Automatically Fixing Syntax Errors Using the Levenshtein Distance

Leckraj Nagowah, Antish Jeetun, Suneil Jooty, Soulakshmee Nagowah

Computer Science & Engineering Department,
University of Mauritius, Réduit, Mauritius
l.nagowah@uom.ac.mu, {antish.jeetun, suneil.jooty} @umail.uom.ac.mu, s.ghurbhurrun@uom.ac.mu

Abstract: ***To ensure high quality software, much emphasis is laid on software testing. While a number of techniques and tools already exist to identify and locate syntax errors, it is still the duty of programmers to manually fix each of these uncovered syntax errors. In this paper we propose an approach to automate the task of fixing syntax errors by using existing compilers and the levenshtein distance between the identified bug and the possible fixes. The levenshtein distance is a measure of the similarity between two strings. A prototype, called ASBF, has also been built and a number of tests carried out which show that the technique works well in most cases. ASBF is able to automatically fix syntax errors in any erroneous source file and can also process several erroneous files in a source folder. The tests carried out also show that the technique can also be applied to multiple programming languages. Currently ASBF can automatically fix software bugs in the Java and the Python programming languages. The tool also has auto-learning capabilities where it can automatically learn from corrections made manually by a user. It can thereafter couple this learning process with the levenshtein distance to improve its software bug correction capabilities.***

Keywords: Automatically fixing syntax errors, bug fixing, auto-learn, levenshtein distance, Java, Python

(Article history: Received 16 September 2016 and accepted 9 December 2016)

1. Introduction

Three types of defects can occur in a computer program: syntax, semantic, and logic. While syntax errors are identified by a compiler or interpreter, semantic and logic defects must be identified by the programmer. Debugging is the process of removing defects from computer programs. If a computer program does not work according to its specification, programmers must debug the code and correct the defects [1]. A bug is an amalgam of one or more errors in the code (software errors), which may produce errors in execution (runtime faults), which in turn may produce failures in program behavior (runtime failures) [2, 3]. Debugging has been defined as "determining what runtime faults led to a runtime failure, determining what software errors were responsible for those runtime faults, and modifying the code to prevent the runtime faults from occurring" [2]. While a number of studies focuses on debugging and fixing runtime faults, novice programmers experience much difficulty with syntax errors as well [4]. They waste lots of productive time to manually go and check every line of code to fix syntax errors. Although these syntax errors are highlighted by current IDEs, they do not automatically fix these errors. Hence, the need for an automated program correction arises which will automatically fix all the identified syntax errors and which will eventually be beneficial to both programmers and companies.

The primary contribution of this paper is the introduction of the ASBF Tool, an Automated Software Bug Fixing tool that can fix simple syntax errors, can cater for several languages and can fix several files at one go and which also provides for automatic learning capabilities that help the system to improve its bug fixing process.

The rest of the paper is structured as follows: Section 2 gives an overview of some existing automatic bug fixing techniques and tools. Section 3 presents a critical analysis of these existing tools and techniques and presents some ongoing works in the field. Some design issues for an ideal automated software bug fixing tool are highlighted in Section 4. Section 5 gives the design of ASBF. Details about the implementation and testing of a prototype based on ASBF are given in Section 6. Section 7 presents some discussions on and evaluations of ASBF and finally Section 8 concludes the paper and identifies some venues for further improvement.

2. Literature Review

This section provides an overview of some techniques used for automated bug fixing, existing automatic bug fixing tools, and also presents some related works in the automated software bug fixing field.

A number of techniques already exist to automate the software bug fixing process, the most popular being Genetic Programming, Search Based Software Testing and Static Analysis Engine.

Genetic programming [5] is a systematic method for getting computers to automatically solve a problem starting from a high-level statement of what needs to be done. Genetic programming is a domain-independent method that genetically keeps a population of computer programs to solve a problem. Genetic programming iteratively transforms a population of computer programs into a new generation of programs by applying analogs of naturally occurring genetic operations.

Search-Based Software Testing [6] is the use of a meta-heuristic optimizing search technique, such as a Genetic Algorithm, to automate testing. The key to the optimization process is a problem-specific fitness function. The role of the fitness function is to guide the search to good solutions from a potentially infinite search space, within a practical time limit.

Static analysis [7] also known as static code analysis is a method to locate and fix bugs in computer programs. Program understanding or program comprehension is the process of examining the codes by visual inspection alone (without using any tools) which is quite cumbersome. Automated tools have been designed to use static analysis and can therefore assist programmers/developers in carrying out static analysis.

A number of automatic software bug fixing software already exist that are based on the above mentioned techniques.

KeshMesh [8], devised by Mohsen Vakilian, is a static analysis tool which works by automatically fixing concurrency bug patterns in Java and provides the automation process through FindBugs which acts as its user interface and complex analysis engine that can predict bug patterns where multiple methods and object classes are involved. The Library for Analysis is based on WALA, a static analysis engine.

Starting from the bug reports such as CTrigger being a bug reporting tool, AFix [9] analyses the bugs with static analysis to construct a suitable patch for each bug report. It further tries to combine the patches of multiple bugs for better performance and code readability. AFix's run-time component provides testing customized for each patch. AFix can save developers' manual bug fixing effort by automatically generating patches or patch candidates for concurrency bugs detected during in-house testing or for concurrency failures discovered during production runs. Some key features of this tool are: ability to work on any bug reporting tool to get results, based on static analysis for fast performance and makes use of patch to eliminate real world bugs.

Autofix [10] works on the Eiffel classes along with contracts. Contracts are made up of class specifications and consist of preconditions, post conditions, intermediate assertions and class invariants. The contracts provide certain criterion to determine the correctness of a routine. Testing the routine with varying inputs reveals errors in the form of an assertion. Autofixgenerates as many calls as possible in the available testing time to find faults in the production software. The tool can find several fixes for a given fault and ranks the valid fixes according to a simple metric which combines dynamic and static information.

PACHIKA [11] uses object models from program executions and determines differences between pass and failed runs. It further generates possible fixes and assesses them via a regression test suite. The test suite is run to prevent the risk of introducing new problems. Fixes which successfully passes the test suite run are then presented to the programmer.

AppPerfect [12] is a static Java code analysis program that has been mainly design to enforce good Java coding practice and to automate the debugging process of Java codes. By applying over 750 Java coding rules gathered by leading experts in the Java field, several violations can be scanned,analyzed by AppPerfect and theproposed solutions that have been applied can be viewed.

The "FIXATION Tool" [13] is an approach being developed by Gu, that automatically detects bad fixes for Java Programs. Upon detection of errors, it returns an input that still triggers the bug. When a coverage failure is detected, it outputs a counter example that triggered the bug in the fixed program. Programmers may then use those counter examples to further reformulate their desired type of fix to be applied.

GenProg [14] is an automatic, scalable, competitive bug repair tool which uses genetic programming to locate errors. Fitness evaluations can run in parallel and the developers are evaluating GenProg on several bug sets and the different benchmarks outcomes the tool produces. The tool still has many kinds of defects such as infinite loops and segmentation

violations. Through the use of genetic programming, the tool does not require formal specifications to locate errors but instead, it consists of a database where previously known errors are stored and is constantly updated with newer ones.

Test obsolescence is one of the most known reasons to test-suite evolution. Pinto and Sinha [15] state that repairing test cases can be time consuming for large scale projects and to further address this issue, investigations are being carried out to optimize the automated testing techniques where the study of a new tool called "TESTEvol" is being done. The tool aims at providing a systematic study of test-suites evolution where several aspects of the test repair techniques are being analyzed.

Parnin and Orso [16] describe that research carried out in the developing automated techniques still lack behind in their practical effectiveness. Their goal is to gain insight on how to build a better debugging tool and to identify promising research directions in the area. To further advance the state of art in this area, Parnin research still aims towards more promising directions that take into account how programmers actually debug in real scenarios.

The "WhyLine Tool" [17] focuses on assisting novice users with formulating hypotheses and asking questions about a program behavior. Research is still being carried out concerning the hypotheses phase. The Tarantula technique is being used to search for errors based on statistical ranking and the development of an Eclipse Plugin helped the developers to better understand faulty statements rankings.

In [18], the authors used Recurrent Neural Networks (RNNs) to find repairs for syntax errors in student programs. They used a number of syntactically correct student submissions to train a RNN for learning a token sequence model for all valid token sequences that is specific to the problem. The trained model is then used to predict token sequences for finding repairs for student submissions with syntax errors and fix them by replacing or inserting the predicted token sequence at the location of the syntax error. The main limitation of this technique is that it currently can handle and fix only one syntax error in a program.

3. Analysis

This section depicts a critical comparison between the existing tools and techniques highlighted above. The focus is mainly on KeshMesh and AppPerfect and these tools have been assessed based on the criteria described below:

Availability of set of coding rules for solving errors: The set of coding rules defined for some of the tools will be rated. Coding rules are mainly the codes that the tool proposes upon encountering an error.

KeshMesh: KeshMesh is based only on its predefined rules and cannot solve errors apart from those rules specified.
AppPerfect: Based on 750 coding rules, it can solve mostly all types of errors and provide their fixes.

Ability to solve syntax errors: This criterion checks whether the tool can automate the debugging process.

KeshMesh: It cannot solve syntax errors due to its restricted set of rules.

AppPerfect: It can solve most syntax errors due to its enormous set of rules acquired.

Ability to underline problematic areas: The criterion will be rated upon if codes are being highlighted enough to be visible to the user on the spot.

KeshMesh: This software does underline problematic areas and displays where the errors occurred.

AppPerfect: AppPerfect can display the list of errors in highlighted form.

Provision of feedback of why an error has been corrected in a particular way: The output of how the code has been automatically corrected shown in the tool will be assessed.

KeshMesh: No feedback is given on why the error has been corrected in this way. User has no idea of how the program is auto fixing the bugs.

AppPerfect: AppPerfect does not provide explanations on why a particular error has been corrected in a specific way.

Ability to optimize codes: This section checks whether the tool can make the code more resource efficient through optimization.

KeshMesh: KeshMesh cannot optimize the codes further to make it more efficient due its limitation of rules available.

AppPerfect: Optimization can be made by applying several rules to the error and check which one is more appropriate.

The analysis clearly demonstrates that AppPerfect is most suitable to perform automatic software bug fixing but still relies on the limited number of coding rules. The analysis also helped to identify a list of features that an automated software bug fixing tool should have. The following ideal features have been identified:

- Ability to solve a wide range of syntax errors
- Underlining of problematic area in the program
- Provide proposals in case errors have/have not been solved
- Give feedback of why a fix has been applied
- Choosing the best possible fix for an error

The techniques described in the previous section have then been assessed based on the above criteria.

Ability to solve a wide range of syntax errors:

Genetic Programming: It is possible to solve a wide range of syntax errors using genetic programming as programs in genetic programming are represented as syntax trees rather than lines of code thus easing the process of solving syntax errors.

Search-Based: It is more unlikely to solve syntax errors as search based is oriented towards fixing logical errors by applying different test methods to see if their outcomes yields the same result.

Static Analysis: Static analysis only solves logical errors though different path testing and is unable to perform syntax checks.

Underlining of problematic area in the program:

Genetic Programming: In Genetic Programming, the program is represented as a syntax tree and not in terms of lines of codes. It is quite difficult to underline the lines where errors have occurred.

Search-Based: Underlining of problematic cannot be performed through Search based technique as it only checks whether the output of the code is correct.

Static Analysis: Underlining of error is not performed in static analysis.

Proposals in case errors have/have not been solved:

Genetic programming: Data mining and genetic programming can be combined together and in case solutions for errors have not been found, the program may therefore consult its database and provide some proposals of how to solve the problem.

Search-Based: Search based technique does not provide proposals if ever no solutions are found. It only tries already predefined methods to test for different outcomes.

Static Analysis: To our understanding, static analysis cannot provide for proposals. It aims only at analyzing logical errors and fixing them.

Provision of feedback of why a fix has been suggested:

Genetic Programming: Genetic Programming may incorporate machine learning algorithms. Proper feedback may be provided by those algorithms.

Search-Based: Feedback is not generated on upon how the error has been fixed. It is up to the programmer to check on how the error has been solved.

Static Analysis: Static analysis does not generate feedbacks as it only tests for logical errors and tries to fix it by applying several path tests.

Choosing the best possible fix for an error:

Genetic Programming: Genetic Programming swaps lines of codes until a solution has been found. It then removes the unnecessary lines of code which can be viewed as an optimization made to the code. Finally the resultant code is one which is error free and optimized.

Search-Based: Search Based chooses the methods to test randomly, so the probability of having the best possible fix is low.

Static Analysis: Static analysis can choose the best fix as it tests for different paths through the lines of codes to see which one gives the best outcome.

4. Design Issues

From a thorough analysis of existing tools and techniques, a set of design issues for an automated software bug fixing tool are presented in this section. It complements the set of ideal features identified in the previous section. An ideal automated software bug fixing tool shall:

- Accept a program (a single file) as input or a source folder,
- Open and display the file(s) on the user interface,
- Scan the file(s) for syntax errors,
- Highlight error(s) found in the file(s),
- Provide fixes to the identified errors,
- Highlight the applied fixes,
- Present a well detailed diagnosis of the errors found in the file(s) and their solution(s). This shall include line numbers, errors, solution and percentage highlighting the accuracy of the fix provided,
- Allow the user to make changes to the corrected version of the file(s),
- Capture the changes made by a user, in a file corrected by the system, to enable the system to learn and consequently propose better fixes,

- Be compatibility with both compiled and interpreted languages and hence support multiple languages,
- Be easy to use and user friendly.

After identifying the above recommended features of an automated software bug fixing tool, a further investigation on how IDEs provide possible fixes for syntax errors, has been carried out. The Eclipse IDE [19] was investigated. It was observed that the internal compiler of the IDE discovers the errors and the QuickFix component then tries to propose solutions for a bug. However, the fixing of these bugs has to be done manually. Based on the compiler results consisting of line numbers, type of error, expected message, start and end position of the error returned for the whole set of code, the QuickFix component, relying on these parameters, provide the user with a list of solutions for a specific error in terms of markers. The problematic text in the editor is automatically underlined whenever there is a syntax error. Syntax errors and the use of undeclared variables are some examples of the type of errors that are more likely to be discovered by an IDE.

A further assessment of the use of the inbuilt compiler and the QuickFix component against the list of ideal features identified in the previous section has been carried out.

It is possible to solve syntax errors using the inbuilt compiler of IDEs and retrieving essential information to perform bug correction. When an error has been found, the compiler returns the start and end positions of the error, thus highlighting of erroneous codes can easily be done. It may be possible to generate proposals to the user in case the compiler has not been able to provide the most appropriate fix for specific error by consulting a database of keywords. The database may also contain information about types of errors, fixes for the errors and the reason why to choose this specific fix. The best possible fix may be returned after a comparison between the syntax error and similar keywords retrieved from the database.

5. Design

Our Automated Software Bug Fixing Tool, ASBF, is based upon the design issues identified above. ASBF uses the inbuilt compilers, together with the levenshtein distance between a syntax error and the corresponding fixes, which are mainly reserved words of the language, to automatically fix erroneous files. This section presents mainly the main components of ASBF, the algorithms used and the database of ASBF.

It comprises of three distinct layers namely: GUI Layer, Bug Fixing Layer and Database Layer. Figure 1 below shows the component diagram of ASBF and depicts the interaction between the three core layers when a file is being automatically fixed. Each layer consists of a number of components vital to the correct operation of the ASBF Tool.

5.1 Graphical User Interface

This component provides a user friendly interface to the user with the possibility to navigate through various files, to manipulate the various components on the interface, to give feedback to the user.

5.2 Bug Fixing Layer

This layer consists of a number of components as highlighted below:

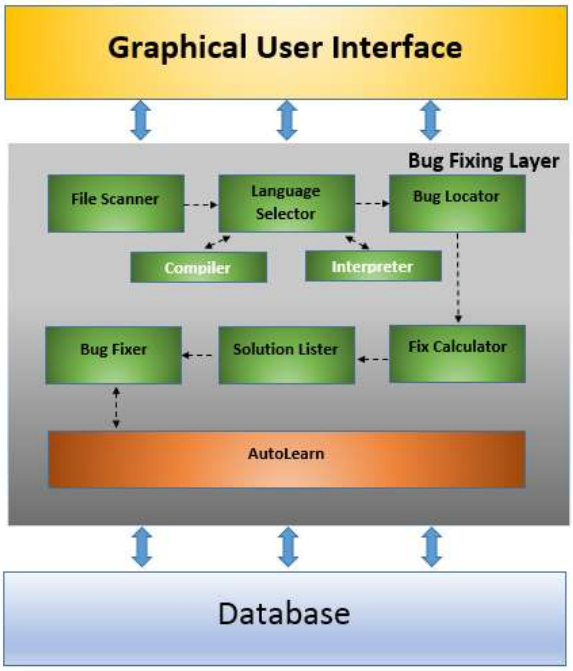

Figure 1:ASBF Component Diagram

File Scanner: This component takes as input the path of the erroneous file or folder from user. The file(s) are then loaded into the system and then passed to the compiler for further processing.

Language Selector: This component automatically identifies the programming language and calls the appropriate Compiler or Interpreter for that specific language.

Compiler: The compiler scans a file at one go and identifies all the errors in the file. It returns a well detailed diagnostic such as line numbers where errors have been identified, type of the errors.

Interpreter: This interpreter takes a source file as input and scans the file line by line. It identifies

errors on the line and reports the errors. Only when the reported errors have been corrected, only then it will move to the next line.

Bug Locator:The task of the bug locator is to capture all errors returned by the compiler where each of them will be analyzed in greater details at the next stage. An example of errors captured for a Java file by the IDE compiler returns the details as in Figure 2.

Fix Calculator: For each of the errors returned by the compiler or interpreter, the Fix Calculator uses the Levenshtein distance, which is a measure of the similarity between 2 strings, and return their percentage difference in order to return a possible fix using the database of keywords for that language. Figure 3 outlines the pseudo code for the Fix Calculator which uses the Percentage_Diff_Calc as shown in Figure 4.

Code	compiler.err.cant.resolve.location
Kind	ERROR
Line Number	8
End Position	102
Start Position	99

Figure 2:Compiler Errors Captured

```
Function: Fix Calculator
Pass in: list of error details
Set most appropriate percentage to 100
Set most appropriate solution to null
Set list of fixes to null
FOR each syntax error in Diagnostic
  IF autolearn tables has fixes
    Get best fix
  ELSE
Call to Percentage_Diff_Calc
    Add to list of fixes
  END IF
END FOR
Pass out: list of fixes
End function
```

Figure 3: Fix Calculator Pseudocode

```
Function: Percentage_Diff_Calc
Pass in: syntax error
FOR each keyword available
FOR each character in error
    Compare with keyword for that
      language
    Identify best fix
END FOR
END FOR
Pass out: best fix
Endfunction
```

Figure 4: Percentage_Diff_Calc Pseudocode implementing the levenshtein distance

Figure 5 shows an output of Fix Calculator returning the best fix (char) for a syntax error (cha) with the least percentage difference (25%) compared to other possible fixes.

Line Number	8
Errors	cha
Solutions	char
Percentage Difference (%)	25

Figure 5: Best Fix for a syntax error

Solution Lister:The SolutionLister gets all the fixes proposed by the fix calculator and stores them in a collection to be used by the bug fixer at a later stage.

Bug Fixer: Using the table of fixes and for each of the syntax errors, the Bug Fixer replaces the bugs in the file by the fix proposed by the fix calculator and writes the whole source code to a new file where the user can compile it and run.

Autolearn: If the user does not accept a specific or multiple fixes proposed by the system and makes correction to a file already fixed by ASBF, the AutoLearn component is activated. It checks the appropriate keywords table to see if the change proposed by user is present in the table. If not, it inserts the corrected keyword into the keywords table.If the changes made by user already exist in keywords table, the component then accesses the appropriate autolearn table and records the error and changes made by the user, i.e. the initial error and fix proposed. In the future, if ASBF encounters the same error again, it initially checks the autolearn table to locate any present fix. If a fix already exists, it is used; else a call is made to the levenshtein distance as in Figure 3. Figure 6 shows the pseudocode for the Autolearn component.

```
Function: Autolearn
Pass in: syntax errors and corrected
   keywords
FOR each syntax error
   IF corrected keyword does not
      exist in keywords table
      Add it in keywords table
 ELSE
      Add syntax error and corrected
      keyword in autolearn table
END IF
END FOR
```

Figure 6: Autolearn Pseudocode

5.3 Database

The database layer consists of the necessary tables that shall make ASBF perform automated software bug fixing. The database stores all the reserved keywords for the languages that can be supported by ASBF. For each syntax errors identified, a reference is made to fetch the most appropriate solution for that specific language. It also consists of an autolearntable which records the initial error identified by ASBF and the user proposed fix.

6. Implementation and testing

A prototype of ASBF has been implemented to show that the principle of having an automated software bug fixing tool relying on an inbuilt compiler and the levenshtein distance is feasible. The following development tools have used to develop the ASBF prototype: Eclipse Java EE IDE for Web Developers, Java™ SE Development Kit 7, XAMPP and GWT Designer. As proof of concept, the prototype currently caters forthe implementation of a compiled language (Java) and an interpreted language (Python).

To test the ASBF Tool, a number of test scenarios have been considered:

- Testing a Java file containing syntax errors
- Testing a Python program containing syntax error
- Testing a Java project folder with several erroneous files
- Testing the Autolearn capabilities of ASBF

Figure 7 shows the results for the first test case. When the erroneous Java file is imported into the ASBF, the latter identifies and highlights the lines containing syntax errors. ASBF then uses the Java compiler and the levenshtein distance to fix the syntax errors. The corrected file is then displayed with the lines containing the corrections highlighted. ASBF also displays a table showing the percentage error between the identified syntax error and the proposed fix.

Similarly, Figure 8 shows the results of fixing a Python program having syntax errors. When the erroneous Python file is imported into the ASBF, the latter identifies and highlights the lines containing syntax errors. ASBF then uses the Python Interpreter compiler and the levenshtein distance to fix the syntax errors. The corrected file is then displayed with the lines containing the corrections highlighted. ASBF also displays a table showing the percentage error between the identified syntax error and the proposed fix.

The next test scenario aims at testing a Java source folder with a number of syntax errors in the different Java files, with the aim of fixing all the files in the folder. The project folder is called “Error Analyze” and has three erroneous files namely: Book, Booklevel1 and Student. In this process, a temporary table, visible only for the current fix, is used where ASBF captures the class names of all the files and populates them in the table. If the user wrongly typed a class name, ASBF automatically detects that error and during the fixing process, it uses the class names from the temporary table to provide the closest match possible. The temporary table is automatically dropped at the end of this fix. Figure 9 shows the files used in this test scenario. All the highlighted syntax errors have been fixed by ASBF.

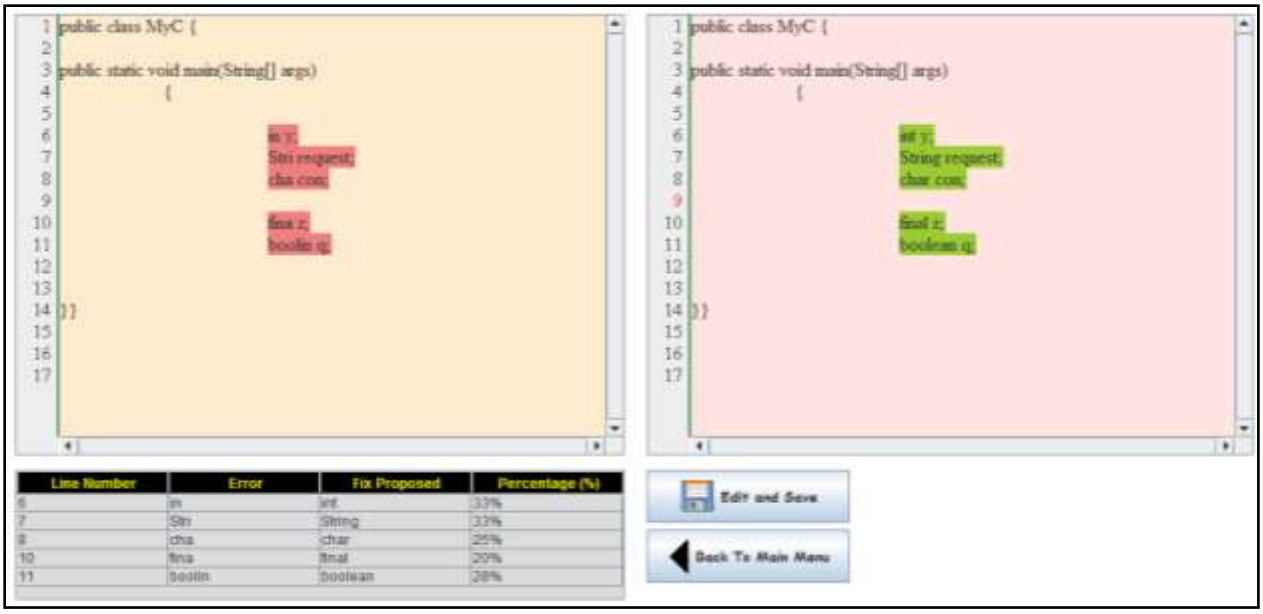

Figure 7: Erroneous Java file fix

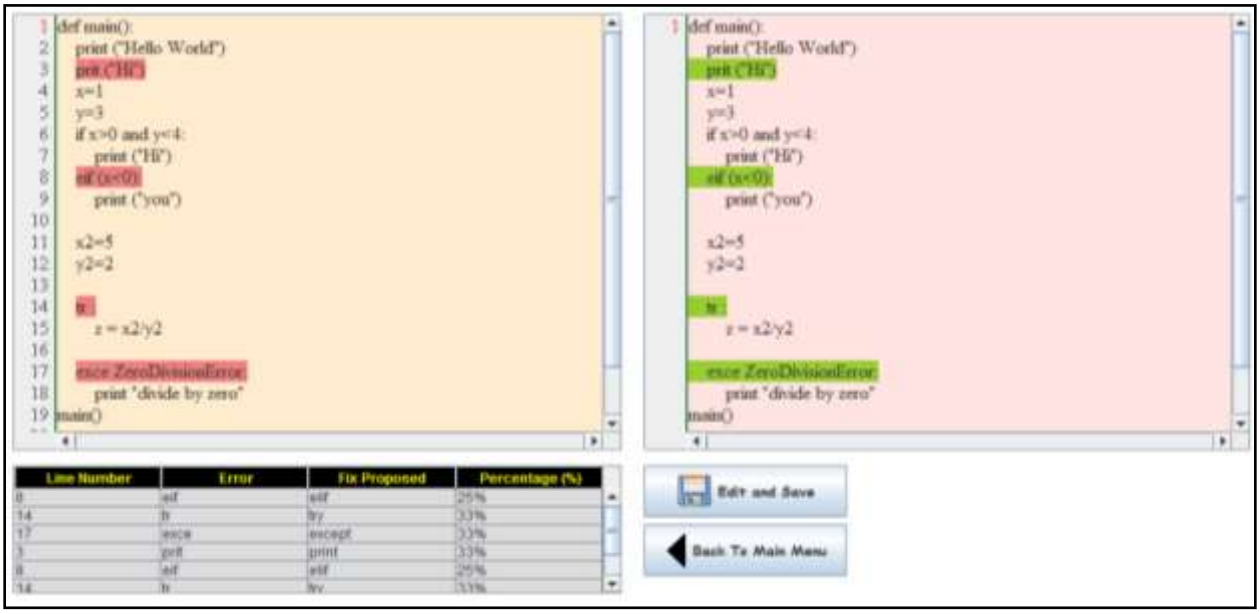

Figure 8: Erroneous Python file fix

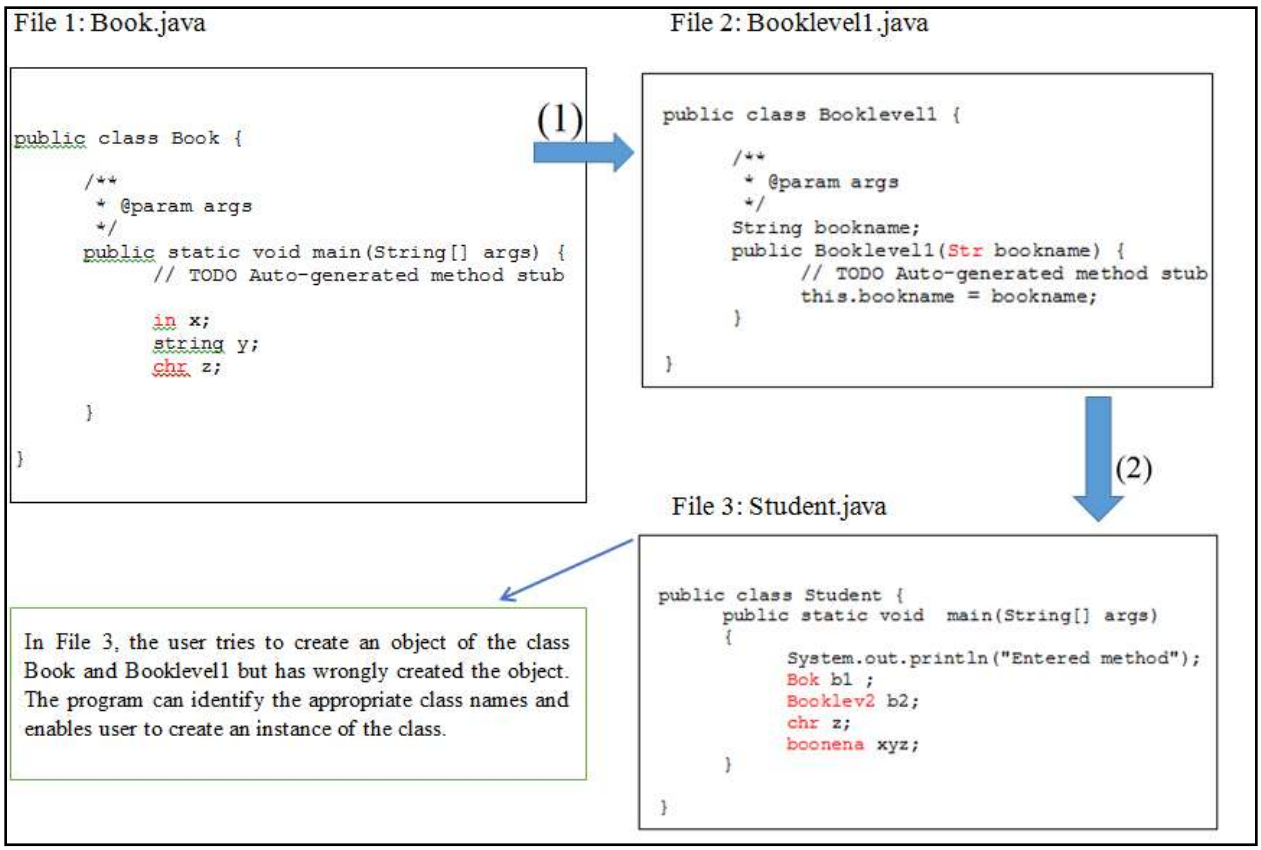

Figure 9: Multiple Java file fix

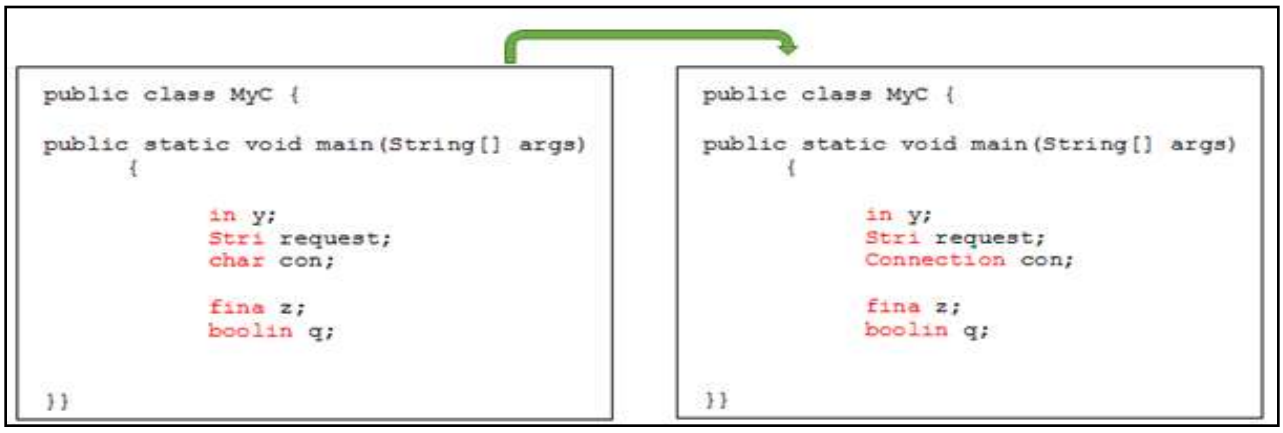

Figure 10. ASBF Autolearn

The final test scenario explains the Autolearn feature of ASBF. In case, the user does not accept certain fix, he/she can always edit those fixes and the system captures them to better provide more accurate fixes in the future. A sample preview of this feature is shown in Figure 10.

ASBF expects the fixes for the buggy file to be: *int*, *string*, *char*, *final* and *boolean*. If the user edits the keyword "*char*" to "*connection*" and the changes are captured and stored in the autolearn table. When the same erroneous file is tested again, ASBF checks the autolearn table first, identifies a user defined fix for

the keyword "*char*" and uses this fix "*connection*" to fix the file.

7. Discussions and evaluations

From the tests carried out, it has been observed that ASBF is a functional automated software bug fixing tool capable of fixing syntax errors. It automatically identifies the programming language, makes use of existing and appropriate compilers/interpreters, to locate errors, and a keywords table to calculate the levenshtein distance between the identified syntax errors and the keywords for that language. It automatically identifies the best fix for each syntax error and is able to correct an erroneous program. It is also possible to correct multiple files in a source folder and supports different languages. ASBF supports auto-learning which helps to improve the bug fixing process in case similar errors are encountered in future.

The main limitations of ASBF are:

- It cannot fix semantic errors
- Swapping cannot be done – e.g. correct *"public void static main"* into *"public static void main"*
- It heavily depends on existing compilers. If the compilers do not return the correct position of syntax errors, it is difficult for ASBF to fix that error. E.g. In the following: *System.out.println"Hello");* - a "(" is missing just before the string "Hello". The compiler identifies the error but it returns "missing ;" missing instead of "(".

To our analysis, it was found that KeshMesh is most powerful compared to AppPerfect, AFix,AutoFix Tool and PACHIKA. So KeshMesh has been used as a benchmark to evaluate ASBF. The following evaluation criteria have been used:

- Automatic recognition of the language in the file.
- Ease of Use: The user friendliness of the tool.
- GUI for interaction with the system.
- Suggest a list of potential solutions for each error.
- Auto Learn new keywords from user.
- Highlight errors discovered in erroneous file.

These criteria were rated based on a score of 1 to 5 where:
1 = Poor, 2 = Fair, 3 = Good, 4 = Very Good and 5 = Excellent.

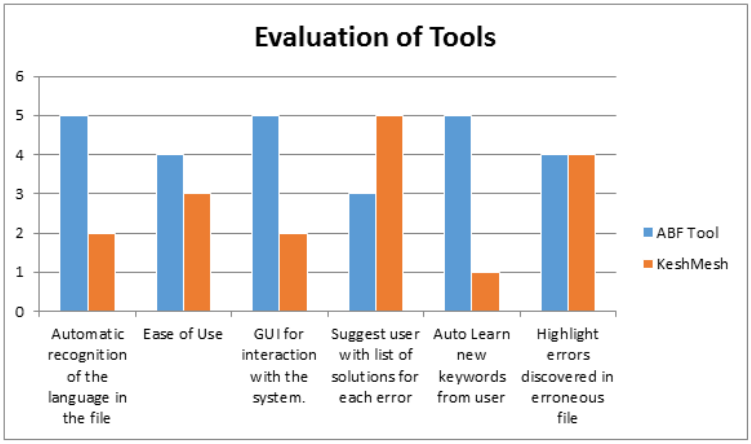

Figure 11: ASBF Evaluation

Figure 11 below shows a bar chart for the evaluation of ASBF against the different evaluation criteria. ASBF supports and automatically identifies different languages; it is much easier to use with its GUI; it provides for auto-learning capabilities; hence obtaining higher ratings for these criteria. However, it lacks in terms of showing a list of possible fixtures in case a fix cannot be found. It however performs similar highlighting of errors and corrections as KeshMesh.

8. Conclusion and Future Works

In view of finding a solution to the problem of automating the bug fixing process, ASBF has been proposed as a solution towards the automatic correction of bugs, more specifically syntax errors, in an erroneous file. One of the main characteristics of ASBF is that it makes use of existing compilers and hence can be relatively easy to implement in existing IDEs. Overall, ASBF fixes bugs to a high level of accuracy. It makes use of the levenshtein distance between a syntax error and keywords for a specific language to identify the best fix for that error. ASBF supports multiple programming languages and is able to fix a single erroneous file and also a complete source folder containing multiple erroneous files. It also supports automatic learning whereby it can subsequently correct errors which it initially could not. Experimental results have also shown that finding, fixing all the bugs found and outputting the corrected files are done quickly by ASBF. One shortcoming of the ASBF Tool is that it heavily depends on the compilers of the language and especially on how compilation errors are being returned. ASBF also relies heavily on databases and rule based programming is a good option to be considered in the future to refine the auto learning

feature. However, the development of the ASBF does not stop at successfully implementing these features. There are a number of enhancements that can be done to the system to make it more powerful and useful in the world of automated software engineering. For scanning and auto correcting semantic errors, rule based reasoning can be a really good option to ponder upon. Rules, facts and an inference engine can be used to further automate the bug fixing process and at the same time bring added value to ASBF. Currently, ASBF pins in and out of a database of two tables, and this clearly depicts a single point of failure. Extending the QuickFix Component in Eclipse to automatically choose the best option can be a good way to refine ASBF in the future, thus alleviating the dependency of ASBF on the database and making it more robust. The autolearn feature in the ASBF Tool also has room for future development. ASBF currently captures the changes made by user and adds it in the database. The problem lies in the fact that users may unknowingly put erroneous keywords while making changes. Access to an online library to verify the correctness of the new keywords inserted can be a very appropriate and simple way to improve the auto learn feature in ASBF.ASBF can be further extended to support other languages if their compilers and keywords are available. The same concept used in the ASBF can be followed to support much more programming languages. Currently the keywords for a programming language are manually entered in the keyword database. To enable ASBF to support more languages, an automatic loading of the keywords in the database must be envisaged. In a nutshell, ASBF provides a way of automatically locating syntax errors using existing compilers and performing reliable automatic correction without user intervention and basing itself on the levenshtein distance between the syntax errors and the keywords for that programming language.

References

[1] R. Chmiel, &M. C. Loui, “Debugging: From novice to expert”, ACM SIGCSE Bulletin 36.1, pp 17-21, 1992.

[2] K. Pan, S. Kim and E. J. Jr. Whitehead, “Toward an understanding of bug fix patterns”, Empirical Software Engineering 14, no. 3, pp 286-315, 2009

[3] A. Ko &B. Myers, “A framework and methodology for studying the causes of software errors in programming systems”, Journal of Visual Languages and Computing, 16, pp 41–84, 2005.

[4] S. Fitzgerald, G. Lewandowski, R. McCauley, L. Murphy, B. Simon, L. Thomas and C. Zander, “Debugging: Finding, Fixing and Flailing –A multi-institutional study of novice debuggers”, Computer Science Education 18, no. 2, pp. 93-116, 2008.

[5] S. Forrest, T. Nguyen, W. Weimerand C. Le Goues, “A Genetic Programming Approach to Automated Software Repair”, In Proceedings of the 11th Annual conference on Genetic and evolutionary computation, pp. 947-954, ACM, 2009.

[6] P. McMinn, “Search-based software testing: Past, present and future”,In Software testing, verification and validation workshops (icstw), IEEE fourth international conference on, pp. 153-163. IEEE, 2011.

[7] N. Ayewah, D. Hovemeyer, J. D. Morgenthaler, J. Penix, and W. Pugh, “Experiences Using Static Analysis to Find Bugs”, IEEE Software, vol. 25, pp. 22-29, 2008.

[8] M. Vakilian, S. Negara, S. Tasharofi and R. E. Johnson,“Keshmesh: A Tool for Detecting and Fixing Java Concurrency Bug Patterns”, In Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion, pp. 39-40. ACM, 2011.

[9] G. Jin, L. Song, W. Zhang, S. Lu and B. Liblit, “Automated Atomicity-Violation Fixing”, In ACM SIGPLAN Notices, vol. 46, no. 6, pp. 389-400. ACM, 2011.

[10] Y. Pei, C. A. Furia, M. Ordio, Y. Wei, B. Meyer and A. Zeller, “Automated fixing of programs with contracts”, IEEE transactions on software engineering, 40(5), pp.427-449, 2014.

[11] Vancouver V. Dallmeier, A. Zeller, and B. Meyer, “Generating fixes from object behavior anomalies”, In Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering, pp. 550-554. IEEE Computer Society, 2009.

[12] AppPerfect Java Code Test. Available at http://www.appperfect.com/products/java-code-test.html [Accessed: Feb. 12, 2016].

[13] Z. Gu, E. T. Barr, D. J. Hamilton and Z. Su, “Has the bug really been fixed?”, In Proceedings of the 32nd ACM/IEEE

International Conference on Software Engineering - Volume 1, pp. 55–64, 2010.

[14] C. Le Goues, M. Dewey-Vogt, S. Forrest and W. Weimer, "A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each", In Proceedings of the 2012 International Conference on Software Engineering, pp. 3- 13, 2012.

[15] L. S. Pinto, S. Sinha,and A. Orso, "Understanding myths and realities of test-suite evolution" , In Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering, Article No. 33, 2012.

[16] C. Parnin, and A. Orso, "Are automated debugging techniques actually helping programmers?", In Proceedings of the 2011 International Symposium on Software Testing and Analysis, pp. 199 – 209, 2011.

[17] A. J. Ko and B. A. Myers, "Debugging reinvented: asking and answering why and why not questions about program behavior", In Proceedings of the International Conference on Software Engineering (ICSE 08), pp. 301–310, Leipzig, Germany, 2008.

[18] S. Bhatia and R. Singh, "Automated Correction for Syntax Errors in Programming Assignments using Recurrent Neural Networks", arXiv preprint arXiv:1603.06129, 2016.

[19] Eclipse IDE, Available at http://www.eclipse.org/ [Accessed: Jan. 10, 2016]

Author Profile

***Nagowah Leckraj,*MSc,** is a lecturer at the Computer Science and Engineering Department of the University of Mauritius. He has a BSc (Hons) in Computer Science with First Class Honours and an MSc in Computer Science with specialization in Software Engineering. He has about 10 years teaching experience. His research interests are mainly in Software Engineering, Automated Testing, Mobile Technologies, Internet of Things and Analytics. He has published more than 15 research papers in international conferences and journals, and has supervised projects in these areas. He is also a scientific committee member for a number of international conferences.

Jooty Suneil, MSc,is a software engineer currently working in an offshore company. He has a BSc (Hons) in Computer Science with First Class Honours and an MSc in Software Engineering Projects and Management, both from the University of Mauritius. He has 3.5 years of experience in software development. His research interests are mainly in Cloud computing, Big Data and Software Engineering.

Jeetun Antish has successfullygraduated with a degree in BSc (Hons) Computer Science & Engineering in 2013 from the University of Mauritius. Currently, he is pursuing the LLB Degree as a third year student at the University of London International Programmes. He has keen interest in the field of Law and Information Technology.

Nagowah Soulakshmee Devi, MSc, is a lecturer at the Computer Science and Engineering Department of the University of Mauritius. She has a BSc (Hons) in Computer Science with First Class Honours and an MSc Computer Science with specialization in Software Engineering. She has over ten years of teaching experience. Her research interests are directed towards Mobile Computing, Context-Awareness, Internet of Things, Software Engineering, Knowledge Engineering and Big Data. She has published a number of research papers in renowned international conferences and journals and has supervised projects in these areas. She has also worked on a research project entitled "A Secure Data Access Model for the Mauritian Healthcare Service" funded by the Mauritius Research Council. She has been awarded *Best Paper Award in the second place* by IEEE Software for the paper "An Automatic and Intelligent Workflow Design" for the IEEE co-sponsored ACSEAC – African Conference on Software Engineering and Applied Computing, 2011.