What is a Fault? And Why Does It Matter?

Nafi Diallo[1], Wided Ghardallou[2], Jules Desharnais [3], Marcelo Frias[4], Ali Jaoua[5] and Ali Mili[1]
[1]: NJIT, Newark NJ 07102-1982, USA. [2]: University of Tunis El Manar, Tunis, Tunisia
[3]: Laval University, Quebec, Canada, [4]: Marcelo Frias, ITBA, Buenos Aires, Argentina,
[5]: Qatar University, Doha, Qatar

February 12, 2015

Abstract

In the process of trying to define what is a software fault, we have found it necessary to introduce a concept of relative correctness, i.e., the property of a program to be more-correct than another with respect to a given specification. In this paper, we introduce a definition of relative correctness, discuss its properties, and explore its impact and applications in many fault-related software engineering activities, such as: software testing, monotonic fault removal, software repair, mutation testing, and software design. Disserting on the virtues of relative correctness is an interesting exercise, but it would also be a futile exercise if we did not know how to establish that a program is more-correct than another; in this paper, we also discuss how to test two programs for relative correctness, and how to prove by static analysis that a program is more-correct than another with respect to a given specification.

Keywords

Correctness, Partial Correctness, Total Correctness, Relative Correctness, Absolute Correctness, Software Fault, Fault Removal, Software Testing, Software Repair, Software Design.

1 Motivation and Background

1.1 The Trouble with Faults

In [2, 12–14] Laprie et al. define the hierarchy of faults, errors and failures as part of the conceptual basis of dependable computing. In this hierarchy, faults are defined as *the adjudged or hypothesized cause of an error* [2]; we argue that, as far as software is concerned, this definition is not sufficiently precise, first because adjudging and hypothesizing are highly subjective human endeavors, and second because the concept of error is itself insufficiently defined, since it depends on a detailed characterization of correct system states at each stage of a computation (which is usually unavailable). We further argue that a formal/unambiguous definition of faults is indispensable, given that faults play a crucial role in the study of software dependability, that they are the basis of the classification of methods of dependability (fault avoidance, fault removal, fault tolerance), and that they play an important role in several software engineering concepts, such as fault density, fault proneness, and fault forecasting. But defining software faults is fraught with difficulties:

- *Discretionary determination.* Usually we determine that a program part is faulty because we think we know what the designer intended to achieve in that particular part, and we find that the program does not fulfill the designer's intent; clearly, this determination is only as good as our assumption about the designer's intent.

- *Contingent determination*. The same faulty behavior of a software product may be repaired in more than one way, possibly involving more than one location; hence the determination that one location is a fault is typically contingent upon the assumption that other parts are not in question.

- *Tentative determination*. The determination that a program part is faulty is usually made in conjunction with a substitution that would presumably repair the program; clearly, this determination is valid only to the extent that the substitution is an adequate repair.

- *Inconclusive determination*. Usually, we determine that a fault has been removed from a program if upon substituting the allegedly faulty part by an allegedly correct part, we find that the program runs satisfactorily on some test data T. In fact, the successful execution of the program on test data T is neither a necessary condition nor a sufficient condition to the actual removal of the fault.

In order to overcome the difficulties raised above, we resolve to proceed as follows:

- We introduce a concept of *relative correctness*, i.e. the property of a program to be more correct than another program with respect to a specification [21].

- We define a fault in a program as any program part (be it a simple statement, a lexical token, an expression, a compount statement, a block of statements, a set of non-contiguous statements, etc.) for which there exists a substitution that would make the program more-correct than the original with respect to a relevant specification [21].

With such a definition, we address all the difficulties raised above, namely:

- *A Fault as an Intrinsic Attribute*. The definition of a fault is not dependent on any design assumptions, but involves only the (incorrect) program, the faulty program part, and the specification with respect to which correctness (and failure) is defined.

- *A Fault as a Definite Property*. If we let a fault be any program part that admits a substitution that makes the program more-corect, then the designation of a fault is no longer contingent on any hypothesis; we need not make any assumption on whether other parts of the program are faulty or not.

- *A Fault as an Opportunity for Correctness Enhancement*. By definition, every fault represents an opportunity to make the program more-correct, i.e. closer to being correct; the challenge of the tester is to find an appropriate substitution, knowing that one does exist.

- *Fault Removal as a Verifiable Process*. Whether a fault has been removed is not dependent on the program's behavior on some (partial) test data, but rather on a formally verifiable property.

In order to reap all these benefits, we must now introduce a definition of relative correctness; this is the subject of section 3. In preparation for this objective, we present some mathematical definitions and notations, and some elements of relations-based program semantics in section 2.2. In section 4, we consider in turn several properties that we would want a concept of relative correctness to satisfy, and we prove that our proposed definition does satisfy all of them; the goal of this section is to give the reader a measure of confidence in the soundness of the proposed definition, as a prelude to the subsequent discusions. In section 5, we discuss the uses of the concept of relative correctness, and its implications for relevant software processes. Once we know how to use relative correctness, the next issue we wish to address is: how do we establish relative correctness, i.e. how to build the case that a program is more-correct than another with respect to a specification; this is the subject of section 6. Section 7 summarizes and assesses our findings, then sketches directions of future research.

I would say it is necessary as long as the tests are assumed correct

2 Mathematics for Program Analysis

2.1 Relational Notations

In this section, we introduce some elements of relational mathematics that we use in the remainder of the paper to carry out our discussions. Dealing with programs, we represent sets using a programming-like notation, by introducing variable names and associated data type (sets of values). For example, if we represent 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 in lower case s, and are triplets of elements of X, Y, and Z. Given an element s of S, we represent its X-component by $x(s)$, its Y-component by $y(s)$, and its Z-component by $z(s)$. When no risk of ambiguity exists, we may write x to represent $x(s)$, and x' to represent $x(s')$, letting the references to s and s' be implicit.

A (binary) relation on S is a subset of the Cartesian product $S \times S$; given a pair (s, s') in R, we say that s' is an *image* of s by R. 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

$$\widehat{R} = \{(s, s')|(s', s) \in R\}.$$

Finally, 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 $\widehat{R}$.

A relation R is said to be *reflexive* if and only if $I \subseteq R$, *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 *deterministic* (or, a *function*) if and only if $\widehat{R}R \subseteq I$. Notice that two functions f and f' are identical if and only if $f \subseteq f'$ and $f'L \subseteq fL$. A relation R is said to be a *vector* if and only if $RL = R$; a vector on space S is a relation of the form $R = A \times S$, for some subset A of S; we use vectors to represent subsets of S, and we may by abuse of notation write $s \in R$ to mean $s \in A$; in particular, we use the product RL as a relational representation of the domain of R.

2.2 Relational Semantics

For the purposes of our discusions, we consider a simple programming notation that includes variable declarations, in the syntax discussed above, as well as a number of C-like executable statements. The semantics of variables declarations allows us to define the state space of the program, in the way we discussed above; as for the semantics of executable statements, we define them by means of a relation that captures the effect of the execution on the state of the program. Given a program or program part p, we let its semantics be represented by $[p]$ (or by upper case P) and defined by:

$$[g] = \{(s, s')| \text{ if execution of } g \text{ starts in state } s \text{ then it terminates normally in state } s'\}.$$

We present the following executable statements, along with their semantic definition.

- *Assignment Statements* have the form $\mathtt{s} = \mathtt{E(s)}$, where s is a shorthand for the program variables, and $\mathtt{E(s)}$ is an expression that involves the program variables. We define the semantics of assignment statements as follows:
  $[\mathtt{s} = \mathtt{E(s)}] \equiv \{(s, s') | s \in def(\mathtt{E}) \wedge s' = E(s)\}$,
  where $def(E)$ is the set of states where expression $E(s)$ can be evaluated.
- *Sequence* has the form $\mathtt{g1}; \mathtt{g2}$; its semantics is defined by:
  $[\mathtt{g1}; \mathtt{g2}] \equiv [\mathtt{g1}] \circ [\mathtt{g2}]$.
- *Alternation* has the form $\mathtt{if}\ (\mathtt{t})\ \{\mathtt{g1}\}\ \mathtt{else}\ \{\mathtt{g2}\}$, and the following semantic definition:
  $[\mathtt{if}\ (\mathtt{t})\ \{\mathtt{g1}\}\ \mathtt{else}\ \{\mathtt{g2}\}] \equiv (T \cap [\mathtt{g1}]) \cup (\overline{T} \cap [\mathtt{g2}])$,
  where $T = \{(s, s') | t(s)\}$.
- *Conditional* has the form $\mathtt{if}\ (\mathtt{t})\ \{\mathtt{g1}\}$ and the following semantic definition:
  $[\{\mathtt{if(t)\{g1\}}\}] \equiv (T \cap [g1]) \cup (\overline{T} \cap I)$.
- *Iteration* has the form $\mathtt{while}\ (\mathtt{t})\ \{\mathtt{b}\}$ and the following semantic definition:
  $[\{\mathtt{while}\ (\mathtt{t})\ \{\mathtt{b}\}\}] \equiv (T \cap [\mathtt{b}])^* \cap \widehat{\overline{T}}$.
- The *skip* statement is written as `skip` and its semantics is defined as the identity relation I on S.

It is clear from the foregoing discussions that the semantic definition of a program written in the notation introduced therein is a deterministic relation, i.e., a function, which we call the program's *function*. While this may affect the generality of our study, we do restrict our investigation to deterministic programs, and discuss in section 7 how we envision to lift this restriction.

As a notational convention, we use lower case letters (possibly indexed) to represent programs, and we use the same letters in upper case to represent the relational semantic denotation of these programs. For the sake of readability, we may sometimes identify a program with its function, i.e., use the program and its function interchangeably.

A proper definition should first define the semantics for terms and appeal to that definition at this point.

2.3 Refinement Ordering

The concept of refinement is at the heart of any programming calculus; the exact definition of refinement (the property of a specification to refine another) varies from one calculus to another; the following definition captures our concept of refinement.

Definition 1 *We let R and R' be two relations on space S. We say that R* refines *R' if and only if*

$$RL \cap R'L \cap (R \cup R') = R'.$$

We write this relation as: $R \sqsupseteq R'$ or $R' \sqsubseteq R$. Intuitively, $R \sqsupseteq R'$ if R is more deterministic than R' inside the domain of R' (although it might have a bigger domain). The following proposition provides an important property of refinement.

Proposition 1 *The refinement relation is a partial ordering between relations on a space S.*

For the sake of readability, we do not include the proof of this proposition here, but place it in the appendix instead. Because the refinement relation is a partial ordering, we may refer to it as the *refinement ordering*. The following proposition provides simple properties of refinement in two special cases.

Proposition 2 *Let R and R' be two relations on set S.*

- *If R and R' have the same domain, then $R \sqsupseteq R'$ if and only if $R \subseteq R'$.*
- *If R and R' are deterministic, then $R \sqsupseteq R'$ if and only if $R \supseteq R'$.*

Proof. If R and R' have the same domain, then the condition of refinement can be written as: $(R \cup R') = R'$, which means $R \subseteq R'$. To prove the second clause of the proposition, we consider again the lemma introduced in the proof of proposition 1. The proof of sufficiency is trivial: if R and R' are functions and $R \supseteq R'$ then $R' = R'L \cap R$. As for the proof of necessity, let R and R' be functions such that $R \sqsupseteq R'$, and let R'' be defined as $R'L \cap R$. By hypothesis, $R'' \subseteq R'$; on the other hand,

$$
\begin{array}{ll}
 & R'L \\
\subseteq & \text{by hypothesis, and by construction} \\
 & RL \cap R'L \\
= & \text{by construction} \\
 & R''L.
\end{array}
$$

Hence $R'' = R'$, from which we infer (by construction of R'') that $R' \subseteq R$. **qed**

2.4 Refinement Lattice

Since refinement is a partial ordering between specifications, it is legitimate to ponder its lattice-like properties. Let $\mathbb{R} = \langle \mathcal{R}, \sqsubseteq \rangle$ be a structure in which $\mathcal{R}$ is the set of specifications on some space S, and $\sqsubseteq$ is the 'is-refined-by' relation. Then, from proposition 1, $\mathbb{R}$ is a partial ordering. The following Proposition, due to [**?**], summarizes the main findings with regards to the lattice of relational specifications.

Proposition 3

- *Any two specifications R and R' admit a greatest lower bound by the refinement ordering, denoted by $R \sqcap R'$ (R* meet *R') and defined by:*

$$R \sqcap R' = RL \cap R'L \cap (R \cup R') \; .$$

- *Any two specifications R and R' that satisfy the following condition (called the* consistency condition*) admit a least upper bound, denoted by $R \sqcup R'$ (R* join *R') and defined by:*

$$R \sqcup R' = (\overline{R'L} \cap R) \cup (\overline{RL} \cap R') \cup (R \cap R') \; .$$

missing consistency condition

Interpretation: The meet of two specifications represents the requirements information that they have in common; the consistency condition between two specifications is the condition under which the specifications admit a joint refinement; the join of two specifications represents the sum of all the requirements of each term. The join of R and R' represents all the requirements of R, all the requirements of R', and nothing else; it is possible to combine the requirements of R and R' only if they do not contradict each other, whence the consistency condition. The join is a natural operator to compose complex specifications from simpler ones.

3 Absolute Correctness and Relative Correctness

Whereas absolute correctness characterizes a program with respect to a specification, relative correctness ranks two programs with respect to a specification; in order to discuss the latter, it helps to review the former, to see how it is defined in our notation.

3.1 Absolute Correctness

We use the refinement ordering introduced in this section to define program 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 (the function defined by program p on space S) refines R.*
- *We say that program p is* partially correct *with respect to specification R if and only if P refines $R \cap PL$.*

Whenever we want to contrast correctness with partial correctness, we may refer to it as *total correctness*. This definition is consistent with traditional definitions of partial and total correctness [4, 7–9, 20]. The following proposition gives a simple characterization of correctness, and sets the stage for the definition of relative correctness.

Proposition 4 *Program g is correct with respect to specification R if and only if $(G \cap R)L = RL$.*

Proof. Proof of necessity: The condition $(G \cap R)L \subseteq RL$ stems readily from set theory, hence we focus on proving the condition $RL \subseteq (G \cap R)L$. Given that g is correct with respect to R, we know that G refines R. By virtue of the lemma introduced in the proof of proposition 1, we have the hypotheses: $RL \subseteq GL$ and $RL \cap G \subseteq R$. Let s be an element of the domain of R; by the first clause, it is necessarily an element of the domain of G. By virtue of the second clause, $(s, G(s))$ is necessarily an element of R. Because $(s, G(s))$ is an element of R and G (by definiton), it is an element of $(G \cap R)$; hence s is an element of the domain of $(G \cap R)$.

Proof of sufficiency: From $RL = (G \cap R)L$ (hypothesis) and $(G \cap R)L \subseteq GL$ (set theory) we infer $RL \subseteq GL$. Let (s, s') be an element of $(RL \cap G)$; then s is in the domain of R and $s' = G(s)$. By hypothesis, we know that s is in the domain of $(G \cap R)$, which means that $(s, G(s))$ is in R. Since $G(s) = s'$, we infer that (s, s') is in R. **qed**

I would go for a purely relational proof

In [23], Mills et al. define correctness of a program p with function P with respect to the specification R, by the formula $(R \cap P)L = RL$; hence this proposition is inspired by their definition (though for us it is a proposition rather than a definition because we define correctness by means of refinement). Note that we could likewise characterize partial correctness by the formula: $(R \cap P)L = RL \cap PL$; but since relative correctness if a generalization of (total) correctness rather than partial correctness, proposition 4 only talks about (total) correctness.

3.2 Relative Correctness

Definition 3 *Let R be a specification on space S and let p and p' be two programs on space S whose functions are respectively P and P'.*

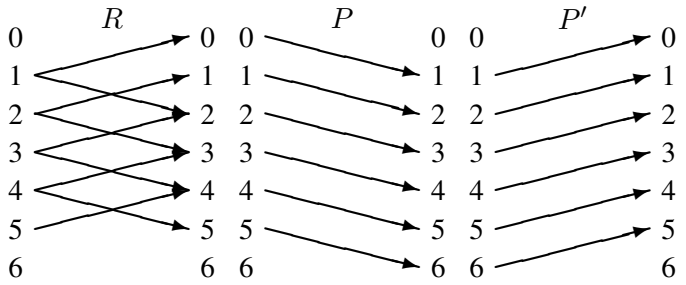

Figure 1: Enhancing Correctness Without Duplicating Behavior: p' is more-correct than p with respect to R

- *We say that program p' is* more-correct *than program p with respect to specification R (abbreviated by: $P' \sqsupseteq_R P$) 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 (abbreviated by: $P' \sqsupset_R P$) if and only if* $(R \cap P')L \supset (R \cap P)L$.

Whenever we want to contrast correctness (given in Definition 2) with relative correctness, we may refer to it as *absolute correctness*. Note that when we say *more-correct* we really mean *more-correct or as-correct-as*; we use the shorthand, however, for convenience. We give a simple intuitive interpretation of this definition: The relation (actually a vector) $(R \cap P)L$ represents the set of initial states for which program p behaves according to the requirements of specification R (we refer to this set as the *competence domain* of program p with respect to specification R); so that to be more-correct merely means to have a larger competence domain, i.e. to behave according to the specification on a larger portion of the domain of the specification. Note that in order for program p' to be more-correct than program p, it does not have to duplicate the behavior of p over the competence domain of p: It may have a different behavior (since R is potentially non-deterministic) provided this behavior is also correct with respect to R; see Figure 1. 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.

As an illustrative example of this definition, we consider the space S defined by two integer variables x and y, and we let R be the following specification on S:

$$R = \{(s, s') | x^2 \leq x'y' \leq 2x^2\}.$$

We consider the following candidate programs, denoted p_0 through p_7. For each program p, we compute the function of the program P, then the program's competence domain with respect to R, i.e., $(P \cap R)L$.

p_0: {`x=1; y=-1;`}.
We find: $P_0 = \{(s, s') | x' = 1 \wedge y' = -1\}$. Whence,

$$(P_0 \cap R)L = \{(s, s') | \exists s' : x' = 1 \wedge y' = -1 \wedge x^2 \leq -1 \leq 2x^2\} = \phi.$$

p_1: {`x=2*x; y=0;`}.
We find: $P_1 = \{(s, s') | x' = 2x \wedge y' = 0\}$. Whence,

$$(P_1 \cap R)L = \{(s, s') | \exists s' : x' = 2x \wedge y' = 0 \wedge x^2 \leq 0 \leq 2x^2\} = \{(s, s') | x = 0\}.$$

p_2: {x=x*x; y=0;}.
We find: $P_2 = \{(s, s') | x' = x^2 \wedge y' = 0\}$. Whence,

$$(P_2 \cap R)L = \{(s, s') | \exists s' : x' = x^2 \wedge y' = 0 \wedge x^2 \leq 0 \leq 2x^2\} = \{(s, s') | x = 0\}.$$

p_3: {x=2*x; y=1;}.
We find: $P_3 = \{(s, s') | x' = 2x \wedge y' = 1\}$. Whence,

$$(P_3 \cap R)L = \{(s, s') | \exists s' : x' = 2x \wedge y' = 1 \wedge x^2 \leq 2x \leq 2x^2\} = \{(s, s') | 0 \leq x \leq 2\}.$$

p_4: {x=2*x; y=2;}.
We find: $P_4 = \{(s, s') | x' = 2x \wedge y' = 2\}$. Whence,

$$(P_4 \cap R)L = \{(s, s') | \exists s' : x' = 2x \wedge y' = 2 \wedge x^2 \leq 4x \leq 2x^2\} = \{(s, s') | x = 0 \vee 2 \leq x \leq 4\}.$$

p_5: {x=2*x; y=x/2;}.
We find: $P_5 = \{(s, s') | x' = 2x \wedge y' = x\}$. Whence,

$$(P_5 \cap R)L = \{(s, s') | \exists s' : x' = 2x \wedge y' = x \wedge x^2 \leq 2x^2 \leq 2x^2\} = L.$$

p_6: {y=x/2; x=2*x;}.
We find: $P_6 = \{(s, s') | x' = 2x \wedge y' = x/2\}$. Whence,

$$(P_6 \cap R)L = \{(s, s') | \exists s' : x' = 2x \wedge y' = x/2 \wedge x^2 \leq x^2 \leq 2x^2\} = L.$$

p_7: {x=x*x; y=2;}.
We find: $P_7 = \{(s, s') | x' = x^2 \wedge y' = 2\}$. Whence,

$$(P_7 \cap R)L = \{(s, s') | \exists s' : x' = x^2 \wedge y' = 2 \wedge x^2 \leq 2x^2 \leq 2x^2\} = L.$$

Figure 2 shows how these candidate programs are ranked with respect to relative correctness. This example illustrates a number of properties:

- Note that this relation is not antisymmetric; so that two programs may be mutually related and still be distinct (such is the case for P_1 and P_2; as it the case for P_5, P_6 and P_7).
- It is well understood that this relation is reflexive and transitive; hence we do not need to reprsent self-arcs, nor transitive arcs.
- The top of the graph represents the programs that are (absolutely) correct with respect to specification R: P_5, P_6 and P_7. They are more-correct than all the other programs.
- Note that when a program is more-correct than another, it has a larger competence domain; but it does not necessarily duplicate its behavior on the smaller competence domain (due to the non-determinacy of R). Hence, for example, p_3 is more-correct than p_1 because it has a larger competence domain ($\{(s, s') | 0 \leq x \leq 2\}$ vs $\{(s, s') | x = 0\}$); yet p_3 does not behave as p_1 on the competence domain of p_1. In other words, to be more-correct than a program p does not mean to preserve the correct behavior

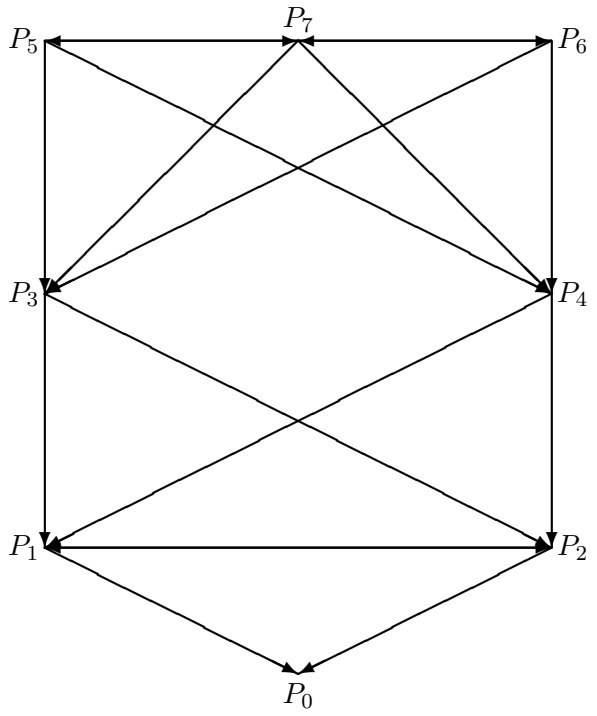

Figure 2: Relative Correctness Relations

of p and behave correctly outside the competence domain of p; it may mean having a different correct behavior on the competence domain of p.

3.3 Faults and Fault Removal

Now that we have a definition for relative correctness, we are ready to define faults, and to characterize monotonic fault removal (i.e., fault removal that makes the program provably better, rather than to make the program work successfully for some inputs only to find that this was done at the expense of other inputs). In these definitions, we use the term *program part* to refer to any set of lexemes of the program's source code; these may range from a single lexeme (e.g. a comparison operator) to an expression to a simple statement to a compound a statement to a set of non-contiguous lexemes or statements spread across the source code of the program.

Definition 4 Faults, and Fault Removals. *Let p be a program on space S and R be a specification (relation) on S;*

- *let f be a program part of p. We say that f is a* fault *if and only if there exists a substitution f' of f such that the program p' obtained from p by substituting f by f' is strictly more-correct than p.*
- *Let f be a fault in p and let f' be a substitute for f. We say that the pair (f, f') is a* (monotonic) fault removal *if and only if the program p' obtained from p by substituting f by f' is strictly more-correct than p.*

Whereas *fault* is an intrinsic property of a program part, *monotonic fault removal* is a binary property involving a fault and a corresponding substitution. We argue that the qualifier *monotonic* is redundant (though

we may still use it, for emphasis), as no substitution ought to be considered a fault removal unless it does make the program strictly more-correct than the original.

For illustration, we consider the following program, say p, taken from [5] (with some modifications):

```
#include <iostream>  ...  ...  ... //  line 1
void count (char q[])                      // 2
   {int let, dig, other, i, l; char c;     // 3
   i=0;let=0;dig=0;other=0;l=strlen(q);    // 4
   while (i<l) {                           // 5
      c = q[i];                            // 6
      if ('A'<=c && 'Z'>c) let+=2;         // 7
      else                                 // 8
      if ('a'<=c && 'z'>=c) let+=1;        // 9
      else                                 //10
      if ('0'<=c && '9'>=c) dig+=1;        //11
      else                                 //12
         other+=1;                         //13
      i++;}                                //14
   printf ("%d %d %d\n",let,dig,other);} //15
```

To define the space of this program, we introduce the following notations:

- $\alpha_A =' A' \ldots ' Z'$.
- $\alpha_a =' a' \ldots ' z'$.
- $\nu =' 0' \ldots ' 9'$.
- $\sigma = \{' +', ' -', ' =', \ldots ' /'\}$, the set of all the ascii symbols.

We let $list\langle T\rangle$ denote the set of lists of elements of type *T*, and we let $\#_A$, $\#_a$, $\#_\nu$ and $\#_\sigma$ be the functions that to each list l assign (respectively) the number of upper case alphabetic characters, lower case alphabetic characters, numeric digits and symbols; also, we let $\#_\alpha$ be defined as $\#_\alpha(l) = \#_a(l) + \#_A(l)$, for an arbitrary list l. We let the space of this program be defined by all the variables declared in line 2. Also, by virtue of the `include` statement of line 1, we add a variable of type `stream`, that serves as the stream variable of the output file (in the parlance of C++). We let this variable be named os (for output stream), we assume (for the purposes of our example) that the stream is a sequence of natural numbers. Using these notations, we write the following specification on S:

$$R = \{(s, s') | q \in list\langle \alpha_A \cup \alpha_a \cup \nu \cup \sigma\rangle \wedge os' = os \oplus \#_\alpha(q) \oplus \#_\nu(q) \oplus \#_\sigma(q)\},$$

where $\oplus$ represents the concatenation operator. We introduce the following programs, which are derived from p by some modifications of its source code:

p_{01} The program obtained from p when we replace (`let+=2`) by (`let+=1`).

p_{10} The program obtained from p when we replace (`'Z'>c`) by (`'Z'>=c`).

p_{11} The program obtained from p when we replace (`let+=2`) by (`let+=1`) and (`'Z'>c`) by (`'Z'>=c`).

We compute the expression $(R \cap P)L$ for each candidate program, and find the following:

- $(R \cap P)L = \{(s, s') | q \in list\langle \alpha_a \cup \nu \cup \sigma\rangle\}$.

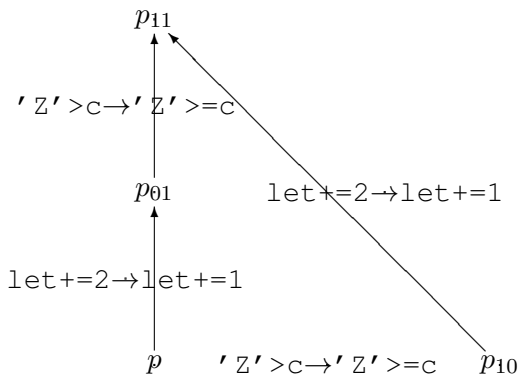

Figure 3: Monotonic and Non-monotonic Fault Removals

- $(R \cap P_{01})L = \{(s, s') | q \in list\langle(\alpha_A \setminus \{'Z'\}) \cup \alpha_a \cup \nu \cup \sigma\rangle\}$.
- $(R \cap P_{10})L = \{(s, s') | q \in list\langle\alpha_a \cup \nu \cup \sigma\rangle\}$.
- $(R \cap P_{11})L = \{(s, s') | q \in list\langle\alpha_A \cup \alpha_a \cup \nu \cup \sigma\rangle\}$.

Figure 3 illustrates the 'more-correct-than' relationships between programs p, p_{01}, p_{10}, and p_{11}, with respect to specification R; each ordering relationship is labeled with the corresponding substitution. From this Figure, we can make the following observations:

- The statement (`let+=2`) is a fault in p, and its substitution by (`let+=1`) is a monotonic fault removal, yielding the more-correct program p_{01}.
- The statement (`'Z'>c`) is a fault in p_{01}, and its substitution by (`'Z'>=c`) is a monotonic fault removal, yielding the more-correct program p_{11}.
- The program part defined by the two statements (`let+=2`) and (`'Z'>c`) is a fault in p, and its substitution by (`let+=1`) and (`'Z'>=c`) is a monotonic fault removal, yielding the more-correct program p_{11}.
- The program p_{11} is correct with respect to R, hence it has no faults.

Note that the statement (`'Z'>c`) is a fault in p_{01} but it is not a fault in p; also note that the statement (`'Z'>c`), *in combination with* the statement (`let+=2`) forms a program part which is a fault in p, but it not a fault in p by itself. The reason is that program p fails for all upper case letters due to the fault (`let+2`), so whether it tests correctly for 'Z' or not does not matter.

4 Validation of Relative Correctness

4.1 Litmus Tests

Now that we have defined the concept of relative correctness, how do we know that our definition is sound? To answer this question, we list in this section some properties that, we believe, a definition of relative

correctness ought to meet; then, in the next section, we check that our definition does indeed meet these conditions. For each property cited below, we discuss why we believe that a definition of relative correctness needs to satisfy this property.

- *Reflexivity and Transitivity, and non-Antisymmetry*. Referring to the loose version of relative correctness (more-correct-than-or-as-correct-as), we feel that this property ought to be transitive and reflexive, for obvious reasons. We also feel that it must not be antisymmetric: In other words, two programs may be mutually more-correct (each is more-correct than the other), and still be distinct (not only syntactically distinct, but computing different functions as well). In particular, we want to maintain the possibility that a given specification admit more than one correct program: all the correct programs are more-correct than one another, without necessarily being identical.

- *Absolute Correctness as the Culmination of Relative Correctness*. Relative correctness ought to be defined in such a way that if a program keeps getting more and more-correct with respect to a specification, it will eventually be (absolutely) correct. Alternatively, we want relative correctness to be defined in such a way that a correct program is more-correct than any candidate program.

- *Relative Correctness as a Sufficient Condition of Higher Reliability, but not a Necessary Condition Thereof*. If program p' is more-correct than program p, then of course we want p' to be more reliable than p; but we do not want *more-correct* to be equivalent to *more reliable*, as the former is a logical/functional property, whereas the latter is a stochastic property.

- *Refinement Implies Relative Correctness with respect to any Specification*. When program p' refines program p, we interpret that to mean that whatever p can do, p' can do as well or better; in particular, it means that p' is more-correct than (or as-correct-as) p with respect to R, for any specification R.

- *Relative Correctness with respect to Arbitrary Specifications Implies Refinement*. The only way for a program p' to be more-correct than a program p with respect to all possible specifications is for p' to refine p; that way, we are assured that whatever p can do, p' can do as well or better.

4.2 Passing the Tests

In this section, we review in turn the desirable properties we have listed above, and show that our definition of relative correctness satisfies every one of them.

4.2.1 Reflexivity, Transitivity, and non-Antisymmetry

Program p' is more-correct than program p if and only if $(R \cap P')L \supseteq (R \cap P)L$. Transitivity and reflexivity stem readily from the definition, as does non-antisymmetry: Indeed, two functions P and P' may satisfy $(R \cap P)L = (R \cap P')L$ while P and P' are distinct. Consider $R = \{(0,1),(0,2)\}$, $P = \{(0,1)\}$ and $P' = \{(0,2)\}$.

4.3 Absolute Correctness as the Culmination of Relative Correctness

A program is more-correct than another if it has a larger competence domain, where the competence domain of a program p with respect to specification R is defined as $(R \cap P)L$. By set theory, the competence domain of a program p is necessarily a subset of RL; when it actually equals RL, the program is correct.

Proposition 5 *Let R be a specification on space S and let p be a program on S. Then p is correct with respect to R if and only if p is more-correct with respect to R than any program on S.*

Proof. Proof of necessity: Let p' be correct with respect to R; then, according to proposition 4, $RL = (R \cap P')L$. Let p be an arbitrary program on space S; by set theory, we have $RL \supseteq (P \cap R)L$. Hence p' is more-correct with respect to R than p.

Proof of sufficiency: Let p' be more-correct with respect to R than any candidate program p on S. Let p'' be a correct program with respect to R; then $(R \cap P'')L = RL$. Since p' is more correct with respect to R than p'', $(R \cap P')L \supseteq (R \cap P'')L$, hence $(R \cap P') \supseteq RL$, which is equivalent to $(R \cap P')L = RL$ since the inverse inclusion is a tautology. **qed**

This proposition provides in effect that (absolute) correctness is the ultimate form of relative correctness: to be correct with respect to a specification a candidate program must be more-correct than any candidate program. We write this as:

$$\boxed{P' \sqsupseteq R \Leftrightarrow (\forall P : P' \sqsupseteq_R P)}.$$

4.4 Relative Correctness and Reliability

The next proposition links the concept of relative correctness to a familiar property: reliability.

Proposition 6 *Let p and p' be two programs on space S and let R be a specification on S. If program p' is more-correct than program p with respect to specification R then p' is more reliable than p.*

Proof. We interpret *more reliable* to mean *less likely to fail*. Reliability is usually estimated with respect to a probability distribution over the input domain (specifically, the domain of specification R), which reflects the likelihood of occurrence of each element of the domain. Given a candidate program p and a probability distribution θ on the domain of R, the probability that a random execution of p succeeds is the integral of θ over the competence domain of p; clearly, the larger the competence domain (with respect to inclusion), the bigger the probability of successful execution. **qed**

If a program is more-correct than another, then it is more reliable. The reverse is not true, of course: a program may be more reliable than another without being more-correct; it may be more reliable because it runs successfully on more frequently occuring input states, and fails on seldom occuring input states. We write:

$$\boxed{P' \sqsupseteq_R P \Rightarrow \int_{(R \cap P')L} \theta(s)ds \geq \int_{(R \cap P)L} \theta(s)ds}.$$

4.5 Relative Correctness and Refinement

The following proposition links relative correctness with the concept of refinement, by casting relative correctness as a form of pointwise refinement.

Proposition 7 *Let p and p' be programs on space S. Then p' refines p if and only if p' is more-correct than p with respect to any specification R on S.*

Proof. Proof of necessity: We have seen in proposition 2 that if P and P' are two functions that P' refines P if and only if $P' \supseteq P$. The condition $(P' \cap R)L \supseteq (P \cap R)L$ stems readily, by monotonicity (from set theory).

Proof of sufficiency: Let p' be more-correct than p with respect to any specification R on S. Then p' is more-correct than p with respect to specification $R = P$. This can be written as: $(P \cap P')L \supseteq (P \cap P)L$, which we simplify as: $(P \cap P')L \supseteq PL$. On the other hand, we have, by construction, $(P \cap P') \subseteq P$. Combining the two conditions, we obtain: $(P \cap P') = P$, from which we infer (by set theory) $P' \supseteq P$ and, by proposition 2, $P' \sqsupseteq P$. **qed**

This proposition provides in effect that traditional refinement is the ultimate form of relative correctness: whereas relative correctness is a tripartite relation linking two programs and a specification, refinement is a bipartite relation linking two programs when one is systematically more-correct than the other regardless of the specification being considered. We can write this as:

$$\boxed{P' \sqsupseteq P \Leftrightarrow (\forall R : P' \sqsupseteq_R P)}.$$

5 Implications and Applications

We briefly review some implications of relative correctness for fault-related artifacts and processes in software engineering, as well as some possible applications of relative correctness.

5.1 Counting Faults

A naive interpretation of fault density in a program views the faults in a program as if they were black balls in a bucket full of otherwise white balls: they are intrinsically identifiable (black vs. white), their number is well-defined, they are independent of each other (in the sense that removal of one ball does not change the color of the other balls), they can be removed in an arbitrary order, and whenever one is removed, their number is reduced by one. In this section we see to what extent this analogy is invalid, since none of these assumptions holds for faults: unlike black balls in a bucket of white balls, faults can be viewed at different levels of granularity, they are highly inter-related, removal of one fault may affect the nature, number and location of other faults, a fault may need to be corrected at more than one location, the same fault may be corrected in more than one way, and the order in which faults are removed matters, as does the way faults are removed.

5.1.1 Elementary Faults

Let p be a program on space S and R be a specification on S; let $f1$ be a fault in p, $f1'$ be a monotonic substitution of $f1$, let p' be the program obtained from p by substituting $f1$ by $f1'$, and let $f2$ be a fault in p'. We argue that the program part made up of $f1$ and $f2$ is a fault in p, since there exists a substitution of $(f1, f2)$ that would make p more-correct: we can substitute $f1$ by $f1'$ to obtain program p', and since by hypothesis $f2$ is a fault in p', there exists a substitution $f2'$ of $f2$ that would produce a program p'' that is more-correct than p. This raise the question: how many faults do we count in program p, one fault (program part $(f1, f2)$), two faults (program part $f1$ and program part $f2$), or three faults (program parts $f1$, $f2$ and $(f1, f2)$). In order to settle this matter, we have to introduce the following definition.

Definition 5 *Let f be a fault in program p on space S with respect to specification R on S. We say that f is an* elementary fault *in p if and only if no part of f is a fault in p with respect to R.*

So that if we are going to count faults, we need to count elementary faults rather than arbitrarily large faults. If we consider the program we introduced in section 3.3, we find that the statement {let+2} is an elementary fault with respect to R, and that the program part ({'Z'>c}, {let+2}) is a fault but is not an elementary fault.

```
#include <iostream>  ...  ...  ... //  line 1
void count (char q[])                    // 2
   {int let, dig, other, i, l; char c;   // 3
   i=0;let=0;dig=0;other=0;l=strlen(q);  // 4
   while (i<l) {                         // 5
      c = q[i];                          // 6
      if ('A'<=c && 'Z'>c) let+=2;       // 7
      else                               // 8
      if ('a'<=c && 'z'>=c) let+=1;      // 9
      else                               //10
      if ('0'<=c && '9'>=c) dig+=1;      //11
      else                               //12
         other+=1;                       //13
      i++;}                              //14
   printf ("%d %d %d\n",let,dig,other);} //15
```

5.1.2 Multi-Site Faults

The foregoing discussion about elementary (and non-elementary) faults may leave the reader with the impression that elementary faults are merely single-site faults, i.e. faults that involve a single statement (or, more broadly, a single contiguous program part). The purpose of this section is to dispel this notion, and to characterize multi-site elementary faults. The distinction between elementary multi-site faults and multiple elementary faults is important from the standpoint of multiple mutation generation, which we discuss in section 5.5. We consider the following space S, specification R, and program p:

- Space, S: {x: real; i: int; a: array [0..N] of real;}.
- Specification, $R = \{(s, s')|x' = \sum_{i=1}^{N} a[i]\}$.
- Program p: {x=0; i=0; while (i<=N-1) {x=x+a[i]; i=i+1;}}

We compute the function of this program, then its competence domain:
$P = \{(s, s')|a' = a \wedge i' = N \wedge x' = \sum_{j=0}^{N-1} a[j]\}$.
$(R \cap P) = \{(s, s')|a' = a \wedge i' = N \wedge a[0] = a[N]\}$.
$(R \cap P)L = \{(s, s')|a[0] = a[N]\}$.
Since $(R \cap P)L$ is not equal to RL, which is L, this program is not correct; indeed, it computes the sum of the array from 0 to N_1 while the specification mandates computing the sum between indices 1 and N. One way to correct this program is to change {i=0} to {i=1} and to change {i<=N-1} to {i<=N}. The question that we raise here is: do we have two faults here ({i=0} and {i<=N-1}) or just one fault that spans two sites? To answer this question we consider the proposed substitutions and check whether they produce more-correct programs. We find:

- p_{01} = {x=0; i=1; while (i¡=N-1) {x=x+a[i]; i=i+1;}}.
  $P_{01} = \{(s, s')|a' = a \wedge i' = N \wedge x' = \sum_{j=1}^{N-1} a[j]\}$.
  $(R \cap P_{01}) == \{(s, s')|a[N] = 0 \wedge a' = a \wedge i' = N \wedge x' = \sum_{j=1}^{N} a[j]\}$.

$(R \cap P_{01})L == \{(s, s')|a[N] = 0\}$.

- p_{10} = {x=0; i=0; while (i¡=N) {x=x+a[i]; i=i+1;}}.
  $P_{10} = \{(s, s')|a' = a \wedge i' = N + 1 \wedge x' = \sum_{j=0}^{N} a[j]\}$.
  $(R \cap P_{10}) = \{(s, s')|a[0] = 0 \wedge a' = a \wedge i' = N + 1 \wedge x' = \sum_{j=1}^{N} a[j]\}$.
  $(R \cap P_{10})L = \{(s, s')|a[0] = 0\}$.

Since the competence domain of p is not a subset of the competence domains of p_{01} and p_{10}, neither p_{01} nor p_{10} is more-correct than p. We extrapolate: no substitution to {`i=0`} can cause program p to include cell $a[N]$ in the sum it is computing in x, and no substitution to {`i<=N-1`} can preclude program p from including cell $a[0]$ in the sum it is computing in x. Hence neither program part {`i=0`} nor program part {`i<=N-1`} is a fault in program p with respect to R, but program part ({`i=0`}, {`i<=N-1`}) is a fault in program p with respect to R, since substitution of this fault by ({`i=1`}, {`i<=N`}) produces a more-correct (and actually correct) program. We say about this fault that it is a *multi-site fault*. We will revisit the concept of multi-site faults when we discuss multiple mutations, in section 5.5.

5.1.3 Fault Density and Fault Depth

It is common for software researchers and practitioners to talk about fault density of a program as a measure of program quality and/or as a measure of the effort that it takes to transform the program into a correct program. In this section we show that fault density reflects neither quality nor fault removal effort: we can show a simple example of a program with a single fault that may still go through several monotonic fault removals before it is correct; also, we can show an example of a program that has several faults, but can be corrected in one *elementary* fault removal. In this discussion, we use the term *fault density* to mean: the number of faults in a program; strictly speaking, fault density is the number of faults per line of code, but for a given program size, these quantities are linearly related.

As an example of a program with a single fault but many fault removals, consider the following space S, specification R, and propgram p:

- Space, S: `int i; float a[0..N];` // $N \geq 2$.
- Specification, $R = \{(s, s')|\forall j : 0 \leq j \leq N : a'[j] = 0\}$.
- Program, p: {`i=2; while (i<=N) {a[i]=0; i=i+1;}`}

Clearly, $RL = L$. To determine correctness, we must compute $(R \cap P)L$ and compare it to RL. We find:
$P = \{(s, s')|a[0] = a'[0] \wedge a[1] = a'[1] \wedge \forall j : 2 \leq j \leq N : a'[j] = 0\}$.
$R \cap R = \{(s, s')|a[0] = a'[0] \wedge a[1] = a'[1] \wedge \forall j : 2 \leq j \leq N : a'[j] = 0\}$.
$(R \cap P)L = \{(s, s')|a[0] = 0 \wedge a[1] = 0\}$.
Hence p is not correct with respect to R. We can check easily that {`i=2`} is a fault in p with respect to R, and we show that the substitution of {`i=2`} by {`i=1`} produces a more-correct program:

- p': {`i=1; while (i<=N) {a[i]=0; i=i+1;}`},
- $P' = \{(s, s')|a[0] = a'[0] \wedge \forall j : 1 \leq j \leq N : a'[j] = 0\}$.

- $(R \cap P') = \{(s, s') | a[0] = a'[0] \wedge \forall j : 1 \leq j \leq N : a'[j] = 0 \wedge \forall j : 0 \leq j \leq N : a'[j] = 0\}$.

- $(R \cap P')L = \{(s, s') | a[0] = 0\}$.

Since RL is (still) L, program P' is not correct with respect to R. We perform another fault removal when we replace {`i=1`} by {`i=1; a[0]=0;`}. We find:

- p'': {{`i=1; a[0]=0;`} `while (i<=N)` {`a[i]=0; i=i+1;`}}.

- $P'' = \{(s, s') | \forall j : 0 \leq j \leq N : a'[j] = 0\}$.

- $(R \cap P'') = \{(s, s') | \forall j : 0 \leq j \leq N : a'[j] = 0\}$.

- $(R \cap P'')L = L$.

Hence the same fault {`i=2`} has been corrected twice, first when we replaced it by {`i=1`} then we replaced {`i=1`} by {`i=1; a[0]=0;`} (we could have replaced {`i=1`} by {`i=0`}, though that would have looked far too artificial).

As for an example of a program with several faults that can be corrected with a single elementary fault removal, consider the following space, specification, and program:

- Space, S: {`x: real; i: int; a: array [0..N] of real;`}.
- Specification, $R = \{(s, s') | x' = \sum_{i=1}^{N} a[i]\}$.
- Program p: {`x=0; i=0; while (i<=N-1)` {`x=x+a[i]; i=i+1;`}}

We compute the function of this program then its competence domain with respect to R:
$P = \{(s, s') | a' = a \wedge i' = N \wedge x' = \sum_{j=0}^{N-1} a[j]\}$.
$(R \cap P) = \{(s, s') | a' = a \wedge i' = N \wedge a[0] = a[N]\}$.
$(R \cap P)L = \{(s, s') | a[0] = a[N]\}$.
Since this is not equal to RL (which is L), we conclude that this program is not correct with respect to R. We see at least two faults in this program, i.e. two program parts that admit substitutions that would make the program more-correct:

- The multi-site fault that we had identified in section 5.1.2, namely the program part ({`i=0`},{`i<=N-1`}).
- The single-site fault {`a[i]`} in the body of the loop; this is a fault since replacing it by {`a[i-1]`} yields a correct program.

If we do substitute {`a[i]`} by {`a[i-1]`} then we obtain a program p' where the multi-site program part ({`i=0`},{`i<=N-1`}) is not a fault, hence the same fault removal action has removed more than one fault. This leads us to introduce the following definition, as a possible alternative metric to fault density.

Definition 6 *We consider a specification R on space S and a program p on space S. The* fault depth *of program p with respect to specification R is the minimal number of elementary fault removals that are required to transform p into a correct program.*

In light of this definition, we find that the depth of the array sum program above is 1, the depth of the array initialization program is also 1, and the depth of the character-counting program (section 6.2.2) is 2. Note that the array sum program has a fault density of two, but a fault depth of 1; interestingly, the fault density, which is usually perceived as indicative of a flaw, in this case appears to be a sign of quality, since it gives us two distinct opportunities to correct the program. Hence not only is fault density a poor measure of program unsoundness, it may sometimes reflect quite the opposite: it may measure the range of possibilities for making the program correct.

We argue that this metric is a more meaningful reflection of program quality, and certainly more directly related to the effort required to make the program correct; also, unlike fault density, this metric does decrease by one whenever we remove a fault (provided the fault is in the minimal path). Given a faulty program p and a program p' obtained from p by monotonic fault removal; if the fault removal is in a minimal sequence of faults removals to a correct program, then we can write:

$$\boxed{depth(p) = 1 + depth(p').}$$

If p'' is obtained from p by an arbitrary monotonic fault removal then all we can claim about the depths of p and p' is:

$$\boxed{depth(p) \leq 1 + depth(p'').}$$

Revisiting the sample program above, we find that if we remove the multi-site fault in the original program p, we find the following program p', which is also correct:

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

Interestingly, note also that if we proceed to remove both faults at once, this yields the following program, which is *not* correct:

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

This program has a fault density of 2 and a fault depth of 1; removing one fault makes it correct; but removing two faults makes it incorrect. The only meaningful characterization of fault density is whether the program has no faults (if it is correct) or whether it has at least one fault (if it is not correct); once we determine that it has at least one fault, then we need to remove that fault then ask the same question about the new program; the answer to that question depends on what substitution we have used to remove the first fault. In other words, fault density can take only two meaningful values: 0, or > 0; fault depth, by contrast, takes its values in the set of natural numbers.

5.2 Monotonic Fault Removal

As programmers, we are all familiar with the frustration of trying to remove faults from a program, only to find that we are running in circles, patching the program at one end only to break it down at another. This would not happen if we restrict program transformations to provably monotonic fault removals; with such a discipline, we are assured that with each transformation, the program becomes more-correct, which means not only that it becomes more reliable, but also that its competence domain grows monotonically with each fault removal, i.e. that it fails for fewer and fewer initial states. Of course, ensuring that a program transformation qualifies as a monotonic fault removal is generally a non-trivial exercise; we postpone the discussion of how to do this to section 6. Here we simply want to argue the case that in the same way that stepwise refinement provides a logical framework for software design, which proceeds monotonically from a specification to a program through *correctness-preserving* transformations, relative correctness provides a logical framework for stepwise fault removal, which starts from an incorrect program and proceeds

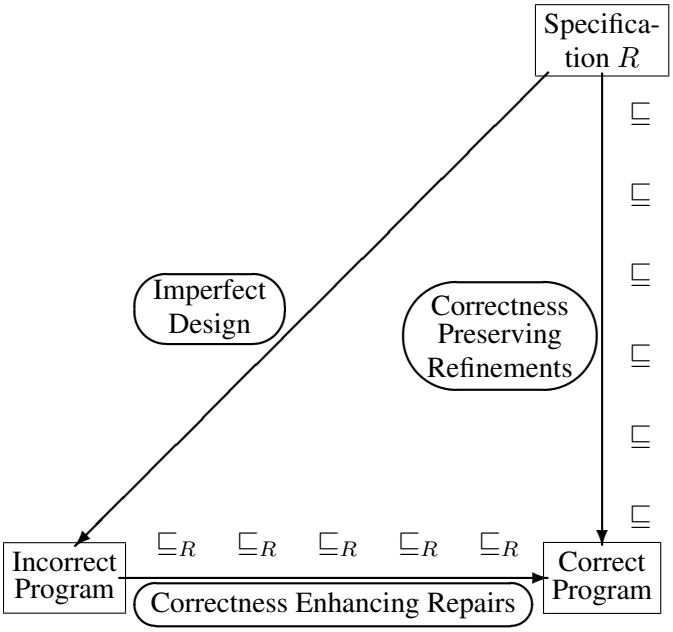

Figure 4: Fault Avoidance vs. Fault Removal

monotonically towards a correct program through *correctness-enhancing* transformations. This process is illustrated in Figure 4. The concept of relative correctness ought to play for software fault removal the same role that refinement plays for software design: first, as a logical framework (for reasoning about faults and fault removal); second, as an ideal process to be followed scrupulously when the stakes warrant it; and third, as a yardstick against which large scale methods are evaluated.

5.3 A Logical Framework for Software Testing

The traditional lifecycle of software testing is triggered by an observation of failure and proceeds by analyzing the failure, tracing it back to a hypothetical fault, removing the fault, then testing the program for correctness. We argue that it is wrong to test the program for correctness at the end of this process, unless we have reason to believe that the fault we have just removed is the last fault of the program. Given that in general we have no way to ascertain that the fault we have just removed was the last fault of the program, there is no reason we should expect the program to be correct, even if we assume that the fault was properly removed. Instead, the most we can hope for is that the new program is more-correct than the original, and we should be testing it for relative correctness rather than absolute correctness.

5.4 Software Repair

Most techniques for program repair [1, 3, 6, 10, 15, 26] proceed by applying transformations on an original faulty program. These transformations may be macro-transformations (including program modifications that span across several statements), or micro-modifications (intra-statement) using mutation operators as those provided by the muJava [19] program mutation tool. Two main approaches exist towards assessing the suitability of the generated transformations: test-based techniques [1, 3, 10, 15] (which use the passing

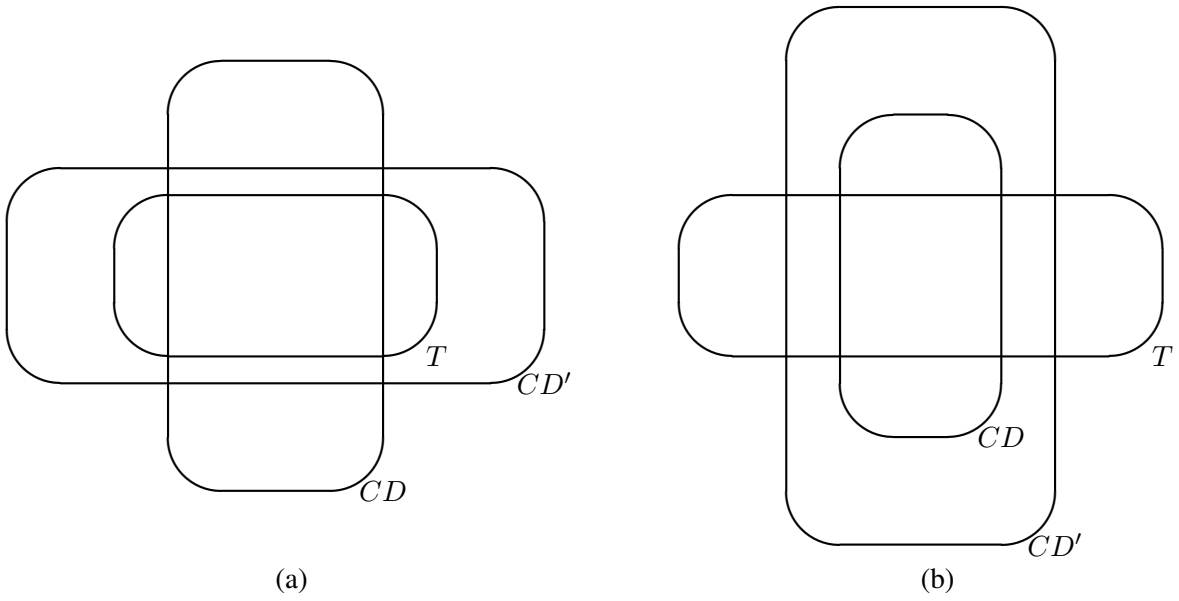

Figure 5: Poor Precision, Poor Recall

of a test suit as the acceptance criterion) or specification-based techniques [6, 26] (which use a specification and some sort of constraint-solving to determine if the new code complies with the specification). Both techniques share the following: repair candidates that fail in some test, or that do not comply with the specification, are immediately discarded and new candidates are examined, while those that do not fail are considered as potential fixes.

We argue that both, the decision to retain successful mutants and the decision to reject unsuccessful mutants, are wrong:

- As Figure 5(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 is not more-correct than the original (since CD is not a subset of CD').
- As Figure 5 (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 is a subset of CD').

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

5.5 Multiple Mutation

Debroy and Wong [3] use a single muJava mutation in order to generate fix candidates. A clear limitation of such an approach is that many faults will not be fixed; this will happen in the case of faults that span through more than one program location, as well as whenever the program under analysis has multiple faults. The natural alternative is to apply multiple mutations. This is the case in tools as the ones presented in [6] and [26]. We will discuss these tools when reviewing related work in section 7.2. The impact of relative correctness on multiple mutation testing depends on the reason for deploying multiple mutations; we see

two possible scenarios, which we will discus in turn: either we use multiple mutations to simultaneously repair multiple faults, or we use multiple mutations to repair single multi-site elementary faults.

- If multiple mutations are deployed to repair multiple faults. When one uses a test of absolute correctness to assess the validity of program repairs, one has to remove all faults at once in order for the test to be meaningful; also, removing all fauts simultaneously spares us the trouble that stems from interference between faults (the condition where one fault precludes us from analyzing another). Multiple mutation proceeds by applying multiple mutation operators at different places in the program then testing the resulting program for absolute correctness. We argue that with relative correctness, it is no longer necessary to consider several faults at once, since we can characterize fault removals one fault at a time. Managing faults one a time and testing programs for relative correctness rather than absolute correctness offers many advantages: first, it spares us the massive combinatorial explosion that stems from applying several mutations through the program; second, it spares us the trouble of reasoning on the basis of fault density, a concept whose significance we have discredited in favor of fault depth; third, it spares us the trouble of dealing with many fault removals at once, when we do not know how each fault removal affects others. Consider for example the array initialization program given in section 5.1.3: Though we have two distinct faults, removing any one of them (with the proper substitution) yields a correct program, but removing them both yields in fact an incorrect program (p''') as we have seen in section 5.1.3.

- If multiple mutations are deployed to repair multi-site elementary faults. In this case, it is sensible to deploy multiple mutations, but note that the mutiplicity of the mutation is not the estimated number of faults we are trying to repair simultaneously (as in the case above) but rather the multiplicity of the multi-site elementary faults we are trying to repair individually. If we assume for example that only 1% of the elementary multi-site faults in the program have more than two sites, then multi-site mutation with only two mutations at a time covers 99% of the targeted faults. Here again, relative correctness is useful, since it enables us to rule conclusively on whether the multiple mutation has produced a more-correct program, regardless of whether the resulting mutant is correct or not.

5.6 Software Design

Of all the possible applications of relative correctness, this is perhaps the most tentative, and that which requires the most research before it can be applied. This research direction stems from the observation that a program p' refines a program p if and only if program p' is more-correct than program p with respect to any specification. If we consider the process of software design by stepwise refinement from a given specification R, we could legitimately ask the question: Why does each step of this process require that we produce a program that is more-correct than the previous step with respect to *all* specifications, when all we care about is specification R? Alternatively, this question can be formulated as: How about designing a program from a specification R, not by successive refinements, but rather by successive steps of relative correctness with respect to R. This may offer us two advantages:

- First, relative correctness with respect to a given specification is a much weaker requirement than refinement (which is relative correctness with respect to any specification); this weaker requirement offers greater latitude to the designer, and is prone to produce simpler programs (as we briefly illustrate below).

- Second, proceeding by relative correctness rather than refinement offers an opportunity, at each design step, to remedy design faults we may have introduced at previous steps (by definition, since relative

correctness enhances, rather than preserves, correctness at each step). By contrast, in a stepwise refinement process, a fault at one step of the refinement dooms all the subsequent steps.

To illustrate how relative correctness offers greater latitude than refinement, we consider a very simple example. Let S be the space defined by integer variables x and y, and let R and p be the following specification and program on S:

$R = \{(s, s')|x' = x + y\}$

p: {`while (y!=0) {x=x+1; y=y-1;}`}.

We consider the following two programs p' and p'', and we leave it to the reader to verify that: p' refines p; p'' does not refine p but is more-correct than p with respect to R. Note that program p'' is simpler than program p' (because it is subject to a weaker requirement).

p': `x=x+y; y=0;`.

p'': `x=x+y;`.

The functions of p, p' and p'' are given below:

$$P = \{(s, s')|y \geq 0 \wedge x' = x+y \wedge y' = 0\}, P' = \{(s, s')|x' = x+y \wedge y' = 0\}, P'' = \{(s, s')|x' = x+y \wedge y' = y\}.$$

6 Establishing Relative Correctness

All the insights that relative correctness may afford us would be useless if we were unable to establish relative correctness relationships between programs; in this section, we review briefly how relative correctness can be established through testing, and through static analysis of source code.

6.1 Testing for Relative Correctness

How do we test a program for relative correctness over another with respect to a given specification? How is that different from testing the program for absolute correctness? We argue that testing a program for relative correctness rather than absolute correctness affects two separate aspects of testing, namely test data generation and oracle design.

- *Test Data Generation*. The essence of test data generation is to approximate an infinite or very large input space by a small representative test data set; clearly, what input space we are trying to approximate influences what test data we select, regardless of what selection criterion we apply. When we test a program for absolute correctness with respect to a specification R, the relevant input space is $dom(R)$. By contrast, when we test a program for relative correctness over program p with respect to specification R, the relevant input space is $dom(R \cap P)$. Given that we are trying to approximate two distinct sets, the data we generate to approximate them are likely to be equally distinct.

- *Oracle Design*. Let $\omega(s, s')$ be the oracle that we use to test a program for absolute correctness with respect to specification R. To test a program p' for relative correctness over program p, we need to check that oracle $\omega(s, s')$ holds only for those states s on which program p runs successfully. Hence the oracle of relative correctness, $\Omega(s, s')$, should be written as follows:

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

  This formula shows how to derive the oracle of relative correctness (Ω) from the oracle of absolute correctness (ω). In [22] we show how to derive the oracle of absolute correctness ($\omega(s, s')$) from specification R.

6.2 Proving Relative Correctness

6.2.1 Relative Correctness of Iterative Programs

In principle, we can prove a program p' more-correct than a program p with respect to specification R by computing P and P' and comparing $(R \cap P)L$ and $(R \cap P')L$; of course, in practice this is usually very difficult. In section 7 we sketch a research agenda for the purpose of deriving practical approaches to the verification of relative correctness by static analysis. In this section, we briefly discuss a preliminary result we have, which allows us to verify the relative correctness of iterative programs, using invariant relations. We consider a while loop w on space S, of the form $\{$`while (t) {b}`$\}$, and we denote by B the function of the loop body b and by T the vector that represents the loop condition $T = \{(s, s')|t(s)\}$. An *invariant relation* of loop w is a reflexive transitive superset of $(T \cap B)$; the interested reader is referred to [24] for more details on invariant relations. For the purposes of our discussions in this paper, we need to remember two properties: smaller invariant relations are better invariant relations (since they capture more functional information of the loop); and the intersection of invariant relations is an invariant relation. Whereas invariant assertions can only be used to prove that a loop is correct, invariant relations can be used to prove correctness, and can also be used to prove incorrectness, by virtue of the following proposition (due to [18]), which gives a sufficient condition and a necessary condition of correctness.

Proposition 8 *Let R be a specification on space S, let w be a while statement on S of the form w:* $\{$`while (t) {b}`$\}$*, such that w terminates normally for any state in S, and let V be an invariant relation of w.*

- Sufficient Condition of Correctness*: If V satisfies the following condition $V\overline{T} \cap RL \cap (R \cup V \cap \widehat{\overline{T}}) = R$, then w is correct with respect to R.*
- Necessary Condition of Correctness*: If w is correct with respect to R then the following condition holds: $(R \cap V)\overline{T} = RL$.*

Intuitive interpretation: The sufficient condition of correctness means in effect that the invariant relation V captures enough information about the loop to subsume the specification R; the necessary condition of correctness means that no loop that admits an invariant relation that violates this condition can possibly be correct with respect to R. If we encounter an invariant relation V that does not satisfy the necessary condition of correctness, we conclude that the loop w is not correct with respect to specification R, irrespective of what else the loop may be doing. We say about such invariant relations that they are *incompatible* with specification R; when a relation does satisfy the necessary condition of correctness, we say about it that it is *compatible* with the specification, even though *not incompatible* is a better characterization of such a relation (when an invariant relation and a specification are not incompatible, it could be because they are talking about totally different aspects of loop behavior, hence are irrelevant to each other). Using these two conditions, we develop an algorithm that generates successive invariant relations and matches them against the specification R until one of the following conditions arises [18]:

- Either we generate a sufficient number of invariant relations to meet the sufficient condition of correctness; in which case we exit the iteration and conclude that the loop is correct with respect to R.
- Or we encounter an invariant relation that violates the necessary condition; then we exit the iteration and conclude that the loop is incorrect with respect to R.

- Or we run out of invariant relations before we reach one of the two conclusions above; then we exit the iteration and conclude that we do not have sufficient information about the loop to rule on its correctness.

We apply a variation of this algorithm, where we generate all the invariant relations at once, and check them for compatibility with respect to the specification. If at least one relation (say Q) is incompatible with specification R, then we conclude that the loop is incorrect, and we prepare to repair it; the following proposition provides the basis for doing so.

Proposition 9 *Let R be a specification on space S and let w be a while loop on S of the form, w:* `{while (t) {b}}` *which terminates for all s in S. Let Q be an invariant relation of w that is incompatible with R, i.e. such that $(R \cap Q)\overline{T} \neq RL$; and let C be the largest invariant relation of w such that $W = (C \cap Q) \cap \widehat{\overline{T}}$. 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 $W' = (C \cap Q') \cap \widehat{\overline{T}}$. Then w' is strictly more-correct than w.*

Interpretation: This proposition provides that if we change the loop in such a way as to replace an incompatible invariant relation (Q) with a compatible invariant relation (Q') of equal strength (so that $((C \cap Q') \cap \widehat{\overline{T}})$ is deterministic, just as $((C \cap Q) \cap \widehat{\overline{T}})$), while preserving all the other invariant relations (C), then we obtain a more-correct while loop.

Proof. By hypothesis, Q is incompatible with R, hence we write:

$$
\begin{array}{ll}
 & (R \cap Q)\overline{T} \neq RL \\
\Rightarrow & \quad \{ \text{ by set theory } (R \cap Q)\overline{T} \subseteq (R \cap Q)L \subseteq RL \ \} \\
 & (R \cap Q)\overline{T} \subset RL \\
\Rightarrow & \quad \{ \text{ By hypothesis, } Q' \text{ is compatible } \} \\
 & (R \cap Q)\overline{T} \subset (R \cap Q')\overline{T} \\
\Rightarrow & \quad \{ \text{ Taking the intersection with C on both sides } \} \\
 & (R \cap Q \cap C)\overline{T} \subset (R \cap Q' \cap C)\overline{T} \\
\Rightarrow & \quad \{ \text{ For any vector } v \text{ and relation } R,\ Rv = (R \cap \widehat{v})L \ \} \\
 & (R \cap Q \cap C \cap \widehat{\overline{T}})L \subset (R \cap Q' \cap C \cap \widehat{\overline{T}})L \\
\Rightarrow & \quad \{ \text{ associativity } \} \\
 & (R \cap (Q \cap C \cap \widehat{\overline{T}}))L \subset (R \cap (Q' \cap C \cap \widehat{\overline{T}}))L \\
\Rightarrow & \quad \{ \text{ substitution } \} \\
 & (R \cap W)L \subset (R \cap W')L.
\end{array}
$$

Hence w' is strictly more-correct than w with respect to R. **qed**

To use this proposition, we proceed as follows:

- We generate all the invariant relations that we can by means of our invariant relations generator [18], and we divide them into two categories, those that are compatible with respect to R and those that are not.

- If there is at least one incompatible relation, we conclude that the loop is incorrect, hence in need of repair; and we select an incompatible invariant relation to be the relation Q that the proposition above refers to.

- Note that even though the proposition refers to relation C, we do not need to compute C to apply the proposition; the sole purpose of C is to ensure that if $(C \cap Q) \cap \widehat{\widehat{T}}$ is deterministic, then so is $(C \cap Q') \cap \widehat{\widehat{T}}$. We interpret this condition to mean that Q' is at least as strong as Q; in practice, it means Q' has as many independent conjuncts as Q.

- To modify Q while preserving C we do not need to know C in full; it suffices to know the part of C that is related to Q. Hence for example if C can be written as the intersection of two terms, say $C = C' \cap C''$, where C'' is not related to Q (e.g. C'' refers to variables that are not involved in the definition of Q) then it suffices to preserves C' since there is nothing we may do in Q that will affect C''.

- In light of the foregoing discussion, we must modify the loop so as to change Q while preserving C', the the intersction of invariant relations that involve the variables of Q. We argue that it is advantageous to preserve, not all the invariant relations that form C', but only those invariant relations that are compatible with specification R. This yields fewer constraints (hence a broader solution space), but also leaves the possibility that by changing Q, we may also change other incompatible relations for the better.

Let Q be the incompatible invariant relation that we choose to change, and let C_1, C_2, ... C_k, be the compatible invariant relations that form C'. By inspecting the variables that are involved in the definition of relation Q, we know what program variables we must modify to alter Q; what we must do now is analyze the constraints under which these variables must be modified. Let x_1, x_2, x_3, ... x_n be the variables of the program, and let us assume that only x_1 and x_2 are involved in the definition of Q. In order to know how to modify x_1 and x_2 in the loop, we write the following condition on x_1, x_2, x'_1, x'_2:

$$\exists x_3, x_4, ...x_n, x'_3, x'_4, ...x'_n : \begin{pmatrix} x_1 & x'_1 \\ x_2 & x'_2 \\ ... & ... \\ x_n & x'_n \end{pmatrix} \in C'.$$

This is a condition on x_1, x_2, x'_1, x'_2 that dictates how we may modify variables x_1 and x_2 that appear in relation Q in such a way as to preserve all the compatible relations in C'. Even if it does not specify uniquely how we should change the code that alters these variables, this condition reduces the range of changes we can make so that only a few changes (mutants) are possible. For each mutant, we recompute the corresponding invariant relation and check whether it is now compatible.

6.2.2 Illustration

We consider the following loop, taken from a C++ financial application, where all the variables except `t` are of type `double`, and where a and b are positive constants.

```
w: while (abs(r-p)>upsilon) {t=t+1; n=n+x; m=m-1; l=l*(1+b);
    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 consider the following specification, which we are judging the loop against:

$$R = \{(s, s') | b < a < 1 \wedge x' = x \wedge w' = w - z \times \frac{1 - (1+a)^{t'-t}}{a}$$

$$\wedge k' = k + 1000 \times (t' - t) \wedge t \leq t' \wedge 0 < l \leq l' \wedge z > 0 \wedge l \times (1+b)^{-t} = l' \times (1+b)^{-t'}\}.$$

Analysis of this loop by an invariant relations generator derives fourteen invariant relations, of which five are found to be incompatible with the specification. We select the following incompatible invariant relation for remediation:

$$Q = \{(s, s') | l \times (1+b)^{-\frac{z}{1+a}} = l' \times (1+b)^{-\frac{z'}{1+a}}\}.$$

We resolve that to remediate this incompatibility, we must alter variable z and/ or variable l. We compute the condition on z and l under which a change in these variables does not alter any of the existing compatible relations, and we find:

$$z' \geq z \wedge (l = l' \vee l \times (l' - l) > 0).$$

We focus our attention on variable z, and consider the possible mutations of the statement `z = (1+a)+z` that preserve the equation $z' \geq z$; for each mutant of this statement, we recompute the new invariant relation that substitutes for Q and check whether it is compatible with R. We find that the statement `z=(1+a)*z` produces a compatible invariant relation, and conclude, by virtue of proposition 9, that the following loop is more-correct with respect to R than the original loop.

```
wm: while (abs(r-p)>upsilon) {t=t+1; n=n+x; m=m-1; l=l*(1+b);
     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;}
```

To test this loop, we generate random test data, run the loop on it, and check the oracle derived from specification R; the loop fails at the third test. *But this does not mean our fault removal was wrong: when we run this loop on randomly generated test data using an oracle that tests for relative correctness rather than correctness (see Section 6.1), it runs for over eight hundred thousand test data without failure*. This increases our confidence that the new loop (wm) is indeed more-correct than the original, even though it is still not correct, due to the existence of at least one more fault in the program.

Running the invariant relations generator on the new loop produces fourteen invariant relations, of which only one is incompatible; it seems that by removing the earlier fault we have remedied four invariant relations at once. Applying the same process to the new loop, we find the following loop, which is correct with respect to R:

```
wc: while (abs(r-p)>upsilon) {t=t+1; n=n+x; m=m+1; l=l*(1+b);
     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;}
```

6.3 Stepwise Proof of Relative Correctness

To prove relative correctness of a program P' over a program P with respect to specification R, we need to compute $(R \cap P)L$ and $(R \cap P')L$ and compare them for inclusion. This, in general, is a tall order, as it is usually very difficult to compute the function of a program; the foregoing discussion (section **??**) explores one way to rule on relative correctness (in a very special case) without having to compute P and P'. But in fact even when we know how to compute P and P', we are not necessarily out of the woods, since R itself may be so complex that it does not lend itself easily to the calculation of $(R \cap P)$ and $(R \cap P')$. Whence the following proposition.

Proposition 10 *Let P and P' be two programs on space S and let R be a specification on S that is written as a join of two subspecification, $R = R_1 \sqcup R_2$. If P' is more-correct than P with respect to R_1 and with respect to R_2 then it is more-correct than P with respect to R.*

7 Concluding Remarks

7.1 Summary

A.M.

7.2 Related Work

In [17] 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 [17] is different from ours, for several reasons: whereas [17] 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 [17] 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 is defined; whereas [17] 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 a successful as simply an initial state/ final state pair that falls with the specification (relation).

In [11] 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. [17] (from the same team), the work of Lahiri uses executable assertions as specifications, represents executions by executaion 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 initil 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 [11]) 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 returns 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 attributes; hence we view abort-freedom as a special form of functional attribute, rather than being an orthogonal attribute. Another important distinction with [11] is that we do not view relative correctness is 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 [16], 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 ([11, 17]), 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 [25], Nguyen et al. present an automated repair method based on symbolic execution, constraint solving, and program synthesisi; 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.

7.3 Assessment and Prospects

establishing relative correctness by static analysis.

if P' is more correct than P with respect to R and with respect to R', and if R and R' have a join, can we conclude that P' is more correct than P with respect to $R \sqcup R'$? if that were true, it would be a valuable tool for separation of concerns. We could for example prove relative correctness with respect to R, and test for relative correctness with respect to R'.

If program p' is more-correct than program p with respect to specification R and R refines R', then program p' is more-correct than p with respect to R'.

also, if P' represents a version of P for a new specification R' that is slightly different from R, what does it mean for P' to be an improved version of P even though it is subject to a different specification? we argue that P' has to be more-correct than P with respect to $R \sqcap R'$.

References

[1] A. Arcuri and X. Yao, *A Novel Co-evolutionary Approach to Automatic Software Bug Fixing*, in CEC 2008.

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

[3] V. Debroy and W.E. Wong, *Using Mutation to Automatically Suggest Fixes to Faulty Programs*, in Proceedings of ICST 2010, pp. 65–74.

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

[5] A. Gonzalez-Sanchez, R. Abreu, H-G. Gross, and A.J.C. van Gemund. Prioritizing tests for fault localization through ambiguity group reduction. In *proceedings, Automated Software Engineering*, Lawrence, KS, 2011.

[6] Gopinath D., Malik M.Z. and Khurshid S., *Specification-Based Program Repair Using SAT*. TACAS 2011: pp. 173–188.

[7] D. Gries. *The Science of programming*. Springer Verlag, 1981.

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

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

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

[11] Shuvendu K. Lahiri, Kenneth L. McMillan, Rahul Sharma, and Chris Hawblitzel. Differential assertion checking. In *Proceedings, ESEC/ SIGSOFT FSE*, pages 345–455, 2013.

[12] J.C. Laprie. Dependability —its attributes, impairments and means. In *Predictably Dependable Computing Systems*, pages 1–19. Springer Verlag, 1995.

[13] Jean Claude Laprie. *Dependability: Basic Concepts and Terminology: in English, French, German, Italian and Japanese*. Springer Verlag, Heidelberg, 1991.

[14] Jean Claude Laprie. Dependable computing: Concepts, challenges, directions. In *Proceedings, COMPSAC*, 2004.

[15] C. Le Goues, T. Nguyen, S. Forrest and W. Weimer, *GenProg: A Generic Method for Automated Software Repair*, IEEE Trans. Software Engineering 38(1), IEEE, 2012.

[16] Francesco Logozzo and Thomas Ball. Modular and verified automatic program repair. In *Proceedings, OOPSLA*, pages 133–146, 2012.

[17] Francesco Logozzo, Shuvendu Lahiri, Manual Faehndrich, and San Blackshear. Verification modulo versions: Towards usable verification. In *Proceedings, PLDI*, 2014.

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

[19] Y.-S. Ma, J. Offutt and Y.-R. Kwon, *MuJava : An Automated Class Mutation System*, Journal of Soft ware Testing, Verification and Reliability, 15(2), Wiley, 2005.

[20] Z. Manna. *A Mathematical Theory of Computation*. McGraw Hill, 1974.

[21] Ali Mili, Marcelo Frias, and Ali Jaoua. On faults and faulty programs. In Peter Hoefner, Peter Jipsen, Wolfram Kahl, and Martin Eric Mueller, editors, *Proceedings, RAMICS: 14th International Conference on Relational and Algebraic Methods in Computer Science*, Lecture Notes in Computer Science, Marienstatt, Germany, April 28-May 1st 2014. Springer.

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

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

[24] Olfa Mraihi, Asma Louhichi, Lamia Labed Jilani, Jules Desharnais, and Ali Mili. Invariant assertions, invariant relations, and invariant functions. *Science of Computer Programming*, 78(9):1212–1239, September 2013.

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

[26] Zemín L., Gutiérrez Brida S., Perez de Rosso S., Aguirre N., Mili A., Jaoua A. and Frias M.F., *Stryker: Scaling Specification-Based Program Repair by Pruning Infeasible Mutants with SAT*, submitted, 2015.