A Generic Algorithm for Program Repair

Besma Khaireddine
University of Tunis Manar, Tunisia
khaireddine.besma@gmail.com

Aleksandr Zakharchenko
NJIT, Newark NJ
az68@njit.edu

Ali Mili
NJIT, Newark NJ
mili@njit.edu

***Abstract*—Relative correctness is the property of a program to be more-correct than another with respect to a specification; whereas traditional (absolute) correctness distinguishes between two classes of candidate programs with respect to a specification (correct and incorrect), relative correctness defines a partial ordering between candidate programs, whose maximal elements are the (absolutely) correct programs. In this paper we argue that relative correctness ought to be an integral part of the study of program repair, as it plays for program repair the role that absolute correctness plays for program construction: in the same way that absolute correctness is the criterion by which we judge the process of deriving a program P from a specification R, we argue that relative correctness ought to be the criterion by which we judge the process of repairing a program P to produce a program P' that is more-correct than P with respect to R. In this paper we build on this premise to design a generic program repair algorithm, which proceeds by successive increases of relative correctness until we achieve absolute correctness. We further argue that in the same way that correctness ideas were used, a few decades ago, as a basis for correct-by-design programming, relative correctness ideas may be used, in time, as a basis for more-correct-by-design program repair.**

***Index Terms*—Oracle design, program repair, absolute correctness, relative correctness, elementary fault removal, Siemens benchmark.**

I. Program Repair with Relative Correctness

Relative correctness (introduced in [19]) is the property of a program to be more-correct than another with respect to a given specification. Whereas traditional (absolute) correctness distinguishes between two classes of candidate programs (correct, incorrect), relative correctness defines a partial ordering between candidate programs, whose maximal elements are the (absolutely) correct programs. While we acknowledge the significant advances achieved in the area of automated program repair, we feel that consideration of relative correctness in the workflow of program repair methods has the potential to make these methods more effective and more efficient.

- *Improved Effectiveness*. Program repair proceeds through two broad steps: *patch generation*, when candidate patches are generated from the original faulty program; and *patch validation*, when the generated patches are tested to assess their validity. As we discuss in related work, section V-B, current program repair methods perform patch validation through a number of criteria, which test various aspects/ dimensions/ approximations of relative correctness, but not relative correctness per se. Yet we feel that program repair is inconceivable without a vetted definition of relative correctness. In the same way that absolute correctness is the criterion by which we judge the derivation of a program P from a specification R, relative correctness ought to be the criterion by which we can judge that a candidate P' is a valid repair for program P with respect to specification R. In this paper, we propose a generic algorithm for program repair, which proceeds iteratively by enhancing relative correctness until it achieves absolute correctness.
- *Improved Efficiency*. The definition of relative correctness enables us, for a given level of granularity at which we want to model faults, to define the concept of *elementary fault removal*, which represents a unitary fault removal increment. This concept enables us, in turn, to distinguish between a single multi-site fault and multiple single-site faults. This distinction is important because if we are interested to remove several single-site faults (which are the most common type) then we can remove them one by one and test the program for relative correctness at each step; on the other hand, if we want to remove a multiple-site fault, then the relevant multiplicity is the number of sites in the faults (two or three, at most), not the number of faults in the program (unbounded).

In this paper, we briefly present a definition of relative correctness, due to [8, 19], then we use it to sketch an algorithm for program repair; our algorithm relies on the existence of a patch generator, and focuses exclusively on the patch validation step. In section II we introduce some elements of mathematical notation, then we present our definition of relative correctness and discuss why we feel that this definition is appropriate for our purposes. In section III we present our algorithm, and discuss its validity in light of the definitions given in section II. In section IV we show the results of an experiment where we apply the algorithm, albeit partially by hand for now (as its automation is under way) on sample programs from the Siemens Benchmark, and draw some lessons from our observations. Finally in section V we briefly summarize our findings and compare them to related work; in particular, we show how the solutions adopted by other researchers for patch validation use approximations of relative correctness, but not quite relative correctness as we define it and validate it.

II. Background

A. Relational Mathematics

We assume the reader familiar with simple relational mathematics and we briefly introduce some notations that we use throughout the paper. Given a program p that operates on some variables x and y, we let the *space* of p be the set S of all the values that the aggregate of variables $\langle x, y\rangle$ may take; elements of S are called *states* of the program, and are usually denoted by lower case s. A *relation* on set S is a subset of $S \times S$; constant relations on a set S include the empty relation (ϕ), the identity relation (I) and the universal relation ($L = S \times S$); operations on relations include the set theoretic operations of union, intersection, difference and complement; other operations include the product of two relations (denoted by $R \circ R'$, or RR' for short), the converse of a relation ($\widehat{R}$) and the domain of a relation ($dom(R)$). The *pre-restriction* of relation R to set T is denoted by $_{T\backslash}R$.

A relation R is said to be reflexive if and only if $I \subseteq R$, symmetric if and only if $R \subseteq \widehat{R}$, antisymmetric if and only if $R \cap \widehat{R} \subseteq I$, and transitive if and only if $RR \subseteq R$. A relation R is said to be *deterministic* if and only if $\widehat{R}R \subseteq I$.

B. Absolute Correctness and Relative Correctness

Refinement is a recurrent theme in the study of correctness; our version of refinement is defined as follows.

Definition 1: Given two relations R and R', we say that R' *refines* R (abbrev: $R' \sqsupseteq R$) if and only if $RL \cap R'L \cap (R \cup R') = R$.

Intuitively, this means that R' captures a stronger requirement (is harder to satisfy) than R.

Given a program p on space S, we define the function of p (denoted by P) as the set of pairs (s, s') such that if program p starts execution in state s it terminates in state s'. We may, by abuse of notation, refer to a program p by its function P.

Definition 2: A program p on space S is said to be *correct* with respect to specification R on S if and only if its function P refines R.

This definition is identical (modulo differences of notation) to traditional definitions of total correctness [12, 13, 17]. The following Proposition, due to [21] sets the stage for the definition of relative correctness.

Proposition 1: Given a specification R, a deterministic program p is correct with respect to R if and only if $dom(R \cap P) = dom(R)$.

The set $dom(R \cap P)$ is the set of initial states on which P behaves according to R; we call it the *competence domain* of P with respect to R.

Definition 3: Due to [19]. Given a specification R and two deterministic programs P and P', we say that P' is *more-correct* (resp. *strictly more-correct*) than P with respect to R if and only if $(R \cap P')L \supseteq (R \cap P)L$ (resp. $(R \cap P')L \supset (R \cap P)L$).

In [7] we generalize this definition to non-deterministic programs, and discuss why this generalization may be the key to scaling up. To contrast relative correctness with correctness (Definition 2), we may refer to the latter as *absolute correctness*. For deterministic programs P and P', P' is more-correct than P if and only if the competence domain of P' is a superset of that of P; note this does not mean that P' duplicates the correct behavior of P on its competence domain (rather P' may have a different correct behavior on the competence domain of P). See Figure 1.

How do we know that our definition of relative correctness is any good? Below are four properties that one would want a definition of relative correctness to satisfy; we find in [8] that our definition satisfies all of them.

- *Ordering Properties*. Relative correctness is reflexive and transitive, but not antisymmetric (i.e. two candidate programs could be equally correct, yet compute distinct functions).
- *Relative correctness culminates in absolute correctness*. An absolutely correct program is more-correct than any candidate program. We write this property as: $P' \sqsupseteq R \Leftrightarrow (\forall P : P' \sqsupseteq_R P)$.
- *Enhanced Correctness Implies Higher Reliability*. If P' is more-correct than P with respect to R, then it is more reliable than P; but more-reliable is not equivalent to more-correct: P' may be more reliable because its competence domain includes states that have higher probability of occurrence than those of the competence domain of P.
- *Relative Correctness and Refinement*. Program P' refines program P if and only if P' is more-correct than P with respect to *any* specification R. We write this property as: $P' \sqsupseteq P \Leftrightarrow (\forall R : P' \sqsupseteq_R P)$.

In order to contrast relative correctness with absolute correctness, we present an example of a specification and ten candidate programs, which we rank by relative correctness as shown in Figure 2; correct programs are at the top of the graph. We consider the specification R on space $S = \{a, b, c, d, e\}$:
$R = \{(a, a), (a, b), (a, c), (b, b), (b, c), (b, d), (c, c), (c, d), (c, e)\}$, and we consider the following candidate programs, along with their competence domains with respect to R:

- $P_0 = \{(a, d), (b, a)\}$. $CD_0 = \{\}$.
- $P_1 = \{(a, b), (b, e)\}$. $CD_1 = \{a\}$.
- $P_2 = \{(a, d), (b, c)\}$. $CD_2 = \{b\}$.
- $P_3 = \{(b, e), (c, d)\}$. $CD_3 = \{c\}$.
- $P_4 = \{(a, b), (b, c), (c, a)\}$. $CD_4 = \{a, b\}$.
- $P_5 = \{(a, d), (b, c), (c, d)\}$. $CD_5 = \{b, c\}$.
- $P_6 = \{(a, c), (b, e), (c, d)\}$. $CD_6 = \{a, c\}$.
- $P_7 = \{(a, a), (b, b), (c, c), (d, d)\}$. $CD_7 = \{a, b, c\}$.
- $P_8 = \{(a, b), (b, c), (c, d), (d, e)\}$. $CD_8 = \{a, b, c\}$.
- $P_9 = \{(a, c), (b, d), (c, e), (d, a)\}$. $CD_9 = \{a, b, c\}$.

See Figure 2; programs P_7, P_8, P_9 are (absolutely) correct while programs $P_0, P_1, P_2, P_3, P_4, P_5, P_6$ are incorrect. Figure 5 shows a more concrete example of programs ordered by relative correctness.

C. Faults and Fault Removals

Any definition of a fault must imply a level of granularity at which we want to isolate faults. We use the term *feature* to refer to any part of the source code at an appropriate level of granularity, including non-contiguous parts.

Definition 4: Due to [19]. Given a specification R and a program P, a *fault* in program P is any feature f that admits

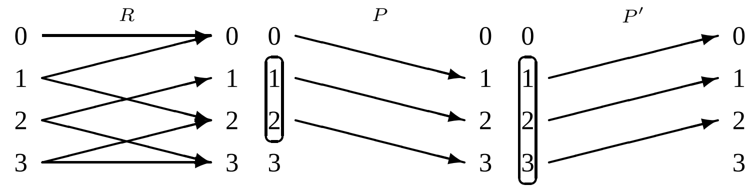

Fig. 1. $P' \sqsupseteq_R P$, Deterministic Programs

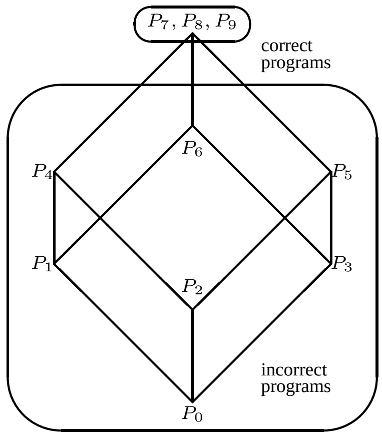

Fig. 2. Ordering Candidate Programs by Relative Correctness

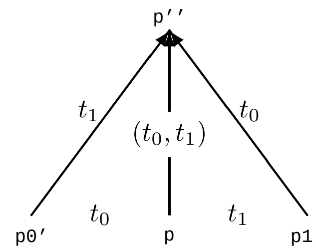

Fig. 3. One Two-site Fault

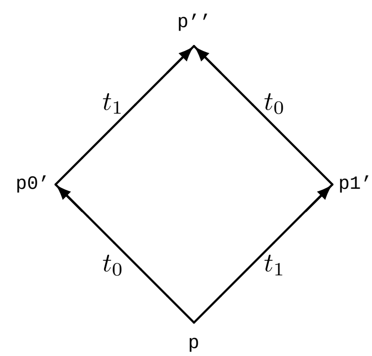

Fig. 4. Two One-site Faults

a substitute f' such that the program P' obtained from P by replacing f with f' is strictly more-correct than P. A *fault removal* in P is a pair of features (f, f') such that f is a feature in P and program P' obtained from P by replacing f with f' is strictly more-correct than P.

This definition of a fault encompasses cases where the feature in question is non-contiguous, i.e. it may involve two statements or for example two lexemes that are found in different locations of the source code. We consider a program P and a specification R and we assume that we have identified two statements, say f_0 and f_1 that admit substitutes, say f_0' and f_1', such that the program P' obtained from P by replacing f_0 by f_0' and f_1 by f_1' is strictly more-correct than P with respect to R. The question that we address is: do we have two single-site faults (f_0 and f_1) or a single two-site fault ($f = \langle f_0, f_1 \rangle$)? The answer depends on whether f_0 alone is a fault, and whether f_1 alone is a fault, whence the following definition.

Definition 5: Given a specification R and a program P, an *elementary fault* in program P is a fault such that no part of it is a fault.

All single-site faults are elementary faults; multi-site faults are elementary faults if and only if no subset of their elements is a fault. Figures 3 and 4 (where t_0 represents the transformation $f_0 \rightarrow f_0'$ and t_1 represents the transformation $f_1 \rightarrow f_1'$) show the contrast between a single two-site fault and two one-site faults: in Figure 3 we need to apply both transformations before the program becomes more-correct; when we apply t_0 alone (resp. t_1), we obtain p_0' (resp. p_1', which is not strictly more-cottect than p; it is only when we apply them both that we obtain a strictly more-correct program (p''). By contrast, Figure 4 illustrates a situation where each individual transformation raises the relative correctness of the program (both p_0' and p_1' are strictly more-correct than p, and p'' is strictly more-correct than p_0' and p_1', hence by transitivity strictly more-correct than p).

III. An Algorithm for Stepwise Program Repair

A. Oracle Design

We consider a program P' on space S and we are interested to design an oracle that tests the execution of P' on some initial state; the oracle takes the form of a binary predicate in (s, s'), where s is the initial state and s' is the final state. What form this oracle takes depends on what property we want to test about P'.

1) Absolute Correctness with respect to R: Given a specification R on space S, the oracle for absolute correctness with respect to R is denoted as $\Omega(s, s')$ and defined by:

$$\Omega(s, s') \equiv (s \in dom(R) \Rightarrow (s, s') \in R).$$

We find in [20] that if a program P satisfies the condition $\Omega(s, P(s))$ for all s in S then it is absolutely correct with respect to R. In practice, since we cannot check $\Omega(s, P(s))$

for all s in S, we check it for a bounded size test data T. Hence we define the following predicate:

$$\Omega_T(P') \equiv (\forall s \in T : \Omega(s, P'(s))).$$

We find in [20] that if a program P' satisfies this predicate then it is absolutely correct with respect to ${}_{T\backslash}R$.

2) Relative Correctness over a program P with respect to a specification R: Given a specification R on space S and a program P on S, the oracle for relative correctness over program P with respect to R is denoted by $\omega(s, s')$ and defined by:

$$\omega(s, s') \equiv (\Omega(s, P(s)) \Rightarrow \Omega(s, s')).$$

This formula stems readily from the definition of relative correctness; a program P' is more-correct than program P with respect to R if and only if $\omega(s, P'(s))$ holds for all s in S. In practice, since we cannot check $\omega(s, P'(s))$ for all s in S, we check it for a bounded size data set T. Hence we define the following predicate:

$$\omega_T(P') \equiv (\forall s \in T : \omega(s, P'(s))).$$

3) Strict Relative Correctness over a program P with respect to a specification R: A program P' is *strictly more-correct* than a program P with respect to a specification R if and only if P' is more-correct than P, and there exists at least one element s in S such that $\Omega(s, P'(s)) \wedge \neg\Omega(s, P(s))$. In practice, since we cannot check $\Omega(s, P'(s)) \wedge \neg\Omega(s, P(s))$ for all s in S, we check it for a bounded size data set T. Hence we define the following predicate:

$$\sigma_T(P') \equiv (\omega_T(P') \wedge (\exists s \in T : \Omega(s, P'(s)) \wedge \neg\Omega(s, P(s))))$$

.

B. Specification

We use the oracles discussed above to design a generic program repair algorithm; before we present the algorithm, as discuss its specification.

- *Inputs*:
  - A program P on S (where S can be inferred from the variable declarations of P).
  - Test data set, T, a subset of S.
  - Specification R on S, in the form of a binary C-like boolean function $R(s, sprime)$.
  - A specification of the domain of R, in the form of a unary C-like boolean function $domR(s)$.
- *Output*: Three possible outcomes, depending on patch generation:
  - A program P' that is absolutely correct with respect to ${}_{T\backslash}R$. Note that if P fails on some state of T and P' is absolutely correct with respect to ${}_{T\backslash}R$ then P' is strictly more correct than P (hence it is a repair of P) with respect to R.
  - A program P' that is strictly more-correct than P with respect to ${}_{T\backslash}R$, though possibly still incorrect.
  - A message to the effect that no correctness enhancement of P with respect to R is possible, given the existing patch generation capability.

Note that whereas other program repair methods require two test data sets (positive test data, negative test data), we do not need this information, because it can be inferred from the available input parameters: The positive test data is, actually ${}_{(T\cap CD)\backslash}R$, and the negative test data is ${}_{(T\cap\overline{CD})\backslash}R$, where CD is the competence domain of P with respect to R.

C. Algorithm

This algorithm relies on the availability of a patch generator, which takes the forms of two functions:

- `nextcandidate(base)`. Given a baseline program `base`, this function returns candidate repairs of `base` in a deterministic sequence; this can be a mutant generator, e.g., which takes `base` along with with some mutation parameters/ options and generates, in sequence, all the relevant mutants for the selected parameters.
- `morecandidates(base)`. Given a baseline program `base`, this boolean function returns true as long it has more candidate repairs to offer, false otherwise.

This algorithm is generic in the sense that it can be composed with any patch generator for which we can provide these two functions.

```
{programtype base=P;  programtype candidate=P;
bool exhausted=false;  bool enhanced=false;
while (! abscorT(candidate) && ! exhausted)
  {while (morecandidates(base) &&
          ! strictrelcor(candidate,base))
    {// no viable candidate, but we have more
    candidate = nextcandidate(base);}
    // if candidate is abs. correct done, else..
    if (! abscorT(candidate))
      {// analysis of exit condition
      if strictrelcorT(candidate,base)
        {// we let candidate be new base
        base = candidate;  enhanced=true;
        } //also reset patch generation
      else
        {// we ran out of candidates
        exhausted = true;}}}
if (! exhausted)
   {cout<<'Correct Program: '<<candidate<<endl;}
else
  if (enhanced)
    {cout<<'No correct program found. '<<endl;
     cout<<'Most correct: '<<candidate<<endl;}
  else
    {cout<<'No correctness enhancement'<<endl;}}
```

The following functions are a direct reflection of the formulas presented in section III-A.

```
bool abscor (candidate, inits)
  {stype s; s=inits; candidate();// alters s
  return (! domR(inits) || R(inits, s));}

bool abscorT(candidate)
  {bool abscorforall; abscorforall=true;
  forall (t in T)
    {abscorforall
     = abscorforall && abscor(candidate,t)};
    return abscorforall;}

bool relcor(candidate, base, inits)
```

```
  {stype s; s=inits; base();//alters s, not inits
  bool abscorbase = (!domR(inits)||R(inits,s));
  s=inits; candidate(); //alters s, not inits
  bool abscorcandidate=(!domR(inits)||R(inits,s));
  return (! abscorbase || abscorcandidate);}

bool relcorT(candidate, base)
  {bool relcorforall; relcorforall=true;
  forall (t in T)
    {relcorforall = relcorforall &&
                  relcor(candidate,base,t)};
    return relcorforall;}

bool strict (candidate, base,inits)
  {stype s; s=inits; base();//alters s, not inits
  bool abscorbase = (!domR(inits)||R(inits,s));
  s=inits; candidate(); //alters s, not inits
  bool abscorcandidate=(!domR(inits)||R(inits,s));
  return (! abscorbase && abscorcandidate);}

bool strictT (candidate, base)
  {bool strictforone; strictforone=false;
  forall (t in T)
    {strictforone = strictforone
                  || strict(candidate,base,t)};
    return strictforone;}

bool strictrelcorT (candidate, base)
  {return   relcorT(candidate,base)
      && strictT(candidate,base);}
```

IV. Illustration

A. Experimental Setup

For the purposes of our experiment, we carry out patch generation by means of a mutation generator, specifically muJava [5, 16]. According to the specification given in section III-B, we must provide the following parameters:

- *A Program to Repair*. We choose the `tcas` program taken from the Siemens benchmark, to which we apply eight modifications (faults) provided in the same benchmark [2, 10].
- *Test Data*. We take the test data set T (of size 1578) provided by the benchmark for this program.
- *Specification*. For the sake of this experiment, we use the original fault-free version of `tcas` as the specification; this yields the following code for R:

  ```
  bool R(s, sprime)  // initial, final states
     {tcas(); // modifies s, preserves sprime
      return (sprime==s);} // candidate = spec?
  ```

  To run this experiment with non-deterministic specifications, we are planning cases where the equality `(sprime==s)` is replaced by the weaker condition `(EQ(s,sprime))`, for some equivalence relations EQ; this is currently under way.
- *Specification domain*. Since we take the correct version of `tcas` as specification, and since this program is defined for all states in T, we let $domR(s)$ be *true*.

  ```
  bool domR(s) {return true;}
  ```

Though the algorithm, as written in section III-C, seeks to build a single path from the faulty version of a program to a correct version (by successive fault removals), what we execute for this experiment is a search for all the possible paths; instead of the inner while loop of the algorithm (section III-C) we actually execute a for loop that covers all the mutants of the current base and catalogs those mutants that are strictly more-correct than the base; the organizational part of this work (management of the evolving graph) is done by hand, as it is not yet fully automatic.

B. Experimental Observations

The resulting graph is shown in Figure 5; each iteration of the outer loop generates a new layer of the graph. The bottom of the graph is the faulty version of `tcas`, and the top is the correct version, as found in the Siemens benchmark. Note that even though we made eight modifications to the original program, our algorithm made only seven fault removals; this may be because the eighth modification does not change the function of the program (it is not a fault) or because the test data T is not large enough to distinguish the original program from the repaired program; in either case, the program at the top of the graph is certified to be absolutely correct with respect to ${}_{T\backslash}R$.

Now, note that even though the program at the bottom of the graph has seven faults, only four of them are visible (since there are four outgoing arcs from the bottom). What happened to the other three? They are masked, and can only be exposed as the first four are removed. The lesson we can draw from this observation: when we observe a failure of a program and we resolve to repair it, we should not define success as remedial to that particular failure, because the fault that causes that failure may be masked by other faults; rather we should view any enhancement in the relative correctness of the program as a measure of success/ progress. In other words, we do not get to decide in what order a program exposes its faults; rather we let the program reveal its faults in the order it determines.

We must acknowledge that what made our experiment look so successful is the combination of three conditions, which do not necessarily prevail in all instances: first, the mutant generator was parameterized in such a way as to perform mutations that are of the same nature and the same scale as the benchmark faults that were introduced; second, all the faults that were introduced are single-site faults, hence we were able to remove them by single mutations; third, we assume the availability of boolean functions that capture the specification R and its domain. The first condition pertains to patch generation, and is a difficult condition to fulfill in general, because it assumes that we know the nature/ scale of the faults. The second condition pertains to patch validation, and is relatively easy to fulfill: first because most faults are single-site faults; and second, because we can run multiple mutations to cover the rare cases where they are not. For illustration, we run the same experiment described above on the *replace* application of the Siemens benchmark, to which we have inserted six modifications. After four iterations (four fault removals) we reach a program that is more-correct than the original, but not absolutely correct; when we deploy double mutation, we break through, generating two separate programs that are absolutely correct with respect to ${}_{T\backslash}R$. So that we were able to remove five faults (four single-site faults and one double-site fault) by doing nothing more than double mutation; if we were using only absolute correctness as the criterion of

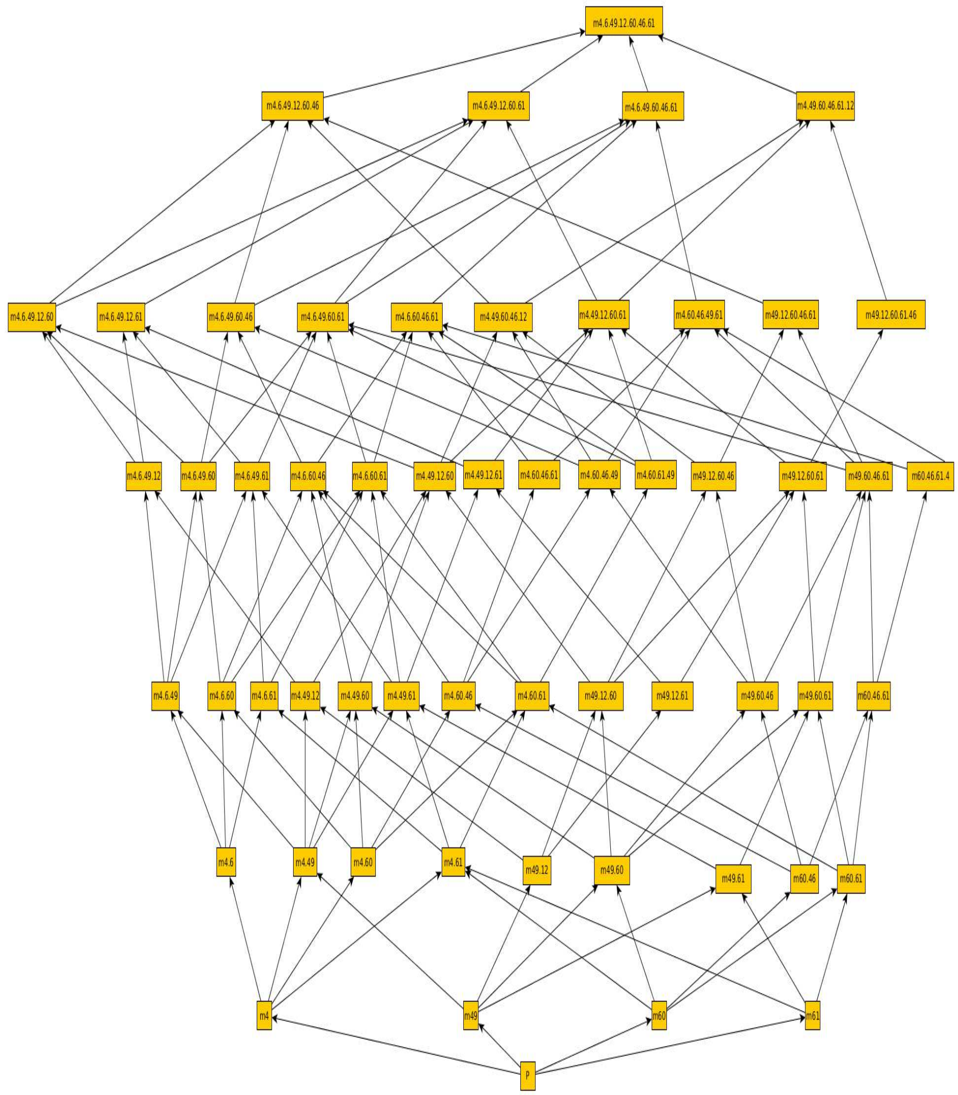

Fig. 5. Stepwise Repair of `tcas` Faults

success, we would have to apply sixtuple mutations to achieve the same result, an outrageously costly proposition. As for requiring predicates $R(s, s')$ and $domR(s)$, we admit that this may limit the scope of our approach; but we also argue that some form of specification is mandated by other methods to generate the required positive test data and the negative test data; all we are doing is making the requirement for $R(s, s')$ explicit.

V. Conclusion

A. Summary and Prospects

Relative correctness plays for program repair the same role that absolute correctness plays for program construction: In the same way that absolute correctness is the criterion by which we judge the derivation of a program from a specification, relative correctness ought to be the criterion by which we judge the transformation of a program P into a program P' through a repair operation. In this paper we derive the skeleton of an algorithm for program repair, which uses strict relative correctness oracles to perform patch validation. Our approach can be characterized by the following premises: it relies on formal definitions of correctness, relative correctness, and strict relative correctness; it derives test oracles from these definitions; it defines success/ progress as any strict enhancement of relative correctness, rather the remediation of a specific failure; it controls combinatorial divergence by removing faults in sequence rather than simultaneously.

What would be more interesting, perhaps, is to explore how we can use relative correctness, not to test existing repair candidates, but rather to generate repair candidates that are more-correct by construction. In the same way that correctness ideas were used by researchers such as Dijkstra [9], Gries [12], Hehner [13], Morgan [23] and others as a basis for correct-by-design programming, we can imagine ways to use relative correctness ideas to generate more-correct-by-design program repairs. This is clearly a long-term research goal, but one that promises great returns, since it has the potential to guide patch generation in addition to patch validation.

B. Related Work

We argue that our approach to patch validation, which is based on the concept of relative correctness, addresses some shortcomings in existing program repair technology, in terms of precision, recall, and efficiency [1, 3, 4, 6, 14, 15, 18, 22, 24, 25].

1) Loss of Recall: GenProg [11, 15], for example, generates candidate repairs by combining a set of elementary mutations and submitting each mutant to a set of positive test data (which the original program passes, and we want to preserve) and a set of negative test data (which the original program fails, and we want candidates to pass). This approach presents two impediments for good recall: First, this condition is sufficient for relative correctness but unnecessary. A candidate program may fail on the positive test data and still be more-correct than the original: because specifications are not necessarily deterministic, correct behavior is not necessarily unique. See Figure 1. Second, a candidate program P' may also fail on the negative test data and still be more-correct than the original program P; the competence domain of P' may be a superset of that of P, yet still does not overlap the negative test data. The loss of recall means that GenProg could very well generate valid program repairs, but fail to recognize them as such.

2) Loss of Precision: GenProg selects candidate repairs by maximizing a *fitness function*, which is computed as the weighted sum of all the test data on which the program runs; weights are assigned to test data according to their preponderance in some usage pattern, so that the fitness function is an approximation of the program's reliability. But we see in section II-B that relative correctness logically implies, but is not equivalent to, enhanced reliability. So that maximizing the fitness function is a necessary condition, but not a sufficient condition, of relative correctness.

3) Inefficiency: We recognize two sources of inefficiency in current practice of program repair. First, as we discuss in section IV-B, faults are prone to mask each other; so that if we define the success of a repair operation as the remediation of a specific failure caused by a specific fault, and that fault is masked by others, we may have to find a combination of patches that fix all the faults involved in this situation before the failing behavior is corrected. A more efficient approach may be to define success as an increase in relative correctness, and accept any patch that fulfills this criterion, until the targeted failure is remedied. Second, whenever they fail to distinguish between a single multi-site fault and several single-site faults, program repair methods may be pursuing unnecessary and costly multiple patches where successive single patches would have been sufficient.

REFERENCES

[1] Martinez M. and Monperrus M. Mining software repair models for reasoning on the search space of automated program fixing. *Empirical Software Engineering*, 2013.

[2] Benchmark. Siemens suite. Technical report, Georgia Institute of Technology, January 2007.

[3] Kim D., Nam J., Song J., and Kim S. Automatic patch generation learned from human-written patches. In *ICSE 2013*, pages 802–811, 2013.

[4] Vidroha Debroy and W. Eric Wong. Combining mutation and fault localization for automated program debugging. *Journal of Systems and Software*, 90:45–60, 2013.

[5] Marcio Eduardo Delamaro, Jose Carlos Maldonado, and Auri Marcelo Rizzo Vincenzi. Proteum /im 2.0: An integrated mutation testing environment. In W. Eric Wong, editor, *Mutation Testing for the New Century*, volume 24, pages 91–101. Springer Verlag, 2001.

[6] F. DeMarco, J. Xuan, D.L. Berra, and M. Monperrus. Automatic repair of buggy if conditions and missing preconditions with smt. In *Proceedings, CSTVA*, pages 30–39, 2014.

[7] J. Desharnais, N. Diallo, W. Ghardallou, M. F. Frias, A. Jaoua, and A. Mili. Relational mathematics for relative correctness. In *RAMICS, 2015*, volume 9348 of *LNCS*, pages 191–208, Braga, Portugal, September 2015. Springer Verlag.

[8] Nafi Diallo, Wided Ghardallou, and Ali Mili. Correctness and relative correctness. In *Proceedings, 37th International Conference on Software Engineering, NIER track*, Firenze, Italy, May 20–22 2015.

[9] E.W. Dijkstra. *A Discipline of Programming*. Prentice Hall, 1976.

[10] Hyunsook Do, Sebastian Elbaum, and Gregg Rothermel. Supporting a controlled experimentation with testing techniques: An infrastructure and its potential impact. *Empirical Software Engineering: An International Journal*, 10(4):405–435, 2007.

[11] C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer. Genprog: A generic method for automated software repair. *IEEE Transactions on Software Engineering*, 31(1), 2012.

[12] David Gries. *The Science of Programming*. Springer Verlag, 1981.

[13] Eric C.R. Hehner. *A Practical Theory of Programming*. Prentice Hall, 1992.

[14] Claire LeGoues, Stephanie Forrest, and Westley Weimer. Current challenges in automatic software repair. *Software Quality Journal*, 21(3):421–443, 2013.

[15] Claire LeGoues, M. Dewey Vogt, S. Forrest, and W. Weimer. A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each. In *Proceedings, ICSE 2012*, pages 3–13, 2012.

[16] Yu-Seung Ma, Jeff Offutt, and Yong-Rae Kwon. Mujava : An automated class mutation system. *Journal of Software Testing, Verification and Reliability*, 15(2):97–133, June 2005.

[17] Zohar Manna. *A Mathematical Theory of Computation*. McGraw-Hill, 1974.

[18] Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. Angelix: Scalable multiline program patch synthesis via symbolic analysis. In *Proceedings, ICSE 2016*, Austin, TX, May 2016.

[19] A. Mili, M. Frias, and A. Jaoua. On faults and faulty programs. In P. Hoefner, P. Jipsen, W. Kahl, and M. E. Mueller, editors, *Proceedings, RAMICS 2014*, volume 8428 of *LNCS*, pages 191–207, 2014.

[20] Ali Mili and Fairouz Tchier. *Software Testing: Operations and Concepts*. John Wiley and Sons, 2015.

[21] Harlan D. Mills, Victor R. Basili, John D. Gannon, and Dick R. Hamlet. *Structured Programming: A Mathematical Approach*. Allyn and Bacon, Boston, Ma, 1986.

[22] Martin Monperrus. A critical review of patch generation learned from human written patches: Essay on the problem statement and evaluation of automatic software repair. In *Proceedings, ICSE 2014*, Hyderabad, India, 2014.

[23] Carroll C. Morgan. *Programming from Specifications, Second Edition*. International Series in Computer Sciences. Prentice Hall, London, UK, 1998.

[24] Hoang Duong Thien Nguyen, DaWei Qi, Abhik Roychoudhury, and Satish Chandra. Semfix: Program repair via semantic analysis. In *Proceedings, ICSE*, pages 772–781, 2013.

[25] Zhchao Qi, Fan Long, Sara Achour, and Martin Rinard. An analysis of patch plausibility and correctness for generate-and-validate patch generation systems. In *Proceedings, ISSTA 2015*, Baltimore, MD, July 2015.