receives two inputs (x and y) whose probabilities of taking value '1' (or likelihood) are represented by a value in the range [0,255]. The Trellis diagram will only have 4 nodes in each slice, thus only 8 branches are possible. Figure 4 presents the behavioral description of the branch metric calculation (first step of one iteration of the Viterbi decoder) on the left. On the right, the figure presents the equivalent polynomial model of the system

M0 = x + y;	x + y – M0 = 0;
M1 = x + (255 – y);	255 + x – y – M1 = 0;
M2 = (255 – x) + y;	255 – x + y – M2 = 0;
M3 = (255 – x) + (255 – y);	510 – x – y – M3 = 0;

Fig. 4. Model of first part of the Viterbi decoder algorithm.

Figure 5 presents the behavioral description of the second step of the Viterbi algorithm (accumulate path distance) on the top. On the bottom, it presents the set of polynomial equations that models it. The 'for' loop has been totally unrolled and the conditional statements have been converted into polynomial inequalities.

Other steps of the Viterbi algorithm are transformed in a similar way. As a conclusion, an iteration of the Viterbi decoder with two inputs and four Trellis nodes is transformed into a set of 28 polynomial restrictions (or inequalities) and 32 internal variables.

IV. Verification Methodology

The goal of the proposed verification technique is to find a point that fulfills the set of integer inequalities that model the hardware system and violates an assertion. Three steps have been defined (figure 6):

1.- Polynomial model generation

The behavioral description is transformed into an inequality system that can be handled mathematically. In order to reduce the complexity of the problem, the discontinuous functions (conditional statements, round operators,…) are transformed into a series of expressions with a collection of additional variables and constraints (see section II). Integer variables are also transformed into real variables, although they are not mathematically equivalent (non-linear problem relaxation). These modifications allow the use of an efficient non-linear solver based on function derivatives.

2.- Solve the inequalities system

A non-linear solver is used to find a solution in the real domain. If there is a real solution, an algorithm that finds an integer solution has to be applied (step 3). If there is no real solution and the input description has "while" statements, a new iteration of a 'while' loop will be added to the polynomial system description before executing step 2 again. In order to limit this unrolling process, the user defines the maximum number of iterations that a 'while' loop can be unrolled. If there is no real solution and the loop-unrolling limit is reached, the system cannot have an integer solution, thus the system will fulfill the assertions until the pre-defined unroll limit.

```
for ( J=0; J < 3; J++) {
    switch (J) {
        case 0:
            upper_branch_distance(J) := M0 + current_distance(0);
            lower_branch_distance(J) := M3 + current_distance(2);
            break;
        case 1:
            upper_branch_distance(J) := M3 + current_distance(0);
            lower_branch_distance(J) := M0 + current_distance(2);
            break;
        case 2:
            upper_branch_distance(J) := M1 + current_distance(1);
            lower_branch_distance(J) := M2 + current_distance(3);
            break;
        case 3:
            upper_branch_distance(J) := M2 + current_distance(1);
            lower_branch_distance(J) := M1 + current_distance(3);
            break;
    };
    if(upper_branch_distance(J) <= lower_branch_distance(J))
        new_distance(J) := upper_branch_distance(J);
    else
        new_distance(J) := lower_branch_distance(J);
};
```

```
-- Modeling of 'for' sentences
// J = 0
M0 + current_distance0 –  upper_branch_distance0 = 0;
M3 + current_distance2 – lower_branch_distance0 = 0;
K0*(l_b_d0 - u_b_d0) + (1 – K0)*(u_b_d0 - l _b_d0) > 0;
K0*l_b_d0 + (1 – K0)*u_b_d0 - new_distance0 = 0;
// J = 1
M3 + current_distance0 –  upper _branch_distance1 = 0;
M0 + current_distance2 –  lower_branch_distance1 = 0;
K1*(l_b_d1 - u_b_d1) + (1 – K1)*(u_b_d1 - l _b_d1) > 0;
K1*l_b_d1 + (1 – K1)*u_b_d1 - new_distance1 = 0;
// J = 2
M1 + current_distance1 –  upper _branch_distance2 = 0;
M2 + current_distance3 –  lower_branch_distance2 = 0;
K2*(l_b_d2 - u_b_d2) + (1 – K2)*(u_b_d2 - l _b_d2) > 0;
K2*l_b_d2 + (1 – K2)*u_b_d2 - new_distance2 = 0;
// J = 3
M2 + current_distance1 –  upper _branch_distance3 = 0;
M1 + current_distance3 –  lower_branch_distance3 = 0;
K3*(l_b_d3 - u_b_d3) + (1 – K3)*(u_b_d3 - l _b_d3) > 0;
K3*l_b_d3 + (1 – K3)*u_b_d3 - new_distance3 = 0;
```

Fig. 5. Model of second part of the Viterbi decoder algorithm.

The solver provides partial results of the inequality system solution. If one of these partial results violates the assertion and fulfils the inequalities, the solver will be aborted and the partial solution will be used as a possible counterexample. The goal of the solver is to find the point that maximizes the equations. Our goal is to find a point that fulfils them, thus the solver is aborted as soon as a partial solution is detected with an important reduction of the solver execution time.

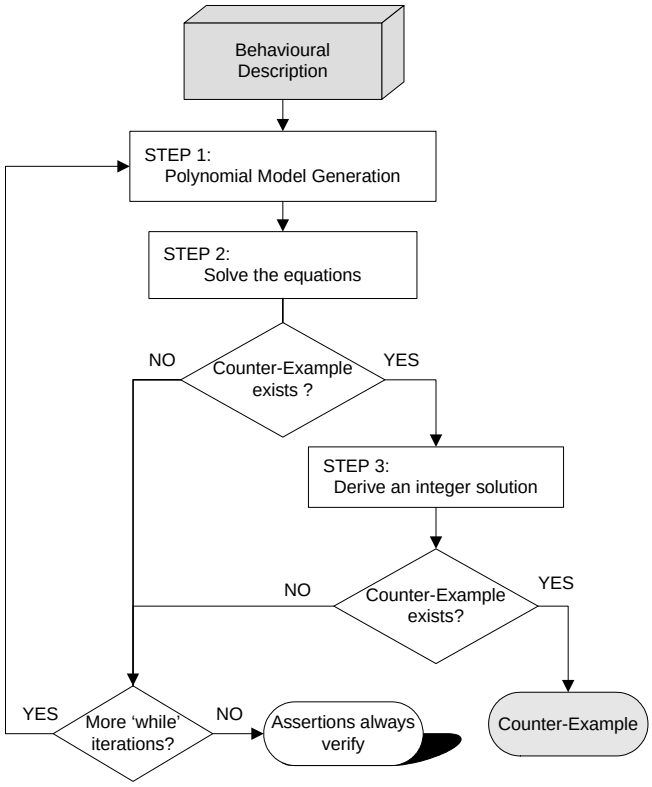

Fig. 6. Verification Methodology.

3.- Derive an integer solution from the real solution

The goal is to find an integer solution taking into account the information that the real-domain solution provides. The technique defines two steps: variable rounding and branch-and-bound exploration of the solution space.

The first step is to round the real variables to the closest integer value. If there are 2 possible values (for example, 11.50 could be rounded to 11 or 12), a value will be randomly selected. Figure 9 presents the decisions that the integer-solution searcher algorithm takes with the figure 7 example. In this figure, the type uint8 models an integer with range 0 to 255. The 'space3' assertion verifies that the 'ret' variable is never greater than 340. In figure 9, the ranges of the inputs are included in ellipses. The rectangular forms contain the solutions that the solver provides in the real domain and the hexagonal forms, the rounded integer points.

```
void space3 (uint8 x, uint8 y, uint8 z)
{
    int temp, dat, ret;
    dat = (x – 110)^2 – (y – 28)^2;
    temp = dat – (z – 170)^2;
    if (10000 > temp)
        if (6*y – 2*x – 4*z > 0)
            ret = x + y + z;
    else
        ret = 0;
    ...
    Assertion → ret ≤ 340;
}
```

Fig. 7. 'Space3' example.

The non-linear solver provides a first solution (x=95.434, y=118.148 and z=129.255) that is rounded by the searcher algorithm (top hexagonal form) to an infeasible solution (the assertion is not violated or the inequalities are not fulfilled with the top hexagonal form values).

In this case, the second step (branch-and-bound based exploration) is applied. Firstly the farthest value from an integer is selected. In Figure 9, the farthest value is 95.434, thus variable x is selected. Secondly, the input space of the selected variable is split into two parts: values greater than the integer part (x>95) of the solution and values less than or equal to the integer part (x<=95). This generates two new set of polynomial inequalities. These sets are solved with the non-linear solver, thus two new set of solutions and maximum values of the assertion can be generated. If a new set has no solution, its branch will be removed. The algorithm will select the set that produces a higher assertion value and it will repeat the searching process. This process will be finished in a branch if one of these conditions is verified:

1.- The solver cannot find a solution, thus the problem is infeasible.

2.- The solver provides a solution, but the assertion is always verified.

In these cases, the current branch will be removed and the last unselected branch will be selected. This process is repeated until a counterexample is found or all the branches are removed (the assertion is always fulfilled).

V. Experimental Results

In order to validate the proposed technique, two examples at behavioral level have been proposed. The first example is the 'Pre_Process' module of the GSM standard (ETSI EN 301.245, December 1997). This module is a second order high pass IIR digital filter with cut off frequency at 80 Hz and 4 taps. Two assertions have been inserted into the code. The first assertion verifies that the accumulated values are not saturated. The second assertion checks if the accumulated values are again saturated after a previous saturation.

The second example is the previously commented Viterbi decoder algorithm. It is a soft decoder with a rate of ½, a constraint length of 3 and a survivor window length of 16. The inserted assertion checks if there is overflow in the maximum value of the path metric accumulator

The CPU times in Table III correspond to seconds on a Pentium IV with 2 GB of RAM at 2.8 GHz under Windows XP.

Table I shows the results of the verification of the first assertion of the GSM filter. The first column shows the number of execution of the process, the second column is the number of the inputs. The range of the inputs is [-32768, 32767]. The third and fourth columns are the number of integer variables and restrictions that model the behavioral description. The last column is the time, in seconds, that the proposed algorithm takes to obtain a result. During the first 21 iterations, the filter values are not saturated and the

assertion is verified. In the 22nd-iteration, a counter example is found.

TABLE I
RESULT OF THE FIRST ASSERTION OF THE FILTER.

Iteration	*#inputs*	*#variables*	*#restrictions*	*Time*
1	1	3	4	0
2	2	10	8	0
3	3	18	15	0
4	4	26	22	0
5	5	34	29	0
6	6	42	36	0
7	7	50	43	1
8	8	58	50	0
9	9	66	57	0
10	10	74	64	0
11	11	82	71	0
12	12	90	78	12
13	13	98	85	8
14	14	106	92	2
15	15	114	99	40
16	16	122	106	38
17	17	130	113	30
18	18	138	120	64
19	19	146	127	74
20	20	154	134	66
21	21	162	141	133
22	22	170	148	21

The other assertion checks the second saturation after the first one (22 nd –iteration). The proposed methodology has found a second overflow at the 35th-iteration. It is interesting to analyze the evolution of the execution time of the non-linear solver (figure 8). Due to the heuristic algorithms that the solver uses, the execution time does not have a "common pattern" and it is very different for close problems. For example, the solver takes more than 9000 seconds to find a solution in iteration 33 but it only needs 33 seconds to solve iteration 34.

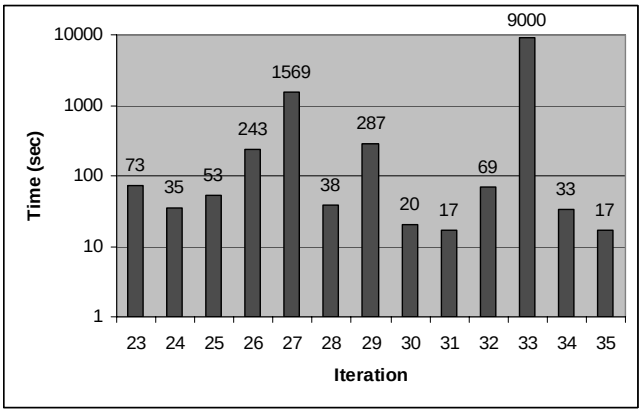

Fig. 8. Result of the Second Assertion of the filter.

The results of the Viterbi verification are shown in table II. In this case, the algorithm always fulfills the assertion. Some variables have been added to model the control statements.

In order to evaluate the methodology, the proposed examples have been checked with a commercial integer non-linear solver. This solver provides solution for only 2 iterations of the GSM filter and it is not able to solve an iteration of the Viterbi decoder.

TABLE II
RESULT OF THE ASSERTION OF THE VITERBI DECODER.

Iteration	*#inputs*	*#variables*	*#restrictions*	*Time*
1	2	32	28	0
2	4	65	58	0
3	6	98	88	1
4	8	131	118	1
5	10	164	148	2
6	12	197	178	3
7	14	230	208	4
8	16	263	238	5
9	18	296	268	7
10	20	329	298	8
11	22	362	328	10
12	24	395	358	11
13	26	428	388	14
14	28	461	418	16
15	30	494	448	19
16	32	527	478	21

VI. CONCLUSIONS AND FUTURE WORK

In this paper, a method to check assertions at behavioral level is presented. The behavioral description is transformed into a set of polynomial inequalities. The proposed methodology is able to derive an integer solution (counterexample) for this inequality set. The technique is based on a commercial non-linear solver that provides a real-domain solution that is used to find an integer solution of the problem. The proposed methodology is able to handle efficiently complex descriptions that cannot be directly checked with integer non-linear solver.

The future work includes the automatic generation of the polynomial inequality set from high-level HDL (for example, SystemC) and the extension of this technique to concurrent descriptions.

REFERENCES

[1] "The International Technology Roadmap For Semiconductor". 2004 Edition. Design. http://www.itrs.net/Common/2004Update/2004_01_Design.pdf

[2] R.E.Bryant, "Graph Based Algorithms for Boolean Function Manipulation", IEEE Transactions on Computers, vol. C-35, pp. 677-691, August 1986.

[3] A. Biere, A.Cimatti, E.M.Clarke, M. Fujita, Y. Zhu, "Symbolic Model Checking Using SAT procedures instead of BDDs*". Proc. Of DAC'99. 1999.

[4] F. Fallah, S. Devadas, and K. Keutzer, "Functional Vector Generation for HDL models using Linear Programming and 3-Satisfiability Infrastructure using the Unite Recursive Paradigm", in Proc. Of DATE 2000, 2000, pp. 232 – 236.

[5] Z. Zeng, P.Kalla, and M. Ciesielski, "LPSAT: A unified approach to rtl satisfiability", in Proc. DATE, March 2001, pp. 398-402.

[6] Zeng Z., Ciesielski M., and Rouzeyere B., "Functional test generation using constraint logic programming", in VLSI-SOC Conference, 2001.

[7] LINDO API., www.lindo.com, Lindo Systems Inc.

[8] John P. Elliott, "Understanding Behavioral Synthesis. A Practical Guide to High-Level Design", Kluwer Academic Publishers, 2000.

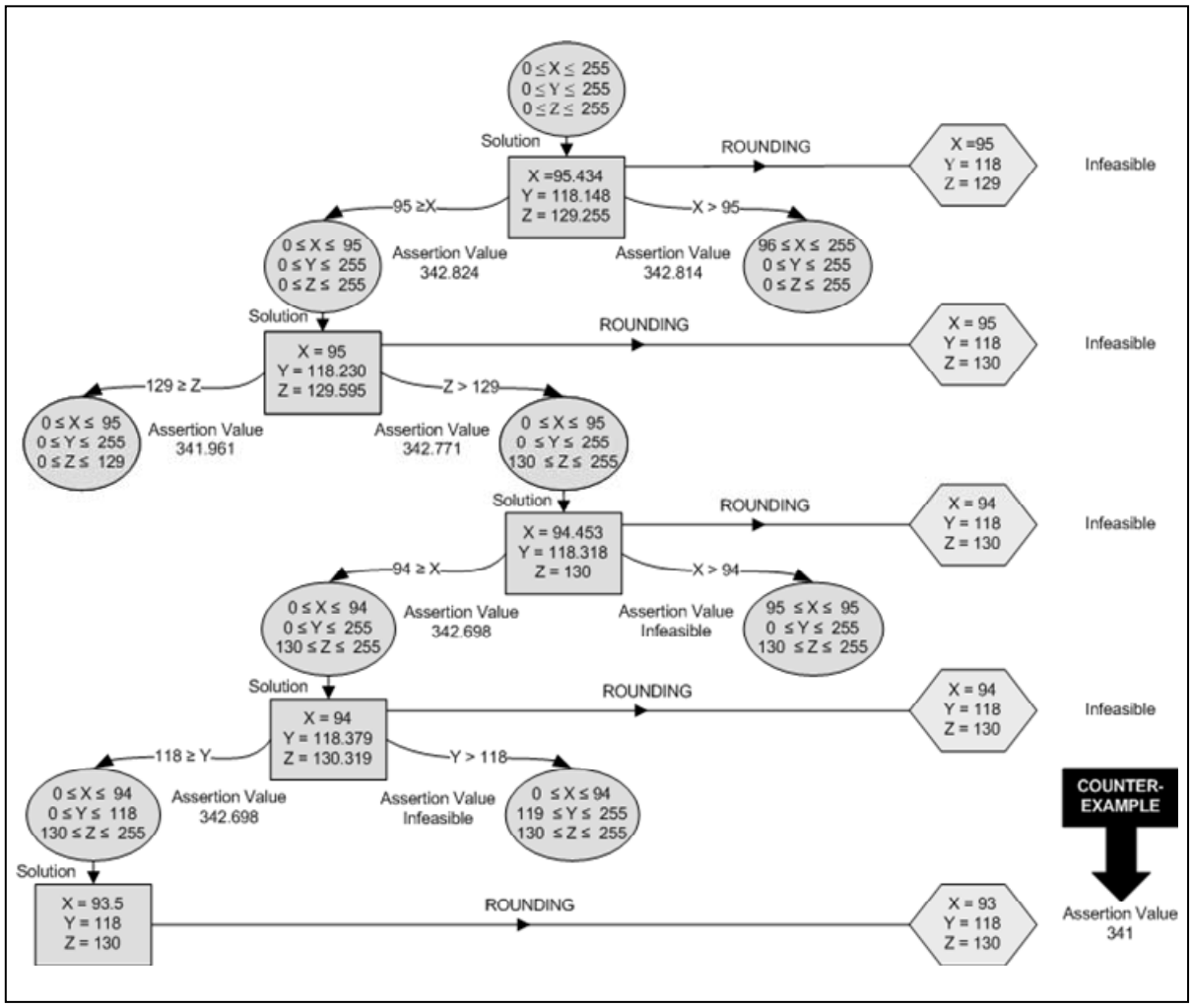

Fig. 9. Steps of the algorithm to find the integer solution in the ‘space3’ example.