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 mutiplicity.**

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 [2], [17]–[19] Laprie et al. define a fault as the *adjudged or hypothesized cause of an error* [2]. 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*". In [10] Gopinath et al. define a fault as *an erroneous part of a program, the syntactic source of a semantically wrong behavior*. None of these definitions provides a sound characterization of faults: The IEEE definition does not even try; the definition of Laprie et al. [2] depends on the definition of an *error*, which in turn assumes that we have a specification of correct states at each step of a computation, clearly an unrealistic assumption; the definition of Gopinath et al. assumes that each semantically wrong behavior has a unique, well-defined source, clearly an unfounded assumption.

In this paper, we adopt a definition of faults (due to [23]) that has the following attributes:

- *A Level of Granularity*. Any definition of a fault must refer to a level of syntactic granularity at which we want to isolate faults. 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; a *syntactic feature* is the aggregate of one or more syntactic atoms.
- *Atomic Change Operators*. Following Gazzola et al. [9], 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 scale of fault removals.
- *A Program and a Specification*. Whereas Laprie et al [2] define faults in terms of errors, we define faults in terms of failures with respect to a specification.
- *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.
- *Faults and Fault Removals*. A fault ia a feature that can be changed to enhance the relative correctness of an incorrect program, and fault removal is the act of performing the change.
- *Faults and Elementary Faults*. We cannot talk about faults in the plural unless we define the concept of a unitary, indivisible 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 mutiplicity* as the number of syntactic atoms that form an elementary fault.

Given that we have been practicing fault removal (debugging) for decades without a definition of faults, one may wonder why we need such a definition. To address this question, we explore the application of our definitions to automated program repair. The field of program repair has itself achieved significant success of its own [5], [7], [11], [12], [20]–[22], [25]–[28], [28], [29], but we argue that the absence of a formal definition of faults has come at a non-negligible cost.

- *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. By using absolute correctness, current methods are restricted to programs that are within striking distance of absolute correctness.
- *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 for patch validation, while others use fitness functions. We argue in [8] that the former option causes a loss of recall, whereas the latter option causes a loss of precision.
- *Fault Repair vs. Failure Remediation*. Current repair methods define success as remedying a failure, hence can only converge if all the faults that cause the failure

This work is partly supported by NSF through grant # DGE1565478.

are repaired simultaneously —a recipe for combinatorial divergence; we favor a stepwise approach where success is defined merely as correctness enhancement.
- *Fault Dependency*. Fault localization tools attempt to identify suspicious statements by highlighting statistical relationships between program failures and statement activations [9]. When, in addition to removing faults one at a time we also run fault localization after each fault removal, we ensure that each fault repair helps us diagnoze the next fault.
- *Fault Depth and Fault Multiplicity*. When we resolve to remove faults one at a time, the only time we ever have to apply multiple mutations to a program is when we are dealing with a multi-site fault. Hence the number of simultaneous atomic changes we have to apply is determined by fault multiplicity (which is seldom greater than 2 or 3) rather than fault depth (typically unknown and unbounded).

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 mutiplicity. In section VII we show how we can improve the performance of GenProg by enhancing its patch validation using relative correctness.

II. Mathematics for Relative Correctness

We assume the reader familiar with elementary relational algebra [4], hence we merely focus on presenting some definitions and notations. We represent sets in a program-like notation; if we write S as: `x: X; y: Y;` then we mean to let S be the cartesian product $S = X \times Y$; elements of S are denoted by s and the X- (resp. Y-) component of s is denoted by $x(s)$ (resp. $y(s)$); 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 operations of union, intersection, and complement; they also include the *converse* of a relation R defined by $\widehat{R} = \{(s,s')|(s',s) \in R\}$, the *domain* of a relation defined by $dom(R) = \{s|\exists s' : (s,s') \in R\}$, and the product of two relations defined by: $R \circ R' = \{(s,s')|\exists s'' : (s,s'') \in R \wedge (s'',s') \in R'\}$; we may write RR' for $R \circ R'$. The *pre-restriction* of relation R to set T is the relation denoted by ${}_{T\backslash}R$ and defined by: ${}_{T\backslash}R = \{(s,s')|s \in T \wedge (s,s') \in R\}$.

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

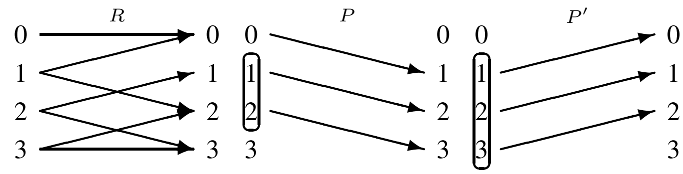

Fig. 1. $P' \sqsupseteq_R P$, Deterministic Programs

A. Program Semantics

Given a program p on space S, 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 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: $R' \sqsupseteq R$ or $R \sqsubseteq R'$) if and only if $RL \cap R'L \cap (R \cup R') = R$.

Intuitively, this definition means that R' has a larger domain than R and that on the domain of R, R' assigns fewer images to each argument than does R.

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

This definition is equivalent to traditional definitions of (total) correctness [13].

Proposition 1: Due to [24]. Given a specification R and a deterministic program P, program P is correct with respect to R if and only if $(R \cap P)L = RL$.

We refer to the domain of $(R \cap P)$ as the *competence domain* of P with respect to R.

Definition 3: Due to [23]. 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. $(R \cap P')L \supset (R \cap P)L$).

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. In [8] Diallo et al. prove that this definition of relative correctness satisfiees the following properties (which, we argue, any definition of relative correctness must satisfy):

- 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.

III. Faults and Fault Removals

A. Faults

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. As an illustration, we consider the following space:

```
float x; float a[N+1];
```

We let R and P be, respectively, the following specification and program on space S:

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

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

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:

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

Program P' is absolutely correct, as can be checked readily, hence it is strictly more-correct than P; therefore the feature $f = \langle 0, < \rangle$ (where 0 is the initialization of i and $<$ is the condition of the loop) is a fault in 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.
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. In the example above, we find the following fault removal: $(\langle 0, < \rangle, \langle 1, \leq \rangle)$ at the appropriate locations.

C. Elementary Faults

According to the definition of a fault, if f_1 and f_2 are faults in program P with respect to specification R, then so is the aggregate $\langle f_1, f_2 \rangle$ (by transitivity of relative correctness). Hence we must distinguish between a fault that spans two (or more) syntactic atoms 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.
If f is defined by a set of syntactic atoms $f = \{a_1, ..., a_k\}$ then no subset thereof is a fault. This definition is important because we cannot talk about faults in the plural (2 faults, 3 faults, etc..) if we cannot even tell the difference between one fault and two faults. We consider the array sum program discussed above and we ponder the question of whether $f = \langle 0, < \rangle$ is a single two-site fault or two single-site faults. To this effect, we consider the following programs:

```
P1' = {int i=1; x=0;
      while (i<N) {x=x+a[i];i=i+1;}}
P2' = {int i=0; x=0;
      while (i<=N) {x=x+a[i];i=i+1;}}
```

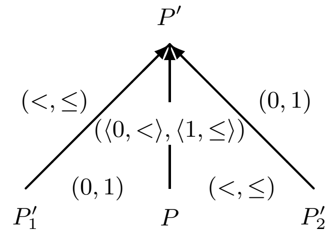

Fig. 2. A Single Fault, Multiplicity 2

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 or P'_2 is strictly more-correct than P. The functions of P, P'_1 and P'_2 are:

$P = \{(s, s')|i' = N \wedge a' = a \wedge x' = \sum_{k=0}^{N-1} a[k]\}$.
$P'_1 = \{(s, s')|i' = N \wedge a' = a \wedge x' = \sum_{k=1}^{N-1} a[k]\}$.
$P'_2 = \{(s, s')|i' = N \wedge a' = a \wedge x' = \sum_{k=0}^{N} a[k]\}$.

The competence domains of P, P'_1 and P'_2 with respect to R are, respectively:

$CD = \{s|a[0] = a[N]\}$.
$CD'_1 = \{s|a[N] = 0\}$.
$CD'_2 = \{s|a[0] = 0\}$.

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 space S defined by the following program variable:

```
int a[N+1];
```

and we consider specification R and program P defined as follows:

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

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

The function of P is:

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

The competence domain of P is:

$CD = \{s|a[0] = 0 \wedge a[N] = 0\}$.

Since $CD \neq dom(R)$, 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.

$P'_1 = \{(s, s')|a[0] = a'[0] \wedge \forall k : 1 \leq k \leq N - 1 : a'[k] = 0 \wedge a[N] = a'[N]\}$.
$P'_2 = \{(s, s')|\forall k : 0 \leq k \leq N - 1 : a'[k] = 0 \wedge a[N] = a'[N]\}$.
$P' = \{(s, s')|a[0] = a'[0] \wedge \forall k : 1 \leq k \leq N : a'[k] = 0\}$.

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

$CD'_1 = \{s|a[N] = 0\}$.
$CD'_2 = \{s|a[0] = 0\}$.
$CD' = S$.

Considering the inclusion relations between CD, CD'_1, CD'_2

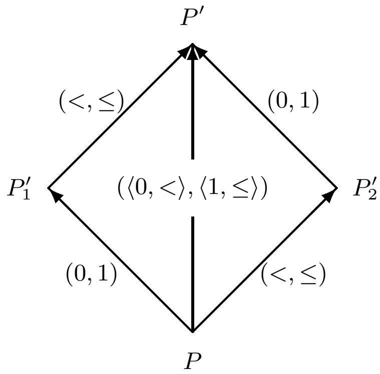

Fig. 3. Two Faults, Multiplicity 1

and CD', we infer that P'_1 and P'_2 are 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

We are interested to define *faultiness*, i.e. the extent to which a program has faults; whereas relative correctness, a semantic attribute, reflects the preponderance of failures, we want faultiness to reflect the preponderance of faults, a syntactic attribute.

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. 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).

Fault depth and fault density do not take the same value, nor do they follow the same laws. Just because we have N faults does not mean we must perform N fault removals before the program is correct. Also, if P' is derived from P by an elementary fault removal then we necessarily have $depth(P) \geq depth(P') + 1$. Equality occurs if P' is on a minimal path from P to a correct program. By contrast, we know of no relation between $density(P)$ and $density(P')$.

IV. A Hierarchy of Oracles

In this section we use a specification to derive an oracle of absolute correctness; then we use the oracle of absolute correctness to derive 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 can be used as follows:

```
inits = s; //save the initial state
P(); // program under test, modifies s
assert(oracle(inits,s)); //check correctness
```

A. 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:

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

The following proposition shows that this definition is sound; we admit it without proof.

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 ${}_{T\backslash}R$ (the pre-restriction of R to T) if and only if execution of P on every element of T satisfies oracle $\Omega(s,s')$.

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

```
bool absoluteCorrectness(testData T)
   {statetype inits,s;  bool abscor=true;
    while (moretestdata(T))
       {inits =gettestdata(T);   //load test datum
        s = inits;p();   // modifies s, not inits
        abscor = abscor && absoracle(inits,s);}
    return abscor}
bool absoracle(statetype s, sprime)
   {return (!domR(s) || R(s,sprime))}
```

where `absoracle(s,sprime)` checks the correctness of P for a single execution and `absoluteCorrectness(T)` checks the absolute correctness of P with respect to the restriction of R to T.

B. Relative Correctness

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; we admit it without proof.

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 ${}_{T\backslash}R$ if and only if execution of P' on every element of T satisfies oracle $\omega(s,s')$.

Based on this proposition, we derive the following oracle:

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

C. Strict Relative Correctness

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 ${}_{T\backslash}R$ is denoted by $\sigma_T()$ and defined by:

$\sigma_T(P') \equiv (\forall s \in T : \omega(s, P'(s))) \wedge (\exists s \in T : \neg\Omega(s, P(s)) \wedge \Omega(s, P'(s)))$.

The following proposition justifies this definition; we admit it without proof.

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 ${}_{T\backslash}R$ if and only if oracle $\sigma_T(P')$ returns true.

The test driver for strict relative correctness is written as:

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

V. Fault Density vs. Fault Depth

We consider the `tot-info` component of the Siemens benchmark [3]; 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* [6], where we activate the following operators:
  - Mutation of a relational operator,
  - Mutation of an arithmetic operator,
  - Mutation of a shorthand assignment operator.

  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. 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; else 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 and every arc represents a strict relative correctness relation. The node at the bottom of the graph is the seeded version of `tot-info` and the node at the top of the graph is the original (correct) version; we refer to it as `tot-info'`. 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.
- 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 undetected by T.

VI. Fault Depth vs. Fault Mutiplicity

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 syntatic 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.
- 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.
- The question that this raises is: how many successive mutations do we have to apply? 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 by the maximum fault multiplicity we are seeking.

In this experiment, we take the `replace` component of Siemens (of size 563 LOC), seed it with six changes, 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 rela-

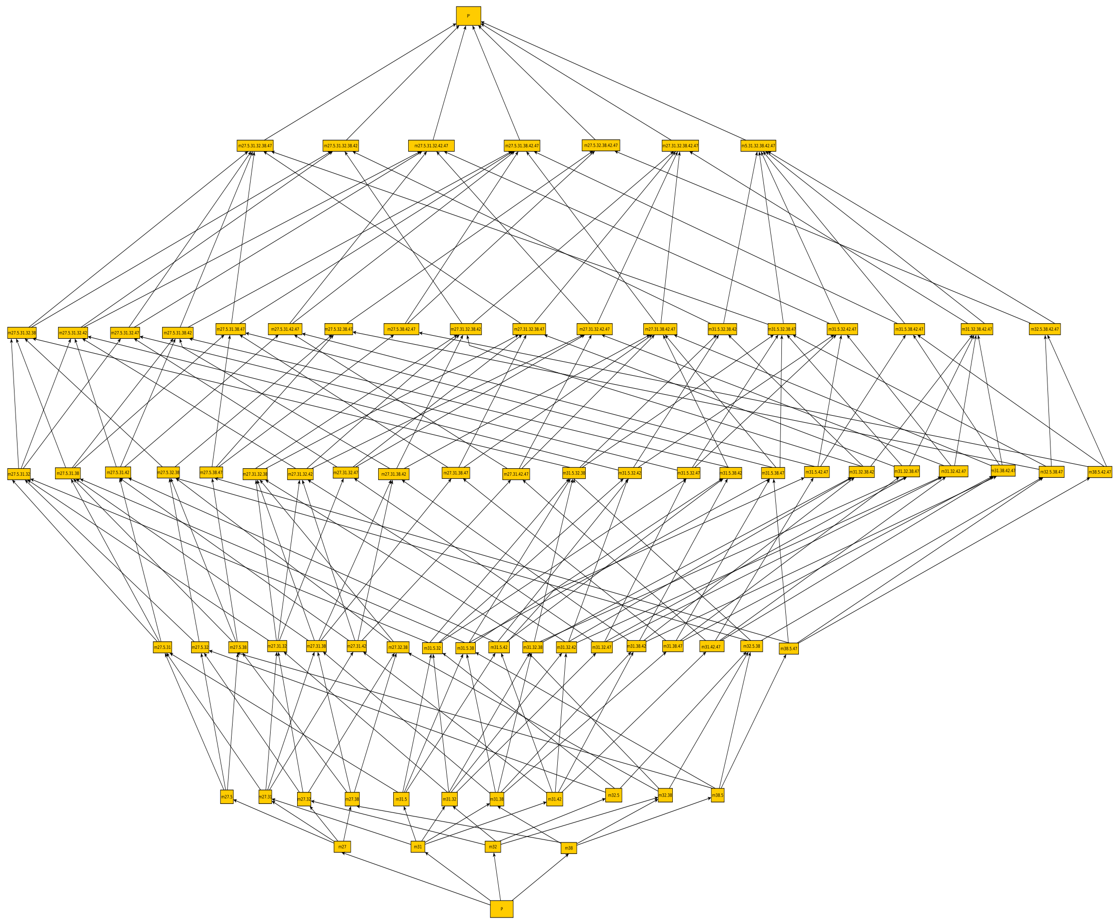

Fig. 4. `tot-info`: Fault Density vs. Fault Depth

tional 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 the sequence of mutations that produce it (each mutation is identified by a number between 1 and 90). The benchmark test data that we used to run the oracles has 5542 elements. 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 stems from 6 changes to the original `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 (=`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.

More generally, to estimate the difference in *Big Oh()* 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

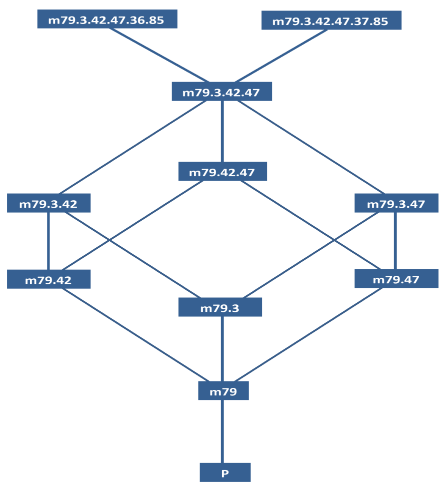

Fig. 5. `replace`: Fault Depth vs. Fault Multiplicity

`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 δ. If all the faults in question involve a single syntactic atom, then δ is the fault depth of the program.
- For the same minimal sequence of elementary fault removals, the maximum mutiplicity of faults in the sequence, say μ.

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 δ typically much larger than μ (the sum of multiplicities vs. their maximum), but δ is usually unknown and unbounded.

VII. RCFix: Fault Repair at Arbitrary Fault Depth

In [16] 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, else by two atomic changes, else by three atomic changes, etc. until it succeeds or until it reaches a user-provided threshold. In [15] we create an instance of the generic algorithm, called *RCFix* which uses GenProg's patch generation [11]. Experimentation of RCFix on benchmark programs and faults of Defects4J shows the superiority of relative correctness-based patch validation: in all the examples that we tested, which involve three benchmark faults, RCFix finds and repairs the three faults one by one, within a short amount of time whereas GenProg times out after 9 hours wihout repairing any fault; we speculate that because GenProg tests candidate repairs for absolute correctness, it converges only if it can repair all three faults simultaneously. By contrast, RCFix repairs faults one at a time, and deploys fault localization after each fault repair, hence each fault repair helps diagnose the next fault. Whereas the experiments reported in [15] involve single site faults, the experiments we discuss here involve elementary faults of mutiplicity 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*real;+ double real2 = 4.0*real; double imaginary2 = 2.0*imaginary; double d = Math.cos(real2)+ MathUtils.cosh(imaginary2);`
- Site 2: `+d=Math.cos(real2)*2.0*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; 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.

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.

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 [14]:

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

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 and fault depth as being identical.
- *Distinction Between Fault Density and Fault Multiplicity*. Because faults may span more than one atom, the number of faults in a program and the number of atoms that must be changed to repair a program are distinct.
- *Distinction Between Fault Repair and Failure Remediation*. Failure remediation consists in altering a program to make it work correctly for some input; fault repair aims for a more modest goal, which is merely to make an incorrect program more-correct.
- *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.
- *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: repairing a program does not necessarily decrement fault density, but it maintains or reduces fault depth.

Our prospects for future research include the use of relative correctness to enhance patch generation.

References

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

[2] 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] Benchmark. Siemens suite. Technical report, Georgia Institute of Technology, January 2007.

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

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

[6] Marcio Eduardo Delamaro, Jose Carlos Maldonado, and Auri Marcelo Rizzo Vincenzi. Proteum /im 2.0: An integrated mutation testing environment. In W. Eric Wong, editor, *Mutation Testing for the New Century*, volume 24, pages 91–101. Springer Verlag, 2001.

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

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

[9] Luca Gazzola, Daniela Micucci, and Leonardo Mariani. Automatic software repair: A survey. *IEEE Trans. on Soft. Eng.*, 45(1), January 2019.

[10] Rahul Gopinath, Amin Alipour, Iftekhar Ahmed, Carlos Jensen, and Alex Groce. Measuring effectiveness of mutant sets. In *Proceedints, Ninth International Conference on Software Testing*, Chicago, IL, April 11-15 2016.

[11] C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer. Genprog: A generic method for automated software repair. *IEEE Transactions on Software Engineering*, 31(1), 2012.

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

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

[14] Rene Just, Darioush Jalali, and Michael D Ernst. Defects4j: A database of existing faults to enable controlled testing studies for java programs. In *Proceedings, ISSTA 2014*, pages 437–440, San Jose, CA, USA, July 2014.

[15] Besma Khaireddine, Matias Martinez, and Ali Mili. Program repair at arbitrary fault depth. In *Proceedings, ICST 2019 Tools Track*, Xi'An, China, April 2019.

[16] Besma Khaireddine, Aleksandr Zakharchenko, and Ali Mili. A generic algorithm for program repair. In *Proceedings, FormaliSE*, Buenos Aires, Argentina, May 2017.

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

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

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

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

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

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

[23] A. Mili, M. Frias, and A. Jaoua. On faults and faulty programs. In P. Hoefner, P. Jipsen, W. Kahl, and M. E. Mueller, editors, *Proceedings, RAMICS 2014*, volume 8428 of *LNCS*, pages 191–207, 2014.

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

[25] Martin Monperrus. A critical review of patch generation learned from human written patches: Essay on the problem statement and evaluation of automatic software repair. In *Proceedings, ICSE 2014*, Hyderabad, India, 2014.

[26] Zhchao Qi, Fan Long, Sara Achour, and Martin Rinard. An analysis of patch plausibility and correctness for generate-and-validate patch generation systems. In *Proceedings, ISSTA 2015*, Baltimore, MD, July 2015.

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

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

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