Computing, Information Systems, Development Informatics & Allied Research Journal
Vol. 7 No. 4, December, 2016 - www.cisdijournal.net

Loop Invariant: A Tool for Program Verification and Correctness

Ogheneovo, E.E.
Department of Computer Science
University of Port Harcourt, Nigeria
edward_oghenenovo@yahoo.com

R. B. Japheth
Dept. of Mathematics/Computer Science
Niger-Delta University
Yenagoa, Nigeria
jbunakiye@gmail.com

ABSTRACT

The development of correct program is a very important problem in computer science and software engineering. These days, most software are full of errors even though they seem to work well after testing. When these software are put into use, some errors which are formally passive becomes active and the result is the failure of such software. Therefore, there is need to develop error-free software these days because software have become inevitable in every area of human endeavor especially software used for safety and critical missions. There are many instances where software failure have occurred causing colossal loss of human and material resources. Therefore, it has become increasingly important to develop error-free due to their importance in society. This paper used the concept of loop invariant to prove the correctness of a program by using mathematical and algebraic proves. The technique used single loop **(while … loop)** as a testing case for this loop invariant. Using this method, it is possible to detect invalid as well as valid loop invariants since those programs that have invalid loop invariants often contain errors and produced wrong results.

Keywords: Loop invariant, software verification, program correctness, program specification, algorithms, flow diagrams

CISDI Journal Reference Format
Ogheneovo, E.E. & Japheth, R.B. (2016): Loop Invariant: A Tool for Program Verification and Correctness. Computing, Information Systems, Development Informatics & Allied Research Journal. Vol 7 No 4. Pp 1-8. Available online at www.cisdijournal.net

1. INTRODUCTION

Software systems are very important part of the modern society. They are used in almost every areas of human endeavor either directly or indirectly in our homes, offices, electrical appliances, automobiles, etc., as embedded software. Due to their importance, there is need to ensure that they are devoid of defects and faults that could result in errors and consequently failure which could result in colossal human and material losses. The process whereby software is checked or tested to ensure that it is free of errors is called program correctness. Program correctness is very important in software engineering, computer science, and the Information Technology industries [1] [2]. This is because these organizations increasingly depend on software or programs that functions very well and produce the expected results. Therefore, program correctness is very crucial for safety and security of lives and property [3].

Therefore, whether safety-critical or life-critical applications; it is very important to verify properties of programs before being used. Several techniques have been proposed in the past for verifying the correctness of software, especially mission-critical software. These include static analysis of programs (e.g. type checking, inference, static checking, etc.), model checking, and theorem proving techniques [4] [5]. Therefore, there is need to verify software critically before they are used to ensure their correctness because software verification and program correctness are very important concepts in program testing [6]. As a result, the success of software verification depends on the ability to find a correct technique [7] [8].

In this paper, we introduced the concept of loop invariant as a tool for verifying the correctness of software. Loop invariant is very important in program verification [9] [10]. Loop invariant is used to prove logically the correctness of a program from a precondition to a post-condition. As noted by Bi et al. [11], assertion used to determine the properties of loop program directly, or we can use it to prove the correctness of a program. The term loop is used to describe a programming language statement that allows a part of a program to be repeatedly executed a given number of times as long as the condition remains true. Invariant help a programmer to ensure that the each time a code reaches a certain point; it determines the correctness of the program and ensures that it does what it is expected to do. In this case, the loop condition is tested and if found true, the loop continues to iterate until the condition become false at which point the loop terminates [12] [13].

During the execution of a program, a method may be called [14]. The called method is used to describe construct used in the program segment which can be a precondition or postcondition. A precondition is a property that is true before the call is made while a postcondition is a property that is true after the call returns. A loop invariant should be true on entry into a loop and it must remain true after every iteration of the loop. Thus on exit from the loop, both the loop invariant and the loop termination conditions should be true. Loop invariants are crucial to understanding loops. They are so important and as such loops should be documented with the invariant used to prove their correctness. Therefore, invariant should be declared before the loop is written which must act as a guide to the development of the loop [15]. Thus loop invariant should be satisfied before entering the loop and at the top and bottom of each iteration. At the completion of each iteration, the loop-invariant and the negation of the loop-condition must be satisfied [16]. Hence, the statement of the algorithm comprising the loop must be chosen to satisfy them [17] [18].

Consider a **while … do** statement shown in figure 1(a). At the entry into the loop (i.e., at point A), the loop invariant is true before the loop begins and at point B just after every execution of the loop body until when the loop finally terminates. However, this is possible only if the code that tests the loop condition does not change the value of any variable appearing in the loop invariant. Thus what a **While loop** does is described by its loop invariant. Since a loop invariant is true every time through a loop, it means that a loop says what does not change; and by implication it says what the loop does not do. To show that a loop terminates, it is sufficient to provide a termination or variant function.

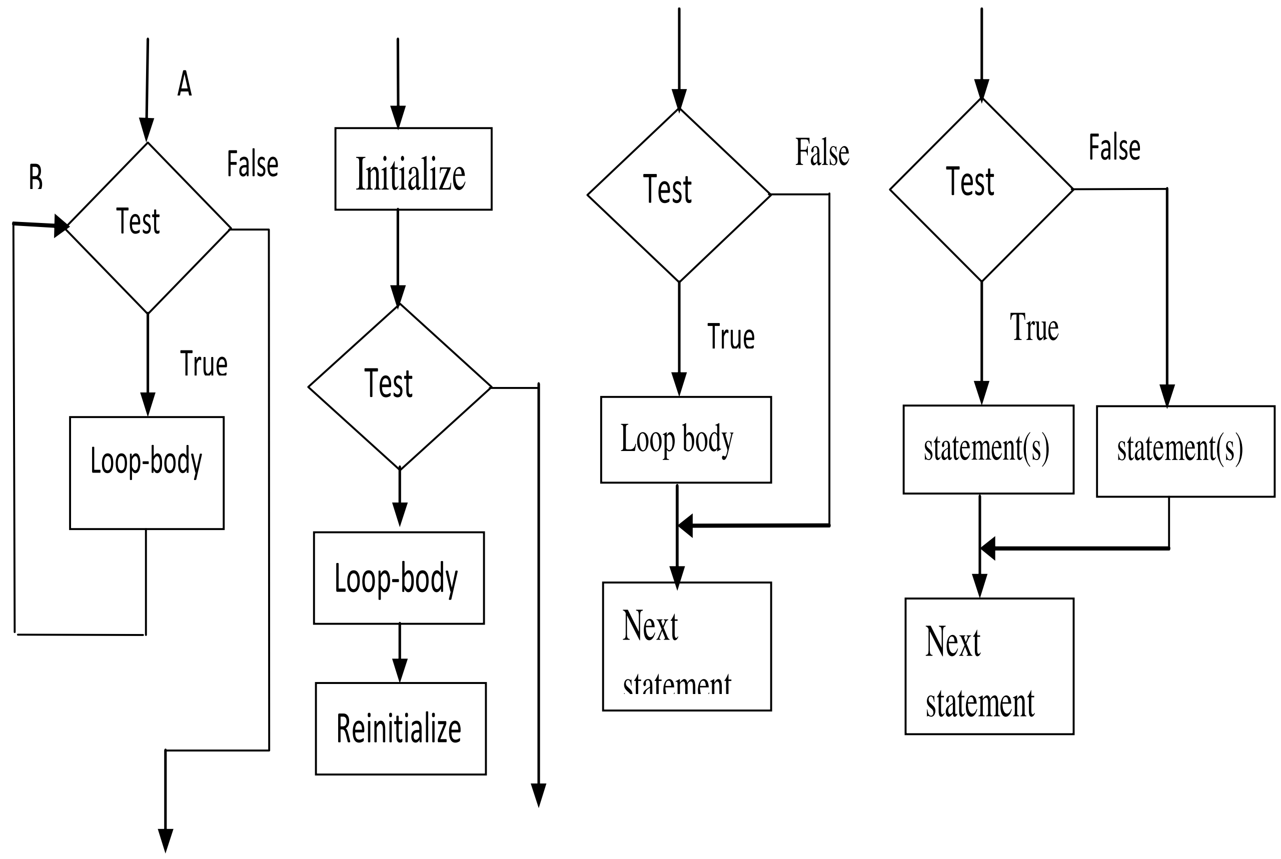

Fig. 1(a): While statement **Fig. 1(b):** For statement **Fig. 1(c):** If statement **Fig. 1(d):** If … then … Else

Figure 1(b) is an example of for loop. The **for loop** combines loop initialization, test condition, and increment into a single construct.

The syntax is:

```
for (int; text condition, increment)
{
  statement(s);
}
```

The **For Loop** starts by evaluating the initialization object; test the condition to see if true or false; if false, pull out of the loop; otherwise continue by executing the statement(s); evaluate the incremental expression, and then continue until the condition becomes false. Figure 1(c) shows an example of an **if … statement** or **If loop.** A simple if statements execute an action only if the test condition is true.

The syntax is as follows:

```
If (expression)
{
    statement(s);
}
```

Figure 1(d) shows the structure of **if … then … else statement.** The actions that an if … else statement specifies differ based on whether the test condition is true or false. The syntax for an **if … then … else** is:

```
If (expression) then
{
    statement (s)
}
else
{
   Statement (s)
}
```

This paper is mainly based on a simple loop condition called **while loop** and will not cover complex nested loops.

2. STATEMENT OF THE PROBLEM

Software systems have become an important part of the modern society. They are used in virtually every areas of human endeavor either directly or as embedded software in our homes, offices, electrical appliances, automobiles, etc. Due to their importance, there is need to ensure that they are devoid of defects and faults that could result in errors and consequently failure which could result in colossal human and material losses. Therefore, whether safety-critical or life-critical applications; it is very important to verify properties of programs before being used. Several techniques have been proposed in the past for verifying the correctness of software, especially mission-critical software. These include static analysis of programs (e.g. type checking, inference, static checking, etc.), model checking, and theorem proving techniques.

Therefore, there is need to verify software critically before they are used to ensure their correctness because software verification and program correctness are very important concepts in program testing [6]. As a result, the success of software verification depends on the ability to find a correct technique. In this paper, we introduced the concept of loop invariant as a tool for verifying the correctness of software. Loop invariant is very important in program verification. The technique used single loop **(while … loop)** as a testing case for our loop invariant. Using this technique, it is possible to detect invalid as well as valid loop invariants since those programs that have invalid contain errors and produced wrong results.

3. METHODOLOGY

Loop invariants can be stated as first or second order predicate calculus or in other similar forms since mathematicians find it easier to solve problems using this method and it also allow computers to follow the proof's correctness logically. Suppose we have a program fragment as shown in figure 2. We proved here that the program fragment is correct. After the code has been executed, the variables have the sum of the n elements of the array. There is also an assertion which shows that the code consists of some mathematical properties that holds from S to Z. we now show the correctness of each assertion using a **while loop**. The code fragment is as shown in figure 2.

```
int m, t;
int x(n);
m ← 0;
 t ← 0;
  while (m < n)
   {
      t ← t + x (m);
      m ← m + 1;
   }
```

Fig. 2: A code segment having a while loop

The flow chart in Figure 3 contains the input specification, output specifications, loop invariant and assertions needed to prove that the program fragment stated earlier is correct.

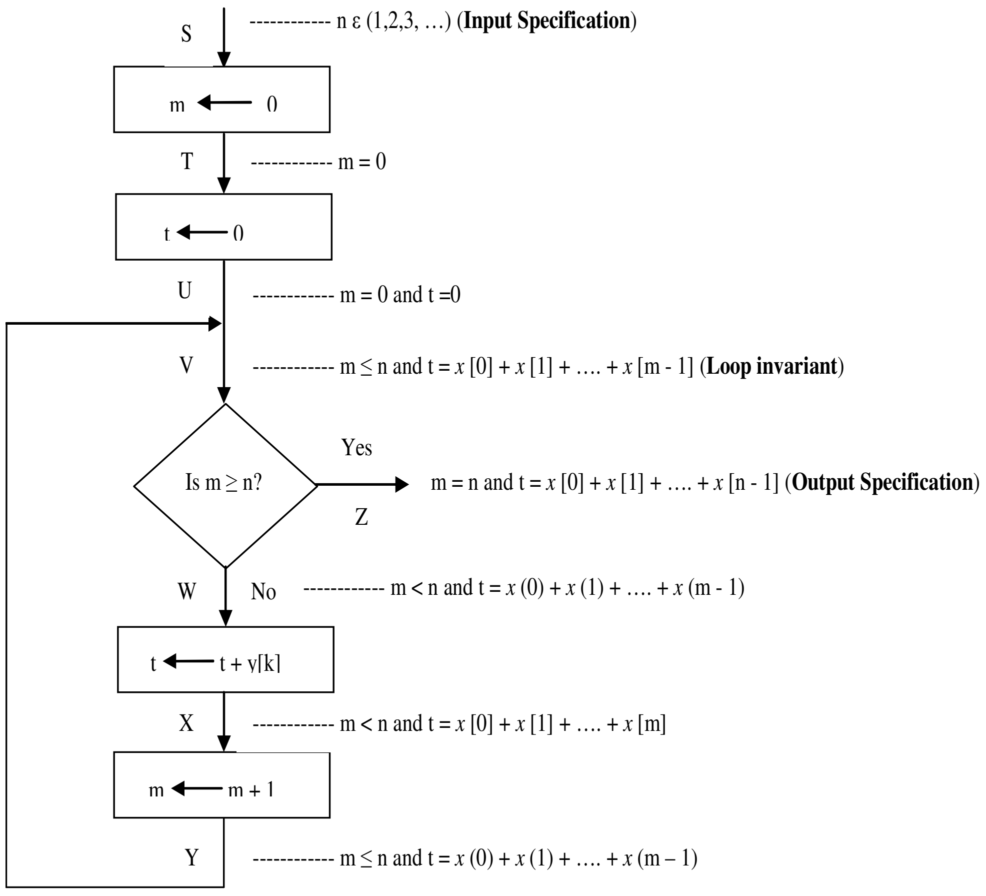
S
------------ n ε (1,2,3, …) (Input Specification)
m ← 0
T
------------ m = 0
t ← 0
U
------------ m = 0 and t =0
V
------------ m ≤ n and t = x [0] + x [1] + …. + x [m - 1] (Loop invariant)
Is m ≥ n?
Yes
m = n and t = x [0] + x [1] + …. + x [n - 1] (Output Specification)
Z
W
No
------------ m < n and t = x (0) + x (1) + …. + x (m - 1)
t ← t + y[k]
X
------------ m < n and t = x [0] + x [1] + …. + x [m]
m ← m + 1
Y
------------ m ≤ n and t = x (0) + x (1) + …. + x (m – 1)

Fig. 3: A flowchart of the program segment

In Figure 3, an assertion is placed before and after each statement labeled from letters S to Z; i.e., a proposition is made at each point that a certain mathematical property holds at that point. We then prove the correctness of each assertion as shown from the next line. At T, where we have the input specification, the condition that holds before the code is executed, is that the variable n is a positive integer. This is shown in equation 2.

$T : n \varepsilon \{1, 2, 3, . . .\}$ (2)

The output specification at this stage is that, if control reaches point Z, then the value of t contains the sum of the n values stored in array x, i.e.,

$Z: t = x (0) + x (1) + . . . + x (n – 1)$ (3)

Therefore, we can prove that the code fragment is correct with respect to correct output specification:

$Z: m = n \text{ and } t = x (0) + x (1) + . . . + x (n – 1)$ (4)

Also an invariant condition must be provided as part of the proof process to show that the prove is correct in addition to the input and output specifications. This is a mathematical equation that must hold at point V regardless of whether the loop has been executed zero, once, or many times. Thus the loop invariant that will be proven to hold is expressed as:

$V: m \leq n \text{ and } t = x (0) + x (1) + . . . + x (m – 1)$ (5)

Now it will be shown that if input specification (2) holds at point S, then output specification (4) will hold at point Z; that is, the code fragment will be proven to be correct. First, the assignment statement $m \leftarrow 0$ is executed. Control now is at point B, where the following assertion holds:

$T: m = 0$ (6)

To be precise, at point T, the assertion should read $m = 0$ and $n \varepsilon \{1, 2, 3, . . .\}$. However, the input specification (1) holds at all points in the flowchart. For brevity, the $n \varepsilon \{1, 2, 3, . . .\}$ therefore is omitted from now on.

At point U, as a consequence of the second assignment statement, $s \leftarrow 0$, the following assertion is true:

$U: m = 0 \text{ and } t = 0$ (7)

The loop has been entered at stage U, using the principle of mathematical induction we then show that the loop invariant (5) is actually correct. Prior to the loop is executed for the first time, assertion (7) holds; i.e., $m = 0$, and $t = 0$. Therefore, consider loop invariant (5), because $m = 0$ by assertion (6) and $n \geq 1$ from input specification (2), we then say that $m - n$ is needed to ascertain that the result is actually correct. Also, since $m = 0$, it then follows that $m – 1 = –1$, hence, the sum in (4) is empty since $t = 0$.

Therefore, loop invariant in equation (5) is true prior to entering the loop first time. Next, we then show the various steps involved in the inductive hypothesis. Suppose that at some point during the execution of the program fragment, the loop invariant holds, that is m equals to m_0 , then, $0 \leq m_0 \leq n$, that is, execution is at point V and the following assertion holds:

V: $m_0 \leq n$ and $t = x(0) + x(1) + \ldots + x(m_0 - 1)$ (8)

Next, control then passes to the conditional test box. That is, if $m_0 \geq n$, then because $m_0 \leq n$ by hypothesis, then $m_0 = n$. By inductive hypothesis (7), this implies that:

Z: $m_0 = n$ and $t = x(0) + x(1) + \ldots + x(n - 1)$ (9)

Thus the output specification (3) is Z: $m_0 = n$ and $t = x(0) + x(1) + \ldots + x(n - 1)$. However, if the test is $m_0 \geq n$ is false, then control passes from point V to point W. However, since m_0 is not greater than or equal to n, $m_0 < n$ and equation (8) changes to:

W: $m_0 < n$ and $s - x(0) + x(1) + \ldots + x(m_0 - 1)$ (10)

Thus the statement $s \leftarrow s + y(k_0)$ is executed, hence, from proposition at point X in (11), the following assertion holds:

X: $m_0 < n$ and $t - x(0) + x(1) + \ldots + x(m_0 - 1) + x(m_0)$

$= x(0) + x(1) + \ldots + x(m_0)$ (11)

Furthermore, the next statement to be executed is $m_0 \leftarrow m_0 + 1$. Let that the value of m_0 before executing this statement is one(3), then the last term in the sum in (10) is x (3). Now the value of m_0 is increased by 1 to 4. The sum s remains constant, so the last term in the sum still is x (3), which is now $x(m_0 - 1)$. Also, at point X, $m_0 < n$, increasing the value of m_0 by 1 means that if the inequality holds at point Y, then $m_0 \leq n$. Therefore, the effect of increasing m_0 by 1 is that the following assertion is true at point Y:

Y: $m_0 \leq n$ and $t = x(0) + x(1) + \ldots + x(m_0 - 1)$ (12)

Assertion (12) which holds at point Y is very similar to assertion (7) that, by assumption, holds at point V. However, point V is identical to point Y. That is, if (7) holds at V for $m = m_0$, then it also hold at V with $m = m_0 + 1$. Thus the loop invariant holds for m = 0 and by induction, we conclude that loop invariant (5) holds for all values of m. Therefore,

$0 \leq m \leq n$ holds for all values of n.

4. RESULTS AND DISCUSSION

From this proof, a set of algebraic equations are provided and it is proved that it is possible to discover invariants by the process of inductive mathematics and from execution traces using static method. This is done by tracing the variables defined in the program to ensure that they are actually used within the loops where they are defined by showing that the preconditions in the program actually implies the loop invariant. Thus we show that if I(m) and test condition U as shown in figure 3 are true, then I(m + 1) is true. Therefore, if U is also true, then it follows that I(n) is true for n = m + 1. Thus it shows that whenever the condition becomes false, the loop stops. Also, a proof is provided to show that if the loop iterates a maximum of N times, then I(n) implies the post condition. The main idea is to test a set of invariants using sufficient conditions without falsification.

Finally, we also proved that the loop terminates at a point. Initially, by assertion (7), the value of m is equal to 0. Each iteration of the loop increases the value of m by 1 when the statement $m \leftarrow m + 1$ is executed. However, m must approach n, during which time the loop is exited and the value of s is given by assertion (9), thus satisfying output specification (4). Given the input specification (1), we show that loop invariant (5) holds whether the loop has been executed zero, once, or more times. Furthermore, it was proven that after n iterations the loop terminates; and when it does, the values of m and t satisfy the output specification (4). That is, the code fragment of Figure 3 has been mathematically proven to be correct. However, as the complexity of the target equation increases, the methodology requires to determine the correctness of an algorithm becomes longer and more complex since it will require complex algebraic equations.

5. CONCLUSION

Loop invariant plays an important role in program verification. Invariant is the property that is true every time the code reaches a certain point. In case of loop invariant, the loop condition is tested and if found true, the loop continues to iterate until the condition become false at which point the loop terminates. During the execution of a program, a method may be called. What a method call does is described by its construct which can be a precondition or postcondition. A precondition is a property that is true before the call is made while a postcondition is a property that is true after the call returns. A loop invariant should be true on entry into a loop and is guaranteed to remain true after every iteration of the loop. On exit from the loop, both the loop invariant and the loop termination conditions should be true. Thus loop invariant should be satisfied before entering the loop and at the top and bottom of each iteration. This paper used the concept of loop invariant to prove the correctness of a program by using algebraic proves. The technique used single loop **(while … loop)** as a testing case for our loop invariant. Using this technique, it is possible to detect invalid as well as valid loop invariants since those programs that have invalid contain errors and produced wrong results.

In the future, we will find the relationship between loops and the result effects of each of the to see which loop is more ideal for a particular situation using a combination of static and dynamic analysis techniques

REFERENCES

1. Furia, C. A., Meyer, B., and Velder, S. Loop Invariants: Analysis, Classification, and Examples, ACM Computing Surveys. 2012 DOI 10.1145/0000000.0000000, http:doi.acm.org/10.1145/0000000.0000000.
2. Ball, T., Cook, B., Lahiri, S. K. and Zhang, L. Zapoto: Automatic Theorem Proving for Predicate Abstraction Refinement. In Computer Aided Verification, 16th Int'l Conference, CAV'04, Vol. 3114 of Lecture Notes in Computer Science, Springer, July, 2004, pp. 457-461.
3. Abramson, D. Foster, I., Michalakes, J., and Socic, R. Relative Debugging: a New Methodology for Debugging Scientific Applications, Comm. ACM. 1996; 39 (11): 69-77.
4. Rodriguez-Carbonell, and Kapour, D. Automatic Generation of Polynomial Loop Invariants: Algebraic Foundations. In Proceedings of ISSAC'04, July 4-7, 2004, Santander, Spain.
5. Mauborgne, L. and Rival, X. Trace Partitioning in Abstract Interpretation Based Static Analyzers. In 14th European Symposium on Programming (ESOP'05), 2005, Vol. 3444 of Lecture Notes in Computer Science, Springer, pp. 5-20.
6. Gupta, J. R. A Fresh Look at Optimizing Array Bound Checking. In Proceedings of SIGPLAN'90 Conf. on Programming Language Design and Implementation. 1990; 272-282.
7. Beyer, D., Henzinger, T. A., Majumdar, R., and Rybalchenko, A. Path Invariant, ACM, PLDI'07, June 11-13, 2007, San Diego, California, USA.
8. Bratko, I., Grobelnik, M. Inductive Learning Applied to Program Construction and Verification, Knowledge Oriented Software Design: Extended Paper from the IFIP TC 12 Workshop on Artificial Intelligence from the Information Processing Perspective (AIFIPP'92), 1993.
9. Chang, B.-Y. E. and Leino, K. R. M. Inferring Object Invariants. In Proceedings of the 1st Int'l Workshop on Abstract Interpretation of Object-Oriented Languages (AIOOL'05), Vol. 131 of Electronic Notes in Theoretical Computer Science Elsevier, January, 2005.
10. Hoare, C. A. R. The verifying compiler: A grand challenge for computing research. Journal of the ACM, 2003:50(1):63–69.
11. Bi, Z. Shen, M., and Tian, X. Automatic Generation of Non-Linear Loop Invariants, Journal of Computational Information System. 2010; 6(10): 3335-3344, http://www.joficis.com
12. Barnett, M. and Leino, K. R. M. (2005). Weakest Precondition of Unstructured Programs. In 6th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE'05), ACM.
13. Bouajjani A. et al. Invariant synthesis for programs manipulating lists with unbounded data. Proc. CAV, 2010.
14. Rustan, K. and Leino, M. Efficient Weakness Preconditions. Information Processing Letters, Vol. 93, No. 6, 2005, pp. 281-288.
15. Fabregat-Traver, D. and Bientinesi, P. Automatic Generation of Loop-Invariants for Matrix Operations. In Proceedings of ICCSA'11, IEEE Trans. 2011. http://www.computer.org/csdl/proceedings/iccsa/2011/4404/00/4404a082-abs.html
16. Sankaranarayanan, S., Spina, H. B. and Manna, Z.. Non-Linear Loop Invariant Generation Using CrÖbner Bases. In Proceedings of 31st ACM SIGPLAN_SIGACT Symposium on Principles of Programming Languages, 2004, pp. 318-329. ISBN: 1-58113-729-X. doi: 10.1145/964001.964028.
17. Galeotti, J. P., Furia, C. A., May, E., Fraser, G. and Zeller, A. Inferring Loop Invariants by Mutation, Dynamic Analysis, and Static Checking. IEEE Transactions on Software Engineering, Vol. 41, No. 10, 2015, pp. 1019-1037, IEEE Computer Society.
18. Eide, E. and Regehr, J. Volatile Are Miscompiled, and What to Do About It. In Proceedings of the 8th ACM and IEEE Int'l Conference on Embedded Software (EMSOFT), Atlanta, Georgia, USA, October 2008, pp. 1-10.