Assertion Checking of Cyclic Behavioral Descriptions

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*— In order to confront the verification of more and more complex Systems, several Design-for-Verification methodologies (DFV) have been proposed. One of them, Assertion-based Verification (ABV) has recently emerged as the functional verification methodology capable of keeping pace with increasingly complex systems.**

This paper presents an ABV technique that automatically searches for counter-examples that violate user specified assertions in behavioral descriptions of hardware systems. The main contribution of this work is an assertion checking algorithm that allows applying interval-based techniques to cyclic descriptions while reducing path explosion problems.

***Index Terms*—Interval Arithmetic, Assertion Checker, Design for Verification.**

I. INTRODUCTION

The 2003 International Technology Roadmap for Semiconductors (ITRS) affirms that "Verification has become the dominant cost in the design process" and "Design conception and implementation are becoming mere preludes to the main activity of verification" [1]. Some studies show that up to 70% of the RTL design effort is spent on making sure that their chips meet specifications and perform as intended [2]. In order to reduce the verification cost several new "Design for Verification" (DFV) methodologies have been proposed. One of the most promising DFV methodologies is "Assertion-based Verification" (ABV) [4]. An assertion is a precise description of what behavior is expected. The main goal of an ABV technique is to verify that the user-specified assertions are not violated. Several dynamic (simulation-time assertion checking) and/or static (assertion checkers) methods have been proposed [14].

Additionally, the use of higher levels of abstraction allows many forms of verification to be performed much earlier in the design process, reducing time to market and lowering cost by discovering problems earlier [2][3]. In this context, several works have proposed polynomial-based specification models that allow representing arithmetic and logic operations and checking system properties and parameters [5][6][7].

In order to check properties of systems described with polynomials, a lot of constraint solvers and global optimizers have been proposed [15]. Some of them are based on Interval Analysis and they have mainly been used for power and timing analysis of software processes [8][9] and assertion checking [13].

One of the problems of these tools is to verify systems with cycles. Loops are basic control elements, commonly used in system descriptions, but they introduce important verification problems. For example, multimedia applications normally have a large number of loops. The verification by means of pseudo-exhaustive simulation in a workstation or (parallel) DSP board can be impossible due to the large amount of memory necessary [10]. Formal techniques (e.g. theorem provers, model checkers) could verify simple systems but they need too much designer knowledge to be automated and they are not able to verify medium size designs. Other BDD-based methods [12] have been proposed (like symbolic simulation [11]) but the verification effort grows so fast that the algorithm explodes even for medium size problems.

This paper presents a static assertion checking technique for hardware behavioral models, which are modeled with polynomials. The algorithm generates vectors automatically to detect the violation of the assertion. If no counter-example is found, the assertion is fulfilled by the description. The technique is based on a modified Interval Analysis and it reduces the verification effort because there is no need to explicitly unroll loops.

MODified Interval Analysis (MODIA) [13] is able to find an input space that violates some assertions while verifying all the control statements (if-then-else structure) of the path, which fulfils the assertion. The proposed verification algorithm expands interval analysis to handle cyclic descriptions without the need of explicitly unrolling the loops.

After this introduction, the hardware system description methodology is presented in section 2. Section 3 describes the interval analysis oriented modeling of control statements: conditional ("if-then-else") and endless loop (process) structures. Section 4 presents the verification algorithm and in section 5, an example is presented. Finally, experimental results and some conclusions and future work are commented.

II. SYSTEM MODELING

In this approach 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. The process is suspended in this statement until the input values change again (Fig. 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. Depending on the number of state variables, the process can be classified as:

a) Process without memory, when there are no memories or state variables.
b) Process with memory, when there are state variables.

The model only considers integer variables and the supported operators are addition, subtraction, multiplication and relational. Word-level logic operators (e.g. "or reduce") and the bit-level logic operations are transformed into integer polynomials and other operators (e.g. dividers) are supported with common RT-synthesis restrictions. Concerning control statements, the conditional 'if' statements are totally supported and the loop statements are supported with restrictions: only one loop (process loop) is supported during analysis, thus other loops have to be statically unrolled.

The process body (dotted lines in Fig. 1) is modeled with polynomials whose input space changes in every iteration. The assertions to be checked will be modeled with polynomial inequalities. In order to verify the assertions of the process, it is not necessary to unroll the loop; it would be enough to determine the next input space after iteration is executed and apply the verification algorithm (with the new input space) to the loop body.

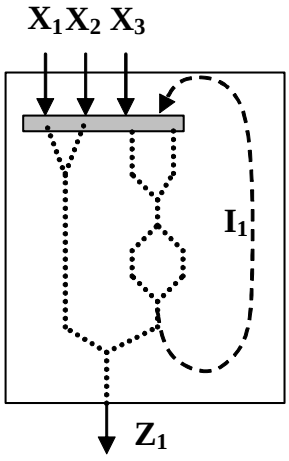

Fig. 1. System Model.

Thus, the proposed technique modifies the ranges of the process inputs (arrow lines in Fig. 1) every time a new iteration is executed. The verification process finishes when the complete input space is analyzed or an assertion is violated. Inside the process, every execution path in the behavioral description is described by a set of inequalities (which model the if-statement conditions and assertion) and polynomials (which model the path functional behavior). Fig. 2 shows a simple C-style example with only two paths in the process ("example" function). The path 'Then' has two inequalities: one to describe the control statement [Then1] and the other the assertion [Then2]. The other path ('Else') has the complementary inequality of the control statement [Else1] and the assertion [Else2].

```
int example (int x, int y) {
  DO {
    Wait until x, y:
    …
    if (5*y > x) {
      // Path Then
      ret = x + y;
    else
      // Path Else
      ret = x*x -y + 25;
    } ...
  } WHILE (true);
  Assertion → ret ≤ 255;
}
```

a) Behavioral description

Constraints:
X ∈ [0,255];
Y ∈ [0,255];
…

Path "Then" polynomials:
[Then1] 5*y - x > 0
[Then2] x + y - 255 > 0

Path "Else" polynomials:
[Else1] x - 5*y + 1 > 0
[Else2] x*x - y + 25 - 255 > 0
…

b) Polynomial description

Fig. 2. Polynomial description of a simple example.

The main disadvantage of this approach is that the number of the paths grows with '2^n', where 'n' is the number of conditional statements in the loop, in the worst case (no-nested conditional statements). 'Case' statement, can be translated into several nesting 'if-then-else' structures. If so , the number of paths is equal to the number of different 'case' options.

III. Analysis of the Loop Statements

The model of loop statement is very important in the proposed approach, so it will be discussed in this section. In order to verify assertions inside these structures, two important properties have to be considered:

1. The loop-body code is equal in all iterations.
2. Every time the loop (or process) is executed (new iteration), the input space of the loop-body is modified. The range of the internal variables for the next iteration is derived from the current iteration results.

A consequence of the first observation is that the same input and internal variable intervals will produce the same results. Thus, only the new portions of the resulting intervals (the new areas of the state variable ranges) have to be analyzed in the next iteration (second observation). These new input intervals reflect the differences between the input spaces of the previous iterations and the current iteration. A simple example is shown in Fig. 3.

The original input space (N-dimensional) is the vertically shaded area that represents the state variable ranges. The external input ranges are not represented because they are the same in all iterations. The execution of the process body (first iteration) generates a resultant space that is composed of horizontally and vertically shaded areas. The vertically shaded area has already been evaluated in the first iteration, thus only the horizontally shaded area has to be evaluated in

the second iteration (next input space). This process is repeated until the third iteration. In this iteration the process body execution generates a resultant space that is included in the previously evaluated input spaces. Thus all input space has been covered and the search step is finished.

The way to detect the conclusion of the verification process without a counter-example is to explore all the input space, that is, to reach an iteration in which the resultant space has already been evaluated. This exploration process will last a finite time because the hardware variable has a predefined range (finite number of bits) and every new iteration reduces the possible new input space.

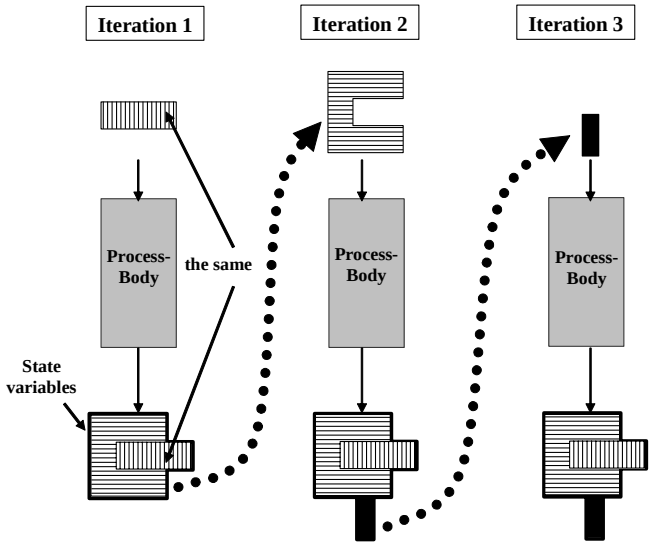

Fig. 3. Example of input space determination

IV. INTERVAL-BASED SYSTEM-LEVEL VERIFICATION

In order to verify a process, an interval analysis technique (MODIA [13], MODified Interval Analysis) is used.

A. *The MODified Interval Analysis*

The proposed verification algorithm is guided by this interval analysis technique. The goal is to calculate the bounds of an inequality system (one execution path), and identify input spaces (intervals) that fulfill the inequalities (the input constraints and 'if-then-else' structures) and violate the assertions. One drawback of this approach is the overestimation of the bounds. A classical solution to this problem is to split the original space into several spaces and apply the interval analysis to them. This reduces the overestimations but increases the algorithm computation complexity. In this proposal, this partition effort is used to find counter-examples (points that violate the assertions). Fig. 4 shows an example with two inputs: X1 and X2. The function P(X1, X2) models the property to verify. The gray areas are input space values that violate the property and the black points (extremities of intervals) are the input space points that MODIA evaluates. Using MODIA bounds, the partition technique selects a space and splits it into two pieces. During this process, it is possible that a new extreme point is selected inside a gray box (white point). This point violates the property, thus it is the target counter example. Some spaces can be deleted if an inequality is not fulfilled by all points of the input interval.

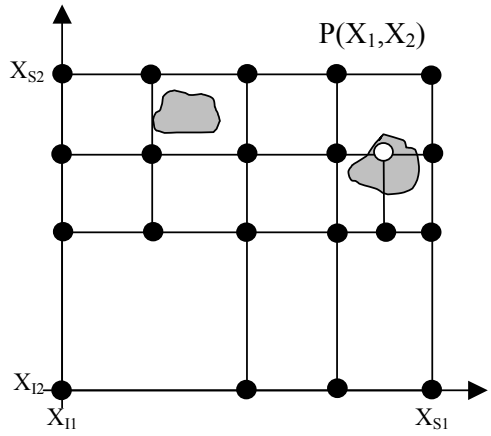

Fig. 4. Evaluated Input Space Points

B. *The algorithm*

The verification algorithm uses a breadth-first search (BFS) technique. First, the algorithm takes the complete process input intervals, and calculates the new intervals of the internal-variables for each path. The following step is the elimination of the part of the new ones that has already been evaluated in the first iteration. In the second iteration, these new intervals and the original external input intervals are applied to generate the new internal space. The evaluated part of this new space is removed. The following iterations repeat the steps until there is an iteration that violates an assertion. In this case, all iterations are removed and the complete space of the first iteration is split into two pieces to increase the precision of the bounds. The algorithm resumes the previously commented steps but with twice as many as the spaces in the first iteration.

Furthermore, the algorithm also calculates the internal values for each extreme point of the input intervals (they are the special points to find a counter-example). In the second iteration, for each new internal value, it calculates the new interval value for each special point of the external input intervals. For example, if there are 2 new internal values and the number of extreme points is 4 (number of external inputs + 2), the result is 8 new internal values, 4 for each new internal value. These steps are repeated for all iterations. If a counter-example is detected, the algorithm is stopped and the sequence of the inputs is shown.

When the partition of the input interval is done, new extreme points are added without eliminating the points that have already been calculated. These new points are evaluated in the following iterations until reaching the iteration in which the violation was produced. From this iteration, all points are considered for the next iterations.

The algorithm pseudo-code is the following:

```
Assign the special points of complete space as possible
    counter-examples.
Violation number = 0;
do {
  If (Violation number > 0) then
      The internal-variable intervals of the different
          iterations are removed.
```

The space of the first iteration is split.
Violation number = 0; Iteration number = 0;
Else
Evaluate the new extreme points in the iteration i.
If *(there is a counter-example) then*
Finish the algorithm: One counter-example is found. The assertion is not fulfilled.
End if;
Loop *(for each new internal-variable interval of the iteration i)*
Loop *(for each path)*
Calculate the internal-variable intervals for the iteration number i + 1.
If *(a violation exists) then*
Violation number = 1;
End if;
End loop;
End loop;
If *(Violation number is equal to zero) then*
Eliminate the evaluated part of the internal-variable intervals of the iteration number i + 1.
Next iteration: i = i + 1;
End if;
} while *(Violation number is different to zero or there is a new internal-variable interval of the iteration i or more).*
Finish the algorithm: The assertion is fulfilled over the whole input interval.

V. EXAMPLE

The example in Fig. 5 has only one process ('proc') with an external input variable, 'x', and an internal variable, 'y'. The initial value of 'y' is also an input of the 'proc' function.

```
int proc (int x, int y)  {
  // Constraints: 0≤x≤63, 32≤y≤63
  int temp, ret;
  DO {
      Wait until x;
      temp = (y – 110)^2 + (x + 57)^2;
[R1]  if (temp < 10000)
[R2]        if (y ≥ 4*x)      // Path  A
                ret = 2*y;
            else              // Path  B
                ret = x + 2*y;
    y = ret;
[A]   Assert(ret < 256);   // Assertion
  } WHILE (true);
}
```

Fig. 5. Example

The verification algorithm begins with the complete input process intervals (x ∈ [0,63], y ∈ [32,63]). For each iteration, the internal variable (y), constraint (R1 and R2), assertion (A) intervals are determined and the output of the interval extremities are computed. After 2 iterations, a possible violation is detected in several paths (framed text) but no extreme points are counterexamples. These intervals are shown in Fig. 6 and the evaluated points in the Table I. The '---' symbol marks input combinations that do not reach the assertion.

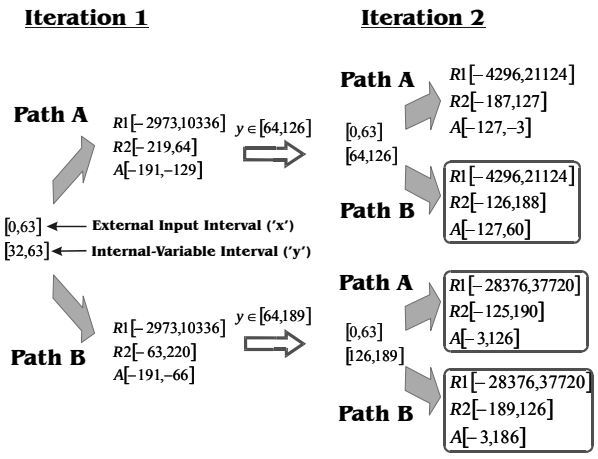

Fig. 6. 'Search' step

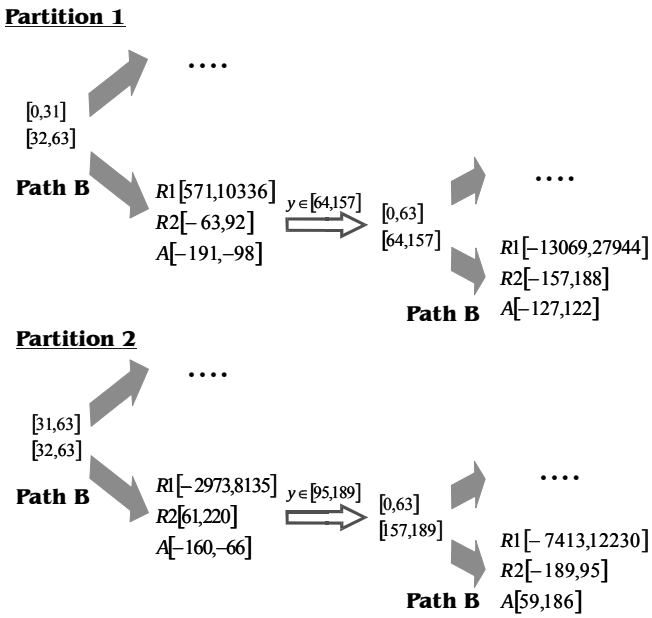

Fig. 7. 'Justification' step

In order to improve the interval accuracy, the first-iteration 'x'-input interval is split into two intervals: partition 1 (0, 31) and partition 2 (31, 63). Using the interval analysis algorithm, new intervals are generated (Fig. 7). Only the BB path is shown to simplify the figure. The assertion is still violated in both partitions. This analysis uses the values of several extremities that are shown in Table II.

TABLE I
Extreme points evaluated in 'search' step (fig. 6)

Iteration 1		Iteration 2	
Input(x,y)	*Output(y)*	*Input(x,y)*	*Output(y)*
(0,32)	64		
(0,63)	126		
(63,0)	---		
(63,63)	---		

TABLE II
Extreme points evaluated in 'justification' step (fig. 7)

(31,32)	---		
(31,63)	157	(0,157)	314
		(63,157)	377

One of these points (first iteration x=31, y=63; second

iteration x=63) violates the assertion, thus a counter-example has been detected and the algorithm finished.

VI. EXPERIMENTAL RESULTS

In order to validate the proposed technique, two sets of examples have been proposed. The first set (4 examples) includes data-dominated examples without memory, and the second set (2 examples) includes examples with memory (internal variable 'D'). The C-like example codes are shown in the appendix. The CPU times in Table III correspond to seconds on a Pentium III with 256 MB of RAM at 300 MHz under Windows 2000. In the case of the "BerkMin" tool, the CPU times of the first column correspond to Sun Fire V120 Ultra Sparc Iii with 512 MB of RAM running at 550 Mhz.

The examples without memory are used to compare the algorithm with classical model checking tools (SMV[14]) and a SAT tool (BerkMin 5.6[15]). The results are presented in Table III(a). It 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 tool 'SMV' is able to verify simple designs but the main disadvantage is that it runs "out of limit" (OFL) when the size of the input space grows. The "BerkMin" SAT results are presented in two columns. The first shows the "Sun Workstation" execution time that a synthesis tool (Synopsys Design Compiler) needs to generate the Conjunctive Normal Form (CNF). The second column shows the time that BerkMin needs to find the correct solution. The last column shows the proposed Assertion Checker results.

TABLE III
Comparison with property checkers.

	SMV	BerkMin		Assertion Checker
		Synt.	Verif.	
Simple	84s	180 s	<1s	1 s
Conditional	OFL	240 s	1s	1 s
Space3	OFL	240 s	1s	8 s
Space4	OFL	1020 s	36s	24 s

(a) Acyclic description

	SMV	Assertion Checker	Number of Evaluated Iterations
Linear	< 1s	1 s	10
Nonlinear	4.42 s	1 s	5

(b) Cyclic descriptions

The other set (examples with memory) is executed by the tool 'SMV' and the proposed Assertion Checker. The results are shown in Table III(b). The tool 'SMV' needs to unroll the loops to handle them, while the proposed tools handle loops without unrolling.

VII. CONCLUSIONS AND FUTURE WORK

In this paper, a method to check assertions at behavioral level is presented. The technique is based on a modified interval analysis (MODIA) that can be directly computed over the CDFG. In this way, the algorithm has been extended to handle processes (cyclic description) without unrolling. The algorithm can also be used to automatically generate functional vectors that exercise predefined paths or assertions. These vectors could be used to increase functional coverage metrics or random test generation.

The advantage of this method is the efficiency of handling data-dominated algorithms independently of the range of the data. However, the main disadvantage is the explosion of the number of paths with the number of 'if-then-else' structures.

During cyclic description verification, the algorithm looks for possible input combinations that violate an assertion taking into account all conditional paths. Thus, the memory consumption grows when the number of iterations increases. In future work, the depth-first search will be implemented to solve this problem. Additionally, heuristic metrics based on statistical probabilities will be used to choose the path with highest probability to reach a violation.

REFERENCES

[1] "The International Technology Roadmap For Semiconductor". 2003 Edition. Design. http://public.itrs.net/Files/2003ITRS/Home2003.htm

[2] Emil Girczyc. "Assertion-based verification streamlines design outsourcing". EEDesign, October25, 2002.

[3] R. Schutten. "Raising the Level of Abstraction Reduces Verification for System on Chip". The Synopsys Verification Avenue Technical Bulletin. Vol.3, issue 3, August 2003.

[4] A. de Geus. "Design for Verification: A new Paradigm". DVCon 2003 Keynote. Feb 2003. http://www.synopsys.com/corporate/exec_presentation/2003/DVCon2003_aart.pdf

[5] J. Smith and G. DeMicheli, "Polynomial Methods for Allocating Complex Components", DATE99, 1999.

[6] J. Smith and G. DeMicheli, "Polynomial Methods for Component Matching and Verification", Proc. of ICCAD'98 Conference. 1998.

[7] P. Sanchez, S. Dey, "Simulation-based System-level Verification Using Polynomials", HLDVT'99. 1999.

[8] D. Ziegenbein, F. Wolf, K. Richter, M. Jersak, R. Ernst, "Interval-Based Analysis of Software Processes", LCTES'01, PAGES 94-101, Snowbird, Utah, USA, June 2001.

[9] B.G. Ryder and M. C. Paull, "Elimination Algorithms for Data Flow Analysis", ACM Computing Surveys, Vol. 18, No. 3, September 1986, 277-316.

[10] M. Čupák, F. Catthoor, and H. De Man, "Efficient Functional Validation of System-Level Loop Transformations for Multi-media Applications", HLDVT 98, IEEE CS Press, 1998, p. 72-79

[11] D. W. Currie, A. J. Hu, S. Rajan, "Automatic Formal Verification of DSP Software", DAC 2000, pp. 130-135.

[12] S. Minato, "Generation of BDDs from Hardware Algorithm Descriptions", ICCAD'96.

[13] I. Ugarte, P. Sanchez, "Functional Vector Generation for Assertion-based Verification at Behavioral level Using Interval Analysis", HLDVT'03. 2003.

[14] K. L. McMillan, "Symbolic Model Checking: An approach to the State Explosion Problem". Kluwer Academic. 1993.

[15] E. Goldberg, Y. Novikov, "BerkMin: a Fast and Robust Sat-Solver", DATE'02. 2002.

[16] Verification tools: ttp://www.haifa.il.ibm.com/projects/verification/sugar/tools.html

[17] NEOS Solvers: ttp://www-neos.mcs.anl.gov/neos/server-solvers.html

APPENDIX

A. *Examples without memory*

The type 'uint8' is an integer with range 0 to 255.

```
void simple (uint8 x, uint8 y)
{
    int temp, dat, ret;
    temp = (x – 110)^2;
    dat = (y – 42)^2 + temp;
    if (dat < 10000)
    {
        if (5*y > x)
            ret = x + y;
        else
            ret = x + y;
    }
    ....
    Assertion ➔ ret ≤ 255;
}
```

```
void space4 (uint8 x, uint8 y, uint8 z, uint8 t)
{
    int temp, ret;
    temp= (x – 40)^2 + (y – 28)^2 + (z – 170)^2;
    if(10000 > temp)
        temp= (x – 6)^2 - (y – 120)^2 + (t – 70)^2 + 28;
        if (2500 > temp)
            temp= x^2 + 3*y*z^2 – t^2*z – 292*t^3;
            if (temp > 0)
                temp= 6*y*x – 2*x^4 – z^3*x + 15*x^2*y^2;
                if(temp>0)
                    temp= x*z*t + y*t^2 – t^3;
                    if(temp>249)
                        ret= t*x*y^2 – 8*z^3;
                    else
                        ret = 0;
    ....    Assertion ➔ ret ≤ 0;
}
```

```
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;
}
```

```
void conditional (uint8 x, uint8 y)
{
    int temp, dat, ret;
    temp = (x – 40)^2;
    dat = (y – 42)^2 + temp;
    if (10000 > dat +2*x-y)
        temp= 6*y – 2*x – 4*(10000 – dat);
    else
        temp= x – 3*y +10000 – dat – 49;
    ret(x – y + temp);
    ....
    Assertion ➔ ret ≤ 1140;
}
```

B. *Examples with memory*