A Semantic Definition of Faults and Its Implications

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

Aleksandr Zakharchenko, Ali Mili  
*NJIT* Newark NJ USA  
az68@njit.edu, mili@njit.edu

**Abstract**—Given that faults are the focus of much software quality assurance (fault avoidance, fault removal, fault tolerance, fault prediction/forecasting), we argue that a formal definition of faults ought to help us enhance the state of the art in this field. In this paper we consider a formal semantic definition of faults, and explore the insights that this definition gives us, and how these insights can be used in practice. Some of these insights are counter-intuitive, which makes them all the more interesting/useful.

**Index Terms**—fault, error, failure, elementary fault, fault removal, fault density, fault depth, fault multiplicity.

I. INTRODUCTION

In this paper we ask the question, *What is a Fault?*, and we show that by trying to answer it, we shed light on the art of fault localization and fault removal. In [3], [20]–[22] Laprie et al. define a fault as the *adjudged or hypothesized cause of an error* [3]. Also, the IEEE Standard *IEEE Std 7-4.3.2-2003* [1] defines a software fault as "An incorrect step, process or data definition in a computer program". Neither of these definitions gives us any useful/usable insight into what a fault is nor how to analyze program faults: The IEEE definition does not even try; the definition of Laprie et al. [3] depends on the definition of an *error*, which in turn assumes that we have a specification of what is a correct state at each step of a computation, clearly an unrealistic assumption (in practice it is barely realistic to assume that we have a specification of the whole program, let alone a specification of every step of its execution).

In this paper, we adopt a definition of faults (due to [28]) that has the following attributes:

- **A Level of Granularity.** Any definition of a fault must refer, perhaps implicitly, to a level of granularity at which we want to isolate faults. An extreme level of granularity may be the whole program, though that is of little interest in practice, as we typically want to isolate faults with greater precision; more common levels may be the level of the elementary statement, the expression, the variable reference, the operator, the operand, the lexeme, etc. Two concepts depend on the level of granularity that we choose: a *syntactic atom* is a unit of source code at the selected level of granularity (statement, expression, variable reference, etc); a *syntactic feature* (or, simply, *feature*) is the aggregate of one or more syntactic atoms. We need the concept of *syntactic feature* because some faults may span more than one syntactic atom, possibly at different locations in the program.

- **Atomic Change Operators.** Following Gazzola et al. [12], we define a vocabulary of *atomic change operators*, where each operator is applied to a syntactic atom to produce a different unit of code. Whereas syntactic atoms define the scale of faults, atomic change operators define the nature and scale of fault removals.

- **A Program and a Specification.** Whereas Laprie et al [3] define faults in terms of errors, hence assume that we can characterize correct states at every step of the program's execution, we aim to define faults by considering only the overall specification of the program.

- **Absolute Correctness and Relative Correctness.** We argue that any definition of fault ought to be based on some concept of relative correctness, i.e. the property of a program to be more-correct than another with respect to a specification. Whereas traditional (absolute) correctness distinguishes between two categories of programs (correct and incorrect), relative correctness ranks candidate programs on a partial ordering, whose maximal elements are the absolutely correct programs. We may well remove a fault from some incorrect program $P$ and obtain a program $P'$, where $P'$ is more-correct than $P$ while still being incorrect.

- **Faults and Elementary Faults.** As we see from the definitions presented in this paper, a fault can be viewed as an opportunity to make an incorrect program more-correct; and fault removal can be viewed as the act of turning that opportunity into an accomplishment (the accomplishment of making the program more-correct). Because some faults may span more than one syntactic atom, we introduce the concept of *elementary fault* as either a fault that is made up of a single syntactic atom, or a fault made up of more than one syntactic atom, but such that no subset of its syntactic atoms is a fault.

- **Fault Density, Fault Depth, and Fault Multiplicity.** Using elementary faults, we define *fault density* as the number of elementary faults in a program, *fault depth* as the minimal number of elementary fault removals that are required to make the program absolutely correct, and *fault multiplicity* as the number of syntactic atoms that form an elementary fault.

While defining faults and fault removals may be an interesting intellectual exercise, one may wonder whether it is useful...
in practice, given that we have been practicing fault removal (debugging) for decades without resorting to such definitions. Analyzing all the theoretical and practical implications of a formal definition of faults is a long term research goal; in this paper, we stride a first step towards this goal: We argue that these definitions are useful in automated program repair, where an intuitive understanding of these concepts is insufficient; we need to codify these concepts into the algorithms that attempt to identify faults and to substitute them with plausible patches [12]. Of course, the field of automated program repair has itself enjoyed significant success of its own, with a continuous stream of engineering solutions over the past decade [2], [6], [7], [9], [13], [15], [23]–[25], [27], [30]–[34], [34], [35], but we argue that the absence of a formal definition of faults has come at a significant cost, as we discuss below. The statements we make below are broad generalizations, which attempt to characterize a wide range of methods and tools with general attributes; like all generalizations, these statements may be unfair to individual methods/ tools, but they are convenient abstractions, for the purposes of our discussions.

- **Limited Range.** To repair a program does not necessarily mean to make it absolutely correct; it only means to make it more-correct than it was originally. Yet in the absence of a concept of relative correctness, current methods and tools validate candidate repairs by means of a test of absolute correctness; as a result, they are restricted to cases when the program under repair is within striking distance of absolute correctness. By contrast, with relative correctness we can repair a program \( P \) into a program \( P' \) that may itself be incorrect, provided it is more-correct than \( P \).

- **Inaccurate Validation.** In the absence of a definition of relative correctness, program repair methods resort to approximations of absolute correctness: some tools use regression testing as the selection criterion of repair candidates; others use fitness functions as the selection criterion. We prove in [10] that the first option causes a loss of recall, because the absence of regression is a sufficient condition of relative correctness, but not a necessary condition. Also, we show that the second option causes a loss of precision, because having a greater fitness value is a necessary condition of relative correctness, but not a sufficient condition.

- **Fault Repair vs. Failure Remediation.** Program repair methods and tools are typically deployed upon an observation of failure (represented by so-called *negative test data*) and aim to derive a variant of the faulty program that ensures that the observed failure is remedied. We argue that this is a recipe for unbounded combinatorial explosion, because when we resolve to remedy a given failure, we have no way to tell how many faults are conspiring to cause this failure. A more judicious approach is to let fault repair, rather than failure remediation, drive the process. In other words, rather than seeking the variant that remedies the observed failure, we ought to seek any variant that is more-correct than the original program. If we keep enhancing correctness repeatedly (by removing one fault at a time) we will eventually remedy the observed failure; but we do so in a way that lets the program expose its faults one at a time, in the order it determines (rather than picking an arbitrary failure and seeking to repair the combination of faults that are causing it).

- **Fault Dependency.** Fault localization tools attempt to identify suspicious statements by highlighting statistical relationships between program failures and statement activations [12]. When, in addition to removing faults one at a time (as we advocate above) we also resolve to run fault localization after each fault removal, we enhance the precision with which we localize the next fault. Each fault repair helps us diagnose the next fault with greater precision.

- **Fault Depth and Fault Multiplicity.** When program repair tools apply all the available atomic changes to a program but fail to produce a valid repair, they attempt to combine two or more atomic changes and test the generated variants. The question that this raises is: is the tool attempting to remove several faults simultaneously, or is it trying to remove a single fault that happens to span several atomic changes? This question is important because if we resolve to remove faults one at a time (as we should), then the number of atomic changes that we attempt ought to be determined by fault multiplicity rather than fault depth. This is an important distinction because fault multiplicity is typically very small (seldom greater than 2 to 3) whereas fault depth is unbounded.

In this paper we review a semantic-based definition of fault, along with related concepts, such as fault removal, elementary fault, fault density, fault depth, fault multiplicity, etc, then we explore how these definitions enable us to analyze, critique, and propose enhancements to the practice of automated program repair. To support our case, we consider an archetypical program repair tool, viz GenProg [24], we modify it by integrating some of the ideas that stem from our definitions, and we show, using benchmark programs and faults, that the new version gives significantly better results than the original.

In section II we briefly introduce some mathematical notations, which we use to define absolute correctness and relative correctness; in section III we introduce definitions of faults and related concepts, and in section IV we present a set of oracles that test for absolute correctness and relative correctness. In sections V and VI we use the oracles of section IV to illustrate the contrast between fault density and fault depth and, respectively, the contrast between fault depth and fault multiplicity. In section VII we show how we can improve the performance of GenProg by replacing its patch validation (which is based on approximations of absolute correctness) with our patch validation, which is based on relative correctness and uses the oracles of section IV. We conclude in section VIII with some remarks about prospects of our work.
II. MATHEMATICS FOR RELATIVE CORRECTNESS

We assume the reader familiar with elementary relational algebra [5], hence we merely focus in this section on presenting some definitions and notations. We represent sets in a program-like notation by writing variable names and associated data types; if we write $S$ as $x: X; y: Y; z: Z$ then we mean to let $S$ be the cartesian product $S = X \times Y; elements of S are usually denoted by $s$ and the $X$- (resp. $Y$-) component of $s$ is denoted by $x(s)$ (resp. $y(s)$); when no ambiguity arises, we may write $x$ for $x(s)$, and $x'$ for $x(s')$, etc. A relation $R$ on set $S$ is a subset of $S \times S$. Special relations on $S$ include the universal relation $L = S \times S$, the identity relation $I = \{(s, s) | s \in S\}$ and the empty relation $\phi = \{\}$. Operations on relations include the set theoretic operations of union, intersection, difference and complement; they also include the converse of a relation $R$ defined by $\bar{R} = \{(s, s') | (s', s) \in R\}$, the domain of a relation defined by $\text{dom}(R) = \{s | \exists s': (s, s') \in R\}$, and the product of two relations $R$ and $R'$ defined by: $R \circ R' = \{(s, s''') | \exists s'' : (s, s') \in R \land (s'', s') \in R'\}$; when no ambiguity arises, we may write $RR'$ for $R \circ R'$. The pre-restriction of relation $R$ to set $T$ is the relation denoted by $T \cap R$ and defined by: $T \cap R = \{(s, s') | s \in T \land (s, s') \in R\}$.

A relation $R$ is said to be reflexive if and only if $I \subseteq R$, symmetric if and only if $R = \bar{R}$, antisymmetric if and only if $R \cap \bar{R} \subseteq I$, asymmetric if and only if $R \cap \bar{R} = \phi$ and transitive if and only if $RR \subseteq R$. A relation $R$ is said to be total if and only if $I \subseteq R \cap \bar{R}$ and deterministic if and only if $RR \subseteq I$. A relation $R$ is said to be a vector if and only if $RL = R$; vectors have the form $R = A \times S$ for some subset $A$ of $S$; we use them as relational representations of sets; in particular, note that $RL$ can be written as $\text{dom}(R) \times S$. We may, by abuse of notation, use the same symbol (e.g. $T$) to represent a set and the vector defined by the set $(T \times S)$.

A. Program Semantics

Given a program $p$ on space $S$ written in a C-like notation, we define the function of $p$ (denoted by $P$) as the set of pairs $(s, s')$ such that if program $p$ starts execution in state $s$ it terminates in state $s'$; we may, when no ambiguity arises, refer to a program and its function by the same name, $P$.

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

Intuitively, this definition means that $R'$ has a larger domain than $R$ and that on the domain of $R$, $R'$ assigns fewer images to each argument than does $R$. Refinement is used to define (total) correctness, as per the definition below.

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

This definition is identical (modulo differences of notation) to traditional definitions of (total) correctness [11], [14], [16], [26]. The following Proposition, due to [29], helps set the stage for the definition of relative correctness (Definition 3).

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

Interpretation: $RL$, the domain of $R$, is the set of states on which execution of candidate programs $P$ must produce correct outputs according to $R$; on the other hand, $(R \cap P)L$ is the set of states on which execution of candidate program $P$ does produce correct outputs according to $R$. The program $P$ is correct if and only if these two sets are identical. We refer to the domain of $(R \cap P)$ as the competence domain of $P$ with respect to $R$.

Definition 3: Due to [28]. Given a specification $R$ and two deterministic programs $P$ and $P'$, we say that $P'$ is more-correct (resp. strictly more-correct) than $P$ with respect to $R$ if and only if $(R \cap P')L \supseteq (R \cap P)L$ (resp. $(R \cap P')L \supset (R \cap P)L$).

To contrast relative correctness with the definition of correctness given in Definition 2, we may refer to the latter as absolute correctness. For deterministic programs, to be more-correct simply means to have a larger competence domain. See Figure 1; note that $P'$ is more-correct than $P$ but does not duplicate the correct behavior of $P$, rather it has a different correct behavior (this is possible because $R$ is non-deterministic, hence allows more than one correct behavior). This explains why we argue that program repair methods and tools that apply regression tests to candidate repairs are imposing a sufficient but unnecessary condition; hence they are prone to loss of recall. In [10] Diallo et al. prove that this definition of relative correctness satisfies the following properties:

- Relative correctness is reflexive and transitive but not antisymmetric.
- An absolutely correct program is more-correct than (or as correct as) any candidate program.
- If $P'$ is more-correct than $P$ then it is more reliable than $P$.
- $P'$ refines $P$ if and only if $P'$ is more-correct than $P$ with respect to any specification $R$.

We argue that any definition of relative correctness ought to satisfy these properties.

III. FAULTS AND FAULT REMOVALS

A. Faults

In this section, we address the question: What is a Fault in a Program? To answer this question, we recognize that a definition of a fault depends on a number of attributes, namely:

- A Level of Granularity. Implicit in any definition of a fault is the level of granularity at which we wish to isolate faults; the level of granularity that we adopt determines what we view as a syntactic atom, i.e. the scale at which we want to isolate and repair faults.
- A Syntactic Feature. A syntactic feature (or feature, for short) of program $P$ is the aggregate of one or more (possibly non-contiguous) syntactic atoms of $P$. We refer...
to the number of syntactic atoms in a feature as the multiplicity of the feature.

- **A Specification.** Of course, the definition of a fault depends necessarily on a specification, which characterizes correct program behavior.

- **A Catalog of Atomic Change Operators.** In [12], Gazzola et al. introduce the concept of atomic change operators as an abstraction of operations that may be applied to syntactic atoms of the source code of a program, presumably for fault removal.

Using these parameters, we introduce the following definitions.

**Definition 4:** Given a specification \( R \), a program \( P \) and a feature \( f \) in \( P \), we say that \( f \) is a fault in \( P \) with respect to \( R \) if and only if there exists a feature \( f' \) such that the program \( P' \) obtained from \( P \) by replacing \( f \) by \( f' \) is strictly more-correct than \( P \) with respect to \( R \).

Implicit in this definition is the assumption that \( f' \) is derived from \( f \) by application of one or more atomic change operators (depending on the multiplicity of \( f \)). As an illustration, we consider the following space:

\[
\begin{align*}
\text{float } & x; \text{ float } a[N+1]; \\
\text{We let } & R \text{ and } P \text{ be, respectively, the following specification and program on space } S: \\
R & = \{(s, s') | x' = \sum_{i=1}^{N} a[i]\}, \\
P & = \{ \text{int } i=0; \ x=0; \\
& \quad \text{while } (i<N) \{ x=x+a[i]; i=i+1; \} \}
\end{align*}
\]

Clearly, \( P \) is not correct with respect to \( R \). We consider the following program, which can be derived from \( P \) by substitutions at the level of the lexical token:

\[
\begin{align*}
P' & = \{ \text{int } i=1; \ x=0; \\
& \quad \text{while } (i<N) \{ x=x+a[i]; i=i+1; \} \}
\end{align*}
\]

Program \( P' \) is absolutely correct, as can be checked readily, hence it is strictly more-correct than \( P \); therefore the (composite) feature \( f' \) is \((0, <)\) (where \( 0 \) is the initialization of \( i \) and \( < \) is the condition of the loop) is a fault in \( P \) with respect to \( R \).

If we consider substitutions at the level of expressions, then we can derive the following program from \( P \):

\[
\begin{align*}
P'' & = \{ \text{int } i=0; \ x=0; \\
& \quad \text{while } (i<N) \{ x=x+a[i+1]; i=i+1; \} \}
\end{align*}
\]

Program \( P'' \) is absolutely correct with respect to \( R \), hence it is strictly more-correct than \( P \) with respect to \( R \); therefore the feature \( i \) in the assignment statement \( x=x+a[i] \) is also a fault in \( P \) with respect to \( R \).

If we consider substitutions at the level of blocks of code, then we can derive the following program from \( P \):

\[
\begin{align*}
P''' & = \{ \text{int } i=0; \ x=0; \\
& \quad \text{while } (i<N) \{ x=i+1; x=x+a[i]; \}
\end{align*}
\]

Program \( P''' \) is absolutely correct with respect to \( R \), hence it is strictly more-correct than \( P \) with respect to \( R \); therefore the feature \( x \) in the assignment statement \( x=x+a[i] \) is also a fault in \( P \) with respect to \( R \).

We have identified three distinct features in \( P \), of varying scale and multiplicity, each admits a substitute that makes the program strictly more-correct than \( P \) with respect to \( R \).

**B. Fault Removals**

**Definition 5:** Given a program \( P \) and a specification \( R \), a pair of features \( (f, f') \) is said to be a fault removal in \( P \) with respect to \( R \) if and only if \( f \) is a fault in \( P \) and program \( P' \) obtained from \( P \) by replacing \( f \) by \( f' \) is strictly more-correct than \( P \).

In practice, fault removals are defined with respect to a vocabulary of atomic change operators, and are restricted to pairs \((f, f')\) such that \( f' \) is derived from \( f \) by one or more atomic changes (depending on the multiplicity of \( f \)). If we revisit the example of the array sum program discussed above, we find the following fault removals:

- **From \( P \) to \( P' \).** The fault removal is: \( ((0, <), (1, \leq)) \) at the appropriate location.

- **From \( P \) to \( P'' \).** The fault removal is: \( (i, i+1) \) at the appropriate location.

- **From \( P \) to \( P''' \).** The fault removal is:

\[
\begin{align*}
(i = i + 1; \ x = x + a[i];) \\
\end{align*}
\]

at the appropriate location.

**C. Elementary Faults**

If we consider the definition of a fault (Definition 4), we find that if \( f_1 \) and \( f_2 \) are faults in program \( P \) with respect to specification \( R \), then so is the aggregate \( (f_1, f_2) \) (by transitivity of relative correctness). This raises the question: how can we distinguish between a fault that spans two (or more) syntactic tokens and two (or more) faults that have a single atom each; whence the following definition.

**Definition 6:** Given a program \( P \), a specification \( R \), and a fault \( f \) in \( P \) with respect to \( R \), we say that \( f \) is an elementary fault in \( P \) with respect to \( R \) if and only if \( f \) has a single syntactic atom, or it has more than one syntactic atom, but no subset thereof is a fault.

In other words, if \( f \) is defined by a set of syntactic atoms \( f = \{a_1, ..., a_k\} \) then no subset thereof is a fault. Why is this definition important? Because we cannot talk about faults in the plural (two faults, three faults, etc.) if we cannot even tell the difference between one fault and two faults.
We revisit the array sum program, and we consider the first fault we had identified; it is \( f = (0, <) \). To determine whether we are looking at two separate elementary faults or a single two-atom elementary fault, we consider the following programs:

\[
P_1' = \{ \text{int } i=1; x=0; \text{ while (i<N) } \{ x=x+a[i]; i=i+1; \} \}
\]

\[
P_2' = \{ \text{int } i=0; x=0; \text{ while (i<N) } \{ x=x+a[i]; i=i+1; \} \}
\]

\( P_1' \) is obtained from \( P \) by the atomic change \((0,1)\) in the initialization of \( i \), and \( P_2' \) is obtained from \( P \) by the atomic change \((<, \leq)\) in the loop condition. In order to determine whether \( f \) is an elementary fault, we must check whether \( 0 \) (in the initialization of \( i \)) alone is a fault, and whether \( < \) (in the loop condition) alone is a fault. To this effect, we must check whether \( P_1' \) is strictly more-correct than \( P \), and whether \( P_2' \) is strictly more-correct than \( P \). The functions of \( P, P_1' \) and \( P_2' \) are:

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

\[
P_1' = \{(s,s')|i'=N \land a'=a \land x'=\sum_{k=0}^{N-1} a[k] \}
\]

\[
P_2' = \{(s,s')|i'=N \land a'=a \land x'=\sum_{k=0}^{N-1} a[k] \}
\]

The competence domains of \( P, P_1' \) and \( P_2' \) with respect to \( R \) are, respectively:

\[
CD = \{ a[N] = 0 \}
\]

\[
CD_1' = \{ a_1[N] = 0 \}
\]

\[
CD_2' = \{ a_2[N] = 0 \}
\]

Since no inclusion relation holds between \( CD \) and \( CD_1' \), \( P_1' \) is not more-correct than \( P \) with respect to \( R \); since no inclusion relation holds between \( CD \) and \( CD_2' \), \( P_2' \) is not more-correct than \( P \) with respect to \( R \). See Figure 2.

Now, we consider the following example: We let the space \( S \) of the program be defined by the following program variable:

\[
\text{int } a[N+1];
\]

and we let the specification \( R \) and program \( P \) be defined as follows:

\[
R = \{(s,s')|a[0] = a'[0] \land \forall k : 1 \leq k \leq N : a'[k] = 0 \}
\]

\[
P = \{ \text{int } i=0; \text{ while (i<N) } \{ a[i]=0; i=i+1; \} \}
\]

The function of \( P \) is:

\[
P = \{(s,s')|a[N] = a[0] \land \forall k : 0 \leq k \leq N-1 : a'[k] = 0 \}
\]

The competence domain of \( P \) is:

\[
CD = \{ a[0] = 0 \land a[N] = 0 \}
\]

Because this is not the same as the domain of \( R \) (which is all of \( S \)), program \( P \) is not correct with respect to \( R \). We consider the following programs, obtained from \( P \) by, respectively, changing the initialization of \( i \) to 1, changing the condition of the loop to \( \leq \), and performing both changes.

\[
P_1' = \{(s,s')|a[0] = a'[0] \land \forall k : 1 \leq k \leq N-1 : a'[k] = 0 \land a[N] = a'[N] \}
\]

\[
P_2' = \{(s,s')|a[N] = a'[N] \}
\]

\[
P' = \{(s,s')|a[0] = a'[0] \land \forall k : 1 \leq k \leq N : a'[k] = 0 \}
\]

The competence domains of these programs with respect to \( R \) are given below:

\[
CD_1' = \{ a_1[N] = 0 \}
\]

\[
CD_2' = \{ a_2[N] = 0 \}
\]

\[
CD' = \{ a_1[N] = 0 \}
\]

Considering the inclusion relations between \( CD, CD_1', CD_2' \) and \( CD' \), we infer that \( P_1' \) and \( P_2' \) are more more-correct than \( P \), and that \( P' \) is more-correct than all of \( P, P_1' \) and \( P_2' \). This is illustrated in Figure 3. The contrast between Figures 2 and 3 illustrates the difference between a single fault of multiplicity 2 and two faults of multiplicity 1.

D. Fault Density and Fault Depth

Now that we have defined faults and elementary faults, we can introduce metrics to reflect faultiness, i.e. the degree to which a program has faults; note the contrast (orthogonality) between relative correctness, which is a semantic property, and faultiness, which is a syntactic property.

Definition 7: Given a program \( P \) and a specification \( R \), the fault density of \( P \) with respect to \( R \) is the number of elementary faults in \( P \) (for the selected scale of syntactic atoms). The fault depth of \( P \) with respect to \( R \) is the minimal number of elementary fault removals that are needed to transform \( P \) into an absolutely correct program (for the selected set of atomic changes).

If we consider the array sum program above, and we allow for a broad definition of syntactic atoms (including lexical tokens, expressions, and compound statements), then we find that the fault density of this program is at least three, since we identified three distinct elementary faults; the fault depth is 1, since this program is within one elementary fault removal from a correct program.

As for the array initialization program, if we consider lexical tokens as syntactic atoms, then the fault density is at least

---

**Fig. 2. A Single Fault, Multiplicity 2**

**Fig. 3. Two Faults, Multiplicity 1**
two, since we found two faults in that program; also, its fault depth is at most two, since we have found (see Figure 3) a sequence of two elementary fault removals that make this program absolutely correct.

Fault depth and fault density do not take the same value, as shown in the discussion of the array sum program, above: just because we have N faults does not mean as must perform N fault removals before the program is correct. Also, fault density and fault depth evolve differently as faults are removed from a programs; let \( P' \) be derived from \( P \) by an elementary fault removal. Then, by definition, we have \( \text{depth}(P) \geq \text{depth}(P') + 1 \). Equality occurs if \( P' \) is on a minimal path from \( P \) to a correct program. By contrast, we know of no relation between \( \text{density}(P) \) and \( \text{density}(P') \), as faults interfere with each other in arbitrary ways. This matter is revisited in section V, where we review a (more) concrete example.

IV. A HIERARCHY OF ORACLES

In practice, fault removals are carried out exclusively through testing, by comparing the behavior of the program at hand before and after fault removal. In this paper we define, and validate, test oracles to this effect; specifically, we use the specification to derive an oracle of absolute correctness; an oracle of relative correctness; then we use the oracle of relative correctness to derive an oracle of strict relative correctness. All the oracles take the form of a binary predicate and refer to the initial state and the final state of the program, as in the following generic pattern:

\[
\text{inits} = s; \quad \text{// save the initial state} \\
\text{P}(); \quad \text{// program under test, modifies s} \\
\text{assert} \{\text{oracle}(\text{inits}, s)\}; \quad \text{// check correctness}
\]

A. Absolute Correctness

We define the oracle of absolute correctness for a given specification \( R \), then we present a proposition to the effect that this oracle does not indeed test absolute correctness.

**Definition 8:** Given a specification \( R \) on space \( S \), the oracle of absolute correctness derived from \( R \) is denoted by \( \Omega(s, s') \) and defined by:

\[
\Omega(s, s') \equiv (s \in \text{dom}(R) \Rightarrow (s, s') \in R).
\]

The following proposition shows that this definition is sound.

**Proposition 2:** Let \( \Omega(s, s') \) be the oracle of absolute correctness derived from specification \( R \) on space \( S \) and let \( T \) be a subset of \( S \). A program \( P \) is absolutely correct with respect to \( T, R \) (the pre-restriction of \( R \) to \( T \)) if and only if execution of \( P \) on every element of \( T \) satisfies oracle \( \Omega(s, s') \).

**Proof.** Proof of Sufficiency. We write what it means for program \( P \) to satisfy oracle \( \Omega(s, s') \) for every element \( s \) of \( T \):

\[
\forall s \in T: \Omega(s, P(s)).
\]

By the definition of \( \Omega(s, s') \), we can rewrite this as:

\[
\forall s \in T: s \in \text{dom}(R) \Rightarrow (s, P(s)) \in R.
\]

Distributing the clause \( (s \in T) \), we write:

\[
\forall s: s \in T \wedge s \in \text{dom}(R) \Rightarrow s \in T \wedge (s, P(s)) \in R.
\]

By set theory, we write the left hand side as:

\[
\forall s: s \in (T \cap \text{dom}(R)) \Rightarrow s \in T \wedge (s, P(s)) \in R.
\]

We have seen in section II that \( T \cap \text{dom}(R) = \text{dom}(T \cap R) \), hence:

\[
\forall s: s \in (\text{dom}(T \cap R)) \Rightarrow s \in T \wedge (s, P(s)) \in R.
\]

Since \( (s, P(s)) \) is also, by definition, an element of \( P \), this can be written as:

\[
\forall s: s \in \text{dom}(T \cap R) \Rightarrow s \in T \wedge (s, P(s)) \in (R \cap P).
\]

If we now view \( T \) as a vector rather than a set, we can rewrite

\[
\forall s: s \in \text{dom}(T \cap R) \Rightarrow (s, P(s)) \in T \wedge (s, P(s)) \in (R \cap P).
\]

Taking the intersection, and using associativity, we find:

\[
\forall s: s \in \text{dom}(T \cap R) \Rightarrow (s, P(s)) \in ((T \cap R) \cap P).
\]

Rewriting \( (T \cap R) \) as the prerestriction of \( R \) to \( T \), we find:

\[
\forall s: s \in \text{dom}(T \cap R) \Rightarrow (s, P(s)) \in ((T \cap R) \cap P).
\]

From the right hand side, we infer that \( s \) is in the domain of \( (T \cap R) \cap P ) : \)

\[
\forall s: s \in \text{dom}(T \cap R) \Rightarrow s \in \text{dom}((T \cap R) \cap P).
\]

Since this is true for all \( s \), we write:

\[
\text{dom}(T \cap R) \subseteq \text{dom}((T \cap R) \cap P).
\]

which we rewrite as:

\[
T, RL \subseteq ((T \cap R) \cap P)L.
\]

Given that the inverse inclusion is a tautology, we find:

\[
T, RL = ((T \cap R) \cap P)L.
\]

Hence, by proposition 1, \( P \) is absolutely correct with respect to \( T, R \).

**Proof of Necessity.** If \( P \) is correct with respect to \( T, R \), then by proposition 1 \( T, RL \subseteq (T \cap R \cap P)L \). Interpreting this formula in logical terms, we find:

\[
\forall s: s \in \text{dom}(T \cap R) \Rightarrow s \in \text{dom}(T \cap R \cap P).
\]

If this formula holds for all \( s \) in \( S \), it holds a fortiori for all \( s \) in \( T \): \( \forall s: s \in \text{dom}(T \cap R) \Rightarrow s \in \text{dom}(T \cap R \cap P) \).

Because \( T \cap R \) can be written as \( T \wedge \text{dom}(R) \), where \( T \) is reinterpreted as a vector, because \( \text{dom}(T \cap R) = T \wedge \text{dom}(R) \), we can write:

\[
\forall s: s \in T \wedge \text{dom}(R) \Rightarrow s \in T \wedge \text{dom}(R \cap P).
\]

Isolating the clause \( (s \in T) \), we get:

\[
\forall s: s \in \text{dom}(R) \Rightarrow s \in \text{dom}(R \cap P).
\]

Removing the clause \( (s \in T) \), which is now redundant, we find:

\[
\forall s: s \in \text{dom}(R) \Rightarrow s \in \text{dom}(R \cap P).
\]

Since \( P \) is deterministic,

\[
\forall s: s \in \text{dom}(R) \Rightarrow (s, P(s)) \in (R \cap P).
\]

By the definition of the oracle of absolute correctness:

\[
\forall s: s \in T \wedge (s, P(s)) \in R.
\]

Qed

Since we are testing \( P \) only on set \( T \), we cannot hope to prove more than the absolute correctness of \( P \) with respect to \( T, R \); this proposition provides that we can prove no less. Based on this proposition, we derive the following oracle:

```c
bool absoluteCorrectness(testData T) {
    statetype inits, s; bool absorc-true;
    while (moretestdata(T)) {
        inits = gettestdata(T); //load test datum
        s = inits;p(); // modifies s, not inits
        absorc = absorc & absorc(inits, s);
    }
    return absorc
}
```

```c
bool absoracle(statetype s, sprime)
return (!domR(s) || R(s,sprime))
```
where \( \text{absoracle}(s,s') \) checks the correctness of \( P \) for a single execution and \( \text{absoluteCorrectness}(T) \) checks the absolute correctness of \( P \) with respect to the restriction of \( R \) to \( T \). We assume that we have at our disposal the unary predicate \( \text{domR}() \) and the binary predicate \( R(), \) that represent the specification \( R \).

### B. Relative Correctness

Given a space \( S \), a specification \( R \) on \( S \) and a program \( P \) on \( S \), we consider the execution of some program \( P' \) on some initial state \( s \), and we assume that this execution terminates normally and returns a final state \( s' \). We want to derive an oracle that analyzes the pair of states \((s, s')\) and determines whether it is consistent with the premise that \( P' \) is more-correct than \( P \) with respect to \( R \).

**Definition 9:** Given a specification \( R \) on space \( S \) and a program \( P \) on \( S \), the oracle of relative correctness over \( P \) with respect to \( R \) is denoted by \( \omega(s,s') \) and defined by:

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

The following proposition shows that this definition is sound.

**Proposition 3:** Let \( \omega(s,s') \) be the oracle of relative correctness over program \( P \) with respect to specification \( R \) and let \( T \) be a subset of \( S \). A program \( P' \) is more-correct than \( P \) with respect to \( T \) if and only if execution of \( P' \) on every element of \( T \) satisfies oracle \( \omega(s,s') \).

**Proof.** Proof of Sufficiency. If the execution of \( P' \) for every element of \( T \) satisfies the oracle \( \omega(s,s') \) then:

\[
\forall s \in T : \omega(s,s') \Rightarrow \Omega(s,P') \Rightarrow \Omega(s,s').
\]

Replacing \( \Omega() \) by its definition, we find:

\[
\forall s \in T : (\Omega(s,P(s)) \Rightarrow (s,P'(s)) \in R) \Rightarrow (s,P'(s)) \in R).
\]

The body of this quantified formula has the form: \((a \Rightarrow b) \Rightarrow (a \Rightarrow c)\). If we simplify this Boolean expression, we find that it can be written as: \((a \land b) \Rightarrow c\). Given that in our case \( b \) (which is \((s,P(s)) \in R\)) logically implies \( a \) (which is \( s \in \text{dom}(R)\)), this can further be simplified to: \((b \Rightarrow c)\). Hence we write:

\[
\forall s \in T : (s,P(s)) \in R) \Rightarrow ((s,P'(s)) \in R).
\]

Because \( P \) and \( P' \) are deterministic, this can be written as:

\[
\forall s \in T : \exists s' : s' = P(s) \land ((s,s') \in R) \Rightarrow \exists s' : s' = P'(s) \land ((s,P'(s)) \in R).
\]

By rewriting \( s' = P(s) \) in relational form as \((s,s') \in P \) and taking the intersection, we find:

\[
\forall s \in T : \exists s' : ((s,s') \in R \land P) \Rightarrow \exists s' : ((s,P'(s)) \in R \land P').
\]

By the definition of domain, we write:

\[
\forall s \in T : s \in \text{dom}(R \land P) \Rightarrow s \in \text{dom}(R \land P').
\]

Factoring the term \((s \in T)\) into the formula, we find:

\[
\forall s \in S : s \in T \land s \in \text{dom}(R \land P) \Rightarrow s \in T \land s \in \text{dom}(R \land P'),
\]

Using the same argument as the proof of the previous proposition, we find:

\[
\forall s \in S : s \in \text{dom}(T,R \land P) \Rightarrow s \in \text{dom}(T,R \land P').
\]

From which we infer, by rewriting in relational form:

\[
(T \land R \land P) \subseteq (T \land R \land P').
\]

In other words, \( P' \) is more-correct than \( P \) with respect to \( T \).

**Proof of Necessity.** If \( P' \) is more-correct than \( P \) with respect to \( T \) then \((T \land R \land P) \subseteq (T \land R \land P')\), which we represent by the following logic formula:

\[
\forall s \in S : s \in \text{dom}(T \land R \land P) \Rightarrow s \in \text{dom}(T \land R \land P').
\]

If this formula holds for all \( s \) in \( S \), it holds necessarily for all \( s \) in \( T \).

\[
\forall s \in T : s \in \text{dom}(T \land R \land P) \Rightarrow s \in \text{dom}(T \land R \land P').
\]

By factoring out the pre-restriction from the domain, we get:

\[
\forall s \in T : s \in T \land s \in \text{dom}(R \land P) \Rightarrow s \in T \land s \in \text{dom}(R \land P').
\]

We remove the clause \((s \in T)\), which is now redundant:

\[
\forall s \in T : s \in \text{dom}(R \land P) \Rightarrow s \in \text{dom}(R \land P').
\]

Because \( P \) and \( P' \) are deterministic, this formula can be written as:

\[
\forall s \in T : (s,P(s)) \in (R \land P) \Rightarrow (s,P'(s)) \in (R \land P').
\]

Using the Boolean manipulations we showed in the previous proof, we find this to be equivalent to:

\[
\forall s \in T : (s,P(s)) \in (R \land P) \Rightarrow (s,P'(s)) \in (R \land P').
\]

Using the formula of the oracle of absolute correctness with respect to \( R \), we find:

\[
\forall s \in T : \Omega(s,P(s)) \Rightarrow \Omega(s,P'(s)).
\]

**qed**

**C. Strict Relative Correctness**

We are given a specification \( R \) on space \( S \) and a program \( P \) on \( S \). The two previous oracles had the form of a predicate over \( S \times S \), which we quantify universally over some test data \( T \) to infer a correctness property with respect to \( R \). The oracle of strict relative correctness is the conjunction of two formulas, bearing different quantifiers; hence it has a different form. We consider a program \( P' \) on \( S \) and we want to write an oracle that checks whether \( P' \) is strictly more-correct than \( P \) with respect to \( R \).

**Definition 10:** Given a specification \( R \) on space \( S \), a subset \( T \) of \( S \) and a program \( P \) on \( S \), the oracle of strict relative correctness over \( P \) with respect to \( R \) is denoted by \( \sigma_T() \) and defined by:

\[
\sigma_T(P') \equiv (\forall s \in T : \omega(s,P'(s)) \land (\exists s \in T : \lnot \Omega(s,P(s)) \land \Omega(s,P'(s))).
\]

The following proposition justifies this definition.
Proposition 4: Let $\sigma_T()$ be the oracle of strict relative correctness over program $P$ with respect to specification $R$ and let $T$ be a subset of $S$. A program $P'$ is strictly more-correct than $P$ with respect to $T \cap R$ if and only if oracle $\sigma_T(P')$ returns true.

Proof. Proof of Sufficiency. Let program $P'$ satisfy the oracle of strict relative correctness; then according to the definition of this oracle, it satisfies the condition \((\forall s \in T : \omega(s, P'(s)))\).

By proposition 3, $P'$ is more-correct than $P$ with respect to $T \cap R$, i.e. \((T \cap R \cap P') \subseteq (T \cap R \cap P)\). To prove strict relative correctness, we must prove that there exists an element $s$ in the domain of $(T \cap R \cap P')$ that is not in the domain of $(T \cap R \cap P)$. To this effect, we consider the second clause of the oracle:

\[(\exists s \in T : \neg \Omega(s, P(s)) \land \Omega(s, P'(s))).\]

By the definition of $\Omega(s, P)$, we find:

\[(\exists s \in T : s \in \text{dom}(R) \land (s, P(s)) \notin R \land (s \in \text{dom}(R) \Rightarrow (s, P'(s)) \in R).\]

Using Boolean identities, we simplify this to:

\[(\exists s \in T : s \in \text{dom}(R) \land (s, P(s)) \notin R \land (s, P'(s)) \in R).\]

Since $(s, P'(s)) \in R$ logically implies $s \in \text{dom}(R)$, we write:

\[(\exists s \in T : (s, P(s)) \notin R \land (s \notin \text{dom}(R) \land (s, P'(s)) \in R).\]

From $(s \in T \land (s, P'(s)) \in R$ we easily infer $s \in \text{dom}(T \cap R \cap P')$, following the same argument that we used in the proofs of propositions 2 and 3. From $(s, P'(s)) \notin R$ we infer $s \notin \text{dom}(T \cap R \cap P)$, whence we infer $s \notin \text{dom}(T \cap R \cap P)$, since $T \cap R \subseteq R$.

Proof of Necessity. If $P'$ is strictly more-correct than $P$ with respect to $T \cap R$, then it is more-correct, hence by proposition 3, $(\forall s \in T : \omega(s, P(s)))$. On the other hand, we know that there exists an element of $\text{dom}(T \cap R \cap P')$ that is not in $\text{dom}(T \cap R \cap P)$. Using the same arguments cited in the proof of proposition 3, we infer: $(s, P'(s)) \notin T \cap R \land (s, P'(s)) \in T \cap R$. From the second clause we infer that $s$ is in $T$, which we use to rewrite the formula as: $(s, P(s)) \notin T \cap R \land (s, P'(s)) \in T \cap R$. Using the Boolean transformation alluded to above, we find this to be equivalent to: $(s, P(s)) \notin \text{dom}(T \cap R \cap P')$. QED

The test driver for strict relative correctness is written as:

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

V. FAULT DENSITY VS. FAULT DEPTH

In this section we use the oracles presented in the previous section to illustrate the contrast between fault density and fault depth, and to show that these two metrics take different values and follow different laws. We consider the tot-info component of the Siemens benchmark [4]; this component has 307 LOC and comes with a test data set $(T)$ of size 1052. We seed this program with 7 changes (faults?) that come with the benchmark, and we run an experiment intended to locate and repair these faults. We adopt a generate-and-validate policy, and we use a mutant generator for the generation phase, and our oracle infrastructure (section IV) for the validation phase.

- **Mutant Generation.** We use the mutant generator Proteum / IM 2.0 [8], where we activate the following operators:
  - Mutation of a relational operator,
  - Mutation of an arithmetic operator,
  - Mutation of a shorthand assignment operator.

For the purposes of this experiment, we take these operators as our atomic change operators; with these operators, Proteum generates 212 mutants whenever it is called.

- **Validation.** For the purposes of this experiment, we use the original version of tot-info (prior to fault seeding) as the specification $R$; this specification determines the oracle of absolute correctness, which in turn determines the oracles of relative and strict relative correctness. Starting from the seeded version of tot-info, we generate all its mutants, then test them for absolute correctness and for strict relative correctness over the base program. If a mutant is found to be absolutely correct, then no further mutation thereof is attempted; if a mutant is found to be strictly more-correct than the base, then this mutant is used as the new base and the process is resumed.

We catalog all the relations of strict relative correctness between the mutants, and we feed them to a graph drawing application, Graphviz (http://www.graphviz.org/). The resulting graph is shown in Figure 4.

Every node of this graph represents a program (obtained by mutation of tot-info) and every arc represents a strict relative correctness relation (the higher node is strictly more-correct than the lower node). The node at the bottom of the graph is the seeded version of tot-info (with the seven changes provided in the benchmark) and the node at the top of the graph is the original (correct) version; we refer to it as tot-info'. All the nodes in the graph are mutations of tot-info, obtained by successive applications of the mutant generator. We make the following observations about this graph:

- Even though tot-info has been seeded with seven faults, it has only 4 faults (only 4 arcs outgoing from the bottom of the graph); its fault density is 4.
- While its fault density is four, tot-info has a fault depth of seven: seven arcs (fault removals) separate it from the top of the graph. Because we adopt Proteum as our set of atomic change operators, and because at each node we have explored all the mutations generated by Proteum, all possible paths are shown in this graph.
- The fault depth of each node decreases by one with each fault removal.
- The fault density evolves erratically from one node to the next; note that whereas the node at the bottom of the graph has a fault density of four, the nodes that are derived from it by one fault removal do not have a density of three. Three have a density of four, and one
has a density of five! Close inspection of the graph shows that sometimes removal of a fault keeps the fault density constant and sometimes increases it: removal of a fault may expose faults that were so far masked.

VI. FAULT DEPTH VS. FAULT MULTIPLICITY

Whereas in the previous section we use the example of tot-info to illustrate the contrast between fault density and fault depth, in this section we use the example of replace, another component of the Siemens benchmark, to illustrate the contrast between fault depth and fault multiplicity. As we recall, the fault depth of a program with respect to a specification is the minimal number of elementary fault removals that separate the program from a correct program; and the fault multiplicity of an elementary fault in a program is the number of syntactic atoms that make up the fault. Why is this distinction important? Because we argue that:

• When we resolve to repair a program, we must do so one fault at a time, for the sake of efficiency (to control the size of the search space).
• But removing one elementary fault at a time does not necessarily mean applying single atomic changes; as some elementary faults span more than one syntactic atom, as we can see in Figure 2. Hence we may have to apply multiple mutations to detect a single elementary fault.
• The question that this raises is: how many successive mutations do we have to apply? or, what determines the number of successive mutations we resolve to apply? We argue that the number of successive mutations we ought to generate is determined by the multiplicity of elementary faults we seek to identify.
• If we were testing repair candidates for absolute correctness rather than relative correctness, then the number of successive mutations we would have to apply is the fault depth of the program (even if each elementary fault were a single syntactic atom).
• Fault depth is typically unknown and unbounded, but fault multiplicity can be bounded: if we are interested to repair single site faults, then we apply single mutations; if we are interested to also repair elementary faults that span two syntactic atoms, then we apply double mutation; etc. In practice, by limiting fault multiplicity to 2 or 3, we probably cover the vast majority of actual faults.

In this experiment, we take the replace component of Siemens (of size 563 LOC), seed it with six changes (faults?)
provided as part of the benchmark, and iteratively apply the mutant generator to the seeded program, as we did for tot-info, but with some differences:

- We apply two mutation operators of Proteum, namely: mutation of a logical operator; and mutation of a relational operator. For this choice of mutation operators, each call to Proteum produces 90 mutants.
- We apply single mutations at each node, except if we reach a node that is not absolutely correct, and such that no mutant thereof is strictly more-correct.
- When we reach such a node, we apply double mutations. The resulting graph is shown in Figure 5; each node is identified by a number between 1 and 90. The benchmark test data that we used to run the oracles has 5542 elements.

![Fig. 5. replace: Fault Depth vs. Fault Multiplicity](image)

Inspection of this graph yields the following observations:

- The contrast between the number of changes applied to a program and the number of faults: Even though the seeded program (at the bottom of the graph) stems from 6 changes to the original (correct) replace program, it has only one fault (only one arc going up).
- The usual observation about the erratic behavior of fault density as faults are removed: While \( P (= \text{replace}) \) has only one fault, removal of that fault yields a program \( m79 \) that has (not zero but) three faults.
- All the fault removals (represented by arcs) below \( m79.3.42.47 \) have multiplicity 1, i.e. were achieved by single mutations. Mutant \( m79.3.42.47 \) is not absolutely correct, and no single-mutant thereof was found to be strictly more-correct than it. Hence we deploy double mutations, and we find two programs, \( m79.3.42.47.36.85 \) and \( m79.3.42.47.37.85 \), that are absolutely correct, hence strictly more-correct than \( m79.3.42.47 \). The latter is actually the original (unseeded/correct) version of replace.

In this experiment, we are able to remove all the faults seeded into the replace program by no more than double mutation. If we were testing candidate repairs for absolute correctness rather than relative correctness, we would have to apply six consecutive mutations before we generate a correct program, which in this case would create a search space of size \((90^6 =)\) half a trillion elements. By testing for relative correctness, we scan search spaces of no more than \((90^2 =)\) 8100 elements (in all but one case, only 90 elements).

More generally, to estimate the difference in \( \text{Big Oh}(\cdot) \) performance between an algorithm that tests candidates for absolute correctness and an algorithm that tests for relative correctness, we consider the following parameters:

- The number of program variants that are generated with each step of patch generation, say \( F \) (for fan-out); in the case of tot-info, we had \( F = 212 \), and in the case of replace, we had \( F = 90 \).
- For a minimal sequence of elementary fault removals, the sum of multiplicities of all the elementary faults in the sequence, say \( \delta \). If all the faults in question involve a single syntactic atom, then \( \delta \) is the same as the fault depth of the program; but if some faults have higher multiplicity, then \( \delta \) is greater than the program’s fault depth. In the case of tot-info, \( \delta = 7 \) (the same as depth, since all elementary faults involve a single mutation); in the case of replace, \( \delta = 6 \) (whereas \( \text{depth} = 5 \)).
- For the same minimal sequence of elementary fault removals, the maximum multiplicity of faults in the sequence, say \( \mu \). In the case of tot-info, \( \mu = 1 \); in the case of replace, \( \mu = 2 \) (one elementary fault has a multiplicity of 2).

An algorithm that tests candidates for absolute correctness must inspect a number of variants (variants/candidate repairs) in the order of \( O(F^\delta) \), whereas an algorithm that tests candidates for relative correctness must only inspect a number of variants in the order of \( O(F^\mu) \); not only is \( \delta \) typically much larger than \( \mu \) (the sum of multiplicities vs. their maximum), but \( \delta \) is usually unknown and unbounded.

VII. RCFix: Fault Removal at Arbitrary Fault Depth

Our discussions on the difference between density, depth and multiplicity is not a mere academic exercise; we believe they give us the required vocabulary of concepts to reason about faults and fault repairs. In [19] we present a generic algorithm for program repair that proceeds by unitary increments of relative correctness. Each increment performs an elementary fault removal, trying to do so by single atomic change, then by two atomic changes, then by three atomic changes, etc.
until it succeeds or until it reaches a user-provided threshold (and exits on a failure). This algorithm is generic in the sense that it details how candidate repairs are validated but does not specify how they are generated; hence we obtain a different instance thereof for each generation method. In [18] we create an instance of the generic algorithm, called RCFix which uses GenProg's patch generation [13]. Experimentation of RCFix on benchmark programs and faults of Defects4J show the superiority of relative correctness-based program repair: in all the examples that we tested (see [18]), which involve three benchmark faults, RCFix finds and repairs the three faults one by one, within a short amount of time (varying between a few minutes and two hours); by contrast, GenProg times out after 9 hours (time limit specified as a threshold) without repairing any of the faults. We believe that the reason for this discrepancy stems from the fact that GenProg is testing repair candidates for absolute correctness (or some approximation thereof), hence does not converge unless it has repaired simultaneously all three faults; when we inspect the variants that GenProg generates, we find that in some instances it has repaired one or two of the faults, but fails to recognize this, because the variant still falls short of absolute correctness as long as the third fault is not removed. The advantage of removing faults one at a time is not only that we have a much smaller search space ($F^n$ vs. $F^3$), but also that by executing fault localization after each fault removal, we ensure that each fault removal enables us to target the following fault with greater precision.

Whereas the experiments reported in [18] involve single site faults, the experiments we discuss here involve elementary faults of multiplicity 2.

A. A Two-Site Fault

We consider the following two-site fault in the complex class of Defect4J, which we call Math-Complex-2:

- Site 1: `-double real2 = 2.0*real1;+ double real2 = 4.0*real1; double imaginary2 = 2.0*imaginary1;
  double d = Math.cos(real2)+ MathUtils.cosh(imaginary2);
- Site 2: `+d=Math.cos(real2)+ MathUtils.cosh(imaginary2); return createComplex(Math.sin(real2)/d,
  MathUtils.sinh(imaginary2)/d);

We run RCFix on this class, with the following parameters:

- Number of failing tests: 2.
- Number of modification points: 4.
- Threshold probability to modify a location: 0.1.
- Maximum execution time: 9 hours.

RCFix performs simple mutations, generates 462 candidates and finds that none of them is absolutely correct nor strictly more correct than the base program; this step takes 963 seconds. Then it takes the first candidate, generates 462 mutants thereof, and finds that none of them is absolutely correct nor strictly more-correct than the selected candidate; the operation takes 1065 seconds. Then it takes the second candidate, generates mutants thereof, which it tests for absolute correctness and strict relative correct; the mutant number 334 that it generates proves to be absolutely correct; the operation takes 712 seconds. The total time is 963+1065+712 = 2740 seconds, i.e. about three quarters of an hour.

We run GenProg on the same program and fault, with the following parameters:

- Threshold probability to modify a location: 0.1.
- Maximal number of generations: 500.
- Population size: 40.
- Maximum execution time: 9 hours.

GenProg times out after 9 hours without repairing the fault; we do not have a satisfactory explanation for why GenProg fails in this case. We cannot blame combinatorics since RCFix is dealing with the same level of combinatorics as GenProg; this matter is under investigation (if GenProg has strategies for pruning the search space, it may have prematurely eliminated a valid candidate in the process).

B. A Two-Site Fault, Two Single-Site Faults

In this experiment, we combine the two-site fault of the previous example with two faults of Defect4J [17]:

- Math70: - return solve(min,max);
  + return solve(f,min,max);
- Math73: + if (yMin*yMax>0) {throw
  MathRuntimeException.createIllegalArgumentException
  (NonBracketing_Message,min,max,yMin,yMax));}

We run RCFix with the following parameters:

- Number of failing tests: 4.
- Number of modification points: 4.
- Threshold probability to modify a location: 0.1.
- Maximum execution time: 9 hours.

RCFix repairs Math-Complex-2 in the same amount of time as in the previous example, i.e. 2740 seconds (slightly more than 46 minutes), then it repairs Math70 in 25 seconds and Math73 in 23 seconds. It repairs all three faults (four atomic changes) in less than 48 minutes.

GenProg is executed on this example, with the same parameters as above (section VII-A); it times out after nine hours without repairing the faults. A possible explanation is that since GenProg tests for absolute correctness, it can terminate successfully only if it corrects all three faults (four atomic changes) simultaneously; the combinatorics of this search are off the charts.

VIII. CONCLUSION

Given that software faults are the focus of much software engineering research (fault avoidance, fault removal, fault tolerance, fault forecasting, etc.), one would think that a formal definition of faults is perhaps overdue. In this paper we use a semantic-based definition of faults to shed light onto some attributes of faults and fault repairs. These include:

- Distinction Between Fault Density and Fault Depth. In the absence of formal definitions, we tend to think of fault density (the number of elementary faults) and fault depth (the minimal number of elementary fault removals that separate a program for absolute correctness) as being identical. In section V we discuss how these two
metrics usually take different values, and have different properties.

- **Fault Depth as a Measure of Faultiness.** Whereas program faultiness is usually quantified by fault density, we argue that a better measure of faultiness is fault depth; whereas repairing a program fault does not necessarily decrement fault density (and may in fact increase it), it preserves or reduces fault depth; this is discussed in section V.

- **Distinction Between Fault Density and Fault Multiplicity.** Because some faults may span more than one syntactic atom, the number of faults in a program and the number of atoms that may be changed to repair a program are distinct; this is discussed in section VI.

- **Distinction Between Fault Repair and Failure Remediation.** Failure remediation consists in altering a program to make it work correctly for some input (where it failed before); fault repair aims for a more modest goal, which is merely to make an incorrect program more-correct. This allows the program to expose its faults one by one, in the order it determines, and enables us to control combinatorial explosion; this is discussed in section VII.

- **Distinction Between Preserving Correctness and Preserving Correct Behavior.** Because specifications are generally vastly non-deterministic, they admit more than one correct behavior; hence it is possible to preserve (or enhance) correctness without preserving a program’s correct behavior; this is discussed in section VII.

Our prospects for future research include the use of relative correctness to enhance patch generation, in addition to patch validation. Also, we envision to extend the work we did on GenProg to other tools and methods.

**REFERENCES**


