A Naïve Diamond Interpolation Algorithm for Transparent Evaluation of Non-Testable Program Expressions in Metamorphic Testing

Ch. Aruna[1], and R Siva Ram Prasad[2]

***Abstract*---**Ubiquitous software Applications depends on Non-Testable programs either directly or indirectly to achieve their computational tasks. Attribute relations based Metamorphic Testing emerged as a popular alternative to test the Non-Testable programs without using any test oracle. Our research analysis on metamorphic testing observed that, since beginning MT was suffering from evaluation of the complex Non-Testable program expressions to reveal hidden bug information. In traditional metamorphic testing strategy, it is even unable to find the erroneous part of complex expression which is more important for debugging and also caused to raise the exception. Best of my knowledge, none of the former scholars of MT was concerned on this serious issue, which effects on quality and scalability of metamorphic testing. In this paper we concentrated on above mentioned serious limitations of MT and proposed a Naïve Diamond Interpolation Algorithm for Transparent Evaluation of Non-Testable Program Expressions in Metamorphic Testing. This algorithm will be adopted as an integral part of metamorphic testing runtime environment to improvise the quality and scalability. Experimental results of this paper on various non-testable programs were compared against the traditional metamorphic testing. Results proven that the adoptions of Diamond Interpolation Algorithm dramatically increases quality and scalability statistics of metamorphic testing and also made the debugging job easier by identifying the erroneous part of non-testable program complex expression.

***Keywords*----**Metamorphic Testing, Metamorphic Relations, Non-Testable Programs, Diamond Interpolation Algorithm, Complex Expressions, Test Oracles

I. INTRODUCTION

Machine Learning, Graph Theory, Multi Precision Arithmetic, Bio-Informatics and some other utility applications were considered as Non-Testable programs [1] due to the unattainability of complex test oracles described by Weyuker. Test Oracle [2] plays a vital role in testing software, which is very complex to design and even not available while testing some non-testable programs. X Y Xie, Christian Murphey et al [2] stated that assuring the quality of more prevalent non-testable programs become very important as they are widely using in real life applications. Designing Test Oracles became the main obstacle to adopt the conventional testing strategy to test and assure the quality of non-testable programs.

Chittineni Aruna[2], Associate Professor, Department of CSE, KKR & KSR Institute of Technology and Sciences and Research Scholar, Acharya Nagarjuna University, Guntur, Andhra Pradesh, India.

R. Siva Ram Prasad[1], Research Director, Acharya Nagarjuna University, Guntur, Andhra Pradesh, India.

In the year 1998, T Y Chen at el [4] proposed, a proven attribute relations based testing strategy was described as Metamorphic Testing [MT] to alleviate the burden of test oracle design while testing non-testable programs. MT uses the exiting attribute relations to compare the perceived test result against expected result for the given input data (test cases). Successful test cases of MT were reused to produce the next generation (follow-up) test cases, to reduce the burden of generating new test cases and to reveal the hidden bug information with exhaustive concentration. Analysis phase of this research identified that MT was suffering from evaluation of the complex Non-Testable program expressions [5] to reveal hidden bug information, considered as a serious limitation of MT explained in problem statement (section 2). As non-testable programs need to deal with rigid operations, frequently they were composed with complex expressions [6 and 12].

This paper is designed to investigate the solutions for the critical problems of MT are: evaluating the complex expressions and generating custom level follow-up test cases for non-testable programs. In order to overcome these limitations in this paper we are introducing Naïve Diamond Interpolation Algorithm for Transparent Evaluation of Non-Testable Program Expressions in Metamorphic Testing. In our approach the complex expressions will explore to various levels as specified in the algorithm to feasible the evaluation process. Instead of validating the expression result at final, this algorithm validates the execution result at each level to identify the real sub-expression (expression construct) reason to generate the bug. We extended the purpose of this algorithm generate the follow-up test cases for successful input values (test sets) to deeply investigate the bugs (follow-up test case customization) at each expression construct level. This procedure will eliminate the ambiguity while creating the follow-up test cases for the successful test input data at run time of MT.This algorithm will be adopted as an integral part of metamorphic testing runtime environment to make MT more robust and scalable.

In addition to this we adopted this algorithm to Automated Metamorphic Testing Tool to analyze the progress of MT in terms of efficiency and scalability while testing non-testable programs. We used a wide set of non-testable programs, which contains complex expressions from various sources to conduct the expressions. Experimental results of this paper on various non-testable programs were compared against the traditional metamorphic testing. Results proven that the adoption of Diamond Interpolation Algorithm to conventional metamorphic testing tools/frameworks will dramatically increases quality and scalability statistics of testing and also

made the debugging job easier by identifying the erroneous sub expression part of non-testable program complex expression. Apart from that, this algorithm adoption increases the number of follow-up test cases and hidden bug count also.

II. Complex Expression testing problem in MT

In General we can classify the non-testable program expressions as simple expressions and complex expressions. Simple expression contains only numerical values as operands and operators to determine the respective operations. A complex expression contains functions, braces (parenthesis, curly brackets and square brackets etc.) and several simple expressions as constructs. Evaluation of a complex expression is very hard and subjective to execution infrastructure. Most of the non-testable programs like MPA, Machine Learning, Scientific Computing and graph theory contain the complex expressions to implement their operations. Determining the result accuracy of this complex expression becomes a challenging issue in software testing due to the unattainable test oracles to verify the results.

While collecting the complex expressions for this research, we considered a non-testable scientific expression posed by K R Ghazi and Vincent et al [7] as defined below:

(173746*Math.sin(1e22))+(Math.log(17.1*94228))-(Math.exp(0.42)*78487)

The above expression would be considered as a complex expression as it contains the functions and multiple constructs separated by brackets. Designing the test oracle for the above expression is too complex and having very less accuracy to validate the test results. Due to this reason testing non-testable applications with complex expressions become hard and applications remains as error-prone (with hidden bug), which cause to huge failures in future.

The correct result (expected) value of the above complex expression is $-1.341818958E^{-12}$. We evaluated the above expression with JAVA on JDK8 – 64 bit system for testing and obtained the result value as -267506.3929083699. As per IEEE 754 standards [8] this perceived value for the above expression is absolutely wrong due to overflow problem. We evaluated the same on C++ platform and obtained the result as same. I used another famous online calculation resource[9] implemented with java script to evaluate the above expression which had given the result as -124616.2792089638162917. After evaluation of this, we were in dilemma to assure that which value is correct for the above complex expression. In the same way, we evaluated the above expression with a few other environments and noticed that none of them we given unique values.

Under these circumstances, to identify the failures (incorrect results) and to determine the test result accuracy of non-testable programs sake, former researchers [3, 4 and 5] were extended metamorphic testing as an alternate for conventional testing. But the MT is suffering from the below problems while testing complex expressions of non-testable programs are:

- Increased complexity in Metamorphic relations mapping with expression constructs
- Undefined transparent evaluations strategy for complex expressions in conventional MT
- Intractability of the suspicious expression constructs, which causes to failures
- Follow-up test case generation difficulties due to unresolved complexity in expressions

These problems need to be addressed efficiently to make metamorphic testing more robust and scalable.

III. Naïve Diamond Interpolation Algorithm for Transparent Evaluation

Recently Metamorphic Testing (MT) becomes a prominent alternative to test non-testable programs by applying Metamorphic Relations (MR's) without test oracles. Our analysis and experiments over MT reveals some heuristic limitations about the complex expression evaluation as stated in section 2. Due to the above limitations MT is unable to find the hidden bugs, which is the main goal. The main reason for the above problems is inexperienced complex expression exploration algorithm in MT while testing non-testable programs.

To evaluate the complex expressions transparently and to overcome the above mentioned limitations, we introduced the Naïve Diamond Interpolation Algorithm (NDIA)in this research work. This algorithm works on split – process (test) – join formula to evaluate the complex expression and to identify the erroneous expression construct in a feasible manner. For any given complex expression $E$with interpreted values while testing at runtime, NDIA describes that as:

$$E = f_1(x)\ \varphi\ f_2(x)\ \varphi\ f_3(x) \ldots f_n(x) \quad where\ \forall\ [f\ \&\ \varphi] \in Math$$

Here f_1 to $f_n$are the simple functions of complex expression which are belongs to Math, φ stands for any Math operator and the static constant $x$is an operand with specified input value. As a part of preprocessing, initially NDIA checks the well-formed status of the given complex expression. Here the given expression will be validated by using the general expression formation standards of math. After the confirmation of the expression E is well-formed, NDIA uses the Dijkstra's Shunting-Yard algorithm[10] to transform the conventional infix expression to postfix notation. Rest of the algorithm will concentrates on implementing the split – process (test) – join formula to evaluate the complex expression to feasible the complex expression testing as described below:

Algorithm NDIA(diamond, E, ƛ,Θ)

Input:E is any well-formed expression and ƛ is the MR repository

Output: Θ is the Bug Repository

1. **Begin**
2. Set flag := true , level := 0, E' := null, diamond [0] [0] := E
3. dLen := diamond[level].length;
4. currPos := 0;
5. // split the complex Expression as diamond

```
6. for ( (currPos<dLen) && (flag is true) ) {
7. E' = diamond [level][currPos];
8. split E'as [ E(x1), E(x2) ... E(xn) ] where [∀ { E(xi) ∈ E'}⊆ E ]
9. level := level +1;
10. N:= E'. size ;
11. if (N>1) {
12.  for k=0to N {
13.  diamond [level][k] := diamondConstruct (diamond, level, k, E'[k]);
14. k := k +1;
15.  }
16. dLen := diamond[level+1].length;
17. currPos := 0;
18. }else { flag := false; }
19.  }
20. //  Apply MR to diamond leaves
21. targetLevel := 2*level -1;
22. for level to targetLevel {
23. i :=0;
24. forj := 0 to diamond[level].length {
25. E' = diamond [level][j];
26. Analyze and Map all possible MR's from ƛ to E'
27. Execute MT on E'
28.  If( !success ) {
29. add bug info to Θ ;
30.  break;
31.  }
32.  j++;
33.  }
34. Join childGroup of E[level-1][i] as [ E(x1) + E(x2) + ... + E(xn) ] where [ ∀ { E(xi) ∈ E[level-1][i] }⊆ E ]
35. level := level +1;
36. }
37. End
```

This algorithm expects the input and output arguments are *diamond, E, ƛ, Θ*. Here the input argument *diamond* is a two-dimensional dynamic array with auto incremental capacity, *E* is the complex input expression to test, *ƛ* is the repository of metamorphic relations for MT and output element *Θ* is the bug repository to store revealed bug information.

In this NDIA at the very beginning we assign the variables flag, level, *E', dLen, currPos* and diamond with respective base values. Pivotal variable *flag* is used to stop the iterations of splitting process while constructing the *diamond (with branches)* for the input expression. *E* is the given input expression and *E'* is the transformed expression to help in processing. Level is the variable to hold the current level of diamond while implementing split-join process. *dLen* specifies the length of diamond level means the number of constructs available at the level. *currPos* points one of the constructs (sub expression) at specified level and aids to traverse all constructs sequentially.

Split: The process of diamond construction for the given complex expression is implemented in the first for loop (from line6 to line 19). This loop iterates until completion of splitting the complex input expression to last level. First the loop gets an expression as input from the diamond and assigns to E' (*E' = diamond [level] [currPos]*). Later E' will be splitting to small constructs by using Dijkstra's Shunting-Yard algorithm [10]. At this movement if E' has more constructs, then each construct will be add to diamond with help of *diamondConstruct (diamond, level, k, E'[k])* sub algorithm or else the process of splitting will be stopped by assigning the false to flag variable. At the end of this loop the complex expression will be explored to least level and the shape looks like top-half part of the diamond.

Process: At this level each construct (sub expression) will be un-dividable and simple to execute. This splitting process helps us to alleviate the complexity in evaluation. Now we map the metamorphic relations form the repository ƛ to each construct in a transparent way to feasible the MT executions. Here the general MT [11] is used to test the expression constructs and to create the follow-up cases from successive test cases in an interpolation model over diamond data structure. Any suspected behavior with failures were observed while testing expression constructs, that construct information along with level value will be added to bug repository. This feature helps us to easily trace which simple construct of the complex expression cause to create the bug. Moreover designing the follow-up cases also become very easy for these small expressions constructs of complex expression.

Join: Join process of the splitted constructs will be starts immediately after the process of each construct MT evaluation at diamond middle level. This join process will continue up to reach of end point (target level) of diamond structure to construct the bottom half part of diamond. Initially we set the target level as as *target Level := 2*level -1* to stop the join process. At each level we will consider the sibling constructs to integrate them and to test with MT after integration of each two constructs to identify the bugs while integration. This way of interrogation will be executed for all possible combinations of expression constructs to reveal the critical hidden bugs. After each level of integration the next level will be considered as child group E[level-1] contains $E(x_1) + E(x_2) + \ldots + E(x_n)$] where [∀{ $E(x_i) \in E$[level-1][i] }⊆ E]. If any bug revealed at any level of joining or reached to the target level without any errors, cause to end the process and the bugs information will be displayed from repository.

IV. Case Study Implementation

To verify the scalability and implementation possibilities of our algorithm we used java programming language to code NDIA. This coded algorithm (in java) we executed on jdk-8 runtime environment (JRE), which is installed on Linux (Red Hat Enterprise Linux (x86_64)) OS with AMD – IA32 processor and 4 GB RAM. We collected 450 plus complex expressions from various non-testable real world and frequently used applications to conduct the experiments with NDIA. Mainly these expressions were extracted from test oracle unattainable domains like multi-precision arithmetic, machine learning, supervised learning, graph theory, dis-create mathematics, simulation/emulations. Along with this

we constructed a supervised metamorphic relations repository with proven relations of the above domain for mapping. While executing the complex expression constructs with NDIA, the respective metamorphic relations will be mapped for testing without test oracles.

A). Complex Expression Evaluation with NDIA

To describe our NDIA behavior at runtime, here we are presenting the evaluation of a complex expression from MPA domain, which was tested in our experiments, is:

$$[(\sin(60)*1932) + [(\cos(18.4)*7849) - \sin(1e22)*173746] + 83269*\log(14.8)]$$

After checking the well-form ness of this expression we had given this expression as input to our proposed NDIA. After receiving this expression initially NDIA explored this expression as diamond for execution with the intent of finding hidden bugs. Figure 1 represents the above expression respective diamond construction in detail.

The levels from L1 to L7 of the expression diamond were created while processing. The method eval(x) is used to execute the given expression/expression construct. This evaluation will starts from the last exploration (split) level (middle of the diamond n/2 for even no of levels or (n-1)/2 for odd). In this expression the evaluation started from Level-4. This evaluation is succeeded by metamorphic testing with respective metamorphic relations (test cases) to testthe evaluation results. If the test case results of MT is success, than those will be used to create the follow-up level test cases. If any failure occurred in MT results, suspects that presence of a bug. In this case, the evaluation process will be stopped and the whole relevant information will be saved in bug repository.

For this expression we evaluated **eval(sin (60)), eval(cos(18.4)), eval(sin(1e22)) and eval(log(14.8))**in a sequential manner. While executing the sin() function we used the given below proven metamorphic relations for testing:

i. $\sin(x) = \cos(pi/2-x)$
ii. $\sin(x) = 2\cos(x/2) * \sin(x/2)$
iii. $\sin(x) = (\text{sqrt}(1/2\ (1-\cos(2\ x))) * \text{sqrt}(x\text{^}2))/x$

For the given sin(60) all three expression were given the same value as -0.3048106211022 which are equal to the sin(60) original value, but the another function sin(1e22) returned the core expression value as -0.8522008497671. But the three specified metamorphic relations were returned the results as 0.523214785395, 0.8660254037844386 and 0.0297462374343378. These identity relations based comparison founds the mismatch and states that the existence of flaw. This procedure clearly specifying that till now the sin(1e22) is the erroneous part of the main expression with bugs. This target construct information helps and reduces the burden of debugging for software engineers.

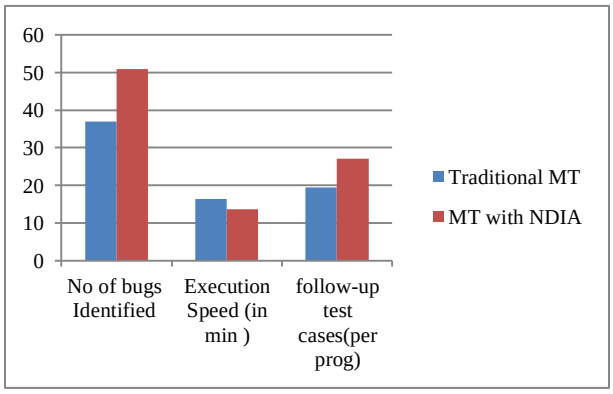
B) Traditional MT vs MT with NDIA

Here we discuss about the results comparison of traditional metamorphic testing and our proposed NDIA strategy. We executed the collected 450 complex expressions from various domains with both mechanisms and monitored the essential statistics are identified bug ratio, execution speed and number of generated test cases.

Because of the deep exploration technique of NDIA, our approach is having the enough capability to reach and verify the correctness at each small construct level for the given complex expression. Due to this reason MT with NDIA revealed more number of hidden bugs than traditional approach as shown in below table 1 and graph1.

TABLE I: TRADITIONAL MT VS MT WITH NDIA

	No of bugs Identified	**Execution Speed (in min)**	**follow-up test cases (per Prog)**
Traditional MT	**37**	**16.3**	**19.35**
MT with NDIA	**51**	**13.65**	**27.06**



Graph 1 Traditional MT vs MT with NDIA Result Graph

Our NDIA is having the better execution speed than traditional MT, because of NDIA tokenized the complex expression to simple tokens which takes the less time to evaluate than complex expression. Moreover we identified that maximum number of bugs were identified at the early state of expression evaluation than end of evaluation. Traditional MT testing will identify the bug after evaluation of the whole big expression with relations. Because of these reasons NDIA is having the better scalability (execution speed) than traditional MT.

Unlike traditional MT, our NDIA applies the metamorphic relations for simple tokens instead of complex expressions. Identifying the MR's for simple expression tokes is very easier than complex ones. Due to this reason our proposed MT with NDIA generates more test cases than traditional MT.

V. CONCLUSION AND FUTURE WORK

Designing a robust mechanism for complex expression evaluation in metamorphic testing become most considerable research area due to its importance in identifying hidden bugs. Former researchers were not explored the conventional MT testing strategies to deal with complex expressions of pervasive non-testable programs. In this research, we designed a Naïve Diamond Interpolation Algorithm for

Transparent Evaluation of Non-Testable Program Expressions in Metamorphic Testing. This algorithm will be adopted as an integral part of metamorphic testing runtime environment to improvise the quality and scalability. We proved that NDIA become the one stop solution for all the problems of complex expression evaluation in MT. Designed case study explained clearly the design and implementation overview of NDIA. Case study results proven that NDIA adoption to metamorphic testing made the debugging job easier by identifying the erroneous part of non-testable program complex expression.

REFERENCES

[1] E. J. Weyuker. On testing non-testable programs. Computer Journal, 25(4):465–470, November 1982. http://dx.doi.org/10.1093/comjnl/25.4.465

[2] L. Baresi and M. Young. Test oracles. Technical Report CIS-TR01 -02, Dept. of Computer and Information Science, Univ. of Oregon, 2001.

[3] X. Y. Xie, J. W. K. Ho, C. Murphy, G. Kaiser, B. W. Xu, and T. Y. Chen. Application of metamorphic testing to supervised classifiers. In Proceedings of the 9h International Conference on Quality Software (QSIC), pages 135–144, Jeju, Korea, 2009. CPS. http://dx.doi.org/10.1109/qsic.2009.26

[4] T. Y. Chen, S. C. Cheung, and S. Yiu. Metamorphic testing: a new approach for generating next test cases. Technical Report HKUST-CS98-01, Department of Computer Science, Hong Kong University of Science and Technology, 1998.

[5] ChAruna, Dr.R.Siva Ram Prasad "Metamorphic relations to improve the test accuracy of Multi Precision Arithmetic software applications" Advances in Computing, Communications and Informatics (ICACCI, 2014 International Conference on 24-27 Sept. 2014 by IEEE, Delhi, INDIA. pages 2244 – 2248

[6] C. Murphy, G. Kaiser, and M. Arias. An approach to software testing of machine learning applications. In Proc. Ofthe 19th international conference on software engineeringand knowledge engineering (SEKE), pages 167–172, 2007.

[7] K.R. Ghazi, V. Lefevre, P. Th ` eveny, and P. Zimmermann, "Why and how to use arbitrary precision", IEEE Computer Society 12, 5 (2010), DOI Bookmark: http://doi.ieeecomputersociety.org/10.1109/MCSE.2010.73.

[8] W. Kahan. 1997. *IEEE Standard 754 for Binary Floating-PointArithmetic*. http://www.cs.berkeley.edu/~wkahan/ieee754status /ieee754.ps, Berkeley, CA..

[9] http://web2.0calc.com/ and http://web2.0calc.com/formulary/math/trigonometry

[10] E. W Dijkstra" Dijkstra 's original description of the Shunting yard algorithm" November 1961 at ALGOL Bulletin. Bookmark: http://www.cs.utexas.edu/~EWD/MCReps/MR35.PDF

[11] Ch Aruna, Dr.R.Siva Ram Prasad Adopting Metamorphic Relations to verify Non-Testable Graph Theory Algorithms Published in ICACCE sponsored by IEEE on 1st and 2nd may 2015 and Dehradone pages 663, 669.

[12] P. Mell, V. Hu, R. Lippmann, J. Haines, and M. Zissman.An overview of issues in testing intrusion detection systems.Tech. Report NIST IR 7007, National Institute of Standardand Technology.

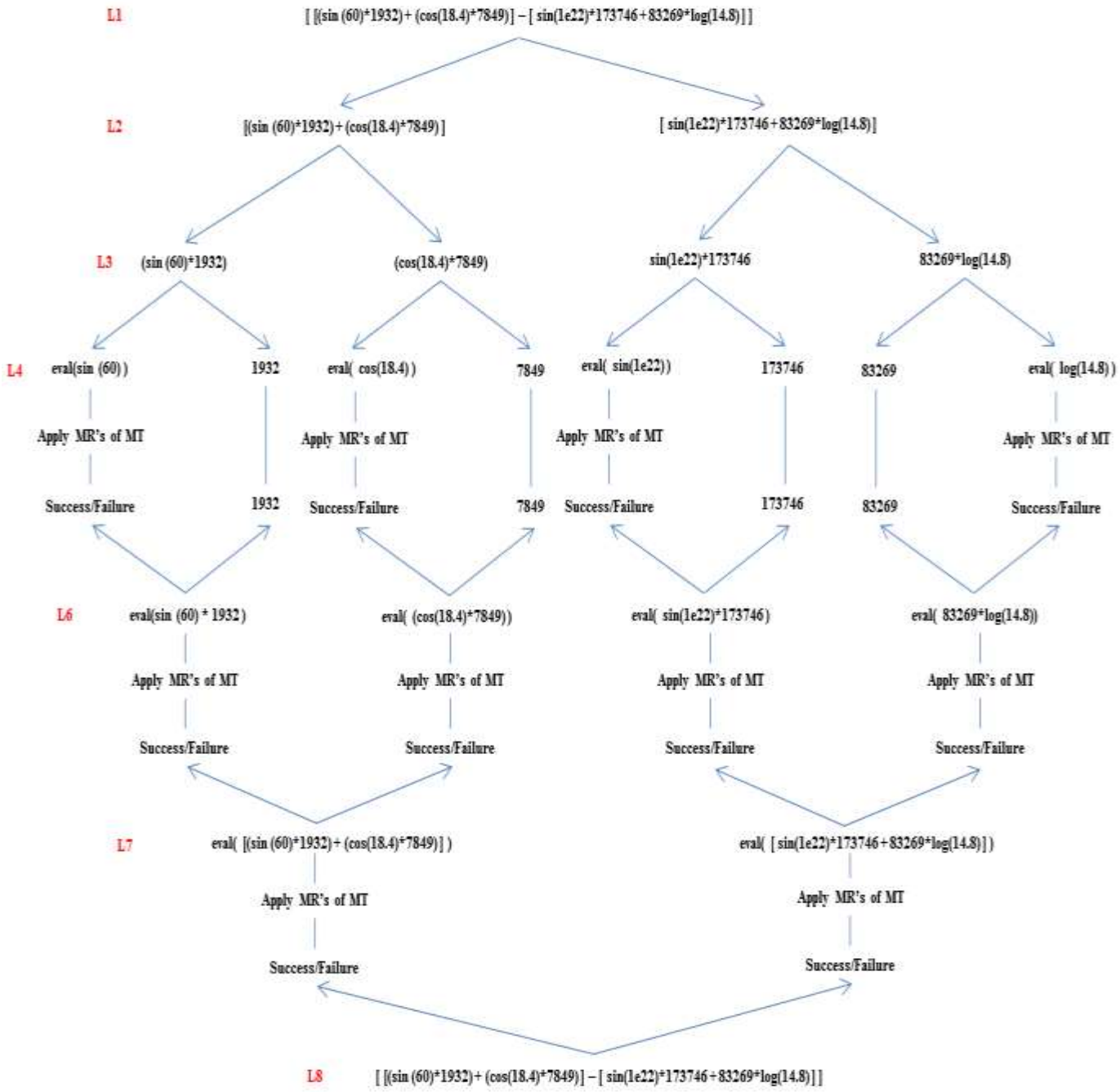


Fig. 1. Diamond Representation of complex expression for processing