International Journal of Engineering Research and Modern Education (IJERME)
ISSN (Online): 2455 - 4200
(www.rdmodernresearch.com) Volume I, Issue I, 2016

DETECTING ALL ACCEPTING CYCLES

M. Sureshkumar* & D. Vinotha**
* M.Tech Scholar, Department of Computer Science and Engineering, PRIST University, Thanjavur, Tamilnadu
** Assistant Professor, Department of Computer Science and Engineering, PRIST University, Thanjavur, Tamilnadu

Abstract:

Software testing is a defect preventive activity that presents many information navigation challenges for code description process. Brief description and short implementation of source code to generate code automatically emerging technology summarization for the code developers. Current packaging techniques to summarize key terms (lexical process) of those statements and information that process including concept first and then code. The proposed solution is to check the function process to identify the initialization, input and output process. In a recent paper, a method was proposed to accelerate the majority Linear Temporal Logic (LTL) of difference set low density parity check in programming logic. As computer system dependability has become increasingly important, there has been a significant amount of research in checking the programming methodology. The method detects whether a code has logical test case generations in the first iterations of majority Linear Temporal Logic (LTL). The propose method, I/O efficient approach for detecting errors is called Detecting All Accepting Cycles using State-of- the-Art algorithm. In this project, we generate a programming code that has predict an output. The proposed State-art algorithm is used to predict the logical error in the I/O accepting programming languages.

Index Terms: Linear Temporal Logic (LTL), State-of-the-Art Algorithm & Detecting All Accepting Cycles

1. Introduction:

The commonly used automata-based approach to LTL model checking reduces the problem of model checking to the problem of accepting cycle detection summarization. Many more-or-less successful reduction techniques have been introduced to fight the problem and to move the frontier of still tractable systems further. Nevertheless, for real-life industrial systems the reduction techniques are not efficient enough to make the verification tractable. A possible solution is to increase the computational resources available to the verification process. The two major approaches applied to increase the computational memory devices (disks).Regarding external memory devices, the main limiting factor of the approach is the amount of I/O operations an algorithm has to perform to complete its task. This is because the access to information stored on the external device is in order of magnitude slower than the access to information stored in the main memory. Thus, it became common to measure the efficiency of an I/O algorithm in the number of I/O operations such as random accesses, scans, and sorts. Thus, an algorithm working with implicit definition may save up to random access operations, which may have significant impact on the performance of the algorithm in practice .Nevertheless, the suggested reduction transforms the graph so that the size of the graph after the transformation is asymptotically quadratic with respect to the original size. As the I/O approach is meant to be applied first of all to large scale graphs, the quadratic increase in the size of the graph is significant and according to our experience often results in unsuccessful termination of the algorithm due to the lack of space. This is especially the case, if the model is valid and the graph has to be traversed completely to prove the absence of an accepting cycle. The approach is thus mainly useful for finding counterexamples in the case the standard verification tools fail to do so due to the lack of memory. The completeness of LTL model checking is, however, very important. A typical scenario is that if the system is invalid and the counterexample found, the system is corrected and the property verified again. In the end, the graph must be traversed completely anyway.

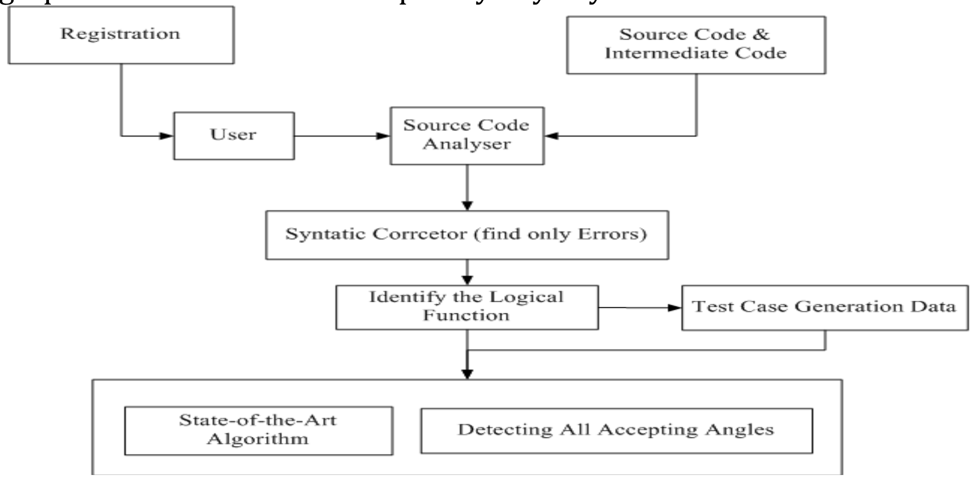

Registration
Source Code & Intermediate Code
User
Source Code Analyser
Syntatic Corrcetor (find only Errors)
Identify the Logical Function
Test Case Generation Data
State-of-the-Art Algorithm
Detecting All Accepting Angles

Figure 1: System Architecture

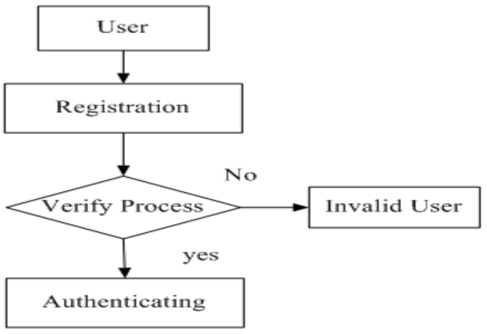
2. Detecting Accepting Cycles:

The long- term goal is to have the automated selection process choose the same keywords based on check sum eye-tracking algorithm that an algorithm process provides a summary of the code. The automated summarization could then be dedicated to the summary building phase based on the initialization and declaration. The propose method, An I/O efficient approach for spotting the test case generation using **state-of-the-art algorithm**. State-of-the-art algorithms to predict fault-detection significantly and reduces memory access time. The detection task is to retrieve code and variables that summarizes the lexical detection such as variable detection, index terms that are matched with the novel summarizes. The **state-of-the-art** error detector module has been designed in a way that is independent of the code size. Very less expensive compare to existing method. It has essential contributions to the good performance of our approach. Error detection in application is measured accurately.

3. Detecting Cycle Process Flow:

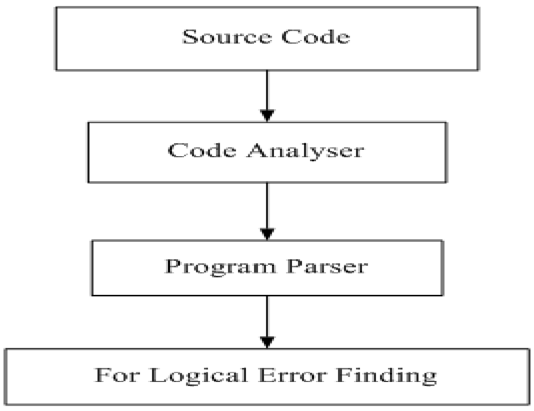
3.1. User Registration and Login:

In this module the client providing information for login, such as the user name and password is to register to detecting of all accepting cycles. Before the registration ensure whether providing user information which are similarity among the required details. The authorized user can login into the system and the unauthorized user cannot enter into the system. If the new user want to login means Compulsory registered. In the schedule process of information stored during registration is responsible for customers as per the service level agreements.



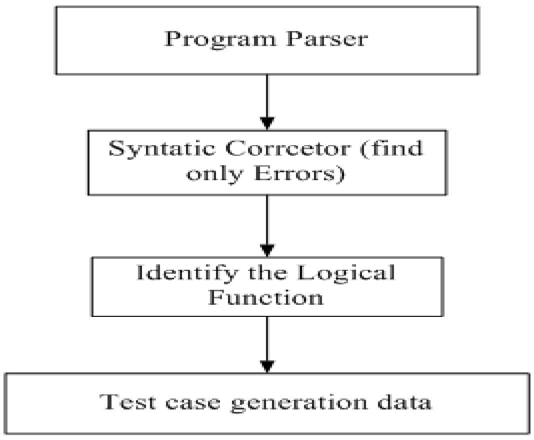
3.2 Program Parser:

Parsing or syntactic analysis is the process of natural language or in computer languages, conforming to the rules. For compiled languages syntax errors occur strictly at compile-time. A program will not compile until all syntax errors are corrected. For interpreted languages, however, not all syntax errors can be reliably detected until run-time, and it is not necessarily simple to differentiate a syntax error from a semantic error. In this module the source program is given as an input for find the logical error of the program.



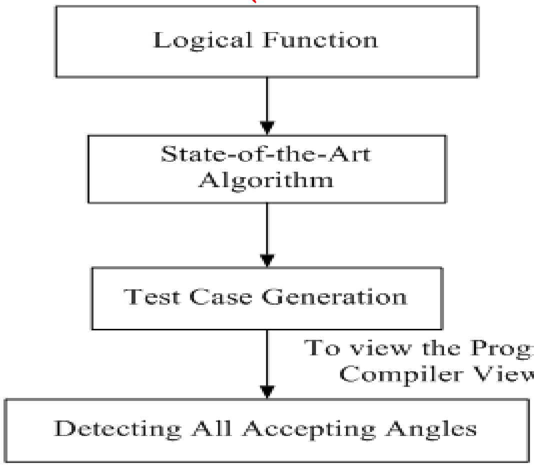
3.3 Function Finder in a Program:

Program submitted to a compiler often have errors of various kinds. So, good compiler should be able to detect as many errors as possible in various ways and also recover from them (i.e.) even in the presence of errors, the compiler should scan the program and try to compile all of it. However, no compiler can do true correction. Because, compiler won't know the intent of the programmer due to errors. Completely accurate error correction can be done only by the programmer. Simply the logical function has to store in the database. Based on the database and errors the program has compiled in the run time.


Program Parser
Syntatic Corrcetor (find only Errors)
Identify the Logical Function
Test case generation data

3.4 Test Case Generation Using State Art of Algorithm:

When the scanner or parser finds an error and cannot proceed, the compiler must modify the input. Function finder has to find the logical function of the program. So that the correct portions of the program can be pieced together and successfully processed in the syntax analysis phase. The problem overlapped by State-of-the-art algorithm. This technique does the job of error recovery not only from the compiler point of view but also from the programmers point based on the function finder. It generates code to be executed, which eases the programmer. Therefore, there should be a considerable forethought from the ultimate solution for the all accepting cycles


Logical Function
State-of-the-Art Algorithm
Test Case Generation
To view the Program from Compiler View Point
Detecting All Accepting Angles

4. A State of the Art Algorithm:

This steady state algorithm works as follows. Its parameters are the population size m, the number s of steady state individuals, the ratio σ/ℓ of the Gaussian mutation operator, and its mutation probability p_m.

1. Set the time (i.e., the generation number) $t := 0$.
2. Initialize the population $P(0)$ using random numbers.
3. Evaluate $P(0)$ using the given objective function f.
4. While the population has not converged and the maximum number of generations has not been reached, do
    1. Increase time, $t := t + 1$.
    2. Construct the new generation $P(t)$ from the old one $P(t-1)$:
        1. Construct $|P(t-1)| - s$ individuals using point in the middle crossover, where individuals are chosen using a linearly scaled roulette wheel.
        2. Mutate the population using the Gaussian mutation operator (cf. Section and the given σ/ℓ ratio using the mutation probability p_m.
        3. The new population $P(t)$ consists of the best s individuals from $P(t-1)$ and the $|P(t-1)| - s$ individuals just constructed.
    3. Evaluate $P(t)$ in parallel. Remember the best individual ever found.
    4. Repeat.
5. Finally return the best individual.

Typical values for the parameters of the algorithm are $m = 50$, $s = 5$, $\sigma/\ell = 0.1$, and $p_m = 0.2$.

The most notable difference to the classic algorithm is the use of a linearly scaled roulette wheel for selecting promising individuals. This ensures a consistent selection process during the whole optimization. In the selection process the right amount of support of individuals of high fitness, i.e., individuals likely to solve the given optimization problem, must be found. If their probability of proliferation is too close to the probability of individuals of low fitness, the population converges slowly or not at all. If the difference is too big, premature convergence to a local extreme is likely. The linearly scaled roulette wheel selection provides a good compromise.

5. Discussion and Future Work:

In the paper, the interesting problem for future research. The application of the proposed technique to memories that use scrubbing is also an interesting topic and was in fact the original motivation that led to new technique scheme to find the runtime errors and checking Software Requirement Specification (SRS).

6. Concluding Remarks:

Developers spend much of their time reading and browsing source code, raising new opportunities for summarization methods. Indeed, modern code editors provide code folding, which allows one to selectively hide blocks of code. On the other hand, the state-of-the-art function error detector module has been designed in a way that is independent of the code. The extension of this proof to the case of four errors would confirm the validity of the state-of-the-art approach for a more general case, something that has only been done through simulation.

7. Acknowledgements:

Part of this paper has regarded in [1]. This new version contains giant revision with new algorithm designs, detecting runtime errors and checking the Software Requirement Specification (SRS).

8. References:

1. M. Vardi and P. Wolper, "An automata-theoretic approach to automatic program verification," in Proc. Symp. Logic Comput. Sci., pp. 332–344, 1986.
2. J. Barnat, L. Brim, and P. Simecek, "I/O efficient accepting cycle detection," in Proc. Int. Conf. Comput. Aided Verification, pp. 281–293, 2007.
3. L. Brim, I. Cerna, P. Moravec, and J. Simsa, "Accepting predecessors are better than back edges in distributed LTL modelchecking," in Proc. Conf. Formal Methods in Comput.-Aided Des., pp. 352–366,2004.
4. S. Edelkamp, P. Sanders, and P. Simecek, "Semi-external LTL model checking," in Proc. Int. Conf. Comput. Aided Verification, pp. 530–542, 2008.
5. J. Barnat, L. Brim, P. Simecek, and M. Weber, "Revisiting resistance speeds up I/O-efficient LTL model checking," in Proc. 14th Int. Conf. Tools Algorithms. Construction Anal. Syst., pp. 48–62, 2008.
6. S. Edelkamp and S. Jabbar, "Large-scale directed model checking LTL," in Proc. 13th Int. Conf. Model Checking Softw., pp. 1–18, 2006.
7. U. Stern and D. Dill, "Using magnetic disk instead of main memory in the Mur' verifier," in Proc. Int. Conf. Comput. Aided Verification, pp. 172–183, 1992.
8. R. Korf, "Best-first frontier search with delayed duplicate detection," in Proc. 19th Nat. Conf. Artif. Intell., pp. 650–657,2004.
9. R. Korf and P. Schultze, "Large-scale parallel breadth-first search," in Proc. 20th Nat. Conf. Artif. Intell., pp. 1380–1385,2005.
10. K. Mehlhorn and U. Meyer, "External-memory breadth-first search with sublinear I/O," in Proc. 10th Annu. Eur. Symp. Algorithms, pp. 723–735, 2002.
11. J. Barnat, L. Brim, and J. Chaloupka, "Parallel breadth-first search LTL model-checking," in Proc. Conf. Autom. Softw. Eng., pp. 106–115, 2003.
12. R. Korf, "Linear-time disk-based implicit graph search," J. ACM, vol. 55, no. 6, pp. 1–40, 2008.
13. S. Leue and M. T. Befrouei, "Counterexample explanation by anomaly detection," in Proc. 19th Int. Conf. Model Checking Softw., pp. 24–42, 2012.