Defining Software Faults
Why It Matters

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

Aleksandr Zakharchenko, Ali Mili
New Jersey Institute of Technology
Newark, NJ USA
{az68,mili}@njit.edu

Abstract—Because faults are at the center of software quality concerns, they ought to be defined formally, by semantics-based criteria that enable us to reason about them. In this paper, we consider a semantics-based definition of a fault, which involves the program, the faulty feature (at the appropriate level of granularity) and the specification against which correctness and incorrectness are defined. We explore the implications of this definition for various aspects of software testing, software reliability, and software repair; and we argue that providing a formal, verifiable definition of faults is not a mere intellectual exercise, but has important practical applications.

Keywords—fault, fault removal, relative correctness, correctness enhancement, software testing, software reliability, software repair.

I. INTRODUCTION

In (Avizienis, Laprie, Randell, & Landwehr, 2004) Avizienis et al. define a fault as the adjudged or hypothesized cause of an error; an error, in turn, is defined as deviation of the system state from the correct state. The IEEE Standard IEEE Std 7-4.3.2-2003 defines a software fault as an incorrect step, process or data definition in a computer program. Whereas these definitions fulfill their purpose as part of a broader ontology, we argue that they do little to support the engineering processes of identifying, inventorying, diagnosing, and removing faults in a program. Indeed, the definition of Avizienis et al. relies on adjudging or hypothesizing, two highly subjective criteria, and assumes, through its definition of an error, that we have means to judge the correctness of arbitrary system states (vs. initial or final states). Also, the IEEE definition is vague as to what constitutes a correct or an incorrect step, and who gets to decide what is or is not correct.

In this paper, we consider a semantics-based definition of a software fault, and discuss, through analytical and empirical arguments, how this definition can be deployed to support the engineering processes that we use routinely to deal with software faults. Specifically, the definition we use has the following attributes:

- It is based on an implicit level of granularity of the source code, which is determined according to the level of precision at which we want to localize faults; we use the term feature to refer to a segment of source code at the chosen level of granularity (e.g. statement, expression, variable reference, lexical token, etc). For the sake of generality, we admit that a feature needs not be contiguous, hence may involve two or more lexemes at different locations in the program.
- Our definition of a fault $f$ in a program $P$ with respect to specification $R$ involves nothing other than $f$, $P$ and $R$; and it is totally formal, modulo the definition of a feature. It does not involve any subjective value judgement (adjudging or hypothesizing) nor does it require that we know the expected state of the program at intermediate steps in its execution.

We have found, and we argue in this paper, that a formal definition of program faults enables us to achieve a number of capabilities, with concrete practical implications:

- A formal characterization of fault removal. Given a fault $f$ in a program $P$, we formulate the condition under which a substitute $f'$ of $f$ constitutes a certifiable fault removal.
- A distinction between a single multi-site fault and multiple single-site faults. This distinction is important in practice because it enables us to control combinatorial explosion when we attempt to generate patches: the only time we ever need to combine patches is if we are looking for a multi-site single fault; other than that, we ought to remove faults one at a time, to avoid combinatorial explosion.
- A definition of a unitary increment of correctness enhancement. We introduce the concept of elementary fault removal, which represents an atomic/ minimal program transformation that enhances the correctness of a program, for a given level of granularity.
- Insights into oracle design. The definition of fault removal enables us to design precise test oracles that characterize valid program repairs; we find in this paper that while non-regression is a sufficient condition for correctness enhancement, it is not a necessary condition; so that traditional regression testing is prone to cause a loss of recall.
- The distinction between removing a fault and remedying a failure. There is no one-to-one mapping between faults and failures; the same fault may cause several failures and
the same failure may be traced back to more than one fault. Hence focusing on faults and focusing on failures yield vastly different policies.

- Letting programs dictate the fault removal schedule. Programs do not expose all their faults at once; rather, some faults may have to be removed before others can be discovered; the order in which the faults of a program come to light depends on the test data we use, and on how each discovered fault is fixed. A given observed failure may be due to a combination of faults (some of which may be visible earlier than others), hence cannot be remedied until we have derived the set of patches that remove simultaneously all the relevant faults; this carries a significant risk of combinatorial explosion. By focusing on fault removal rather than failure remedial, we let the program expose its faults in the order it determines, and we remove them in sequence as they appear.

- Separating debugging from testing. Fault removal (aka debugging) is so inconceivable without testing that these two terms are used almost interchangeably. Yet, with a definition of fault removal, we can identify, remove and prove the removal of a fault by static analysis, without recourse to any testing; this shown in (Ghardallou, Diallo, Frias, & Mili, 2016).

- Fault Density vs Fault Depth. Once we define what a fault is, we discover that there is a difference between the statement “Program P has N faults”, and the statement “Program P requires N fault removals”. This difference stems from the interdependence between faults; we discuss in this paper why we find that the latter is a more meaningful measure of faultiness than the former.

The definitions and propositions we discuss in this paper rely on relational mathematics; while we assume the reader familiar with elementary relational mathematics, we briefly introduce some relevant notations in section II. In section III, we discuss the definition and properties of faults, and in section IV we explore some consequences of this definition. In section V we show the results of an experiment that illustrates some of our discussions, and we conclude in section VI by offering some tentative insights and prospects.

II. RELATIONAL MATHEMATICS

We assume the reader is familiar with simple relational mathematics, and we use this section to introduce some definitions and notations, inspired from (Brink, Wolfram, & Schmidt, 1997). We represent specifications and programs with sets and relations. Sets are represented by programming language-like variable declarations; we refer to such a set as the space of the program of interest, and we refer to its elements as the states of the program. A relation on set $S$ is a subset of the Cartesian product $S \times S$. Special relations on $S$ include the universal relation $L = S \times S$, the identity relation $I$, and the empty relation $\emptyset$. Operations on relations include the usual set theoretic operations of union ($R \cup R'$), intersection ($R \cap R'$) and complement ($\overline{R}$), as well as the relational product ($R \cdot R'$, or $RR'$ for short), the converse ($\overline{R}$), the domain ($\text{dom}(R)$) and the pre-restriction (of relation R to set $T$: $\tau_T(R)$).

It is easy to see that the product $RL$ (of relation $R$ by the universal relation $L$) is: $RL = \{(s, s') | s \in \text{dom}(R)\}$; we may, when this does not lead to confusion, use $RL$ and $\text{dom}(R)$ interchangeably. A relation $R$ is said to be deterministic if and only if $RR \subseteq I$, reflexive if and only if $I \subseteq R$, symmetric if and only if $R \subseteq \overline{R}$, and transitive if and only if $RR \subsetneq R$. A relation is said to be a partial ordering if and only if it is reflexive, antisymmetric and transitive.

III. RELATIVE CORRECTNESS AND FAULTS

In order to define faults, we need to discuss relative correctness, which in turns requires that we discuss (absolute) correctness

A. Programs and Specifications

Given a program $P$ on space $S$, we let the function of program $P$ be the set of pairs $(s, s')$ such that if execution of $P$ starts in state $s$ then it terminates in state $s'$. We may, when this does not lead to confusion, use a program and its function interchangeably. Given a space $S$, we let a specification $R$ on space $S$ be a relation on $S$.

B. Refinement and Absolute Correctness

Given two relations $R$ and $R'$, we say that $R'$ refines $R$, or that $R$ is refined by $R'$ (denoted by: $R' \geq R$, or $R \leq R'$) if and only if $RL \cap R'L \cap (R \cup R') = R'$. This relation (between relations) is a partial ordering. Interpretation: $R'$ refines $R$ if and only if $R'$ has a larger domain and assigns fewer images than $R$ to each element of the domain of $R$. See Figure 1.

![Figure 1: $R' \geq R$.](image)

Given a program $P$ and a specification $R$, we say that $P$ is correct with respect to $R$ if and only if $P$ refines $R$. Though it looks different, this is the exact same definition as traditional definitions of (total) correctness, found in (Gries, 1981) (Hehner, 1993) and others. The following Proposition, due to Mills et al (Mills, Basili, Gannon, & Hamlet, 1986), gives a necessary and sufficient condition of correctness: Program $P$ is correct with respect to specification $R$ if and only if $\text{dom}(R \cap P) = \text{dom}(R)$. The set $\text{dom}(R \cap P)$ is the set of states for which execution of $P$ terminates normally and satisfies specification $R$. We refer to this set as the competence domain of $P$ with respect to $R$.

C. Relative Correctness

In order to define faults (as we do in the next section), we must define relative correctness, i.e. the property of a program $P'$ to be more-correct than a program $P$ with respect to a specification $R$. For the sake of simplicity, we limit our
discussions in this paper to deterministic programs, and we present the following definition, due to (Mili, Frias, & Jaoua, 2014): Program \( P' \) is said to be more-correct (respectively: strictly more-correct) than program \( P \) with respect to specification \( R \) if and only if: \((R \cap P')L \subseteq (R \cap P)L\) (respectively: \((R \cap P')L \subseteq (R \cap P)L\)). We abbreviate this property by: \( P' \geq_R P \) (respectively: \( P' \geq_R P \)). Interpretation: Program \( P' \) is (strictly) more-correct than program \( P \) with respect to specification \( R \) if and only if \( P' \) has a (strictly) larger competence domain with respect to \( R \) than \( P \). See Figure 2. To contrast correctness (defined in the previous section) with relative correctness, we may refer to the former as absolute correctness.

Notice that to be more-correct, a program \( P' \) does not need to imitate the correct behavior of program \( P \); it may have a wholly distinct correct behavior. The ovals in the domains of \( P \) and \( P' \) show the competence domains of \( P \) and \( P' \) with respect to \( R \).

How do we know whether our definition is any good? We check it against some properties that we expect a definition of relative correctness to satisfy.

Relative Correctness is reflexive and transitive but not antisymmetric. It is clear why we want relative correctness to be reflexive and transitive. Figure 3 shows why we do not want it to be antisymmetric: we want to allow two programs to be equally correct yet distinct.

Relative Correctness culminates in absolute correctness. Of course, we want a (absolutely) correct program \( P' \) to be more-correct than (or as correct as) any candidate program. According to Mills’ Proposition (previous section) if \( P' \) is (absolutely) correct with respect to \( R \), then \( \text{dom}(R \cap P') = \text{dom}(R) \); on the other hand, \( \text{dom}(R) \) is an upper bound of \( \text{dom}(R \cap P) \) for any candidate program \( P \). QED.

Relative Correctness and Reliability. The reliability of a program \( P \) is defined by means of two parameters: a specification \( R \) and a probability distribution \( \theta \) on the domain of \( R \); for the sake of simplicity we assume that the domain of \( R \) is finite and that \( \theta \) is a discrete probability distribution. We define the reliability of program \( P \) with respect to specification \( R \) and probability distribution \( \theta \) on the domain of \( R \) as the probability that a random state in \( \text{dom}(R) \) selected according to \( \theta \) yields a correct execution of \( P \) with respect to \( R \). Given the definition of competence domain, this is merely the probability that a randomly selected element of \( \text{dom}(R) \) falls in the competence domain of \( P \) with respect to \( R \). We write:

\[
\rho_R^\theta = \sum_{s \in \text{CD}} \theta(s),
\]

where \( \text{CD} = \text{dom}(R \cap P) \) is the competence domain of \( P \) with respect to \( R \).

We infer from this definition that for any probability distribution \( \theta \), if \( P' \) is more-correct than \( P \) with respect to \( R \), then \( P' \) is also more reliable than \( P \) with respect to \( R \) and \( \theta \). More interestingly, we find that if \( P' \) is more reliable than \( P \) with respect to \( R \) for any probability distribution \( \theta \), then \( P' \) is more-correct than \( P \) with respect to \( R \). To prove this, we assume that \( P' \) is not more correct than \( P \), and we find a probability distribution on \( \text{dom}(R) \) for which \( P \) is more-reliable than \( P' \). If \( P' \) is not more-correct than \( P \), then there exists an element \( \pi \) of the competence domain of \( P \) that is not in the competence domain of \( P' \). We let \( \theta(\pi) = 1 \) and \( \theta(s) = 0 \) for all \( s \neq \pi \), and we find that \( \rho_R^\theta(P') = 1 \) and \( \rho_R^\theta(P) = 0 \), whence \( P \) is more reliable than \( P' \). QED.

Whence we write:

\[
(P' \geq_R P) \Leftrightarrow \left( \forall \theta: \rho_R^\theta(P') \geq \rho_R^\theta(P) \right).
\]

This equation links relative correctness with reliability: to be more-correct means to be more-reliable with respect to \( \text{any} \) probability distribution of inputs.

Relative Correctness and Refinement. If program \( P' \) refines program \( P \), it means that whatever \( P \) does, \( P' \) can do better. We would expect this to imply that \( P' \) refines \( P \) if and only if \( P' \) is more-correct than \( P \) with respect to any specification. The proof of necessity is trivial: if \( P' \) refines \( P \), then \( P' \) is a
superset of $P$, hence it has a larger competence domain with respect to any specification, by monotonicity. The proof of sufficiency relies on a lemma to the effect that if two functions $F$ and $F'$ are such that $F \subseteq F'$ and $\text{dom}(F') \subseteq \text{dom}(F)$ then $F = F'$. If $F'$ is more correct than $P$ for all $R$, then it is more correct than $P$ with respect to $P$, which can be written $\text{dom}(P \cap P') \subseteq \text{dom}(P)$. This, in conjunction with the set theoretic identity $P \cap P' \subseteq P'$, from which we infer $P \subseteq P'$.

Whence we write:

$$ (P' \supseteq P) \iff (\forall R : P' \supseteq_R P). $$

Figure 5 shows relative correctness as an intermediate property between being more reliable (when we quantify with respect to $\theta$), and being more-refined (when we quantify with respect to $R$). For the sake of completeness, we also show the quantifications in reverse order ($R$ then $\theta$).

![Diagram](image)

**Figure 5: Reliability, Relative Correctness, Refinement**

### D. Faults and Fault Removal

Now that we have a vetted definition of relative correctness, we are ready to define faults and fault removals.

Given a program $P$, a specification $R$, 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 substitute $f'$ of $f$ such that program $P'$ obtained from $P$ by replacing $f$ by $f'$ is strictly more-correct than $P$. The pair $(f, f')$ is then called a fault removal of $f$ in $P$ with respect to $R$.

Note that this definition highlights an issue with the traditional practice of regression testing, which checks for fault removal by ensuring that correct behavior is preserved; as we can see from our definition of relative correctness (and from Figure 2), the preservation of correct behavior is a sufficient condition, but not a necessary condition of fault removal; the use of an unnecessary conditions causes a loss of recall (missing programs that are actually more-correct). Whereas regression testing checks the following condition:

$$ s \in \text{dom}(R \cap P) \Rightarrow P'(s) = P(s), $$

relative correctness mandates the (weaker) condition:

$$ s \in \text{dom}(R \cap P) \rightarrow s \in \text{dom}(R \cap P'). $$

Another issue that our definition highlights is the use of fitness functions in program repair (LeGoues, Dewey-Voigt, Forrest, & Weimer, 2012). Fitness functions are usually computed as the sum of weights associated to the test data on which the candidate repair runs successfully, where weights are assigned to test data according to their preponderance in some usage pattern; as such, the fitness function is an approximation of the candidate program’s reliability. Yet, we saw in the previous section that for a given probability distribution ($\theta$) enhanced reliability is a necessary but not a sufficient condition of relative correctness; hence the use of fitness functions may cause a loss of precision (retrieving candidate repairs that are not actually more-correct than the original).

### IV. IMPLICATIONS AND APPLICATIONS

#### A. Elementary Faults

We consider a program $P$ and we let $f_1$ and $f_2$ be two features in the source code of $P$ that admit substitutes, say $f_1'$ and $f_2'$, such that the program $P'$ obtained from $P$ by replacing $f_1$ by $f_1'$ and $f_2$ by $f_2'$ is strictly more-correct than $P$ with respect to some specification $R$. According to our definition of a fault, $(f_1, f_2)$ is a fault in $P$ with respect to $R$. The question we wish to ponder is whether we are looking at a single two-site fault $(f_1, f_2)$ or two single-site faults ($f_1$ and $f_2$). The answer to this question depends, of course, on whether $f_1$ alone is a fault and whether $f_2$ alone is a fault. If neither $f_1$ nor $f_2$ is a fault, but $(f_1, f_2)$ is a fault, then we say that $(f_1, f_2)$ is an elementary fault; we also designate as elementary fault any single-site fault.

Why is it important to characterize elementary faults? Because in a fault removal process, it is advantageous to remove faults one elementary fault at a time. Suppose, for the sake of argument, that each feature of the program admits $N$ possible patches, and that we want to rectify (repair) $k$ features in the program; if we want to repair them all at once then we need to analyze $N^k$ repair candidates, an $O(N^k)$ operation; but if we want to repair them one at a time, assuming each is an individual fault, then we need to analyze $k \times N$ repair candidates, an $O(N)$ operation. Of course, not all elementary faults are single-site faults; we may have to consider two-site faults, at a cost of an $O(N^2)$ operation, or three-site faults, at a cost of an $O(N^3)$ operation, or higher multiplicities. But these are probably a very small fraction of faults, in practice.

To illustrate the difference between a single multi-site fault and multiple single-site faults, we consider two sample examples, with a multiplicity of 2. We let $S$ be the space
defined by a real array \( a[0..N] \), for \( N \geq 1 \), an index variable \( k \), and a real variable \( x \), and we consider the following specification \( (R) \) and program \( (P) \), where \( s \) stands for the aggregate \( (a, k, x) \) and \( s' \) for \( (a', k', x') \):

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

\( P: \{ x=0; \ k=0; \text{while} (k!=N) \{ x=x+a[k]; k=k+1 \} \} \).

The function of program \( P \) can be written as:

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

To compute the competence domain of program \( P \) with respect to specification \( R \), we compute the intersection of \( R \) and \( P \), then take its domain:

\[
R \cap P = \{ \text{substitutions} \}
\]

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

\[
= \{ \text{rewriting} \}
\]

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

\[
= \{ \text{subtracting the sum from 1 to N-1 from both sides} \}
\]

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

The domain of this relation, the competence domain of \( P \) with respect to \( R \), is:

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

Indeed, if the specification mandates to compute the sum from 1 to \( N-1 \), then the program may be correct with respect to the specification for those arrays that satisfy the condition \( a[0] = a[N] \). One way to fix this program is to change \( (k=0) \) into \( (k=1) \) and \( (k=N) \) into \( (k=N+1) \). This raises the question: are we dealing with a single two-site fault or two single-site faults? To answer this question, consider programs \( P0 \) and \( P1 \), where we make the first substitution, and \( P1 \), where we make the second substitution.

\( P0: \{ x=0; \ k=1; \text{while} (k!=N) \{ x=x+a[k]; k=k+1 \} \} \);

\( P1: \{ x=0; \ k=0; \text{while} (k!=N+1) \{ x=x+a[k]; k=k+1 \} \} \).

By computing the functions of these programs then their competence domain the same way we did above for \( P \), we find:

\[
CD0 = \{ s | a[N] = 0 \}.
\]

\[
CD1 = \{ s | a[0] = 0 \}.
\]

Because there is no inclusion relation between \( CD \) and \( CD0 \), nor between \( CD \) and \( CD1 \), neither the substitution of \( (k=0) \) into \( (k=1) \) nor the substitution of \( (k=N) \) into \( (k=N+1) \) is a fault removal. But performing both substitutions simultaneously yields

\[
P' : \{ x=0; \ k=1; \text{while} (k!=N+1) \{ x=x+a[k]; k=k+1 \} \}.
\]

whose competence domain is the domain of \( R \) (as the interested reader can easily check); hence \( P' \) is (absolutely) correct with respect to \( R \), hence more correct than \( P \) with respect to \( R \). This makes the aggregate \( (k=0, k=N) \) a single two-site fault. Figure 6 shows the pattern of relative correctness relations (represented by thick black arrows) that characterize such situations; \( \sigma1 \) and \( \sigma2 \) represent, respectively, the substitution of \( (k=0) \) by \( (k=1) \) and the substitution of \( (k=N) \) by \( (k=N+1) \); thin blue lines represent substitutions that do not engender relative correctness relationships.

For an example of two single-site faults, we consider the same space (without variable \( x \)) and we consider the following specification \( R \) and program \( P \) (where \( a[i..j] = 0 \) is a shorthand for saying that \( a[h] = 0 \) for all \( h \) between \( i \) and \( j \)):

\[
R = \{ (s, s') | a[0] = a'[0..N-1] = 0 \land a'[1..N] = a[N] \},
\]

\( P: \{ k=0; \text{while} (k!=N) \{ a[k]=0; k=k+1 \} \} \).

The function of program \( P \) can be written as:

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

Indeed, \( P \) assigns \( N \) to \( k \), puts 0 in \( a[\ldots] \) between indices 0 and \( N-1 \), and preserves (does not modify) \( a[N] \). Taking the intersection of \( R \) and \( P \), we find:

\[
R \cap P = \{ \text{substitutions} \}
\]

\[
\{ (s, s') | a'[0] = a[0] \land a'[N] = a[N] \land k' = N \land a'[0..N] = 0 \}.
\]

Taking the domain of this relation, we find:

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

We let \( P0 \) and \( P1 \) be the programs obtained from \( P \) by changing \( (k=0) \) into \( (k=1) \) and \( (k=N) \) into \( (k=N+1) \),

\( P0: \{ k=1; \text{while} (k!=N) \{ a[k]=0; k=k+1 \} \} \)

\( P1: \{ k=0; \text{while} (k!=N) \{ a[k]=0; k=k+1 \} \} \).
P1: \{ k=0; while \( k!=N+1 \) \{ a[k]=0; k=k+1 \}\}

We find the following functions of \( P0 \) and \( P1 \):

\[
P0 = \{(s, s')| a[0] = a'[0] \land a[N] = a'[N] \land k' = N \land a'[1..N-1] = 0 \},
\]

\[
P1 = \{(s, s')| a'[0..N]=0 \land k' = N + 1 \}.
\]

Taking the intersection with \( R \), we find:

\[
R \cap P0 = \{(s, s')| k' = N \land a[0] = a'[0] \land a[N] = a'[N] \land a'[1..N] = 0 \}
\]

\[
R \cap P1 = \{(s, s')| k' = N + 1 \land a[0] = a'[0] \land a'[0..N] = 0 \}
\]

From which we derive the competence domains of \( P0 \) and \( P1 \):

\[
CD0 = \text{dom}(R \cap P0) = \{ s| a[N] = 0 \},
\]

\[
CD1 = \text{dom}(R \cap P1) = \{ s| a[0] = 0 \}.
\]

By comparing these competence domains against that of \( P \), we find that they are both supersets of \( CD \), hence each individual transformation has improved the correctness of \( P \). If we let \( P' \) be the program obtained from \( P \) by performing both transformations, we find that it is correct with respect to \( R \), hence its competence domain is the domain of \( R \), which is all of \( S \). Figure 7 shows the pattern of relative correctness relations that characterize a situation where we have two single-site faults; we use the same abbreviations \( \sigma 1 \) and \( \sigma 2 \) to represent program substitutions.

![Figure 7: Pattern of Two Single-Site Faults](image)

The contrast between Figure 6 and Figure 7 illustrates the difference between a single two-site fault and two single-site faults. In Figure 6 the two substitutions need to take place before we can observe correctness enhancement, whereas in Figure 7 each individual substitution enhances correctness.

B. Fault Density and Fault Depth

We consider the array sum program presented earlier, along with the specification it is supposed to satisfy:

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

\[
P: \{ x=0; k=0; while \( k!=N \) \{ x=x+a[k]; k=k+1 \}\}.
\]

We let the level of granularity at which we want to analyze faults be the expression; in other words, we restrict our attention to faults that stem from using the wrong expression (in an assignment statement, an array reference, a function call, etc). Given this restriction, we see two faults in this program, i.e. two features that admit substitutions that would make the program strictly more-correct:

- The fault made up of the aggregate \( k=0, k!=N \), which we had discussed above; we refer to this fault as \( f12 \) and we refer to its corresponding substitution (into \( k=1, k!=N+1 \)) as \( \sigma 12 \).
- The fault \( f3 = (a[k]) \), which admits a substitution \( \sigma 3 \) from \( (a[k]) \) to \( (a[k+1]) \); instead substitution \( \sigma 3 \) would also produce a correct program, hence enhance correctness.

Even though we have two faults, \( f12 \) and \( f3 \), we are only one fault removal away from a correct program, because when we apply substitution \( \sigma 12 \) to remove fault \( f12 \), we find that \( f3 \) is no longer a fault; and when we apply substitution \( \sigma 3 \) to remove fault \( f3 \), we find that \( f12 \) is no longer a fault. But if we apply both substitutions, we find a program \( Q \) that has two faults, just like \( P \). See Figure 8, where thick black arrows represent relative correctness relations.

![Figure 8: Density and Depth](image)

We use the term fault density to refer to the number of faults in a program, and we use the term fault depth to refer to the minimal number of elementary fault removals that separate a program from a correct program. Usually, when we remove a fault from a program \( P \) and obtain a program \( P' \), we want to think that fault density has decreased by 1; the foregoing discussion provides a counter-example to this premise, since \( P \) has a density of 2 whereas \( P' \) and \( P'' \), which are derived by removing one fault from \( P \), have a density of 0. By contrast, the fault depth satisfies the following relation between \( P \) (a faulty program) and \( P' \) (obtained from \( P \) by removing one fault):

\[
depth(P') \geq depth(P) - 1.
\]
Equality holds if the transition from \( P \) to \( P' \) is on a minimal path from \( P \) to a correct program.

Whereas fault density measures the number of faults in a program, fault depth measures the (minimal) number of (elementary) fault removals that separate a program from a correct program; we argue that fault depth is a more meaningful measure of program faultiness than fault density. In the absence of a definition of faults, we are prone to confuse/ equate these two numbers; we illustrate in section V.B to what extent they are in fact orthogonal/ independent.

C. An Infrastructure of Test Oracles

We consider a space \( S \), a specification \( R \) on \( S \), and a test data \( T \) that is a subset of \( S \). We use relational formulae given in (Khaireddine, Zakharchenko, & Mili, 2017) to generate oracles that check the following properties of a candidate program \( P' \):
- The absolute correctness of \( P' \) with respect to \( \tau \_ \_ R \), the pre-restriction of \( R \) to \( T \).
- The relative correctness of \( P' \) over some other program \( P \) with respect to \( \tau \_ \_ R \).
- The strict relative correctness of \( P' \) over \( P \) with respect to \( \tau \_ \_ R \).

Of course, in general we are interested in claims of correctness and relative correctness with respect to \( R \), not \( \tau \_ \_ R \). But if our analysis is based on testing program \( P' \) on test data \( T \), then the only certifiable claims we can make involve \( \tau \_ \_ R \), not \( R \). Whether a claim on \( \tau \_ \_ R \) can be extended to \( R \) depends on whether test data set \( T \) is adequate; we are not discussing the adequacy of \( T \), hence we limit our claims to \( \tau \_ \_ R \).

Absolute Correctness. To fix our ideas, we adopt the following framework in which the oracle of absolute correctness is invoked:

\[
\begin{align*}
\text{bool absorcor; absorcor=true; for all (s in T)} \\
\text{(state type inits; inits=s;)} \\
\text{Pprime(); // modifies s, preserves inits} \\
\text{absorcor = absorcor && absororacle(inits,s);)} \\
\text{return absorcor;} \\
\end{align*}
\]

Now we must write code for the Boolean predicate \( \text{absororacle(inits,s)} \). To this effect, we assume that we have two Boolean functions:
- A binary Boolean function \( R(s,s\prime) \), which represents specification \( R \).
- A unary (in \( S \)) Boolean function \( \text{domR}(s) \), which represents the domain of \( R \).

Whence we write:

\[
\begin{align*}
\text{bool absororacle(s, sprime)} \\
\text{return !\text{domR}(s) || R(s,sprime);} \\
\end{align*}
\]

The following proposition justifies the design of this oracle:

Proposition. If for all \( s \) in \( T \) program \( P' \) terminates and satisfies oracle \( \text{absoracle()} \), then \( P' \) is (absolutely) correct with respect to \( \tau \_ \_ R \).

We do not prove this proposition, but we give an intuitive argument to justify it. To prove the correctness of a candidate program with respect to \( R \), we must check the oracle \( (s \in \text{dom}(R) \Rightarrow (s,s') \in R) \), rather than the oracle \( ((s,s') \in R) \), for the following reason: if we execute the program on an element of \( T \) that is outside the domain of \( R \), then regardless of the final state that the program produces, the predicate \( (s,s') \in R \) will return false (by definition, if \( s \) is not in the domain of \( R \), then no \( s' \) will satisfy the condition \( (s,s') \in R \)). Yet in fact the oracle should return true in such cases since candidate programs are not responsible for initial states outside the domain of \( R \).

Relative Correctness. To fix our ideas, we adopt the following framework in which the oracle of relative correctness over some base program \( P \) is invoked:

\[
\begin{align*}
\text{bool relcor; relcor=true;} \\
\text{for all (s in T)} \\
\text{(state type inits; inits=s;)} \\
\text{Pprime(); // modifies s, preserves inits} \\
\text{relcor = relcor && reloracle(inits,s);} \\
\text{return relcor;} \\
\end{align*}
\]

The Boolean function \( \text{reloracle(inits,s)} \) is then defined as follows:

\[
\begin{align*}
\text{bool reloracle (s, sprime)} \\
\text{(state type inits; inits=s;)} \\
\text{P(); // modifies s, preserves inits} \\
\text{return !(\text{absoracle(inits,s)}) || \text{absoracle(inits, sprime);})} \\
\end{align*}
\]

In other words, predicate \( \text{reloracle(s, sprime)} \) runs program \( P \), checks whether program \( P \) passes the test for absolute correctness, and if it does then it equates absolute correctness of \( P' \) with relative correctness over \( P \); if \( P \) fails, then \( P' \) is off the hook, and is considered (vacuously) to pass relative correctness over \( P \).

Strict Relative Correctness. To fix our ideas, we adopt the following framework to test for strict relative correctness over some base program \( P \):

\[
\begin{align*}
\text{bool relcor, strict; relcor=true; strict=false;} \\
\text{for all (s in T)} \\
\text{(state type inits; inits=s;)} \\
\text{Pprime(); // modifies s, preserves inits} \\
\text{relcor = relcor && reloracle(inits,s);} \\
\text{strict = strict || strictpredicate(inits, s);} \\
\text{return relcor && strict;} \\
\end{align*}
\]
The Boolean function reoracle(init,s) is defined the same way as above, whereas the Boolean function strictpredicate(init, s) is defined as follows:

\[
\text{bool } \text{strictpredicate} (s, \text{sprime}) \\
\{ \text{statetype } \text{init}=s; \\
P(); // modifies s, preserves init \\
\text{return } (!\text{absoracle}(\text{init}, s)) \& \& \text{absoracle}(\text{init}, \text{sprime}); \}
\]

This function returns true whenever the base program has failed and the program under test (Pprime, in the oracle code) succeeds. Program Pprime() is deemed to be strictly more correct than P with respect to \(T \setminus R\) if and only if Pprime() never fails whenever P() succeeds, and it succeeds at least once where P() fails.

\[\text{bool } \text{EQ} (s, \text{sprime}) = (x=xprime \& \& y=yprime);\]

\[\text{return } (!\text{absoracle}(\text{init}, s)) \& \& \text{absoracle}(\text{init}, \text{sprime});\]

#### D. Deterministic and Non-Deterministic Specifications

In the previous subsection, we discuss how to derive an oracle for absolute correctness from a specification, then how to derive an oracle for relative correctness from an oracle for absolute correctness, then how to derive an oracle for strict relative correctness from an oracle for relative correctness. The question we consider now is: How do we derive the specification \(R\) to begin with? Of course, ideally the specification \(R\) stems from the product requirements; we have no doubt that it is usually very difficult to capture software requirements in a cohesive, complete form; and probably even more difficult to do so in the form of a simple Boolean function in some programming language. At the same time, we argue that this study does not depend critically on the availability of a complete specification. We can use the results of this paper even if we have a partial specification, for example a specification capturing the most error-prone aspects of a specification, of the most critical aspects of a specification, or the easiest to capture aspects of a specification, etc. The downside of using an incomplete specification is that it exposes fewer faults in the program.

For the sake of our study, however, we usually use a known correct version of the program under investigation as a basis for producing a specification. Given a program \(P\) that we know to be correct, we use it as follows to generate a deterministic relation \(R\).

\[\text{bool } \text{R} (s, \text{sprime}) = (x=xprime \& \& y=yprime);\]

\[\{ \text{statetype } \text{init}=s; \\
P(); // modifies s, preserves init \\
\text{return } (x=xprime \& \& y=yprime); \}
\]

If, for the sake of experimentation, we want to generate a non-deterministic specification, then we replace the equality (==) in the function above by an equivalence relation:

\[
\text{bool } \text{R} (s, \text{sprime}) \\
\{ \text{statetype } \text{init}=s; \\
P(); // modifies s, preserves init \\
\text{return } \text{EQ}(s, \text{sprime}); \}
\]

In this section, we run an experiment in which we take a program of the Siemens Benchmark (Georgia Tech, 2007), make a number of modifications (aka faults, but we reserve the name to situations that meet our definition) to its source code, then use our infrastructure of oracles to locate and remove faults from it one at a time, until we find an absolutely correct program. In addition to illustrating the use of our test oracles, this experiment corroborates many of the premises that we put forth in this paper, as we discuss in section V.C.

#### A. Premise and Set-Up

We consider the program named replace in the Siemens Benchmark, and we enter six modifications that are listed for it in the benchmark; the size of this program is 563 LOC and the size of the test data set that is provided for it is 5542 elements. We let \(P\) be the correct version of replace, and \(P'\) be the version to which we have applied the six proposed modifications; we refer to these as modifications rather than faults because now that we have a definition of faults, we want to use the term only if the conditions of the definition are met.

Our goal in this experiment is not only to remove all the faults of the program, but in fact to show all the fault removal sequences that lead from the original faulty program to a correct program. Because all the claims we make in this experiment are based on testing candidate programs on test data \(T\), these claims (of absolute, correctness, and strict relative correctness) pertain to specification \(T\), \(R\), rather than specification \(R\). To carry out this experiment, we use two devices:

- **Patch Generation.** A means to generate possible patches that are of the same nature and the same scale as the modifications applied to the original program. To this effect, we have used a mutant generator, which we have fine-tuned to the task at hand (Delamaro, Maldonado, & Vincenzi, 2001). Given the size of the program and the mutation parameters that we have selected, the mutant generator produces 90 mutants at each invocation.

- **Patch Validation.** We use our infrastructure of oracles, as we describe below.

The following algorithm aims to build a graph \(G\) that shows all the fault removal sequences from the faulty program \(P\) to a (absolutely) correct program; the nodes of this graph represent mutants of \(P\) and its arcs represent strict relative correctness relations.
The algorithm starts with $G$ initialized to be a single node that contains program $P$; it proceeds by inspecting the maximal elements of $G$ that are not strictly correct (hence lend themselves to more fault removals), generating their mutants, and checking if any of the mutants are strictly more correct than the base; if a mutant $M$ is found to be strictly more-correct than a maximal program $Q$ then the arc $(Q, M)$ is added to the graph, and now $M$ becomes a maximal node of the graph. This process concludes whenever all the maximal elements of the graph are absolutely correct; if we find maximal elements of the graph that are not absolutely correct but admit no mutant that is strictly more-correct, then we deploy double mutation, or perhaps higher order mutations. If despite deploying higher order mutations we still have maximal nodes that are not absolutely correct, we conclude that either the program has faults of a higher order (highly unlikely) or (more plausible) that the patch generation method is inadequate (wrong type, or wrong scale, etc). In any case, we feel that the patch validation method is sound, and that such deadlocks arise only if patch generation is flawed.

B. Empirical Observations

Figure 9 shows the graph that results from applying the above algorithm to the replace sample of the Siemens Benchmark. At the conclusion of the four first iterations, the algorithm produces m79.3.42.47 as the only maximal element of the graph; this node is not absolutely correct, and none of its simple mutants turned out to be strictly more-correct. When we deploy double mutation, however, we find two double-mutants that are strictly more-correct than it, and they are both absolutely correct. One of them (m79.3.42.47.37.85) is actually the original replace program; the other maximal mutant is different from the original, but is absolutely correct with respect to $\tau R$ all the same.

For the sake of argument, we assume that the mutant generation method used in this experiment is complete, in the sense that if a program $P$ has a fault $f$ and $(f, f')$ is a fault removal for $P$ then the generator will produce a mutant of $P$ that has $f'$ in lieu of $f$ in $P$; considering the modifications we have entered in replace, and the way we have parameterized the mutant generator, this appears to be a legitimate assumption. Under this assumption, the density of each program in this graph is the outgoing degree of the corresponding node. Also, the depth of program is the minimal distance from the node to the top of the graph. We find, from Figure 9, that even though we have applied six modifications to produce $P$, $P$ has only one fault, since the node of $P$ has a single outgoing arc. One may ask: how can $P$ have only one fault if we seeded six? The answer is that the other faults may be hidden by the first one, and can only be seen once the first one is removed. So in fact the fault density of $P$ is one but its fault depth is five, which again shows that fault depth is a more meaningful measure than fault density. In this graph, fault depth decreases by one with each fault removal but fault density does not: For example, $\text{density}(P) = 1$, but $\text{density}(m79) = 3$, rather than 0; as another example, $\text{density}(m79.42.47) = 1$, but $\text{density}(m79.3.42.47) = 2$. Note that even though program $P$ was seeded with six modifications, its fault depth is five, not six, because the last fault removals were done through double mutation; still, we do not claim a numeric relation between the number of modifications and fault depth, as some modifications could be immaterial (i.e. they do not affect the function of the program); we also suspect that the faults of the Siemens benchmark were chosen in such a way as to alter the function of the program.

![Figure 9: Fault Removal Graph of replace](image)

C. Analysis and Lessons Learned

When we say that program $P$ has six faults, we implicitly assume that these faults are fixed features of the source code of $P$, that they are all visible in $P$, that we can fix any one of the six we choose, that there is a unique way to fix each fault, and that the result would be a program with five faults. If this were the case, then the graph we would find by applying the algorithm of section V.B to the replace program would be the graph shown in Figure 10; note that in this ideal graph the density of each program (node) equals its depth. The vast difference between Figure 9 and Figure 10 reflects the extent to which this vision of faults is unrealistic. For a given test data set $T$, program $P$ does not expose all its faults at once, and whenever it exposes more than one fault at a time, the choice of which fault to fix first and how to fix it does matter.

Another observation we can infer from this example is that when we try to repair a program, it is necessary to focus on removing faults rather than remedying failures. When we select a specific failure of the program, defined by an initial
state that leads to an execution that violates the specification, we have no way to tell whether the fault that causes it low or high in the fault removal graph (Figure 9). If the fault is high, say \( k \) arcs away from program \( P \), and each mutant generator produces \( N \) mutants, then we need to search a space of size \( N^k \) to find an adequate repair. By contrast, if we remove elementary faults one at a time, in the order in which the program exposes them, the search space is never larger than \( N^m \), where \( m \) is the multiplicity of highest order multi-site fault (\( m=2 \) in our example) rather than the fault depth of the program (which is usually unknown and unbounded).

Another important lesson offered by this example is the distinction between fault density and fault depth. If programs behaved as shown in Figure 10, then density and depth would be identical: if we have six faults in a program, it takes six fault removals before we can turn it into a correct program. But in reality, density and depth are unrelated: in Figure 9, \( P \) has a fault density of 1 and a fault depth of five; also, as we climb the graph, fault depth decrease by one at each step, but fault depth evolves in an unpredictable manner.

Finally, we point out how a purely semantic test, the test of strict relative correctness, was, with the help of adequate mutant generation, able to detect and remove all the faults of the program, one at a time. Note that if \( P' \) is absolutely correct with respect to \( \tau \), this does not mean that \( P' \) is absolutely correct with respect to \( R \) of course, though we can prove under some conditions that \( P' \) is then more-correct than \( P \) with respect to \( R \) (not merely to \( \tau \)), which is usually what we want to achieve in a program repair operation; we do not show the proof, due to lack of space.

VI. CONCLUSION

In this paper we revisit a definition of software faults (given in earlier work), and discuss its impact on routine software engineering processes such as software testing, software quality analysis, and program repair. Our definition of a fault assumes an implicit level of granularity (at which we want to isolate faults) and involves only the faulty feature, the program in which this feature appears, and the specification against which correctness of the program is judged. Our definition of a software fault is based on a formal definition of relative correctness for deterministic programs, which we have validated by analyzing its intrinsic properties and its relation to (traditional) absolute correctness, reliability, and refinement.

Several other authors have introduced and discussed some approximations of absolute correctness that may be construed as definitions of relative correctness (Zhao, Littlewood, Povyakalo, Strigini, & Wright, 2016) (Littlewood & Rushby, 2012) (von Essen & Jobstman, 2013) (Lahiri S. , McMillan, Sharma, & Hawblitzel, 2013) (Logozzo, Lahiri, Faehndrich, & Blackshear, 2014) (Logozzo & Ball, Modular and Verifiable Program Repair, 2014). Our approach can be characterized by the following distinguishing premises: we model programs as simple input/output mappings (rather than finite state automata); we model specifications as relations (rather than frames of assertions); we model relative correctness as a semantic property between two programs and a specification (rather than an empirical operational property about program executions); we make provisions for the fact that correct program behavior is not unique (by virtue of the fact that specifications are usually vastly non-deterministic).

As far as prospects are concerned, we consider that while so far we have used relative correctness for the sole purpose of validating program repairs, it is possible to envision using them for the purpose of generating program repairs. In the same way that (Gries, 1981) (Hehner, 1993) (Morgan, 1998) (Dijkstra, 1976) and others have used mathematics of absolute correctness to produce methods for deriving programs that are correct by design, we envision the possibility of using the mathematics of relative correctness to produce methods for deriving programs that are, by design, more-correct than a base program. This is clearly a long-term goal, but one that appears to be worthwhile, for the impact it would have on the practice of software engineering.

Another venue of research we are considering is to explore the implications of generalizing the concept of relative correctness to non-deterministic programs, due to (Desharnais, Diallo, Ghardallou, Frias, Jaoua, & Mili, 2015), as this concept may be the key to scaling up. By modeling programs with non-deterministic relations, we can analyze them for relative correctness without capturing their functionality in all its detail; this is currently under investigation.
VII. BIBLIOGRAPHY


