Automated Program Repair Using Formal Verification Techniques

Hadar Frenkel[1], Orna Grumberg[2] , Bat-Chen Rothenberg[2] and Sarai Sheinvald[3]

[1] CISPA Helmholtz Center for Information Security, Saarbrücken, Germany
[2] Department of Computer Science, The Technion, Haifa, Israel
[3] Department of Software Engineering, Braude College of Engineering, Israel

Abstract. We focus on two different approaches to automatic program repair, based on formal verification methods. Both repair techniques consider infinite-state C-like programs, and consist of a generate-validate loop, in which potentially repaired programs are repeatedly generated and verified. Both approaches are *incremental* – partial information gathered in previous verification attempts is used in the next steps. However, the settings of both approaches, including their techniques for finding repairs, are quite distinct. The first approach uses syntactic mutations to repair sequential programs with respect to assertions in the code. It is based on a reduction to the problem of finding unsatisfiable sets of constraints, which is addressed using an interplay between SAT and SMT solvers. A novel notion of *must-fault-localization* enables efficient pruning of the search space, without losing any potential repair. The second approach uses an Assume-Guarantee (AG) style reasoning in order to verify large programs, composed of two concurrent components. The AG reasoning is based on automata-learning techniques. When verification fails, the procedure repeatedly repairs one of the components, until a correct repair is found. Several different repair methods are considered, trading off precision and convergence to a correct repair.

1 Introduction

This work is concerned with automated program repair. It focuses on two specific approaches, presented in [50,48] and [22,21], that demonstrate many of the guiding principles in program repair, when it is based on formal methods. While the two approaches have much in common, they are also quite distinct, due to their different settings, including their type of programs, specifications and repair mechanisms.

Both approaches handle infinite-state C-like programs, for which both the syntax and the semantics must be taken into account. The syntax refers to the program code, which might be updated for the purpose of repair. In [50] a predefined set of mutations is used for syntactic update, where [22] uses *abduction* to derive constraints that are added to the program code. In both cases, SMT solvers are used to answer semantic questions that arise during verification.

As often with program repair, the entire process can be seen as a *generate-validate* loop. *Generate* produces a candidate program, and *validate* checks whether it is a *good repair*, that is, whether the candidate program satisfies the given specification.

In order to prune the search space of candidate programs when validation fails, the goal is not only to remove the failed candidate program, but also to remove "similar" candidates that are likely to fail as well.

In [50] the search space consists of all mutated programs and the goal is to return *all* good repairs that are minimal. A notion of *must-fault localization* is developed in order to guarantee that similarly failed programs will not be considered in the future. This makes the repair process much more efficient.

In [22], the search space consists of sets of executions of the original program, which can be represented by a Control Flow Graph (CFG). Once the current program fails to satisfy the specification, faulty executions are removed by altering the CFG of the program. Several repair methods are proposed, some may remove more executions than necessary. This allows to trade efficiency for completeness. This approach too is *incremental*, meaning that the current validation step makes use of previous validation steps, thus increasing efficiency.

As stated, the differences between the two approaches are quite significant. [50] exploits a predefined set of mutations for repair. Its goal is to return all minimal repairs and its focus is on the notion of must-fault localization, which achieves efficiency and completeness. Its verification notion is bounded. [22], on the other hand, focuses on making the validation step more scalable. To this end, it exploits the Assume-Guarantee (AG) learning-based paradigm for compositional verification [39,46] and adapts it to the setting of infinite-state communicating C programs. The CFG of the verified program is viewed as an automaton, in order to enable automata-learning (e.g., via $\mathbf{L}^*$ [4]). Its verification is unbounded.

Next, we present a high-level description of each of the approaches, followed by a more detailed description.

1.1 The Must-Fault Localization Approach

The first approach we present focuses on repair of imperative, sequential, programs with respect to assertions in the code. We use a bounded notion of correctness. That is, for a given bound wb, we consider only *bounded computations*, along which the body of each loop is entered at most wb times and the maximum depth of the call stack is wb. We say that a program is *repaired* if whenever a bounded computation reaches an assertion, the assertion is evaluated to true.

Our repair method is *sound*, meaning that every returned program is repaired (i.e., no violation occurs in it up to the given bound). Just like Bounded Model Checking, this increases our confidence in the returned program.

Our programs are repaired using a predefined set of mutations, applied to expressions in conditionals and assignments (e.g., replacing a $+$ operator by a $-$), as was shown useful in previous work [16,47]. We impose no assumptions on the number of mutations needed to repair the program and are able to produce repairs involving multiple buggy locations, possibly co-dependent. To make sure that our suggested repairs are as close to the original program as possible, the candidate repaired programs are examined and returned in increasing number of mutations. In addition, only *minimal* sets of mutations are taken into account. That is, if a program can be repaired by applying a set of mutations Mut, then no superset of Mut is later considered. Intuitively, this is our way to make sure all changes made to the program by a certain repair are indeed necessary.

Our method is *complete* in the sense of returning *all* minimal sets of mutations that create a repaired program. Specifically, if no repair is found, one can conclude that the given set of mutations is not enough to repair the program.

Our algorithm, FL-AllRepair, is based on the translation of the program into a set of SMT constraints called the *program formula*, which is satisfiable iff the program contains an assertion violation. This was originally done for the purpose of bounded model checking in [11]. Our key observation is that mutating an expression in the program corresponds to replacing a constraint in the set of constraints encoding the program. Thus, searching the space of mutated programs is reduced to searching unsatisfiable sets of constraints.

The search is conducted using an interplay between SAT and SMT solvers, which realizes a generate-validate loop: The SAT solver is used to sample the search space of mutated programs and to efficiently block sets of undesired programs. The SMT solver is used to verify whether a mutated program is repaired.

Two key factors make this search process efficient: incremental solving, and pruning via blocking. Incremental solving is used in both the SAT solver and the SMT solver, which means that each of them retains learned information between successive calls. Using an SMT solver incrementally constitutes a novel way to exploit information learned while checking the correctness of one program for the process of checking correctness of another program. Note, that if the programs are similar, their encoding as sets of SMT constraints will also be similar (due to our observation presented above), resulting in bigger savings when using incremental SMT.

The second key contributing factor to efficiency is pruning. Pruning occurs after the validate stage, based on its results. Whenever a program is found to be repaired, we use it to prune other mutated programs based on non-minimality. If, however, the program is found to be buggy (i.e., not repaired) our algorithm makes use of fault localization to prune other buggy programs.

Although fault localization and automated program repair have long been combined, our use of fault localization to block undesired programs is non-standard. Traditionally, fault localization suggests a set of locations F in the program that might be the cause of the bug. Then, repair attempts to change those suspicious locations in order to eliminate the bug. Pruning based on such an approach would mean blocking mutated programs where lines outside of F are changed. However if fault localization is too restrictive (i.e., F is too small), we will be missing potential repairs. In fact, a recent study has shown that for test-based repair imprecise fault localizations happen very often in practice [34]. On the other hand, if fault localization is too permissive, this blocking might cause redundant search work.

This identifies the need for fault localization that can narrow down the space of candidates while still promising not to lose potential causes for a bug. For this purpose, we define the concept of a *must* location set. Intuitively, such a set includes at least one location from every repair for the bug. Thus, it *must* be used for repair. In other words, **it is impossible to fix the bug using only locations outside this set**. A fault localization technique is considered a *must* algorithm if it returns a must location set for every buggy program and every bug in the program.

The blocking done in our repair process whenever a buggy mutated program is discovered is based on a must-fault-localization algorithm. This blocking ensures that we do not lose repairs, and therefore do not damage completeness.

We implemented FL-AllRepair in an open-source tool available on GitHub [4], compared it with the methods of [29,30] and got very encouraging results.

1.2 The Assume-Guarantee-Repair (AGR) Approach

The second approach focuses on the Assume-Guarantee (AG) style compositional verification [39,46], which enables making the verification of large systems more scalable. The simplest AG rule checks whether a system composed of components M_1 and M_2 satisfies a property P by checking that M_1 along with an assumption A satisfies P, and that any system containing M_2 as a component satisfies A. Several frameworks have been proposed to support this style of reasoning. Finding a suitable assumption A is a common challenge in such frameworks.

Our fully-automated framework, called *Assume-Guarantee-Repair* (AGR), applies the Assume-Guarantee rule, and while seeking a suitable assumption A, iteratively repairs the given program in case the verification fails. Our framework is inspired by [44], which presented a learning-based method for finding an assumption A for finite-state programs represented by labeled transition systems (LTS). The assumptions are found using the $\mathbf{L}^*$ [4] algorithm for learning regular languages.

In contrast to [44], our AGR framework handles *communicating programs*. These are infinite-state C-like programs, extended with the ability to synchronously read and write data over communication channels. We model such programs as finite-word automata over an *action alphabet*, which reflects the program statements. The accepting states in the automaton model points of interest in the program that the specification can relate to. The automata representation, which enables exploiting automata-learning algorithms, is similar in nature to that of control-flow graphs.

The composition of the two program components M_1 and M_2, denoted $M_1||M_2$, synchronizes on read-write actions on the same channel. Between two synchronized actions, the individual actions of both systems interleave. Fig. 1 presents the code of a communicating program (left) and its corresponding automaton M_2 (right). The automaton alphabet consists of constraints, assignment actions, and communication actions. For example, $enc!x_{pw}$ sends the value of variable x_{pw} over channel enc, and $getEnc?x_{pw2}$ reads a value to x_{pw2} on channel $getEnc$.

The specification P is modeled as an automaton that does not contain assignment actions. It may contain communication actions in order to specify behavioral requirements, as well as constraints over the variables of both system components, that express requirements on their values in various points in the runs.

Consider, for example, the program M_1 and the specification P seen in Fig. 2, and the program M_2 of Fig. 1. M_2 reads a password on channel $read$ to the variable x_{pw}, and once the password is long enough (at least four digits), M_2 sends the value of x_{pw} to M_1 through channel enc. The component M_1 reads this value to variable y_{pw} and

---

[4] Fl-AllRepair is an extension of the AllRepair tool, available here: `https://github.com/batchenRothenberg/AllRepair`. FL-AllRepair is currently enabled by adding the `--blockrepair slicing` option to the AllRepair tool.

```
while(true)
    password:=readInput;
    while(password≤ 999)
        password:=readInput;
    password2:=encrypt(password);
```

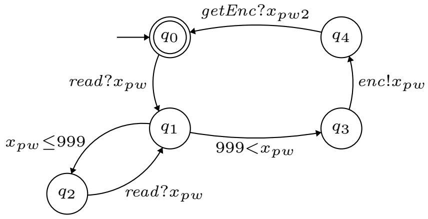

Fig. 1: Modeling a communicating program as an automaton M_2

then applies a simple function that changes its value, and sends the changed variable back to M_2. The property P reasons about the parallel run of the two programs. The pair $(getEnc?x_{pw2}, getEnc!y_{pw})$ denotes a synchronization of M_1 and M_2 on channel $getEnc$. The specification P requests that the parallel run of M_1 and M_2 first reads a value and only then encrypts it – a temporal requirement. In addition, it makes sure that the value after encryption is different from the original value, and that there is no overflow – both are semantic requirements on the program variables. In case that one of the requirements does not hold, P reaches the error state r_4. Note that P here is not complete, for simplicity of presentation.

The $\mathbf{L}^*$ algorithm aims at learning a regular language U. Its entities consist of a *teacher* – an oracle that answers *membership queries* ("is the word w in U?") and *equivalence queries* ("is $\mathcal{A}$ an automaton whose language is U?"), and a *learner*, which iteratively constructs a finite deterministic automaton $\mathcal{A}$ for U by submitting a sequence of membership and equivalence queries to the teacher. In using the $\mathbf{L}^*$ algorithm for learning an assumption A for the AG-rule, membership queries are answered according to the specification P: A trace t should be in A iff $M_1||t$ satisfies P. Once the learner constructs a stable system A, it submits an equivalence query. The teacher then checks whether $M_1||A$ satisfies P, and whether the language of M_2 is contained in the language of A. According to the results, the process either continues or halts with an answer to the verification problem. The learning procedure aims at learning the weakest assumption A_w, which contains all the traces that in parallel with M_1 satisfy P . The key observation that guarantees termination in [44] is that the components in this procedure – M_1, M_2, P and A_w – are all regular.

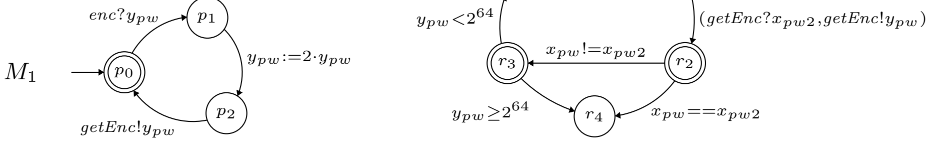

Fig. 2: The program M_1 and the specification P

Our setting is more complicated, since the traces in the components – both the programs and the specification – contain constraints, which are to be checked semantically. These constraints may cause some traces to become infeasible. For example, if a trace contains an assignment $x := 3$ followed by a constraint $x \geq 4$ (modeling an "if" statement), then this trace does not contribute any concrete runs, and therefore does not affect the system behavior. Thus, we must add feasibility checks to the process, and there is more here to check than standard language containment. Moreover, in our setting A_w above may no longer be regular, see Example 3. However, our method manages overcoming this problem.

We proceed to describe the repair process in case that the verification fails. An AG-rule can either conclude that $M_1||M_2$ satisfies P, or return a counterexample, which is a computation t of $M_1||M_2$ that violates P. Instead of returning t, we repair M_2 in a way that eliminates it. Our repair is either syntactic or semantic. For semantic repair we use *abduction* [45] to infer a new constraint, which makes the counterexample t infeasible.

Consider again M_1 and P of Fig. 2 and M_2 of Fig. 1. The composition $M_1||M_2$ does not satisfy P. For example, if the initial value of x_{pw} is 2^{63}, then after encryption the value of y_{pw} is 2^{64}, violating P. Our algorithm finds a bad trace t during the AG stage, which captures this bad behavior. In the repair stage, the abduction finds a constraint $x_{pw} < 2^{63}$ that eliminates t, and adds it to M_2.

Following this step we now have an updated M_2, and we apply the AG-rule again, using information we have gathered in the previous steps. In addition to removing the error trace, we update the alphabet of M_2 with the new constraint. Continuing our example, in a following iteration AGR will verify that the repaired M_2 together with M_1 satisfy P, and terminate.

In case that the current system does satisfy P, we return the repaired M_2 together with an assumption A that abstracts M_2 and acts as a smaller proof for correctness.

We have implemented a tool for AGR and evaluated it on examples of various sizes and of various types of errors. Our experiments show that for most examples, AGR converges and finds a repair after 2-5 iterations of verify-repair. Moreover, our tool generates assumptions that are significantly smaller than the (possibly repaired) M_2, thus constructing a compact and efficient proof of correctness.

1.3 Related Work

There is a wide range of techniques for automated program repair using formal methods [41,38,6,27,53,28,14,42]. Both [16] and [47] also use fault localization followed by applying mutations for repair. But, unlike this work, fault localization is applied only to the original program. The tool MUT-APR [5] fixes binary operator faults in C programs, but only targets faults that require one line modification. The tools FORENSIC [7] and MAPLE [43] repair C programs with respect to a formal specification, but they do so by replacing expressions with templates, which are then patched and analysed. SEMGRAFT [37] conducts repair with respect to a reference implementation, but relies on tests for fault localization of the original program.

Assume-guarantee style compositional verification [39,46] has been extensively studied, using learning-based approaches [13,25,23,8,9,26,10,19,20,36,40]. All these works are limited to finite state systems, and do not repair the system but only address the verification problem. [33] addresses $\mathbf{L}^*$-based compositional verification and synthesis, but it only targets finite-state systems.

The work of [32,2] use logical abduction for synthesis and repair, however, their setting is sequential, while here we target concurrent systems. [51] computes the *interface* of an infinite-state component, but it only analyzes one component at a time. In contrast, we use both components of a system to compute the necessary assumptions.

2 Mutation-Based Repair with Iterative Fault Localization

2.1 Setting

Programs and Program Correctness For our purposes, a *program* is a sequential program composed of standard statements: assignments, conditionals, loops and function calls with their standard semantics. Each statement is located at a certain *location* (or *line*) l_i, and all statements are defined over the set of program variables X. The desired behavior of the program is expressed through `assume` and `assert` statements. An `assume` statement is used to restrict executions to only those of interest (if an `assume` is violated, execution ends without an error), and an `assert` statement is used to express a desired property (if an `assert` is violated, execution ends in an error).

A program P has a *bug on input* I if an assertion violation occurs during the execution of P on I. If no assertion violation occurs during the execution of P on I, then the program is *correct for* I. If P has a bug on some input I then P is said to be *erroneous*, otherwise it is *correct*.

In this work, we focus on bounded executions of the program. A wb-bounded execution of a program P, for some integer bound wb, is an execution of P where the body of each loop is entered at most wb times and the maximum depth of the call stack is wb. A program P where no assertion violation occurs during any of its wb-bounded executions is said to be wb*-violation-free*. Our algorithm repairs programs with respect to a fixed, user-supplied, bound wb. Therefore, we refer to a wb-violation-free program as a *repaired program*, for short.

The Mutation Repair Scheme We use the notion of a *repair scheme* to define which changes to a program are allowed by a repair method. A repair scheme $\mathcal{S}$ is a function mapping a statement to a set of statements. Intuitively, the image of a statement in this function represents all options to replace it allowed by the repair method.

The repair scheme used in our algorithm is the *mutation scheme*. The mutation scheme, $\mathcal{S}_{mut}$, is a repair scheme constructed using a finite list of mutation operations, $M_1, \cdots, M_k$. A mutation operator M_i can be any partial function mapping a program expression to another program expression of the same type. Applying a mutation operator M_i to a statement st, denoted $M_i(st)$, means applying M_i to the expression of st[5]. This application is only defined if the expression of st is in the domain of M_i, in which case we say that M_i is *applicable* to st. For a program statement st, $\mathcal{S}_{mut}(st)$ is defined as $\{M_{i_1}(st), \cdots, M_{i_n}(st)\}$, where $M_{i_1}, \cdots, M_{i_n}$ are all the mutation operators from $M_1, \cdots, M_k$ applicable to st.

---

[5] If st is an assignment of the form `x:=e` then its expression is e. If st is a conditional statement, then its expression is the condition.

Example 1. Suppose we have M_1 which replaces a + operator with a − operator, M_2 which replaces a < operator with a > operator, and M_3 which allows increasing a numerical constant by 1. Let st be the statement `x:=y+1`. The expression of st is thus `y+1`, and the mutation operators applicable to st are M_1 and M_3. Therefore,

$$\mathcal{S}_{mut}(st) = \{\texttt{x:=y-1}, \texttt{x:=y+2}\}$$

Let $\mathcal{S}$ be a repair scheme. An *$\mathcal{S}$-patch* of a program P is thus a set of pairs, $\{(l_1, st_1^r), \cdots, (l_k, st_k^r)\}$, where l_i is a program location and st_i^r is a statement, for which the following holds: for all $1 \leq i \leq k$, let st_i be the statement in location l_i in P, then $st_i^r \in \mathcal{S}(st_i)$. In addition, for every $i \neq j$, $l_i \neq l_j$. Applying an *$\mathcal{S}$-patch* τ to a program P means replacing the statement st_i with st_i^r in every location l_i in τ. The *size* of the patch is the number of mutated statements (k).

We refer to the result of applying an *$\mathcal{S}_{mut}$-patch* to a program as a *mutated program*. The set of all mutated programs created from a program P is the *search space* of P.

2.2 The FL-AllRepair Algorithm

In this section we present algorithm FL-AllRepair, which gets a program P and returns all minimal repairs from within the search space of P, where minimality is defined with respect to inclusion between patches. A high level description of the algorithm is presented in Figure 3. The algorithm follows a generate-validate loop: the *generate* stage chooses a mutated program P' from the search space and the *validate* stage checks whether P' is correct. In both cases, a blocking stage occurs that removes irrelevant programs from the search space. If the program was correct, blocking is based on non-minimality. Otherwise, it is based on the results of a fault localization component.

The following subsections dive into the details of the individual components.

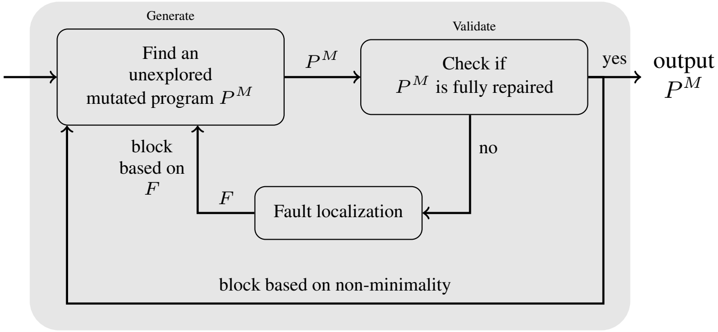

Fig. 3: Outline of algorithm FL-AllRepair for iterative mutation-based program repair.

2.3 Generate

To choose a mutated program from the search space we need to choose which mutation operator to apply to which line. We encode this choosing process in a propositional formula. Specifically, for every mutation operator M and line l, there is a boolean variable

$B_M(l)$ in the formula, which is true if and only if M is applied to line l. Additionally, for every line l there is a variable $B_O(l)$, which is true if and only if line l is not mutated. Then, the formula is constructed by requiring that for every line l exactly one of the variables $B_O(l), B_{M_1}(l), \cdots, B_{M_k}(l)$ be true. This way, there is a 1-1 correspondence between models of the formula and mutated programs in the search space. Hence, the generate stage can be realized using a SAT solver that solves this formula.

The advantage of this SAT encoding is that it allows an easy removal (blocking) of mutated programs from the search space. Such a removal can be realized by simply adding a blocking clause to the propositional formula. For example, to prevent all programs where mutation operator M is applied to line l from being considered, one can simply add the clause $\neg B_M(l)$.

Another advantage of this encoding is that it let's us easily control the size of patches being explored: we can limit the number of variables allowed to be set to true to at most s, for the desired size s. We then explore patches in increasing size by repeatedly increasing s as soon as the formula becomes unsatisfiable.

2.4 Validate

The validation of a mutated program P^M is based on a translation of the program P into a set of constraints, whose conjunction constitutes the *program formula*. In addition to representing assignments and conditionals, the program formula includes constraints representing assumptions, and a constraint representing the negated conjunction of all assertions. Thus, a satisfying assignment (a *model*) of the program formula represents an execution of P that satisfies all assumptions but violates at least one assertion.

From Programs to Program Formulas Next, we explain how the program is translated into a set of constraints. The translation, following [11], goes through four stages. Figure 4 demonstrates certain steps of the translation. First, the program is simplified and each of the branch conditions is replaced with a fresh boolean variable. In the example, g replaces the condition $w > 3$. Second, the body of each loop and each function is inlined wb times. Next, the program is converted to static single assignment (SSA) form. In particular, variables are indexed so that each indexed variable is assigned at most once. Finally, the program in SSA form is translated to a set of constraints, whose conjunction forms the formula φ_P^{wb}.

For a more detailed description see [50].

Theorem 1 ([12]) *A program P is repaired iff the formula φ_P^{wb} is unsatisfiable.*

Validation via SMT Solving Based on Theorem 1, we realize the validate stage using an SMT solver that solves the program formula $\varphi_{P^M}^{wb}$ of the mutated program P^M in question. If $\varphi_{P^M}^{wb}$ is determined unsatisfiable, P^M is added to the list of possible repairs returned to the user.

Incrementality To facilitate the repeated verification of mutated programs during different iterations, we make use of incrementality. A naive, non-incremental, approach would require translating each mutated program into a formula and solving it from scratch during each iteration. Instead, we translate only the original program into a formula as a preliminary step, before the generate-validate loop begins. Then, during

```
   proc. foo(x, w)
1: t := 0
2: y := x - 3
3: z := x + 3
4: if (w > 3) then
5:    t := z + w
6:    assert (t < x)
7:    y := y + 10
8: assert (y > z)
```

```
proc. simFoo(x, w)
t := 0
y := x - 3
z := x + 3
g := w > 3
if (g) then
   t := z + w
   assert (t < x)
   y := y + 10
assert (y > z)
```

```
proc. SSAFoo(x, w)
t0 := 0
y0 := x0 - 3
z0 := x0 + 3
g0 := w0 > 3
t1 := z0 + w0
assert (g0 → t1 < x0)
y1 := y0 + 10
t2 := g0 ? t1 : t0
y2 := g0 ? y1 : y0
assert (y2 > z0)
```

$$\begin{aligned}
&\varphi_{foo} = \{\\
&t_0 = 0,\\
&y_0 = x_0 - 3,\\
&z_0 = x_0 + 3,\\
&g_0 = w_0 > 3,\\
&t_1 = z_0 + w_0,\\
&\\
&y_1 = y_0 + 10,\\
&t_2 = ite(g_0, t_1, t_0),\\
&y_2 = ite(g_0, y_1, y_0),\\
&\neg(y_2 > z_0)\vee\\
&\neg(g_0 \rightarrow t_1 < x_0)\}
\end{aligned}$$

Fig. 4: Example of the translation process of a simple program

each iteration we make the necessary changes to the formula and use incremental SMT solving, which reuses relevant partial results from the previous iteration.

The key observation that makes this process efficient is that replacing one mutated program with another requires making only small changes to the program formula. Consider, for example, the foo program of figure 4. Replacing `y := x - 3` with `y := x * 3` on line number 2 would only require replacing the constraint $y_0 = x_0 - 3$ with the constraint $y_0 = x_0 * 3$ in the program formula. Similarly, any changes made to the right-hand-side of an assignment or to the expression in a condition only require replacing a single constraint in the formula. Therefore, this is a promising application for incremental SMT solving.

2.5 Blocking Based on Non-minimality

As mentioned, FL-AllRepair aims at returning all minimal repairs. Next, we formally define minimality: Let $P^M, P^{M'}$ be two mutated programs constructed using patches pat, pat', resp. We say that $P^M \subseteq P^{M'}$ if $pat \subseteq pat'$. This intuitively means that all lines that are mutated in P^M are mutated in $P^{M'}$, using the same mutation operators, but $P^{M'}$ contains additional mutated lines.

Definition 1 (minimal repair) *A mutated program P^M is said to be a* minimal repair *if it is a repair, and there is no other mutated program $P^{M'}$ s.t. $P^{M'} \subseteq P^M$ and $P^{M'}$ is a repair.*

The rationale for considering only minimal repairs is the observation that the program should remain as syntactically close to the original program as possible (we should avoid making changes to the code if they are not necessary for repair).

Constructing a Blocking Clause Once a program P^M is found to be correct during the validate stage, we block every mutated program $P^{M'}$ s.t. $P^M \subseteq P^{M'}$. This, together with the fact that programs are explored in increasing patch size, guarantees that only minimal repairs are returned. The blocking is realized as follows: let

$pat = \{(l_1, M_{i_1}), \cdots, (l_n, M_{i_n})\}$ be the patch used in creating P^M. Then, the following clause is added to the propositional formula of the generate stage:

$$\neg B_{M_{i_1}}(l_1) \vee \cdots \vee \neg B_{M_{i_n}}(l_n).$$

This clause restricts the search space to those mutated programs where there exists an index i for which line l_i is not mutated using mutation operator M_{l_i}. This will prune from the search space all mutated programs created using a patch pat' s.t. $pat \subseteq pat'$.

2.6 Blocking Based on Fault Localization

When a program P^M is determined buggy by the validate stage, it is passed on to a fault localization component in order to find the root cause of the bug and block other unexplored programs that exhibit the same bug. Specifically, we want fault localization to return a set of locations F whose content alone ensures the recurrence of the bug. This way, all programs in which the content of F remains the same (as in P^M) can be safely blocked. To formalize the above intuition we use the notions of a *must-location-set* and *must-fault-localization*[6].

Let P be a program with a bug on input I. A *repair for I* is a mutated program that is (bounded) correct for I. A *repairable location set (RLS) for I* is a set of locations F such that there exists a repair for I defined over F. An RLS for I is *minimal* if removing any location from it makes it no longer an RLS for I. A location is *relevant to I* if it is a part of a minimal RLS for I.

The aim of fault localization is to focus the programmer's attention on locations that are relevant for the bug. But, returning the exact set of locations relevant to I as defined above can be computationally hard. In practice, what many fault localization algorithms return is a set of locations that *may* be relevant: The returned locations have a higher chance of being relevant to I than those that are not, but there is no guarantee that all returned locations are relevant to I, nor that all locations that are relevant to I, are returned. We call such an algorithm *may fault localization*. In contrast, we define *must fault localization*, as follows:

Definition 2 (must location set) *A* must location set *for I is a set of locations that contains at least one location from each minimal RLS for I.*[7]

Definition 3 (must fault localization) *A* must fault localization *algorithm is an algorithm that for every program P and every buggy input I, returns a must location set.*

Note that a must location set is not required to contain all locations relevant to I, but only one location from each minimal RLS for I. This notion is still powerful, since it guarantees that no repair is possible without including at least one such element.

Also note, that the set of all locations visited by P during its execution on I is always a must location set. This is because any patch where none of these locations is included is definitely **not** a repair for I, since the same assertion will be violated along the same path. However, this set of locations may not be minimal.

---

[6] For brevity, the definitions brought here are an instantiation of the original definitions from [50] to the mutation scheme. Originally, the definitions of both a must-location-set and must-fault-localization depend on the repair scheme.

[7] This is, in fact, a hitting set of the set of all minimal RLS for I.

Fault Localization Algorithm Going back to FL-AllRepair, Let P^M be a program found to be buggy by the validate stage, and let μ be the model of the program formula φ_{PM} obtained by the SMT solver. P^M is then passed to a fault localization component, which receives the formula φ_{PM} and the model μ and returns a set of locations F. This component is realized using a formula slicing-based algorithm that agrees with the definition of a must-fault-localization algorithm. For brevity, we omit the details of this algorithm and refer the reader to section 5 in [50].

Constructing a Blocking Clause Let F be the set of locations returned by the fault localization component. Since the fault localization algorithm is a must-fault-localization algorithm, F is a must-location-set. This means that all mutated programs which are identical to P^M on the locations in F can be safely removed from the search space. We remove them by adding a blocking clause to the propositional formula, encoding that at least one location from F should be changed. For example, suppose that F consists of $\{l_1, l_2, l_3\}$, where l_1 was mutated with M_1, where l_2 was not mutated, and l_3 was mutated with M_3. The constructed blocking clause will then be $\neg B_{M_1}(l_1) \vee \neg B_O(l_2) \vee \neg B_{M_3}(l_3)$. The blocking clause restricts the search space to those mutated programs that either do not apply mutation M_1 to l_1, or do mutate l_2, or do not apply M_3 to l_3.

2.7 Experimental Results

We have implemented the FL-AllRepair algorithm on top of the AllRepair open source tool. This tool previously implemented an earlier version of the algorithm, presented in [49], which avoids the use of fault localization and instead blocks only the one incorrect mutated program found during that iteration.

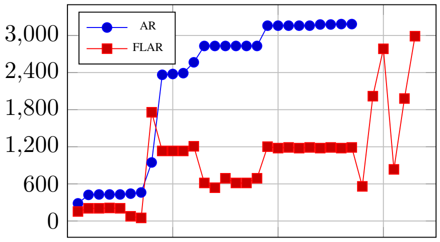

Fig. 5: Time to find a repair using FL-AllRepair and AllRepair (in seconds). Each point along the x axis represents a repair for a single input and the y axis value represents the time to find that repair.

This early version of the algorithm was recently compared against 4 other repair tools in [43] and was found to be very efficient. The tools participating in the experiment were: Angelix [38], GenProg [31], FoRenSiC [7] and Maple [43]. All tools were run on the TCAS benchmark [18], but with different specifications: Angelix and GenProg use a test suite while the rest of the tools use a formal specification. The results showed a significant advantage to the AllRepair tool in terms of efficiency: AllRepair was found to be faster by an order of magnitude than all of the compared tools, taking only 16.9 seconds to find a repair on average, where the other tools take 1540.7, 325.4, 360.1, and 155.3 seconds, respectively. On the other hand, AllRepair's repair ability is limited, due to the use of the mutation scheme: it is only able to repair 18 versions (out of 41), while Angelix, GenProg, FoRenSiC and Maple repair 32, 11, 23, and 26, respectively.

In [50] we have conducted an experiment to check the impact of adding fault-localization-based blocking on efficiency (repair ability is not affected, since must-fault-localization guarantees that we will not lose any of the potential good repairs). We ran FL-AllRepair on the TCAS benchmark as well as a small subset of the Codeflaws benchmark [52]. The Codeflaws benchmark is a collection of programs taken from buggy user submissions to the programming contest site Codeforces[8]. We compared the time to find a repair in FL-AllRepair and All-Repair, using various unwinding bounds and mutation sets. Overall, the experiment consisted of 186 *inputs*, where an input is a combination of buggy program, mutation set, and unwinding bound.

Our conclusion was that FL-AllRepair is able to acheive significant speed-ups, especially for cases of interest where AllRepair struggles to find a repair in the first place. To demonstrate, figure 5 shows the time it took to find a repair using both algorithms, for all the repairs where AllRepair took more than 5 min. Observe that FL-AllRepair saves time for all but one of these repairs, and the savings go up to dozens of minutes.

For a more detailed description of our experiments, see [50,48]

3 Verification and Repair of Communicating systems (AGR)

We now describe our second approach to automatic repair, based on compositional verification.

3.1 Communicating Programs

We first present our programs, which are modeled as *communicating systems*.

The alphabet α of a communicating system uses a set of variables X (whose ordered vector is $\bar{x}$), ranging over a (possibly infinite) data domain $\mathbb{D}$. The alphabet α consists of a set $\mathcal{C}$ of *constraints*, which are quantifier-free first-order formulas over $X \cup \mathbb{D}$, representing the conditions in *if* and *while* statements. It also includes a set of *assignment statements* $\mathcal{E}$, consisting of statements of the type $x := e$, where e is an expression over $X \cup \mathbb{D}$. Finally, α includes a set $\mathcal{G}$ of *communication actions*, over a set G of communication channels. The action $g?x$ is a *read* action of a value to the variable x through channel g, and $g!x$ is a *write* action of the value of x on g. We use $g * x$ to indicate some action, either *read* or *write*, through g. The pairs $(g?x_1, g!x_2)$ and $(g!x_1, g?x_2)$ then represent a synchronization of two programs on read-write actions over g.

Definition 1. *A* communicating program *(or, a program) is* $M = \langle Q, X, \alpha, \delta, q_0, F\rangle$, *where:*

1. *Q is a finite set of states and $q_0 \in Q$ is the initial state.*
2. *X is a finite set of variables over $\mathbb{D}$.*
3. *$\alpha = \mathcal{G} \cup \mathcal{E} \cup \mathcal{C}$ is the action alphabet of M.*
4. *$\delta \subseteq Q \times \alpha \times Q$ is the transition relation.*
5. *$F \subseteq Q$ is the set of accepting states.*

The words that are read along a communicating program are a symbolic representation of the program behaviors. We refer to such a word as a *trace*. Each such trace induces *executions* of the program, which are formed by concrete assignments to the program variables in a way that conforms to the actions along the word. We think of the program automaton as the generator of the behaviors of the program – a word in the language of the automaton is a program run, which induces a set of executions.

---

[8] http://codeforces.com/

More formally, a *run* r in a program automaton M is a finite sequence of states and actions starting with the initial state and following δ. The *induced trace* t of r is the sequence of the actions in r. If r reaches an accepting state, then t is an *accepted trace* of M. An *execution* p of t is a sequence of valuations of X that respects the semantics of the alphabet. That is, the valuation of a variable x can only change by a read action through a communication channel, e.g. $g?x$, or through an assignment $x := e$. In addition, the valuations must satisfy the constraints along t. That is, if $\beta(\bar{x})$ is a valuation in p at location i, and t_i is a constraint at i, then $\beta(\bar{x}) \models t_i$. We say that t is *feasible* if there exists an execution of t.

Example 2. The trace $(x := 2{\cdot}y,\ g?x,\ y := y{+}1,\ g!y)$ is feasible, as it has an execution $(x = 1, y = 3), (x = 6, y = 3), (x = 20, y = 3), (x = 20, y = 4), (x = 20, y = 4)$. The trace $(g?x,\ x := x^2, x < 0)$ is not feasible since no valuation can satisfy the constraint $x < 0$ if $x := x^2$ is executed beforehand.

The *symbolic language* of M, denoted $\mathcal{T}(M)$, is the set of all *accepted* traces induced by runs of M. The *concrete language* of M is the set of all executions of accepted traces in $\mathcal{T}(M)$.

Parallel Composition We now describe the parallel composition of two communicating programs, and the way in which they communicate.

In *parallel composition*, the two components synchronize on their communication interface only when one component writes data through a channel, and the other reads it through the same channel. The two components cannot synchronize if both are trying to read or both are trying to write. We distinguish between communication of the two components with each other (on their common channels), and their communication with their environment. In the former case, the components must “wait” for each other in order to progress together. In the latter case, the communication actions of the two components interleave asynchronously.

Let M_1 and M_2 be two programs, where $M_i = \langle Q_i, X_i, \alpha_i, \delta_i, {q_0}^i, F_i \rangle$ for $i = 1, 2$. Let G_1, G_2 be the sets of communication channels occurring in actions of M_1, M_2, respectively. We assume that $X_1 \cap X_2 = \emptyset$. The *interface alphabet* αI of M_1 and M_2 consists of all communication actions on channels that are common to both components. That is, $\alpha I = \{\, g?x,\ g!x \ :\ g \in G_1 \cap G_2,\ x \in X_1 \cup X_2 \}$.
Formally, the parallel composition of M_1 and M_2, denoted $M_1 || M_2$, is the program $M = \langle Q, X, \alpha, \delta, q_0, F \rangle$, defined as follows.

1. $Q = (Q_1 \times Q_2) \cup (Q_1' \times Q_2')$, where Q_1' and Q_2' are new copies of Q_1 and Q_2, respectively. The initial state is $q_0 = (q_0^1, q_0^2)$; $X = X_1 \cup X_2$; $F = F_1 \times F_2$.
2. $\alpha = \{\, (g?x_1, g!x_2), (g!x_1, g?x_2) \ :\ g * x_1 \in (\alpha_1 \cap \alpha I)$ and $g * x_2 \in (\alpha_2 \cap \alpha I) \} \cup ((\alpha_1 \cup \alpha_2) \setminus \alpha I)$. That is, the alphabet includes pairs of read-write communication actions on channels that are common to M_1 and M_2. It also includes individual actions of M_1 and M_2, which are not communications on common channels.
3. δ is defined as follows.
   (a) For $(g * x_1, g * x_2) \in \alpha$[9]:
      i. $\delta((q_1, q_2), (g * x_1, g * x_2)) = (q_1', q_2')$.

---

[9] According to item 2, one of the actions must be a read and the other must be a write action.

ii. $\delta((q_1', q_2'), x_1 = x_2) = (\delta_1(q_1, g * x_1), \delta_2(q_2, g * x_2))$.

That is, when a communication is performed synchronously in both components, the data is transformed through the channel from the writing component to the reading component. As a result, the values of x_1 and x_2 equalize. This is enforced in M by adding a transition labeled by the constraint $x_1 = x_2$ that immediately follows the synchronous communication.

(b) For $a \in \alpha_1 \setminus \alpha I$ we define $\delta((q_1, q_2), a) = (\delta_1(q_1, a), q_2)$. Similarly, for $a \in \alpha_2 \setminus \alpha I$ we define $\delta((q_1, q_2), a) = (q_1, \delta_2(q_2, a))$. That is, on actions that are not in the interface alphabet, the two components interleave.

Figure 6 demonstrates the parallel composition of components M_1 and M_2. The program $M = M_1 || M_2$ reads a password from the environment through channel $pass$. The two components synchronize on channel $verify$. This synchronization is represented by the constraint $x = y$, which describes the result of the synchronization. Assignments to x are interleaved with reading the value of y from the environment.

3.2 Regular Properties and their Satisfaction

We now describe the syntax and semantics of the properties that we consider. These can be represented as finite automata, hence the name *regular properties*. However, the alphabet of these automata includes communication actions and first-order constraints over program variables. Thus, such automata are suitable for specifying the desired (and undesired) behaviors of communicating programs over time.

We require our properties to be deterministic and complete. Since we consider symbolic representation of systems, we require also *semantic* determinism and completeness. That is, if $\langle q, c_1, q' \rangle$ and $\langle q, c_2, q'' \rangle$ are in δ for constraints $c_1, c_2 \in \mathcal{C}$ such that $c_1 \neq c_2$ and $q' \neq q''$, then $c_1 \wedge c_2 \equiv false$; and let C_q be the set of all constraints on transitions leaving q. Then $(\bigvee_{c \in C_q} c) \equiv true$.

A *property* is a deterministic and complete program with no assignment actions, whose language defines the set of allowed behaviors over the alphabet αP.

A trace is accepted by a property P if it reaches a state in F, the set of accepting states of P. Otherwise, it reaches a state in $Q \setminus F$, and is rejected by P.

Next, we define the satisfaction relation $\vDash$ between a program and a property. Intuitively, a program M satisfies a property P (denoted $M \vDash P$) if all executions induced

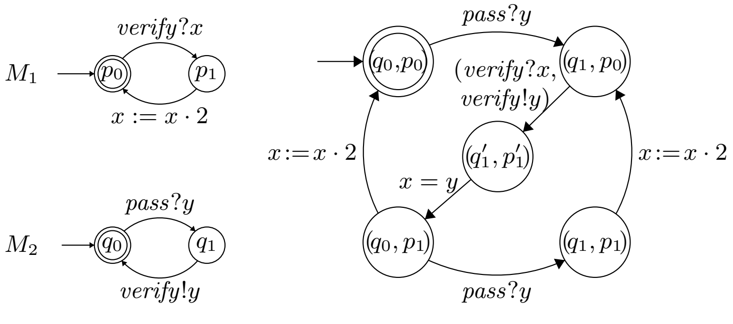

Fig. 6: Components M_1 and M_2 and their parallel composition $M_1 || M_2$.

by accepted traces of M reach an accepting state in P. Thus, the accepted behaviors of M are also accepted by P.

Conjunctive Composition In order to capture the satisfaction relation between M and P, we define a *conjunctive composition* between M and P, denoted $M \times P$. Unlike parallel composition, in conjunctive composition the two components synchronize on their common communication actions when both read or both write through the same communication channel. In addition, if M is the result of a parallel compositions, then they might synchronize on alphabet of the type $(g * x, g * y)$. They interleave on constraints and on actions of αM that are not in αP. The set of accepting states of $M \times P$ is $F = F_M \times (Q_P \setminus F_P)$. As a result, accepted traces in $M \times P$ are those that are *accepted* in M and *rejected* in P. Such traces are called *error traces* and their corresponding executions are called *error executions*. Intuitively, an error execution is an execution along M which violates the properties modeled by P. Such an execution either fails to synchronize on the communication actions, or reaches a point in the computation in which its assignments violate some constraint described by P. Since a feasible error trace in $M \times P$ is an evidence to $M \not\models P$, we define $M \models P$ iff $M \times P$ contains no feasible accepted traces.

3.3 The Assume-Guarantee Rule for Communicating Systems

Let M_1 and M_2 be two programs, and let P be a property. The classical Assume-Guarantee (AG) proof rule [46] assures that if we find an assumption A (in our case, a communicating program) such that $M_1 || A \models P$ and $M_2 \models A$ both hold, then $M_1 || M_2 \models P$ holds as well.

Previous works (e.g. [13]) rely on $\mathbf{L}^*$ for constructing an assumption A, based on the weakest assumption A_w, defined below.

Definition 2 (Weakest Assumption). *Let P be a property and M be a system. The weakest assumption A_w with respect to M and P has the language*

$$\mathcal{L}(A_w) = \{w : \ M || w \models P\}.$$

That is, $\mathcal{L}(A_w)$ is the set of all words that together with M satisfy P.

A crucial point of a learning-based AG method is that A_w is *regular* [24], and so can be learned by $\mathbf{L}^*$. However, this is not always the case in our setting, as the next example shows.

Example 3. Over the alphabet $\{x := 0, \ y := 0, \ x := x + 1, \ y := y + 1\}$ we can construct a system for which the weakest assumption requires an equal number of actions of the form $x := x + 1$ and $y := y + 1$, which is not a regular property.

To cope with this difficulty, we change the focus of learning. Instead of learning the (possibly) non-regular language of A_w, we learn $\mathcal{T}(M_2)$, the set of accepted traces of M_2. This language is guaranteed to be regular, as it is represented by the automaton M_2. As a result, our AG rule is sound and complete, as stated in the theorem below.

Theorem 1. *Our AG rule for communicating programs is sound and complete.*

3.4 The Assume-Guarantee-Repair (AGR) Framework

In this section we discuss our Assume-Guarantee-Repair (AGR) framework for communicating programs. The framework consists of a learning-based Assume-Guarantee algorithm, called $AG_{\mathbf{L}^*}$, and a REPAIR procedure, which are tightly joined.

Recall that the goal of $\mathbf{L}^*$ in our case is to learn $\mathcal{T}(M_2)$. The nature of $AG_{\mathbf{L}^*}$ is such that the assumptions it learns before it reaches M_2 may contain traces of M_2 and more, but still be represented by a smaller automaton. Therefore, similarly to [13], $AG_{\mathbf{L}^*}$ often terminates with an assumption A that is much smaller than M_2. Indeed, our tool often produces very small assumptions (see Fig. 8).

When $M_1||M_2 \nvDash P$, the $AG_{\mathbf{L}^*}$ algorithm returns an error trace t as a witness to the violation. In this case, we initiate the REPAIR procedure, which eliminates t from M_2, resulting in M_2'.

We then return to $AG_{\mathbf{L}^*}$ with a new goal, M_2', to search for a new assumption A' that allows to verify $M_1||M_2' \vDash P$. As we have mentioned, $AG_{\mathbf{L}^*}$ is incremental: when learning an assumption A' for M_2' we can use the membership answers we obtained for M_2, since these have not changed. The difference between the languages of M_2 and M_2' lies in words (traces) whose membership has not yet been queried on M_2. Learning M_2' can then start from the point where learning M_2 has left off, resulting in a more efficient algorithm.

As opposed to the case where $M_1||M_2 \vDash P$, we cannot guarantee the termination of the repair process in case that $M_1||M_2 \nvDash P$. This is because we are only guaranteed to remove one (bad) trace and add one (infeasible) trace in every iteration (although in practice, every iteration may remove a larger set of traces). Thus, we may never converge to a repaired system. Nevertheless, in case of violation, our algorithm always finds an error trace, thus a progress towards a "less erroneous" program is guaranteed.

It should be noted that the $AG_{\mathbf{L}^*}$ part of our AGR algorithm deviates from the AG-rule of [13] in two important ways. First, since our learning goal is M_2 rather than A_w, our membership queries are different in type and order. Second, in order to identify real error traces and send them to REPAIR as early as possible, we add queries to the membership phase that reveal such traces. We then send them to REPAIR without ever passing through equivalence queries, which improves the overall efficiency. Indeed, our experiments include several cases in which all repairs were invoked from the membership phase. In these cases, AGR ran an equivalence query only when it has already successfully repaired M_2, and terminated.

The Assume-Guarantee-Repair (AGR) Algorithm We now present an overview of our AGR algorithm. For a detailed description, see [21,22]. Fig. 7 describes the flow of the algorithm.

AGR comprises two main parts, namely $AG_{\mathbf{L}^*}$ and REPAIR. The input to AGR are the components M_1 and M_2, and the property P. While M_1 and P stay unchanged during AGR, M_2 is repeatedly updated as long as it needs repair. In every iteration of AGR an updated M_2^i is calculated. Initially, $M_2^0 = M_2$. An iteration starts with the membership phase, and ends either when $AG_{\mathbf{L}^*}$ successfully terminates, or when REPAIR is called. When constructing M_2^i (based, as noted, on the construction of M_2^{i-1}), the new iteration is given new trace(s) that have been added or removed from M_2^{i-1}.

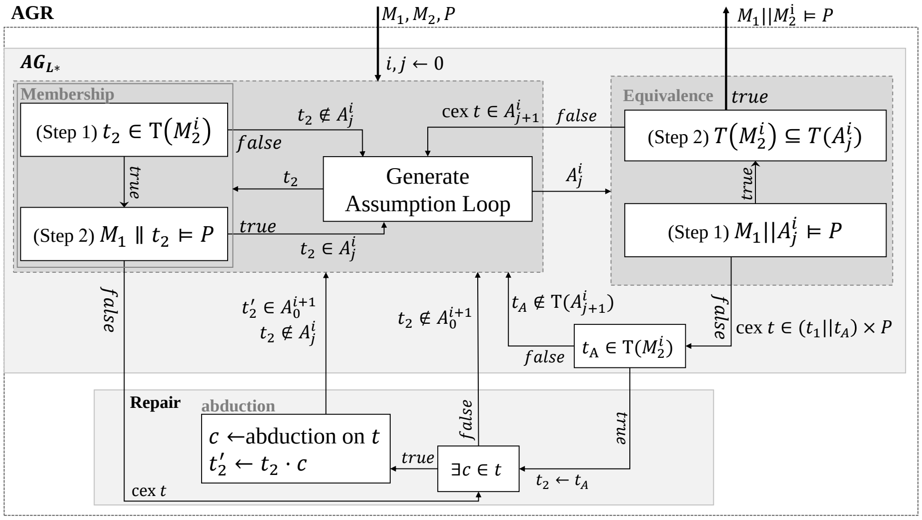

Fig. 7: The flow of AGR

$AG_{\mathbf{L}^*}$ consists of two phases: membership, and equivalence. The membership phase is a loop in which the learner constructs the next assumption A_j^i according to answers it gets from the teacher on a sequence of membership queries on various traces. These queries are answered in accordance with traces we allow in A_j^i. These are the traces in M_2^i that in parallel with M_1 satisfy P. If a trace t in M_2^i in parallel with M_1 does not satisfy P, then t is a bad behavior of M_2^i. Therefore, if such t is found during the membership phase, REPAIR is invoked.

Once the learner reaches a stable assumption A_j^i, it passes it to the equivalence phase. A_j^i is a suitable assumption if both $M_1||A_j^i \models P$ and $\mathcal{T}(M_2^i) \subseteq \mathcal{T}(A_j^i)$ hold. AGR then terminates and returns M_2^i as a successful repair of M_2. In case $M_1||A_j^i \not\models P$, a counterexample t is returned, that is composed of bad traces in M_1, A_j^i, and P. If the bad trace t_2, the restriction of t to the alphabet of A_j^i, is also in M_2^i, then t_2 is a bad behavior of M_2^i, and REPAIR is invoked. Otherwise, AGR returns to the membership phase with t_2 as a trace that should not be in A_j^i, and continues to learn A_j^i.

Next we describe in more detail how repair is applied. We distinguish between semantic and syntactic repairs, which are solved differently.

Semantic Repair by Abduction In case the error trace t contains constraints, we *semantically* repair M_2^i by inferring a new constraint that makes t infeasible. The new constraint is then added to the alphabet of M_2^i and may eliminate additional error traces.

The process of inferring new constraints from known facts about the program is called *abduction* [17]. Given a trace t, let φ_t be the first-order formula (a conjunction of constraints), which constitutes the SSA representation of t [3]. In order to make t infeasible, we look for a formula ψ such that $\psi \wedge \varphi_t \rightarrow \mathit{false}$.[10]

---

[10] Usually, in abduction, we look for ψ such that $\psi \wedge \varphi_t$ is not a contradiction. However, since φ_t is a violation of the specification, we want to infer a formula that makes φ_t unsatisfiable.

Note that $t \in \mathcal{T}(M_1||M_2^i) \times P$, and so it includes variables both from X_1, the variables of M_1, and from X_2, the variables of M_2^i. Since we wish to repair M_2^i, the learned ψ is only over X_2. The formula $\psi \wedge \varphi_t \to \mathit{false}$ is equivalent to $\psi \to (\varphi_t \to \mathit{false})$. Then, $\psi = \forall x \in X_1 : (\varphi_t \to \mathit{false}) = \forall x \in X_1(\neg\varphi_t)$, is such a desired constraint: ψ makes t infeasible and is only over X_2. We now use quantifier elimination [54] to produce a quantifier-free formula over X_2. Computing ψ is similar to the abduction suggested in [17], but the focus here is on finding a formula over X_2 rather than over any minimal set of variables as in [17]; further, [17] looks for ψ such that $\varphi_t \wedge \psi$ is not a contradiction, while we specifically look for ψ that blocks φ_t. We use Z3 [15] to apply quantifier elimination and to generate the new constraint. After generating $\psi(X_2)$, we add it to the alphabet of M_2^i. We also produce a new trace $t_2' = t_2 \cdot \psi(X_2)$, which is returned as the output of the abduction. AGR now returns to $AG_{\mathbf{L}^*}$ in order to learn an assumption for the repaired component M_2^{i+1}, which now includes t_2' but not t_2.

Syntactic Removal of Error Traces In case that the error trace t does not contain constrains, we can remove t from M_2 by constructing a system whose language is $\mathcal{T}(M_2) \setminus \{t\}$. We call this the *exact* method for repair. However, removing a single trace at a time may lead to slow convergence, and exponentially blows-up the repaired systems. Moreover, in some cases there are infinitely many such traces, in which case AGR may never terminate.

For faster convergence, we have implemented two additional heuristics, namely *approximate* and *aggressive*. These heuristics may remove more than a single trace at a time, while keeping the size of the systems small. While "good" traces may be removed as well, the correctness of the repair is maintained, since no bad traces are added. Moreover, an error trace is likely to be in an erroneous part of the system, and in these cases our heuristics manage removing a set of error traces in a single step.

All three methods modify the structure of the underlying automaton. In the *approximate* method we add an intermediate state on the way to an accepting state, to which the error trace, and potentially more erroneous behaviours, are diverted. The *aggressive* method simply makes the state that M_2 reaches upon reading t, non-accepting. In case that every accepting state is reached by some error trace, this might result in an empty language, creating a trivial repair. However, our experiments show that in most cases, this method quickly leads to a non-trivial repair.

For further details of syntactic and semantic repair, see [21,22].

3.5 Experimental Results

We implemented our AGR framework in Java, integrating $\mathbf{L}^*$ implementation from the LTSA tool [35]. We used Z3 [15] as the teacher for the satisfaction queries in $AG_{\mathbf{L}^*}$, and for abduction in REPAIR. Fig. 8 demonstrates the effectiveness of our approach on several examples (the x-axis indicates their indices). The examples are based on simple examples from [24] adapted to our setting. Note that the assumption sizes are mostly shown to be much smaller than the original components. The syntactic repair method presented in Fig. 8 is the approximate repair, however the same holds also for the other repair methods. Additional results are available in [21], and the full examples are available on [1].

3.6 Correctness and Termination

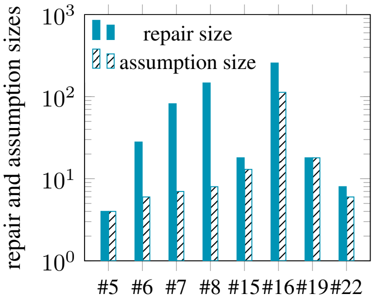

Fig. 8: Repair vs. assumption size (log. scale).

We assume a sound and complete teacher who can answer the membership and equivalence queries in $AG_{\mathbf{L}^*}$. We use Z3 [15] in order to answer satisfiability queries issued in the learning process. Our examples were over the theory of linear arithmetic, for which Z3 is indeed sound and complete.

As noted, AGR may not terminate, and there are cases in which REPAIR is called infinitely many times. However, in case that no repair is needed, or if a repaired system is eventually obtained, then AGR is guaranteed to terminate correctly.

To see why, consider a repaired system M_2^i for which $M_1||M_2^i \models P$. Since the goal of $AG_{\mathbf{L}^*}$ is to syntactically learn M_2^i, which is regular, this stage will terminate at the latest when $AG_{\mathbf{L}^*}$ learns exactly M_2^i (it may terminate sooner if a smaller appropriate assumption is found). Notice that, in particular, if $M_1||M_2 \models P$, then AGR terminates with a correct answer in the first iteration of the verify-repair loop.

REPAIR is only invoked when a (real) error trace t is found in M_2^i, in which case a new system M_2^{i+1}, that does not include t, is produced by REPAIR. If $M_1||M_2^i \not\models P$, then an error trace is guaranteed to be found by $AG_{\mathbf{L}^*}$ either in the membership or equivalence phase. Therefore, also in case that M_2^i violates P, the iteration is guaranteed to terminate. In particular, since every iteration of AGR finds and removes an error trace t, and no new erroneous traces are introduced in the updated system, then in case that M_2 has finitely many error traces, AGR is guaranteed to terminate with a repaired system, which is correct with respect to P.

4 Conclusions and Discussion

We presented two approaches for automated program repair, using formal methods techniques. Both approaches aim to verify infinite state C-like programs and handle both the syntax and the semantics of the program. Both approaches are incremental in the sense of reusing information from previous iterations in order to verify the current program.

Despite the common grounds, each approach handles the verification and repair differently. The mutation-based approach handles sequential imperative programs with assertions in the code. It relies on a reduction to the problem of finding unsatisfiable sets of constraints and uses SAT and SMT solvers to realize a generate-and-validate loop. Efficiency is achieved through incremental solving and efficient pruning.

The AGR approach is based on automata learning and offers a verify-repair algorithm that takes the advantages of the automata representation in order to apply automata learning. It modifies the components both syntactically by eliminating error traces, and semantically by adding constraints using abduction.

We have implemented both algorithms and our experimental results demonstrate the effectiveness of our approaches for program repair.

Acknowledgement: This work was partially supported by the Israel Science Foundation (ISF), Grant No. 979/11

References

1. https://github.com/hadarlh/AGR.
2. A. Albarghouthi, I. Dillig, and A. Gurfinkel. Maximal specification synthesis. In *POPL*, 2016.
3. B. Alpern, M. N. Wegman, and F. K. Zadeck. Detecting equality of variables in programs. In *POPL*, 1988.
4. D. Angluin. Learning regular sets from queries and counterexamples. *Inf. Comput.*, 75(2):87–106, 1987.
5. F. Y. Assiri and J. M. Bieman. MUT-APR: MUTation-Based Automated Program Repair Research Tool. In *FICC*, 2018.
6. P. C. Attie, K. Dak, A. L. Bab, and M. Sakr. Model and Program Repair via SAT Solving. *ACM Transactions on Embedded Computing Systems*, 17(2), 2017.
7. R. Bloem, R. Drechsler, G. Fey, A. Finder, G. Hofferek, R. Könighofer, J. Raik, U. Repinski, and A. Sülflow. FoREnSiC–An automatic debugging environment for C programs. In *Hardware and Software: Verification and Testing*, pages 260–265. Springer, 2012.
8. S. Chaki and O. Strichman. Optimized l*-based assume-guarantee reasoning. In *TACAS*, 2007.
9. Y.-F. Chen, E. M. Clarke, A. Farzan, M.-H. Tsai, Y.-K. Tsay, and B.-Y. Wang. Automated assume-guarantee reasoning through implicit learning. In *CAV*, 2010.
10. Y.-F. Chen, A. Farzan, E. M. Clarke, Y.-K. Tsay, and B.-Y. Wang. Learning minimal separating DFA's for compositional verification. In *TACAS*, 2009.
11. E. Clarke, D. Kroening, and F. Lerda. A tool for checking ANSI-C programs. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 168–176. Springer, 2004.
12. E. Clarke, D. Kroening, and K. Yorav. Behavioral consistency of C and Verilog programs using bounded model checking. In *Design Automation Conference, 2003. Proceedings*, pages 368–371. IEEE, 2003.
13. J. M. Cobleigh, D. Giannakopoulou, and C. S. Pasareanu. Learning assumptions for compositional verification. In *TACAS*, 2003.
14. L. D'Antoni, R. Samanta, and R. Singh. QLOSE: Program repair with quantitative objectives. In *Computer Aided Verification*, 2016.
15. L. De Moura and N. Bjørner. Z3: An efficient smt solver. In *TACAS*, 2008.
16. V. Debroy and W. E. Wong. Using mutation to automatically suggest fixes for faulty programs. In *Software Testing, Verification and Validation (ICST), 2010 Third International Conference on*, pages 65–74. IEEE, 2010.
17. I. Dillig and T. Dillig. Explain: A tool for performing abductive inference. In *CAV*, 2013.
18. H. Do, S. Elbaum, and G. Rothermel. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *Empirical Software Engineering*, 10(4):405–435, 2005.
19. K. A. Elkader, O. Grumberg, C. S. Pasareanu, and S. Shoham. Automated circular assume-guarantee reasoning. In *FM*, 2015.
20. K. A. Elkader, O. Grumberg, C. S. Pasareanu, and S. Shoham. Automated circular assume-guarantee reasoning with n-way decomposition and alphabet refinement. In *CAV*, 2016.
21. H. Frenkel. *Automata over Infinite Data Domains: Learnability and Applications in Program Verification and Repair*. PhD thesis, 2021.
22. H. Frenkel, O. Grumberg, C. S. Pasareanu, and S. Sheinvald. Assume, guarantee or repair. In A. Biere and D. Parker, editors, *TACAS 2020*, volume 12078 of *Lecture Notes in Computer Science*, pages 211–227. Springer, 2020.

23. M. Gheorghiu, D. Giannakopoulou, and C. S. Pasareanu. Refining interface alphabets for compositional verification. In *TACAS*, 2007.
24. D. Giannakopoulou, C. S. Pasareanu, and H. Barringer. Assumption generation for software component verification. In *ASE*. IEEE Computer Society, 2002.
25. D. Giannakopoulou, C. S. Pasareanu, and H. Barringer. Component verification with automatically generated assumptions. *Autom. Softw. Eng.*, 12(3):297–320, 2005.
26. A. Gupta, K. L. McMillan, and Z. Fu. Automated assumption generation for compositional verification. *Formal Methods in System Design*, 32(3):285–301, 2008.
27. B. Jobstmann, A. Griesmayer, and R. Bloem. Program repair as a game. In *Computer Aided Verification*, pages 226–238. Springer, 2005.
28. E. Kneuss, M. Koukoutos, and V. Kuncak. Deductive program repair. In *Computer Aided Verification*, pages 217–233. Springer, 2015.
29. R. Könighofer and R. Bloem. Automated error localization and correction for imperative programs. In *Formal Methods in Computer-Aided Design (FMCAD), 2011*, pages 91–100. IEEE, 2011.
30. R. Könighofer and R. Bloem. Repair with on-the-fly program analysis. In *Hardware and Software: Verification and Testing*, pages 56–71. Springer, 2013.
31. C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer. GenProg: A generic method for automatic software repair. *Software Engineering, IEEE Transactions on*, 38(1):54–72, 2012.
32. B. Li, I. Dillig, T. Dillig, K. L. McMillan, and M. Sagiv. Synthesis of circular compositional program proofs via abduction. In *TACAS*, 2013.
33. S. Lin and P. Hsiung. Compositional synthesis of concurrent systems through causal model checking and learning. In *FM*, 2014.
34. K. Liu, A. Koyuncu, T. F. Bissyande, D. Kim, J. Klein, and Y. Le Traon. You cannot fix what you cannot find! an investigation of fault localization bias in benchmarking automated program repair systems. *ICST*, (0):102–113, 2019.
35. J. Magee and J. Kramer. *Concurrency - state models and Java programs*. Wiley, 1999.
36. K. L. McMillan. Circular compositional reasoning about liveness. In *CHARME*, 1999.
37. S. Mechtaev, M.-D. Nguyen, Y. Noller, L. Grunske, and A. Roychoudhury. Semantic Program Repair Using a Reference Implementation. In *ICSE*, 2018.
38. S. Mechtaev, J. Yi, and A. Roychoudhury. Angelix: Scalable Multiline Program Patch Synthesis via Symbolic Analysis. In *ICSE*. ICSE, 2016.
39. J. Misra and K. M. Chandy. Proofs of networks of processes. *IEEE Trans. Software Eng.*, 7(4):417–426, 1981.
40. K. S. Namjoshi and R. J. Trefler. On the competeness of compositional reasoning. In *CAV*, 2000.
41. H. D. T. Nguyen, D. Qi, A. Roychoudhury, and S. Chandra. SemFix: Program repair via semantic analysis. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 772–781. IEEE Press, 2013.
42. T. Nguyen, W. Weimer, D. Kapur, and S. Forrest. Connecting Program Synthesis and Reachability: Automatic Program Repair Using Test-Input Generation. *TACAS*, 19(6):649–652, 2017.
43. T.-T. Nguyen, Q.-T. Ta, and W.-N. Chin. Automatic Program Repair Using Formal Verification and Expression Templates. In *VMCAI*, 2019.
44. C. S. Pasareanu, D. Giannakopoulou, M. G. Bobaru, J. M. Cobleigh, and H. Barringer. Learning to divide and conquer: applying the L* algorithm to automate assume-guarantee reasoning. *Formal Methods in System Design*, 2008.
45. C. Peirce and C. Hartshorne. *Collected Papers of Charles Sanders Peirce*. Belknap Press, 1932.
46. A. Pnueli. In transition from global to modular temporal reasoning about programs. In *Logics and Models of Concurrent Systems, NATO ASI Series*, 1985.

47. U. Repinski, H. Hantson, M. Jenihhin, J. Raik, R. Ubar, G. D. Guglielmo, G. Pravadelli, and F. Fummi. Combining dynamic slicing and mutation operators for ESL correction. In *Test Symposium (ETS), 2012 17th IEEE European*, pages 1–6. IEEE, 2012.
48. B.-C. Rothenberg. *Formal Automated Program Repair*. PhD thesis, 2020.
49. B.-C. Rothenberg and O. Grumberg. *Sound and complete mutation-based program repair*, volume 9995 LNCS. 2016.
50. B.-C. Rothenberg and O. Grumberg. Must Fault Localization For Program Repair. In *CAV*, 2020.
51. R. Singh, D. Giannakopoulou, and C. S. Pasareanu. Learning component interfaces with may and must abstractions. In *CAV*, 2010.
52. S. H. Tan, J. Yi, Yulis, S. Mechtaev, and A. Roychoudhury. Codeflaws: A programming competition benchmark for evaluating automated program repair tools. *Proceedings - 2017 IEEE/ACM 39th International Conference on Software Engineering Companion, ICSE-C 2017*, pages 180–182, 2017.
53. C. Von Essen and B. Jobstmann. Program repair without regret. *Formal Methods in System Design*, 47(1):26–50, 2015.
54. V. Weispfenning. Quantifier elimination and decision procedures for valued fields. *Models and Sets.Lecture Notes in Mathematics (LNM)*, 1103:419—472, 1984.