Toward a Theory of Program Repair

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

Aleksandr Zakharchenko
NJIT, Newark NJ
az68@njit.edu

Matias Martinez
University of Valenciennes, France
matias.sebastian.martinez@gmail.com

Ali Mili
NJIT, Newark NJ
mili@njit.edu

***Abstract*—As a field of research and development, the discipline of program repair has made great strides over the past decade, producing a wide range of effective engineering solutions. In this paper, we use the concept of relative correctness to propose a theory of program repair, on the grounds that to repair a program means to make it more-correct. We discuss how this theory can be used to enhance the current state of the art, and what promise it holds for the future of the discipline.**

***Index Terms*—Absolute correctness, relative correctness, program repair, test oracles, fault removal, fault, failure.**

I. Introduction: Why Do We Need a Theory?

As a field of research and development, the discipline of program repair has made great strides over the past decade, producing a continuous stream of increasingly sophisticated engineering solutions spanning several programming languages and several categories of faults [2, 7, 9, 10, 16, 18, 24–26, 32, 36, 39, 40, 42, 45, 46, **?**, 46]. In [**?**], Gazzola et. al. present a sweeping survey of program repair, spanning more than two decades and encompassing more than 100 papers, and conclude that it is important to improve the maturity of the field and obtain a better understanding of useful strategies and heuristics. The following premises, which form the basis of our paper, are consistent with the above conclusions.

- As an engineering discipline, the field of program repair can achieve the desired level of maturity by evolving theoretical foundations, which would be used to analyze, verify and (longer term) design program repair solutions.
- Any theoretical approach to program repair ought to be based on some concept of *relative correctness*, i.e. the property of a program to be more-correct than another with respect to some specification. Indeed, to repair a program means to make it more-correct, in some sense; hence any program repair method ought to be based on some definition of relative correctness, and ought to be validated by showing that it does enhance correctness, as defined.

To practice what we preach, we present in this paper a sample definition of relative correctness; whereas traditional (absolute) correctness partitions the set of candidate programs into two categories (correct vs incorrect), relative correctness defines a partial ordering among candidate programs, whose maximal elements are the absolutely correct programs. We validate our definition of relative correctness against canonical attributes (given in section **??**), then we use it to design a *generic* algorithm of program repair. This algorithm is an instance of the *generate-and-validate* model of program repair, and it is generic in the sense that it specifies how candidate repairs are validated, but does not specify how they are generated; hence we can create as many instances of this algorithm as we have generators of repair candidates. To illustrate this algorithm, we combine its validation module with GenProg's generation module, and show that the resulting tool is more powerful than the original GenProg tool [16].

Given that the discipline of programa repair has been very successful in producing increasingly sophisticated engineering solutions, it is legitimate to ask the question: why do we need a theory? To address this question, we present in section **??** a brief survey of the state of the art, then we discuss in what way and to what extent some of the issues in this discipline stem from the absence of theoretical foundations, and most notably from the absence of a concept of relative correctness. On the other hand, we argue that it is not uncommon for an engineering discipline to precede the theory that can be used as its foundation: Consider that programming in its modern form (using high level programming languages) emerged in the mid to late nineteen fifties with the emergence of such languages as Fortran, Cobol, and Algol; yet, it is only in the late nineteen sixties that theories of program correctness emerged to provide a theoretical foundation thereof [14, 20]. We argue that in the same way that traditional (absolute) correctness provides the theoretical basis of program derivation from a specification, relative correctness ought to provide the theoretical basis of the transformation of a program to make it more-correct with respect to a specification. In the same way that absolute correctness is used to judge the derivation of a program from a specification, relative correctness ought to be used to judge the transformation of a program by program repair with respect to a specification.

We present our definition of relative correctness in section [**?**], and we contrast it to the traditional notion of (absolute) correctness. In section **??** we present our generic algorithm, and prove its correctness using Hoare-like logic [20]; then in section **??** we illustrate it by executing it on benchmark data and comparing its performance to GenProg. We do not claim that ours is the only definition of relative correctness, nor that it is the best definition. What we do argue, however, is that any program repair method or tool ought to include an explicit specification of a concept of relative correctness (as we do insection **??**), and some evidence that the method/ tool does indeed enhance correctness as defined (as we do in section **??**).

II. State of the Art: Role of a Theory

A. A Survey

Program repair has attracted so much attention recently that it is nearly impossible to do justice to all the relevant work in this area; we limit ourselves to a sample of references.

1) Generate-and-Validate Methods: In [16, 44], Weimer at al. present *GenProg*, a program repair tool that generates candidate repairs using genetic programming. This tool uses three artifacts, namely the faulty program, a set of cases (divided into positive tests, and negative tests), and (in some versions) a fitness function that is used to rank repair candidates. GenProg manipulates programs at different levels of abstraction, including the abstract syntax tree and the source code, and it uses quantitative/ semantic criteria as well as qualitative/ syntactic criteria to select repair candidates; it favors repairs that involve minimal patches.

In [23], Kim et al. argue that because of its random mutation operators, GenProg is prone to generate too many non-viable patches, which impede its performance and its precision. Also, they propose a catalog of common faults, along with corresponding repair templates, which they have compiled from an analysis of more than 60 000 human-generated patches. Kim et al. implement their method in an automated tool, called *PAR*, for Pattern Based Automatic Program Repair. By design, PAR handles only single-site faults. While *PAR* uses a different (from GenProg) approach for patch generation, it adopts essentially the same patch validation approach.

In [28, 29] Long and Rinard present a program repair method (called *SPR*: Staged Program Repair), whose patch generation method relies on three techniques: The first deploys a set of fault-specific *parameterized transformation schemas*.faults.The second technique attempts to derive parameter values of the transformation schema to ensure successful repair. The third technique aims to fine tune the parameters of the transformation schemas.Long and Rinard argue that the combination of these three techniques reduces the search space by two orders of magnitude. As for patch validation, SPR checks repair candidates for a variable combination of positive tests and negative tests. In [27] Long and Rinard integrate the staged program repair (SPR) method into a tool they call *Prophet*, which learns patch generation by analyzing a database of past successful patches. Patch validation assigns a score to each candidate patch by considering its plausibility, along with relevant patch characteristics; then candidate patches are tested in the order of decreasing score; the first candidate that runs successfully on all the test cases is selected.

In [24] Le et al. argue that semantic based repair techniques typically suffer from weak specifications, which produce sparse search spaces. Also, they argue that many repair methods are prone to overfitting because their patches are too specific to the test data used for patch validation. To enhance the generality of patches, they propose a three-pronged approach: a domain-specific language to constrain the search space; an efficient search strategy; and ranking functions that prioritize patches that are most likely to generalize. They validate their approach on standard benchmarks and on real-world programs.

In [46], Xin and Reiss introduce an automated program repair technique (*ssFix*) that retrieves patches from a code database, which includes segments from the local faulty program as well as an auxiliary code repository. Once suspicious statements are identified, *ssFix* performs syntactic code search to find candidate code segments that are structurally similar and conceptually related to the target statement and its local context (refered to as a *chunk*). The tool selects a patch by unifying repair candidates against the chunk defined by the target statement and its context, then choosing the candidate that passes the validation test while minimizing discrepancies with the target chunk. Xin and Reiss validate their approach using the Defect4J benchmark.

In [45], Wen et al. use empirical data from past research to fine-tune a program repair method that is based on two premises: first, that patch generation is better carried out at a fine level of granularity; second, that context information can be used to steer the mutation operators to patches that are most likely to be correct. They test their prototype, called *CapGen*, on the Defect4J benchmark, and find that it outperforms and complements existing tools and methods.

The main difficulty with generate-and-validate methods is their predisposition to combinatorial explosion: they are prone to generate far too many candidate patches, without a commensurate gain in recall; hence it may be advantageous to push the validation step upstream, to reduce the potential for generating useless patches. This is the subject of the next Section.

2) Constraint-Based Methods: In [39] Nguyen et al. introduce *SemFix*, an automatic repair method that takes as input a faulty program and a test data set, divided into positive set and negative set, and proceeds in three steps: First it uses fault localization techniques to identify program locations that are most likely to be faults. Then it considers these faults in sequence by order of decreasing probability, and for each fault it infers a local specification by deriving constraints on its behavior from controlled symbolic execution. Then it deploys program synthesis techniques to derive a substitute for the faulty statement that ensures the successful execution of the program for all the test data. Because SemFix relies on symbolic execution, it can only be applied to small programs that have no loops, a clear obstacle to scalability.

To overcome this shortcoming, Mechtaev et al. [32] introduce a new tool, they call *Angelix*, which relies on fault localization and program synthesis but uses a new artifact to represent the semantic constraints they impose on the search space of patches. This artifact, which they call the *Angelic Forest*, is a set of *Angelic Paths*, where each path represents an execution trace of the program in which expressions are associated with constraints that ensure the successful execution of the program. The main characteristic of this artifact is the fact that it is independent of the size of the program under repair. This property forms the basis for Angelix's claim of scalability, a claim that is borne out by empirical observations.

In [24] Le et al. introduce *JFix*, a semantics-based program repair method that extends *Angelix* to deal with Java programs, using Symbolic PathFinder, a symbolic execution engine for Java programs. Though it extends Angelix specifically, it is designed to be sufficiently generic to support other repair methods as well.

In [11, 47] DeMarco et al. present a program repair tool that is geared toward repair conditions in if-statement and unguarded statements.This tool, called *Nopol*, takes as input a faulty program and two sets of test data (positive and negative), It proceeds by instrumenting the program, executing it on the positive and negative test data and keeping track of the values that the program state takes at different locations of its source code. This trace information is then compiled into a SMT problem, which is submitted to an SMT solver, and if a solution exists, it is translated into a source code patch in the form of a modified if-condition or a newly generated guard.

In [18] Gupta et al. aim to fix what they call *common C language errors*; most of these are of a syntactic nature, i.e. of the same level that compilers are supposed to detect. But while a compiler merely declares that there is a fault, *DeepFix* also identifies the location that must be repaired and generates a fix to the fault. Gupta et al. model the problem of program repair as a mapping from an input sequence (the erroneous program) to an output sequence (the fixed program). To this effect, they design a special type of neural network, and train it on correct C programs. They use the compiler to select repair candidates, and to monitor progress towards a repair candidate (in the case of multiple faults).

In [43] Tan and Chodhoury propose *ReliFix*, an automated tool for repairing software regressions that may result from program changes. They propose five criteria to guide the repair operation, which are: limit repairs to small changes; produce readable code; pass progression tests; pass previously failed regression tests; ensure that no new regression is introduced. The ReliFix approach is based on the hypothesis that the fixes for regression faults can be crafted using code from the original program, prior to the change, and the changed program. Patch validation checks for successful execution on test data.

In [21] Ke et al. propose a method for program repair, called *SearchRepair*, which relies on software reuse technology. This method is based on a repository of human generated code fragments that are mined from open source software and codified using specifications formulated as SMT constraints. When a program fragment is suspected of being faulty, a specification of the desired behavior is derived as SMT constraints on local input behavior. Then a constraint solver is invoked to find a match for the search key in the repository. Retrieved code fragments are validated syntactically (by ensuring that they compile) and semantically (by ensuring that they pass test criteria).

B. A Critique

In this section, we discuss in what way and to what extent the adoption of a concept of relative correctness may enhance the state of the art in program repair, and may address some of the current issues thereof. Because we are commenting on a wide range of diverse methods, the statements we make are necessarily broad generalizations; as such, they may fail to do justice to individual methods, but serve as convenient abstractions for the sake of our discussions.

- *Limited Scope*. In the absence of a definition of relative correctness, program repair methods are limited to programs that are within striking distance of absolute correctness (i.e. programs for which the patch generator can generate an absolutely correct program from the current faulty program). With relative correctness, we can repair a program P to obtain a program P', where P' is more-correct than P without being absolutely correct.
- *Preserving Correctness vs. Preserving Correct Behavior*. To validate candidate repairs, some repair methods check that candidates preserve the correct behavior of the original program (represented by the so-called *positive test data*). As we see in section **??**, preserving the correct behavior of the original program is a sufficient condition of relative correctness, but an unnecessary condition. Hence using the preservation of correct behavior as the criterion of patch validation leads to a loss of recall, since it rules out programs that are more-correct than the original without duplicating its correct behavior.
- *Relative Correctness vs. Fitness Functions*. To select candidate repairs, some repair methods maximize some user-defined (or system-defined, by default) fitness function. We argue that fitness functions are approximations of reliability, and we find in section **??** that enhanced reliability is a necessary condition of relative correctness, but not a sufficient condition. Hence using fitness functions as the criterion of patch validation leads to a aloss of precision, since it retrieves programs that are more reliable, but not necessarily more-correct, than the original program.
- *Removing a Fault vs. Remedying a Failure*. Most program repair methods consider negative test data provided by the user, and focus on modifying the program until it corrects the observed failure. We argue that this approach is counter-productive because when we resolve to remedy a given failure, we have no idea how much change we need to apply to (i.e. how many faults we need to remove from) the program before the failure is remedied. A more judicious approach is to let *fault removal*, rather than *failure remediation* drive this process; in [**?**, 12] we define *fault removal* as any modification of the program that makes it more-correct. Hence rather than ask the question: *what program modification remedies the observed failure*, we ask the question: *what minimal program modification enhances the relative correctness of the program?* If we keep enhancing the correctness of the program repeatedly, then eventually we will remove enough faults to remedy whatever failure we have originally observed. But we do it in such a way that we let the program expose its faults in the order it determines, and we remove them as they

appear, rather than pinning down a target observation of failure and trying to find the combination of faults that remedies that particular failure.

- *Correctness Enhancement and Test Size*. Many program repair methods mandate or assume a small negative test data set, sometimes a singleton. While this may be grounded in a divide-and-conquer philosophy, we find that for our purposes, this may be counter-productive. As we shall see in section **??**, having a large set of negative test data is useful because it affords us greater precision to track the evolution of the program towards absolute correctness; this is borne out by findings of Kong et al. [**?**].
- *Support for Patch Generation*. Theories of program correctness emerged in the late nineteen sixties [14, 20], and were used initially to verify the correctness of programs with respect to specifications. It took several years, even decades, for such theories to be turned into methodologies for deriving correct-by-design programs [5, 17, 19, 37]. Likewise, we envision that while relative correctness can be used for patch validation, as we show in this paper, it may, in time, be turned into constructive methods for patch generation (if we learn how to construct more-correct-by-design versions of a given program). This is briefly/ summarily discussed in Section IX.

III. Criteria for Relative Correctness

Before we propose a definition of relative correctness, we consider the question: *What constitutes a sound definition of relative correctness?* In other words, how cana we tell that our definition is anay good? We have identified four properties we believe characterize relative correctness.

- *Reflexivity and Transitivity, but no Antisymmetry*. We want to think of relative correctness as being a partial ordering, except we do not want it to be antisymmetric: in other words, we want to admit that two programs be mutually more-correct (each is more-correct than the other) and yet distinct; in particular, two programs may both be absolutely correct yet distinct, of course. Note that a more faithful name for this ordering may be: *more-correct-than-or-as-correct-as*; yet for the same of convenience we use the shorter version; andd whenever we want to exclude the clause *as-correct-as* we will talk about being *strictly more-correct*.
- *Culmination in Absolute Correctness*. We want relative correctness to culminate in absolute correctness, in the sense that if we make a program increasingly more-correct it will eventually be absolutely correct. In other words, an absolutely correct program is more-correct than any candidate program.
- *Relative Correctness and Reliability*. We define the reliability of a program in terms of two parameters, namely the specification with respect to which correct and incorrect behavior is judged, and the probability distribution of the possible inputs (aka the program's *operational profile* [38]). We want to think that if P' is more correct than P with respectc to specification R then for any operational profile, P' is more reliable than P with respect to the same specification. But the opposite is not true, i.e. P' may be more reliable than P without being more-correct: we want to think of relative correctness as a logical property whereas reliability is a stochastic property (that depends on the operational profile); also we certainly do not want to think of *more-correct* as being just another name for *more reliable*.
- *Relative Correctness and Refinement*. Refinement is a binary relation between two programs that means essentially that one program is more capable than another: P' refines P if and only if whatever P can do, P' does as well or better. By contrast, relative correctness is a tri-partite relation that means: P' is better than P for the purposes of a particulara specification R. Given these two interpretations, we want to think that P' refines P if and only if P' is more-correct than P with respect to any specification R.

These are the four properties that, we feel, a definition of relative correctness ought to satisfy; in section **??** we check that our definition satisfies them all. We do not claim that this list is complete; if other desired properties emerge, we will be anxious to check them as well.

IV. Mathematics of Relative Correctness

A. Relational Mathematics

We assume the reader familiar with elementary relational algebra [6], hence this section is not a tutorial on relations as much as it is an introduction to some definitions and notations. We represent sets in a program-like notation by writing variable names and associated data types; if we write S as: `x: X; y: Y;` then we mean to let S be the cartesian product $S = X \times Y$; elements of S are usually denoted by s and the X- (resp. Y-) component of s is denoted by $x(s)$ (resp. $y(s)$). When no ambiguity arises, we may write x for $x(s)$, and x' for $x(s')$, etc. A *relation* R on set S is a subset of $S \times S$. Special relations on S include the *universal relation* $L = S \times S$, the identity relation $I = \{(s, s) | s \in S\}$ and the empty relation $\phi = \{\}$. Operations on relations include the set theoretic operations of union, intersection, difference and complement; they also include the *converse* of a relation R defined by $\widehat{R} = \{(s, s') | (s', s) \in R\}$, the *domain* of a relation defined by $dom(R) = \{s | \exists s' : (s, s') \in R\}$, the *range* of a relation defined by $rng(R) = dom(\widehat{R})$, and the product of two relations R and R' defined by: $R \circ R' = \{(s, s') | \exists s'' : (s, s'') \in R \wedge (s'', s') \in R'\}$; when no ambiguity arises, we may write RR' for $R \circ R'$. The *pre-restriction* of relation R to set T is the relation denoted by ${}_{T\backslash}R$ and defined by: ${}_{T\backslash}R = \{(s, s') | s \in T \wedge (s, s') \in R\}$.

A relation R is said to be reflexive if and only if $I \subseteq R$, symmetric if and only if $R = \widehat{R}$, antisymmetric if and only if $R \cap \widehat{R} \subseteq I$, asymmetric if and only if $R \cap \widehat{R} = \phi$ and transitive if and only if $RR \subseteq R$. A relation R is said to be *total* if and only if $I \subseteq R\widehat{R}$ and *deterministic* (or: a function) if and only

if $\widehat{R}R \subseteq I$. A relation R is said to be a *vector* if and only if $RL = R$; vectors have the form $R = A \times S$ for some subset A of S; we use them as relational representations of sets. In particular, note that RL can be written as $dom(R) \times S$; we use it as a relational representation of the domain of R. We admit without proof (a well known property of functions) that if F and G are functions then $F = G$ if and only if $F \subseteq G$ and $GL \subseteq FL$.

B. Program Semantics

Given a program p on space S written in a C-like notation, we define the function of p (denoted by P) as the function that p defines on S, i.e. the set of pairs (s, s') such that if program p starts execution in state s it terminates in state s'; we may, when no ambiguity arises, refer to a program and its function by the same name, P. Since the criteria of relative correctness discussed in section **??** involve the property of refinement, we define this property below.

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

Intuitively, this definition means that R' has a larger domain than R and that on the domain of R, R' assigns fewer images to each argument that does R.We admit without proof that if R and R' are deterministic then R' refines R if and only if R' is a superset of R.

V. Absolute and Relative Correctness

We limit the scope of this paper to deterministic programs, i.e. programs that produce a single final state for any initial state; in section **??**, we generalize the definition of relative correctness to non-deterministic programs, and explain why this is useful.

A. Definitions

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.

The following Proposition, due to [35], helps introduce the definition of relative correctness.

Proposition 1: Due to Mills et al. [35]. Given a specification R and a program P, program P is correct with respect to R if and only if $(R \cap P)L = RL$.

We refer to the domain of $(R \cap P)$ as the *competence domain* of P with respect to R. The following derivation shows that this formula of correctness is identical, modulo differences of notation, to traditional definitions of total correctness [17, 19, 20, 31].

Proof. Since $(R \cap P)L \subseteq RL$ is a tautology, the condition of this proposition is equivalent to $RL \subseteq (R \cap P)L$, which we interpret as follows:

$\forall s : s \in dom(R) \Rightarrow s \in dom(R \cap P)$
{Interpreting the definition of domain}
$\forall s : s \in dom(R) \Rightarrow \exists s' : (s, s') \in (R \cap P)$
{P is deterministic}
$\forall s : s \in dom(R) \Rightarrow \exists s' : s' = P(s) \wedge (s, s') \in R$
{substitution}
$\forall s : s \in dom(R) \Rightarrow \exists s' : s' = P(s) \wedge (s, P(s)) \in R$
{Interpreting the definition of domain}
$\forall s : s \in dom(R) \Rightarrow s \in dom(P) \wedge (s, P(s)) \in R$

qed

If we interpret $s \in dom(R)$ as *s satisfies the precondition*, $s \in dom(P)$ as *execution of P on s terminates normally*, and $(s, P(s)) \in R$ as *the final state ($P(S)$) satisfies the postcondition*, then this formula mimicking the exact definition of total correctness as defined in, e.g. [17, 19, 20, 31].

Definition 3: Due to [33]. 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. $(R \cap P')L \supset (R \cap P)L$).

To contrast relative correctness with correctness (Definition 2), we may refer to the latter as *absolute correctness*. See Figure 1. Specification R is shown in the middle. To the left, we show two programs, Q and Q', such that Q' is more-correct than Q with respect to R; to the right, we show two programs P and P' such that P' is more-correct than P with respect to R; the competence domain of each program is indicated by the ovals. In reference to our discussion in Section II-B about the difference between preserving correct behavior and preserving correctness, notice that Q' is more-correct than Q by virtue of imitating the correct behavior of Q, whereas P' is more-correct than P by virtue of a different correct behavior.

More generally, program P' preserves the correct behavior of program P with respect to R if and only if $(R \cap P) \subseteq (R \cap P')$; whereas P' is more-correct than P with respect to R if and only if $(R \cap P)L \subseteq (R \cap P')L$; clearly the latter logically implies (but is not equivalent to) the former.

As a concrete, yet simple, example illustrating relative correctness, let S be the space defined by the following (C++) declarations:

```
int a[10]; int avg;
```

where we assume that all entries of the array are non-negative; let R be the following specification on S:

$R = \{(s, s') | avg' = \frac{\sum_{i=0}^{9} a[i]}{10}\}$,

and let P and P' be the following programs on S.

```
void P()
   {int sum=0;
    for (int i=0; i<10;i++) {sum=sum+a[i];}
    avg = sum/10;}

void Pprime()
   {avg=0;
    for (int i=0; i<10;i++) {avg = avg + a[i]/10;}
```

We argue that P' is more-correct than P with respect to R, because P' is less prone to arithmetic overflow than P. Indeed, we find that their competence domains (denoted respectively by CD and CD') are:

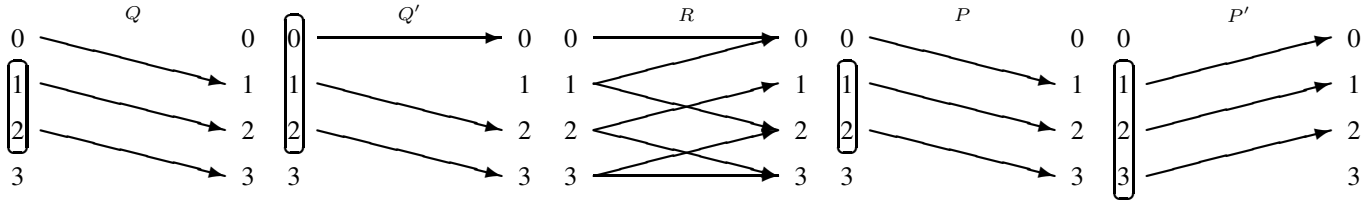


Fig. 1. $(Q' \sqsupseteq_R Q)$, $(P' \sqsupseteq_R P)$: Preserving Correctness $((R \cap P)L \subseteq (R \cap P')L)$ vs. Preserving Correct Behavior $((R \cap Q) \subseteq (R \cap Q'))$

$CD = \{s | \sum_{i=0} 9a[i] \leq MaxInt\}$,
$CD' = \{s | \sum_{i=0} 9a[i] \leq 10 \times MaxInt\}$.

For another example of illustration of relative correctness, consider the graph of Figure 4: the nodes in this graph represent mutants of the *tcas* component of the Siemens benchmark [**?**]; the node at the top of the graph represents the original *tcas* component, whereas the node at the bottom of the graph represents the *tcas* component in which we have seeded eight changes (faults) provided within the benchmark; each arc represents a relative correctness relation between the nodes it connectss (the higher node in the graph is more-correct than the lower node).

B. Properties

How do we know that our definition of relative correctness is any good? One way to gain some confidence in the soundness of our definition is to check whether it satisfies the properties listed in section **??**.

1) Ordering Properties: Relative correctness is clearly reflexive and transitive, by virtue of its definition. It is not antisymmetric, as can be seen from the following trivial example:

$S = \{0, 1\}$.
$R = \{(0, 0), (0, 1)\}$.
$P = \{(0, 0)\}$.
$P' = \{(0, 1)\}$.

2) Culminating in Absolute Correctness:

Proposition 2: Given a specification R on space S and a program P on S. P is absolutely correct with respect to R if and only if P is more-correct than any candidate program Q on S.

Proof. Proof of necessity. As a set theoretic tautology, we have $(R \cap Q)L \subseteq RL$ for any Q. Given that P is absolutely correct with respect to R, we have, by proposition 1: $(R \cap P)L = RL$.

Proof of sufficiency. Let P' be more-correct than any program with respect to R and let P be an absolutely correct program. From $(R \cap P')L \supseteq (R \cap P)L$ and $(R \cap P)L = RL$ we infer that P' is absolutely correct. **qed**

3) Relative Correctness and Reliability: The reliability of a program is defined in terms of two parameters: the specification that the program is intended to satisfy, and the operational profile of the program (as it is intended to be used). We represent the specification by a relation on the space of the program, and we represent the operational profile by a probability distribution on the domain of the specification.

Definition 4: The *reliability* of program P on space S with respect to specification R on S and operational profile $\theta()$ is denoted by $\rho_R^\theta(P)$ and defined as the probability that execution of P on a random state of $dom(R)$ selected according to $\theta()$ terminates normally in a state s' that satisfies specification R With this definition of reliability, we have the following proposition.

Proposition 3: Given two programs P and P' on space S and a specification R on S, and ana operational profile $\theta()$ on the domain of R, if P' is more-correct than P with respect to R then the reliability of P' with respect to R and $\theta()$ is greater than or equal to that of P.

Proof. According to definition 4, the reliability of P with respect to R and $\theta()$ can be interpreted as the probability that a random state of $dom(R)$ selected according to probability distribution $\theta()$ satisfies the condition $(s, P(s)) \in R$; this condition can be rewritten as: $s \in dom(R \cap P)$. Hence the reliability of P with respect to R and $\theta()$ can be written as:
$\rho_R^\theta(P) = \sum_{s \in dom(R \cap P)} \theta(s)$.
Likewise,
$\rho_R^\theta(P') = \sum_{s \in dom(R \cap P')} \theta(s)$.
Since $dom(R \cap P) \subseteq dom(R \cap P')$, we infer: $\rho_R^\theta(P) \leq \rho_R^\theta(P')$. **qed**

4) Relative Correctness and Refinement:

Proposition 4: Program P' refines program P if and only if P' is more-correct than P with respect to any specification R.

Proof. Necessity stems readily from the definition of relative correctness and the property that for deterministic programs P and P', $P' \sqsupseteq P$ if and only if $P' \supseteq P$.

Proof of Sufficiency. We take $R = P$ and we write that P' is more-correct than P with respect to P, which yields $(P \cap P')L \supseteq PL$. This, in conjunction with the tautology $(P \cap P') \subseteq P$, and the hypothesis that P and P' (hence also $(P \cap P')$ are functions, yields $P \cap P' = P$, i.e. $P' \supseteq P$, whence we infer that P' refines P. **qed**

C. Relative Correctness: Non Deterministic Programs

Given that all programming languages to which we apply program repair methods are deterministic, one may wonder why we need to define relative correctness for non-deterministic programs. There are two reasons, of unequal importance:

- The main reason is scalability: Non deterministic relations enable us to capture relevant functional attributes of large programs, and abstract away functional details that are irrelevant, uninteresting, and/or cumbersome.
- A secondary reason is the need to specify non-deterministic behavior, such as the behavior of a program that uses a random data generator, or a program whose behavior depends on run-time information, or one user-provided data.

Definition 5: Given two (possibly non-deterministic) programs P and P' on space S and a specification R on S, we say that P' is *more-correct* than P with respect to R if and only if: $(R \cap P')L \cap (R \cup P') \sqsupseteq (R \cap P)L \cap (R \cup P)$.
Intuitive interpretation: This definition provides that P' is more-correct than P with respect to R if and only if P' has a larger competence domain with respect to R than P, and that, on the competence domain of P, P' never violates R unless P also does.

VI. Faults and Fault Repairs

A. What is a Fault?

If program repair is the art of fault removal, then it may be useful to ponder the question: what is a *fault* in a program? The first matter we must consider is that any definition of a fault refers to a level of granularity at which we want to isolate faults. The coarsest level of granularity is to consider the whole program as a possible fault, but that is clearly unhelpful as far as diagnosing and repairing faults; more typical levels of granularity include a line of code, a lexeme, an operator, an expression, an assignment statement, etc. 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 6: Due to [33]. Given a specification R and a program P, a *fault* in program P is any feature f that admits a substitute f' such that the program P' obtained from P by replacing f with f' is strictly more-correct than P. A *fault repair* in P is a pair of features (f, f') such that program P' obtained from P by replacing f with f' is strictly more-correct than P.
Examples of illustration can be found in [12, 33]. For the sake of completeness, we consider $\{$`skip`$\}$ (or, more generally, a blank lexeme) as part of our language, so that the insertion and deletion of features fall under this definition.

For comparison, consider that Avizienis et al. define a fault as the *adjudged or hypothesized cause of an error* [4]. Also, according to the IEEE Standard *IEEE Std 7-4.3.2-2003* [1], a software fault is "*an incorrect step, process or data definition in a computer program*". We argue that both of these definitions are too vague to support reasoning.

B. Implications

1) Elementary Faults: We consider space S defined by:
`x: float; i: int; a: float[0..N];`
for $N \geq 1$, and program P defined as follows:
`p:` $\{$`x=0;i=1;while (i<=N) {x=x+a[i];i=i+1}`$\}$.
We consider the following specification on S:

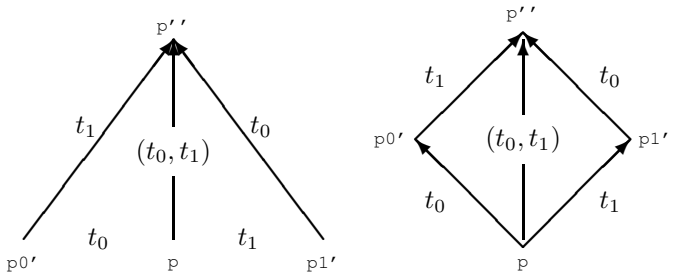


Fig. 2. One Two-site Fault vs. Two One-Site Faults

$R = \{(s, s') | x' = \sum_{k=0}^{N-1} a[k]\}$.
This program computes in x the sum of a from 1 to N while the specification mandates it to compute the sum of a from 0 to $(N-1)$. One way to correct this program is to change $\{$`i=1`$\}$ into $\{$`i=0`$\}$ and `(i<=N)` into `(i<N)`. This raises the question: do we have two single-site faults here or a single double-site fault? To answer this question, we consider the following programs:
`p0':` $\{$`x=0;i=0;while (i<=N) {x=x+a[i];i=i+1}`$\}$.
`p1':` $\{$`x=0;i=1;while (i<N) {x=x+a[i];i=i+1}`$\}$.
`p'':` $\{$`x=0;i=0;while (i<N) {x=x+a[i];i=i+1}`$\}$.
We let CD, CD'_0, CD'_1 and CD'' be the competence domains of `p`, `p0'`, `p1'`, and `p''`. We find: $CD = \{s | a[0] = a[N]\}$, $CD'_0 = \{s | a[N] = 0\}$, $CD'_1 = \{s | a[0] = 0\}$, $CD'' = S$. By comparing these competence domains, we find that `p''` is more-correct than all of `p`, `p0'`, and `p1'`, but the latter three are not mutually comparable. See Figure 2(a), where t_0 represents the transition from $\{$`i=1`$\}$ to $\{$`i=0`$\}$ and t_1 represents the transition from `(i<=N)` to `(i<N)`. Note that $\{$`i=1`$\}$ is a fault in `p1'`, but it is not a fault in `p`; likewise, `(i<=N)` is a fault in `p0'`, but it is not a fault in `p`; only the combination ($\{$`i=1`$\}$, `(i<=N)`) is a fault in `p`. We say that it is an elementary fault in `p` with respect to R. More generally we refer to a fault as an *elementary fault* if and only if it is either a single-site fault or a multi-site fault such that no part of it is a fault. Implicit in this discussion is the fact that when we talk about substituting a feature f by a feature f', we mean for f and f' to be of approximately the same scale/ granularity (the kind of substitution that a mutant generator would produce, for example). If it were not for this assumption, then one could replace $\{$`i=1`$\}$ by a segment that computes the sum of the array from 0 to $N-1$ and assign $N+1$ to i, so as to disable the subsequent loop.

We now consider the following example.
Space, S: `a: float [0..N]; i: int`
Specification $R =$
$\{(s, s') | (\forall k, 0 \leq k < N : a'[k] = 0) \wedge a[N] = a'[N]\}$.
`p:`$\{$`i=1; while(i<=N) {a[i]=0; i=i+1}`$\}$.
This program is incorrect because its competence domain ($CD = \{(s, s') | a[0] = 0 \wedge a[N] = 0\}$) is not equal to the domain of R (which is S); we consider the following mutants thereof:
`p0':`$\{$`i=0; while(i<=N) {a[i]=0; i=i+1}`$\}$.

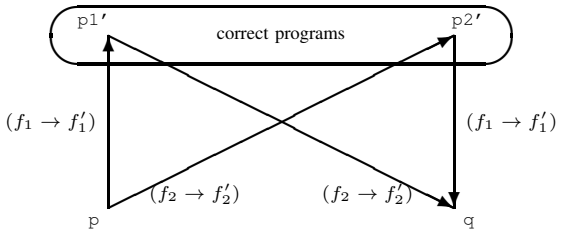


Fig. 3. Fault Density vs. Fault Depth: Density of p, q=2, Depth=1.

p1′:{i=1; while(i<N) {a[i]=0; i=i+1}}.
p′′:{i=0; while(i<N) {a[i]=0; i=i+1}}.

The competence domains of these programs are, respectively: $CD_0' = \{s|a[N] = 0\}$, $CD_1' = \{s|a[0] = 0\}$, $CD'' = S$. Figure 2(b) reflects the inclusion relationships that exist between the competence domains, hence the relative correctness relationships that exist between the candidate programs; we use t_0 and t_1 to designate the same transitions as the example above. The contrast between Figure 2(a) and Figure 2(b) illustrates the difference between a single multi-site (two-site, in this case) fault and and multiple (two, in this case) single-site faults.

2) Counting Faults, Measuring Faultiness: . We let the *fault density* of a program P be defined as the number of *elementary* faults in P. Also, we let the *fault depth* of a program P be the minimal number of elementary fault removals that separate P from a correct program. Whereas we want to think of these two quantities as identical (if we have N faults, we need N fault removals before the program is correct), not only are they usually different, but they have vastly different properties, due to the interaction between faults [8, 13]. Also, out of the two, we find that fault depth is a better measure of faultiness than fault density [12]. Consider the array sum program presented above, and consider that we have two faults therein:

- The two-site elementary fault that we have alluded to above, namely $f_1 = (\{\mathtt{i} = 1\}, (\mathtt{i} <= \mathtt{N}))$ which can be substituted by $f_1' = (\{\mathtt{i} = 0\}, (\mathtt{i} < \mathtt{N}))$.
- The single site fault $f_2 = (\mathtt{x} = \mathtt{x} + \mathtt{a}[\mathtt{i}])$, which can be substituted by $f_2' = (\mathtt{x} = \mathtt{x} + \mathtt{a}[\mathtt{i} - 1])$.

As one can easily check,

- If we replace f_1 by f_1' then f_2 is no longer a fault in the new program (which we call p1′).
- If we replace f_2 by f_2' in p then f_1 is no longer a fault in the new program (which we call p2′).
- If we replace f_1 by f_1' and f_2 by f_2' then we obtain a new program, say q, in which f_1' and f_2' are both faults.

In this example, program p has a fault density of (at least) two but a fault depth of 1; so does program q. Programs p1′ and p2′ are both correct; see Figure 3.

Whereas we want to think that fault density and fault depth decrease by one whenever we remove an elementary fault, neither property holds. Fault density can be very erratic (increase, decrease, stay unchanged),and fault depth satisfies the following inequality between a program P and a program P' obtained from P by an elementary fault removal:

$depth(P) \leq depth(P') + 1$.

Equality holds if P' is on a minimal path from P to a correct program.

VII. A Generic Algorithm for Program Repair

Given the definitions of absolute correctness and relative correctness, we argue that program repair methods ought to instantiate the following generic algorithm:

```
ProgramRepair(program P)
   {while not absolutelyCorrect(P)
      {P := StrictlyMoreCorrectThan(P);}}
```

The first step toward implementing this algorithm is to design oracles that test for absolute correctness, relative correct, and strict relative correctness.

A. Absolute Correctness

We consider a program P on space S and a specification R on S; and we consider the derivation of an oracle for absolute correctness of P with respect to R. To fix our ideas, we imagine the following test driver:

```
 bool absoracle(statetype s, sprime) {TBD;}
{statetype inits,s;  bool abscor=true;
 while (moretestdata())
    {inits =gettestdata();   //load test datum
     s = inits;p();   // modifies s, not inits
     abscor = abscor && absoracle(inits,s);}
 return abscor}
```

We must now define absoracle(), the oracle of absolute correctness. We propose the following formula:

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

The following Proposition, due to [34], justifies this formula.

Proposition 5: We consider a program P on space S and a specification R on S. We let T be a subset of S. If program P is tested against oracle $\Omega(s, s')$ and it is successful for all elements of T then it is correct with respect to the pre-restriction of R to T.

We write the oracle as follows, where domR(s) returns true if and only if s is in $dom(R)$ and R(s,s′) returns true if and only if (s, s') is in R:

```
bool absoracle(statetype s, sprime)
   {return (!domR(s) || R(s,sprime))}
```

Using the pointwise oracle $\Omega(s, s')$, we define the following predicate that represents the absolute correctness of P with respect to ${}_T\backslash R$: $\Omega_T(P) \equiv (\forall s \in T : \Omega(s, P(s))$.

B. Relative Correctness

The following oracle reflects the definition of relative correctness: $\omega(s, s') \equiv (\Omega(s, P(s)) \Rightarrow \Omega(s, s'))$.
The test driver that tests for relative correctness of P' over P with respect to R then looks as follows:

```
{statetype inits, s; bool relcor=true;
 while (mortestdata())
    {inits = gettestdata();// load test datum
     s = inits; p(); // modifies s, not inits
     bool abscor = absoracle(inits,s);
     s = inits; Pprime();
     relcor =relcor &&(!abscor ||
```

```
                          absoracle(inits,s));}
    return relcor}
```

Using the pointwise oracle $\omega(s, s')$, we define the following predicate that represents the relative correctness of P' over P with respect to $_T\backslash R$:

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

C. Strict Relative Correctness

Program P' is strictly more-correct than program P with respect to specification R if and only if P' is more-correct than P with respect to R and there exists at least one element in S such that P' satisfies the oracle of absolute correctness and P does not. This can be written as:

$\sigma_T(P') \equiv \omega_T(P') \wedge (\exists s \in S : \neg\Omega(s, P(s)) \wedge \Omega(s, P'(s)))$.

The test driver for strict relative correctness is written as:

```
{statetype inits, s;
 bool strict,relcor;strict=false;relcor=true;
 while (mortestdata())
   {inits = gettestdata();// load test datum
    s = inits; p(); // modifies s, not inits
    bool abscor = absoracle(inits,s);
    s = inits; Pprime();
    relcor =relcor &&(!abscor ||
                      absoracle(inits,s));
    strict =strict ||(!abscor &&
                       absoracle(inits,s))}
 return (strict && relcor)}
```

A criterion similar to strict relative correctness is used in the experiments reported by Smith et al. [41].

D. A Unitary Increment of Correctness

Using the oracles introduced above, we derive a function that takes a faulty program P, a specification R, and a test data set T, and attempts to return a program P' that is obtained from P by a single elementary fault repair. This algorithm is *generic* in the sense that it specifies how patches are evaluated, but does not specify how they are generated; hence it can be instantiated by selecting a patch generation policy. For the sake of argument, we adopt mutation as a patch generation policy; to account for the fact that some elementary faults may have more than one site, we may call the mutant generator more than once to repair a single elementary fault. Inputs to our algorithm include:

- The faulty program, P on space S.
- The specification R that the program is supposed to satisfy, as a binary predicate on S.
- The domain of R, as a unary predicate on S.
- The test data set T, a subset of S.
- The maximum fault multiplicity that the user wants to consider, say $MaxM$.

We assume that the patch generation tool avails us of two functions: `NextPatch(P, mult)`, which returns the next candidate repair of P obtained by performing `mult` mutations on P; `MorePatches(P, mult)`, a boolean function that returns True if and only if there are more patches of multiplicity `mult` to be considered. We propose the following algorithm:

```
programtype UnitIncCor(programtype P, int MaxM)
  {//increase correctness of P with patches of
   // multiplicity <= MaxM. R and T are global
  int mult=1; incremented=false;
  while (not incremented && mult<=MaxM)
    {//increase correctness with mult sites
    program Pp=P;
    while (not smc(Pp,P)&&MorePatches(P,mult))
      {//smc: Pp strictly more correct than P
      Pp = NextPatch(P,mult);}
      if smc(Pp,P) {incremented=true;}
      else {mult=mult+1;}}//try higher mult
  if incremented {return Pp} else {return P}}
```

This function performs a unitary increment of correctness enhancement by attempting to remove a single elementary fault of multiplicity no greater than $MaxM$.

E. Stepwise Ascent to Absolute Correctness

The algorithm we propose for program repair starts with a program of arbitrary fault depth, and invokes the unitary increment of correctness enhancement repeatedly until it reaches an absolutely correct program or it stalls, i.e. it fails to find a patch that enhances relative correctness.

```
void ProgramRepair(program P, specification R,
                           testdata T, int MaxM)
 {bool incremented=true;
  while (incremented && not abscor(P))
    {P=UnitIncCor(P, MaxM);}}
```

The output of this program depends, of course, on the performance of the patch generation machinery: according to the quality of patch generation, this will return an absolutely correct program (modulo the adequacy of T), a program that is more-correct than P but not absolutely correct, or just return P unchanged.

To estimate the *Big Oh* performance of this algorithm, we assume that the patch generation has a fixed fan-out for program P, say $F(P)$, when the multiplicity is 1; we also assume that for a greater multiplicity, say $mult$, the fan-out is $F(P)^{mult}$. If we designate the fault depth of program P by $\delta(P)$, then the performance of this algorithm is bounded by: $O(\delta(P) \times F(P)^{MaxM} \times |T|)$. Indeed, $\delta(P)$ is the number of elementary fault repairs that are needed to generate a correct program, $F(P)^{MaxM}$ is the maximal number of repair candidates that are evaluated for each fault repair, and $|T|$ is the number of tests that must be executed each time an oracle is called. If it were not for the use of `UnitIncCor()`, which performs a unitary increment of correctness, the Big Oh performance would be $F(P)^{\delta(P)}$; given that $\delta(P)$ is unbounded, this creates a search space of unbounded size.

VIII. Illustration: Proof of Concept

In this section, we report on an experiment that we ran to illustrate some of the discussions of this paper, most notably:

- The effectiveness of using the oracles of absolute correctness, relative correctness and strict relative correctness to select program repairs within a pool of candidates, with optimal precision and recall.
- The ability to transform an incorrect program into another incorrect, yet verifiably more-correct, program.

- The difference between fault depth and fault density: how they have different values, and how they evolve differently as faults are removed.
- The ability to repair programs of arbitrary depth, by an interative application of `UnitIncCor()`, which represents a unitary increment of correctness enhancement.
- The importance of using a large data set, that includes in particular a large set of negative tests, so as to monitor the growth in relative correctness.
- The importance of letting fault removal, rather than failure remediation, drive program repair; this allows us to remove faults as they are exposed, and enhancing the overall correctness of the program with each removal.

To this effect, we consider the following parameters:

- *Program*, P: We use `tcas`, a component of the Siemens benchmark of size 173 LOC, to which we apply 8 changes (provided as part of the benchmark).
- *Specification*, R: We use the original version of `tcas` as the specification that represents correct behavior; we refer to it as `TCAS`.
- *Test Data*, T: We use the test data set provided by the benchmark, of size 1608.
- *Maximum Fault Site Multiplicity*, $MaxM$: We set $MaxM$ to 2 but `UnitIncCor(tcas,2)` only used $mult = 1$ because all the faults in this experiment are single-site faults; in other experiments, reported in [22], with higher values of $MaxM$, `UnitIncCor()` increases $mult$ beyond 1 to repair multi-site faults.
- *Patch Generation*: The following operators are activated in the mutant generator: mutation of a logical operator; mutation of a relational operator. The number of mutants generated by each call is 87. For the purposes of our experiment, it is not important for our mutants to imitate realistic faults [3]; it suffices that the mutations satisfy our definition of fault removal.

With this experimental set-up in place, we start with the seeded version of `tcas` and apply `unitIncCor()` repeatedly to all the nodes that admit a strictly more-correct mutant until we reach absolutely correct nodes. The result is the graph depicted in Figure 4. Each node of this graph represents a mutant of `tcas`, and each arc represents a successful application of `UnitIncCor()`, i.e. an instance of fault removal. The bottom of the graph represents the faulty version of `tcas` and the top of the graph represents a mutant of the program that passes the oracle of absolute correctness for all the test data in T. The fault density of each node in this graph is the number of outgoing (going upwards) arcs, and the fault depth of each node is the shortest distance from that node to the top of the graph. We have conducted similar experiments with other components of the Siemens benchmark, whose results can be seen in [22]. Some noteworthy observations about this graph include:

- Even though it is routine to say, upon applying the eight changes to the original version of `tcas` that we have seeded eight faults, note that `tcas` (at the bottom of the graph) has a density of only four.
- Even though it is tempting to think that if we have N faults and we remove one, we are left with $(N-1)$ faults, the graph undermines this assumption: the bottom of the graph has a fault density of 4, and so do three out of the four mutants that are derived from it by fault repair.
- Even though it may be tempting to think that if we have N faults (density=N) then it takes N fault repairs to make the program correct (depth=N), the graph undermines this assumption as well: the node at the bottom of the graph has a fault density of 4 (four faults) but a fault depth of 7 (seven fault repairs before we reach the top of the graph). Note that neither the fault density nor the fault depth is equal to the number of modifications (eight) applied to the original program.
- Note that while fault density is not decremented by one at each fault repair, fault depth (in this special case) is; it turns out that all the paths from any node to the top of the graph are minimal paths.
- Even though this graph has a single node at the top (that satisfies the oracle of absolute correctness for all test data in T, i.e. is correct with respect to ${}_{T\setminus}R$); this needs not necessarily be the case; in [22], the graph for `replace` has two maximal nodes, one of which is the original (correct) version of the program.

This program was run on an *ASUS* laptop with a *Core i5 3317U* Intel processor featuring a clock speed of 1.7 GHz, a cache memory of 3 Mo, 6 Go of main memory and 500 Go of hard disk space. Upon generating all 133 fault repairs shown in Figure 4 (counted as the number of arcs in the graph), we pass this information to an open source graph drawing program, *Graphviz* (http://www.graphviz.org/), which produces this graph. The whole operation took one hour and 41 minutes.

Of course, what makes this operation successful is that patch generation is adapted to the faults, i.e. the mutation operators that we used are of the same nature and scale as the modifications applied to `tcas`. We have other experiments (not shown in [22]) where the process removes some of the faults of the seeded program, then stalls because neither of the top programs in the graph is absolutely correct, and neither admits a patch that is strictly more-correct; this arises whenever the patch generation is imperfect.

IX. Patch Generation by Correctness Enhancement

Whereas we have used relative correctness to perform patch validation, in this Section we briefly discuss how it can also be used for patch generation, in a way that makes patch validation unnecessary. In [15], titled *Debugging Without Testing*, Ghardallou et al present a method for fault repair that relies exclusively on static analysis of the source code. This method is applicable to iterative programs, and uses the concept of *invariant relation*, which is an approximation of the transitive closure of the guarded loop body function [30]. The method is based on a distinction between two categories

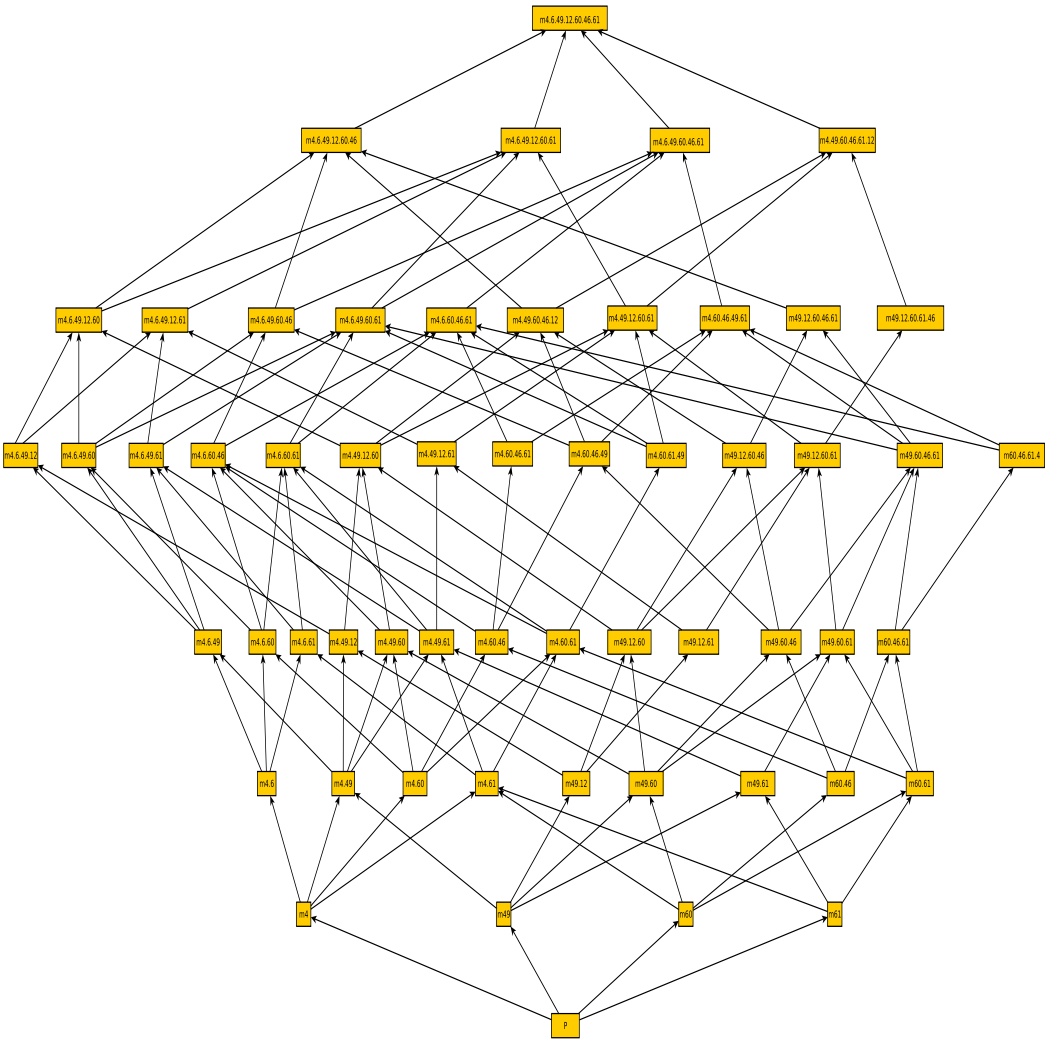


Fig. 4. Stepwise Fault Removal in `tcas`

of invariant relations, for any given specification: *compatible invariant relations*, which an iterative program may admit and still be correct with respect to the specification; and *incompatible invariant relations*, which characterize iterative programs that cannot possibly be correct with respect to the given specification.

Given an iterative program, we generate its invariant relations and separate them into two columns: compatible and incompatible. The existence of incompatible invariant relations is proof that the program is incorrect; we enhance its correctness by migrating invariant relations from the *incompatible* to the *compatible* column.

X. Conclusion: Summary, Assessment and Prospects

A. Summary

In this paper, we present some ingredients of a theory of program repair. This theory is based on the premise that to repair a program means to make it more correct; once we adopt this premise then it is incumbent on us to ensure that our definition of relative correctness is valid; we validate it by ensuring that it meets four properties that we expect from a definition of relative correctness.We have explored the short term and medium term applications of this theory, including:

- An elucidation of some distinctions that may improve the state of the practice in program repair: the distinction between a single multi-site fault and multiple single-site faults; the distinction between repairing a fault and remedying a failure; the distinction between preserving correctness and preserving correct behavior; the distinction between fault density and fault depth; the distinction between being more-correct and being more reliable; the distinction between fault mutiplicity and fault depth.
- An elucidation of a number of repair-relevant concepts, such as: what is a fault? what is fault repair? how do we test that a fault has been repaired? how do we quantify faultiness?
- A definition of test oracles that feature better precision

and recall thana existing technology.
- A definition of a unitary increment of correctness enhancement, which can be used as the building block of a program repair algorithm.
- The introduction of a criterion of correctness enhancement that generalizes traditional regression testing.
- Design of a generic program repair algorithm that relies on the availability of a patch generation method and deploys the infrastructure of test oracles cited above to select valid repairs. This algorithm proceeds by enhancing relative correctness until it achieves absolute correctness.
- Exploration of the potential of the theory of program repair to be used for patch generation, in addition to its straightforward use in patch validation.

B. Assessment and Threats to Validity

In the absence of a precise criterion for checking that a a program has been repaired, program repair methods have resorted to two broad families of devices:

- Using approximations of absolute correctness, by ensuring that repair candidates preserve correct behavior and behave correctly where the original program failed. But this criterion is a sufficient condition but a vastly unnecessary condition of relative correctness: First because the repair candidate may be correct without imitating the correct behavior of the original program; second, because absolute correctness is a sufficient but not a necessary condition of relative correctness.Using unnecessary conditions causes a loss of recall.
- Using fitness functions, which reward candidates for successful executions and penalize them for failing executions. These are approximations of reliability;enhanced reliability is a necessary condition of relative correctness, but not a sufficient condition. Using insufficient conditions causes a loss of precision.

Using an oracle of absolute correctness for patch validation raise another set of issues:

- If patch generation can only fix one fault at a time, then the method can only be applied to programs whose fault depth is 1.
- If patch generation can fix several faults simultaneously, then we have to consider all the combinations of faults that produce an absolutely correct program, a receipe for runaway combinatorial explosion.

With the oracle of strict relative correctness, we can (contingent upon adequate patch generation) tackle a program of arbitrary fault depth, by removing one elementary fault at a time, at a cost that is linear (rather than exponential) in the fault depth of the program. This is illustrated in the experiment of Section VIII; see also Figure 4.

Program repair methods and tools typically take as input a faulty program and a specification in the form of positive test data that must be preserved and negative test data that must be remedied. Very often, the negative test data consists of a single input/output pair, representing an observation of failure. While this choice may be viewed as a sensible divide-and-conquer strategy (resolving one failure at a time), we argue that it is actually counter-productive: First, because the divide-and-conquer argument is valid only if each failure can be traced to a single fault, but that is clearly untrue; second, because having a single (or too few) failure point(s) robs us of information that would enable us to monitor the progress of our repair effort; third, we speculate that this approach may be responsible for the tendency of repair methods to overfitting repairs to specific failures and not doing much good to the overall (relative) correctness of candidate repairs. See Figure 5. When we resolve to remedy a particular failure (T^-), we commit to find all the faults that may be causing this failure and repairing them. But when we focus on repairing faults, we let the program expose its faults one by one, in the order it determines, and we expand the competence domain of the program until the observed failure is remedied (until the competence domain encompasses T^-).

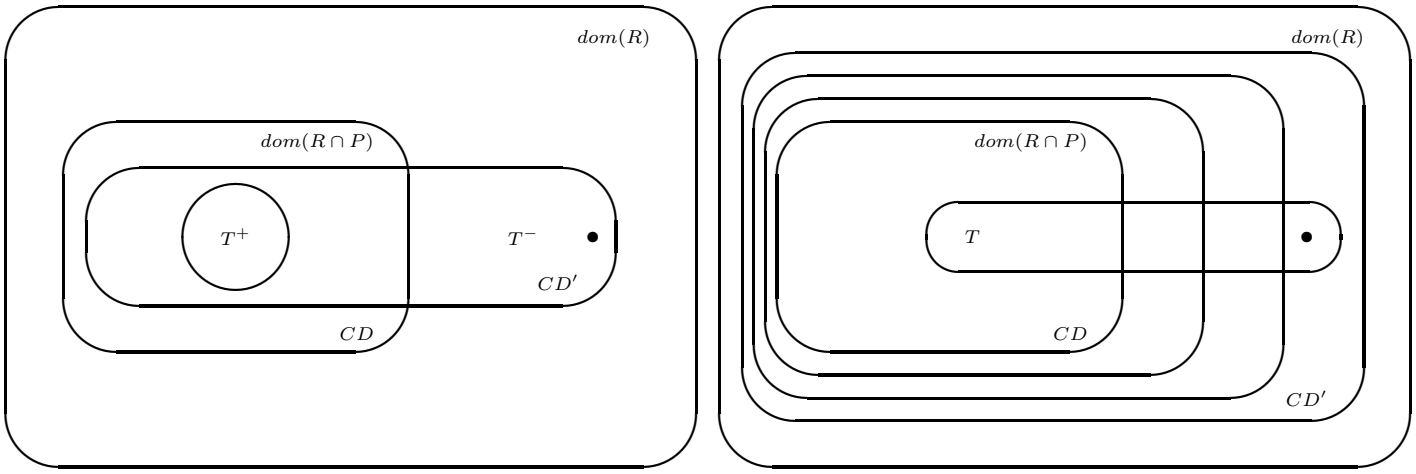
The two main threats that we see to the validity of our work are: scalability, and the requirement to provide a specification. We consider them in turn, below:

- *Scalability*. Relative correctness scales as much as (or as little as) absolute correctness; and we argue that it ought to play for program repair the same role that absolute correctness plays for program construction; it serves as a reference model, as a yardstick for assessing methods and tools, and as a last resort when the stakes warrant the effort to deploy it. Consider that the definition of relative correctness for non-deterministic programs (Definition 5), though it is not discussed in this paper, plays an important role in supporting scalability: We can determine that one program is more-correct than another without having to compute their functions in detail; it suffices to capture the behavior of these programs by non-deterministic relations.
- *The Requirement to Provide a Specification*. It is impossible to claim that a program is correct unless we have a reference specification; likewise, it is impossible to claim that a program is more-correct than another without referring to a specification. When an engineer using a program repair tool provides a set of positive test data and a set of negative test data, the engineer is doing so in reference to an implicit specification; all we require is to make this specification explicit.

C. Prospects

Our short to medium term goal is to apply the generic algorithm of program repair presented in Section VII in combination with a number of existing patch generation tools to see whether and to what extent we streamline their performance; this is currently under way.

Our medium to long term goal is to explore means to perform program repair by pure static semantic analysis, in the style of the example of Section IX, but in broader contexts (with weaker restrictions on program structure).



(a): Focus on Failure Remediation, Single Negative Test

(b): Focus on Fault Repair, Large Test Data

Fig. 5. Fault Repair vs. Failure Remediation

Acknowledgements

The authors are very grateful to the anonymous reviewers for their thoughtful, insightful comments, which have greatly improved the substance and presentation of this paper.

REFERENCES

[1] IEEE Std 7-4.3.2-2003. Ieee standard criteria for digital computers in safety systems of nuclear power generating stations. Technical report, The Institute of Electrical and Electronics Engineers, 2003.

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

[3] J.H. Andrews, L.C. Briand, and Y. Labiche. Is mutation an appropriate tool for testing experiments? In *Proceedings, ICSE*, 2005.

[4] Algirdas Avizienis, Jean Claude Laprie, Brian Randell, and Carl E Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable and Secure Computing*, 1(1):11–33, 2004.

[5] R.J. Back and J. von Wright. *Refinement Calculus: A Systematic Introduction*. Graduate Texts in Computer Science. Springer Verlag, 1998.

[6] Chris Brink, Wolfram Kahl, and Gunther Schmidt. *Relational Methods in Computer Science*. Advances in Computer Science. Springer Verlag, Berlin, Germany, 1997.

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

[8] Vidroha Debroy and W. Eric Wong. Insights on fault interference for programs with multiple bugs. In *Proceedings, 20th International Symposium on Software Reliability Engineering*, Mysuru, India, November 2009.

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

[10] 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.

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

[12] Nafi Diallo, Wided Ghardallou, Jules Desharnais, Marcelo Frias, Ali Jaoua, and Ali Mili. What is a fault? and why does it matter? *ISSE*, 19:219–239, 2017.

[13] Nicholas DiGiuseppe and James A. Jones. Fault interaction and its repercussions. In *27th IEEE Conference on Software Maintenance*, September 2011.

[14] R.W. Floyd. Assigning meaning to programs. *Proceedings of the American Mathematical Society Symposium in Applied mathematics*, 19:19–31, 1967.

[15] Wided Ghardallou, Nafi Diallo, Ali Mili, and Marcelo Frias. Debugging without testing. In *Proceedings, International Conference on Software Testing*, Chicago, IL, April 2016.

[16] 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.

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

[18] Rahul Gupta, Soham Pal, Aditya Kanade, and Shirish K. Shevade. Deepfix: Fixing common c language errors by deep learning. In *Proceedings, AAAI 2017*, pages 1345–1351, 2017.

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

[20] C.A.R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–583, October 1969.

[21] Y. Ke, K. T. Stolee, C. Le Goues, , and Y. Brun. Repairing programs with semantic code search. In *International Conference on Automated Software Engineering*, 2015.

[22] Besma Khaireddine, Aleksandr Zakharchenko, Matias Martinez, and Ali Mili. Addendum to: Toward a theory of program repair. Technical report, NJIT, Newark, NJ, available at http://web.njit.edu/~mili/oracledesign.pdf, 2016.

[23] D. Kim, J. Nam, J. Song, and S. Kim. Automatic patch generation learned from human-written patches. In *International Conference on Software Engineering (ICSE)*, pages 802–811, 2013.

[24] Xuan-Bach D. Le, Duc-Hiep Chu, David Lo, Claire Le Goues, and Willem Visser. S3: Syntax and semantic guided repair synthesis via programming examples. In *Proceedings, FSE 2017*, Paderborn, Germany, September 4-8 2017.

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

[26] 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.

[27] F. Long and M. Rinard. Prophet: Automatic patch generation via learning from successful patches. Technical Report Technical Report MIT-CSAIL-TR-2015, MIT, 2015.

[28] F. Long and M. Rinard. Staged program repair with condition synthesis. In *Proceedings, ESEC-FSE*, 2015.

[29] F. Long and M. Rinard. An analysis of the search spaces for generate-and-validate patch generation systems. In *ICSE 2016*, 2016.

[30] Asma Louhichi, Wided Ghardallou, Khaled Bsaies, Lamia Labed Jilani, Olfa Mraihi, and Ali Mili. Verifying loops with invariant relations. *International Journal of Critical Computer Based Systems*, 5(1/2):78–102, 2014.

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

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

[33] 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.

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

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

[36] 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.

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

[38] John D. Musa. Operational profile in software reliability engineering. *IEEE Software*, 10(2):14–32, 1993.

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

[40] 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.

[41] Edward K. Smith, Earl T. Barr, Claire Le Goues, and Turyi Brun. Is the cure worse than the disease: Overfitting in automated program repair. In *Proceedings, ESEC/ FSE 2015*, Bergamo, Italy, August 30- September 4 2015.

[42] Mauricio Soto and Claire Le Goues. Using a probabilistic model to predict bug fixes. In *Proceedings, SANER 2018*, pages 221–231, 2018.

[43] S. H. Tan and A. Roychoudhury. Relifix: Automated repair of software regressions. In *ICSE*, 2015.

[44] W. Weimer, T. Nguyen, C. Le Goues, and S. Forrest. Automatically finding patches using genetic programming. In *Proceedings, International Conference on Software Engineering (ICSE)*, pages 364–374, 2009.

[45] Wing Wen, JunJie Chen, Rongxin Wu, Dan Hao, and Shing Chi Cheung. Context-aware patch generation for better automated program repair. In *Proceedings, ICSE 2018*, Gothenburg, Sweden, May 27-June 3 2018.

[46] Qi Xin and Steven P. Reiss. Leveraging syntax-related code for automated program repair. In *Proceedings, ASE 2017*, Urbana Champaign, IL, October 30-November 3 2017.

[47] J. Xuan, M. Martinez, F. Demarco, M. Clement, S. Lamelas Marcotte, T. Durieux, D. LeBerre, and M. Monperrus. Nopol: Automatic repair of conditional statement bugs in java programs. *IEEE-TSE*, 2016.

Besma Khaireddine received the BS and MS degrees from the University of Monastir and is currently a PhD student at the University of Tunis El Manar. She is interested in software testing and verification.

Aleksandr Zakharchenko holds a Bachelor of Science from Strayer University and a Master of Science in Computer Science from NJIT. He is a PhD candidate at NJIT; his research interests are in program repair and software tools.

Matias Martinez is an assistant professor at the University of Valenciennes. He holds a diploma in CS from UNICEN (Argentina) and a PhD from the University of Lille. His research interests are in software evolution, automated software repair, software testing and mobile applications.

Ali Mili holds a PhD from the University of Illinois, Urbana, and a Doctorat es-Sciences d'Etat from the Joseph Fourier University of Grenoble. He is Professor and Associate Dean of NJIT's Ying Wu College of Computing.