Debugging Without Testing

Wided Ghardallou
University of Tunis, El Manar, Tunis, Tunisia
wided.ghardallou@gmail.com

Nafi Diallo and Ali Mili
CCS, NJIT, Newark NJ 07102-1982
ncd8@njit.edu, mili@njit.edu

Marcelo F. Frias
ITBA, Buenos Aires, Argentina
mffrias@gmail.com

***Abstract*—It is so inconceivable to debug a program without testing it that these two words are used nearly interchangeably. Yet we argue that using the concept of relative correctness we can indeed remove a fault from a program and prove that the fault has been removed, by proving that the new program is more correct than the original. This is a departure from the traditional roles of proving and testing methods, whereby static proof methods are applied to a correct program to prove its correctness, and dynamic testing methods are applied to an incorrect program to expose its faults.**

***Keywords*—debugging; testing; correctness; relative correctness; faults; fault removal.**

I. Introduction

In (Ali Mili, 2014) Mili et al introduce a definition of *relative correctness*, i.e. the property of a program to be more-correct than another with respect to a specification; to contrast relative correctness with the traditional definition of program correctness, we may refer to the latter as *absolute correctness*. Given a specification, we can use absolute correctness to divide candidate programs into two classes: correct programs, and incorrect programs. But by using relative correctness, we can arrange candidate programs according to a rich partial ordering structure, rather than simply dividing them into two classes. Also, whereas traditionally the division of labor between testing methods and proving methods is clear cut, whereby proving methods are deployed on correct programs to prove their correctness and testing methods are deployed on incorrect programs to expose their faults, relative correctness enables us to straddle this dividing line. Specifically, we can use relative correctness to prove that a program, though it may be incorrect, is still more-correct than another. This approach can usefully complement testing activities, by virtue of the law of diminishing returns.

An obvious application of this concept is in fault removal: when we remove a fault from a program, we have no reason to expect the new program to be correct, since it may have other hidden faults; but we ought to expect it to be more-correct than the original, since that is the only condition under which we may consider that the fault has indeed been removed. In this paper, we explore the possibility of removing faults by static analysis of the program's source code, and proving that the fault has effectively been removed by proving that the new program is more-correct than the original. Broadly speaking, this method has the same advantages and disadvantages as traditional methods for proving correctness by static analysis: namely that it offers the confidence and certainty of formally provable results, at the cost of mathematical formalisms and limited scalability. At the same time as we present the method, we also discuss means to capitalize on its advantages while mitigating its disadvantages.

In section II, we introduce some elements of relational mathematics that we use throughout the paper to formulate our results; we use this background in section III to introduce the concept of relative correctness and in section IV to discuss how relative correctness can be used to provably remove faults in programs. We summarize our work, compare it to related work and sketch future research directions in sevtion V.

II. RELATIONAL MATHEMATICS

We assume the reader familiar with elementary relational concepts (Chris Brink, 1997); the goal of this section is merely to introduce notations and definitions that we use throughout the paper. Given a set S, we let a relation R on S be a subset of the Cartesian product $S \times S$. Elements of a relation are usually denoted by pairs of the form (s, s'). Constant relations on S include the empty relation ($\emptyset$), the identity relation (I) and the universal relation (L). Operations on relations include the usual set theoretic operations ($\cup, \cap, /, \overline{\ }$), as well as the relational product, which we merely represent by concatenating the operands (as we do in arithmetic). Given a relation R on S, we let the reverse of R be the relation denoted by $\widehat{R}$ (or $R^\wedge$) and defined by $\widehat{R} = \{(s, s') | (s', s) \in R\}$. We use the notation R^i, where R is a relation and i is a natural number to denote the product of R by itself i times if $i > 0$, and I if $i = 0$. We let the *transitive closure* of relation R be the relation denoted by R^+ and defined by $R^+ = \{(s, s') | \exists i > 0: (s, s') \in R^i\}$, and we let the *reflexive transitive* closure of R be denoted by R^* and defined by: $R^* = I \cup R^+$.

We say that a relation R is a *vector* if and only if $RL = R$. Vectors are relations of the form $R = A \times S$, for some subset A of S. We use vectors as a relational representations of sets; hence for example, we represent the domain of relation R by the vector RL. We say that relation R *refines* relation R' if and only if: $RL \cap R'L \cap (R \cup R') = R'$. We say that relation R is reflexive if and only if $I \subseteq R$, that relation R is symmetric if and only if $R = \widehat{R}$, and that relation R is transitive if and only if $RR \subseteq R$.

Given a program p on variables $x, y, \dots z$ of types $X, Y, \dots Z$; we let the *space* of p be defined as the Cartesian product $S = X \times Y \times \dots Z$. We usually use the name s as an element of S, to stand for the aggregate $\langle x, y, \dots z \rangle$. Program p defines a function from its initial states to its final states, which we represent by upper case P. Specifications on space S are relations on S, and we say that a program p is *correct* with respect to a specification R if and only if P refines R. We admit without proof that a program p is *correct* (or, for contrast, *absolutely correct*) with respect to a specification R if and only if $(R \cap P)L = RL$.

III. RELATIVE CORRECTNESS

A. Deterministic Programs

Given a specification R on space S and a program p on space S, we find that the domain of $(R \cap P)$ is the set of initial states for which program p behaves according to specification R; we refer to this set as the *competence domain* of p with respect to R. We say that a program p' is more-correct with respect to R than a program p if and only if the competence domain of p' is a superset of the competence domain of p, i.e.

$$(R \cap P')L \sqsupseteq (R \cap P)L.$$

By construction, the competence domain of any candidate program with respect to a specification R is necessarily a subset of (or equal to) RL; according to the discussion of the previous section, when it equals RL, the program is correct with respect to R. Hence relative correctness culminates in absolute correctness. To illustrate relative correctness, and contrast it to absolute correctness, we present a simple example of a specification and ten candidate programs, and show how these ten candidates are ranked by relative correctness, while absolute correctness merely divides them into two broad classes. We consider the following specification R on space S of the natural numbers:

$$R = \{(x, x') | x^2 \le x' \le x^3\},$$

and we consider the following candidate programs; for each program, we present the program function, then the competence domain with respect to R.

0. **`p0: {abort}`**; $P_0 = \emptyset$. $CD_0 = \emptyset$.
1. **`p1: {x=0;}`**;
   $P_1 = \{(x, x') | \, x' = 0\}$. $CD_1 = \{0\}$.
2. **`p2: {x=1;}`**;
   $P_2 = \{(x, x') | \, x' = 1\}$. $CD_2 = \{1\}$.
3. **`p3: {x=2*x^3-8}`** ;
   $P_3 = \{(x, x') | \, x' = 2x^3 - 8\}$. $CD_3 = \{2\}$.
4. **`p4: {skip}`**;
   $P_4 = \{(x, x') | \, x' = x\}$. $CD_4 = \{0,1\}$.
5. **`p5: {x=2*x^3-3*x^2+2}`** ; $P_5 = \{(x, x') | \, x' = 2x^3 - 3x^2 + 2\}$.$CD_5 = \{1,2\}$.
6. **`p6: {x=x^4-5*x}`**;
   $P_6 = \{(x, x') | \, x' = x^4 - 5x\}$. $CD_6 = \{0,2\}$.
7. **`p7: {x=x^2}`**;
   $P_7 = \{(x, x') | \, x' = x^2\}$. $CD_7 = S$.
8. **`p8: {x=x^3}`** ;
   $P_8 = \{(x, x') | \, x' = x^3\}$. $CD_8 = S$.
9. **`p9: {x=(x^2+x^3)/2}`** ;
   $P_9 = \{(x, x') | \, x' = \frac{x^2 + x^3}{2}\}$. $CD_9 = S$.

The following figure shows the graph of the relative correctness relationships between these programs; note that while absolute correctness divides this set of candidates into two classes (correct vs incorrect), relative correctness defines a richer structure of partial ordering.

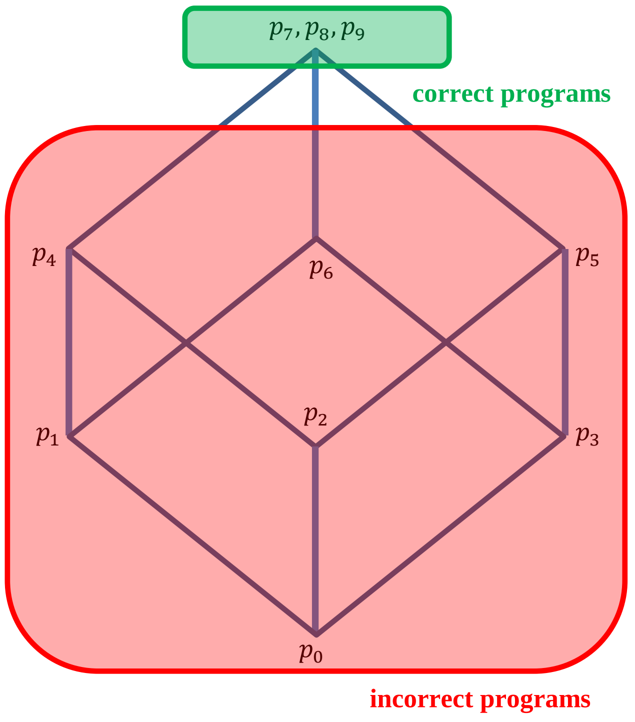

Figure 1: Ranking Candidates by Relative Correctness

B. Non Deterministic Programs

In [1], Desharnais et al generalize the definition of relative correctness to non-deterministic programs. Doing so is important not only because we want to compare non-deterministic programs, but also because we want to compare deterministic programs without having to compute their deterministic function in all its detail. In this section, we merely introduce this definition, and discuss its significance, reverting for the remainder of this paper to the definition for deterministic programs.

Given a specification R and two (not necessarily deterministic) programs p and p', we say that p' is more-correct than p with respect to specification R if and only if:

$$(R \cap P)L \sqsubseteq (R \cap P')L \wedge (R \cap P)L \cap \overline{R} \cap P' \sqsubseteq P.$$

Informally, p' is more-correct than p if and only if it has a larger competence domain, and does not violate R except whenever p does (i.e. violates R less often than p). Note that while the competence domain of a deterministic program p with respect to a specification R is the set of initial states for which p <u>*does*</u> behave according to R, the competence domain of a non-deterministic program is the set of initial states where program p <u>*may*</u> behave according to R.

C. *Faults and Fault Removal*

Before we define faults, we must recognize that any definition of a fault implicitly assumes a scale of granularity; when we resolve to locate a fault in a program, we usually mean to identify a line of code, or a statement, or a condition, or a lexical token that may be faulty. We use the term *feature* to refer to a piece of source code at the appropriate level of granularity, and we introduce the following definition: Given a specification R and a program p, and given a feature f in p, we say that f is a *fault* in program p with respect to R if and only if it admits a substitution that would make the program more-correct. We assume that <u>*skip*</u> (the empty statement) is part of our vocabulary of statements, so that this definition includes missing statements and extraneous statements as possible faults.

Also, we define monotonic fault removal as follows: Given a specification R, a program p, and a fault f in p, we say that the pair of features (f, f') represents a fault removal of f in p with respect to R if and only if the program p' obtained from p by replacing f with f' is more-correct than p with respect to R.

We argue that this definition of (monotonic) fault removal provides us with a logical framework for corrective maintenance through correctness-enhancing transformations, in the same way as (and for the same purpose as) refinement provides us with a logical framework for program derivation through correctness-preserving transformations. Also, while we use the term *monotonic* (in monotonic fault removal) for emphasis, we consider that a substitution (f, f') cannot be considered a fault removal unless it is indeed monotonic (i.e. it makes p' more-correct than p).

D. *Provable Fault Removal*

If we find a fault f in a program p, replace f by a feature f' to obtain a new program p', then test program p' on some test data T, then depending on the configuration of the competence domain of p (say, CD), the competence domain of p' (say, CD') and test data T, we expose ourselves to two risks:

- Program p' may fail on test data T even though it is more-correct than p; this may happen if CD' is a superset of CD, but it is not a superset of T. See Figure 2(a).
- Program p' may run successfully on test data T even though it is not more-correct than program p; this may happen if CD' is a superset of T without being a superset of CD. See Figure 2(b).

In both of these situations, the test misleads us to the wrong conclusion about the fault removal. As an alternative, we consider proving that p' is more-correct than p, rather than trying to infer relative correctness through testing. One of the main obstacles to this alternative approach is that it requires that we compute the function of the two programs, a rather steep requirement, usually. Hence we consider ways to establish relative correctness without having to compute program functions. This is the focus of the next section.

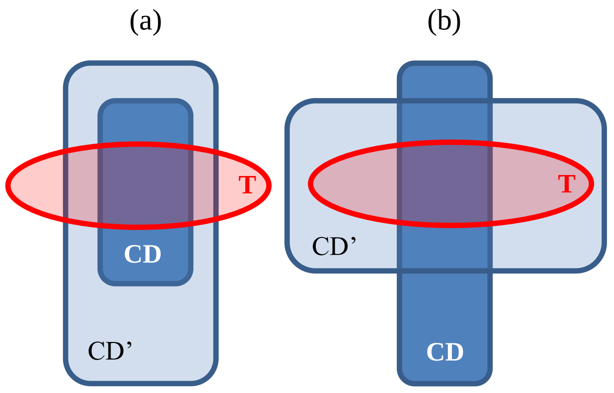

Figure 2: Misleading Tests

IV. FAULT REMOVAL IN ITERATIVE PROGRAMS

A. *Invariant Relations*

Let w be a while loop on space S of the form

```
w:  while (t) {b;}
```

and let B be the function of the loop body and T be the following relation, which represents the loop condition, i.e. $T = \{(s, s')|t(s)\}$. We assume that this while loop terminates for all states in S, and we define an *invariant relation* of w to be a reflexive transitive superset of $(T \cap B)$. According to this definition, an invariant relation of w is a superset (an approximation) of the reflexive transitive closure of $(T \cap B)$. Because invariant relations have a similar name to the widely known invariant assertions [2], we highlight here the main differences and relations between them.

- Whereas an invariant assertion is a unary predicate (characterizing a single state), an invariant relation is a binary relation (characterizing two states).
- Whereas an invariant assertion characterizes states of the iteration after an arbitrary number of iterations, an invariant relation characterizes two states separated by an arbitrary number of iterations (hence, in particular, the initial state and final state).
- Whereas an invariant assertion depends on the loop as well as on the precondition of the loop, an invariant relation depends exclusively on the loop.

- Whereas all invariant assertions stem from invariant relations, only a small class of invariant relations can be derived from invariant assertions.

As an illustration, we consider the following simple loop on natural variables n, f, and k:

```
{k=1; f=1; while (k<=n) {f=f*k; k++;}}
```

Then

- An invariant assertion of this loop, for the given initialization, is: $A \equiv (f = (k-1)!)$.
- An invariant relation for the while loop, regardless of its initialization, is: $V = \left\{(s,s') | \ \frac{f}{(k-1)!} = \frac{f'}{(k'-1)!}\right\}$.
- Note that we can derive the invariant assertion A from the invariant relation V by taking its post-restriction to the precondition $(f = 1 \wedge k = 1)$.

B. *Invariant Relations and Absolute Correctness*

In [3] we present a method to prove the correctness or incorrectness of a loop with respect to a specification, using invariant relations. This method is based on the following two propositions, which give, respectively, a sufficient condition and a necessary condition of correctness of a (uninitialized) while loop with respect to a specification R.

Proposition: Sufficient condition of correctness. Let w be a while loop of the form **`while (t) {b;}`** on space S and let R be a specification on S. If w admits an invariant relation V that satisfies the following condition,

$$R\bar{T} \cap VL \cap \left(V \cup R \cap \hat{\bar{T}}\right) = V$$

then w is correct with respect to R.

Interpretation: this condition provides that invariant relation V captures sufficient information about w to subsume specification R; we do not need to compute the function of w, V captures enough information to conclude that w is correct.

Proposition: Necessary condition of correctness. Let w be a while loop of the form **`while (t) {b;}`** on space S, let R be a specification on S, and let V be an invariant relation of w. If w is correct with respect to R then we have necessarily:

$$(V \cap R)\bar{T} = RL.$$

Interpretation: while this is a necessary condition of correctness, it is best to interpret it by considering that its negation is a sufficient condition of incorrectness. This proposition provides in effect that any while loop that admits an invariant relation V that does not satisfy this condition could not possibly be correct with respect to R. In other words, any while loop that admits an invariant relation V that satisfies the condition (note the change from = to ≠)

$$(V \cap R)\bar{T} \neq RL.$$

is necessarily incorrect with respect to R. Any invariant relation V that satisfies this condition is said to be *incompatible* with respect to specification R. Any invariant relation that is not incompatible is said to be *compatible*.

In [3] we present an algorithm for proving the correctness or incorrectness of a loop with respect to a specification, which proceds as follows:

- Using an invariant relations generator, we generate invariant relations one by one, and test the sufficient condition and necessary condition.
- If the aggregate of invariant relations found so far satisfy the sufficient condition then we conclude that the loop is correct, and we exit.
- If one of the invariant relations proves to be incompatible with R, we conclude that the loop is incorrect, and we exit.
- If we run out of invariant relations before we reach the conclusion that the loop is correct or that the loop is incorrect, then we conclude that we do not know enough about the loop to rule on its correctness (hence we must upgrade our invariant relations generator), and we exit.

In the next section we discuss how we can use a variation of this algorithm to establish relative correctness, rather than absolute correctness.

C. *Invariant Relations and Relative Correctness*

Given a while loop w of the form **`while (t) {b;}`** on space S and a specification R on S, we are interested to determine whether w is correct with respect to R, and if not how we can locate and remove a fault in w. Ideally, we want to support all the steps in this process, namely:

- Determine that the loop is incorrect (for else there is no fault to remove).
- Determine the location of the fault.
- Determine what to replace the fault with.
- Prove that the substitution constitutes a monotonic fault removal.

To this effect, we consider the following proposition, which we give without proof.

Proposition: Fault Removal Through Enhanced Compatibility. Let R be a specification on space S and let w be a while loop on S of the form **`while (t) {b;}`** which terminates for all s in S. Let Q be an invariant relation of w that is incompatible with R, and let C be the largest invariant relation of w such that $W = (C \cap Q) \cap \bar{T}^\wedge$. Let w' be a while loop that has C as an invariant relation, terminates for all s in S, and admits an invariant relation Q' that is compatible with R and satisfies the condition $W' = (C \cap Q') \cap \hat{\bar{T}}^\wedge$. Then w' is strictly more-correct than w with respect to R.

Interpretation: This proposition provides that if we change loop w in such a way as to replace an incompatible invariant relation (Q) into a compatible invariant relation (Q') of equal strength (so that $(C \cap Q') \cap \hat{\bar{T}}^\wedge$ is deterministic, just like $(C \cap Q) \cap \bar{T}^\wedge$) while preserving all the other invariant relations (C), then we obtain a strictly more-correct while loop. What we mean by *strictly* more-correct is, of course, that the competence domain of w' is a proper superset of the competence domain of w; in other words, w' behaves correctly for all states on which w behaves correctly, and it behaves correctly for at least one state on which w fails.

Using this Proposition, we propose the following algorithm for fault removal in while loops:

1. *Determination that the loop is faulty*. Given the specification R and the while loop w, we generate all the invariant relations we can, and place them in two separate columns, one for compatible relations and one for incompatible relations.
   - If the *incompatible* column has at least one invariant relation, then the loop is incorrect, hence it has a fault.
   - If the *incompatible* column is empty and the intersection of all the compatible invariant relations satisfies the sufficient condition of correctness, then the loop is correct.
   - If neither of the conditions above hold, then we cannot rule on the correctness of the loop, and the algorithm fails (the invariant relations generator needs to be upgraded).
2. *Localization of the Fault.* We consider the *incompatible* column and select from it an invariant relation that involves the fewest possible variables; for the same number of variables, we select the invariant relation (say Q) whose variables are involved in the smallest number of statements in the loop. We select one of these statements as the feature that we want to correct.
3. *Guidance to modify the selected statement*. We need to modify the selected statement in such a way as to replace the current incompatible invariant relation (Q) with a compatible invariant relation (Q'). But we want to do so without affecting the compatible invariant relations. This constraint is used to generate a condition that guides us in the modification process. Let C be the intersection of all the compatible invariant relations, let $x_1, x_2, x_3, \ldots\ x_n$ be the program variables, and let x_1, x_2 be the two variables that appear in Q. Then, to preserve the compatible invariant relations of the loop, variables x_1, x_2 , x'_1, x'_2 must satisfy the following constraint:

   $$\exists\ x_3, \ldots\ x_n, , x'_3, \ldots\ x'_n:$$
   $$(x_1, x_2, x_3, \ldots\ x_n, x'_1, x'_2, x'_3, \ldots\ x'_n) \in C.$$

   We refer to this condition as the *condition of compatibility preservation.*
4. *Verification of Fault Removal.* Once we have changed the selected statement in such a way as to preserve the compatible invariant relations, we recompute the invariant relations and ensure that the selected incompatible invariant relation is now replaced by a compatible invariant relation. This ensures that we now have a more-correct program than we did before. This sends us back to step 1, to check whether the loop has now become correct (if its compatible relations subsume the specification) or whether it is still incorrect (if the incompatible column is still not empty).

D. *Invariant Relations and Relative Correctness Proofs*

As an illustrative example, we consider the state space S defined by the following variable declarations:

```
const float upsilon = 0.00001;
const float a= 0.15;
const float b= 0.08;
//  we always have:  0<b<a<1.0;
float r, p, n, x, m, l, k, y, w, y, z, v, u, d;  int t;
```

and we consider program w on a state space S defined by:

```
p1: while (abs(r-p)>upsilon)
        {t=t+1;  n=n+x;  m=m-l;  l=(1+b)*l;
         k=k+1000; y=n+k;  w=w+z; z=(1+a)+z;
         v=w+k;r=(v-y)/y;u=(m-n)/n;d=r-u;}
```

The invariant relations generator produces fourteen invariant relations:

- $V_1 = \{(s, s') |\ x' = x\}$.
- $V_2 = \{(s, s') | t \leq t'\}$.
- $V_3 = \{(s, s') | k \leq k'\}$.
- $V_4 = \{(s, s') || l | \leq | l' |\}$.
- $V_5 = \{(s, s') | z \leq z'\}$.
- $V_6 = \{(s, s') | k - 1000 \times t = k' - 1000 \times t'\}$.
- $V_7 = \{(s, s') | l \times (1+b)^{-z} = l' \times (1+b')^{-z'}\}$.
- $V_8 = \{(s, s') | l \times (1+b)^{-\frac{k}{1000}} = l' \times (1+b')^{-\frac{k\prime}{1000}}\}$.
- $V_9 = \{(s, s') | l \times (1+b)^{-\frac{z}{(1+a)}} = l' \times (1+b')^{-\frac{z\prime}{(1+a)}}\}$.
- $V_{10} = \{(s, s') | z - (1+a) \times t = z' - (1+a) \times t'\}$.
- $V_{11} = \{(s, s') | 1000 \times z - (1+a) \times k = 1000 \times z' - (1+a) \times k'\}$.
- $V_{12} = \{(s, s') | m + \frac{l}{b} = m' + \frac{l\prime}{b}\}$.
- $V_{13} = \{(s, s') | w + z \times \frac{z-1-a}{2\times(1+a)} = w' + z' \times \frac{z\prime-1-a}{2\times(1+a)}\}$.
- $V_{14} = \{(s, s') | 1000 \times n - k \times x = 1000 \times n' - k' \times x'\}$.

We consider the following specification R on space S:

$$R = \{(s, s') | z > 0 \wedge x = x' \wedge w' = w - z \times \frac{1 - (1+a)^{t'-t}}{a} \wedge m' \geq 0 \wedge l' \geq 0\}$$

We review all the invariant relation for compatibility with respect to R; this is done using Mathematica (© Wolfram Research), by writing a logical formula that corresponds to the condition of compatibility discussed above. We find:

Compatible	Incompatible
$V_1, V_2, V_3, V_4, V_5, V_6, V_{11}, V_{14}$	$V_7, V_8, V_9, V_{10}, V_{12}, V_{13},$

We select invariant relation V_7 for remediation; the variables that appear in this relation are l and z. We compute the condition of compatibility preservation, and we find:

$$|l|\leq l' \wedge z\leq z'.$$

We focus on variable z, consider the statement where this variable is modified, and consider alternative statements that satisfy the constraint. For each alternative, we recompute the new invariant relation that stems from the new statement and check for compatibility. We find the following substitute:

```
z=(1+a)*z;
```

Hence the new program becomes:

```
P2: while (abs(r-p)>upsilon)
        {t=t+1;n=n+x;m=m-l;l=(1+b)*l;
         k=k+1000;y=n+k;w=w+z;z=(1+a)*z;
         v=w+k;r=(v-y)/y;u=(m-n)/n;d=r-u;}
```

We do not know whether this program is correct, but we know that it is more-correct than the original program; if we test it and it fails, it will not be because our fault removal was wrong; rather it will be because it has other faults. When we run the invariant relations generator on this program, we find the following list.

- $V_1 = \{(s,s')|\ x = x'\}$.
- $V_2 = \{(s,s')|\ t\leq t'\}$.
- $V_3 = \{(s,s')|\ k\leq k'\}$.
- $V_4 = \{(s,s')|\ |l|\leq|l'|\}$.
- $V_5 = \{(s,s')|\ z\leq z'\}$.
- $V_6 = \{(s,s')|\ k - 1000 \times t = k' - 1000 \times t'\}$.
- $V_7 = \{(s,s')|\ l\leq l'\}$.
- $V_8 = \{(s,s')|\ 1000 \times l - (1+b) \times k = 1000 \times l' - (1+b) \times k'\}$.
- $V_9 = \{(s,s')|\ (1+b) \times z - (1+a) \times l = (1+b) \times z' - (1+a) \times l'\}$.
- $V_{10} = \{(s,s')|\ 1000 \times z - (1+a) \times k = 1000 \times z' - (1+a) \times k'\}$.
- $V_{11} = \{(s,s')|\ 1000 \times n - x \times k = 1000 \times n' - x' \times k'\}$.
- $V_{12} = \{(s,s')|\ (1+b) \times n - x \times l = (1+b) \times n' - x' \times l'\}$.
- $V_{13} = \{(s,s')|\ z \times (1+a)^{-t} = z' \times (1+a)^{-t'}\}$.
- $V_{14} = \{(s,s')|\ z \times (1+a)^{-k/1000} = z' \times (1+a)^{-k'/1000}\}$.
- $V_{15} = \{(s,s')|\ w - \frac{z}{a} = w' - \frac{z'}{a}\}$.
- $V_{16} = \{(s,s')| m + \frac{l}{b} = m' + \frac{l'}{a}\}$.

Checking these invariant relations for compatibility against specification R, we find the following clasification:

Compatible	Incom-patible
$V_1, V_2, V_3, V_4, V_5, V_6, V_7, V_8, V_9\ V_{10}, V_{11}, V_{12}, V_{13}, V_{14}, V_{15}$	V_{16}

Note that the same fault removal can turn several incompatible relations into compatible relations; also, when we change a statement in a loop, our invariant relations generator may have to use different code patterns to generate invariant relations.

Relation V_{16} refers to variables l and m, hence these are the variables we may modify. We generate the condition on variables l and m under which modification of these variables does not affect compatible invariant relations, and find the following:

$$(l = 0 \wedge l' = 0) \vee \left(l\leq l' \wedge\ (l\geq 0\ \vee l + l' \geq 0)\right).$$

Looking at the statement that updates variable l, we find that it meets (the second clause of) this condition as it is; hence if we do not change it, we are assured not to affect any compatible invariant relation. We focus on variable m, and we suggest to change statement (**m=m-l**) into (**m=m+l**). This yields the following program:

```
P3: while (abs(r-p)>upsilon)
        {t=t+1;n=n+x;m=m+l;l=(1+b)*l;
         k=k+1000;y=n+k;w=w+z;z=(1+a)*z;
         v=w+k;r=(v-y)/y;u=(m-n)/n;d=r-u;}
```

We compute the invariant relations of this program and find:

- $V_1 = \{(s,s')|\ x = x'\}$.
- $V_2 = \{(s,s')|\ t\leq t'\}$.
- $V_3 = \{(s,s')|\ k\leq k'\}$.
- $V_4 = \{(s,s')|\ |l|\leq|l'|\}$.
- $V_5 = \{(s,s')|\ z\leq z'\}$.
- $V_6 = \{(s,s')|\ k - 1000 \times t = k' - 1000 \times t'\}$.
- $V_7 = \{(s,s')|\ l\leq l'\}$.
- $V_8 = \{(s,s')|\ 1000 \times l - (1+b) \times k = 1000 \times l' - (1+b) \times k'\}$.
- $V_9 = \{(s,s')|\ (1+b) \times z - (1+a) \times l = (1+b) \times z' - (1+a) \times l'\}$.
- $V_{10} = \{(s,s')|\ 1000 \times z - (1+a) \times k = 1000 \times z' - (1+a) \times k'\}$.

- $V_{11} = \{(s, s') | \ 1000 \times n - x \times k = 1000 \times n' - x' \times k'\}$.
- $V_{12} = \{(s, s') | \ (1 + b) \times n - x \times l = (1 + b) \times n' - x' \times l'\}$.
- $V_{13} = \{(s, s') | \ z \times (1 + a)^{-t} = \ z' \times (1 + a)^{-t'}\}$.
- $V_{14} = \{(s, s') | \ z \times (1 + a)^{-k/1000} = \ z' \times (1 + a)^{-k'/1000}\}$.
- $V_{15} = \{(s, s') | \ w - \frac{z}{a} = w' - \frac{z'}{a}\}$.
- $V_{16} = \{(s, s') | m - \frac{l}{b} = m' - \frac{l'}{b}\}$.

When we check these invariant relations against specification R for compatibility, we find that they are all compatible.

Compatible	**Incom-patible**
$V_1, V_2, V_3, V_4, V_5, V_6, V_7, V_8, V_9$ $V_{10}, V_{11}, V_{12}, V_{13}, V_{14}, V_{15}, V_{16}$	

<u>This does not mean that program p3 is correct.</u> All it means is that program p3 is more-correct than programs p2 and p1; the absence of incompatible relations is not sufficient to ensure correctness; all it means is that we did not prove the program incorrect).

We do find that program p3 is correct with respect to R, by virtue of the proposition of sufficient correctness, because we find that relation *V*, the intersection of all the invariant relations of p3, satisfies the sufficiency condition:

$$R\bar{T} \cap VL \cap \left(V \cup R \cap \hat{\bar{T}}\right) = V.$$

This has been proved using Mathematica; details of the proof are given at http://selab.njit.edu/icst2016proof.

E. Initialized While Loops

As a second illustrative example, we consider the following program that purports to compute Fibonacci numbers; its space is defined by the following declarations:

```
const int cN = …;
int i, j, fb, nc, np;
```

The source code of the loop w is:

```
while (j!=cN)
     {j=j+i;  nc=fb;  i=i+1;
      fb=np+nc;  np=nc; j=j-i;}
```

Deployment of the invariant relations generator produces the following invriant relations (where *F* is the Fibonacci function):

- $V_1 = \{(s, s') | \ i \leq i'\}$.
- $V_2 = \{(s, s') | j \geq j'\}$.
- $V_3 = \{(s, s') | i + j = i' + j'\}$.
- $V_4 = \{(s, s') | np' = fb \times F(i' - i) + np \times F(i' - i - 1)\}$.
- $V_5 = \{(s, s') | fb' = fb \times F(i' - i + 1) + np \times F(i' - i)\}$.

We consider the following specification:

$$R = \{(s, s') | \ j > cN \wedge fb' = F(j + 2 - cN) \wedge$$
$$nc' = F(j + 1 - cN) \wedge np' = F(j + 1 - cN) \wedge$$
$$i' = i + j \wedge \ j' = cN\}.$$

In the table below, we show how the invariant relations listed above are classified between compatible relations and incompatible relations with respect to specification R.

Compatible	Incompatible
$V_1, V_2, V_3,$	V_4, V_5

Because we have found invariant relations that are incompatible with specification R, we infer that this loop is incorrect with respect to R; hence there is a fault.

A theorem by H.D. Mills [4] provides a condition under which a function W can be computed by an uninitialized while loop:

$$(LW \cap I)W = (LW \cap I).$$

In [5] we generalize this result to give a condition on a relation *R* to admit an uninitialized while loop as a correct program (i.e. a condition under which specification *R* can be refined by a function *W* that satisfies Mills' condition, above):

$$RL \subseteq R(R \cap I)L.$$

Interestingly, we find that our relation *R* given above does not satisfy this condition. Indeed, we find:

$$RL = \{(s, s') | j > cN\}.$$

On the other hand, we find

$$R \cap I = \{(s, s') | \ s' = s \wedge \ j > cN \wedge fb' = F(j + 2 - cN) \wedge \ nc' = F(j + 1 - cN) \wedge np' = F(j + 1 - cN) \wedge i' = i + j \wedge \ j' = cN\}.$$

This relation is empty, since it is a subset of

$$\{(s, s') | \ j > cN \wedge \ j = cN\},$$

which is itself empty. Hence $R(R \cap I)L$ is empty, and the condition

$$RL \subseteq R(R \cap I)L$$

does not hold. So that specification *R* cannot be satisfied by an unitialized while loop; in other words, even though w is incorrect with respect to *R* (as shown by the existence of incompatible relations), there is nothing we can do to w to correct it; instead, any correction must be outside the loop, say in the initialization. In light of this example, we may want to refine the algorithm discussed above (in section IV-C) by adding a step where we check the condition ($RL \subseteq R(R \cap I)L$) before attempting to remedy the loop; indeed, if this condition is not satisfied, then no loop can satsify specification R, hence the focus of fault removal ought to divert away from the loop (e.g. towards its initialization).

To get some guidance for how to initialize this loop, we compute its competence domain with respect to R. To this effect, we calculate the function of w from its invariant relations using a formula provided by [3]; this calculation is done automatically, using the computer algebra program Mathematica (© Wolfram Research). We find:

$$W = \{(s,s')|\ j \geq cN \wedge i' = i + j - cN \wedge j' = cN \wedge np' = np \times F(j - cN - 1) + fb \times F(j - cN) \wedge nc' = np' \wedge fb' = np \times F(j - cN) + fb \times F(j - cN + 1)\}.$$

The competence domain of w can be computed in Mathematica by simplifying the following logical expression (where each relation is represented by its characteristic predicate):

$\exists s'\colon\ R(s,s') \wedge W(s,s')$.

We find:

$$CD = \{(s,s')|\ j > cN \wedge \left((fb = 1 \wedge np = 1) \vee \left(fb \times \left(1+\sqrt{5}\right) + 2 \times np = 3 + \sqrt{5}\right)\right)\}.$$

Because variables fb and np are of type integer, this competence domain can be written simply as:

$CD = \{(s,s')|\ j > cN \wedge fb = 1 \wedge np = 1\}$.

In order for w to behave according to specification R, variables fb and np have to be 1; this suggests that the required initialization is

`fb=1; np=1;`

We find (as shown below) that these initializations ensure that the program is now correct with respect to R. Interestingly, we also find that doing only one of these two initializations produces more-correct (albeit not absolutely correct) programs, as we show below.

Let $p1$ be the program obtained from w by adding the initialization **`fb=1;`**. We find

$P1 = \{(s,s')|\ j \geq cN \wedge i' = i + j - cN \wedge j' = cN \wedge np' = np \times F(j - cN - 1) + F(j - cN) \wedge fb' = np \times F(j - cN) + F(j - cN + 1) \wedge nc' = np'\}$,

From which we infer the competence domain of $p1$ as:

$CD1 = \{(s,s')|\ j > cN \wedge np = 1\}$.

Likewise, we compute the function then competence domain of $p2$, obtained by adding **`np=1;`** to the while loop, and we find:

$P2 = \{(s,s')|\ j \geq cN \wedge i' = i + j - cN \wedge j' = cN \wedge np' = F(j - cN - 1) + fb \times F(j - cN) \wedge fb' = F(j - cN) + fb \times F(j - cN + 1) \wedge nc' = np'\}$,

Whence,

$CD2 = \{(s,s')|\ j > cN \wedge fb = 1\}$.

Finally, we compute the function and competence domain of the program p3 obtained from w by adding the two initializations, **`fb=1; np=1;,`** and we find

$P3 = \{(s,s')|\ j \geq cN \wedge i' = i + j - cN \wedge j' = cN \wedge np' = F(j - cN + 1) \wedge fb' = F(j - cN + 2) \wedge nc' = np'\}$,

Whence

$CD3 = \{(s,s')|\ j > cN\}$.

Hence to summarize:

- $CD = \{(s,s')|j > cN \wedge fb = 1 \wedge np = 1\}$.
- $CD1 = \{(s,s')|j > cN \wedge np = 1\}$.
- $CD2 = \{(s,s')|j > cN \wedge fb = 1\}$.
- $CD3 = \{(s,s')|j > cN\}$.
- $RL = \{(s,s')|j > cN\}$.

This is reflected in the following figure:

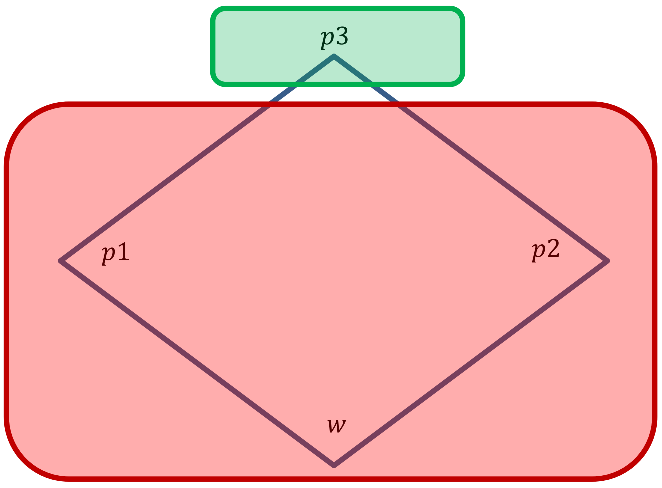

Figure 3. Ranking Candidates by Relative Correctness

V. CONCLUDING REMARKS

In this section, we summarize our main contributions in this paper, discuss related work, then give a candid assessment of this work and its future prospects as we envision them.

A. *Summary*

In this paper we argue that it is possible to remove faults from a program and build an argument to the effect that the program is now better for it. We argue that relying exclusively on testing to ensure fault removal carries some inherent risks, especially when each fault is treated as if it were the last fault of the program. To support our argument, we introduce the concept of relative correctness, i.e. the property of a program to be more-correct than another with respect to a specification, and show how one can conceivably prove relative correctness without getting involved in all the minute details of the two programs in question.

Specifically, we focus on a framework based on invariant relations, which allows us to support all the phases of fault removal, including:

- Evidence of incorrectness (hence the need to locate and remove a fault).
- Localisation of the fault, by identifying the variables whose assignments may have to be modified.
- Guidance in how to change the code, by formulating a constraint that ensures correctness enhancement.
- Proof that the fault has indeed been removed, i.e. that the program will experience no regression in its behavior.

Even though we do not, yet, have an integrated tool that supports this work, many of its complex steps (such as generation of invariant relations, verification of logical properties) are automated. The generation of invariant relations is discussed in other publications, including [3] and an online description of the prototype tool can be found at https://selab.njit.edu/tools/fxloop.php. Invariant relations are generated from a denotational semantic representation of the loop, by matching this representation against pre-stored code patterns and instantiating the corresponding invariant relation patterns. As for verifying logical properties that arise in our analysis, we typically use Mathematica.

B. Related Work

In [6] Logozzo et al. introduce a technique for extracting and maintaining semantic information across program versions: specifically, they consider an original program P and a variation (version) P' of P, and they explore the question of extracting semantic information from P, using it to instrument P' (by means of executable assertions), then pondering what semantic guarantees they can infer about the instrumented version of P'. The focus of their analysis is the condition under which programs P' and P can execute without causing an abort (due to attempting an illegal operation), which they approximate by sufficient conditions and necessary conditions. They implement their approach in a system called VMV (*Verification Modulo Versions*) whose goal is to exploit semantic information about P in the analysis of P', and to ensure that the transition from P to P' happens without regression; in that case, they say that P' is *correct relative to* P. The definition of relative correctness of Logozzo et al [6] is different from ours, for several reasons: whereas [6] talk about relative correctness between an original program and a subsequent version in the context of adaptive maintenance (where P and P' may be subject to distinct requirements), we talk about relative correctness between an original (faulty) software product and a revised version of the program (possibly still faulty yet more-correct) in the context of corrective maintenance with respect to a fixed requirements specification; whereas [6] use a set of assertions inserted throughout the program as a specification, we use a relation that maps initial states to final states to specify the standards against which absolute correctness and relative correctness are defined; whereas Logozzo et al. represent program executions by execution traces (snapshots of the program state at assertion sites), we represent program executions by functions mapping initial states into final states; finally, whereas Logozzo et al define a successful execution as a trace that satisfies all the relevant assertions, we define it as an initial state/ final state pair that falls within the relational specification.

In [7] Lahiri et al. introduce a technique called *Differential Assertion Checking* for verifying the relative correctness of a program with respect to a previous version of the program. Lahiri et al. explore applications of this technique as a tradeoff between soundness (which they concede) and lower costs (which they hope to achieve). Like the approach of Logozzo et al. [6] (from the same team), the work of Lahiri uses executable assertions as specifications, represents executions by traces, defines successful executions as traces that satisfy all the executable assertions, and targets abort-freedom as the main focus of the executable assertions. Also, they define relative correctness between programs P and P' as the property that P' has a larger set of successful traces and a smallest set of unsuccessful traces than P; and they introduce relative specifications as specifications that capture functionality of P' that P does not have. By contrast, we use input/ output (or initial state/ final state) relations as specifications, we represent program executions by functions from initial states to final states, we characterize correct executions by initial state/ final state pairs that belong to the specification, and we make no distinction between abort-freedom (a.k.a. safety, in [7]) and normal functional properties. Indeed, for us the function of a program is the function that the program defines between its initial states and its final states; the domain of this function is the set of states for which execution terminates normally and returns a well-defined final state. Hence execution of the program on a state s is abort free if and only if the state is in the domain of the program function; the domain of the program function is part of the function rather than being an orthogonal attribute; hence we view abort-freedom as a special form of functional attribute, rather than being an orthogonal attribute. Another important distinction with [7] is that we do not view relative correctness as a compromise that we accept as a substitute for absolute correctness; rather we argue that in many cases, we ought to test programs for relative correctness rather than absolute correctness, regardless of cost. In other words, whereas Lahiri et al. argue in favor of relative correctness on the grounds that it optimizes a quality vs. cost ratio, we argue in favor on the grounds that it optimizes quality.

In [8], Logozzo and Ball introduce a definition of relative correctness whereby a program P' is correct relative to P (*an improvement* over P if and only if P' has more good traces and fewer bad traces than P Programs are modeled with trace semantics, and execution traces are compared in terms of executable assertions inserted into P and P'; in order for the comparison to make sense, programs P and P' have to have the same (or similar) structure and/or there must be a mapping from traces of P to traces of P'. When P' is obtained from P by a transformation, and when P' is provably correct relative to P, the transformation in question is called a *verified repair*. Logozzo and Ball introduce an algorithm that specializes in deriving program repairs from a predefined catalog that is targeted to specific program constructs, such as: contracts, initializations, guards, floating point comparisons, etc. Like the work cited above [6], [7], Logozzo and Ball model programs by execution traces and distinguish between two types of failures: contract violations, when functional properties are not satisfied; and run-time errors, when the execution causes an abort; for the reasons we discuss above, we do not make this distinction, and model the two aspects with the same relational framework. Logozzo and Ball deploy their approach in an automated tool based on the static analyzer cccheck, and assess their tool for effectiveness and efficiency.

In [9] Nguyen et al. present an automated repair method based on symbolic execution, constraint solving, and program synthesis; they call their method SemFix, on the grounds that it performs program repair by means of semantic analysis. This method combines three techniques: fault isolation by means of statistical analysis of the possible suspect statements; statement-level specification inference, whereby a local specification is inferred from the global specification and the product structure; and program synthesis, whereby a corrected statement is computed from the local specification inferred in the previous step. The method is organized in such a way that program synthesis is modeled as a search problem under constraints, and possible correct statements are inspected in the order of increasing complexity. When programs are repaired by SemFix, they are tested for (absolute) correctness against some predefined test data suite; as we argue throughout this paper, it is not sensible to test a program for absolute correctness after a repair, unless we have reason to believe that the fault we have just repaired is the last fault of the program (how do we ever know that?). By advocating to test for relative correctness, we enable the tester to focus on one fault at a time, and ensure that other faults do not interfere with our assessment of whether the fault under consideration has or has not been repaired adequately.

In [10], Weimer et al. discuss an automated program repair method that takes as input a faulty program, along with a set of positive tests (i.e. test data on which the program is known to perform correctly) and a set of negative tests (i.e. test data on which the program is known to fail) and returns a set of possible patches. The proposed method proceeds by keeping track of the execution paths that are visited by successful executions and those that are visited by unsuccessful executions, and using this information to focus the search for repairs on those statements that appear in the latter paths and not in the former paths. Mutation operators are applied to these statements and the results are tested again against the positive and negative test data to narrow the set of eligible mutants.

Whereas the definitions of relative correctness, faults and fault removal are introduced in [11], the potential impact of these concept on software engineering is discussed in [12].

C. Assessment and Prospects

This work is clearly in its infancy; it includes the definition of a new concept, the premise that this concept can be used for a provably monotonic fault removal process, and some initial results that enable us to apply this concept with some automated support, and without getting involved into the minute functional details of the program and the specification.

The question that arises with this type of work is, of course, whether it scales up to programs of realistic size and complexity. We argue that relative correctness scales up to the same degree as absolute correctness. The fact that it cannot be readily employed to software products of arbitrary size and complexity does not make it any less worthy of investigation, just as the same constraints do not make absolute correctness less worthy of study; it is still useful as a logical reasoning framework; and it can be applied in practice with the proper balance of formality, expressiveness, and usability, and with judicious automated support where possible. Also, we argue that in software quality assurance as in other endeavors, the law of diminishing returns advocates the use of diverse methods and tools to maximize impact; the use of relative correctness to support fault diagnosis and removal stands to play an important role as a tool in the engineer's toolbox.

We envision to continue exploring applications of relative correctness in fault removal, to enhance and integrate our tool support, and to consider other results (theorems) that enable us to streamline the verification of relative correctness.

VI. Bibliography

[1] J. Desharnais, N. Diallo, W. Ghardallou, M. F. Frias, A. Jaoua and A. Mili, "Relational Mathematics for Relative Correctness," in *Relational and Algebraic Methods in Computer Science*, Braga, Portugal, 2015.

[2] A. C. Hoare, "An Axiomatic Basis of Computer Programming," *Communications of the ACM,* pp. 576-580, 1969.

[3] A. Louhichi, W. Ghardallou, K. Bsaies, L. J. Labed, O. Mraihi and A. Mili, "Verifying While Loops with Invariant Relations," *IJCCBS,* vol. 5, no. 1/2, pp. 78-102, 2014.

[4] H. D. Mills, "The New Math of Computer Programming," *Communications of the ACM,* vol. 18, no.

1, pp. 43-48, January 1975.

[5] A. Mili, J. Desharnais and f. Mili, "Relational Heuristics for the Design of Deterministic programs," *Acta Informatica*, vol. 24, no. 3, pp. 239-276, 1987.

[6] F. Logozzo, S. Lahiri, M. Faehndrich and S. Blackshear, "Veriifcation Modulo Versions: Towards Usable Verification," in *PLDI*, 2014.

[7] S. K. Lahiri, K. L. McMillan, R. Sharma and C. Hawblitzel, "Differential Assertion Checking," in *ESEC/ SIGSOFT FSE*, 2013.

[8] F. Logozzo and T. Ball, "Modular and Verified Automatic Program Repair," in *OOPSLA*, 2012.

[9] T. H. D. Nguyen, D. Qi, A. Roychudhury and S. Chandra, "SemFix: Program Repair via Semantic Analysis," in *ICSE*, 2013.

[10] W. Weimer, T. Ngyuen, C. Le Gouess and S. Forrest, "Automatically Finding Patches Using Genertic Programming," in *ICSE*, 2009.

[11] A. Mili, M. F. Frias and A. Jaoua, "On Faults and Faulty Programs," in *Relational and Algebraic Methods in Computer Science*, Marienstatt, Germany, 2014.

[12] N. Diallo, W. Ghardallou and A. Mili, "Correctness and Relative Correctness," in *Proceedings, ICSE*, Firenze, Italy, 2015.

[13] C. Brink, W. Kahl and G. Schmidt, Relational Methods in Computer Science, Berlin, Germany: Springer Verlag, 1997.