Repair with On-The-Fly Program Analysis[⋆]

Robert Könighofer and Roderick Bloem

Institute for Applied Information Processing and Communications (IAIK),
Graz University of Technology, Austria

Abstract. This paper presents a novel automatic repair approach for incorrect programs. It applies formal methods and analyzes program behavior only on demand. We argue that this is beneficial, especially if exhaustive program analysis is infeasible. Our approach computes repair candidates and refines them based on counterexamples. It can be used with various verification techniques and specification formats to check a candidate's correctness. This includes test suites, model checkers verifying assertions, or even the user checking candidates manually, in which case no explicit specification is needed at all. We use concolic execution to analyze programs and SMT-solving to compute repair candidates. We implemented our approach in the open-source debugging environment FoREnSiC and present first experimental results.

Keywords: Program Repair, Formal Methods, Abstraction-Refinement, Concolic Execution, SMT-Solving.

1 Introduction

Debugging is a labor-intensive and costly activity in every software and hardware development process. Errors must be detected, located and fixed. Clearly, automation can reduce effort and costs dramatically. Automatic error detection is already widely used (e.g., model checking or test case generation). Also, automatic error localization is increasingly applied. The actual correction of the error, however, is usually done manually. Yet, fixing an error is often difficult, even if its location is known. This is especially true in other people's code, and if the error has not been tracked down manually. As an illustration, consider the algorithm to compute the greatest common divisor in Section 6.2. There is a bug in line 35, now try to think of a fix. Other difficulties in fixing bugs manually are the danger of eliminating only (some but not all) symptoms, or even introducing new errors. Automatic error correction methods aim to improve this situation.

Formal methods for automatic error correction typically suffer from limited scalability. They often model the correctness of the *entire* program with respect to a given specification in a logic formula [15,19] or game structure [12,13] to compute repairs. More scalable approaches are usually less systematic. They are

[⋆] This work was supported in part by the European Commission through project DIAMOND (FP7-2009-IST-4-248613), and by the Austrian Science Fund (FWF) through the national research network RiSE (S11406-N23).

A. Biere, A. Nahir, and T. Vos (Eds.): HVC 2012, LNCS 7857, pp. 56–71, 2013.

often based on brute-force search guided by genetic mechanisms [1,8] or other heuristics [7,16]. In this work, we attempt to close this gap a little further.

We present a new formal program repair method that addresses the scalability issue by not transforming the entire program into a formula at once. Instead, program analysis is done on-the-fly whenever information about program behavior is missing. We compute repairs by refining a candidate as long as it is still incorrect. If incorrect, we extract a counterexample, i.e., inputs for which the specification is violated, and analyze only program behavior that is possible under this counterexample. This information is then used to refine the repair candidate such that the counterexample is resolved. Our experience shows that, often, a few counterexamples (and corresponding refinements) suffice. This implies that the program needs to be analyzed for a few inputs only.

The main advantage of our new approach is that program analysis focuses on the information needed for repair. For complex programs, exhaustive analysis is usually infeasible or at least inefficient. When setting a bound on the analysis depth (e.g., by limiting the number of loop unrollings) one runs the risk of abstracting away the wrong information. In contrast, the on-the-fly approach is aware of which program behavior can safely be ignored until further notice. Another advantage is the flexibility regarding the verification technique used to check repair candidates and extract counterexamples. It can be a model checker, a test suite, or even a human checking candidates manually. Consequently, there is also flexibility in the specification format. Possibilities are assertions in the code or test vectors together with expected outputs. Assertions can also be used to check a program against a reference implementation. In case of the user checking candidates manually, no explicit specification is needed at all.

We do not expect a complete specification right from the beginning, nor do we suggest to replace humans in the debugging process. We rather strive to assist the user and keep her in the loop. If only incorrect repairs are found due to an incomplete specification, the user needs to refine the specification with additional properties or test cases. This has the nice side-effect that the quality of the specification is improved at the same time.

From a technical perspective, our new program repair method builds on the template-based method introduced in [15], from which we inherit many features. The input is an incorrect program, a specification, and a set of potentially faulty components. The output is a set of replacements of the components such that the specification is satisfied. We assume that the faulty components are identified in a preceding automatic error localization phase. In our implementation we use model-based diagnosis [15], but other diagnosis methods are also possible. Not only single-faults but also multiple faulty components can be handled. Replacements follow templates, which ensures the understandability and maintainability of the repaired program. This is important for keeping the user in the loop. Program analysis is done with concolic execution, repair candidates are computed using a Satisfiability Modulo Theories (SMT) solver. We see the main application of our method in debugging simple software programs, e.g., first software models of hardware designs. Our implementation works on C programs. It is integrated in the open-source debugging environment FoREnSiC [2] and can be downloaded[1]. We also present first experimental results.

This paper is structured as follows. Section 2 discusses related work, and Section 3 briefly explains existing techniques underlying our approach. Our new repair method is presented Section 4. The sections 5 and 6 describe our implementation and present first experimental results. Section 7 concludes the paper.

2 Related Work

The repair method presented in [15] uses templates to synthesize new expressions, symbolic or concolic execution for program analysis, and counterexample-guided refinements for repair. We adopt these strategies and their benefits. However, in [15], program analysis transforms the entire program (and specification) into one large correctness constraint before the repair starts. If complete analysis is infeasible, the number of examined program paths can be limited. Depending on what was omitted, the approach may then find incorrect repairs or no repair at all. Nothing ensures that only irrelevant program behavior is omitted. The novelty of our new approach is that program analysis is done on-the-fly during the repair process, focused towards the required information. Moreover, while [15] only allows `assert` statements in the code, our new approach is flexible regarding the specification and verification technique.

The repair method in [12,13] transforms a finite-state program into a finite-state game and computes repairs as strategies for this game. In [11], this idea is extended to programs with virtually infinite state space using predicate abstraction. To the best of our knowledge, this is the only existing formal error correction method which does not consider the entire program, but uses abstraction instead. Its main shortcoming is that it only considers the predicates found during the preceding verification phase; there is no mechanism to refine the abstraction if it is too coarse for finding a repair. Our method of performing program analysis on-the-fly for one counterexample after the other can also be seen as an abstraction mechanism. The abstraction captures the program behavior for certain inputs only. Our method can also refine an abstraction, simply by analyzing more inputs. Besides the different way of abstracting, we also use a different repair computation method (SMT-based instead of game-based).

Program repair is also related to program sketching [19,18], where a program with missing parts has to be completed in such a way that a given specification is satisfied. In [19], this problem is solved using counterexample-guided refinements, just like in our approach. But also here, program correctness is encoded into one large formula before synthesis starts. The counterexamples are applied to this formula and not to the program to refine candidates. The repair method of [4] also uses counterexample-guided refinements, but for combinational circuits. Less formal repair approaches include methods of repeatedly mutating the incorrect program and checking if it becomes correct [7,16]. Genetic programming methods typically combine mutation with crossing and selection according

[1] `http://www.informatik.uni-bremen.de/agra/eng/forensic.php`

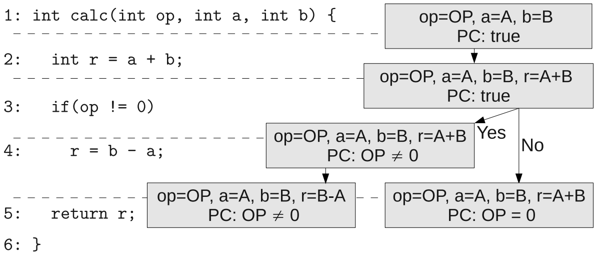
to some notion of fitness [1,8,10]. The work in [6] infers preconditions of methods from passing test cases, and fixes errors by deleting or inserting method calls such that precondition violations are resolved. An extension [21] uses contracts, Boolean query abstraction, and various heuristics.

3 Preliminaries

3.1 Symbolic and Concolic Execution

Symbolic execution [5,14] is a program analysis technique which executes a program using symbols instead of concrete values as inputs. Symbols are placeholders for any concrete value from a given domain. Symbolic execution tracks the symbolic values of all program variables. If a branching point is encountered, a constraint solver is used to check if both branches are feasible. If so, the execution forks. For each execution path, a *path condition* is computed. It evaluates to `true` iff the respective path is activated. For a path that results in a specification violation, the corresponding path condition states when the problem occurs.

Concolic execution [9,17] is a variant of symbolic execution where the program is simultaneously executed on concrete and symbolic input values. The execution path is determined by the concrete values. In parallel, the symbolic values of all program variables are tracked and a symbolic path condition is computed. After one execution run, one conjunct of the path condition is negated and all succeeding conjuncts are discarded. Solving this constraint with a constraint solver gives inputs that trigger a different execution path. A systematic method to negate the different conjuncts of the path condition makes sure that all execution paths are analyzed, or at least a high coverage is obtained [3].



```
1: int calc(int op, int a, int b) {
2:   int r = a + b;
3:   if(op != 0)
4:     r = b - a;
5:   return r;
6: }
```

Fig. 1. Illustration of the concept of symbolic execution

Example 1. Fig. 1 illustrates symbolic execution on a simple program. This program will be used as a running example. It is a simple calculator which can only add and subtract. Boxes contain the state of the symbolic execution, upper-case letters are input symbols, and "PC" indicates the path condition. Concolic execution of this program could start with the concrete values $\mathtt{op} = 0$, $\mathtt{a} = 0$, and $\mathtt{b} = 0$. It would execute the path that skips line 4 and compute the path condition $\mathsf{OP} = 0$. Negating the only conjunct in this path condition and solving the constraints could give the next concrete input values $\mathtt{op} = 1$, $\mathtt{a} = 0$, and $\mathtt{b} = 0$. This triggers the other path with the path condition $\mathsf{OP} \neq 0$.

3.2 Template-Based Repair

We briefly summarize the repair approach of [15], since our current work builds on it. The approach targets simple software, the implementation works on C[2]. `assert`-statements serve as specification. The error is assumed to be an incorrect expression. That is, bugs like missing code or incorrect control flow (like having an `if` instead of a `while`) cannot be handled. The main reason is efficiency.

In a first step, the program is executed symbolically. If an assertion violation is encountered, model-based diagnosis is used to compute sets of potentially faulty expressions. For every set of expressions, error correction then attempts to synthesize replacements. This is reduced to the search for integer constants using repair templates. A repair template is an expression involving program variables and template parameters, which are the unknown constants.

Example 2. Assume that the program in Fig. 1 is supposed to compute `a+b` if `op`=`0`, and `a-b` otherwise. This specification can be formalized with the assertion `assert(op==0 ? r==a+b : r==a-b)`. Assume further that model-based diagnosis identifies the expression `b-a` in line 4 as potentially faulty. This expression is now replaced with a template like `k0 + op*k1 + a*k2 + b*k3 + r*k4`. Finally, the approach computes constant values for the template parameters `k0` to `k4` such that the program satisfies its specification.

Template parameter value computation works as follows. Let $\overline{i}$ be a vector of program inputs, and let $\overline{k}$ be the vector of template parameters. The program and its specification are first transformed into a formula $\mathsf{correct}(\overline{i}, \overline{k})$ which evaluates to true iff the program satisfies the specification when executed on input $\overline{i}$ and repaired with the expression induced by $\overline{k}$. The formula is computed using symbolic or concolic execution. To obtain a repair, one needs to find values for $\overline{k}$ such that for all values of $\overline{i}$, $\mathsf{correct}(\overline{i}, \overline{k})$ holds true. This corresponds to solving $\exists \overline{k} \,.\, \forall \overline{i} \,.\, \mathsf{correct}(\overline{i}, \overline{k})$. To avoid solving this quantifier alternation directly, counterexample-guided repair refinement is performed as illustrated in Fig.2.

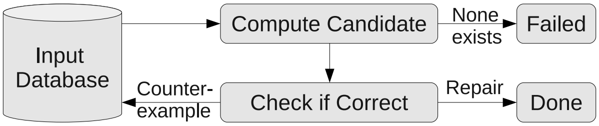


Fig. 2. Counterexample-guided repair refinement, as done in [15]

There is a database I of concrete input vectors $\overline{v_i}$, which is initially empty. In a loop, the following steps are performed. First, a repair candidate is computed as a satisfying assignment $\overline{v_k}$ for the variables $\overline{k}$ in $\bigwedge_{\overline{v_i} \in I} \mathsf{correct}(\overline{v_i}, \overline{k})$. This candidate is correct for the inputs $\overline{v_i} \in I$. Next, the method checks whether the

[2] The implementation handles certain features like pointer arithmetic only approximatively and does not guarantee the repair to be correct in this case.

candidate is correct for *all* inputs by computing a satisfying assignment $\overline{v_i}$ for $\overline{i}$ in $\neg\,\mathsf{correct}(\overline{i},\overline{v_k})$. If no such $\overline{v_i}$ exists, a correct repair has been found. Otherwise, the vector $\overline{v_i}$ is a counterexample for the correctness of the repair candidate $\overline{v_k}$. It is added to I to render the next candidate correct also for this input.

Example 3. For the specification and template from Example 2 we have $\overline{i} = (\mathsf{OP}, A, B)$, $\overline{k} = (k_0, k_1, k_2, k_3, k_4)$, and $\mathsf{correct}(\overline{i},\overline{k}) = (\mathsf{OP} = 0) \vee (k_0 + k_1 \cdot \mathsf{OP} + k_2 \cdot A + k_3 \cdot B + k_4 \cdot (A+B) = A - B)$. The first candidate is arbitrary and could be $\overline{v_k} = (0,0,0,0,0)$, which corresponds to replacing `b-a` in line 4 by `0`. This is not correct and a counterexample is $\overline{v_i} = (1,3,2)$. The next candidate must satisfy $(k_0 + k_1 \cdot 1 + k_2 \cdot 3 + k_3 \cdot 2 + k_4 \cdot 5 = 1)$. A solution, and hence a refined repair candidate, is $\overline{v_k} = (0,0,0,-2,1)$. Since $\neg(\mathsf{OP} = 0 \vee -2 \cdot B + A + B = A - B)$ is unsatisfiable, no counterexample exists for this candidate. Hence, the method would suggest to replace `b-a` in line 4 by `r-2*b`.

A limit to the number of iterations and a time-out for all constraint solving steps ensures termination within reasonable time. If no repair is found using a particular template, the approach switches to a more expressive one. The current implementation includes the linear template of Example 2 and also templates involving bitwise operations and bit shifts. Templates for conditions are currently of the form $t_e\,\mathsf{OP}\,0$ where t_e is a template for a non-Boolean expression and $\mathsf{OP} \in \{<, \leq, >, \geq, =, \neq\}$. The template-based approach ensures that the repairs are understandable, which is crucial for keeping the user in the loop and for the maintainability of the corrected program. Our new approach with on-the-fly program analysis inherits these advantages.

4 Repair with On-The-Fly Program Analysis

The repair method outlined in the previous section first analyzes the entire program and computes one large correctness condition covering the program behavior for all possible inputs and for all possible implementations of the expressions to be synthesized. However, for non-trivial programs, complete program analysis is typically not feasible. Even if feasible, it may take long and produce an unnecessarily large condition. When limiting the program analysis depth (e.g., the maximum number of loop unrollings or execution paths) information needed for finding a correct repair may be missing. Yet, for computing a repair candidate with iterative refinements, the correctness condition needs to be accurate for some inputs only. Only for candidate verification, all inputs need to be covered. However, candidate verification need not be performed on the same formula as candidate computation. A completely different method can be used instead.

We remedy these shortcomings by doing program analysis on-the-fly, analyzing the program only for the counterexamples that have been encountered. Furthermore, we decouple the repair candidate computation from the verification. This allows us to use various verification techniques and specification formats.

4.1 Overview

Fig. 3 outlines our approach. As input, it takes a specification and an incorrect program. The output are repairs in form of expression replacements such that the specification is satisfied. First, potentially faulty expressions are inferred using existing techniques like [15] or [16]. In this work, we focus on repair, i.e., computing replacements. This is sketched in the dashed box of Fig. 3.

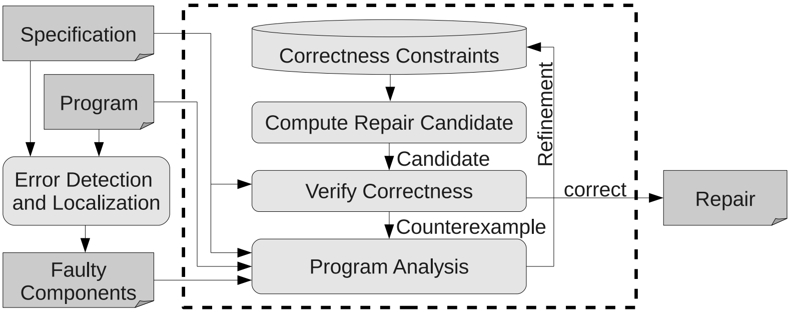


Fig. 3. Overview of our new error correction method

We maintain a database of correctness constraints that must be satisfied by any repair. This database is initially empty. In a loop, we first compute a repair candidate that satisfies these constraints using an SMT-solver. Next, we verify if this candidate satisfies the specification. If not, the verification step returns a counterexample. We analyze the program behavior on this counterexample using concolic execution and add constraints ensuring correctness for this input to the database. In this way, our method keeps analyzing the program for more and more inputs, and improving the repair candidates until a correct one is found. Our experience shows that often a few iterations are enough (see also Section 6).

The next subsections explain the different steps of our method in more detail. Then, we give an example and discuss benefits and limitations of our approach.

4.2 Repair Candidate Computation

Our database contains correctness constraints $\varphi(\overline{k})$ over the template parameter values $\overline{k}$ in some logic (our implementation supports linear integer arithmetic and bitvector arithmetic). We use an SMT-solver with appropriate theory to find a satisfying assignment for these constraints. The concrete values of the template parameters $\overline{k}$ can be mapped back to concrete expressions using the repair templates. This gives a candidate program that can be checked.

4.3 Repair Candidate Verification

The verification of repair candidates can be performed in many ways. The prerequisite is that the verification method is able to produce a counterexample in case of incorrectness. One possibility is to execute a test suite. In our setting, a test suite is a set of input vectors together with corresponding expected output vectors. In addition to the expected outputs, assertions in the code can be used. A counterexample is an input vector together with the corresponding expected output vector such that the actual output does not match the expected one, or an assertion is violated. In case of assertions in the code, the test suite can also consist of input vectors only. A counterexample is then an input vector (together with an empty vector of expected outputs) that triggers an assertion violation.

Another possibility is a model checker taking assertions in the code as specification. Model checkers typically prove incorrectness by giving a counterexample. In our case, this is an input vector for which an assertion is violated. The candidate verification can even be performed by a human. Here, a counterexample could be an input vector together with the expected output. In this case, no explicit specification is needed at all. The repair engine simply learns a fix using the input-output examples given by the user in response to the candidates.

The flexibility in the verification comes with flexibility in the specification. Test cases and assertions have already been mentioned. Assertions can also be used to compare a program with a reference implementation. One simply executes the program and the reference implementation on the same inputs and compares the outcome using suitable assertions. This allows the user to flexibly define what equivalence between the programs means.

4.4 Program Analysis

The crucial step of our approach is program analysis, which is incomplete in our case. We only look at behavior that is possible under one particular input assignment, namely the counterexample found in the preceding verification step.

We take the program where incorrect expressions have already been replaced by templates for new ones, and infer correctness constraints using concolic execution. The inputs are fixed to the concrete values given by the counterexample. Only the template parameters $\overline{k}$ are left open. Symbols represent their yet unknown values. The idea is to execute all feasible paths in this program, and to compute the respective path conditions, which are predicates over $\overline{k}$. Assertions in the code are handled just like other branching points: If an encountered assertion holds for the concrete values of the concolic execution run, the symbolic condition under which it holds is added to the path condition and concolic execution continues. If the assertion is violated, the condition under which this is the case is added and the execution run terminates. Program outputs are handled similarly. Whenever the program outputs a value (which can be modeled by a call to a special function `output(x)`), a conjunct is added to the path condition. If the concrete output matches the expected value, the conjunct states when this is the case. Otherwise, the conjunct expresses when the output does not match and the execution run terminates. If no expected output values are included in the counterexample (see Section 4.3), the output of the program is ignored.

The concolic execution runs are now divided into failing and passing runs. A *failing run* is one that either terminates in an assertion violation or with a mismatch between the actual and expected output. The corresponding path condition states when this happens. A *passing run* terminates without violating the specification. Let F be the set of failing runs and $PC^r(\overline{k})$ be the path condition of run r. We compute a necessary condition for program correctness as

$$\varphi(\overline{k}) = \neg \bigvee_{r \in F} PC^r(\overline{k}).$$

This condition is added to the database of correctness constraints. It will exclude candidates that fail on the input vector for which the program has been analyzed.

Since the inputs to the program are fixed, typically many execution paths become infeasible. Nevertheless, the set of execution paths may still be very large or even infinite. One reason for an infinite number of paths can be a loop where the termination condition depends on the implementation of the components to be synthesized, i.e., on the template parameter values. This problem arises also in [15], with the open inputs making the situation even worse. Just like [15], we address this problem by limiting the program analysis depth with a user-given bound on the number and length of execution paths to consider. A consequence of these limits can be that the repair refinement loop does not converge. However, since there is also a bound on the number of refinement iterations, the program will terminate and the user can try again with higher limits. In general, our approach is tailored towards finding repairs efficiently for many cases instead of having a sound and complete method that does not scale.

4.5 Example

Example 4. Assume that the program from Example 1 is specified with the test cases ($\texttt{op=}0, \texttt{a=}3, \texttt{b=}5 \rightarrow \texttt{r=}8$), $(1, 5, 3 \rightarrow 2)$, and $(1, 6, 1 \rightarrow 5)$. We use the template from Example 2. Initially, the database of correctness constraints is empty, so the first repair candidate is arbitrary. It could be $\overline{v_k} = (0, 0, 0, 0, 0)$, which means that `b-a` in line 4 of the program is replaced by `0`. Verifying this candidate with the test cases, we get $(1, 5, 3 \rightarrow 2)$ as counterexample. We now use concolic execution to analyze the program behavior for the input vector $(1, 5, 3)$. Only the execution path including line 4 is feasible. After line 4, `r` has the symbolic value $k_0+k_1{\cdot}1+k_2{\cdot}5+k_3{\cdot}3+k_4{\cdot}8$, because `b-a` has been replaced by the template. Taking `r` as the output and 2 as expected value, concolic execution distinguishes two cases, which are activated with two concolic execution runs r_1 and r_2. The run r_1 is passing and has the path condition $PC^{r_1}(\overline{k}) = k_0+k_1{\cdot}1+k_2{\cdot}5+k_3{\cdot}3+k_4{\cdot}8 = 2$. The run r_2 is failing and has $PC^{r_2}(\overline{k}) = k_0 + k_1 \cdot 1 + k_2 \cdot 5 + k_3 \cdot 3 + k_4 \cdot 8 \neq 2$. Since $F = \{r_2\}$, we add $\varphi_1(\overline{k}) = k_0 + k_1 + k_2 \cdot 5 + k_3 \cdot 3 + k_4 \cdot 8 = 2$ to the database of correctness constraints. The next iteration starts. Now, the candidate has to satisfy $\varphi_1(\overline{k})$. A solution is $\overline{v_k} = (2, 0, 0, 0, 0)$, which corresponds to replacing `b-a` in line 4 by `2`. This candidate fails on the test case $(1, 6, 1 \rightarrow 5)$. For this counterexample, concolic execution produces the correctness constraint $\varphi_2(\overline{k}) = k_0 + k_1 + k_2 \cdot 6 + k_3 \cdot 1 + k_4 \cdot 7 = 5$. It is added to the database and another iteration starts. The next repair candidate must fulfill $\varphi_1(\overline{k})$ and $\varphi_2(\overline{k})$.

A solution is $\overline{v_k} = (0, 0, 1, -1, 0)$, which means that `b-a` is replaced by `a-b`. This candidate passes all tests and is presented to the user.

4.6 Discussion

This section discusses the main benefits and limitations of our new repair method.

More Focused Program Analysis: The main advantage of our new repair method is that program analysis is very focused towards the information needed for computing repair candidates. Complete analysis is infeasible for complex programs. In [15], this issue is addresses by setting a limit on the number and length of the execution paths to consider. However, since there is no guidance on what to analyze, this limit can render the probability of obtaining the information relevant for finding a repair very low. In contrast, our new repair method analyzes the program only for the counterexamples that are relevant for the repair finding process. There is also a bound on the number and length of the execution paths. However, since these limits apply locally for each invocation, our new approach learns at least something about the behavior under each counterexample.

Simpler Program Analysis: Compared to [15], our new method renders program analysis with concolic execution simpler because the inputs are always fixed to one counterexample at a time. This does not only drastically reduce the number of feasible execution paths, it also simplifies the analysis per concolic execution run. We can start to track the symbolic values of the program variables only after a repair template with unknown parameters has been executed for the first time. In particular, if a reference implementation is used as a specification, our new approach needs to execute the entire reference code with concrete variable values only (see Section 6.2 for such a scenario). The approach without on-the-fly analysis needs to track the symbolic values right from the beginning because the inputs are not fixed but have a symbolic value.

Flexibility in the Specification: From the user's perspective, the most important benefit is probably the flexibility in the specification of the desired behavior. Existing formal correction approaches often support assertions only. However, writing assertions which accurately reflect the desired behavior and do not only check for basic properties is difficult. Test cases (possibly together with some assertions) are often more natural. This flexibility is also important for keeping the user in the loop. Writing additional test cases if only incorrect repairs are produced is often simpler than coming up with better assertions.

Better Scalability: Our method addresses the scalability issue, which is common for all formal error correction approaches, from several sides. Doing program analysis for typically only a few concrete counterexamples has already been mentioned. The flexibility in the technique for verifying repair candidates is another factor. Where formal approaches like model checking or symbolic execution fail, test case execution can still produce meaningful results.

Limitations: A drawback of the separation of concerns is that little information (only the counterexample) is passed from the verification phase to the program analysis phase. Furthermore, certain program paths may be feasible under several counterexamples, and may thus be analyzed multiple times using concolic execution. The limitation of the approach to incorrect expressions can be easily weakened in principle, but at the cost of efficiency. Whenever there are finitely many (and not too many) options to replace a certain construct in the program, we can analyze all of them. This way, we could handle bugs like having an `if` instead of a `while`, or bugs in the left-hand side of an assignment.

5 Implementation

We implemented our new repair method as a proof-of-concept in the open-source debugging environment FoREnSiC [2], re-using the provided infrastructure and parts of the implementation of [15]. Our new repair engine is integrated with the existing error localization engine. So far, we implemented two mechanisms to verify the correctness of repair candidates. The fist one is test case execution, operating on a given set of input vectors together with corresponding expected output values. Assertions in the code can be used in addition or as alternative to the expected outputs. The second verification mechanism uses concolic execution to compute one large correctness condition, just like [15], and uses this condition to verify correctness with an SMT-solver query. This second mechanism was implemented mainly to have a fair comparison with the existing technique. In the future, we also plan to implement verification engines using software model checkers such as CBMC or SATABS. We also want to implement an interface which asks the user to verify correctness. It would be interesting to see whether this way of doing semi-automatic program repair is useful and not too laborious.

Our implementation is able to use the SMT-solvers Yices and Z3, either via their C-API or via the SMT-LIBv2 format. We support linear integer arithmetic as well as bitvector arithmetic. The concolic execution engine we use for program analysis is an extension of CREST [3]. Our repair method also implements the heuristics of [15] for preferring simple repairs using Maximum Satisfiability (MAX-SAT) solving. Besides the source code, the FoREnSiC archive also contains the scripts to reproduce our experimental results.

6 Experimental Results

In this section, we experimentally compare our new repair method with the method of [15] to support the following informal claim.

Claim. *If program analysis is done before repair starts, the analysis needs to be fairly detailed to deliver the information required for finding a repair with counterexample-guided refinements. Repair with on-the-fly program analysis requires only a fractional amount of this information about the program behavior.*

This property is important because complex programs cannot be analyzed exhaustively. Section 6.2 shows an example were the method of [15] even fails because upfront program analysis does not produce the required information within reasonable time.

One could expect that our new repair method is also significantly faster because the constraints that are used for computing repair candidates are typically much smaller, and should hence be easier to solve. Unfortunately, if the constraints do not lack information that is needed for repair, this does not hold true. The reason is that modern constraint solvers, especially SMT-solvers, are good in ignoring information that is not needed. The additional time they require for parsing and simplifying the large formula is usually not so significant.

6.1 Performance Results

Table 1 summarizes performance results for repairing different faulty versions of the `tcas` program from the Siemens suite [20]. The `tcas` program implements a traffic collision avoidance system for aircrafts. It has about 180 lines of code, 12 integer inputs and one output. It comes in 41 faulty versions, together with a reference implementation and 1608 test cases. Table 1 only contains those faulty versions for which our fault model (incorrect expressions) applies. Versions with missing code or incorrect control flow are not considered. An exception are the versions `tcas21`, `tcas22`, and `tcas23`. They feature a missing function call, but this can be compensated by modifying an expression. Table 1 only compares the error correction step, assuming perfect information about the error location.

The columns 1 to 4 contain results for our on-the-fly repair method using test case execution to verify repair candidates. Column 1 indicates whether a repair could be found. Column 2 gives the number of execution paths that had to be analyzed using concolic execution to find a correct repair candidate. The number of iterations of the repair refinement loop is listed in Column 3. Column 4 shows the overall repair time (including program analysis and candidate verification). The columns 5 to 8 contain exactly the same information for the on-the-fly method that verifies candidates using a correctness formula expressing equivalence with the reference implementation (see Section 5). Finally, the columns 9 to 12 show results of the method without on-the-fly analysis. The specification is the same, namely an assertion requiring equivalence with the reference implementation. Column 10 gives the minimum number of execution paths that need to be analyzed for the method to find a repair. For this number of analyzed execution paths, the last two columns list the number of iterations of the repair refinement loop and the overall repair time, respectively.

All experiments were performed on an Intel P7350 processor with 2 × 2.0 GHz and 3 GB RAM, running 32-bit Linux. As SMT-solver we used Z3 version 3.1 with linear integer arithmetic, interfaced via its SMT-LIBv2 interface. A time-out of 60 seconds was set for all SMT-solver calls.

Discussion

What stands out in Table 1 is that repair with on-the-fly program analysis and test cases is able to fix all benchmarks, and is significantly faster than the

Table 1. Performance results

Col.	1	2	3	4	5	6	7	8	9	10	11	12
	On-the-fly with testing				On-the-fly with equivalence checking				Method of [15]			
	repair found	# paths analyzed	# iterations	repair time	repair found	# paths analyzed	# iterations	repair time	repair found	min. # paths	# iterations	repair time
	[-]	[-]	[-]	[sec]	[-]	[-]	[-]	[sec]	[-]	[-]	[-]	[sec]
tcas01	yes	16	9	23	no	-	-	-	yes	337	8	65
tcas02	yes	40	11	30	yes	8	3	12	yes	753	5	26
tcas06	yes	32	5	12	yes	60	8	79	yes	1393	7	55
tcas07	yes	4	3	10	yes	2	2	6	yes	305	3	11
tcas08	yes	12	7	18	yes	32	17	38	yes	305	5	17
tcas09	yes	6	4	13	yes	8	5	28	yes	593	6	41
tcas10	yes	104	10	32	no	-	-	-	no	-	-	-
tcas13	yes	14	8	33	no	-	-	-	no	-	-	-
tcas14	yes	24	13	71	no	-	-	-	no	-	-	-
tcas16	yes	2	2	9	yes	2	2	6	yes	305	2	9
tcas17	yes	4	3	10	yes	2	2	6	yes	305	3	12
tcas18	yes	2	2	8	yes	34	18	40	yes	305	4	14
tcas19	yes	12	7	18	yes	32	17	37	yes	305	5	18
tcas20	yes	10	6	15	yes	8	5	26	yes	593	9	85
tcas21	yes	32	17	206	no	-	-	-	no	-	-	-
tcas22	yes	32	17	206	no	-	-	-	no	-	-	-
tcas23	yes	22	12	113	no	-	-	-	no	-	-	-
tcas24	yes	22	12	112	no	-	-	-	no	-	-	-
tcas25	yes	10	6	17	yes	14	8	100	yes	337	7	82
tcas28	yes	8	3	10	yes	20	6	35	yes	1329	4	34
tcas35	yes	8	3	9	yes	20	6	46	yes	1329	5	41
tcas36	yes	2	2	7	yes	2	2	6	yes	305	2	8
tcas39	yes	10	6	17	yes	14	8	101	yes	337	7	82
avg.	100%	18.6	7.3	43	65 %	17.2	4.7	38	70 %	397	5.1	38

other methods. Of course, the 1608 test cases form a less restrictive specification than equivalence with the reference implementation. However, manual analysis of the computed repairs showed that they are reasonable – they do not just exhibit "holes" in the test suite. This illustrates that repair with test cases as specification can be useful and has the potential to scale better.

The columns 3, 7, and 11 show that a few iterations of the repair refinement loop are often enough to find a repair. Our new repair method with on-the-fly program analysis exploits this circumstance by doing program analysis only for the few counterexamples that show up. On average, it analyzes only 18 execution paths (see Column 2 and 6). Without on-the-fly analysis, at least 397 execution paths need to be analyzed on average (Column 10). These observations confirm the Claim about analysis depth postulated earlier. With the same mechanism to verify the correctness of repair candidates, the running times are almost the same (Column 8 vs. 12). The scalability benefits of our new method become more evident when exhaustive program analysis is not feasible anymore, as illustrated in the next section.

6.2 Greatest Common Divisor Example

The `tcas` example in the previous section fits the repair method with upfront program analysis well because it has only a finite (and small) number of execution paths. Let us now increase the level of difficulty. Consider the following C code implementing a sophisticated algorithm to compute the Greatest Common Divisor of two integers in the function `gcd`. The code also contains the Euclidean algorithm as reference implementation (`gcdR`), and an equivalence assertion in line 19. The `gcd` implementation contains a bug which is not easy to see and even more difficult to fix: line 35 should read `u - v` instead of `u >> 1`.

```
 1  #include <assert.h>
 2  #include <forensic.h>
 3  #define UI unsigned int
 4
 5  //<ASSUME_CORRECT>
 6  UI gcd(UI u, UI v);
 7  UI gcdR(UI a, UI b) {
 8   if(a == 0)     return b;
 9   while(b != 0){
10    if(a > b)     a = a - b;
11    else          b = b - a;
12   }
13   return a;
14  }
15  void main() {
16   UI a, b;
17   FORENSIC_input_UI(a);
18   FORENSIC_input_UI(b);
19   assert(gcdR(a,b) ==
                 gcd(a,b));
20  } //</ASSUME_CORRECT>
```

```
21  UI gcd(UI u, UI v) {
22   UI s = 0;
23   if(u == 0 || v == 0)
24    return u | v;
25   for(;((u|v)&1)==0;++s){
26    u >>= 1; v >>= 1;
27   }
28   while((u & 1) == 0)
29    u >>= 1;
30   do {
31    while((v & 1) == 0)
32     v >>= 1;
33    if(u <= v) { v -= u;
34    } else {
35     UI tmp = u >> 1;
36     u = v; v = tmp;
37    }
38    v >>= 1;
39   } while(v != 0);
40   return u << s;
41  }
```

We first apply our new repair method with on-the-fly program analysis and test-based repair candidate verification. As test inputs, we simply take all pairs a, b with $0 \leq a, b < 100$. The number 100 was chosen arbitrarily, the correct repair is also found with lower numbers like 15. For program analysis, we do not limit the number of execution paths to analyze, but rather their length. With two or three invocations of our method, we found out that a length of 55 is enough.[3] Using these parameters and Z3 with bitvector arithmetic, our new

[3] This number roughly corresponds to the number of executed statements.

method found the sequence of repair candidates "`873`", "`85`", "`u - 2`", and "`u - v`" in 101 seconds. The last one, "`u - v`", was found to be correct. Only 1570 program paths had to be analyzed. With a more careful choice of the parameters, a correct repair can also be found with less than 1000 paths analyzed.

We failed in applying the method with upfront program analysis to find a repair. Again, we experimented with an increasing maximum execution path length during program analysis. With a maximum length of 75, the faulty statement was not even executed in such a way that a wrong result was produced. With a maximum length of 80, we were already analyzing 10 402 execution paths, which took more than half an hour. Still, the program analysis was so inaccurate that only incorrect repairs (usually replacing the faulty expression with some constant) were found.[4]

Discussion

This example nicely demonstrates the scalability benefits of our new approach. Complete program analysis is simply infeasible for the `gcd` program. The many loops and branching points lead to huge numbers of possible execution paths, even if one considers only paths of relatively short length. Hence, one can only analyze small parts of the program behavior. When doing random program analysis before repair starts, it is very unlikely that the information needed for repair is obtained. With on-the-fly program analysis one cannot completely analyze the program either, not even for the counterexamples of interest. However, focusing on these counterexamples helps to extract enough information to find repairs.

7 Conclusion

In this work, we presented a novel method for automatic error correction in simple software programs using on-the-fly program analysis. In contrast to existing repair methods which perform program analysis before the repair process starts, our approach analyzes only those parts of the behavior of the program that are relevant for finding a repair. This is important if exhaustive program analysis is infeasible. Not looking at the entire program right from the beginning can be seen as an abstraction method. Unlike existing methods that use abstraction for repair [11], our method can also refine the abstraction. Compared to existing formal methods, our new approach is also more flexible regarding the form of specification and the correctness verification method. Assertions and test cases can be used as well as on-line feedback from the user. All this contributes towards making automatic error correction more practicable.

In the future, we plan to address limitations of the approach and its implementation by experimenting with fault models that go beyond incorrect expressions,

[4] The reader may wonder why no repair is found although the path length is already higher than with on-the-fly analysis. The reason is that we only consider symbolic operations for the path length, so the absolute values cannot be compared. In the on-the-fly approach, the symbolic computation begins only when the repair template (i.e., line 35) is executed for the first time. See also Section 4.6.

interfacing additional verification engines, and implementing memory models to accurately reason about array operations and pointer arithmetic.

References

1. Arcuri, A.: On the automation of fixing software bugs. In: ICSE, pp. 1003–1006. ACM (2008)
2. Bloem, R., Drechsler, R., Fey, G., Finder, A., Hofferek, G., Könighofer, R., Raik, J., Repinski, U., Sülflow, A.: FoREnSiC – An automatic debugging environment for C programs. In: Biere, A., Nahir, A., Vos, T. (eds.) HVC 2012. LNCS, vol. 7857, pp. 260–265. Springer, Heidelberg (2013)
3. Burnim, J., Sen, K.: Heuristics for scalable dynamic test generation. In: ASE, pp. 443–446. IEEE (2008)
4. Chang, K.-H., Markov, I.L., Bertacco, V.: Fixing design errors with counterexamples and resynthesis. In: ASP-DAC, pp. 944–949. IEEE (2007)
5. Clarke, L.A.: A system to generate test data and symbolically execute programs. IEEE Trans. Software Eng. 2(3), 215–222 (1976)
6. Dallmeier, V., Zeller, A., Meyer, B.: Generating fixes from object behavior anomalies. In: ASE, pp. 550–554. IEEE (2009)
7. Debroy, V., Wong, W.E.: Using mutation to automatically suggest fixes for faulty programs. In: ICST, pp. 65–74. IEEE (2010)
8. Forrest, S., Nguyen, T., Weimer, W., Le Goues, C.: A genetic programming approach to automated software repair. In: GECCO, pp. 947–954. ACM (2009)
9. Godefroid, P., Klarlund, N., Sen, K.: DART: Directed automated random testing. In: PLDI, pp. 213–223. ACM (2005)
10. Le Goues, C., Nguyen, T., Forrest, S., Weimer, W.: GenProg: A generic method for automatic software repair. IEEE Trans. Software Eng. 38(1), 54–72 (2012)
11. Griesmayer, A., Bloem, R., Cook, B.: Repair of Boolean programs with an application to C. In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 358–371. Springer, Heidelberg (2006)
12. Jobstmann, B., Griesmayer, A., Bloem, R.: Program repair as a game. In: Etessami, K., Rajamani, S.K. (eds.) CAV 2005. LNCS, vol. 3576, pp. 226–238. Springer, Heidelberg (2005)
13. Jobstmann, B., Staber, S., Griesmayer, A., Bloem, R.: Finding and fixing faults. Journal of Computer and System Sciences 78(2), 441–460 (2012)
14. King, J.C.: Symbolic execution and program testing. Communications of the ACM 19(7), 385–394 (1976)
15. Koenighofer, R., Bloem, R.: Automated error localization and correction for imperative programs. In: FMCAD, pp. 91–100. IEEE (2011)
16. Raik, J., Repinski, U., Hantson, H., Jenihhin, M., Di Guglielmo, G., Pravadelli, G., Fummi, F.: Combining dynamic slicing and mutation operators for ESL correction. In: ETS, pp. 1–6. IEEE (2012)
17. Sen, K., Marinov, D., Agha, G.: CUTE: A concolic unit testing engine for C. In: ESEC/FSE, pp. 263–272. ACM (2005)
18. Solar-Lezama, A.: The sketching approach to program synthesis. In: Hu, Z. (ed.) APLAS 2009. LNCS, vol. 5904, pp. 4–13. Springer, Heidelberg (2009)
19. Solar-Lezama, A., Tancau, L., Bodik, R., Saraswat, V., Seshia, S.A.: Combinatorial sketching for finite programs. In: ASPLOS, pp. 404–415. ACM (2006)
20. Siemens suite, `http://pleuma.cc.gatech.edu/aristotle/Tools/subjects`
21. Wei, Y., Pei, Y., Furia, C.A., Silva, L.S., Buchholz, S., Meyer, B., Zeller, A.: Automated fixing of programs with contracts. In: ISSTA, pp. 61–72. ACM (2010)