1 Introduction

Let P be a *deterministic*[1] program that when runs against an input i always returns an output $P(i)$. A program P' is *different* than P if there exists some inputs i such that $P(i) \neq P'(i)$. For abbreviation $P(i)$ also means running P against input i. Examples of different programs include the return values of $P(i)$ are not the same as those of $P'(i)$ or $P(i)$ returns but $P'(i)$ does not due to a crash or an infinite loop[2].

Let $T_{\text{diff}} = \{i_1, \ldots, i_m\}$ be a testsuite containing test inputs such that each input $i \in T_{\text{diff}}, P(i) \neq P'(i)$ and $T_{\text{same}} = \{i_1, \ldots, i_n\}$ contains inputs such that each input $i \in T_{\text{diff}}, P(i) = P'(i)$. We say program P' is different than P iff $|T_{\text{diff}}| \geq 1$.

Based on the above definitions, if P is the correct program we want, then P' is the incorrect (or buggy) program that we want to fix. We assume that bug is an unintentional mistake made by the programmer; that is, for most parts, P' behaves very much like P. In other words, we do not intend to create program P from scratch. Moreover, our approach is to modify the program based on the semantics of the bug. For instance if the error occurs due to unnecessary checking condition then the fix will be removing such condition. Our objective is to solve the following problem: "Given a buggy program P', a set of testcases T, and *oracle* that can partition T into two sets T_{same} and T_{diff}, we attempt modify P' to achieve similar results as P on all inputs from T, i.e., $\forall i \in T_{\text{diff}} \cup T_{\text{same}}, P(i) = P'(i)$".

Our proposed approach to fix bugs relies on *invariants* detection. First, we obtain the *invariants* of P' using a hybrid of *dynamic* analysis (by observing the behaviors of P' when runs on T) and *static* analysis (by examining the source code of P'). Invariants discovered from dynamic analysis allows us to simulate a *good* and a *bad* model when running P' against test inputs from T_{diff} and T_{same}, respectively. Static analysis helps capture more precise invariants of P' which represents the *bad* model. Furthermore, if we know what P is supposed to do (e.g., the specifications (or a partial subset) of P are given), we can also strengthen the *good* model with these additional information. Next we compare the two models and get their differences. We hypothesize that the bug is caused by these differences (e.g., if the good model has invariant $x < C$ but the bad model has $x \leq C$, then perhaps the bug occurs dues to this weaker *less than* condition in P'). Once we obtain the differences and infer the causes of the bug, we modify P' to diminish the anomalies from the bad model while preserve the properties of the good model (e.g., change $x \leq C$ to $x < C$). After the modification we rerun the modified program P'' against testcases in T to see if the results match with those provided from the oracle. If all of them match, we say P'' is our candidate fix.

2 Approach

Our procedure is analogous to typical human debugging process, i.e., after confirming the existence of a bug, determine what properties are unique when the bug occurs and fix the problem accordingly. Our main contributions are 1) provide new and improved techniques to capture precise program specifications, i.e., to understand why bug occurs and 2) present algorithm to modify (or recreate) program that adheres to certain properties and avoids others (a form of *program synthesis*).

Hereafter we use the terms properties, specifications, invariants, assertions interchangeably. These terminologies refer to first-order logic (FOL) formulae that hold at various program points. Examples of FOLs: $x \leq y, 0 \leq y \leq 10, x = y^2 + 10, y >= 0 \wedge x = \text{sqrt}(y), \text{prime}(x) \wedge x = f(y,z), \exists x.\ A[x] = y, \forall x.\ x > 2 \wedge \text{even}(x) \rightarrow (\exists(y,z).\ \text{prime}(y) \wedge \text{prime}(z) \wedge x = y + z)$[3].

3 Generating Program Invariants

Invariants in essence provide properties that are guaranteed to hold throughout the program. The more precise a property is, the more accurate it describes the program. The specifications of the program (if available) are considered the most accurate; program that have properties not conform to these specifications are considered incorrect. However in most cases specifications are very limited or not available, thus we use dynamic analysis to obtain them. Dynamic analysis reports properties that occur during a program run (execution trace). Due to quality of testcases and difficulties in generating invariants of complex forms, properties observed by dynamic analysis are most likely imprecise. In the case that the source code of the program (or a partial set of the specifications) is provided, we statically analysize these to get more accurate program properties. Static analysis does not require testcases and can provide more precise (and provable) forms of invariants than those discovered by dynamic analysis (e.g., we could analysize the source code to reason about relationships between variables).

The Daikon system is a well-known dynamic invariant detector in recent literatures. Nonetheless the types of invariants it can generate are quite weak, e.g., of simple forms: xRy where $R = \{<, \leq, =, \geq, >\}$. It seems very unlikely that Daikon can discover more complex forms such as $y = x^3 + d$ or $x = (2*a + 3*b + 4*c)/y$. Well established statical methods such as Karr's analysis use abstract domain to discover invariants of form $c_0 + c_1x_1 + \ldots + c_nx_n = 0$. The algorithm is heuristic and works reasonably well; however only for invariants of linear form.

Consequently, in this Phase we propose to 1) improve the types of invariants that can be detected from existing dynamic analysis methods such as Daikon (and perhaps static analysis method such as Karr's) and 2) hybridize dynamic and static analysis

[1]in this work we only consider deterministic programs, however the approach given can be generalized to undeterministic programs as well
[2]determining if a program halts is *undecidable* in general, however we assume a program loops infinitely if it is not finished within a preset timing threshold
[3]Goldbach's conjecture

to get more precise program specifications (i.e., use information provided by both testcases and source code to strengthen the inferred properties).

4 Fixing Bugs Using Invariants

In previous work, we developed and evaluated GenProg, an approach to automatic program repair. GenProg uses pre-existing test cases and genetic programming (GP) to create new programs that avoids the faulty behavior while retains required functionality. GenProg operates over the program's abstract syntax tree (AST), randomly selects and modifies statements, and uses the number of test cases passed as the fitness of the new programs. While the initial results are promising, challenges remain, namely expressivity–type of programs errors that can be repaired– and scalability –.

We hypothesize that software deffects occur due to violations of program invariants. Assuming the correct properties of the program (or functions or code block in the program) are obtained from Phase 1 (or partially given), we determine the violated properties, i.e., faults localizing. Given an invariant P that holds at location L_1, a set of executable statements $S = \{s_1; \ldots; s_n;\}$, and invariant Q at location L_2, we verify that $\{P\}\ S\ \{Q\}$, i.e., if P holds and the execution S succeeds (i.e., eventually terminates) then Q holds. The verification steps uses *backward propagation* in which we start from Q at L_2 and undo the effects of statements in S to see if P at L_1 can be reached. If we can not then that implies the defect is within S, we modify statements in S to preserve the given properties P and Q.

Consider this example where $P = \{x = 2 * y\}, S = \{\ldots\}, Q = \{x = 10 * y\}$. One way of satisfying Q from P is having $S = \{y = 8*y; x = x+y;\}$, however mistake was made and $S = \{y = 9*y; x = x+y;\}$ instead. We then work backward from $Q = \{x = 10 * y\}$ to see if $P = \{x = 2 * y\}$ can be reached: $(((x' = 10 * y')|^{x'=x+y'})|^{y'=8*y}) \leftrightarrow ((x + y' = 10 * y)|^{y'=9*y}) \leftrightarrow (x + (9 * y) = 10 * y) \leftrightarrow (x = y)$[4]. This result says: after undoing the effects of each statement in S in reversed order, we arrive at $x = y$ instead of $x = 2 * y$ as asserted in P, thus the bug occurs within S.

Next we assume the bug is within one of the statements s_i in S and attempt to modify it so that P and Q hold. We set $s_i =?$, leave the other statements as is, and solve for ? as an equation solver problem. Using the above example, assume statement s_1, $y = 9*y$, is incorrect, we solve for $y =?$. Given $P = \{x = 2*y\}; y =?; x = x+y; Q = \{x = 10*y\}$, we derive several possible solutions for y: $y = 8*y, y = 4*x, y = x+6*y, y = y+7*y, \ldots$. Any of these answers would satisfy the assertions in P, Q. Moreover we could also make the assumption that s_2, $x = x + y$, is incorrect and solves for $x =?$. From $P = \{x = 2 * y\}; y = 9 * y; x =?; Q = \{x = 10 * y\}$, we achieve several possible solutions for x: $x = 10 * y, x = 5 * x, x = x + y - 1/2 * x, \ldots$. Again, any of these answers would fix the bug which violate the invariants P and Q.

Intuitively the technique above is a form of *program synthesis* in which specifications are given and the algorithm automatically generates code to fulfill these properties. These code are not necessarily what human would do (e.g., using $x = y+y+y+y+y$ instead of $x = 5*y$) nor it would make mistake that human would make (in fact, the generated code are provably correct). Due to its extremely large search space, program synthesis remains challenging. Nonetheless we believe the above technique works well for us due to the underlying assumption that the buggy program is very similar to the correct one. We provide almost a complete *skeleton* to program synthesis so that it only has to do very little work.

5 Task Proposals

We list common tasks and those that are specific for each research phase below.

1. **Formalize the Problem and Proposed Directions**.

   First we need to define common notations such as *correct*, *incorrect*, *bugs*, *testsuites*, *invariants* in a formal way. Second, we formulate our proposed solution and attempt to prove it - or at least define it rigorously; i.e., given a problem P' under certain assumptions A, we show our proposed solution S can solve P'.

2. **Taxanomy of Bugs**.

   We classify software defects based on their *causes* to differentiate between those we can and cannot fix.

   Invariants capture the semantics of program which can help us determine *why* bugs occur. Thus a reasonable way to classify bugs is based on their causes (instead of the commonly used classifications of bug *defects*). For example, a cause of an *off-by-one* array indexing produces different defects such as incorrect arithmetic evaluation, segmentation fault, memory violation, etc.

   We concentrate on several common errors below:

   - **Incorrect operand**. E.g., $x = a + c$ instead of $x = a + b$ or $x = a[i] + 1$ instead of $x = a[i-1] + 2$
   - **Incorrect use of operator**. E.g., a instead of $a - 1$, x^2 instead of x^3

[4]Note our variables are renamed during the substitution process, v indicates the original v and v' indicates the modified v

- **Missing or having extra conditions**. E.g., a && b && c instead of a && b or a || b instead of a || b || c
- **Misordering of statements**. E.g., $a = b + c;\ c = a;$ instead of $c = a; a = b + c;$
- **Improper guard**. E.g., if $(a < b) \ldots$ instead of if $(a \leq b) \ldots$

3. **Experiment with Real Program Defects.** The incentive for this proposed approach is based on our experience with fixing *real* software bugs. For instance we observe that common software deffects are among those listed in the Bug Taxanomy section above. Consequently we propose to benchmark our methods on *real* programs. Several available bug suite candidates include those from *Siemens Corp.* containing approximately 150 bugs in C programs (mostly Unix utilities) and from Weimer et al. containing about 20 real world applications (100000+ LoCs). In addition to being real and non-trivial bugs, these testsuites also include adequate testcases to exercise the functionalities of the program (and the bugs).

4. **Phase 1: Understanding Program**

   We propose to do the following tasks

   - **Determine the capabilities of the Daikon invariant detector.** From experience, we believe that Daikon can only capture invariants in limited and simple forms (even if we give it adequate testcases that exercises more precise properties). We propose to send email to the Daikon developers asking if our observations about Daikon capabilities are correct and if it is possible to improve Daikon by extending its framework to detect new and more complex invariant forms.
   - **Invariant Detection**: we propose to 1) improve existing dynamic (and perhaps static) invariant analysis and 2) hybridize both techniques.

     For dynamic analysis, we want to generate more detail and stronger invariant forms than those given by Daikon. Two candidate techniques include *Genetic Programming* or *Equation Solver*. Given the observed data from execution traces, we want to determine the relationship of certain variables x with other variables y, z, e.g., $x = y^3 + 5z - 100$. We use GP to evolve a set of equations (population of individuals), assign higher fitness to equations that evaluate to values with lesser difference than the target value, and make changes via crossover and mutations to achieve new generations. GP is known for its capability to evolve quite complex equations. A more rigorous method (and easier for correctness reasoning) is using Equation Solver that iteratively makes assumption about the form of the equation from simple to more complicated (e.g., 1: linear, 2: quadratic, 3: third degree polynomial, ..) and solve for the correct coefficients that satisfy the observed data.

     If provided with the source code, we can use static analysis methods to provide further information about the relationships within the invariants. For instance if we could determine that x has no relationship whatsoever with y, z or that x is at least y^3 then we can eliminate the unrelated variables and deduce that x might have polynomial degree of at least three. In any case, we reduce the search space which is a vital issue to any heuristic method.

5. **Phase 2: Fixing Bugs**

   - **Bug types.** We create examples of each of the classes of bugs and fix them using backward propagation and equation solver. We start with simple types of bugs (e.g., arithmetic with integers only) and gradually expand to more complex ones. For this part we assume the specifications are provided and our objective is modifying program statements to conform these properties.

     We gradually move to more complex and real problems such as a bug that is caused by multiple statements, a program that contain multiple errors, or arithmetics that are not restricted to integers only.

     Additionally, a language like C contains constructs and data structures such as array and pointer which might be tricky to reason about. For instance how do we know if a variable v has been modified in a code block given that it could be indirectly changed by another variable v' that references v? Techniques used in *program slicing* might be helpful here.
   - **Program Synthesis** We solve or modify program statements by treating statements as unknown equations (e.g., $x = ?$) and solve for them. We want to evaluate how well our approach works and determine how flexible and scalable it is, e.g., can it fix non-trivial bug? what is the time complexity if our equation solver contains large amount of data ? We anticipate to face additional challenges that require more than simple deterministic equation solvers. E.g., we might have to use heuristic constraint solver if we have a large set of equations and variables to solve. Furthermore when applying to a *real* program (instead of simple function), we might have to use some existing *fault localization* technique to help us further localize where the error occurs.

     In addition we also consider other developments in the program synthesis field to see if they are applicable to our work. The *Program Synthesis by Sketching* project from UC. Berkeley is a reasonably well-developed framework that we plan to look at.