2015 IEEE/ACM 37th IEEE International Conference on Software Engineering

A Synergistic Analysis Method for Explaining Failed Regression Tests

Qiuping Yi*†, Zijiang Yang‡, Jian Liu*, Chen Zhao* and Chao Wang§
* Institute of Software, Chinese Academy of Sciences, Beijing, China
† University of Chinese Academy of Sciences, Beijing, China
‡ Department of Computer Science, Western Michigan University, Kalamazoo, Michigan, USA
§ Department of Electrical and Computer Engineering, Virginia Tech, Blacksburg, Virginia, USA

***Abstract*—We propose a new automated debugging method for regression testing based on a synergistic application of both dynamic and semantic analysis. Our method takes a failure-inducing test input, a buggy program, and an earlier correct version of the same program, and computes a minimal set of code changes responsible for the failure, as well as explaining how the code changes lead to the failure. Although this problem has been the subject of intensive research in recent years, existing methods are rarely adopted by developers in practice since they do not produce sufficiently accurate fault explanations for real applications. Our new method is significantly faster and more accurate than existing methods for explaining failed regression tests in real applications, due to its synergistic analysis framework that iteratively applies both dynamic analysis and a constraint solver based semantic analysis to leverage their complementary strengths. We have implemented our new method in a software tool based on the LLVM compiler and the KLEE symbolic virtual machine. Our experiments on large real Linux applications show that the new method is both efficient and effective in practice.**

I. Introduction

Experience has shown that software updates often introduce new bugs. Therefore, it is good practice to conduct regression testing during software development, which determines whether new bugs have been introduced into the code with previously working functionality. Although there exist many tools to automate this process in practice, e.g., re-running regression tests periodically and reporting failures as soon as they occur, detecting these failures is only the first step. The more challenging task is to *identify* the relevant code changes and *explain* why these changes lead to the failure. This is where existing methods fall short.

Although there has been a large body of work on automated debugging in the context of regression testing, few of the existing methods are actively used by developers in practice, for several reasons. First, they are not accurate enough in that faulty code changes are either missed or buried in a large number of irrelevant ones. Second, the causal relationship between faulty code changes and the manifested failures are not explained well enough. Third, in many cases, merely reverting the faulty code changes is not enough because other code changes may be needed as well to make the modified program compile successfully. Due to these problems, developers are forced to rely on manual efforts to interpret the failures.

We propose a new synergistic analysis framework to significantly improve the accuracy of the automatically computed fault explanations, by leveraging a re-execution based dynamic analysis together with a constraint solver based semantic analysis to take advantage of their complementary strengths. Specifically, dynamic analysis is effective in identifying the correlation between code changes and the manifested failure, i.e., by reverting some of these changes and re-executing the program to see if it still fails, but this is ineffective in identifying the causal relationship between the code changes and the failure. In contrast, semantic analysis is effective in identifying the causal relationship between the code changes and the failure, but is ineffective in identifying the actually faulty code changes from a large number of possible ones. By leveraging both types of analysis, we can locate the root cause more accurately as well as more quickly.

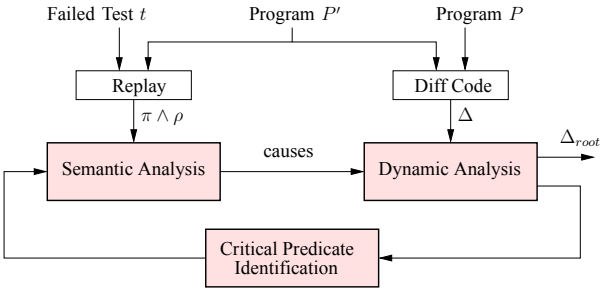

Fig. 1. Synergistic (Dynamic and Semantic) Analysis Framework.

Fig. 1 shows the overall flow of our method. Given a correct program P, its faulty evolution P', and a failed test case t, our method first computes the code difference between P and P' (denoted Δ). Then, it replays the erroneous execution π and obtains the failed assert condition (ρ). Here, we assume the failure is modeled as a failed assertion. Once π, ρ, and Δ are available, our method starts the iterative steps of applying *semantic* analysis and *dynamic* analysis, which are connected with a third component called *critical predicate identification*. Initially, the critical predicate fed to the semantic analysis is the failed assert condition ρ, based on which our semantic analysis computes the cause (causal chain of events responsible for ρ). In the subsequent dynamic analysis, we identify, among the code changes in Δ, a subset (Δ_{root}) that is responsible for the critical predicate ρ. If Δ_{root} can be found, we are done. Otherwise, we identify another critical predicate from the current causes and try again. In the end, we report Δ_{root} together with all related causes and

978-1-4799-1934-5/15 $31.00 © 2015 IEEE
DOI 10.1109/ICSE.2015.46

```
--- find4.15/find.c
+++ find4.18/find.c
@@ -377,0 +421,6 @@
+
+#ifdef O_NOFOLLOW
+  options.open_nofollow_available = check_nofollow();
+#else
+  options.open_nofollow_available = false;
+#endif
```

Fig. 2. Incorrect code changes reported by ADD on *find-a*.

present them in a *tree*-like structure, to highlight the causal relationships between the changes and the failure.

Our new method has significant advantages over existing methods such as delta debugging (DD) [1] and its variants such as augmented delta debugging (ADD) [2], due to its synergistic application of both semantic analysis and dynamic analysis. Delta debugging, in contrast, relies on dynamic analysis only. As an example, consider a regression test failure in version 4.2.18 of Linux application *find*, which has 24K lines of code and 71 code changes since the last correct version 4.2.15. Fig. 2 shows the code changes localized by ADD, which completely missed the real bug. Although reverting these changes can modify the value of `open_nofollow_available` in function `check_nofollow`, thereby avoiding the buggy function `safely_chdir_nofollow`, it merely dodges the failure for the given test input, without fixing the bug.

In contrast, our new method would report the changes $ch1$ and $ch2$ as shown in Fig. 3. A careful study of the bug fix provided by the developers shows that Change $ch1$ matches the actual bug fix. The bug is due to the fourth argument `symlink_handling` of the function `safely_chdir_nofollow`, which was ignored in the new version. The developers fixed the bug by adding a switch statement to handle the previously ignored argument. Although Change $ch2$ does not need to be reverted or modified in order to fix the bug, it is still important since it explains why the faulty function is invoked in the first place. Therefore, the failure explanation computed by our method is more accurate and helpful to debugging. In addition, our method identifies 14 auxiliary code changes that need to be reverted together with $ch1$ and $ch2$ to make the modified program compile successfully – in previous methods, this time-consuming step requires the developers' manual effort.

Although DD reports the two faulty code changes $ch1$ and $ch2$, which is better than ADD (since it missed them), these two changes are buried among eight other code changes that are irrelevant to the failure. The developers have to sift through these other changes manually to understand the root cause. Furthermore, the program obtained by reverting only these ten changes cannot be compiled successfully, which prevents the developers from quickly checking the correlation between them and the failure. Our new method, in contrast, can automatically identify the auxiliary changes needed to be reverted to make the program compile successfully. Finally, neither DD nor ADD can guarantee that there is a causality relationship between the reported code changes and the manifested failure, whereas our new method can.

```
--- find4.15/find.c
+++ find4.18/find.c
@@ -987,0 +1082,78 @@                                  // ch1
...
+static enum SafeChdirStatus
+safely_chdir_nofollow(const char *dest,
...
+static enum SafeChdirStatus
+safely_chdir(const char *dest,
...
@@ -1368 +1641 @@                                      // ch2
-  enum SafeChdirStatus status = safely_chdir
   (name, TraversingDown, &stat_buf);
+  enum SafeChdirStatus status = safely_chdir
   (name, TraversingDown, &stat_buf,
   SymlinkHandleDefault);
```

Fig. 3. Correct code changes reported by AFTER on *find-a*.

There is also a large body of work on identifying the root cause of a manifested failure [3], [4], [5], [6] based on semantic analysis only. However, the problem with these methods is that they focus only on the failures without considering the code changes between two versions, and therefore do not leverage the fact that the original version can serve as a model of the intended program behavior. Furthermore, some existing methods rely on a test suite to provide sufficiently many passing and failing test runs, which may not always exist in practice. Without a golden model or formal specification, the number of possible causes of a manifested failure tends to be very large, since changing any part of the control or data flow along the faulty execution trace could lead to the flip of the assertion condition. Our new method, in contrast, can mitigate the potential explosion of possible causes of a manifested failure by restricting the analysis only to code changes committed between the two versions.

Finally, our new method can guarantee that there is not only correlation but also causality relation between the reported code changes and the manifested failure. We present the causes (causal chains of events) that connect the code changes and the failure in a *tree-like* structure for ease of comprehension. Each event in the causal chain corresponds to a program statement responsible for propagating the fault. Such explanation is more informative than a ranked list of warnings reported by existing methods – it is the reason why we call our approach fault explanation rather than fault localization.

We have implemented our method in a tool built upon LLVM [7] and KLEE [8] and evaluated it on a set of Linux applications such as *find*, *bc*, *make*, *gawk*, and *diff*. Our experimental results show that the new method is both accurate and efficient in computing faulty code changes and explaining the causality relation between them and the failures. To summarize, this paper makes the following contributions:

- We propose a new synergistic analysis method for explaining failed regression testes by leveraging both semantic and dynamic analysis in a unified framework.
- We implement the method in a software tool based on the LLVM compiler and the KLEE symbolic virtual machine.
- We evaluate the new method on a set of large Linux applications and demonstrate its effectiveness in practice.

The remainder of the paper is organized as follows. We present the overall algorithm in Section II, which is then

followed by a detailed description of each major component. We illustrate our method using an example in Section III. We discuss implementation details in Section IV. We present the experimental results in Section V, review related work in Section VI, and finally give our conclusions in Section VII.

II. The Synergistic Analysis Method

Algorithm 1 shows the overall flow of our method based on three inputs: the correct program P, the faulty revision P', and the failure-inducing test input t. Let Δ be the set of code changes between P and P', π be the faulty execution trace, and ρ_0 be the failed assertion. The main part of the algorithm is a loop with three steps: the semantic analysis, the dynamic analysis, and the extraction of critical predicates. These three steps are described as follows:

- In the semantic analysis, our goal is to compute a causal chain of events explaining why the path π leads to the critical predicate ρ. The result is a subset of executed statements, denoted θ (called a cause), that forces ρ to become valid. We use Θ to denote the accumulative set of all causes computed by this analysis. The detailed algorithm is presented in Section II-A.
- In the dynamic analysis, our goal is to determine whether θ is indeed the root cause. In the context of regression testing, we assume that a root cause must be one that involves some of the code changes committed between P and P'. Given the code changes identified by semantic analysis ($\bigcup_{\theta\in\Theta}\cap\Delta$), we repeatedly execute P' with different subsets of these code changes reverted, to see if it can avoid the failure. The detailed algorithm is presented in Section II-B.
- If the current iteration of semantic-dynamic analysis fails to locate the root cause, we need to continue the analysis in the upstream of the cause θ. Toward this end, we identify the set $\mathcal{P}_\theta$ of critical predicates, which are branching conditions along the faulty execution trace π that determine whether the current cause θ can occur. These critical predicates will in turn be used as seeds for the next round of semantic-dynamic analysis. The detailed algorithm is presented in Section II-C.

If the root cause is found, we return the code changes in Δ_{root} together with the relevant subset of causes in Θ. The reason why this is a better failure explanation result is because, in many cases, merely pointing out the faulty code changes is not enough for the developers to understand how they lead to the manifested failure. Therefore, we augment Δ_{root} with the relevant causes to illustrate their causality relationship between the code changes and the manifested failure.

Algorithm 1 Explain (Program P, Program P', Test Input t)

```
1: Let Δ be the set of code changes between P and P';
2: Let π be the faulty execution trace of P' under input t;
3: Let ρ0 be the first critical predicate (failed assertion);
4: Initialization: predicate set P = {ρ0}; cause set Θ = ∅;
5: while P ≠ ∅ do
6:     Remove a predicate ρ from P;
7:     θ ← SemanticAnalysis (π, ρ);
8:     Θ = Θ ∪ θ;
9:     Δ_root ← DynamicAnalysis (P', ⋃_{θ∈Θ} ∩Δ);
10:    if Δ_root ≠ ∅ then
11:        return Δ_root together with the relevant causes in Θ;
12:    end if
13:    P_θ ← ExtractCriticalPredicates (θ);
14:    P = P ∪ P_θ;
15: end while
```

A. Semantic Analysis

Given the faulty path π and the predicate ρ (the negated assert condition), the objective of semantic analysis is to find out why ρ holds in π. In other words, why the execution π would not lead to $\neg\rho$ (passing of the assertion).

Our semantic analysis is based on computing the weakest precondition of $\neg\rho$ along the faulty execution path. Following the notation of Dijkstra [9], we define the weakest precondition of a predicate ϕ with respect to an instruction s as a function mapping ϕ to the formula $wp(s,\phi)$, such that $wp(s,\phi)$ is the *weakest* condition satisfied before executing s that guarantees ρ to be satisfied after executing s. Formally, the weakest precondition (WP) over an assignment, a branching statement, and a sequence of instructions are defined as follows:

- **Assignment** $x := expr$: We define $wp(x{:=}expr,\phi) = \phi[x \leftarrow expr]$. That is, each appearance of x in ϕ is replaced by the right-hand-side expression $expr$.
- **Branch** $if(c)$: We define $wp(if(c),\phi) = (c \wedge \phi)$.
- **Sequence of Instructions**: We define $wp(s_1; s_2, \phi) = wp(s_1, wp(s_2,\phi))$.

During dynamic analysis, assume that the faulty execution trace is $\pi = \langle s_1 \dots s_n\rangle$ and ρ is the critical predicate that holds at the end of π, our WP computation is an iterative procedure $wp(s_1, \dots wp(s_n, \neg\rho))$. Since π actually led to ρ, the logical formula representing the intermediate result of the WP computation must become false at some point. As soon as $wp(s_i, \dots wp(s_n, \neg\rho))$ becomes false, we stop the WP computation, and invoke a satisfiability modulo theory (SMT) solver to compute the minimal unsatisfiable (UNSAT) core. Modern *SMT* solvers such as Yices [10] and Z3 [11] can produce an UNSAT core for a unsatisfiable logical formula, which is a minimal subset of the conjunctive constraints that is still unsatisfiable. In the context of fault localization, the UNSAT core succinctly explains why π cannot lead to $\neg\rho$.

Unfortunately, there still is a gap because the UNSAT core and the cause (causal chain of events) for critical predicate ρ because the UNSAT core itself does not tell us which program statements are responsible for generating the constraints in the UNSAT core. Therefore, we need to map the UNSAT core back to the original program statements by leveraging the so-called *generator instructions*. Given the UNSAT core, the generator instructions of the UNSAT core are all the assignments that participate in the creation of the manifested failure. Intuitively, these statements collectively are responsible for causing $\neg\rho$ to become invalid at the end of the execution π.

Definition 1: A **generator instruction** of a predicate ϕ is either an assignment $v := expr$ such that v appears in the transitive support of ϕ, or a branching condition where the predicate ϕ originally comes from.

It is worth pointing out that, during the WP computation, an *if(c)* condition can only add a new conjunctive constraint (c) to the existing formula, but not transform an existing predicate. In the running example to be introduced in Section III, the generator instructions of the predicate $(sum = 4)$ at Line 13

Algorithm 2 DynamicAnalysisRecur (Program P', Set Δ, Size n)

```
1: if |Δ| < n then
2:     Execute the program P' with changes in Δ reverted;
3:     return Δ if the execution passes
4:     and ∅ otherwise;
5: end if
6: Partition Δ into n subsets: Δ_1, ..., Δ_n;
7: for each subset Δ_i do
8:     if the execution of P' with Δ_i reverted passes then
9:         return DynamicAnalysisRecur (P', Δ_i, 2);
10:    else if the execution of P' with (Δ \ Δ_i) reverted passes then
11:        return DynamicAnalysisRecur (P', (Δ \ Δ_i), 2);
12:    end if
13: end for
14: return DynamicAnalysisRecur (P', Δ, 2n);
```

```
1 int x;
2 int y;   //c1: int m;
  ...
3 x=...;
4 y=...;   //c2: m=...;
  ...
5 z=x+y+2; //c3: z=x+m+3;
6 assert(z=10)
```

Fig. 4. Code snippet cannot be compiled after reverting only c_3.

```
1 int a=2, b=1, c=1, d=0;
2 if (a>0) {
3   if (b<0)
4     if (c!=2)
5       c=2; //end if\@L4
6    d=c+3;   //end if\@L3
  }          //end if\@L2
7 assert(d==5);
```

Fig. 5. An example for computing the critical conditions.

are $\{s_2, s_3, s_7, s_{10}, s_{13}\}$, the bold line numbers in Fig. 7. We call these statements collectively a *cause* as they form a causal chain of events that eventually triggers the failure.

However, the first cause (θ) computed from the failed assertion ρ_0 may not be the root cause. In regression testing, if θ is not triggered by some of the code changes committed between the correct program P and the buggy version P', we would not consider θ as a root cause. The assumption is that the failure manifested in P' but not in P is triggered by some of the recent code changes. In general, it is possible for the root cause to trigger the manifested failure indirectly, by transitively affecting the direct cause θ computed from the failed assertion ρ_0. Therefore, following the semantic analysis, we shall apply the dynamic analysis to determine whether θ is a root cause, and if the answer is no, we shall identify the new starting points for the subsequent semantic analysis.

B. Dynamic Analysis

After the semantics analysis in Section II-A produces a new cause θ and updates the set of causes Θ, the objective of dynamic analysis is to determine which cause, together with the related code changes, is the root cause. The idea, which is similar to the *trial-and-error* approach used in delta debugging, is to execute the buggy program P' with various combination of code changes reverted, to see if the execution passes or fails.

Given the set of causes Θ discovered so far, Algorithm 2 presents our approach to finding the smallest set of changes, such that the execution of P' with the changes reverted passes. The algorithm is recursive with an initial set of changes being $\Delta_\Theta = \bigcup_{\theta \in \Theta} \cap \Delta$, i.e., any change that appears in at least one cause, and the value $n = 2$. In other words, the call *DynamicAnalysis(P', Δ)* in Algorithm 1 is implemented as *DynamicAnalysisRecur($P', \Delta, 2$)*.

Specifically, at Line 6, the set Δ of changes is first partitioned into n subsets. Then, for each subset Δ_i, we first execute P' with Δ_i reverted. If the execution passes, we would like to find out if an execution with a reverted subset of Δ_i can still pass, thus we invoke the recursive call at Line 9. If the execution with reverted Δ_i fails, we try its complement set $(\Delta \setminus \Delta_i)$ at Lines 10 and 11. If no subset or its complement can make an execution pass, we double the value of n so that bigger subsets can be tested. Eventually n becomes larger than the size of Δ itself and it becomes the base case of the recursive function. In Lines 1 to 5, if the number of changes under consideration is less than n, the dynamic analysis either returns all the changes if the execution with reverted Δ passes, or returns $\emptyset$ if the execution still fails.

Algorithm 2 differs from the approach used in delta debugging in that it is *asymmetric* – we consider passing executions only. A *symmetric* approach would be to consider both passing and failing executions: if an execution with reverted change c passes, c is fault related; but if an execution with reverted change c still fails, c is irrelevant to the failure. Although this approach has been employed in many prior works such as delta debugging, it may lead to some faulty code changes to be missed, especially when the failure is caused by multiple code changes. Consider the program in Fig. 6 as an example. The buggy program P' can be executed without failure only if both changes $c2$ and $c3$ are reverted. However, since a re-run with $c2$ reverted fails, a *symmetric* approach would wrongly claim that $c2$ is irrelevant. In contrast, our *asymmetric* approach would not exclude such faulty code changes.

Finally, it is worth pointing out our method presented in Algorithm 2 may not obtain the optimal solution. It is designed in this way to improve the runtime performance in practice, because re-runs can be expensive, especially when the faulty code changes are far away from the failure in terms of the control flow distance. Therefore, in practice, we need to make a trade-off between the accuracy of the computation result and the runtime overhead.

C. Critical Predicate Identification

The dynamic analysis presented in section II-B may not discover the set of code changes responsible for the failure in one shot. In such case, the current cause merely propagates the fault instead of causing the failure. Therefore, we need to examine the upstream of this cause θ along the faulty path π.

Recall that in Section II-A, the generator instructions identified from a cause mandate the validity of a critical predicate (which initially is the failed assertion). If we make changes to at least one generator instruction, the predicates can be evaluated differently. Given a generator instruction $s : v := expr$, there are two ways to change s. One is to change the conditional statement that s depends on so that s may not be executed. The other one is to change the values of the variables in exp so v can be evaluated differently. Based on this observation, we now define the critical predicates with respect to a generator instruction s.

Definition 2: The predicate in a branching statement b is called a **critical predicate** if it has potential impact on a generator instruction t by:

- direct influence $b \rightsquigarrow t$: b immediately determines whether t will be executed; or
- indirect influence $b \not\rightsquigarrow t$: b determines whether an unexecuted statement s will be executed, and s in turn redefines a variable read by t.

Our use of *indirect influence* in the above definition is similar to the *potential dependence* used in relevance slicing [12]. Based on the definition, the set of critical predicates of P_θ in a cause θ is $P_\theta = \bigcup_{t \in \theta}\{b \mid (b \rightsquigarrow t) \vee (b \not\rightsquigarrow t)\}$, where t is a generator instruction.

Consider the program in Fig. 5 as an example. It has an assertion failure along the execution path $\langle 1, 2, 3, 6, 7\rangle$, because the value of d is 4, not 5, at Line 7. Using the algorithm presented in Section II-A, we can obtain the first cause $\theta_0 = \{1, 6, 7\}$. Assuming θ_0 is not the root cause, now we need to look for the critical predicates with respect to the generator instruction at Line 6.

Based on Definition 2, the predicate at Line 2 is critical since it controls whether Line 6 will be executed. The predicate at Line 3 is also critical: were it evaluated to *true*, the statement at Line 5 would have been executed and c would have been redefined. Note that we only consider the executed predicates. The predicate at Line 4 is not considered critical because it is not executed during the test. The critical predicates serve as new starting points for the subsequent semantics analysis, as indicated in Algorithm 1.

Algorithm 3 ExtractCriticalPredicates(Cause θ)

```
1: for each s_l ∈ θ do
2:     let s_j be the closest enclosing branch of s_l not post-dominated by s_l;
3:     P_θ.add(s_j);                    // direct influence
4:     for each variable var used by s_l do
5:         let s be the last instruction that defines var before s_l;
6:         for every branch s_j between s and s_l do
7:             if indirectInfluence(s_j, var) then
8:                 P_θ.add(s_j);        // indirect influence
9:             end if
10:        end for
11:    end for
12: end for
```

The pseudo code for computing the critical predicates is shown in Algorithm 3, which follows Definition 2 with the following modification. For each generator instruction $s_l \in \theta$, we choose only the immediate preceding instance of s_l that is not post-dominated by s_l (Lines 2-3), because other critical predicates will be computed during the subsequent iterations of our analysis. Therefore, no root cause will be missed due to this simplification.

In Algorithm 3, Lines 4-11 deal with the *indirect influence*. Here, s denotes the last instruction found in θ that assigns a value to the variable *var*. Only direct influence in a prefix denoted $\{s_1, ..., s_x\}$ can affect the causes generated in suffix $\{s_x, ..., s_n\}$. If s is the last statement before s_l that assigns the variable var used by s_l, then during the backward analysis, s and s_l represent the upper and lower bound of indirect influence, respectively. For each var used by s_l, the loop at Line 6 identifies every branch instance s_j between s and s_l, which can indirectly affect the value of var as a critical predicate. An example of such instance is the one at Line 3 in Fig. 5.

```
1 bool sorted = True;
2 void f(int x,int y,int z){
3   int sum = 0;
4   if (!sorted) {
5     if (x > y)
6       sum += x;
      else
7       sum += y;
    }else
8     sum += x;
9   if (z > 0)
10    sum += z;
    else
11    sum += (0-z);
12  printf("sum=%d\n",sum);
13  assert(sum == 4);
  }
```

```
1 bool sorted = False; //c1
2 void f(int x,int y,int z){
3   int sum = 0;
4   if (!sorted) {
5     if (x < y)      //c2
6       sum += x;
      else
7       sum += y;
    }else
8     sum += x;
9   if (z > 0)
10    sum += (0-z);   //c3
    else
11    sum += z;       //c4
12  printf("sum=%d\n",sum);
13  assert(sum == 4);
  }
```

Fig. 6. Correct and buggy programs, with four code changes $c1, c2, c3, c4$ and failure-inducing test input $\{x = 3, y = 2, z = 1\}$.

Step	Line Num.	Weakest Precondition (WP)	Satisfiability
1	**13**	**sum=4**	SAT
2	**10**	**sum-z=4**	SAT
3	9	$z > 0 \wedge sum - z = 4$	SAT
4	**7**	$z > 0 \wedge$ **sum+y-z=4**	SAT
5	5	$x \geq y \wedge z > 0 \wedge sum + y - z = 4$	SAT
6	4	$\neg sorted \wedge x \geq y \wedge z > 0 \wedge sum + y - z = 4$	SAT
7	**3**	$\neg sorted \wedge x \geq y \wedge z > 0 \wedge$ **y-z=4**	SAT
8	**2**	$\neg sorted \wedge 3 \geq 2 \wedge 1 > 0 \wedge$ **2-1=4**	UNSAT

Fig. 7. Applying our semantic analysis to the example in Fig. 6, with the faulty execution trace and the critical predicate $(sum \neq 4)$.

III. The Running Example and Comparison to Existing Methods

In this section, we use an example to further illustrate the three steps of our new method. We also compare our method with existing approaches to show that it can return better failure explanations in regression testing.

Fig. 6 shows, side by side, a correct program (left) that computes $max(x, y)+|z|$ and a buggy revision of the program (right) with four code changes, denoted $c1, c2, c3$ and $c4$, respectively. In this example, the variable $sorted$ indicates whether the three input variables have been sorted in the descending order and the variable sum stores the computation result. Due to the code changes at Lines 1, 5, 10, and 11, executing the revised program under the test input $\langle 3, 2, 1\rangle$ (for x, y, z) leads to a wrong result $(sum = 1)$ instead of $(sum = 4)$. The actual code changes responsible for this failure are $c2$ and $c3$. However, existing methods such as delta debugging may either report redundant code changes or miss the faulty code changes. Our new method, in contrast, can identify the faulty code changes precisely as well as explain why they are responsible for the failure.

A. Applying Our New Method

Our method starts by replaying the failed test case $\langle 3, 2, 1\rangle$ on the buggy program (right). Based on the failed execution trace $\pi = \langle s_1, s_2, s_3, s_4, s_5, s_7, s_9, s_{10}, s_{12}, s_{13}\rangle$, our method performs the first *semantic analysis* to identify the *cause* of the failed assert condition $(sum \neq 4)$. The cause returned by the semantic analysis is a minimal set of events (a causal chain) linking some code changes to the failed assertion.

Specifically, we negate the initial predicate $(sum \neq 4)$ at Line 13 and compute the weakest precondition of $(sum = 4)$

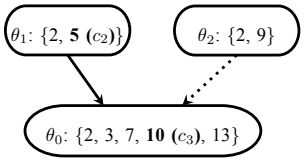

Fig. 8. Our *tree-like* structure for explaining the cause of the failure.

Step	c1	c2	c3	c4	P/F	Max_Pass	Min_Fail	Diff
0	– √	– √	– √	– √	P F	{}	{c1, c2, c3, c4}	{c1, c2, c3, c4}
1	–	–	√	√	F	{}	{c3, c4}	{c3, c4}
2	–	–	–	√	P	{c4}	{c3, c4}	{c3}

Fig. 9. Steps of applying *delta debugging* to the example in Fig. 6.

along the erroneous execution trace backwardly. Since the execution led to $(sum \neq 4)$, the weakest precondition of $(sum = 4)$ is guaranteed to become false at some point during the backward traversal. Fig. 7 shows the steps of this computation, where the WP becomes an unsatisfiable formula at Line 2 after 8 steps, and the UNSAT core is shown as follows:

$$(x = 3) \wedge (y = 2) \wedge (z = 1) \wedge (sum_0 = 0) \wedge (sum_1 = sum_0 + y) \\ \wedge (sum_2 = sum_1 + 0 - z) \implies (sum_2 \neq 4) \quad (1)$$

For ease of comprehension, we have used the static single assignment (SSA) form in the above formula to differentiate multiple occurrences of sum.

Next, we map the constraints in this unsatisfiable subformula (UNSAT core) back to the program statements that produce them (generator instructions).

We get a causal chain of events (or a cause): $\phi_0 = \{s_2, s_3, s_7, s_{10}, s_{13}\}$, which explains why the assert condition $(sum = 4)$ failed. Constraints that are not in the UNSAT core are deemed as irrelevant.

However, a cause returned by the first semantic analysis may not be the *root* cause of the failure. In the subsequent *dynamic analysis*, we check if it is root cause by inspecting the code changes committed between the correction program P and the buggy revision P'. Since $c3$ is the only code change in ϕ_0, to decide if ϕ_0 is the root cause, we revert the change $c3$, re-compile, and re-execute the program. Since the assertion still fails after $c3$ is reverted, we conclude that ϕ_0 is not the root cause – otherwise, reverting $c3$ would have fixed the bug. Unlike existing methods such as delta debugging, our use of this *trial-and-error* style dynamic analysis is guided by the cause computed by the preceding semantic analysis.

Since ϕ_0 is not the root cause but a link between the root cause and the failure, we need to analyze the chain of events in ϕ_0 to identify other critical predicates. A critical predicate is a branching condition whose value determines whether events in ϕ_0 can occur during the execution. In this example, the two critical predicates come from s_5 and s_9, respectively, since they determine whether assignments at Lines 7 and 10 can be executed. These critical predicates are new starting points for the next round of *semantic analysis* based on the weakest predication computation.

The two new causes returned by the subsequent semantic analysis are $\theta_1 = \{s_2, s_5\}$ and $\theta_2 = \{s_2, s_9\}$, the first of which is triggered by the code change $c2$. Furthermore, both changes $c2$ and $c3$ are included in the accumulative set Θ of discovered causes. A subsequent dynamic analysis confirmed that reverting both $c2$ and $c3$ would make the failure go away. Therefore, θ_1 is the root cause. In contrast, $\theta_2 = \{s_2, s_9\}$ is irrelevant.

To report the code changes responsible for the failure, we present $c2$ and $c3$, as well as the causal chains of events (causes), in a tree-like structure shown in Fig. 8. In this figure, nodes are the code changes responsible for triggering the manifested failure and the causal chains of events, whereas edges are the critical predicates linking the causal chains together. Specifically, the result in Fig. 8 shows that the cause θ_1, which includes the change $c2$, leads the incorrect outcome at Line 5, and the cause θ_0 propagates the effect of $c2$ to the failure at Line 13, which includes the code change $c3$.

B. Comparing to Other Methods

The reason why our method is more robust than existing methods is because of its use of semantic analysis to guide dynamic analysis, and vice versa. Therefore, our method can identify not only the *correlation* but also the *causality relation* between the faulty code changes and the manifested failure. To illustrate this advantage, we apply some of the existing methods to the example in Fig. 6 and compare the results.

Fig. 9 shows the results of applying delta debugging (DD) [1] to the running example. In Columns $c1 - c4$, the symbol $\surd$ means that a code change is applied to the correct version P and $-$ indicates that the change is omitted. Therefore, $----$ represents the previously correct program P and $\surd\surd\surd\surd$ represents the buggy program P'. Column P/F shows whether the execution passed without failure or failed. Column *Max_Pass* shows the maximal set of changes applied to P while the execution still passes, whereas Column *Min_Fail* shows the minimal set of changes applied to P while the execution still fails. Delta debugging starts with the correct program ($----$) and the faulty program ($\surd\surd\surd\surd$), for which *Max_Pass* is empty whereas *Min_Fail* is the complete set of changes. The goal is to iteratively reduce *Min_Fail* and enlarge *Max_Pass* such that the difference, shown in Column *Diff*, is minimized. When *Diff* can not be reduced further, it contains the explanation for the failure.

Initially, the set of changes is partitioned into subsets $\{c1, c2\}$ and $\{c3, c4\}$. Since applying $\{c3, c4\}$ to the correct program P causes the execution to fail, delta debugging assumes that the faulty changes are inside $\{c3, c4\}$. Therefore, it decreases *Min_Fail* from $\{c1, c2, c3, c4\}$ to $\{c3, c4\}$ and partitions $\{c3, c4\}$ into $\{c3\}$ and $\{c4\}$. Since applying $\{c4\}$ to program P avoids the failure, $c4$ is added to *Max_Pass*. Delta debugging terminates after this step as $\{c3\}$ cannot be partitioned any further. Therefore, *Diff*=$\{c3\}$ is reported as the explanation. However, this is not the correct result because if we keep the changes of $c1, c2$ and $c4$ in the revised version, and only revert $c3$, the execution still leads to the assertion failure. Therefore, the code changes localized by *Delta Debugging* is not accurate in this example.

Besides delta debugging, there are also fault localization methods based on dynamic slicing [13] and symbolic techniques such as DARWIN [14], [15]. Our method is also more accurate than these methods. Dynamic slicing eliminates program statements that are irrelevant to the manifested failure based on computing the data- and control-dependence. It is a popular technique because of its low cost, but unfortunately, is not accurate [16]. In the running example, dynamic slicing would not be able to prune away any of the program statements in the faulty trace π because the failed assertion transitively depends on all the statements.

DARWIN may produce a better result than dynamic slicing, but the result is still inferior to the one returned by our method because DARWIN does not apply semantic and dynamic analysis synergistically. Specifically, DARWIN tries to explain failure by comparing the weakest precondition of the assertion along the execution paths in the correct program and its faulty version. Applied to the running example, it would generate the weakest precondition *WP1* in passing execution as $sorted \wedge (z > 0) \wedge (x + z = 4)$, and the weakest precondition *WP2* in failing execution as $\neg sorted \wedge (x \geq y) \wedge (z > 0) \wedge (y - z = 4)$. Since all conditions in *WP1 (WP2)* are *unexplained* by *WP2 (WP1)*, except for $(z > 0)$, DARWIN would report almost all the executed statements in both the passing and the failing executions as failure-related. As a result, the developers would have to sift through the irrelevant code changes and program statements in order to understand the root cause of the failure. Compared with [14], [15], our method exploits the use of UNSAT cores to prune away redundant predicates, and stops the backward exploration as soon as error-inducing code changes are identified. In addition, our method presents a tree based structure to illustrate the fault propogation.

IV. Computing Auxiliary Code Changes

Besides passing or failing, an execution of the program P' with some reverted code changes may have a third possibility – the program may not be compiled successfully. Consider the example in Fig. 4. When only the change $c2$ is reverted, the resulting program cannot be compiled because the variable m has not been declared before its usage. During our study of real-world regression test examples, we have found that such cases are common in practice and it is time-consuming for the developers to identify such code changes manually. In this section, we present a solution to this problem.

The problem we want to solve is formally stated as follows. Assume P' can no longer be compiled with the code changes in Δ^- reverted, our goal in Algorithm 4 is to find an augment set of code changes, denoted Δ^+, such that if Δ^+ is reverted together with Δ^-, the resulting program can be compiled.

While computing Δ^+, we ensure that Δ^- remains the set of code changes that have caused the compilation error in the first place. In contrast, we decrease the set Δ^+ monotonically during the recursive application of the function *FindAuxChange()*. Initially the value of Δ^+ is $(\Delta \setminus \Delta^-)$, the set of all changes in P' except Δ^-. Since $(\Delta^- \cup \Delta^+)$ is the same as Δ, reverting these changes in P' leads to P, which can be compiled successfully. The goal of each recursive call is thus trying to find a subset of Δ^+ that can still work together with Δ^- to make the program compile. This is achieved by partitioning

Algorithm 4 FindAuxChange (Set Δ^-, Set Δ^+, Size n)

```
1: assert(Program P' with (Δ⁻ ∪ Δ⁺) reverted can be compiled);
2: if |Δ⁺| < n then
3:     return Δ⁺ ;
4: end if
5: Partition Δ⁺ into n subsets: Δ_1, ..., Δ_n;
6: for each Δ_i do
7:     if Program P' with (Δ⁻ ∪ Δ_i) reverted can be compiled then
8:         return FindAuxChange(Δ⁻, Δ_i, n);
9:     else if P' with Δ⁻ ∪ (Δ⁺ \ Δ_i) reverted can be compiled then
10:        return FindAuxChange(Δ⁻, (Δ⁺ \ Δ_i), n);
11:    end if
12: end for
13: return FindAuxChange(Δ⁻, Δ⁺, 2n);
```

```
--- FileA
+++ FileB
@@ -BeginA,SpanA +BeginB,SpanB @@
- Line_A_1
- ...
- Line_A_SpanA
+ Line_B_1
+ ...
+ Line_B_SpanB
```

Fig. 10. The template of the unified code changes.

Δ^+ and trying out each subset. The invariant that $(\Delta^- \cup \Delta^+)$ *solves the compilation problem* is always maintained. We note that all changes in the program P' have to be considered, not just the changes appearing in the faulty path π.

To improve the performance, we rely on the program structure to partition the change set. For example, a change often grammatically depends on the changes within the same file or function, so we partition accordingly. Furthermore, in Algorithm 4, we cache the results of *FindAuxChange* to prevent redundant computation. For example, after $\Delta_1^+ =$*FindAuxChange*$(\Delta_1^-, \Delta \setminus \Delta_1^-, 2)$ is computed, we need to find an auxiliary set for $\Delta_2^- \subset \Delta_1^-$. In this case, we invoke *FindAuxChange*$(\Delta_2^-, \Delta_1^+, 2)$ because the program with reverted $(\Delta_2^- \cup \Delta_1^+)$ can always be successfully compiled.

The set of all code changes in the program is computed using the Linux utility application *diff* (which happens to be a benchmark application used in our experimental evaluation of the new method), which is used with the option *u* to compute the difference between two versions. Fig. 10 shows the template of such comparisons. It starts with two lines of the two file names under comparison (the time part is omitted) and then describes the change hunks. Each change hunk starts with a line `"@@ -BeginA,SpanA +BeginB,SpanB @@"` that specifies the starting and ending line numbers of the changes between the two files. Following the ranges are the detailed differences. That is, Lines `BeginA` $\sim$ `BeginA+SpanA-1` from `FileA` is replaced by Lines `BeginB` $\sim$ `BeginB+SpanB-1` from `FileB`. During the implementation, we have chosen the smallest granularity possible because the size of the change hunks may affect the precision of the subsequent analysis.

V. Experiments

We have implemented our method in a tool based on the LLVM compiler [7] and the KLEE symbolic virtual machine [8]. The tool, called AFTER (Automated FaulT Explanation for Regression testing), can handle C/C++ applications

TABLE I
CHARACTERISTICS OF THE BENCHMARK APPLICATIONS USED IN OUR EXPERIMENTS.

Name	LoC	Correct (P)	Buggy (P')	#Change	Failure Description	Reported Site
find-a	24k	V4.2.15	V4.2.18	71	Using -L/-H produces wrong output	http://savannah.gnu.org/bugs/?12181
find-b	40k	V4.3.5	V4.3.6	243	Using -mtime produces wrong output	http://savannah.gnu.org/bugs/?20005
find-c	40k	V4.3.5	V4.3.6	243	Using -size produces error message	http://savannah.gnu.org/bugs/?30180
bc	10k	V1.05a	V1.06	534	Argument processing error	https://bugs.gentoo.org/show_bug.cgi?id=51525
make	23k	V3.80	V3.81	1,257	Using -r produces wrong output	http://savannah.gnu.org/bugs/?20006
gawk	37k	V3.1.0	V3.1.1	897	Use of strtonum causes abort	https://bugs.debian.org/cgi-bin/bugreport.cgi?bug=159279
diff	20k	V2.8.1	V2.9	373	Adds additional newline	https://bugs.debian.org/cgi-bin/bugreport.cgi?bug=577832

that work on the LLVM/KLEE platform. We use the Yices SMT solver [10] to implement the computation of UNSAT cores. During our experiments, we have evaluated the failure explanation capability of our method and compared it with two existing techniques: the classic delta debugging (DD) [1] and a recent improvement called augmented delta debugging (ADD) [2] in the same tool.

Our experimental evaluation was designed to answer the following research questions:

1) How accurately can our new method localize the set of code changes responsible for the manifested failure? Specifically, we want to know (a) whether the faulty code changes will be missed, and if the answer is no, then (b) whether the faulty code changes will be buried in a large number of irrelevant code changes.
2) How scalable is our new method in handling real applications? Compared to existing methods such as DD and ADD, which use dynamic analysis but not semantic analysis, our use of the SMT solver may bring some overhead but may also reduce the number of redundant tests. We want to know if our method has a good overall runtime performance.

We conducted experiments on seven widely used Linux applications such as `find`, `bc`, `make`, `gawk`, and `diff`. Each application has tens of thousands of lines of C code and hundreds of code changes committed between the correct and buggy versions. Table I shows their characteristics, including the name, the number of lines of code, the versions of the correct and buggy programs, a description of the error, and the website on which the bug was reported. By studying the bug reports and patches proposed by the developers, we identified the minimal set of faulty code changes responsible for each failure and understood how they triggered the failure. Then, we ran our tool on these benchmarks and compared the results of the following three methods: DD, ADD, and AFTER. All the experiments were conducted on a computer with a 2.66GHz Intel dual core CPU and 4 GB RAM.

A. Accuracy of the Failure Explanation

In computer aided debugging, we generally expect the analysis tool to provide only hints as to where the faulty code changes are and then rely on the developers to identify the root cause from the reported changes. Therefore, the quality of a fault explanation method is evaluated using the following criteria: (a) whether the faulty code changes are included in the set of reported changes, and if the answer to the previous question is yes, then (b) whether the faulty code changes are buried in a large number of irrelevant ones.

To compare the performance of different methods, for each benchmark and each method, we classified the result into one of the following three categories:

- *Matched*, meaning that the reported code changes matched the actual bug fixes provided by the developers. In this case, the developers proposed to either revert these code changes or revise them in order to fix the reported bug.
- *Missed*, meaning that reverting the code changes would not avoid the failure, or merely dodge it since a repaired program simply chose a different branch and could no longer reach the buggy code. In this case, the result is not helpful to debugging.
- *Partial*, meaning that reverting the code changes would make the failure disappear, but the developers have decided that these are necessary changes. Instead, other parts of the code should be revised to accommodate these changes. For example, in the code snippet `a=2;b=3;assert(a+b==5);` changing `b=3` to `b=2` would cause an assertion failure. Although `b=2` is the actual root cause of this failure, the developer may decide that the fix should be to revise `a=2` to `a=3`.

Table II shows the results of comparing the code changes returned by the three methods. Columns 1 and 2 show the name of the benchmark and the total number of code changes between the correct and buggy versions. Column 3 shows the number of code changes localized by DD. Column 4 shows whether the root cause is included in the reported Δ. Columns 5-6 show the result of ADD and Columns 7-8 show the result of AFTER in the same format. Column 9 shows the actually faulty code changes for each benchmark program, obtained by our inspection of the software code and comments from the developers.

The results in Table II indicate that our method is more accurate in localizing the faulty code changes. In all cases, the changes localized by AFTER include the actual bug fixes provided by the developers. In contrast, ADD missed the actual bug in *find-a*, and both DD and ADD reported *partial* results on *diff*. Furthermore, the false positives of AFTER are significantly fewer than the other two methods. On average, among the 516.8 code changes between the two versions, our method will report only 2.4 code changes, among which 1.7 are the actual faulty code changes.

B. Comparing the Runtime Performance

Our new method relies on a synergistic analysis framework that leverages both the *trial-and-error* style dynamic analysis

TABLE II
RELEVANT CODE CHANGES COMPUTED BY DD, ADD, AND AFTER.

Name	#Changes	DD [1]		ADD [2]		AFTER		Actual
		Δ	Match	Δ	Match	Δ	Match	Δ
find-a	71	10	yes	1	*missed*	2	yes	2
find-b	243	108	yes	6	yes	5	yes	2
find-c	243	2	yes	2	yes	1	yes	1
bc	534	1	yes	1	yes	1	yes	1
make	1,257	129	yes	63	yes	6	yes	4
gawk	897	1	yes	1	yes	1	yes	1
diff	373	1	*partial*	1	*partial*	1	yes	1
Avg.	**516.8**	**36.0**	–	**10.7**	–	**2.4**	–	**1.7**

TABLE III
COMPARING THE RUNTIME PERFORMANCE OF DD, ADD, AND AFTER.

Name	DD [1]		ADD [2]		AFTER		Speedup	
	#Test	time(s)	#Test	time(s)	#Test	time(s)	#S1	#S2
find-a	158	82	34	17	264	125	0.7X	0.1X
find-b	1,161	1,199	35	61	223	321	3.7X	0.2X
find-c	30	37	6	10	32	39	0.9X	0.3X
bc	486	287	29	25	1	12	23.9X	2.1X
make	2,368	7,640	526	1,833	257	946	8.1X	1.9X
gawk	638	4,112	7	73	1	5	822.4X	14.6X
diff	376	522	35	45	3	31	16.8X	1.5X
Avg.	–	**1,983**	–	**295**	–	**211**	**9.4X**	**1.4X**

(which is similar to DD and ADD) and the SMT solver based semantic analysis. Therefore, a natural question is whether the use of the SMT solver would slow down the method. To answer this question, we compared the runtime performance of the three methods. Table III shows the result. Columns 2-7 compare the number of test runs explored by the dynamic analysis component of each method, and the total execution time. Columns 8 and 9 show the speedup of our new method, AFTER, over the other two methods. Specifically, we define $\#S1 = \frac{\#Time_{DD}}{\#Time_{AFTER}}$ and $\#S2 = \frac{\#Time_{ADD}}{\#Time_{AFTER}}$.

The result in Table III shows that although AFTER spends additional time on the semantic analysis, whereas the other two methods do not, the runtime overhead often is more than compensated by the smaller number of test runs needed. On average, our method is 9.4 times faster than DD and 1.4 times faster than ADD. The use of semantic analysis can drastically reduce the number of re-executions needed by dynamic analysis. This is especially important for programs where the code changes are far away from the manifested failures, for which re-execution takes a long time.

C. Statistics of our Synergistic Analysis

Recall that our method has two additional features that DD and ADD do not have. The first one is the capability of computing Δ_{Aux}, the set of code changes that must be reverted together with Δ_{root} to make it compile. The second one is the capability of reporting a tree of causal event chains to explain how the faulty code changes lead to the failure.

Table IV shows the statistics of running our method on the benchmark programs. We break down Δ_{Total}, the total number of code changes reported by our tool, into two parts. That is, $\Delta_{Total} = \Delta_{Aux} + \Delta_{Root}$, where Δ_{Aux} is the number of addition changes that must be reverted to make the program compile, and Δ_{Root} is the set of changes responsible for the failure. One main advantage of our method over existing methods is the capability of computing Δ_{Aux}. We also report,

TABLE IV
STATISTICS OF RUNNING AFTER ON THE BENCHMARK PROGRAMS.

Name	#Changes	AFTER				
		Δ_{Total}	Δ_{Aux}	Δ_{Root}	SMT-time(s)	#causes
find-a	71	16	14	2	5	2
find-b	243	5	0	5	72	6
find-c	243	2	1	1	3	1
bc	534	1	0	1	2	1
make	1,257	31	25	6	135	7
gawk	897	1	0	1	1	1
diff	373	1	0	1	27	5
Avg.	**516.8**	**8.1**	**5.7**	**2.4**	**35.0**	**3**

```
--- diffutils-2.8/src/io.c
+++ diffutils-2.9.1/src/io.c
@@ -664,2 +650,2 @@
-       for (; p0 != beg0; p0--, p1--)
- if (*p0 != *p1)
+       while (p0 != beg0)
+ if (*--p0 != *--p1)
```

Fig. 11. The code change of program *diff* reported by DD, ADD, and AFTER.

among the total execution time, how many seconds are spent on running the SMT solver based semantic analysis.

The last column shows the number of causes (causal chains of events) that link the faulty code changes to the manifested failure. The causes computed by our SMT solver based analysis can help the developers understand the causality relationship between the code changes and the failure – it is the main reason why we call our approach fault explanation instead of fault localization.

Besides establishing the causality relationship, the reported *tree of causes* can provide hints to programmers on how to fix the faulty program. Consider the `diff` benchmark program, where all three methods reported the code change in Fig. 11. However, the result reported by AFTER is a *match* whereas the results reported by the other two methods are *partial*, for the following reasons. First, reverting the change in Fig. 11 can indeed make the failure go away. However, based on the comments from the developers, this change itself was not faulty since it was introduced to fix a bug appeared in an earlier version (http://git.savannah.gnu.org/cgit/diffutils.git/commit/?id=58d0483b621792959a485876aee05d799b6470de), and the bug eventually was fixed by adding another condition to catch and correct the affected variable. Our method correctly explained this failure because, in addition to this faulty change, our method also reported 5 causes, which formed a chain of propagation. The actual bug fix proposed by the developers was on our causality chain (https://bugs.debian.org/cgi-bin/bugreport.cgi?bug=577832).

Consider another example, *make*, which is the benchmark application on which DD and ADD reported significantly more code changes than AFTER. For this example, the bug fix provided by the developers is to remove the check of `f→is_target` shown in Fig. 12. Although this change is also reported by by DD and ADD – which earns them a *match* – this faulty code change is buried among 129 and 63 irrelevant changes, respectively. In practice, it would be too time-consuming for the developers to sift through such large numbers of potential code changes. In contrast, our method

```
--- make-3.80/implicit.c
+++ make-3.81/implicit.c
@@ -402 +698,2 @@
-       if (lookup_file (p) != 0
+    /* @@ dep->changed check is disabled. */
+    if (((f = lookup_file (name)) != 0
       && f->is_target)
```

Fig. 12. Actual bug fix reported by AFTER for the *make* benchmark.

reported only 6 code changes, together with 25 auxiliary code changes to make the program compile successfully.

VI. Related Work

There is a large body of work on debugging evolving programs based on the *trial-and-error* style analysis as pioneered by Zeller *et al.* [1], [17], [18]. The approach, commonly known as *delta debugging (DD)*, has been combined with other techniques including execution coverage (also known as the *augmented delta debugging (ADD) [2]*), observation-based slicing [19], dual slicing [20] and hierarchical information [21] to improve precision. However, these methods rely on dynamic analysis only, whereas in our synergistic analysis framework, we also use SMT based semantic analysis to compute causality chains and provide guidance to the dynamic analysis.

Another line of influential work on debugging evolving programs is based on symbolic techniques. For example, the DARWIN [14] method relies on the assumption that the path conditions of buggy and correct executions often differ from each other. By comparing the difference, DARWIN is effective in finding control logic errors. However, it is less effective in finding errors in other parts of the code such as the assignments, since they do not alter the control flow. Banerjee et al. [15] proposes a remedial method for DARWIN by comparing the erroneous instructions against a golden program. However, neither method is based on the synergistic application of both dynamic analysis and semantic analysis. Furthermore, neither provides the tree-like explanation of the fault propagation as in our method.

The concept of *as correct as the previous version* has become popular in the subfield of regression verification [22], [23]. Although these works do not directly focus on explaining failed regression tests as in this paper, they complement our work at the high level.

Dynamic slicing [13], together with many variations [12], [24], is another widely used error triaging technique. However, it may not be able to remove many of the semantically irrelevant program statements, thereby limiting its usefulness [16]. Therefore, in practice, slicing typically is used as an auxiliary technique to complement other methods in automated debugging. For example, the problem tackled in this paper cannot be solved by simply combining dynamic slicing with a weakest precondition computation over the backward slice – the use of re-execution based analysis is also crucial to identify the actual causality relationship between the recent code changes and the observed failure.

BugAssist [25] is a Boolean SAT solver based tool for localizing potentially faulty program statements in a buggy C programs. The tool leverages a SAT solver's capability to compute minimal unsatisfiability core in the CBMC verification tool. Ermis *et al.* [26], [27] propose a Graig interpolant [28] based method for computing *error invariants*, which are then used to identify portions of a faulty trace that are irrelevant. There are also other fault localization methods based on weakest precondition [29], [30], [31] and inductive interpolant [32] to explain the encountered failure. However, none of these methods is geared toward regression testing, and as such, they do not utilize the code changes between the correct and buggy versions.

There are bug triaging methods based on comparing the passing and failing execution traces [3], [4], [5], [6], [33], [34]. For a given failing execution, they find passing executions that are as similar to the failing one as possible. Then, they identify the difference between passing and failing executions and present them as an explanation of the failure. Methods based on the use of dynamically learned likely program invariants [35] can also be efficient for catching the differences between failing and passing executions. However, the effectiveness of these methods is limited by the quality and sometimes the availability of the test suite.

Fault localization methods based on identifying anomalous events in program executions [36], [37], [38] rely on the assumption that *rarely occurring* events are likely faulty. A representative tool that falls in this category is RADAR [37], [38], which derives multiple models from the base version of the program and compares them with the failed execution to identify a chain of *suspicious* anomalous events. Such techniques differ from our method in that they do not rely on semantic analysis and therefore can only discover *correlation* between the anomalous events and the failure, but not the *causal relationship*. Nevertheless, they are complementary to our method in that the anomalous events can be used by our method to further prune the irrelevant predicates.

VII. Conclusions

We have presented a new synergistic analysis method for localizing faulty code changes in the context of regression testing and explaining how these code changes lead to the manifested failure. The method relies on an iterative framework that leverages dynamic analysis to identify the correlation between the code changes and the failure, and also leverages semantic analysis to identify the causality relationship between them. Our experiments on widely used Linux applications show that the new method is effective in localizing relevant code changes in practice. Furthermore, our method can report a tree of causes to help explain the chain of fault propagation events from the code changes to the manifested failure.

Acknowledgments

We thank our colleagues Changhee Jung and Dongyoon Lee for comments that greatly improved the manuscript. This research was supported by the National Science Foundation of China (NSFC) under grant 61472318, the National Science and Technology Major Project of China under grant 2012ZX01039-004, and the National Science Foundation (NSF) under grants CCF-1149454, CCF-1500365, and CCF-1500024. Any opinions, findings, and conclusions expressed in this material are those of the authors and do not necessarily reflect the views of the funding agencies.

REFERENCES

[1] A. Zeller, "Yesterday, my program worked. today, it does not. why?" in *ACM SIGSOFT Symposium on Foundations of Software Engineering*, 1999, pp. 253–267.

[2] K. Yu, M. Lin, J. Chen, and X. Zhang, "Practical isolation of failure-inducing changes for debugging regression faults," in *IEEE/ACM International Conference On Automated Software Engineering*, New York, NY, USA, 2012, pp. 20–29.

[3] A. Groce and W. Visser, "What went wrong: explaining counterexamples," in *International SPIN Workshop on Model Checking Software*, 2003, pp. 121–136.

[4] T. Ball, M. Naik, and S. K. Rajamani, "From symptom to cause: localizing errors in counterexample traces," in *ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, 2003, pp. 97–105.

[5] M. Renieris and S. P. Reiss, "Fault localization with nearest neighbor queries," in *IEEE/ACM International Conference On Automated Software Engineering*, 2003, pp. 30–39.

[6] A. Groce, S. Chaki, D. Kroening, and O. Strichman, "Error explanation with distance metrics," *Int. J. Softw. Tools Technol. Transf.*, vol. 8, no. 3, pp. 229–247, Jun. 2006.

[7] C. Lattner, "LLVM: An infrastructure for multi-stage optimization," in *Masters Thesis*, 2002.

[8] C. Cadar, D. Dunbar, and D. Engler, "KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs," in *USENIX Symposium on Operating Systems Design and Implementation*, 2008, pp. 209–224.

[9] E. W. Dijkstra, *A Discipline of Programming*. Englewood Cliffs, NJ: Prentice-Hall, 1976.

[10] B. Dutertre and L. de Moura, "A fast linear-arithmetic solver for DPLL(T)," in *International Conference on Computer Aided Verification*. Springer, 2006, pp. 81–94, LNCS 4144.

[11] L. De Moura and N. Bjørner, "Z3: an efficient SMT solver," in *International Conference on Tools and Algorithms for Construction and Analysis of Systems*, 2008, pp. 337–340.

[12] T. Gyimóthy, A. Beszédes, and I. Forgács, "An efficient relevant slicing method for debugging," in *ACM SIGSOFT Symposium on Foundations of Software Engineering*, 1999, pp. 303–321.

[13] B. Korel and J. Laski, "Dynamic program slicing," *Inf. Process. Lett.*, vol. 29, no. 3, pp. 155–163, Oct. 1988.

[14] D. Qi, A. Roychoudhury, Z. Liang, and K. Vaswani, "Darwin: an approach for debugging evolving programs," in *ACM SIGSOFT Symposium on Foundations of Software Engineering*, 2009, pp. 33–42.

[15] A. Banerjee, A. Roychoudhury, J. A. Harlie, and Z. Liang, "Golden implementation driven software debugging," in *ACM SIGSOFT Symposium on Foundations of Software Engineering*, 2010, pp. 177–186.

[16] F. Tip, "A survey of program slicing techniques," *J. of programming languages*, vol. 3, pp. 121–189, 1995.

[17] A. Zeller and R. Hildebrandt, "Simplifying and isolating failure-inducing input," *IEEE Trans. Softw. Eng.*, vol. 28, 2002.

[18] A. Zeller, "Isolating cause-effect chains from computer programs," in *ACM SIGSOFT Symposium on Foundations of Software Engineering*, 2002, pp. 1–10.

[19] D. Binkley, N. Gold, M. Harman, S. Islam, J. Krinke, and S. Yoo, "Observation-based slicing," Department of Computer Science, University College London, Tech. Rep. RN/13/13, 2013.

[20] W. N. Sumner and X. Zhang, "Comparative causality: Explaining the differences between executions," in *International Conference on Software Engineering*, 2013, pp. 272–281.

[21] G. Misherghi and Z. Su, "HDD: hierarchical delta debugging," in *International Conference on Software Engineering*, 2006, pp. 142–151.

[22] M. Böhme, B. C. d. S. Oliveira, and A. Roychoudhury, "Partition-based regression verification," in *International Conference on Software Engineering*, 2013, pp. 302–311.

[23] F. Pastore, L. Mariani, A. E. J. Hyvärinen, G. Fedyukovich, N. Sharygina, S. Sehestedt, and A. Muhammad, "Verification-aided regression testing," in *International Symposium on Software Testing and Analysis*, 2014, pp. 37–48.

[24] X. Zhang, S. Tallam, N. Gupta, and R. Gupta, "Towards locating execution omission errors," in *ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2007, pp. 415–424.

[25] M. Jose and R. Majumdar, "Cause clue clauses: error localization using maximum satisfiability," in *ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2011, pp. 437–446.

[26] E. Ermis, M. Schäf, and T. Wies, "Error invariants," in *International Symposium on Formal Methods*, 2012, vol. 7436, pp. 187–201.

[27] J. Christ, E. Ermis, M. Schäf, and T. Wies, "Flow-sensitive fault localization," in *International Conference on Verification, Model Checking, and Abstract Interpretation*, 2013, vol. 7737, pp. 189–208.

[28] W. Craig, "Three uses of the herbrand-gentzen theorem in relating model theory and proof theory," *J. Symb. Log.*, vol. 22, no. 3, pp. 269–285, 1957.

[29] C. Wang, Z. Yang, F. Ivančić, and A. Gupta, "Whodunit? causal analysis for counterexamples," in *International Symposium on Automated Technology for Verification and Analysis*, 2006, pp. 82–95.

[30] X. Zhang, N. Gupta, and R. Gupta, "Locating faults through automated predicate switching," in *International Conference on Software Engineering*, 2006, pp. 272–281.

[31] Q. Yi, Z. Yang, J. Liu, C. Zhao, and C. Wang, "Explaining software failures by cascade fault localization," *ACM Transactions on Design Automation of Electronic Systems*, 2015.

[32] V. Murali, N. Sinha, E. Torlak, and S. Chandra, "What gives? A hybrid algorithm for error trace explanation," in *International Conference on Verified Software: Theories, Tools and Experiments*, 2014, pp. 270–286.

[33] A. Perez, R. Abreu, and A. Riboira, "A dynamic code coverage approach to maximize fault localization efficiency," *Journal of Systems and Software*, vol. 90, no. 0, pp. 18 – 28, 2014.

[34] M. T. Befrouei, C. Wang, and G. Weissenbacher, "Abstraction and mining of traces to explain concurrency bugs," in *International Conference on Runtime Verification, Toronto, ON, Canada, September 22-25, 2014.*, 2014, pp. 162–177.

[35] S. K. Sahoo, J. Criswell, C. Geigle, and V. Adve, "Using likely invariants for automated software fault localization," in *International Conference on Architectural Support for Programming Languages and Operating Systems*, 2013, pp. 139–152.

[36] L. Mariani and F. Pastore, "Automated identification of failure causes in system logs," in *International Symposium on Software Reliability Engineering*, 2008, pp. 117–126.

[37] F. and Pastore, , L. Mariani, , A. Goffi, , M. Oriol, and M. Wahler, "Dynamic analysis of upgrades in C/C++ software," in *International Symposium on Software Reliability Engineering*, 2012.

[38] F. Pastore, L. Mariani, and A. Goffi, "RADAR: a tool for debugging regression problems in C/C++ software," in *International Conference on Software Engineering*, 2013, pp. 1335–1338.