Program Repair by Stepwise Correctness Enhancement

Nafi Diallo
CCS, NJIT, Newark NJ USA
ncd8@njit.edu

Wided Ghardallou
FST, UTM, Tunis, Tunisia
wided.ghardallou@gmail.com

Ali Mili
CCS, NJIT, Newark, NJ USA
mili@njit.edu

Relative correctness is the property of a program to be more-correct than another with respect to a given specification. Whereas the traditional definition of (absolute) correctness divides candidate program into two classes (correct, and incorrect), relative correctness arranges candidate programs on the richer structure of a partial ordering. In other venues we discuss the impact of relative correctness on program derivation, and on program verification. In this paper, we discuss the impact of relative correctness on program testing; specifically, we argue that when we remove a fault from a program, we ought to test the new program for relative correctness over the old program, rather than for absolute correctness. We present analytical arguments to support our position, as well as an empirical argument in the form of a small program whose faults are removed in a stepwise manner as its relative correctness rises with each fault removal until we obtain a correct program.

Keywords

Program correctness, Relative correctness, Absolute correctness, Program repair.

1 Relative Correctness and Quality Assurance Methods

Relative correctness is the property of a program to be more-correct than another with respect to a given specification. Intuitively, P' is more-correct than P with respect to specification R if and only if P' obeys R more often (for a larger set of inputs) than P, and violates R less egregiously (in fewer ways) than P.

Traditionally, we distinguish between two categories of candidate programs for a given specification: correct programs, and incorrect programs; but the introduction of relative correctness enables us to generalize this binary classification into a richer structure that ranks all candidate programs by means of a partial ordering whose maximal elements are (absolutely) correct.

Also, in our quest for enhancing program quality, we have traditionally used static analysis methods and dynamic testing methods for distinct purposes:

- Program verification methods are applied to correct programs to ascertain their correctness; they are of little use when applied to incorrect programs, because even when a proof fails, we cannot conclude that the program is incorrect (the proof may have failed because the documentation of the program in terms of intermediate assertions and invariant assertions is inadequate).
- Program testing methods are applied to incorrect programs to expose their faults and remove them; but they are of little use when applied to correct programs, since they cannot be used to prove the absence of faults.

Here again, we argue that relative correctness can act as a disruptive concept, since it blurs this neat separation of duties. In [7], we present a relative correctness-based static analysis method that enables us to locate and remove a fault from a program, and prove that the fault has been removed —all without testing. This technique, which we call *debugging without testing*, shows that we can apply static analysis

Submitted to:
PrePost 2016

to an incorrect program to prove that, although it may be incorrect, it is still more correct than another. Given that there are orders of magnitude more incorrect programs than there are correct programs, the pursuit of this idea may expand the scope of static analysis methods.

In [5], we discuss how relative correctness can be used in the derivation of a correct program from a specification. Whereas traditional programming calculi derive programs from specifications by successive refinement-based correctness-preserving transformations starting from the specification, we show that we can derive a program by successive correctness-enhancing transformations (using relative correctness) starting from the trivial program `abort`. We refer to this technique as *programming without refining* [5].

In this paper, we explore the use of relative correctness in program repair. Specifically, we discuss how to perform program repair when we test candidate mutants for relative correctness rather than absolute correctness. We are not offering a viable, validated, empirically supported solution; rather, we are merely analyzing current practice, discussing why we believe a relative-correctness-based approach may offer better outcomes, and supporting our case with analytical arguments as well as a simple illustrative example.

In section 3 we define relative correctness and explore its main properties; since our definitions and discussions rely on relational calculi, we devote section 2 to a brief discussion of relational concepts. In section 4 we critique the current practice of program repair, which is based on a test of absolute correctness, and argue, on the basis of analytical arguments, that using a test of relative (rather than absolute) correctness leads to better outcomes. We complement the analytical argument of section 4 by an empirical illustration in section 5 in the form of a faulty program, which we repair in a stepwise manner by removing its faults one by one, making it increasingly more-correct until it becomes absolutely correct. We summarize and assess our findings in section 6, and we briefly sketch directions for future research.

2 Relational Mathematics

We assume the reader familiar with relational algebra, and we generally adhere to the definitions of [18][2]. Dealing with programs, we represent sets using a programming-like notation, by introducing variable names and associated data types. If, e.g. we define set S by the variable declarations

$x : X; y : Y; z : Z$,

then S is the Cartesian product $X \times Y \times Z$. Elements of S are denoted by s, and are triplets of elements of X, Y, and Z. Given s in S, we represent its X-component (resp. Y-component, Z-component) by (resp.) $x(s)$, $y(s)$, $z(s)$. When no risk of ambiguity exists, we may write x to represent $x(s)$, and x' to represent $x(s')$. A relation on S is a subset of the Cartesian product $S \times S$. Special relations on S include the *universal* relation $L = S \times S$, the *identity* relation $I = \{(s,s')|s' = s\}$, and the *empty* relation $\phi = \{\}$. Operations on relations (say, R and R') include the set theoretic operations of *union* ($R \cup R'$), *intersection* ($R \cap R'$), *difference* ($R \setminus R'$) and *complement* ($\overline{R}$). They also include the *relational product*, denoted by ($R \circ R'$), or (RR', for short) and defined by:

$$RR' = \{(s,s')|\exists s'' : (s,s'') \in R \wedge (s'',s') \in R'\}.$$

The *power* of relation R is denoted by R^n, for a natural number n, and defined by $R^0 = I$, and for $n > 0$, $R^n = R \circ R^{n-1}$. The *reflexive transitive closure* of relation R is denoted by R^* and defined by $R^* = \{(s,s')|\exists n \geq 0 : (s,s') \in R^n\}$. The *converse* of relation R is the relation denoted by $\widehat{R}$ and defined by $\widehat{R} = \{(s,s')|(s',s) \in R\}$. The *domain* of a relation R is defined as the set $dom(R) = \{s|\exists s' : (s,s') \in R\}$, and the *range* of relation R is defined as the domain of $\widehat{R}$. Note that given a relation R, the product of

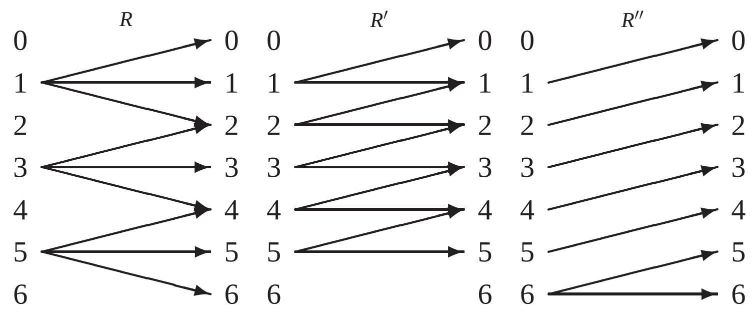

Figure 1: Refinement: $R' \sqsupseteq R$, $R'' \sqsupseteq R'$

R by L represents the relation $dom(R) \times S$; hence this expression is in effect a relational representation of the domain of R; we may sometimes use $dom(R)$ and RL interchangeably to refer to the domain of R. Operator precedence is adopted as follows: unary operators apply first, followed by product, then intersection, then union.

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 is said to be a *partial ordering* if and only if it is reflexive, antisymmetric, and transitive. Also, a relation R is said to be *total* if and only if $I \subseteq R\widehat{R}$, and a relation R is said to be *deterministic* (or: a *function*) if and only if $\widehat{R}R \subseteq I$.

We use relations to represent specifications or programs. A key concept in any study of program correctness is the refinement ordering; the following definition lays out our version of this ordering.

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, a relation R' refines a relation R if and only if it has a larger domain and assigns fewer images than R to elements of the domain of R. See Figure 1, where R'' refines R', which in turns refines R.

3 Absolute Correctness and Relative Correctness

3.1 Program Functions

If a program p manipulates variables, say $x : X$ and $y : Y$, we say that set $S = X \times Y$ is the *space* of p and we refer to elements of S as *states* of p. Given a program p on space S, we denote by $[p]$ the function that p defines on its space, i.e.

$[p] = \{(s, s')|$if program p executes on state s then it terminates in state $s'\}$.

We represent programs by means of C-like programming constructs, which we present below along with their semantic definitions:

- *Abort:* $[\texttt{abort}] \equiv \phi$.
- *Skip:* $[\texttt{skip}] \equiv I$.
- *Assignment:* $[s = E(s)] \equiv \{(s, s')|s \in \delta(E) \wedge s' = E(s)\}$, where $\delta(E)$ is the set of states for which expression E can be evaluated.
- *Sequence:* $[p_1; p_2] \equiv [p_1] \circ [p_2]$.
- *Conditional:* $[\texttt{if}\ (t)\ \{p\}] \equiv T \cap [p] \cup \overline{T} \cap I$, where T is the relation defined as: $T = \{(s, s')|t(s)\}$.

- *Alternation:* $[\texttt{if}\ (t)\ \{p\}\ else\ \{q\}] \equiv T \cap [p] \cup \overline{T} \cap [q]$, where T is defined as above.
- *Iteration:* $[\texttt{while}\ (t)\ \{b\}] \equiv (T \cap [b])^* \cap \widehat{\overline{T}}$, where T is defined as above.
- *Block:* $[\{x : X; p\}] \equiv \{(s,s')|\exists x,x' \in X : (\langle s,x\rangle,\langle s',x'\rangle) \in [p]\}$.

We will usually use upper case P as a shorthand for $[p]$. By abuse of notation, we may refer to a program and its function by the same name.

3.2 Absolute Correctness

Definition 2 *Let p be a program on space S and let R be a specification on S. We say that program p is* correct *with respect to R if and only if P refines R. We say that program p is* partially correct *with respect to specification R if and only if P refines $R \cap PL$.*

This definition is consistent with traditional definitions of partial and total correctness [10][11][6]. Whenever we want to contrast correctness with partial correctness, we may refer to it as *total correctness*. The following proposition, due to [17], gives a simple characterization of correctness for deterministic programs.

Proposition 1 *Program p is correct with respect to specification R if and only if $(R \cap P)L = RL$.*

By construction, $(R \cap P)L$ is a subset of RL, and correct programs are those that reach the maximum of $(R \cap P)L$, which is RL.

3.3 Relative Correctness: Deterministic Programs

Definition 3 *Let R be a specification on space S and let p and p' be two deterministic programs on space S whose functions are respectively P and P'. We say that program p' is* more-correct *than program p with respect to specification R (abbrev: $P' \sqsupseteq_R P$) if and only if: $(R \cap P')L \supseteq (R \cap P)L$. Also, we say that program p' is* strictly more-correct *than program p with respect to specification R (abbrev: $P' \sqsupset_R P$) if and only if $(R \cap P')L \supset (R \cap P)L$.*

Interpretation: $(R \cap P)L$ represents (in relational form) the set of initial states on which the behavior of P satisfies specification R. We refer to this set as the *competence domain* of program P with respect to specification R. For deterministic programs p and p', relative correctness of p' over p with respect to specification R simply means that p' has a larger competence domain than p. Whenever we want to contrast correctness with relative correctness, we refer to it as *absolute correctness*. Note that when we say *more-correct* we really mean *more-correct or as-correct-as*. Note also that program p' may be more-correct than program p without duplicating the behavior of p over the competence domain of p; see Figure 2. In the example shown in this figure, we have:
$(R \cap P)L = \{1,2,3,4\} \times S$,
$(R \cap P')L = \{1,2,3,4,5\} \times S$,
where $S = \{0,1,2,3,4,5,6\}$. Hence p' is more-correct than p with respect to R.

In order to highlight the contrast between relative correctness (as a partial ordering) and absolute correctness (as a binary attribute), 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 programs, along with their competence domains:

- $P_0 = \{(a,d),(b,a)\}$. $CD_0 = \{\}$.
- $P_1 = \{(a,b),(b,e)\}$. $CD_1 = \{a\}$.

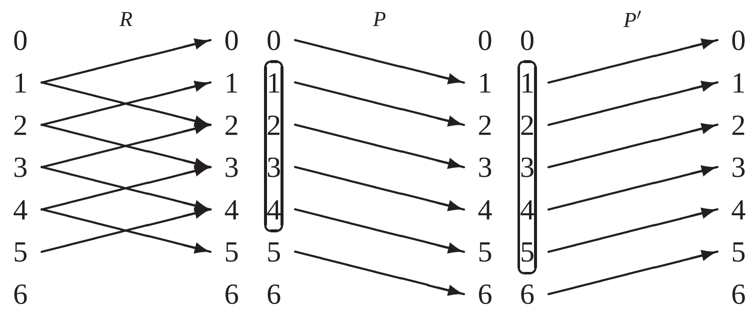

Figure 2: Enhancing correctness without duplicating behavior: $P' \sqsupseteq_R P$

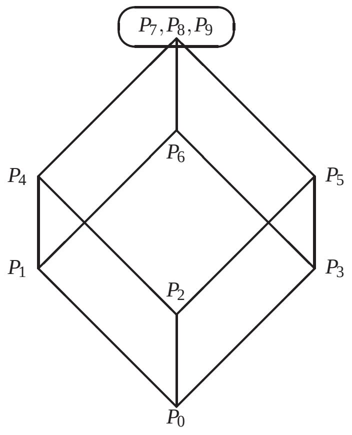

Figure 3: Ordering Candidate Programs by Relative Correctness

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

Figure 3 shows how these programs are ordered by relative correctness with respect to R; in this sample, 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 because their competence domain are different from (smaller than) the domain of R ($\{a,b,c\}$). See Figure 8 for a more concrete example of programs ordered by relative correctness.

3.4 Relative Correctness: Non-Deterministic Programs

We let S be the space defined by non-zero natural variables x and y and we let R be the following specification on R: $R = \{(s,s')|x' \geq 3x^2\}$. Let p and p' be the following programs:

```
    p:  {x=13*x; y=y+2*x;},
    p':  {x=19*x; y=y+7*x;}.
```

The functions of these programs are:

$P = \{(s,s')|x' = 13\times x \wedge y' = y + 26\times x\}$

$P' = \{(s,s')|x' = 19\times x \wedge y' = y + 133\times x\}$.

We are interested to analyze the relative correctness of p and p' with respect to R; to this effect, according to definition 3, we must analyze the functions of p and p'. But the effect of these programs on variable y could not possibly be relevant to this analysis since R does not refer to y. Hence it ought to be possible for us to analyze the relative correctness of p and p' with respect to R by considering the effect of p and p' on x alone; this yields the following relations:

$\pi = \{(s,s')|x' = 13\times x\}$,

$\pi' = \{(s,s')|x' = 19\times x\}$.

Yet, we cannot apply Definition 3 to π and π' because they are not deterministic (since they fail to specify a final value for variable y). In [4] we present a definition of relative correctness that generalizes Definition 3 and applies to (possibly) non-deterministic programs; referring to the definition given in [4], we find (by inspecting π and π' rather than P and P') that p' is more-correct than p with respect to R. Indeed, the competence domain of program p with respect to R (i.e. the set of initial states for which p behaves according to R) is characterized by the equation $13\times x \geq 3\times x^2$, which is equivalent (since x is a natural variable) to $x \leq 4$. Likewise, we find that the competence domain of p' is characterized by the equation $x \leq 6$.

4 Program Repair by Relative Correctness

4.1 Faults and Fault Removal

Now that we know what it means for a program to be more correct than another, we are in a position to define what is a fault, and under what condition we can say that we have removed a fault. Any definition of a fault assumes, implicitly, some level of granularity at which we want to define faults; a typical level of granularity for C-like languages is the single assignment statement, while finer grained features include expressions or operators within expressions. We use the term *feature* to refer to any program part, or set of program parts, at the selected level of granularity and we present the following definition, due to [16].

Definition 4 *Given a program p on space S and a specification R on S, we say that a feature f of p is a* fault *in p with respect to R if and only if there exists a feature f' such that program p' obtained from program p when we replace f by f' is strictly more-correct than p with respect to R.*

When such an f' is found, the pair (f, f') is called a fault removal *in p with respect to R.*

As an illustrative example, we consider the following program p on space S defined by variables a, x and i declared therein:

```
p:  int main () {int a[N+1]; int x=0;
            int i=0; while (i<N) {x=x+a[i]; i=i+1;}}
```

and we consider the specification R defined by:

$R = \{(s,s')|x' = \sum_{i=1}^{N} a[i]\}$.

The function of program p is:

$P = \{(s,s')|a' = a \wedge i' = N \wedge x' = \sum_{i=0}^{N-1} a[i]\}$.

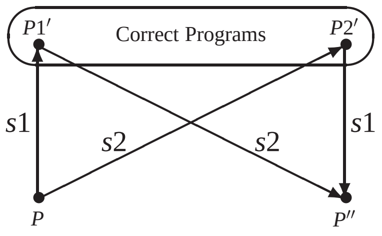

Figure 4: Both P and P'': fault density of 2; fault depth of 1.

The competence domain of p with respect to R is:
$(R \cap P)L = \{(s,s')|a[0] = a[N]\}$,
which makes sense, since this is the condition under which what the program does (the sum of a from 0 to $N-1$) coincides with what the specification mandates (the sum from 1 to N). Because the competence domain of p is distinct from $dom(R) = S$, this program is incorrect. At the level of granularity of assignment statements and logical expressions, we see two faults in program p with respect to specification R:

- The fault made up of the aggregate of statements $f1$ = `(i=0, (i<N))`; the substitution $f1'$ = `(i=1, (i<N+1))` constitutes a fault removal for $f1$.
- The fault $f2$ = `(x=x+a[i])`; substitution of $f2$ by $f2'$ = `(x=x+a[i+1])` constitutes a fault removal for $f2$.

Note that `(i=0)` alone is not a fault in p, nor is `(i<N)` as they admit no substitution that would make the program more-correct. If we let $p1'$ be the program obtained from p by substituting $f1$ by $f1'$, then we find that $f2$ is not a fault in p', even though it is a fault in p; the same goes for program $p2'$ obtained by substituting $f2'$ for $f2$. If we substitute $f1$ by $f1'$ and $f2$ by $f2'$ we find two faults again, namely $f1'$ and $f2'$. See Figure 4, where we label each transformation by the corresponding substitution, and we let $s1$ be the substitution $(f1, f1')$ and $s2$ be the substitution $(f2, f2')$. Even though p has two faults, it is one fault removal away from being correct; we say that it has a *fault density* of 2 and a *fault depth* of 1.

4.2 Testing for Relative Correctness

Now that we have defined what is a fault removal, we addres the question: how do we ascertain that we have removed a fault? As is customary for matters pertaining to software products, we can do so in one of two ways:

- Either through a static analysis of the products' (p and p') source code; this is discussed in [7].
- Or through execution and monitoring of the products in question; this is the subject of this paper.

This raises the question: how do we test a program p' for relative correctness over some program p with respect to specification R, and how is that different from testing program p' for absolute correctness with respect to R? For the sake of simplicity, we address this question in the context of deterministic programs, and we argue that testing a program p' for relative correctness over some program p with respect to some specification R differs from testing program p' for absolute correctness with respect to R in three important ways:

- *Test Data Selection*. The problem of test data selection can be formulated in the following generic terms: Given a large or infinite input space, say D, find a representative subset T of D such that analysis of the behavior of candidate programs on T enables us to infer claims about their behavior on D. Regardless of what selection criterion is adopted to derive T from D, testing for relative correctness differs from testing for absolute correctness in a fundamental way: for absolute correctness, the input space D we are trying to approximate is $D = dom(R)$, whereas for relative correctness the input space is $D = dom(R \cap P)$, i.e. the competence domain of p. Indeed, to prove that p' is more-correct than p with respect to R, we must prove that p' runs successfully for all elements of the competence domain D of p, which we do by checking that p' runs successfully for all elements of T (an approximation of D).

- *Oracle Design*. Let $\Omega(s,s')$ be the oracle for absolute correctness derived from specification R. Then the oracle for relative correctness of program p' over program p, which we denote by $\omega(s,s')$ must ensure that program p' satisfies $\Omega(s,s')$ for all s in the competence domain of p with respect to R. We write it as:
$$\omega(s,s') \equiv (\Omega(s,P(s)) \Rightarrow \Omega(s,s')).$$
As for oracle $\Omega(s,s')$ it must be derived from specification R according to the following formula:
$$\Omega(s,s') \equiv (s \in dom(R) \Rightarrow (s,s') \in R).$$
Indeed, we do not want a test to fail on some input s outside the domain of R, as candidate programs are only responsible for their behavior on $dom(R)$. Hence the condition $(s,s') \in R$ is checked only for s in $dom(R)$; for s outside the domain of R, the test is (vacuously) considered successful.

- *Test Coverage Assessment*. When we test a program p' for absolute correctness with respect to some specification R using a test data of size N, we gain a level of confidence in the correctness of p', to an extent that is commensurate with N. On the other hand, when we test a program p' for relative correctness over a program p with respect to some specification R on a test data of size N, N does not tell the whole story: We also need to know whether p' behaves better than p because p fails often or because p' succeeds often. Hence we may need to quantify the outcome of the experiment by means of three variables: N_0, the number of test cases when both p and p' succeed; N_1, the number of test cases when p fails and p' succeeds; and N_2, the number of test cases when both fail. While $N = N_0 + N_1 + N_2$ tells us to what extent p' is better than p (i.e. to what extent we can be confident that p' is more-correct than p), the partition of N into N_0, N_1 and N_2 tells us whether p' is better than p because p' succeeds often or because p fails often. See Figure 5.

4.3 Program Repair with Absolute Correctness

Most techniques for program repair [1][19][8][3][9][20] proceed by applying transformations on an original faulty program. These transformations may be macro-transformations (including multi-site program modifications), or micro-modifications (intra-statement) using mutation operators such as those provided by the muJava [15] program mutation tool. Two main approaches exist towards assessing the suitability of the generated transformations: test-based techniques [1][19][3][9] (which use the successful execution of the candidate program on a test suite as the acceptance criterion) or specification-based techniques [8][20] (which use a specification and some sort of constraint-solving to determine if the new code complies with the specification). In both cases, mutants are selected on the basis of an analysis of their absolute correctness with respect to the specification at hand (embodied in the oracle in the case of testing).

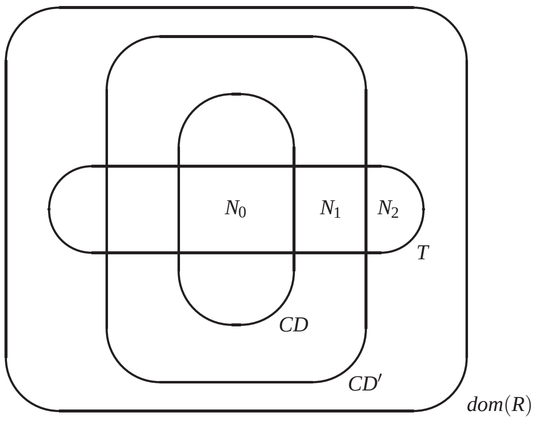

Figure 5: Test Coverage of Relative Correctness: $N = N_0 + N_1 + N_2$

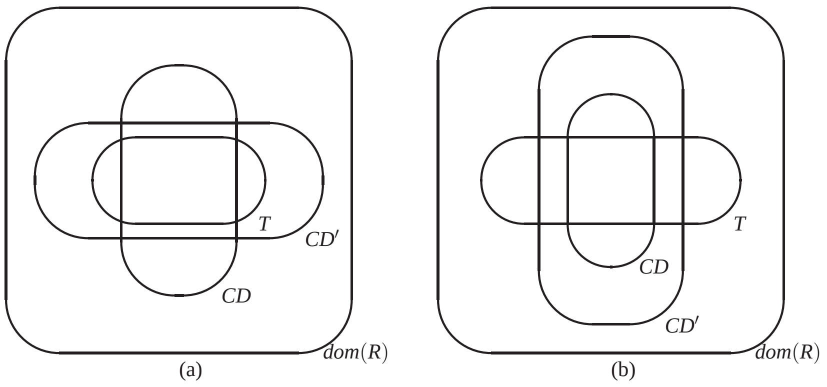

Figure 6: Absolute Correctness-based Repair: Poor Precision, Poor Recall

We argue that selecting mutants on the basis of their absolute correctness is flawed, because when we remove a fault from the original program, we have no reason to believe that the new program is correct, unless we assume that the fault that we have just removed is the program's last fault. Instead, the best we can hope for when we generate a mutant from a base program is that the mutant is more-correct than the base program with respect to the specification at hand; consequently, we should be testing mutants for relative correctness rather than absolute correctness. Specifically, we argue that when mutants are evaluated on the basis of their absolute correctness on a sample test data T, both the decision to retain successful mutants and the decision to reject unsuccessful mutants, are wrong:

- As Figure 6(a) shows (if CD is the competence domain of the original program and CD' is the competence domain of the mutant), a mutant may pass the test T (since $T \subseteq CD'$) yet not be more-correct than the original (since CD is not a subset of CD').
- As Figure 6 (b) shows, a mutant may fail the test T (since T is not a subset of CD') and yet still be more-correct than the original (since $CD \subseteq CD'$).

As a result, neither the precision nor the recall of the selection algorithm is assured.

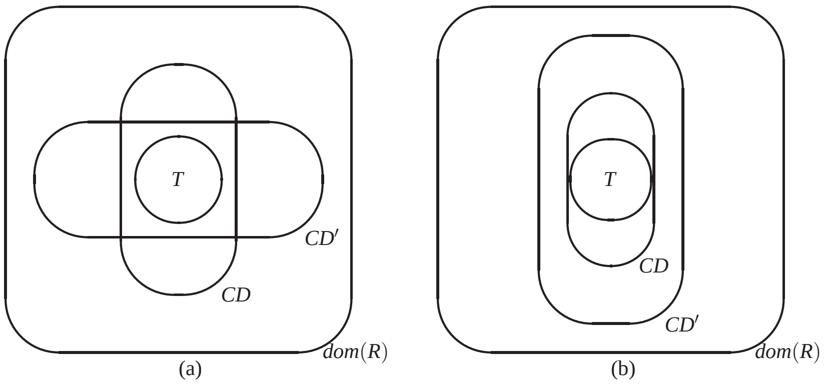

Figure 7: Relative Correctness-based Repair: Poor Precision, Perfect Recall

4.4 Program Repair with Relative Correctness

In light of the foregoing discussion, we argue that if a fault removal is expected to make a program more-correct than the original (vs absolutely correct) then it is only fair that it be tested for relative correctness rather than absolute correctness. In this way, if a program has several faults, we can remove them one at a time in a stepwise manner. To do so, we adopt the test data selection criterion and the oracle design discussed in section 4.2. As we recall, in order to test a program for relative correctness over a program P with respect to a specification R, we have to select test data in the competence domain of P with respect to R (CD in Figure 7) rather than to select it in $dom(R)$. As we can see from Figure 7(b), this ensures perfect recall since all programs that are relativeley correct with respect to P are selected. As for retrieval precision, it depends on the quality of the test data, but as Figure 7(a) shows, we can still select programs that are not more-correct than P, if T is not adequately distributed over CD; hence precisions remains an issue.

5 Illustration: Fermat Decomposition

5.1 Experimental Setup

To illustrate the distinction between program repair by absolute correctness and by relative correctness, we consider a program that performs the Fermat decomposition of a natural number, in which we introduce three changes. The space of a Fermat decomposition is defined by three natural variables, n, x and y and the specification is defined as follows:

$$R = \{(s,s')|((n \ \mathbf{mod} \ 2 = 1) \vee (n \ \mathbf{mod} \ 4 = 0)) \wedge n = x'^2 - y'^2\}.$$

A correct Fermat program (which we call p') is:

```
void fermatFactorization() {
   int n, x, y; // input/output variables
   int r; // work variable
   x = 0; r = 0;
   while (r < n) {r = r + 2 * x + 1; x = x + 1; }
   while (r > n) {int rsave; y = 0; rsave = r;
```

```
      while (r > n) {r = r - 2 * y - 1; y = y + 1; }
      if (r < n) {r = rsave + 2 * x + 1; x = x + 1; }}}
```

The three changes we introduce in this program are shown below; we do not call them faults yet because we do not know whether they meet our definition of a fault (Definition 4). A given number of changes (re: three in this case) can lead to fewer faults (if some changes cancel each other, or if one or more changes have no effect on the function of the program); also, a given number of changes (three in this case) can also lead to a larger number of faults (the same change can be remedied either by reversing the change or by altering the program elsewhere to cancel the change). We revisit this discussion in the next section.

We let p be the program obtained after introducing the changes to p':

```
void basep(int& n, int& x, int& y) {
   int r; x = 0; r = 0;
   while (r < n) {r = r + 2 * x - 1; /*change in r*/ x =x+1;}
      while (r > n) {int rsave; rsave = r; y = 0;
         while (r > n) {r =r-2*y+1; /*change in r*/ y =y+1;}
         if (r < n) {r =rsave+2*x-1; /*change in r*/ x =x+1;}}}
```

Most program repair methods proceed by generating mutants of the base program and testing them for absolute correctness; all we are advocating in this paper is that instead of testing mutants for absolute correctness, we ought to test them for relative correctness. To illustrate our approach, we generate mutants of program p, test them for absolte correctness, and show that none of them are (absolutely) correct. If absolute correctness were our only criterion, then this would be the (unsuccessful) end of the experiment. But we find that while none of the mutants are absolutely correct, some are strictly more-correct than p; hence the transition from p to these mutants represents a fault removal (by Definition 4). If we take these mutants as our base programs and apply the mutation generator to them, then test them for strict relative correctness, we can iteratively remove the faults of the program in a stepwise manner, climbing the relative correctness ordering until we reach a (absolutely) correct program.

Specifically, we start from program p and apply muJava to generate mutants using the single mutation option with the AORB operator (Arithmetic Operator Replacement, Binary). Whenever a set of mutants are generated, we subject them to three tests:

- A test for absolute correctness, using the oracle $\Omega(s,s')$.
- A test for relative correctness, using the oracle $\omega(s,s')$.
- A test for strict relative correctness, which in addition to relative correctness checks the presence of at least one state in the competence domain of the mutant that is not in the competence domain of the base program.

The mutants that are found to be strictly more-correct than the base program are used as new base programs, and the process is iterated again until at least one mutant is found to be absolutely correct; we select this mutant as the repaired version of the original program p. The main iteration of the test driver is given below. All the details of our experiment are posted online at `https://selab.njit.edu/programrepair/`.

```
int main ()
 {for (int mutant =1; mutant<= nbmutants; mutant++)
  {// test mutant vs spec. R for abs and rel correctness
   bool cumulabs=true; bool cumulrel=true; bool cumulstrict=false;
   while (moretestdata)
```

```
    {int n,x,y; int initn,initx,inity; //initial, final states
     bool abscor, relcor, strict;
     initn=td[tdi]; tdi++;  // getting test data
     n=initn; x=initx; y=inity;   // saving initial state
     callmutant(mutant, n, x, y);
     abscor = absoracle(initn, initx, inity, n, x, y);
     cumulabs = cumulabs && abscor;
     n=initn; x=initx; y=inity;  //  re-initializing
     basep(n, x, y);
     relcor = ! absoracle(initn, initx, inity, n, x, y) || abscor;
     strict = ! absoracle(initn, initx, inity, n, x, y) && abscor;
     cumulrel = cumulrel && relcor;
     cumulstrict = cumulstrict || strict;
 }}}
bool R (int initn, int initx, int inity, int n, int x, int y)
   {return ((initn%2==1) || (initn%4==0)) && (initn==x*x-y*y);}
bool domR (int initn, int initx, int inity)
   {return ((initn%2==1) || (initn%4==0));}
bool absoracle (int initn,int initx,int inity,int n,int x,int y)
   {return  (! (domR(initn, initx, inity))
             || R(initn, initx, inity, n, x, y));}
```

The main program includes two nested loops; the outer loop iterates over mutants and the inner loop iterates over test data. For each mutant and test datum, we execute the mutant and the base program on the test datum and test the mutant for absolute correctness (`abscor`), relative correctness (`relcor`) and strict relative correctness (`strict`); these boolean results are cumulated for each mutant in variables `cumulabs`, `cumulrel` and `cumulstrict`, and are used to diagnose the mutant. As for the Boolean functions `R`, `domR` and `absoracle`, they stem readily from the definition of R and from the oracle definitions given in section 4.2.

5.2 Experimental Results

Starting with program p, we apply muJava repeatedly to generate mutants, taking mutants which are found to be strictly more-correct as base programs and repeating until we generate a correct program. This proceeds as follows:

- When muJava is executed on program p, it produces 48 mutants, of which two ($m12$ and $m44$) are found to be strictly more-correct than p, and none are found to be absolutely correct with respect to R; we pursue the analysis of $m12$ and $m44$.
- *Analysis of* $m44$. When we apply muJava to $m44$, we find 48 mutants, none of them prove to be absolutely correct, nor relatively correct, nor strictly relatively correct.
- *Analysis of* $m12$. We find by inspection that $m12$ reverses one of the modifications we had applied to p' to find p; since $m12$ is strictly more-correct than p with respect to R, we conclude that the feature in question was in fact a fault in p with respect to R. When we apply muJava to $m12$, it generates 48 mutants, three of which prove to be strictly more-correct than $m12$: we name them $m12.19$, $m12.20$ and $m12.28$. All the other mutants are found to be neither absolutely correct with respect to R, nor more correct than $m12$.

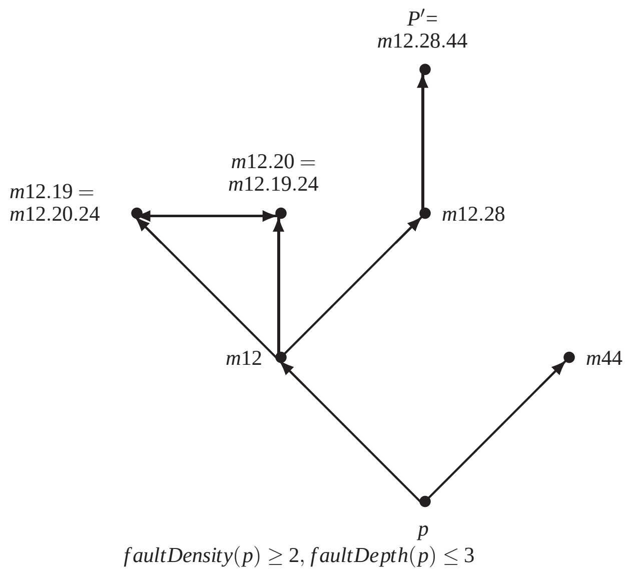

Figure 8: Relative Correctness-based Repair: Stepwise Fault Removal

- *Analysis of* $m12.19$. When we apply muJava to $m12.19$, it generates 48 mutants, none of which is found to be absolutely correct nor strictly more-correct than $m12.19$, but one ($m12.19.24$) proves to be identical to $m12.20$ and is more-correct than (but not strictly more-correct than, hence as correct as) $m12.19$.
- *Analysis of* $m12.20$. When we apply muJava to $m12.20$, it generates 48 mutants, none of which is found to be absolutely correct nor strictly more-correct than $m12.20$, but one ($m12.20.24$) proves to be identical to $m12.19$ and is more-correct than (but not strictly more-correct than, hence as correct as) $m12.20$.
- *Analysis of* $m12.28$. We find by inspection that $m12.28$ reverses a second modification we had applied to p' to obtain p; since $m12.28$ is strictly more-correct than $m12$, this feature is a fault in $m12$; whether it is a fault in p we have not checked, as we have not compared $m12.28$ and p for relative correctness. When we apply muJava to $m12.28$, we find a single mutant, namely $m12.28.44$ that is absolutely correct with respect to R, more-correct than $m12.28$ with respect to R, and strictly more-correct than $m12.28$ with respect to R.
  - *Analysis of* $m12.28.44$. We find by inspection that $m12.28.44$ is nothing but the original Fermat decomposition program we have started out with: p'.

The results of this analysis are represented in Figure 8. Note that $m12$ and $m44$ are strictly more-correct than p with respect to R; hence (according to Definition 4) the mutations that produced these programs from p constitute fault removals; whence we can say that p has at least two faults, which we write as $faultDensity(p) \geq 2$. On the other hand, this experiment shows that we can generate a correct program (p') from p by means of three fault removals; if we let the *Fault Depth* of a program be the minimal number of fault removals that separate it from a correct program, then we can write: $faultDepth(p) \leq 3$.

6 Conclusion

In this paper we discuss how we can use the concept of relative correctness to refine the technique of program repair by mutation testing. We argue that when we remove a fault from a program, in the context of program repair, we have no reason to expect the resulting program to be correct unless we know (how do we ever?) that the fault we have just removed is the last fault of the program. Therefore we should, instead, be testing the program for relative correctness rather than absolute correctness. We have found that testing a program for relative correctness rather than absolute correctness has an impact on test data selection as well as oracle design, and have discussed practical measures to this effect. As an illustration of our thesis, we take a simple example of a faulty program, which we can repair in a stepwise manner by seeking to derive successively more-correct mutants; by contrast, the test for absolute correctness keeps excluding all the mutants except the last, and fails to recognize that some mutants, while being incorrect, are still increasingly more correct than the original. We are not offering a seamless validated solution as much as we are seeking to draw attention to some opportunities for enhancing the practice of software testing.

Our research agenda includes further exploration of the technique proposed in this paper to assess its feasibility and effectiveness on software benchmarks, as well as techniques to streamline test data selection to enhance the precision of relative-correctness-based program repair (re: Figure 7).

Other researchers [14][12][13] have introduced a concept of relative correctness and have explored this concept in the context of program repair. Our work differs significantly from theirs in many ways: we represent specifications by relations whereas they specify them with assertions; we capture program semantics with input/output functions whereas they capture them by means of execution traces; we define relative correctness by means of competence domains and specification violations whereas they define it by means of correct traces and incorrect traces; we introduce relative correctness as a way to define faults whereas they introduce it as a way to compare program versions; we have explored the implications of relative correctness on several aspects of software engineering, whereas they focus primarily on sooftware testing.

This paper complements our earlier work in the following manner: In [16] we introduce relative correctness for deterministic programs, and explore the mathematical properties of this concept; in [4] we generalize the concept of relative correctness to non-deterministic programs and study its mathematical properties. In [5] (*Programming without Refinement*) we argue that while we generally think of program derivation as the process correctness preserving transformations using refinement, it is possible to derive programs by correctness-enhancing transformations using relative correctness; one of the interesting advantages of relative correctness-based correctness enhancing transformations is that they capture, not only the derivation of programs from scratch, but also virtually all software maintenance activities. We can argue in fact that software evolution and maintenance is nothing but an attempt to enhance the correctness of a software product with respect to a specification. In [7] (*Debugging without Testing*) we show how relative correctness can be used to define faults and fault removals, and that we can use these definitions to remove a fault from a program and prove that the fault has ben removed, all by static analysis, without testing.

Acknowledgements

The authors are very grateful to the anonymous reviewers for their insightful feedback, which has greatly contributed to the clarity and content of this paper.

References

[1] A. Arcuri & X. Yao (2008): *A Novel Co-evolutionary Approach to Automatic Software Bug Fixing*. In: *CEC 2008*, pp. 162–168, doi:10.1109/CEC.2008.4630793.

[2] Ch. Brink, W. Kahl & G. Schmidt (1997): *Relational Methods in Computer Science*. Springer Verlag.

[3] Kim D., Nam J., Song J. & Kim S. (2013): *Automatic patch generation learned from human-written patches*. In: *ICSE 2013*, pp. 802–811, doi:10.1109/ICSE.2013.6606626.

[4] Jules Desharnais, Nafi Diallo, Wided Ghardallou, Marcelo Frias, Ali Jaoua & Ali Mili (2015): *Mathematics for Relative Correctness*. In: *Relational and Algebraic Methods in Computer Science, 2015*, Lisbon, Portugal, pp. 191–208, doi:10.1007/978-3-319-24704-5_12.

[5] Nafi Diallo, Wided Ghardallou & Ali Mili (2015): *Program Derivation by Correctness Enhancements*. In: *Proceedings, Refinement 2015*, Oslo, Norway.

[6] E.W. Dijkstra (1976): *A Discipline of Programming*. Prentice Hall.

[7] Wided Ghardallou, Nafi Diallo, Ali Mili & Marcelo Frias (2016): *Debugging without Testing*. In: *Proceedings, International Conference on Software Testing*, Chicago, IL.

[8] Divya Gopinath, Mohammad Zubair Malik & Sarfraz Khurshid (2011): *Specification Based Program Repair Using SAT*. In: *Proceedings, TACAS*, pp. 173–188, doi:10.1007/978-3-642-19835-9_15.

[9] C. Le Goues, T. Nguyen, S. Forrest & W. Weimer (2012): *GenProg: A Generic Method for Automated Software Repair*. *IEEE Transactions on Software Engineering* 31(1), doi:10.1109/TSE.2011.104.

[10] D. Gries (1981): *The Science of programming*. Springer Verlag, doi:10.1007/978-1-4612-5983-1.

[11] E.C.R. Hehner (1992): *A Practical Theory of Programming*. Prentice Hall, doi:10.1007/978-1-4419-8596-5.

[12] Shuvendu K. Lahiri, Kenneth L. McMillan, Rahul Sharma & Chris Hawblitzel (2013): *Differential Assertion Checking*. In: *Proceedings, ESEC/ SIGSOFT FSE*, pp. 345–455, doi:10.1145/2491411.2491452.

[13] Francesco Logozzo & Thomas Ball (2012): *Modular and Verified Automatic Program Repair*. In: *Proceedings, OOPSLA*, pp. 133–146, doi:10.1145/2384616.2384626.

[14] Francesco Logozzo, Shuvendu Lahiri, Manual Faehndrich & Sam Blackshear (2014): *Verification Modulo Versions: Towards Usable Verification*. In: *Proceedings, PLDI*, p. 32, doi:10.1145/2594291.2594326.

[15] Yu Seung Ma, Jeff Offutt & Yong Rae Kwon (2005): *Mu Java: An Automated Class Mutation System*. *Software Testing, Verification and Reliability* 15(2), pp. 97–133, doi:10.1002/stvr.v15:2.

[16] Ali Mili, Marcelo Frias & Ali Jaoua (2014): *On Faults and Faulty Programs*. In Peter Hoefner, Peter Jipsen, Wolfram Kahl & Martin Eric Mueller, editors: *Proceedings, RAMICS: 14th International Conference on Relational and Algebraic Methods in Computer Science*, *Lecture Notes in Computer Science* 8428, Springer, Marienstatt, Germany, pp. 191–207, doi:10.1007/978-3-319-06251-8_12.

[17] H.D. Mills, V.R. Basili, J.D. Gannon & D.R. Hamlet (1986): *Structured Programming: A Mathematical Approach*. Allyn and Bacon, Boston, Ma.

[18] G. Schmidt & T. Stroehlein (1990): *Relationen und Graphen*. Springer-Verlag, Berlin, Germany, doi:10.1002/zamm.19910710911.

[19] Debroy V. & Wong W.E. (2010): *Using Mutation to Automatically Suggest Fixes to Faulty Programs*. In: *Proceedings, ICST 2010*, pp. 65–74.

[20] L. Zemín, S. Guttiérrez, S. Perez de Rosso, N. Aguirre, A. Mili, A. Jaoua & M. Frias (2015): *Stryker: Scaling Specification-Based Program Repair by Pruning Infeasible Mutants with SAT*. Technical Report, ITBA, Buenos Aires, Argentina.