AFRL-RI-RS-TR-2020-122

ASSURANCE-BASED DEVELOPMENT AND VALIDATION FOR AUTO PROGRAM REPAIRS

WESTERN MICHIGAN UNIVERSITY

JULY 2020

FINAL TECHNICAL REPORT

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED

STINFO COPY

AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE

■ **AIR FORCE MATERIEL COMMAND** ■ **UNITED STATES AIR FORCE** ■ **ROME, NY 13441**

NOTICE AND SIGNATURE PAGE

Using Government drawings, specifications, or other data included in this document for any purpose other than Government procurement does not in any way obligate the U.S. Government. The fact that the Government formulated or supplied the drawings, specifications, or other data does not license the holder or any other person or corporation; or convey any rights or permission to manufacture, use, or sell any patented invention that may relate to them.

This report is the result of contracted fundamental research deemed exempt from public affairs security and policy review in accordance with SAF/AQR memorandum dated 10 Dec 08 and AFRL/CA policy clarification memorandum dated 16 Jan 09. This report is available to the general public, including foreign nations. Copies may be obtained from the Defense Technical Information Center (DTIC) (http://www.dtic.mil).

AFRL-RI-RS-TR-2020-122 HAS BEEN REVIEWED AND IS APPROVED FOR PUBLICATION IN ACCORDANCE WITH ASSIGNED DISTRIBUTION STATEMENT.

FOR THE CHIEF ENGINEER:

/ S /
WILMAR SIFRE
Work Unit Manager

/ S /
GREGORY HADYNSKI
Assistant Technical Advisor
Computing & Communications Division
Information Directorate

This report is published in the interest of scientific and technical information exchange, and its publication does not constitute the Government's approval or disapproval of its ideas or findings.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YYYY)*	2. REPORT TYPE	3. DATES COVERED *(From - To)*
JULY 2020	FINAL TECHNICAL REPORT	NOV 2018 – JAN 2020

4. TITLE AND SUBTITLE

ASSURANCE-BASED DEVELOPMENT AND VALIDATION FOR AUTO PROGRAM REPAIRS

5a. CONTRACT NUMBER
FA8750-19-2-0007

5b. GRANT NUMBER
N/A

5c. PROGRAM ELEMENT NUMBER
62788F

6. AUTHOR(S)

Wuwei Shen, Zhongqiu Gao, Charles DeAndre Donte Noble, and Ioannis Nearchos Nearchou

5d. PROJECT NUMBER
T2EE

5e. TASK NUMBER
GP

5f. WORK UNIT NUMBER
AS

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Western Michigan University
1903 W Michigan Ave
Kalamazoo MI 49008

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Air Force Research Laboratory/RITA
525 Brooks Road
Rome NY 13441-4505

10. SPONSOR/MONITOR'S ACRONYM(S)
AFRL/RI

11. SPONSOR/MONITOR'S REPORT NUMBER
AFRL-RI-RS-TR-2020-122

12. DISTRIBUTION AVAILABILITY STATEMENT
Approved for Public Release; Distribution Unlimited. This report is the result of contracted fundamental research deemed exempt from public affairs security and policy review in accordance with SAF/AQR memorandum dated 10 Dec 08 and AFRL/CA policy clarification memorandum dated 16 Jan 09.

13. SUPPLEMENTARY NOTES

14. ABSTRACT

In this project, a novel technique to evaluate the quality of an auto program repair by means of employing the D-S theory as well as the symbolic execution engine called KLEE to reveal the difference between two versions of a program as much as possible was developed. Specifically, we generate various test case reports covering as many different paths in both versions as possible. Based on the test case reports, we apply the safety pattern to generate an assurance case in the Goal Structuring Notation (GSN) showing how an auto program repair technique is evaluated. Last, our technique adopts the D-S theory to evaluate the assurance case as the final evaluation of quality of an auto program repair.

15. SUBJECT TERMS

Auto program repair, D-S theory, assurance case, symbolic execution, software testing, Goal Structuring Notation (GSN)

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			**WILMAR SIFRE**
U	U	U	UU	38	19b. TELEPHONE NUMBER (*Include area code*) **N/A**

TABLE OF CONTENTS

Section **Page**

List of Figures

List of Tables

1. Summary

Automatic program repair (APR) has drawn great interest in the software community since it has been shown to be an effective and efficient technique that otherwise requires extensive human effort and time. However, the quality of automatic program repair heavily depends on a test suite. So, automatic program repair techniques have been far from practical in terms of deployment. One major obstacle is the limited number of test cases, in a test suit, considered by an automatic program repair approach. When additional test cases are considered, the test oracle problem becomes another major obstacle of having an automatic program repair approach effectively fix a faulty version. For instance, the lack of test oracles makes it hard to predict whether a new test input produces a correct output or not. Furthermore, even when a test oracle is present, a weak or inappropriate test oracle can lead to inaccurate classifications of passing and failing test cases. Thus, these obstacles make it doubtful to trust the claim, i.e., that a faulty version is *really* fixed via a new patch produced by an automatic program repair approach. In this paper, we aim to evaluate the reliability of the claim made by an automatic program repair approach from a certification perspective. To achieve this goal, we present a novel framework to evaluate the quality of a patched version as a repair to a faulty version via a symbolic-based testing process. Our evaluation is based on a logic reasoning process that is recorded in an argument structure, called an assurance case, given in the Goal Structuring Notation. A logic reasoning process is given a safety pattern that shows not only test case reports as evidence but also how test requirements and the test cases are generated as the main inference structure. Finally, the framework applies the Dempster-Shafter (D-S) theory to quantitatively evaluate an assurance case to determine if it is a good repair to a faulty program.

2. Introduction

Automatic program repair (APR) [1] was proposed more than ten years ago [2] and since then we have seen a research boom in developing various APR approaches [3, 4, 5, 6, 7] due to potential reduction in the maintenance of large-scale software [8]. For APR, a program is considered faulty if, given a test suite consisting of many test cases, the program fails at least one test case, called a failing test case, but passes the rest of the test cases, called passing test cases. Existing APR approaches aim to find a plausible patch that can pass all test cases, including previously failing test cases. In general, a plausible patch is generated from a faulty program by means of performing some simple modifications on suspicious statements.

Obviously, how to choose test cases in a test suite has a profound impact on the quality of APR approaches. However, most state-of-the-art APR approaches focus on how to find and then modify a suspicious statement to ensure that the modified version can pass all the test cases in the test suite but fail to consider the quality of test cases in a test suite [9, 10, 4, 11, 12, 13]. Since an APR approach claims to fix a faulty program only based on a test suite, the claim about whether the APR approach has really fixed the faulty program is still worthy of further study. Currently, there are two major factors that are blamed for the low-quality patches. One factor is the inaccurate test oracle problem and the other one is the weakness of a test suite [9]. The latest study shows that low-quality patches lead to low performance compared to when developers are not provided with any patch [14].

In this project, we propose a novel framework that studies the quality of a patched version generated by an APR approach with a concentration on the overall execution behavior. Thus, the

framework adopts Symbolic Execution (SE) [15] to not only study the behavior of two versions via the test cases in a test suite but also explore the additional behavior of two versions by means of finding as many other execution paths as possible. Thus, the difference between two versions can be fully investigated through various execution paths, facilitating understanding, and further evaluating whether a patched version can really fix errors in the original version. In this case, the framework is *not* restricted to the passing and failing test cases given in a test suite. As an evaluation approach, we notice that every evaluation approach has its own criteria such as judgement rules, context, and assumptions. Moreover, various stakeholders benefit from an evaluation approach if its judgement rules, context, and assumptions as well as other factors behind the approach can be explicitly shown. Therefore, in this project, we adopt the certification-based technique as the main manifestation of our evaluation for APR. But, how the various execution paths between two versions can be integrated into the evaluation of the quality of an APR approach, especially from the certification perspective, becomes a crucial issue in the framework. A reason for us to choose the certification perspective is because certification always requires some explicit logic structure among various artifacts so different stakeholders such as developers and certifiers can evaluate the quality of a system via the same notation.

In the certification community, assurance cases have been increasingly considered in many emerging standards or guidelines as a structured argument to show that a safety critical system is acceptably safe [16]. Advancement in generation and evaluation of assurance cases can assist various stakeholders such as developers and certifiers to efficiently and effectively evaluate a system [17, 18, 19, 20, 21, 22, 23, 24]. In this framework, a structured argument via an assurance case is automatically generated to demonstrate the quality evaluation of a patched version by means of how symbolic testing has been carried out in revealing the difference between two versions from the evolution of test requirements to test cases; and finally, to test case reports. The framework quantifies the confidence of the claim about a patched version as a good repair to a faulty version by employing the Dempster-Shafter (D-S) theory to a generated assurance case [25]. Specifically, the confidence of a claim is deduced by deducing the confidence of a root claim in an assurance case. To do so, the framework adopts a bottom-up strategy to propagate the confidence from leaf nodes, i.e., evidence nodes, each of which corresponds to a test case report, to their parent nodes and finally to a root claim.

3. Methods, Assumptions and Procedures

3.1 An Illustrative Example

To evaluate the confidence of a claim made by an APR approach, i.e., a fault in a faulty program is fixed via a patch generated by the APR approach, the framework employs an assurance case in Goal Structure Notation (GSN) [26] as a main vehicle to not only demonstrate a logical reasoning process of an evaluation process but also approximate a human certifier's evaluation process by deducing the confidence of a root goal in the assurance case. The root goal in an assurance case claims that a patched version *P'* is a good repair to a faulty program *P* by an APR approach. To support the root claim, an assurance case provides a structured argument supported by a body of evidence which refers to test reports. A structured argument shows the rationale behind

the evaluation process, i.e., generations of test requirements, test cases and test case reports. A higher value of confidence of a root claim in an assurance case accounts for higher confidence about a patch *P'* being a good repair to a faulty program *P*.

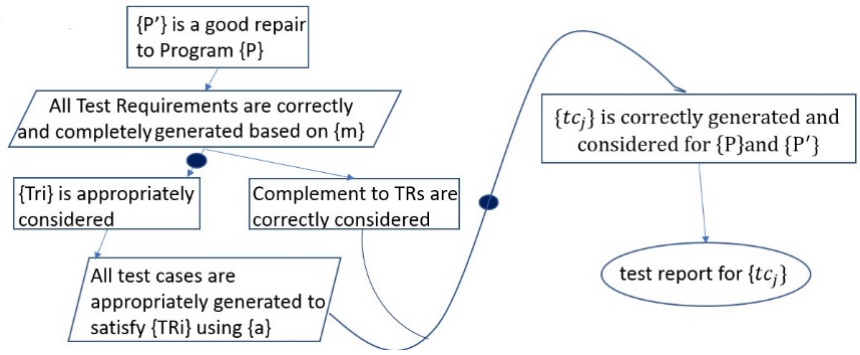


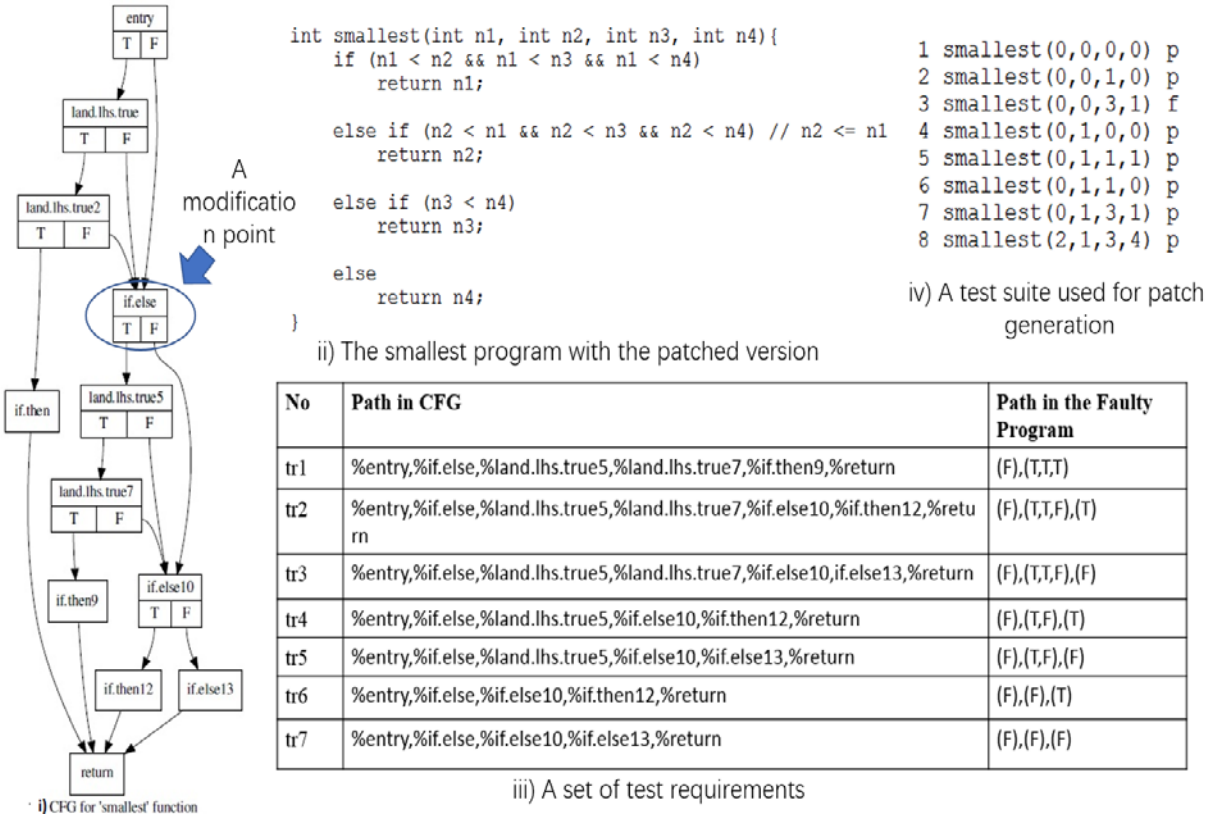
Figure 1 A Safety Pattern.

To generate an assurance case, the framework adopts the safety pattern shown in *Figure 1* where x in $\{x\}$ denotes a variable to be instantiated for a specific APR approach. For instance, the top claim, also called a goal and rendered as a box in GSN, has two variables, i.e., *P'* and *P*, denoting a patched version and faulty version respectively. These two variables are replaced with a specific patched version and faulty version respectively during instantiation, i.e., when the pattern is applied to a specific APR approach. The top goal is justified by an argument step on the basis of the subgoal, claiming a {TRi} is correctly generated. Again, *Tri* is replaced with a concrete test requirement during instantiation. The argument step is given by a strategy node, rendered as a parallelogram in GSN, showing the nature of the argument step between a goal and its supporting subgoal(s). In this case, it shows all test requirements are correctly generated based on an approach-related variable $\{m\}$ denoting the rationale behind test requirement generation. The line denoting the *supportedBy* relation between the strategy node and subgoal node has a bullet, meaning that the number of the subgoal nodes, i.e. the number of test requirements created, depends on real programs *P* and *P'*.

Likewise, a subgoal of each $\{Tri\}$ being appropriately considered is justified by an argument step on the basis of the subgoal, claiming a specific test case is correctly generated and considered for *P* and *P'*. The argument step is given by the strategy node showing all test cases are appropriately generated to satisfy *Tri* using an approach-related variable $\{a\}$, denoting how test cases are generated from a test requirement. Likewise, the line denoting the *supportedBy* relation between the strategy node and subgoal node has a bullet, indicating the number of subgoals, i.e. the number of test cases, depends on real programs *P* and *P'*. Finally, the subgoal claiming "each test case is correctly generated and considered" is supported by an evidence node, rendered as an oval, which is a report of the test case, showing how the test case is run on both versions. However, there exist some test cases not satisfying any *Tri* in a test suite. To show this scenario, we introduce a new subgoal node claiming that "complement to TRs are correctly considered". Thanks to the new subgoal node, the first strategy node is updated to "All Test Requirements are correctly *and completely* generated based on $\{m\}$".

Next, we use the program shown in Figure 2 to illustrate our approach. Figure 2 ii) shows the two versions where a patched version is given by a comment, meaning the statement given in a

comment replaces the original statement. For instance, condition *n2<n1* in the second if statement is replaced with *n2<=n1* in the patched version. The framework takes an LLVM [27] as representation of a program. So, Figure 2 i) shows a Control Flow Graph (CFG) of the faulty version, in which we use the circle to mark the difference between two versions. Note that an LLVM method consists of a set of basic blocks (BBs) and there is only one BB associated with a return statement, which we call an *exit basic block (BB)*. Contrary to an exit BB, there is an *entry basic block (BB)* marking the beginning of a method.



No	Path in CFG	Path in the Faulty Program
tr1	%entry,%if.else,%land.lhs.true5,%land.lhs.true7,%if.then9,%return	(F),(T,T,T)
tr2	%entry,%if.else,%land.lhs.true5,%land.lhs.true7,%if.else10,%if.then12,%return	(F),(T,T,F),(T)
tr3	%entry,%if.else,%land.lhs.true5,%land.lhs.true7,%if.else10,if.else13,%return	(F),(T,T,F),(F)
tr4	%entry,%if.else,%land.lhs.true5,%if.else10,%if.then12,%return	(F),(T,F),(T)
tr5	%entry,%if.else,%land.lhs.true5,%if.else10,%if.else13,%return	(F),(T,F),(F)
tr6	%entry,%if.else,%if.else10,%if.then12,%return	(F),(F),(T)
tr7	%entry,%if.else,%if.else10,%if.else13,%return	(F),(F),(F)

iii) A set of test requirements

Figure 2 An Illustrative Example with Generated Test Requirements.

To generate all test requirements, we adopt the Reachability, Infection, Propagate, and Revealability (RIPR) model, to reveal the difference between two versions as much as possible. For a given modification point, the BB including the modification point is denoted as *Mod_BB*. The framework finds one path from the entry BB to *Mod_BB*, denoted as *prefix*, and all paths from *Mod_BB* to the exit BB, denoted as $\{p_1,..,p_n\}$. Finally, all complete test requirements, or simply called test requirements, are given as $\{prefix \oplus p_1,..,prefix \oplus p_n\}$ where $a \oplus$b is a sequence concatenation, except for the first element of b, which is removed from the final sequence that includes *Mod_BB* only once. In Figure 2 i), given the modification point, i.e. the %if.else BB, the framework finds 7 paths from the %if.else BB to the exit BB in a faulty program. After that, the framework selects one of the three paths from the entry BB to the %if.else BB, i.e. %entry,%if.else,

as a prefix and then attaches it to all the 7 paths as complete test requirements as shown in Figure 2 iii).

In order to explicitly relate test requirements to execution paths in an original faulty version, Figure 2 iii) shows, for each test requirement, how the if statements are evaluated in the last column, i.e. the Path in the Faulty Program column. Since there are three if statements in the faulty program, three tuples are given, and each tuple corresponds to one if statement and is given by a pair of "(" and ")". For instance, for *tr2*, the corresponding path in the faulty version, i.e., (F),(T,T,F),(F), is explained as follows:

1. (F), the first conjunct in the first if statement is false, making the first if statement false; and
2. (T,T,F), the first two conjuncts in the second if statement are true while the third is false, making the second if statement false, and
3. (F), the only condition in the third if statement is false, and so the false branch of the third if statement, i.e., "return n4" is executed.

Note that the condition evaluation adopts the short circuit strategy and so if a conjunct is false in an if statement, the runtime system does not evaluate the rest of conjuncts in the if statement and directly jumps to the false branch. That is why there is only one "F" in the first tuple. Likewise, test requirement *tr1* is given by (F),(T,T,T) meaning that the first if statement is false due to the first conjunct being false and the second if statement is true due to all the three conjuncts being true, making "return n2" execute.

After generating all test requirements, the framework employs the KLEE engine to symbolically execute two versions of the program. First, for a given test requirement, denoted as {*entry,…,modification,…,exit*}, the framework finds one execution path of a faulty program satisfying the path given by the test requirement via a path constraint returned by KLEE, denoted as p_{old}. Further, the framework calculates all execution paths of a patched program such that all execution paths share the same path prefix given by {*entry,…,modification*} in the first section while the second sections of all the execution paths list all possible paths from the modification BB to the exit BB in a patched version. Assume a patched program has *n* execution paths, such that they are represented by path constraints $p_{new_1}, \dots, p_{new_n}$, that are returned by KLEE, respectively. Finally, the framework checks whether $p_{old} \wedge p_{new_i}\ where\ i \in \{1,2, \dots, n\}$ is satisfiable by means of the Z3 solver [28]. If it is satisfiable, it means there exists a test case that covers the paths given by p_{old} and p_{new_i} in a faulty version and a patched version respectively. Otherwise, the paths covered by p_{old} and p_{new_i} in two versions are not feasible via one test case.

Table 1 shows how test cases are generated to satisfy their corresponding test requirement. First, the framework executes the patched program and finds 7 execution paths, each of which is recorded by its corresponding path constraint returned by KLEE. Then, the framework checks each test requirement to find whether a concrete test case can be found to satisfy the test requirement. As a rule, the framework gives priority to the test cases in a test suite, such as the one shown in Figure 2

Table 1 Test Cases Generated for the Illustrative Example.

	Covered Test Case	Old Path	New Path
tr1	(2,1,3,4)	(F),(T,T,T) !(n1 < n2) and (n2 < n1) and (n2 < n3) and (n2 < n4)	(F),(T,T,T) !(n1 < n2) and (n2 <= n1) and (n2 < n3) and (n2 < n4)
tr2		(F),(T,T,F),(T) !(n1 < n2) and (n2 < n1) and (n2 < n3) and !(n2 < n4) and (n3 < n4)	
tr3	(32769,32768,32769,1)*	(F),(T,T,F),(F) !(n1 < n2) and (n2 < n1) and (n2 < n3) and !(n2 < n4) and !(n3 < n4)	(F),(T,T,F),(F) !(n1 < n2) and (n2 <= n1) and (n2 < n3) and !(n2 < n4) and !(n3 < n4)
tr4	(49153,32768,2,3) *	(F),(T,F),(T) !(n1 < n2) and (n2 < n1) and !(n2 < n3) and (n3 < n4)	(F),(T,F),(T) !(n1 < n2) and (n2 <= n1) and !(n2 < n3) and (n3 < n4)
tr5	(57345,57344,40962,8195)*	(F),(T,F),(F) !(n1 < n2) and (n2 < n1) and !(n2 < n3) and !(n3 < n4)	(F),(T,F),(F) !(n1 < n2) and (n2 <= n1) and !(n2 < n3) and !(n3 < n4)
tr6	(55137,55137,22402,22403)*	(F),(F),(T) !(n1 < n2) and !(n2 < n1) and (n3 < n4)	(F),(T,F),(T) !(n1 < n2) and (n2 <= n1) and !(n2 < n3) and (n3 < n4)
	(65520,65520,65522,65523)*		(F),(T,T,T) !(n1 < n2) and (n2 <= n1) and (n2 < n3) and (n2 < n4)
tr7	(0,0,0,0)	(F),(F),(F) !(n1 < n2) and !(n2 < n1) and !(n3 < n4)	(F),(T,F),(F) !(n1 < n2) and (n2 <= n1) and !(n2 < n3) and !(n3 < n4)
	(0,0,1,0)		(F),(T,T,F),(F) !(n1 < n2) and (n2 <= n1) and (n2 < n3) and !(n2 < n4) and !(n3 < n4)
	(0,0,3,1)		(F),(T,T,T) !(n1 < n2) and (n2 <= n1) and (n2 < n3) and (n2 < n4)
others	(0,1,0,0)	(T, F), (F), (F) (n1 < n2) and !(n1<n3) and !(n2 < n1) and !(n3 < n4)	(T, F), (F), (F) (n1 < n2) and !(n1<n3) and !(n2 < n1) and !(n3 < n4)
	(0,1,1,1)	(T, T, T) (n1 < n2) and (n1<n3) and (n1<n4)	(T, T, T) (n1 < n2) and (n1<n3) and (n1<n4)
	(0,1,1,0)	(T, T, F), (F), (F) (n1 < n2) and (n1<n3) and !(n1<n4) and !(n2 < n1) and !(n3 < n4)	(T, T, F), (F), (F) (n1 < n2) and (n1<n3) and !(n1<n4) and !(n2 < n1) and !(n3 < n4)

* Indicates the test case generated by the framework.

iv) when satisfying a test requirement. For instance, for *tr1*, the framework finds that one of the seven path constraints in the patched program combined with the path constraint in the faulty program via the ∧ operator can be satisfied via (2,1,3,4). Thus, test case (2,1,3,4) and its execution paths for both versions are shown in the *tr1* row. Likewise, Table 1 shows the three tuple values corresponding to the execution scenario of the three if statements in both versions via the Old Path and New Path columns, representing a faulty version and a patched version respectively. Also, we include a corresponding path constraint in the Old Path and New Path columns as well. For the *tr7* row, the framework finds that three path constraints in the patched version combined with the path constraint in the faulty version are satisfiable. Thus, three test cases are generated to satisfy *tr7*. Last, there is no test case that can satisfy *tr2* since all path constraints given by the 7 execution paths of the patched version combined with the path constraint of the faulty version via the ∧ operator, are *all* not satisfiable. Note that APR employs a test suite to generate a "valid" patch. If there is no test requirement in the test suite in Figure 2 iv) that can be found to satisfy a test requirement, then the framework employs the Z3 solver to find a concrete test case. For instance, test cases are generated for tr3, tr4, tr5, and tr6 as shown in *Table 1*.

Last, there exist some test cases in a test suite in Figure 2 iv) that do not satisfy any test requirement. For instance, three test cases, i.e., (0,1,0,0), (0,1,1,1), and (0,1,1,0), do not satisfy any test requirement in the program shown in Figure 2 i). So, we show these test cases in the "others" row in Table 1. After that, the framework evaluates each test report based on how the test case is run on both versions. If a test case is newly generated, the framework evaluates the execution path by means of measuring its distance to the known execution paths in a faulty version as well as the difference between two execution paths running on both versions. We adopt the two-tuple format,

i.e., *(dec, conf),* to represent a confidence of a test report as an approximation of a human certifier's evaluation. For a given node, *dec(A)* denotes a certifier's confidence value on *A* and *conf(A)* represents the confidence about how a certifier derives the value for *dec(A)*. All the results are output to a file called a GSN variable mapping file that not only maps all variables to the corresponding artifacts/results but also presents a two-tuple value for each test report.

Next, the framework outputs an assurance case in GSN as shown in Figure 3 after reading a GSN variable mapping file. Then, the framework applies the Dempster-Shafter(D-S) theory to evaluate an assurance case by means of the bottom-up strategy, starting with evaluation of leaf claims in an assurance case. Finally, a final two-tuple value for a root claim is deduced as confidence about the claim made by the root claim. In Figure 3, a two-tuple value (0.89318, 0.37595) is deduced for the claim made on the two versions of the program shown in Figure 1.

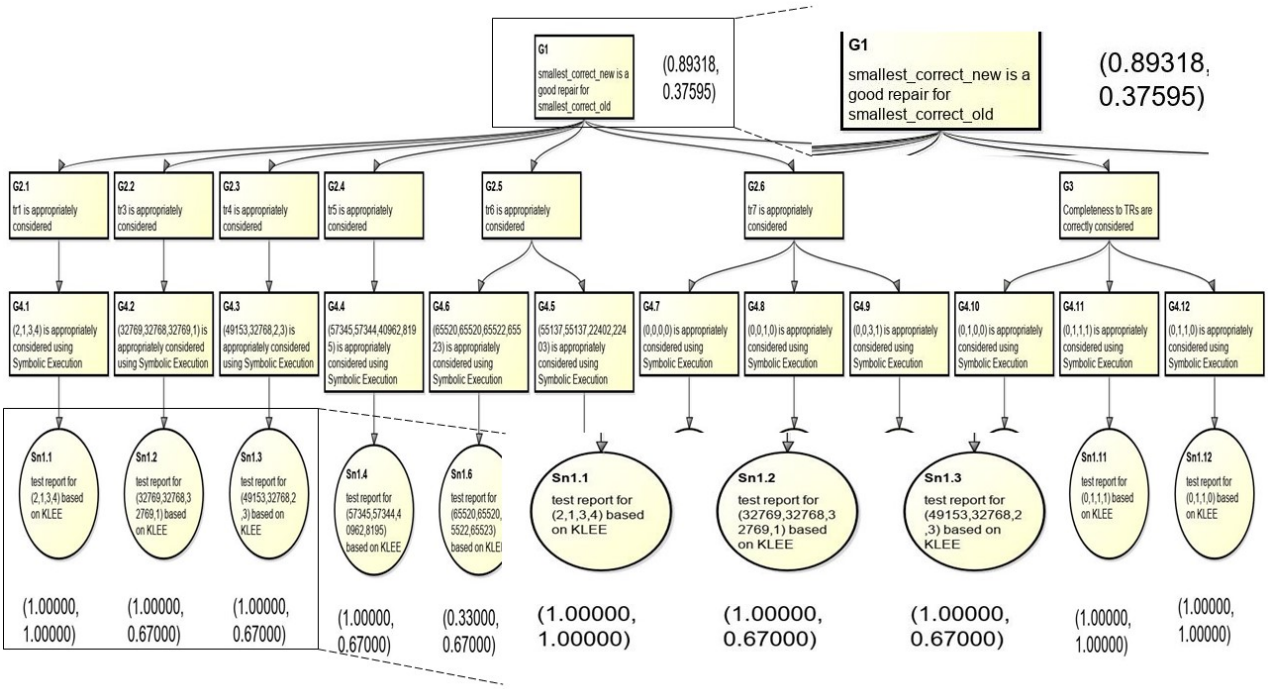


Figure 3 An Assurance Case for path entry 3 of Smallest project correct patch.

3.2 Overview of the Framework

Figure 4 illustrates the main flow of our framework that primarily consists of four modules. The first module called the Program Analyzer module takes as input a faulty version at the bitcode level, i.e., in LLVM IR format, and a text file showing the difference between two versions; and the module returns a set of test requirements. Next, the Symbolic Execution Engine module takes over to generate test cases that satisfy all test requirements output by the previous module. To do so, the module employs the KLEE engine as well as the Z3 solver to explore various execution paths in both versions. Then, the Test Report Evaluator module evaluates a report of each test case by studying the execution paths of both versions as an approximation of a human certifier's decision

on the test report. All the outputs produced by the first three modules are recorded in a text file, called a GSN variable mapping file, sent to the next module, i.e., the Assurance Case Generator module. This module applies the safety pattern to construct an assurance case based on all artifacts provided in a GSN variable mapping file. Last, the assurance case calculator module adopts the D-S theory using the bottom-up strategy to calculate the confidence of a root claim in an assurance case as quality evaluation of a patched version.

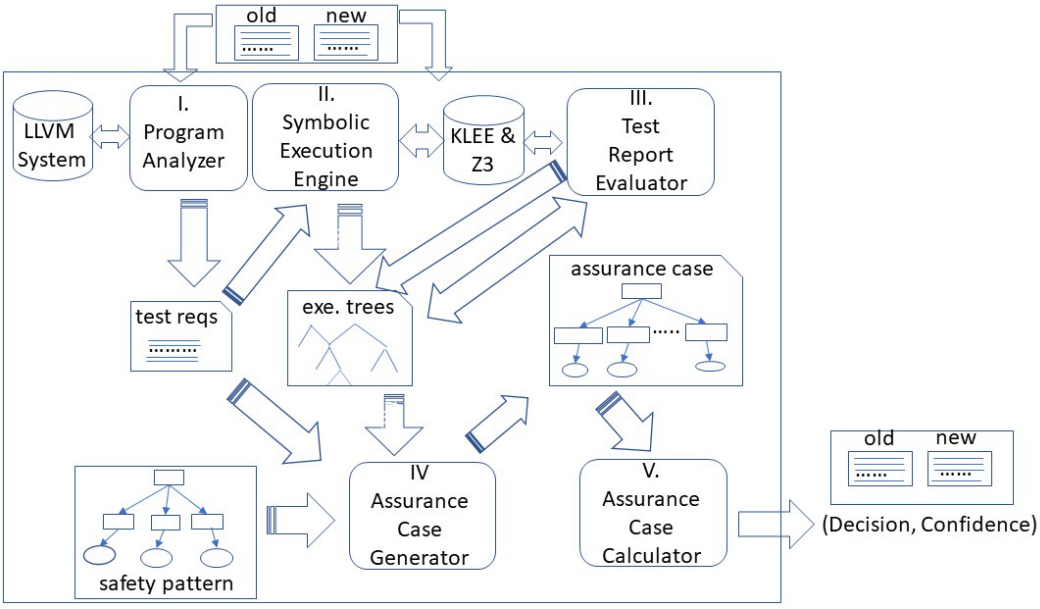


Figure 4 Overview of the Framework.

Implementation Details of the Framework

3.2.1 The Program Analyzer Module

The program analyzer module mainly outputs a set of test requirements based on the difference between two versions of a program. Specifically, the module takes a faulty version at the LLVM level as well as the difference between two versions as input and generates a set of complete test requirements, each of which consists of a sequence of basic blocks (BBs) starting with an entry basic block and ending with an exit basic block. In between, a complete test requirement includes one basic block showing the difference between two versions, called a modification basic block (BB). A sequence of basic blocks as a complete test requirement shows a potential execution path from the first block to the last block via a modification BB in the CFG denoting a faulty version. A subpath from the entry BB to a modification BB is called an entry path/requirement of a complete test requirement; and the remaining subpath is called a post-path of a complete test requirement.

There are three methods involved in the implementation. From the top down perspective, the first method is *list<Path> find_All_Complete_TR_paths(list<BB> modifications)* which takes a list of modification BBs as the *modifications* parameter and returns a list of BBs as all test requirements. The second method *list<Path> find_all_TR_pathsBTW(BB start_BB, BB end_BB)*

returns all paths between the two parameters *start_BB* and *end_BB*, denoting the starting BB and ending BB respectively. The last method is *list<Path> findAllPaths(BB start, BB end)* which returns all paths starting with the *start* BB and ends with either the *end* BB or an exit BB. A path ending with an exit BB is called a dead path since it cannot become a test requirement between *start* and *end.* Next, we individually introduce these three methods.

```
1  list<Path> find_All_Complete_TR_paths(list<BB> modifications)
2  {
3   list<Path> final_complete_TR_paths = {} ;
4   foreach(BB m_BB: modifications)
5   {   list<Path> pre_paths=find_all_TR_pathsBTW(entry,m_BB);
6       list<Path> mod_paths=find_all_TR_pathsBTW(m_BB, return);
7       Path pre_path = randomly selected path from pre_paths;
8       foreach(Path p:mod_paths)
9          append p to pre_path to form a new path and add it to final_complete_TR_paths;
10  }
11  return final_complete_TR_paths;
12 }
```

Figure 5 Pseudocode for the find_All_Complete_TR_paths function.

Figure 5 shows the algorithm in method *list<Path> find_All_Complete_TR_paths(list<BB> modifications)* that takes a list of modification BBs as the *modifications* parameter and returns a list of BBs as all test requirements. To this end, the method iterates on each modification point denoted as *m_BB* from line 4 to 10. For each *m_BB*, the method first collects all entry paths, i.e., the paths from the entry BB to *m_BB*, and all post-paths from *m_BB* to the exit BB by calling method *find_all_TR_pathsBTW* on line 5 and 6 respectively. After that, the method randomly selects one path, denoted as *pre_path*, from the entry paths on line 7 and appends each post-path to *pre_path* as a complete test requirement, which is added to variable *final_complete_TR_paths*, on lines 8 and 9 respectively. Finally, the method returns all test requirements via variable *final_complete_TR_paths*.

```
1   list<Path> find_all_TR_pathsBTW(BB start_BB, BB end_BB)
2   {   list<Path> paths = findAllPaths(start_BB,end_BB)
4       list<Path> find_TR_set ={};
5       foreach(p: paths)
6       if(Path p ends with end_BB)
7           add p to final_TR_set;
8       return final_TR_set;
9    }
```

Figure 6 Pseudocode for the *find_all_TR_pathsBTW* function

Method *list<Path> find_all_TR_pathsBTW(BB start_BB, BB end_BB)* intends to find all paths between *start_BB* and *end_BB* as shown in Figure 6. So, the method first calls *findAllPaths(start_BB, end_BB)* to collect all paths starting with *start_BB* and ending with *end_BB* or an exit BB. Note that a path ending with an exit BB is a dead path. So, lines 5 – 7 of the method iterate on each returned path to find whether it is a dead path or not. Specifically, line 6 checks whether a path includes *end_BB*. If yes, the method adds the path to variable *final_TR_set* on line

7. After the iteration, line 8 of the method returns *final_TR_set* as result of all test paths between *start_BB* and *end_BB.*

```
1       list<Path> findAllPaths(BB start, BB end)
2       {   if (start==end) return {{start}};
3           else if ( start is a return BB) return {{return_BB}};
4           else if (start is a loop condition)
5           {  Path s1={start};
6              list<Path> s2_false_path = findAllFalsePaths(start.getFalse(),start.getlastWhileCond());
7              list<Path> s2_true_path = findAllTruerPaths(start.getTrue(),start.getlastWhileCond());
8              list<Path> s3_end_path = findAllPaths(start.gtlastWhileCond().getFalse(),end);
9              list<Path> s3_body_path = findBodyPaths(start.getlastWhileCond().getTrue(),start);
10             list<Path> false_paths = s1 ⊕ s2_false_path ⊕ s3_end_path;
11             list<Path> true_paths = s1 ⊕ s2_true_path ⊕ s3_body_path ⊕ s3_end_path;
12             return false_paths U true_paths;
13          } else
14          {  Path s1 = {start};
15             list<Path> s2_true_path = findAllPaths(start.getTrue(),end);
16             list<Path> s2_false_path = findAllPaths(start.getFalse(),end);
17             list<Path> true_paths = s1 ⊕ s2_true_path;
18             list<Paths> false_paths = s1 ⊕ s2_false_path;
19             return true_paths U false_paths;
20          }
21      }
```

Figure 7 Pseudocode for *findAllPaths*.

Method *list<Path> findAllPaths(BB start, BB end)* returns all paths starting with the *start* BB and ending with either the *end* BB, i.e., a successful path, or an exit BB, i.e., a dead path, shown in Figure 7. Method *findAllPaths* is a recursive method. Lines 2 and 3 return a successful and dead path respectively if *start* is the same as *end* or a return BB. Otherwise, if the *start* parameter denotes a loop condition, then there are two kinds of test requirements generated. The first kind of test requirements includes the paths starting with the *start* BB but skipping all body BBs of the loop statement and ending with the *end* BB. The second kind of test requirements only includes the paths 1) starting with the *start* BB, 2) including some BBs in the body of the loop statement, and 3) ending with the *end* BB. So, the calculation of test requirements involves four sections of paths. The first section includes the *start* BB, as shown on line 5 in Figure 7; and the second section includes the true and false paths related to a loop condition, that are given by lines 6 and 7 respectively. The third section includes all paths from the end of a loop statement to the *end* BB, shown on line 8. The fourth section includes all body paths within a loop statement and is given on line 9. After that, the method generates the two kinds of test requirements by combining the paths from the different corresponding sections together via the $\oplus$ sequence concatenation on lines 10 and 12 respectively. Finally, the method returns all the test requirements on line 13. If parameter *start* refers to a condition in an if statement, then the method calls the *findAllPaths* method to deduce the true and false paths on lines 15 and 16 respectively followed by the sequence concatenation with *start* on lines 17 and 18 respectively. Finally, the method returns all the test requirements on line 19.

3.2.2 The Symbolic Execution Engine Module

The symbolic execution engine module is mainly responsible for test case generation based on the test requirements output by the program analyzer module. To achieve this goal, the framework makes two important modifications based on KLEE's implementation. The first modification is to add an attribute in class *ExecutionState* to store all the branch BBs from the entry BB to a current state as shown on line 11 in Figure 8.

```
1.  class ExecutionState{
2.      //The list of path constraints
3.      List<Constraint> PCList;
4.      //the pointer to the current instruction to be executed
5.      //will be modified after exectuion of each instruction
6.      int instructionPointer;
7.      //the stack which stores the functions
8.      stack<function> functionStack;
9.      //newly added attribute stores the basic block lists and other
10.     //attributes related to basic block information
11.     BasicBlockInfo bbinfo;
12.     //other attributes
13.     ...
    }
```

Figure 8 Modification of Class *ExecutionState* in KLEE.

The second modification revises the strategy in the *executeInstructions* method in KLEE's *Executor.cpp* so the framework only keeps the states, i.e., instance of class *ExecutionState* in KLEE, that satisfy the test requirements, pruning those states not satisfying any test requirement.

Before diving into some implementation details about the second modification, we briefly introduce how symbolic execution is carried out in KLEE. First, KLEE maintains the symbolic memory via some special instruction such as *klee_make_symbolic* that is instrumented in some location of LLVM bitcode. Constraints based on symbolic memory information are collected when a branch BB is encountered. Specifically, when a branch referring to symbolic memory is met, KLEE forks to explore all sides of the branch via states, each of which denotes one executable side of the branch, i.e., a solution to symbolic constraints found by a constraint solver.

There are two main methods in this module. The first method is *list<Path> executeInstructions(list<Path> TestReq)* which takes a list of paths as parameter *TestReq* and returns a list of execution paths satisfying parameter *TestReq*. This method is originally implemented by KLEE and we modify it based on our demand in the framework. The second method is *bool hasSameBasicBlocks(list<Path>TestReq, ExecutionState state)* which takes a list of paths and a state via parameters *TestReq* and *state* respectively. The method returns true if the path leading to *state* is a subpath of one path from *TestReq*. Otherwise,−there is no path from *TestReq* that has the path leading to *state* as its subpath−, the method returns false. We illustrate the implementation of these two methods as follows.

```
1  list<Path> executeInstructions(List<Path> TestReq){
2   list<ExecutionState> StatesToBeExecuted.add(entry_BB);
3   list<Path> resultList = []
4   while (StatesToBeExecuted is not empty){
5          ExecutionSate state =  a state chosen from StatesToBeExecuted by KLEE
6          if(!hasSameBasicBlocks(TestReq, state)){
7              StatesToBeExecuted.remove(state);
8              continue;
9          }
10         executeInsruction(state, state.currentInstruction);
11             .....
12         if(state is finished){
13             resultList.add(state.coveredBasicBlocks);
14             StatesToBeExecuted.remove(state);   }
15  }
16  return resultList
17 }
```

Figure 9 Pseudocode for the *executeInstructions* function.

Figure 9 shows the pseudo code of method *list<Path> executeInstructions(List<Path> TestReq)* which takes one parameter, i.e., parameter *TestReq*, which denotes a list of test requirement paths. The method returns a set of test case paths based on a set of test requirements given by parameter *TestReq.* Specifically, a test requirement path corresponds to a test case path and we say the test case path satisfies the test requirement path. By using KLEE's implementation, we add some statements in the method highlighted in bold. First, line 2 of the method initially adds *entry_BB* to variable *StatesToBeExecuted* so KLEE starts the symbolic execution of the entry BB in a program. If *StatesToBeExecuted* is not empty on line 4, the method selects the next state, i.e., variable *state*, via KLEE's strategy on line 5. Note we extend KLEE's class *ExecutionState* via adding a new attribute to retrieve all BBs reaching an object of *ExecutionState*. So, the method calls another method *hasSameBasicBlocks(TestReq, state)* to check whether the path leading to *state* can satisfy one test requirement path given in *TestReq* on line 6. If condition *!hasSameBasicBlocks(TestReq, state)* is true, i.e., the path leading to *state* is *not* a beginning subpath of any path in *TestReq*, then the method removes the current state from *StatesToBeExecuted* and chooses a next step. Otherwise, the method continues to execute *state* using KLEE engine. Finally, the method checks whether *state* denotes a state after executing an exit BB. If yes, the method assigns a set of BBs leading to *state*, as one path, to variable *resultList* on line 13 and then removes *state* from *StatesToBeExecuted* on line 14. If *StatesToBeExecuted* is empty on line 4, line 16 of the method returns *resultList*, i.e., a set of paths, as a set of test case paths satisfying all test requirement paths given in parameter *TestReq*.

```
1. bool hasSameBasicBlocks(list<Path> TestReq, ExecutionState state){
2.     list<Path> state_path = state.coveredBasicBlocks;
3.     for(tr_path in TestReq)
4.         if(tour_with_sidetrip(tr_path, state_path)
5.             return true;
6.     return false
7. }
```

Figure 10 Pseudocode for the *hasSameBasicBlocks* function.

Figure 10 shows the pseudo code of method *hasSameBasicBlocks (List<Path> TestReq, ExecutionState state)*. The method returns true if the path leading to parameter *state* is a beginning subpath of one path in parameter *TestReq*. To this end, the method first stores the path leading to *state* in variable *state_path* and then iterates each test requirement starting on line 3. For each test requirement, stored in variable *tr_path*, the method calls method *tour_with_sidetrip(tr_path, state_path)* to check whether the test case, i.e., *state_path*, is a beginning subpath of *tr_path*. Here we adopt the concept of a tour with sidetrip [29] to ensure the subpath relationship. If yes, the method returns *true* immediately on line 5. Otherwise, the method checks the next path from *TestReq*. If no path from *TestReq* includes *state_path* as its beginning subpath, then the method returns *false* on line 6.

The framework calls the *executeInstructions* method on an old version and a patched version individually. For the efficiency, we only choose the preceding modification BBs instead of all modification BBs as an argument when calling the *executeInstructions* method on an old version. A modification BB *x* is a preceding modification BB if there exists *y* such that *x* precedes *y*. A modification BB *x* precedes another modification BB *y* if *x* occurs in a path from the entry BB to *y*. A reason for skipping *y* is that *y* is considered by the paths from *x* to the exit BB. When calling *executeInstructions* on a new version, the framework uses the entry path of all test requirements as an argument. Thus, for each test requirement, there is no difference between the subpaths in two versions before a modification BB. However, the framework explores all execution post-paths in a patched version after the modification BB to reveal as many behavioral differences between two versions as possible. Last, the framework uses the *state* information provided by KLEE to collect the path constraint and calls the Z3 solver to find a concrete test case.

Since the symbolic execution engine module attempts to demonstrate behavioral difference between two versions of a program, it explores as many execution paths of a patched version as possible by generating new test cases besides the ones in a test suite. Therefore, the module can find some types of run-time errors such as a null/out-of-bounds memory reference and division by zero thanks to KLEE. We call a new test case a failing test case if a patched version fails the test case due to the run-time error detected by the module which also marks the new test case as a failing test case.

3.2.3 The Test Report Evaluator Module

As an initial step toward the confidence calculation about the claim of how a patched version fixes the errors in a faulty version, the test report evaluation module evaluates each leaf claim by checking each test report, i.e., the supporting evidence node. Here, the two-tuple format, i.e., (*dec(A)*, *conf(A)*), is used to evaluate a claim node as well as an evidence node in an assurance case as an approximation of a human's certification decision. In the two-tuple format representing confidence of a node *A*, *dec(A)* denotes certifier's confidence value on *A* and *conf(A)* represents the confidence about how a certifier derives the *dec(A)* value. Figure 11 shows all the four values that can be taken by *dec* and *conf* respectively. When applying the D-S theory, we convert these values to a numerical value. For instance, the four *dec* values, i.e., *acceptable, tolerable, opposable,* and *rejectable,* correspond to 1,0.67,0.33, and 0 respectively. Likewise, the four *conf* values, i.e., *for sure*, *with high confidence, with low confidence,* and, *lack of confidence*, are represented by 1,0.67,0.33, and 0 respectively.

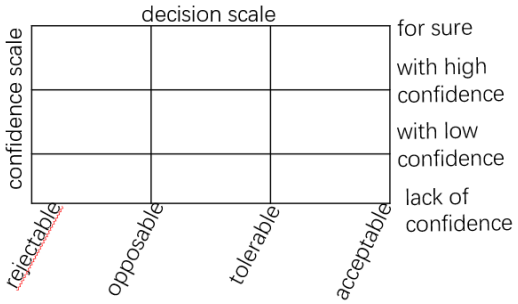


Figure 11 A two-tuple confidence value for a claim node.

Next, we illustrate how a two-tuple value is derived for an evidence node based on a test report. Note that a test report shows how a test case is executed on both versions, i.e., a faulty version and a patched version. Moreover, some additional test cases are generated by the symbolic execution module. Derivation of a two-tuple value for a test report is only performed on newly generated test cases. A *distance classification* value is generated based on the distance between two paths executed by the two versions, called execution paths. There are four values to classify a test case by means of the distance between two execution paths in two versions, i.e., *very close*, *close*, *not close*, and *far away*. An *observation* value consists of a classification value and a classification confidence value. A classification value shows how the framework classifies whether a test case is a passing one or failing one as well as the confidence. Further, there are four values that can be assigned to a classification confidence value, i.e., *observed*, *Calculated I*, *Calculated II*, and *Calculated III*, once a new test case is deemed as a passing one or a failing one. The classification confidence degree decreases from the *observed* value, the highest value, to the *Calculated III*, the lowest value. Once a distance classification value and an observation value are derived for a new test case, a two-tuple value is derived, and more details will be explained later.

Now, we discuss how two execution paths are classified. If a new test case is marked as a failing test case by the symbolic execution engine module, then the following classification process is skipped. Otherwise, we employ the longest common subsequence between two paths to measure their distance. A longest common subsequence (LCS) is the longest sequence between two

execution paths that can be obtained from the two paths by deleting some elements. After finding a longest sequence, we normalize the length of LCS into a value between 0 and 1 using the following formula where a and b are two execution paths and |x| denotes the length of path x.

$$distance(a,b) = 1 - \frac{|LCS(a,b)|}{\max(|a|,|b|)} \quad (1)$$

Figure 12 illustrates how the framework derives a distance classification value for a test case based on its two execution paths in two versions. Specifically, the framework iterates on each test case by checking two execution paths covered by the test case, on the two versions. Here, the framework employs three parameters to measure the distance between two execution paths, i.e., *x1*, *x2*, and *x3*. Initially, the framework calculates a distance value between two paths and then assigns a distance classification value based on the category into which the distance value falls.

```
foreach newly generated test case denoted as t do
   calculate the distance between the two paths based on t running on the two versions
   mark t as very close based on the value [0,x1]
   mark t as close [x1,x2]
   mark t as not close [x2,x3]
   mark t as far away [x3,1]
```

Figure 12 Pseudocode for showing how a distance result is derived.

Figure 13 shows how the framework derives an observation value, indicating how the framework derives a classification of a test case as well as its confidence about the classification. Specifically, the framework calculates the distances between a path executed by a newly generated test case and all existing known execution paths; and then applies the classification of the closest known execution path as the one of the newly generated test cases. For instance, if a path executed by a new test case is closest to a passing execution path, then the test case is classified as passing. Likewise, the test case is classified as failing if its execution path is closest to a failing execution path. Next, the framework calculates a confidence value about how it classifies a test case. In this case, the framework takes parameter *X*, used to choose the top X% of closest existing execution paths. Among the selected execution paths, if more than 66% of execution paths have the same category as the closest execution path, then the framework assigns *Calculated I*. As such, the other two values, i.e., *Calculated II* and *Calculated III*, can be derived similarly based on different percentages of the selected execution path, as shown in Figure 13.

```
foreach test case denoted as t in a test suite as well as all new generated test case do
   if a test case, denoted as tc, is new then
      calculate the distances between tc and all passing and failing test paths and the closet path decides tc's
failing/passing category. Mark tc as calculated I/II/III based on the top X% closet paths
         I: .Among the X% closet paths, >66% paths from the same category of the test path deciding tc's category
         II: Among the X% closet paths, [.33,.66] paths from the same category of the test paths deciding tc's category
         III: Among the X% closet paths, [0,.33] paths from the same category of the test paths deciding tc's category
   else
      mark each test case from a test suite as observed
```

Figure 13 Pseudo Code to show an observation value is derived.

After a distance classification value and an observation value are deduced for a newly created test case, the framework employs *Table 2* to derive a two-tuple value by the classification of the closest execution path/test case. For instance, during the calculation of an observation value for a new test case, if a passing test case is found as a closest path to the newly created test case by Figure 13, then the row "Passing in old version" is used to derive a two-tuple value. A general idea behind generation of a two-tuple value is that both a faulty version and its correct patched version have the similar behavior, i.e., similar execution paths, on a passing test case while the two versions have different behavior on a failing test case [9]. Thus, a newly created test case has a higher value if it is deemed *passing* and the two execution paths running on the two versions by the test case are quite similar. For instance, assume that a new test case is deemed as *passing* with the *Calculated I* confidence. If the two execution paths of the test case are *very close*, then the framework assigns (*acceptable, with high confidence*) as a two-tuple value for its corresponding test case report. But, if the two execution paths are *far away*, then (*rejectable, with high confidence*) is assigned as a two-tuple value for the test case report.

Table 2 The Two-Tuple Value Generation.

		Very close	close	No Close	Far Away
Passing in old version	observed	(acpt, for sure)	(tolerable, for sure)	(opposable, for sure)	(rejectable, for sure)
	Calculated I	(acpt, w/high confidence)	(tolerable, w/high confidence)	(opposable, w/high confidence)	(rejectable, w/high confidence)
	Calculated II	(accpt, w/low confidence)	(tolerable, w/low confidence)	(opposable, w/low confidence)	(rejectable, w/low confidence)
	Calculated III	(acpt, lack of confidence)	(tolerable, lack of confidence)	(opposable, lack of confidence)	(rejectable, lack of confidence)
Failing in old versions	observed	(rejectable, for sure)	(opposable, for sure)	(tolerable, for sure)	(acpt, for sure)
	Calculated I	(rejectable, w/high confidence)	(opposable, w/high confidence)	(tolerable, w/high confidence)	(acpt, w/high confidence)
	Calculated II	(rejectable, with low)	(opposable, w/low confidence)	(tolerable, w/low confidence)	(accpt, with low)
	Calculated III	(rejectable, lack of conf)	(opposable, lack of confidence)	(tolerable, lack of confidence)	(acpt, lack of conf)

3.2.4 The Assurance Case Generator and Calculator Modules

The assurance case generator module takes as input a GSN variable mapping file from the previous three modules. A GSN variable mapping file presents the relationship from variables in the safety pattern to the artifacts produced by two versions of a program. Applying the safety pattern and a GSN variable mapping file, the AC generator constructs an assurance case using GSN. In this framework, we employ the Structured Assurance Case Metamodel (SACM) [22] as a metamodel of an assurance case represented in GSN. Therefore, the framework can support any assurance case in the XML Message Interface (XMI) format once it is compatible with the SACM.

The implementation consists of three parts. The first part parses an Extensible Markup Language (XML) file denoting the safety pattern and a GSN variable mapping file. As a result, this

part converts to a tree-like structure denoting the safety pattern as well as establishes a mapping relation between variables in the safety pattern and artifacts retrieved from the GSN variable mapping file. Here, a tree-like structure of a safety pattern is based on our GSN metamodel (see in Appendix C). A reason for designing a new GSN metamodel is that we simplify the GSN metamodel provided by the SACM to concentrate on the elements of an AC related to the purpose of the framework. The second part reads the tree-like structure as well as the mapping relations between variables and their artifacts to generate an assurance case in GSN by means of the visitor pattern [30]. Likewise, the third part applies the visitor pattern to calculate the confidence of a generated Assurance Case (AC). A class diagram showing the overall structure is given in Figure 14.

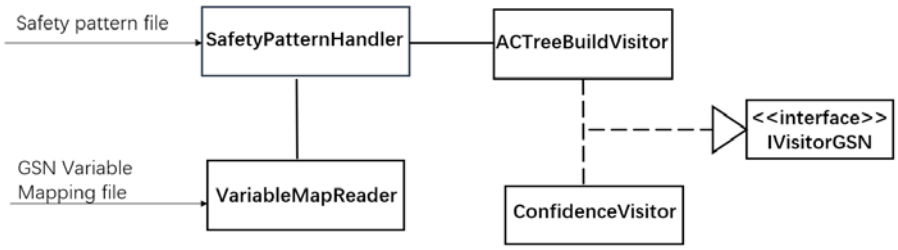


Figure 14 Flow diagram of the AC generator and Calculator modules.

Class *SafetyPatternHandler* is implemented to parse an XML file and generate a tree-like structure for the safety pattern as well as establishes a mapping relation between all variables and their artifacts retrieved from a GSN variable mapping file. Specifically, the class employs SAX to read an XML file denoting the safety pattern. The class iterates on each element in the XML file. If the class finds a relevant element such as a goal node and strategy node, it creates a corresponding object of a class in our GSN metamodel. Furthermore, class *SafetyPatternHandler* calls class *VariableMapReader* that parses a GSN variable mapping file to retrieve artifacts based on a variable in the current relevant element and then adds a mapping relation between a node in the safety pattern, a variable and its corresponding artifacts to a mapping structure. Namely, the mapping structure (*pattern node id*, (*variable*, *artifact list*)) is implemented by means of class *HashMap* in Java, where *pattern node id* represents a node in the safety pattern, *variable* denotes a variable in a node in the safety pattern and *artifact list* contains the artifacts to replace *variable* in a generated AC.

We next employ the visitor pattern to generate an assurance case. Namely, according to the structure of the visitor pattern, the object structure is our GSN metamodel and the AC generation is given by a visitor class *ACTreeBuildVisitor* that implements the *IVisitor_GSN* interface. For each concrete class in our GSN metamodel, class *ACTreeBuildVisitor* has a visit method. For instance, our GSN metamodel has class *Goal* and therefore class *ACTreeBuildVisitor* has method *visitGoal(..)*. The visitor is applied to a tree-like structure representing a safety pattern, to generate an AC. Due to space, we only show the pseudocode of *visitGoal(Goal goal)* in Figure 15. On line 4, the method creates an instance of class Goal in a generated AC via the *createGoal(..)* method. Creation of an instance depends on not only variable *goal* denoting the goal node in the safety pattern but also the aforementioned mapping structure (*pattern node id*, *(variable*, *artifact list)*) by

calling *goal.getMappingRel()*. Next, the method checks whether the *goal* node is a root node in the safety pattern or not on line 5.

- If yes, line 6 of the method adds the newly created instance of *Goal* to variable *newGoals* which is used to connect to future child nodes via some relation in a generated AC when a child node of *Goal* is visited. Finally, the method sets the newly created instance to a root node in the new AC on line 7.

- Otherwise, the method connects the newly created node to its corresponding parents. To this end, the method iterates on each node denoted by variable *existing_goal* in variable *newGoals* to check whether the node referenced by *existing_goal* is a parent of the newly created instance referenced by variable *g* via calling *isParent(eixisting_goal, goal)* on line 10. If yes, the method iterates on each incoming relation to *Goal* based on the safety pattern for setting up a new incoming relation. For each incoming relation referenced by variable *rel*, line 12 of the method creates a new incoming relation in a generated AC and the new relation connects *g* and *existing_goal* together via calling method *create_link(g,existing_goal,rel)*. A type of a new incoming relation such as the *supporteBy* and *InContextOf* links depends on the corresponding relation in the safety pattern. After processing *goal* and its incoming relations, lines 16-18 denote one iteration of continuously visiting an outgoing relation coming out of *goal*. Specifically, line 17 of the method visits one outgoing relation by calling the *accept(this)* method. In this way, the construction of a new AC continues.

```
1.  Class ACTreeBuildVisitor implements IVisitorGSN{
2.     private list<ArgumentElement> newGoals;
3.     void visitGoal(Goal goal){
4.          Goal g = new createGoal(goal, goal.getMappingRel());
5.          if (goal is a root){
6.                    newGoals.add(g);
7.                    g.setacRoot();
8.          }else{
9.                    foreach(existing_goal: newGoals){
10.                        if(isParent(existing_goal, g)
11.                        {     foreach(rel: goal.getSourceOf())
12.                                        create_link(g, existing_goal, rel)
13.                        }
14.                   }
15.         }
16.         foreach( child_rel: goal.getTargetOf()){
17.                   child_rel.getTargetOf.accept(this);
18.         }
19.    }
20.    ……
    }
```

Figure 15 Pseudocode of visitGoal method for AC generating visitor.

After an AC is constructed, the calculator module is called to calculate the confidence for an assurance case. Here, we largely follow Wang et al's formulation for the D-S theory calculation

[25]. For a claim A, a frame of discernment Ω_P is $\{A, \overline{A}\}$, where $\overline{A}$ denotes logical negation of A. The mass $m^{\Omega_A}(A)$ shows the degree of belief committed to the hypothesis that truth lies in A. When applying the D-S theory, confidence of a claim A is denoted as a three-tuple (*bel(A), dis(A), uncer(A))* representing belief, disbelief, and uncertainty of A, respectively. The three-tuple of a claim is thus defined as follows:

$$\begin{cases} \text{bel}(A) = m^{\Omega_A}(A) = g_A \\ \text{dis}(A) = \text{bel}(\overline{A}) = m^{\Omega_P}(\overline{A}) = f_A \\ \text{uncer}(A) = m^{\Omega_A}(\Omega_A) = 1 - g_A - f_A \end{cases} \quad (2)$$

A confidence model [23] consists of only claims and evidence nodes that are connected to each other via the *supportedBy* link. An argument is composed of a claim as a conclusion and the corresponding sub-claims as predicates via its *supportedBy* links. The confidence calculation of a claim depends on the nature of the argument relating the claim to its sub-claims; different types of assessment parameters are used for showing the nature of an argument. The first assessment parameter is the type of an argument: either a dependent or redundant argument.

A redundant argument means the contribution of one sub-claim to support its parent claim does not depend on another (i.e., sibling) sub-claim. For example, the argument-claim A: "the system is acceptably safe" is supported by claim B: "the system is passed by verification" and claim C: "the system is passed by testing"- is redundant since two different techniques, B and C supports A to some degree without being dependent on the other.

A dependent argument means that the contribution of a sub-claim for supporting its parent claim has some degree of overlap with another sub-claim. For instance, the following argument-claim A: "The system is acceptably safe" is supported by claim B "the test process is sound" and claim C "the test results are correct" - is a dependent argument since the test results given by C to support claim A depend on the test process given by claim B. The second assessment parameter is the completeness of an argument, denoted as *v*, referring to a degree value between 0 and 1 and showing a scenario where its claim as a conclusion cannot be fully derived from all its sub-claims as predicates. For instance, for the above dependent argument, unless the validity of the claim made by A can be verified, we cannot completely guarantee the claim of A via any testing approach and so v can only have a value less than 1.

Another assessment parameter relates to the degree of correspondence of all sub-claims, denoted as *co*, which collectively sum to a value between 0 and 1 to capture the contributions of all sub-claims to their parent claim. All these assessment parameters are based on an argument. The final parameter is a disjoint contributing weight that differs from the other three assessment parameters because its value is set based on a claim instead of an argument. A disjoint contributing weight of a claim, say A, denoted as w_A, denotes a degree value between 0 and 1 showing how much the claim independently contributes to the belief/confidence of its parent claim along the argument.

Once the type of an argument is set, confidence of a claim can be calculated using the D-S theory. In this project, the nature of the type of all arguments in the safety pattern is dependent. Then the confidence of a claim A via the three-tuple is given by the following formula:

$$\begin{cases} \text{bel(A)} = \text{v}[(co)\prod_{\text{i=1}}^{\text{n}}\text{g}_\text{i} + \sum_{\text{i=1}}^{\text{n}}\text{g}_\text{i}\text{w}_\text{i}] = \text{g}_\text{A} \\ \text{dis(A)} = \text{v}[(co)[1 - \prod_{\text{i=1}}^{\text{n}}(1 - \text{f}_\text{i})] + \sum_{\text{i=1}}^{\text{n}}\text{f}_\text{i}\text{w}_\text{i}] = \text{f}_\text{A} \\ \text{uncer(A)} = 1 - \text{g}_\text{A} - \text{f}_\text{A} \end{cases} \quad (3)$$

where w_i (i=1,2,…,n) denotes a disjoint contributing weight of the i-th sub-claim to support its parent claim. The only exception to formula 3 is the calculation of a leaf claim. In this case, the framework just assigns a confidence value of an evidence node to its supporting leaf claim. Since a redundant argument with n sub-claims is not applied to the safety pattern, due to space constraints, we omit the formula in this report that can be otherwise found in [25].

Note that the two-tuple format is used to evaluate each leaf claim as an approximation of a certifier's certification value. Since the two-tuple and three-tuple formats reflect two different perspectives on the confidence of a claim, a type conversion between the two formats is necessary for different purposes. For instance, a certifier's evaluation on a leaf claim via a two-tuple is converted to a three-tuple so the D-S theory can be carried out. Likewise, a three-tuple of a root claim is converted to a two-tuple after the calculation to obtain certifier's final evaluation. Formulas (4) and (5) show the conversions between the three-tuple and two-tuple formats of confidence of a claim respectively.

$$\begin{cases} bel(A) = conf(A) \times dec(A) \\ dis(A) = conf(A) \times (1 - dec(A)) \\ uncer(A) = 1 - bel(A) - disb(A) \end{cases} \quad (4)$$

$$\begin{cases} conf(P) = bel(P) + disb(P) \\ dec(P) = bel(P) \,/\, (bel(P) + disb(P)), \;\; if\; bel(P) + disb(P) \neq 0 \\ dec(P) = 0, \qquad if\; bel(P) + disb(P) = 0 \end{cases} \quad (5)$$

Class *ACTreeBuildVisitor* also transforms a generated assurance case into a confidence model, containing only goal and solution nodes. After that, class *ConfidenceVisitor* takes over to carry out the calculation by adopting the visitor pattern. For instance, the *visitGoal(goal)* method is implemented to return a three-tuple value as a confidence value of the *goal* node, shown in Figure 16. Thus, line 7 of the *visitGoal(goal)* method checks whether *goal* is a leaf or not after initialization of some variables from line 3 to line 6.

- If yes, the method calls the supporting solution node and then converts from a two-tuple value returned by the solution node to a three-tuple value on line 8.
- Otherwise, the method iterates on each *goal*'s outgoing link referenced by variable *link* to get a three-tuple value of a corresponding child node from line 10 to line 20. Specifically, the method obtains a weight value for each link via calling the accept method, i.e., *link.accept(this)* on line 11. Next, the method gets a three-tuple value of a child node via *link*

and then updates the three-tuple value for the current *goal* from line 13 to line 16 based on formula 3. After visiting all outgoing links, the method manufactures a result to variable *result* from line 18 to line 20.

- Finally, the method returns the result on line 23.

```
1.  Class  ConfidenceVisitor implements IVisitorGSN {
2.     Object visitGoal(Goal goal){
3.          ThreeTuples result = new ThreeTuples();
4.          ThreeTuples childTuple = null;
5.          double weight = 0.0;
6.          double belSum = 0.0; belProd = 1.0; disProd = 1.0; disSum = 0.0;
7.          if (goal is a leaf goal) {
8.                        result=convert(goal.getSolutionNode.accept(this));
9.          } else {
10.                    foreach (link : goal.getTargetOf()) {
11.                              weight=link.accept(this);
12.                              childTuple = (ThreeTuple) link.getSource().accept(this)
13.                              belSum = belSum + (weight * childTuple.getbelief());
14.                              belProd = belProd * childTuple.getbelief();
15.                              disSum = disSum + (weight * childTuple.getdisbelief());
16.                              disProd = disProd * (1 – childTuple.getdibelief());
17.                    }
18.                    result.setdisbelief(v * ((co * (1 - disProd)) + disSum));
19.                    result.setbelief (v * ((co * belProd) + belSum));
20.                    result.setuncertainty (1 – (result.getbelief() + result.getdisbelief()));
21.         }
23.        return result;
24.   }
        .....
```

Figure 16 Pseudocode of visitGoal method for confidence visitor.

However, if a new failing test case is found by the symbolic execution engine module, then the calculator module does not apply the D-S theory to calculate the confidence of an AC. Instead, the module assigns a two-tuple value (0,1) to the root claim of the AC, indicating that a patched version is totally incorrect due to the failing test case.

4. Results and Discussions

4.1 Setup and Results

The framework was tested against the *IntroClass* benchmark [31], which consists of 6 programming assignments from an introductory C programming course at UC Davis. While the projects in the *IntroClass* benchmark have small sizes, these projects feature comprehensive and controlled evaluation factors affecting many APR approaches. To test our framework's capability to evaluate an APR approach, we divide all the patches into three groups, i.e., a correct patch, an incorrect patch, and a plausible patch. A correct patch is the program that correctly implements its requirements. An incorrect patch refers to the patch that fails at least one test case in a test suite. A plausible patch is the one that passes all test cases in a test suite. But a plausible patch may not be a correct patch due to the limited number of test cases in a test suite. If we cannot find a correct patch, then we manually implement one based on the plausible patches we consider. For each project, we choose three patches from the *IntroClass* benchmark that include one correct patch and

two plausible patches. To fully investigate the capability of the framework, we consider the following research questions:

1. Can some types of run-time errors in a plausible patch be revealed by newly generated test cases?
2. Given a correct fault localization, can the framework deduce a high value for a correct patch compared with a plausible patch?
3. Can a correct fault localization enable the framework to deduce a high value for a correct patch compared to a plausible patch?

To answer the first question, we concentrate on some types of run-time errors such as memory errors and division by zero. After checking all six projects, we find that the checksum project has an array in its implementation and one of the two plausible patches do have an out-of-bounds memory reference. Running on the checksum project, the framework does successfully reveal an out-of-bounds memory reference on line 7 in Figure 17 by exploring additional behavior of one plausible patch after generating a new test case, i.e., when the length of the *instring* array variable is less than parameter *len*. This confirms that using symbolic execution can overcome the obstacle due the constraint imposed by a test suite. Once a failing test case is found, the framework assigns a two-tuple value (0,1) to the root claim of the assurance case, indicating the patched version is an incorrect patch.

```
1. char checksum(char instring[], int len){
2.     char checksumchar;
3.     int checksum_summation = 0;
4.     for(int i=0; i < len; i++)
5.     // correct: for(int i=0; instring[i] != '\0'; i++)
6.     {
7.         checksum_summation = checksum_summation + 100;
8.     // correct & plausible1: checksum_summation = checksum_summation + (int)
   instring[i];
9.     }
10.     checksum_summation %= 64;
11.     checksum_summation += 32;
12.     checksumchar = (char)checksum_summation;
13.     return checksumchar;
14. }
```

Figure 17 The checksum project with a correct patch and plausible patch

For the second and third research questions, we choose v=0.7 and co=0.1, and let all child nodes in a confidence model be equally weighted. For a loop statement such as a while statement, we set the execution time of a loop body to 2 due to efficiency consideration. Furthermore, we choose all paths from an entry BB to a modification BB to see whether a randomly chosen path affects a final calculation result. Since the checksum project has one plausible patch with a run-time error, we exclude it from the consideration; so five of the six projects have two plausible patches and the checksum project has only one plausible patch. We found there are three projects (i.e., the median, smallest, and syllables projects) in which some test requirements have multiple entry paths, while the other three projects do not have.

Table 3 Comparison between the projects based on the same fault localization.

project	total	better	equal	worse
digits	2	0	2	0
checksum	1	0	1	0
smallest	6	4	2	0
median	4	4	0	0
grade	2	0	0	2
syllables	12	0	12	0
Summary	27	8	17	2

To address the second research question, Table 3 shows the results for all five projects and we find there are only two plausible patches that have a better final two-tuple value than their counterpart, claiming about a correct patch. These two plausible patches are all in the grade project as shown in Figure 18. After checking the two patches, we find that both patches have a fault localization on line 11, i.e., the condition of the third if statement, resulting in the true branch of the if statement becoming dead code after a repair for both versions. Consequently, there is no new test case to be generated for both plausible patches while a new test case is generated for a correct patch. However, the new test case, e.g., 73, covers the same execution path exercised by a failing test case, e.g., 65, in the test suite and thus it is deemed as *failing* with the *Calculated II* as the confidence. Furthermore, the framework expects that a failing test case behaves differently between two versions but in fact the two execution paths are identical. Thus, a low two-tuple value is assigned to the test case report. As a result, the assurance cases claiming about the plausible patches outperform their counterpart claiming about the correct patch.

```
1. char grade(int g){
2.     int A=90, B=80, C=70, D=60;
3.     if(g >= A){
4.         return 'A';
5.     }
6.
7.     else if(g >= B){
8.         return 'B';
9.     }
10.
11.     else if(g >= D){
12.     // correct: else if( g >= C) {
13.     // plausible1: else if(g >= B) {
14.     // plausible2: else if(g >= A){
15.         return 'C';
16.     }
17.
18.     else if(g >= D){
19.         return 'D';
20.     }
21.
22.     else return 'F';
23. }
```

Figure 18 The grade project with three patches

We further study the projects where a final two-tuple value of the assurance cases claiming about a plausible patch is equal to the counterpart claiming about a correct patch since we expect an assurance case claiming about a correct patch should outperform its counterpart. For the digits

project, we find that the fault localization, on line 12 in Figure 19, is right before the return statement on line 17 while the other return statement on line 24 is not reachable from line 12 in the three patches. Thus, there is only one test requirement that includes the while statement as part of its entry path. Due to the loop limit (=2) set for a while statement, there is only one new test case to be generated to execute the while body twice. Unfortunately, for the newly generated test case, there is no difference among the execution paths in the three patched versions. So, the three assurance assurances denoting a correct patch and two plausible patches have the same final two-tuple value.

```
1. int digits(int num){
2.     if (num==0){
3.         printf("%d\n",num);
4.         return 0;
5.     }
6.     if (num<0){
7.         num= -num;
8.         while (num > 10) {
9.             printf("%d\n", num % 10);
10.             num/=10;
11.         }
12.         num = num - num;
13.         //correct:  num = num - 2 * num;
14.         //plausible1: num = num - 2;
15.         //plausible2: num = -1;
16.         printf("%d\n", num);
17.         return 1;
18.     }
19.     else{
20.         while (num != 0) {
21.             printf("%d\n", num % 10);
22.             num/=10;
23.         }
24.         return 2;
25.     }
26. }
```

Figure 19 The digits project with three patches.

```
1. char checksum(char instring[], int len){
2.     char checksumchar;
3.     int checksum_summation = 0;
4.     for(int i=0; instring[i] != '\0'; i++)
5.     {
6.         checksum_summation = checksum_summation + 100;
7.        // correct: checksum_summation = checksum_summation+(int)instring[i];
8.        // plausible2: checksum_summation=checksum_summation+(int)instrin[1];
9.     }
10.     checksum_summation %= 64;
11.     checksum_summation += 32;
12.     checksumchar = (char)checksum_summation;
13.     return checksumchar;
```

Figure 20 The checksum project with a correct patch and one plausible patch.

The checksum project, as shown in *Figure 20*, has the fault localization on line 6, within the for loop. Thus, a new test case is generated for both versions. Unfortunately, the new test case covers the same execution path for both versions, resulting in no behavioral difference between two

versions. As a result, there is no difference for the two-tuple values for both test reports and this explains why both assurance cases have the same final two-tuple value. Likewise, the syllables project has the fault localization on a line within a true branch of an if statement. Thus, while there are many newly generated test cases to satisfy the test requirements, there are no differences between the execution paths, running the same test case on the three versions and this explains why the assurance case claiming about a correct patch has no better two-tuple value than its counterpart.

Table 4 Comparison between the projects based on different fault localizations.

project	total	better	equal	worse
digits	2	0	2	0
checksum	1	1	0	0
smallest	6	6	0	0
median	3	0	3	0
grade	2	0	1	1
syllables	12	12	0	0
Summary	26	19	6	1

To address the third research question, Table 4 shows the results for all five projects and we find only one plausible patch has a better final two-tuple value than its corresponding correct patch. We find that the patch is in the grade project which also results in dead code, which is similar to the analysis for the second research question.

```
1. int median(int a, int b, int c){
2.     if(((a>=b)&&(a<=c))||((a<=b)&&(a>=c))){
3.         return a;
4.     }
5.     else if(((b>=a)&&(b<=c))||((b<a)&&(b>c))){
6.     //correct: else if(((b>=a)&&(b<=c))||((b<=a)&&(b>=c))){
7.     //plausible1: else if(((b>=a)&&(b<=c))||((b<=a)&&(b>c))){
8.     //plausible2: else if(((b>=a)&&(b<=c))||((b<a)&&(b>=c))){
9.         return b;
10.     }
11.     else if (((c>a)&&(c<b))||((c<a)&&(c>b))){
12.         return c;
13.     }
14.     return -1;
15.     //plausible1: return 0;
16.     //plausible2: return 0;
17. }
```

Figure 21 The median project with three patches.

Then, we investigate the projects in which the assurance case claiming about the correct patch does not outperform the counterpart claiming about a plausible patch. First, we study the median project, shown in *Figure 21*, which has the same final two-tuple value among all the three patches. While the fault localization of each plausible patch is slightly different from the counterpart of the correct patch, there is only one test case generated for all three patches. A reason to only generate one test case is that either the execution path is covered by an existing test case in the test suite or the execution path is not feasible. Furthermore, the newly generated test case covers the same

execution path for all the three patches. Therefore, there is no difference between the two-tuple values for the newly generated test case report of each patch, resulting in the three assurance cases having the same final two-tuple value.

```
1. int digits(int num){
2.     if (num==0){
3.         printf("%d\n",num);
4.         return 0;
5.     }
6.     if (num<0){
7.         num=-num;
8.         while (num > 10) {
9.             printf("%d\n", num % 10);
10.             num/=10;
11.             //plausible1: num = 0;
12.             //plausible2: num = num / 10 - 2;
13.         }
14.         num = num - 2;
15.         //correct: num = num - 2 * num;
16.         printf("%d\n", num);
17.         return 1;
18.     }
19.     else{
20.         while (num != 0) {
21.             printf("%d\n", num % 10);
22.             num/=10;
23.         }
24.         return 2;
25.     }
26. }
```

Figure 22 The digits project with three patches.

The digits project, shown in *Figure 22*, has the same value for the *dec* element in the final two-tuple value of the ACs claiming about the three patches, but the *conf* value of the assurance case claiming about the correct patch is better than the counterparts of the assurance cases claiming about the two plausible patches. Both patches have an earlier fault localization than that of the correct patch, but this difference does not produce any new test requirement and all three versions have only one test requirement, i.e., a path from the modification to line 17. However, to satisfy the only test requirement, the framework generates a new test case, i.e., -304, for all three patches due to the limit set for a while statement. The new test case covers the same execution paths in the faulty version and the correct patch. But unfortunately, the framework assigns a very low two-tuple value (0,0) to the test case report in the correct version since the test case is deemed *failing* with the value *Calculated III* as the confidence but the two same execution paths in both versions are classified as *very close*. However, the framework assigns a better two-tuple value (0.33,0) to the test case report in the first plausible patch since the two execution paths in the correct and first plausible versions are slightly different. While assigning the low two-tuple value (0,0) to the test case report that covers the same execution path in the second patched version, the framework generates another test case, i.e., -112, due to the limit set for the while statement. The new test case is deemed *failing* with the value *Calculated III* as the confidence but the two execution paths in the correct and second plausible versions are slightly different and thus receives (0.33,0) as a two-tuple value for the test case report. Note that all the other test case reports have (1,1) as their two-tuple value. When

converting to a three-tuple value, all *dis* values are zero. This leads to the fact that the *dis* value of the final three-tuple value for the root claim has zero too when applying the D-S theory. Thus, all three assurance cases have one as the value for *dec*.

4.2 Discussion

In this project, we propose a novel framework to evaluate a claim made by every APR approach that a patched version has really fixed a faulty version from the certification perspective. More important, the framework quantifies a confidence value for the claim by means of an assurance case in GSN. Thus, various APR approaches can be explicitly compared and thus evaluated. Furthermore, the framework can find many run-time errors in a patched version such as a null/out-of-bounds memory reference and division by zero thanks to the symbolic execution.

However, there are still several potential threats to validity of the framework. First, due to the efficiency consideration, we adopt the RIPR model to generate the test requirements to study the claim made by an APR. The RIPR concentrates on a modification point of two versions to reveal the difference between them. But, how to find a modification point, i.e., a suspicious statement, in a faulty version is another focus of APR. Obviously, the incorrect detection of a suspicious statement might have an unfavored final calculation for an AC. In this case, we need to further study such a consequence in the future. A more accurate calculation can consider two ACs that are generated by two APR approaches running on the same faulty version instead of one AC.

The second threat is the classification of a new test case. Our classification algorithm is built based on the observation made by many other approaches, i.e., that both versions should behave similarly on the passing test cases but differently on the failing test cases. But this observation is *not* always true according to the projects we have used in this project. For instance, in the grade project, a new test case, e.g., 73, covers the same execution path exercised by a failing test case in the test suite and thus it is deemed as “failing”. In fact, the test case shows a correct execution path in the correct patch that is identical to the one in the faulty version. This leads to the assurance case claiming about a plausible patch outperforming the counterpart claiming about a correct patch. Thus, a better classification algorithm should be further investigated by considering the behavioral difference between a passing test case and a failing test case.

The last threat is the way of individual evaluation of an assurance case. While our original thought was to employ symbolic execution to reveal potential errors in a patched version, due to practical consideration, the framework adopts some strategies to combine the symbolic execution, generation of an AC, and its evaluation together in an efficient fashion. We thus find that the framework will be more useful if it can compare different patches by different assurance cases. Furthermore, we can expand the safety pattern to include some additional finding such as dead code, so we can adopt a different strategy via assigning different weight configuration when employing the D-S theory.

5 Conclusions

In this project, we implement a novel framework to evaluate the quality of a patched version generated by an automatic program repair approach by means of an assurance case. While every APR approach claims that a generated patch fixes the error in a faulty version, plausible patches have been tremendously produced due to some obstacles such as limited number of test cases in a test suite and the test oracle problem, thus impeding the developers' performance in development of a software system. To provide a fair judgement on claims made by every APR approach, the framework initially employs the symbolic execution to reveal difference between two versions of a program to overcome the obstacles facing many APR approaches. Then, the framework applies an assurance case as an argument structure showing how the symbolic-based approach is used to determine the claim on the quality of a patched version and finally uses the D-S theory to evaluate the generated assurance case to achieve our goal.

6 Acknowledgement

We would like to appreciate the great support from the program managers including Patrick Hurley, Steven Drager, Wilmar Sifre, William McKeever, Carl Thomas, and Matthew Anderson from the AFRL. We also thank many teams involved with this program for providing us tremendous help and comments so that we can finish the entire project within one year. Without these helps, the project cannot make such a great progress.

7 References

[1] L. Gazzola, D. Micucci and L. Mariani, "Automatic Software Repair: A Survey," *IEEE Trans. on Software Eng.,* vol. 45, no. 1, pp. 34-67, 2019.

[2] S. Forrest, T. Nguyen, W. Weimer and C. Le Goues, "A genetic programming approach to automated software repair," in *Proceedings of Genetic and Evolutionary Computation Conference (GECCO 2009)*, Montreal, Québec, Canada, 2009.

[3] A. Ghanbari, S. Benton and L. Zhang, "Practical program repair via bytecode mutation," in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2019, July 15-19,*, Beijing, China., 2019.

[4] Z. Qi, F. Long, S. Achour and M. Rinard, "An analysis of patch plausibility and correctness for generate-and-validate patch generation systems.," in *Proceedings of the 2015 International Symposium on Software Testing and Analysis, ISSTA 2015, July 12-17, 2015.*, Baltimore, MD., 2015.

[5] C. Le Goues, T. Nguyen, S. Forrest and W. Weimer, "GenProg: A Generic Method for Automatic Software Repair," *IEEE Trans. Software Eng.,* vol. 38, no. 1, pp. 54-72, 2012.

[6] Q. Gao, H. Zhang, J. Wang and Y. Xiong, "Fixing Recurring Crash Bugs via Analyzing Q&A Sites.," in *Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering, ASE 2015, November 9-13, 2015.*, Lincoln, NE, USA, 2015.

[7] Q. Gao, Y. Xiong, Y. Mi, L. Zhang, W. Yang, Z. Zhou, B. Xie and H. Mei, "Safe Memory-Leak Fixing for C Programs," in *In Proceedings of the 37th International Conference on Software Engineering, ICSE 2015, May 16-24, 2015, Volume 1.*, Florence, Italy, 2015.

[8] 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 34th International Conference on Software Engineering(ICSE 2012)*, Zurich, Switzerland, 2012.

[9] Y. Xiong, X. Liu, M. Zeng, L. Zhang and G. Huang, "Identifying patch correctness in test-based program repair.," in *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, May 27 - June 03, 2018.*, Gothenburg, Sweden., 2018.

[10] M. Martinez and M. Monperrus, "ASTOR: A Program Repair Library for Java," in *Proceedings of the 25th International Symposium on Software Testing and Analysis, ISSTA 2016, Demonstration Track*, Saarbrücken, Germany, 2016.

[11] E. K. Smith, E. T. Barr, C. Le Goues and Y. Brun, "Is the cure worse than the disease? overfitting in automated program repair," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015, August 30 - September 4, 2015*, Bergamo, Italy, 2015.

[12] Q. Xin and S. Reiss, " Identifying Test-Suite-Overfitted Patches through Test Case Generation," in *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis, July 10 - 14, 2017*, Santa Barbara, CA, USA, 2017.

[13] J. Yang, A. Zhikhartsev, Y. Liu and L. Lin Tan, "Better Test Cases for Better Automated Program Repair," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2017, September 4-8, 2017.*, Paderborn, Germany, 2017.

[14] Y. Tao, J. Kim, S. Kim and C. Chang Xu, "Automatically Generated Patches As Debugging Aids: A Human Study," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE'22, November 16 - 22, 2014*, Hong Kong, China, 2014.

[15] J. King, Symbolic Execution and Program Testing, Communications of the ACM, 1996.

[16] N. R. Council, Critical code: software producibilityfor defense, Washington D.C.: Academies Press, 2010.

[17] J. Rushby, "The Interpretation and Evaluation of Assurance Cases," Computer Science Laboratory, SRI International, SRI-CSL-15-01, Menlo Park, CA, 2015.

[18] T. Chowdhury, C.-W. Lin, B. Kim, M. Lawford, S. Shiraishi and A. Wassyng, "Principles for Systematic Development of an Assurance Case Template from ISO 26262," in *Proceedicknгs of 2017 IEEE International Symposium on Software Reliability Engineering, Industry Track*, Toulouse, France, 2017.

[19] J. Cleland-Huang and R. Lutz, "Traceability Support For Evolving Safety Assurance Cases," Air Force Research Laboratory's Safe & Secure Systems and Software Symposium (S5) 2017, Dayton, Ohio, 2017.

[20] E. Denney and G. Pai, "Tool support for assurance case development," in *Automated Software Engineering*, Springer , to appear, 2018, p. TBA.

[21] P. Graydon, J. C. Knight and E. Strunk, "Assurance Based Development of Critical Systems," in *Proceedings of the 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, Edinburgh, UK, 2008.

[22] R. Hawkins, I. Habli, T. Kelly and J. McDermid, "Assurance Case and Prescriptive Software Safety Certification: A Comparative Study," *Journal of Safety Science,* vol. 59, no. 11, pp. 55-71, 2013.

[23] C. Lin, W. Shen and S. Drager, "A Framework To Support Generation and Maintenance of An Assurance Case," in *27th IEEE International Symposium on Software Reliability Engineering 2016 (ISSRE 2016)* , 2016.

[24] D. Meacham, J. Michael, M.-T. Shing and J. Voas, "Standards interoperability: Applying software safety assurance standards to the evolution of legacy software," in *Proceedings of IEEE International Conference on System of Systems Engineering.*, 2009.

[25] R. Wang, J. Guiochet and G. Motet, "Confidence Assessment Framework for Safety Arguments," in *Proc. of SafeComp'17*, Trento, Italy, 2017.

[26] GSN COMMUNITY STANDARD VERSION 1, Origin Consulting (York) Limited, on behalf of the Contributors, Nov., 2011.

[27] C. Lattner and V. Adve, "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation," in *Proc. of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, USA, 2004.

[28] L. D. Moura and N. Bjørner, "Z3: An efficient smt solver," in *Proceedings of the Tools and Algorithms for the Construction and Analysis of Systems, 2008*, 2008.

[29] P. Ammann and J. Offutt, Introduction to Software Testing (Second Edition), Cambridge University Press, 2016.

[30] E. Gamma, R. Helm, R. Johnson and J. Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley, 1995.

[31] C. Le Goues, N. Holtschulte, E. Smith, Y. Yuriy Brun, P. Devanbu, S. Forrest and W. Weimer, "The ManyBugs and IntroClass Benchmarks for Automated Repair of C Programs," *IEEE TRANSACTIONS ON SOFTWARE ENGINEERING,* vol. 41, no. 12, pp. 1236-1256, December DECEMBER 2015.

Appendix A: Publications and Presentations

1. Chung-Ling Lin, Wuwei Shen, Betty H. C. Cheng, Measuring Confidence of Assurance Cases in Safety-Critical Domains, In *Proceedings of the 53rd Hawaii International Conference on System Sciences (HICSS-53)*, Maui, Hawaii, Jan 6-10, 2020.

Appendix B: Abstracts

In this project, we develop a novel framework to evaluate the quality of a patch generated by an automatic program repair (APR) approach from the certification perspective. The framework initially adopts a symbolic execution engine called KLEE to reveal the difference between two versions of a program as much as possible. Specifically, the framework generates various test cases covering as many different paths in both versions as possible. Next, the framework applies the safety pattern to generate an assurance case in Goal Structuring Notation (GSN) showing how a patch made by an auto program repair technique is evaluated. Finally, our technique employs the D-S theory to deduce the confidence of the assurance case as an approximation of a human certifier's evaluation of an APR approach.

Appendix C: Additional Figures

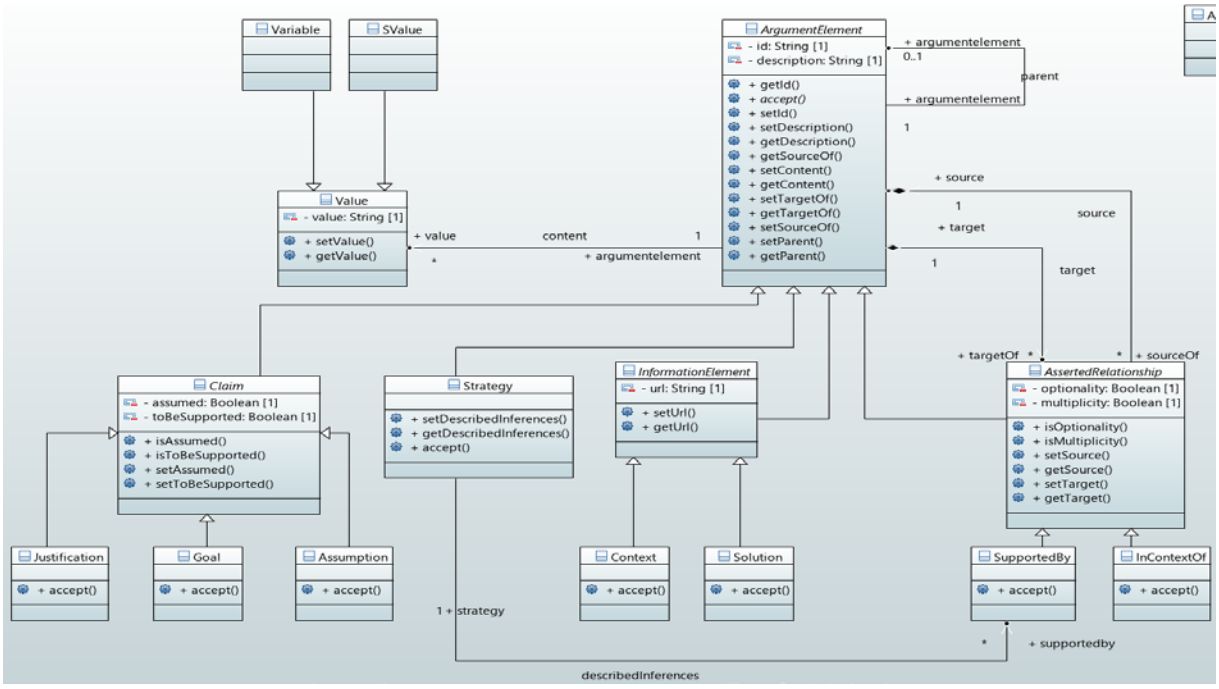


Figure C- 1 Assurance Case Generator Module GSN Class Diagram

LIST OF SYBMOLS, ABBREVIATIONS, AND ACRONYMS

AC: Assurance Case

AFRL: Air Force Research Laboratory

APR: Automatic Program Repair

BB: Basic Block

CFG: Control Flow Graph

D-S theory: Dempster-Shafter theory

GSN: Goal Structuring Notation

IR: Intermediate Representations

LCS: longest common subsequence

LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation

RIPR model: Reachability, Infection, Propagate, and Revealability model

SACM: Structured Assurance Case Metamodel

SAX: Simple API for XML

SE: Symbolic Execution

XMI: XML Message Interface

XML: Extensible Markup Language