Innovations Syst Softw Eng (2017) 13:219–239
DOI 10.1007/s11334-017-0300-7

What is a fault? and why does it matter?

Nafi Diallo[1] · Wided Ghardallou[2] · Jules Desharnais[3] · Marcelo Frias[4] · Ali Jaoua[5] · Ali Mili[1]

Received: 10 June 2016 / Accepted: 26 July 2017 / Published online: 22 August 2017
© Springer-Verlag London Ltd. 2017

Abstract Faults are an important concept in the study of system dependability, and most approaches to dependability can be characterized by the way in which they deal with faults (e.g., fault avoidance, fault removal, fault tolerance, fault forecasting). In their seminal work on modeling dependable computing, Laprie et al. define a fault as the adjudged or hypothesized cause of an error. In this paper, we propose a more formal definition of a fault in the context of software products and discuss some of its implications.

Keywords Correctness · Partial correctness · Total correctness · Relative correctness · Absolute correctness · Software fault · Fault removal · Fault density · Fault depth · Software testing · Software repair · Software design

1 Motivation and background

1.1 The trouble with faults

This research stems from asking ourselves the question *what is a software fault?* and from realizing that the answer to this question is more than a mere academic exercise.

We argue that a formal definition of faults is indispensable, given that faults play a crucial role in the study of software dependability, that they are the basis of the classification of methods of dependability (fault avoidance, fault removal, fault tolerance), and that they are at the center of several software engineering processes and metrics, such as fault density, fault proneness, fault forecasting, program repair, mutation testing, multiple mutation. In [3,22–24] Laprie et al. define a fault as the *adjudged or hypothesized cause of an error* [3]; we argue that, as far as software is concerned, this definition is not sufficiently precise, first because adjudging and hypothesizing are highly subjective human endeavors, and second because the concept of error is itself insufficiently defined, since it depends on a detailed characterization of correct system states at each stage of a computation (which is usually unavailable). According to the IEEE Standard *IEEE Std 7-4.3.2-2003* [1], a software fault is *An incorrect step, process or data definition in a computer program*. We argue that this definition is equally inadequate, in the sense that it merely replaces an undefined concept (fault) by another (correctness/ incorrectness), fails to capture many of the properties of software faults (as we discuss below) and fails to acknowledge the role of specifications in the definition of a fault (the same feature of a program may be a fault or not depending on the specification being considered). In [11], Gartner distinguishes between two types of state transitions in a computing system: transitions that result from normal system operation and transitions that result from fault occurrences, and models a fault as an unwanted, though possible, state transition of a process. Like Laprie's definition, this definition relies on the availability of a precise characterization of incorrect (unwanted) process states at each step of process execution.

In fairness, we acknowledge that defining software faults is fraught with difficulties:

- *Discretionary determination* Usually we determine that a program part is faulty because we think we know what

✉ Ali Mili
mili@njit.edu

1 New Jersey Institute of Technology, Newark, NJ, USA
2 University of Tunis El Manar, Tunis, Tunisia
3 Laval University, Quebec City, QC, Canada
4 ITBA, Buenos Aires, Argentina
5 Qatar University, Doha, Qatar

the designer intended to achieve in that particular part, and we find that the program does not fulfill the designer's intent; clearly, this determination is only as good as our assumption about the designer's intent.
- *Contingent determination* The same faulty behavior of a software product may be repaired in more than one way, possibly involving more than one part; hence, the determination that one part is a fault is typically contingent upon the assumption that other parts are not in question.
- *Tentative determination* Usually, we determine that a program part is faulty because we believe that if we could change it in some specific way, the program would be better, but in the absence of a clear definition of what it means for the program to be better, this determination is tentative.

In order to overcome the difficulties raised above, we resolve to proceed as follows: We introduce a concept of *relative correctness*, i.e., the property of a program to be more correct than another program with respect to a specification [8,31]. Then we define a fault in a program as any program part (be it a simple statement, a lexical token, an expression, a compound statement, a block of statements, a set of non-contiguous statements, etc.) for which there exists a substitution that would make the program more-correct than the original with respect to a relevant specification [31]. With such a definition, we address the difficulties raised above, namely:

- *A Fault as an Intrinsic Attribute* The definition of a fault is not dependent on any design assumptions, but involves only the (incorrect) program, the faulty program part and the specification with respect to which correctness is defined.
- *A Fault as a Definite Property* If we let a fault be any program part that admits a substitution that makes the program more-correct, then the designation of a fault is no longer contingent on any hypothesis; we need not make any assumption on whether other parts of the program are faulty or not.
- *Fault Removal as a Verifiable Process* By testing a program for relative correctness rather than (traditional) absolute correctness, we can determine with greater confidence that a particular fault has been removed, regardless of whether that makes the program correct or not (due to the presence of other residual faults).

In order to reap these benefits, we must first introduce a definition of relative correctness; this is the subject of Sect. 3. In preparation for this objective, we present some mathematical definitions and notations and some elements of relations-based program semantics in Sect. 2. In Sect. 4, we consider in turn several properties that we would want a concept of relative correctness to satisfy, and prove that our definition does satisfy all of them. In Sect. 5, we discuss the uses of the concept of relative correctness and its implications for relevant software processes. Once we know how to use relative correctness, the next issue we address is: How do we establish relative correctness, i.e., how to build the case that a program is more-correct than another with respect to a specification; this is the subject of Sect. 6. Section 7 summarizes and assesses our findings, discusses related work and sketches directions of future research.

2 Mathematics for program analysis

We assume the reader familiar with discrete mathematics, most notably relational algebra; this section introduces definitions and notations, but it is our assumption that the reader is familiar with these concepts [5].

2.1 Relational notations

Dealing with programs, we represent sets using a programming-like notation, by introducing variable names and associated data types. For example, if we represent set S by the variable declarations $\{\ x : X;\ y : Y;\ z : Z,\}$ then S is the Cartesian product $X \times Y \times Z$. Elements of S are denoted in lower case s and are triplets of elements of X, Y and Z. Given an element s of S, we represent its X-component by $x(s)$, its Y-component by $y(s)$ and its Z-component by $z(s)$. When no risk of ambiguity exists, we may write x to represent $x(s)$ and x' to represent $x(s')$.

A (binary) relation on 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 $I = \{(s, s')|s' = s\}$ and the *empty* relation $\phi = \{\}$. Operations on relations (say, R and R') include the set theoretic operations of *union* $(R \cup R')$, *intersection* $(R \cap R')$, *difference* $(R \backslash R')$ and *complement* $(\overline{R})$. They also include the *relational product*, denoted by $(R \circ R')$, (or RR', for short) and defined by:

$$RR' = \left\{(s, s')|\exists s'' : (s, s'') \in R \wedge (s'', s') \in R'\right\}.$$

The *converse* of relation R is the relation denoted by $\widehat{R}$ and defined by $\widehat{R} = \{(s, s')|(s', s) \in R\}$. The *domain* of a relation R is defined as the set $dom(R) = \{s|\exists s' : (s, s') \in R\}$. A relation R is said to be *reflexive* if and only if $I \subseteq R$, *antisymmetric* if and only if $(R \cap \widehat{R}) \subseteq I$, *asymmetric* if and only if $(R \cap \widehat{R}) = \phi$ and *transitive* if and only if $RR \subseteq R$. A relation is said to be a *partial ordering* if and only if it is reflexive, antisymmetric and transitive. Also, a relation R is said to be *total* if and only if $I \subseteq R\widehat{R}$, and *deterministic* (or, a *function*) if and only if $\widehat{R}R \subseteq I$. A relation R is said to be

a *vector* if and only if $RL = R$; we use vectors to represent subsets of S in relational form.

2.2 Relational semantics

Given a program p on space S, we denote by $[p]$ the function that p defines on its space, i.e.,

$$[p] = \{(s, s')| \text{ if program } p \text{ executes on state } s \text{ then it terminates in state } s'\}.$$

We represent programs by means of a few simple C-like programming constructs, which we present below along with their semantic definitions:

- *Abort:* $[abort] \equiv \phi$.
- *Skip:* $[skip] \equiv I$.
- *Assignment:* $[s = E(s)] \equiv \{(s, s')|s \in \delta(E) \wedge s' = E(s)\}$, where $\delta(E)$ is the set of states for which expression E can be evaluated.
- *Sequence:* $[p_1; p_2] \equiv [p_1] \circ [p_2]$.
- *Conditional:* $[\texttt{if}\ (t)\ \{p\}] \equiv T \cap [p] \cup \overline{T} \cap I$, where T is the vector defined as: $T = \{(s, s')|t(s)\}$.
- *Alternation:* $[\texttt{if}\ (t)\ \{p\}\ else\ \{q\}] \equiv T \cap [p] \cup \overline{T} \cap [q]$, where T is defined as above.
- *Iteration:* $[\texttt{while}\ (t)\ \{b\}] \equiv (T \cap [b])^* \cap \widehat{\overline{T}}$, where T is defined as above.
- *Block:* $[\{x : X; p\}] \equiv \{(s, s')|\exists x, x' \in X : (\langle s, x\rangle, \langle s', x'\rangle) \in [p]\}$.

Rather than using the notation $[p]$ to denote the function of program p, we will usually use upper case P as a shorthand for $[p]$. Also we may, when it causes no confusion, refer to a program and its function by the same name.

2.3 Refinement ordering

The concept of refinement is at the heart of any programming calculus; the following definition captures our interpretation of refinement.

Definition 1 We let R and R' be two relations on space S. We say that R' *refines* R if and only if $RL \cap R'L \cap (R \cup R') = R$.

We write this relation as: $R' \sqsupseteq R$ or $R \sqsubseteq R'$. Intuitively, R' refines R if and only if R' has a larger domain than R and has fewer images than R inside the domain of R. It is easy to prove that $\sqsubseteq$ is a partial ordering; also, it is easy to prove that for functions R and R', $R \sqsubseteq R'$ if and only if $R \subseteq R'$.

2.4 Refinement lattice

Since refinement is a partial ordering between specifications, it is legitimate to ponder its lattice-like properties. The following proposition, due to [4], provides a useful result with regard to the lattice of specifications.

Proposition 1 *Any two specifications R and R' that satisfy the following condition $(R \cap R')L = RL \cap R'L$ (called the* consistency condition*) admit a least upper bound, denoted by $R \sqcup R'$ (read: R join R') and defined by: $R \sqcup R' = (\overline{R'L} \cap R) \cup (\overline{RL} \cap R') \cup (R \cap R')$.*

Interpretation: The consistency condition between two specifications is the condition under which the specifications admit a joint refinement; the join of two specifications R and R' captures all the requirements of R and all the requirements of R' and nothing else; it is possible to combine R and R' only if they do not contradict each other (whence the consistency condition).

3 Absolute correctness and relative correctness

Whereas absolute correctness characterizes a program with respect to a specification, relative correctness ranks two programs with respect to a specification; for a given specification, it defines a partial ordering on candidate programs.

3.1 Absolute correctness

Definition 2 Let p be a program on space S and let R be a specification on S. We say that program p is *correct* with respect to R if and only if P (the function of program p on space S) refines R. We say that program p is *partially correct* with respect to specification R if and only if P refines $R \cap PL$.

This definition is consistent with traditional definitions of partial and total correctness [10,16–18,29]. Whenever we want to contrast correctness with partial correctness, we may refer to it as *total correctness*.

Proposition 2 *A deterministic program p is correct with respect to specification R if and only if $(P \cap R)L = RL$.*

This formula is used by Mills et. al. [33] as a definition of correctness; in [30] we show that it is equivalent to our definition.

3.2 Relative correctness

3.2.1 Deterministic programs

Definition 3 Let R be a specification on space S and let p and p' be two programs on space S whose functions are, respectively, P and P'. We say that program p' is *more-correct* than program p with respect to specification R (denoted by: $p' \sqsupseteq_R p$) if and only if: $(R \cap P')L \supseteq (R \cap P)L$. Also, we

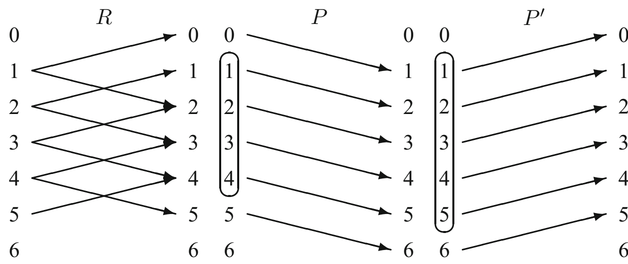

Fig. 1 Enhancing correctness without imitating behavior

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

Whenever we want to contrast correctness (given in Definition 2) with relative correctness, we may refer to it as *absolute correctness*. Note that when we say *more-correct* we really mean *more-correct or as-correct-as*; we use the shorthand, however, for convenience. We give a simple intuitive interpretation of this definition: The relation (actually a vector) $(R \cap P)L$ represents the set of initial states for which program p behaves according to the requirements of specification R; we refer to this set as the *competence domain* of program p with respect to specification R, so that to be more-correct merely means to have a larger competence domain. See Fig. 1; in this figure, the competence domains of P and P' are, respectively, $CD = \{1, 2, 3, 4\}$ and $CD' = \{1, 2, 3, 4, 5\}$. Hence p' is (strictly) more-correct than p with respect to R.

To illustrate this definition, we consider the space S defined by two integer variables x and y, and we let R be the following specification on S:

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

We consider the following candidate programs, denoted p_0 through p_7. Next to each program p_i, we represent its function P_i and then its competence domain (CD_i). Figure 2 shows how these candidate programs are ranked by relative correctness with respect to R; this graph merely reflects the inclusion relationships between the competence domains.

- p_0: `{x=1; y=-1;}`. We find: $P_0 = \{(s, s')|x' = 1 \wedge y' = -1\}$. Whence, $(P_0 \cap R)L = \{(s, s')|\exists s' : x' = 1 \wedge y' = -1 \wedge x^2 \leq -1 \leq 2x^2\} = \{\}$.
- p_1: `{x=2*x; y=0;}`. We find: $P_1 = \{(s, s')|x' = 2x \wedge y' = 0\}$. Whence, $(P_1 \cap R)L = \{(s, s')|\exists s' : x' = 2x \wedge y' = 0 \wedge x^2 \leq 0 \leq 2x^2\} = \{(s, s')|x = 0\}$.
- p_2: `{x=x*x; y=0;}`. We find: $P_2 = \{(s, s')|x' = x^2 \wedge y' = 0\}$. Whence, $(P_2 \cap R)L = \{(s, s')|\exists s' : x' = x^2 \wedge y' = 0 \wedge x^2 \leq 0 \leq 2x^2\} = \{(s, s')|x = 0\}$.
- p_3: `{x=2*x; y=1;}`. We find: $P_3 = \{(s, s')|x' = 2x \wedge y' = 1\}$. Whence, $(P_3 \cap R)L = \{(s, s')|\exists s' : x' = 2x \wedge y' = 1 \wedge x^2 \leq 2x \leq 2x^2\} = \{(s, s')|0 \leq x \leq 2\}$.
- p_4: `{x=2*x; y=2;}`. We find: $P_4 = \{(s, s')|x' = 2x \wedge y' = 2\}$. Whence, $(P_4 \cap R)L = \{(s, s')|\exists s' : x' = 2x \wedge y' = 2 \wedge x^2 \leq 4x \leq 2x^2\} = \{(s, s')|x = 0 \vee 2 \leq x \leq 4\}$.
- p_5: `{x=2*x; y=x/2;}`. We find: $P_5 = \{(s, s')|x' = 2x \wedge y' = x\}$. Whence, $(P_5 \cap R)L = \{(s, s')|\exists s' : x' = 2x \wedge y' = x \wedge x^2 \leq 2x^2 \leq 2x^2\} = L$.
- p_6: `{y=x/2; x=2*x;}`. We find: $P_6 = \{(s, s')|x' = 2x \wedge y' = x/2\}$. Whence, $(P_6 \cap R)L = \{(s, s')|\exists s' : x' = 2x \wedge y' = x/2 \wedge x^2 \leq x^2 \leq 2x^2\} = L$.
- p_7: `{x=x*x; y=2;}`. We find: $P_7 = \{(s, s')|x' = x^2 \wedge y' = 2\}$. Whence, $(P_7 \cap R)L = \{(s, s')|\exists s' : x' = x^2 \wedge y' = 2 \wedge x^2 \leq 2x^2 \leq 2x^2\} = L$.

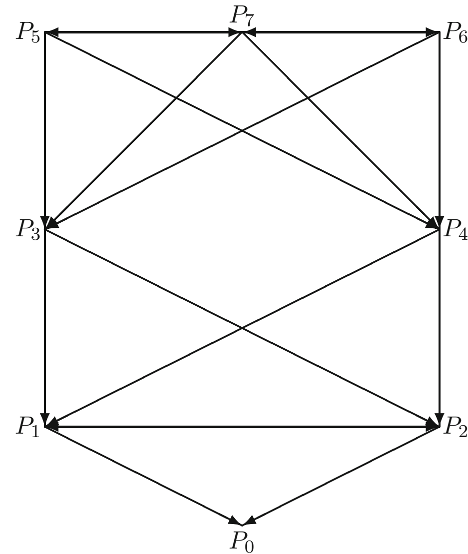

Fig. 2 Relative correctness relations

This example illustrates a number of properties:

- Note that this relation is not antisymmetric, so that two programs may be mutually related and still be distinct (such is the case for P_1 and P_2, for example).
- The top of the graph represents the programs that are (absolutely) correct with respect to specification R: P_5, P_6 and P_7.
- A program may be more-correct than another without imitating its correct behavior. For example, p_3 is more-

correct than p_1 and yet it does not behave as p_1 on the competence domain of p_1.

3.2.2 Non-deterministic programs

So far, we have discussed relative correctness between deterministic programs, i.e., programs whose behavior/outcome is uniquely determined for each initial state. Yet it is useful to define relative correctness for non-deterministic programs, for two distinct reasons:

- First, because we want to discuss relative correctness for programs that are written in non-deterministic languages or programs whose behavior is non-deterministic as a result of, e.g., randomly generated data.
- Second, because we want to reason about the relative correctness of deterministic programs without having to compute their function in all its minute detail; their representation is then a non-deterministic relation rather than a function.

The following definition, due to [7], defines relative correctness for potentially non-deterministic programs; it is equivalent to Definition 3 when P and P' are deterministic, and it satisfies for non-deterministic programs many of the properties we discuss in Sect. 4 for deterministic programs.

Definition 4 We let R be a specification on set S, and we let p and p' be (possibly non-deterministic) programs on space S. We say that p' is more-correct than p with respect to R (abbrev: $p' \sqsupseteq_R p$) if and only if:

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

Interpretation: p' is more-correct than p with respect to R if and only if it has a larger (or equal) competence domain, and for the elements in the competence domain of p, program p' has fewer (or the same) images that violate R than p does. In other words, a program p' is more-correct than a program p with respect to R if and only if p' obeys R more often than p and violates R less egregiously (in fewer ways) than p. For the sake of simplicity, we focus on deterministic programs in the remainder of this paper, and we refer interested readers to [7].

3.3 Faults and fault removals

Any definition of a fault must be based on a level of granularity at which we want to isolate faults. Typically, faults are implicitly isolated at the level of the *line of code* (LOC); other common levels of granularity include the programming language statement, the expression, the operator/ operand, the variable reference, the lexeme. We let *feature* designate any program part at the appropriate level of granularity; we further assume that a feature does not need to be contiguous and can be made up of two or more program parts at different locations in the source code.

Definition 5 Let p be a program on space S and R be a specification on S; let f be a feature in p. We say that f is a *fault* in p with respect to specification R if and only if there exists a substitution f' of f such that the program p' obtained from p by substituting f by f' is strictly more-correct than p with respect to R.

Of course, we assume that replacing feature f by feature f' in P does not break the syntactic integrity of the program; i.e., the new program p' passes the compilation (in order for P' to be defined).

Definition 6 Let p be a program on space S and R be a specification on S, let f be a fault in p, and let f' be a substitute for f. We say that the pair (f, f') is a *(monotonic) fault removal* if and only if the program p' obtained from p by substituting f by f' is strictly more-correct than p.

For illustration, we consider the following program, say p, taken from [13] (with some modifications):

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

We let S be the space defined by the declarations of line 3, to which we add variable os which represents the output stream (in C++ parlance), and we let R be the following specification:

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

where we let $\alpha_A = 'A' \ldots 'Z', \alpha_a = 'a' \ldots 'z', \nu = '0' \ldots '9'$, and $\sigma = \{\text{set of ASCII symbols}\}$. Also, we let $\oplus$ denote the concatenation, we let $list\langle T \rangle$ denote the set of lists of type T, and we let $\#_A$, $\#_a$, $\#_\nu$ and $\#_\sigma$ be the functions that to each list l assign (respectively) the number of upper case alphabetic characters, lower case alphabetic characters, numeric digits and symbols; also, we let $\#_\alpha$ be defined as

$\#_{\alpha}(l) = \#_a(l) + \#_A(l)$. We introduce the following programs, which are derived from p by some modifications of its source code:

- p_{01} The program obtained from p when we replace (`let+=2`) by (`let+=1`).
- p_{10} The program obtained from p when we replace (`'Z'>c`) by (`'Z'>=c`).
- p_{11} The program obtained from p when we replace (`let+=2`) by (`let+=1`) and (`'Z'>c`) by (`'Z'>=c`).

We find the following competence domains for these programs:

- $CD = \{s|q \in list\langle\alpha_a \cup \nu \cup \sigma\rangle\}$.
- $CD_{01} = \{s|q \in list\langle(\alpha_A \backslash \{'Z'\}) \cup \alpha_a \cup \nu \cup \sigma\rangle\}$.
- $CD_{10} = \{s|q \in list\langle\alpha_a \cup \nu \cup \sigma\rangle\}$.
- $CD_{11} = \{s|q \in list\langle\alpha_A \cup \alpha_a \cup \nu \cup \sigma\rangle\}$.

By comparing the competence domains, we draw the following conclusions:

- The feature (`let+=2`) is a fault in p, and its substitution by (`let+=1`) is a fault removal, yielding the more-correct program p_{01}.
- The feature (`'Z'>c`) is a fault in p_{01}, and its substitution by (`'Z'>=c`) is a fault removal, yielding the more-correct program p_{11}.
- The feature defined by the two statements (`let+=2`) and (`'Z'>c`) is a fault in p, and its substitution by (`let+=1`) and (`'Z'>=c`) is a fault removal, yielding the more-correct program p_{11}.
- The program p_{11} is correct with respect to R.

Note that the statement (`'Z'>c`) is a fault in p_{01} but we conjecture that it is not a fault in p (to assertively claim that it is not a fault in p, we must prove that no substitute of (`'Z'>c`) could made p strictly more-correct); also note that the statement (`'Z'>c`), *in combination with* the statement (`let+=2`) is a fault in p, but we conjecture that it is not a fault in p by itself (to assertively claim that it is not a fault in p, we must prove that no substitute of (`'Z'>c`) could made p strictly more-correct).

4 Validation of relative correctness

4.1 Litmus tests

How do we know that our definition of relative correctness is sound? To answer this question, we list some properties that a definition of relative correctness ought to meet; then we check that our definition does satisfy them.

- *Reflexivity and Transitivity, and non-Antisymmetry*. Of course, we want relative correctness to be reflexive and transitive; we do not want it to be antisymmetric, since we want to have programs that are mutually more-correct, yet distinct (not only syntactically distinct, but computing different functions as well).
- *Absolute Correctness as the Culmination of Relative Correctness*. Relative correctness ought to be defined in such a way that if a program keeps getting more and more-correct with respect to a specification, it will eventually be (absolutely) correct. This property can also be formulated as follows: A program that is absolutely correct with respect to a specification is more-correct than (or as-correct-as) any candidate program with respect to the same specification.
- *Relative Correctness as a Sufficient Condition, but not a Necessary Condition, of Higher Reliability*. If program p' is more-correct than program p, then of course we want p' to be more reliable than p, but we do not want *more-correct* to be equivalent to *more reliable*, as the former is a logical/functional property, whereas the latter is a stochastic property.
- *Refinement is equivalent to Relative Correctness with respect to any (all) Specification(s)*. When program p' refines program p, we interpret this to mean that whatever p can do, p' can do as well or better; in particular, it means that p' is more-correct than (or as-correct-as) p with respect to *any* specification R.

4.2 Reviewing the criteria

4.2.1 Reflexivity, transitivity and non-antisymmetry

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

4.2.2 Absolute correctness as the culmination of relative correctness

Proposition 3 *Let R be a specification on space S, and let p' be a program on S. Then p' is correct with respect to R if and only if p' is more-correct with respect to R than any candidate program p on S.*

Proof Proof of necessity: Let p' be correct with respect to R; then, according to Proposition 2, $RL = (R \cap P')L$. Let p be an arbitrary program on space S; by set theory, we have $RL \supseteq (P \cap R)L$. Hence p' is more-correct with respect to R than p.

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

We write this as:

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

4.2.3 Relative correctness and reliability

The reliability of a program p on space S can be defined with respect to two parameters: a specification R on S and a probability distribution θ on the domain of R. For the sake of simplicity, we assume that the domain of R is a finite set and that θ is a discrete probability distribution.

Definition 7 The reliability of a program p on space S with respect to specification R on S and probability distribution θ on $dom(R)$ is the probability that the execution of p on a random element of $dom(R)$ selected according to distribution θ satisfies specification R. We denote it by $\rho_\theta^R(p)$.

We have the following proposition.

Proposition 4 *Let p and p' be two programs on space S, and let R be a specification on S. Then p' is more-correct than p with respect to R if and only if for any probability distribution θ on $dom(R)$, p' is more reliable than p with respect to R and θ.*

Proof The proof of necessity is trivial: According to Definition 7, the reliability of program p with respect to specification R and probability distribution θ can be written as:

$$\rho_\theta^R(p) = \sum_{s \in dom(R \cap P)} \theta(p).$$

Clearly, the bigger the competence domain, the greater the reliability.

Proof of sufficiency: Let us assume that p' is not more-correct than p; then there exists an element, say s_0 that belongs to the competence domain of p and does not belong to the competence domain of p'. If we let $\theta_0()$ be defined by:

$$\theta_0(s_0) = 1,$$
$$\forall s \neq s_0 : \theta_0(s) = 0,$$

then we find $\rho_\theta^R(p) = 1$ and $\rho_\theta^R(p') = 0$, which contradicts the hypothesis that p' is more reliable than p for any probability distribution. □

We write:

$$\boxed{p' \sqsupseteq_R p \Leftrightarrow (\forall \theta : \rho_\theta^R(p') \geq \rho_\theta^R(p))}.$$

For a given probability distribution θ relative correctness logically implies (but is not equivalent to) enhanced reliability, but when we quantify enhanced reliability for all probability distributions, we obtain equivalence: In other words, to be more-correct means to be more reliable for any probability distribution.

4.2.4 Relative correctness and refinement

The following proposition casts relative correctness as a form of pointwise refinement.

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

Proof Proof of necessity: We have seen in Sect. 2.3 that if P and P' are two functions, then P' refines P if and only if $P' \supseteq P$. The condition $(P' \cap R)L \supseteq (P \cap R)L$ stems readily, by set theory.

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

We write this as:

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

Hence, relative correctness can be seen as an intermediate property between enhanced reliability and refinement. Reliability depends on two parameters: the specification R and the probability distribution θ on $dom(R)$; when we quantify for θ we obtain relative correctness and when we further quantify for R we obtain refinement.

5 Implications and applications

5.1 Measuring faultiness

5.1.1 The difference between software faults and bad apples

A naive interpretation of fault density in a program views the faults in a program as if they were black balls in a bucket full of otherwise white balls: They are all visible; they are intrinsically identifiable (black vs. white); their number is

well defined (and can be estimated); they are independent of each other (removal of one ball does not change the color of the others); they can be removed in an arbitrary order; there is only one way to remove each ball; and whenever one is removed, their number is reduced by one. In this section, we see to what extent this analogy is unfounded: Unlike black balls in a bucket of white balls, faults are highly inter-related; removal of one fault may affect the nature, number and location of other faults; faults are not all visible at once, some may hide others; there may be more than one way to remove a fault, and how a fault is removed affects the subsequent fault configuration of the program; a fault may need to be corrected at more than one location; and the order in which faults are removed matters, as does the way faults are removed.

5.1.2 Elementary faults

We consider a program p on space S and a specification R on S, and we let f_1 and f_2 be two features in p for which we have found substitutes, say f_1' and f_2' that would produce a program p' which is strictly more-correct than p with respect to R. The question we want to consider in this section is: Are we looking at two single-site faults in p or a single fault that spans two sites? The answer to this question depends, of course, on whether f_1 alone is a fault and whether f_2 alone is a fault in p with respect to R. We consider the following space S, specification R and program p:

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

We compute the function of this program and then its competence domain with respect to R, and we find:

$$dom(R \cap P) = \{s|a[0] = a[N]\}\,.$$

Since $dom(R \cap P)$ is not equal to $dom(R)$, which is S, this program is not correct. One way to correct this program is to change `{i=0}` to `{i=1}` and to change `{i<=N-1}` to `{i<=N}`. The question that we raise here is: Do we have two elementary faults here (`{i=0}` and `{i<=N-1}`) or just one elementary fault that spans two sites? To answer this question we consider separately the proposed substitutions and check whether they produce more-correct programs. We let p_{01} be the program obtained from p by replacing `{i=0}` by `{i=1}`, we let p_{10} be the program obtained from p by replacing `{i<=N-1}` by `{i<=N}`, and we let p_{11} be the program obtained from p by performing the two substitutions simultaneously; we find the following competence domains for these programs.

$$dom(R \cap P_{01}) = \{s|a[N] = 0\}.$$
$$dom(R \cap P_{10}) = \{s|a[0] = 0\}.$$
$$dom(R \cap P_{11}) = S.$$

Since the competence domain of p is not a subset of the competence domains of p_{01} and p_{10}, neither p_{01} nor p_{10} is more-correct than p. We conjecture: Neither `{i=0}` nor `{i<=N-1}` is a fault in p, but the composite feature (`{i=0}`, `{i<=N-1}`) is a fault in program p with respect to R, since program p_{11} is more-correct than p with respect to R. We say that this is a *multi-site fault*; in this example we do not have two single-site faults but a single multi-site elementary fault.

This inspires the following definition.

Definition 8 Let f be a fault in program p on space S with respect to specification R. We say that f is an *elementary fault* in p if and only if no part of f is a fault in p with respect to R.

So that if we are going to count faults, we need to count elementary faults rather than arbitrarily large/composite faults. Implicit in this definition is the premise that all single-site faults are elementary faults; a multi-site fault is an elementary fault if and only if no subset of the components that form it is a fault. For further illustration, we consider program p given in Sect. 3.3, and we remember that we have found it to have two faults: First the statement `{let+=2}`; second the combination of two statements `{let+=2, 'Z'>c}` (remember, `{'Z'>c}` is a fault in p_{01} but is not a fault in p). In this example, `{let+=2, 'Z'>c}` is not an elementary fault, because `{let+=2}` by itself is a fault.

5.1.3 Fault density and fault depth

We use the term *fault density* to refer to the number of elementary faults in a program. The trouble with counting faults in a program is that the number of faults in a program violates a simple rule that counting any other commodity satisfies: If we have ten black balls in a bucket of otherwise white balls, and we remove one black ball, we are left with nine black balls. But if we have ten faults in a program and we remove one fault, the number of remaining faults is undetermined: It depends on which of the ten faults we have removed, and on how we have removed it. So that whereas we want to think of fault density as a measure of program faultiness/imperfection or as a measure of repair effort, we can argue that it measures neither imperfection nor repair effort; we consider an alternative.

Definition 9 We let R be a specification on space S and p be a program on space S. The *fault depth* of program p with respect to specification R is the minimal number of elementary fault removals that are required to transform p into a correct program.

If program faults were like black balls, then density and depth would be identical: If we have ten black balls, it takes ten removals to get rid of them, but faults are different. We consider below a case where several faults can be removed by a single fault removal, and a case where a single fault can be remedied multiple times (each producing a strictly more-correct program) before the program is correct.

We consider the array sum program discussed in the previous section (Sect. 5.1.2). We had identified a single elementary multi-site fault, which is the aggregate `f1:({i=0}, {i<=N-1})`; we argue that there is another possible fault in this program, namely `f2:{x=x+a[i]}`. Indeed, this statement admits a substitution, namely `f2':{x=x+a[i+1]}`, that would make the program more correct (as the reader can easily see). If we replace fault `f1` with the feature `f1':({i=1}, {i<=N})`, then we find a correct program, say p_1'; hence while `f2` is a fault in p, it is not a fault in p_1. On the other hand, if we replace fault `f2` with the feature `f2'`, then we find a correct program, say p_2'; hence while `f1` is a fault in p, it is not a fault in p_2'. But if we substitute both `f1` and `f2` by, respectively, `f1'` and `f2'`, we would end up with an incorrect program (say p'') that has two faults `f1'` and `f2'`. Hence program p has a fault depth of 1 and a fault density of 2. Note, incidentally, that having two faults means that we have two distinct opportunities to enhance the correctness of the program. More generally, for a given fault depth, the higher the fault density the better; hence, not only is fault density not representative of program imperfection, it can actually be seen as representing a quality attribute of the program. See Fig. 3, where s_1 represents the substitution (`f1`, `f1'`) and s_2 represents the substitution (`f2`, `f2'`).

The following example shows a case where a fault removal makes the program strictly more-correct without changing its

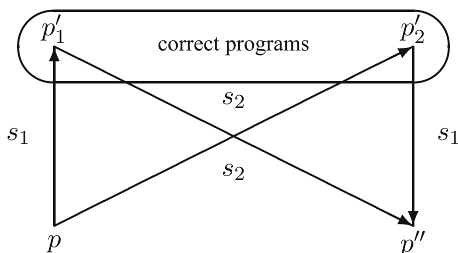

Fig. 3 Fault density = 2, fault depth = 1

fault density; it is an artificial example, but is illustrative nevertheless. We consider the following space S, specification R and program p (where $N \geq 2$):

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

Clearly, the domain of R is S. We compute the competence domain of p, and we find:

$$dom(R \cap P) = \{s|a[0] = 0 \wedge a[1] = 0\}.$$

Hence p is not correct with respect to R. We can check easily that `{i=2}` is a fault in p with respect to R, and we show that the substitution of `{i=2}` by `{i=1}` produces a more-correct program:

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

whose competence domain is:

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

Even though p' is more-correct than p, it is still not correct, since its competence domain is not equal to $dom(R)$. We find that `{i=1}` is a fault in p', and that substituting it by `{i=0}` yields a program, say p'', which is correct with respect to R. Even though it is grossly artificial, this example shows that the same fault may require more than one removal to be completely eliminated from a program.

Hence we adopt fault depth as a measure of program faultiness. Unlike fault density, fault depth does decrease by one whenever we remove a fault that is in the minimal path. Given a faulty program p and a program p' obtained from p by monotonic fault removal, the following formula holds:

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

If the transition from p to p' is part of a minimal sequence of fault removals toward a correct program, then we have equality rather than inequality, i.e., $depth(p) = 1 + depth(p')$. Note that neither equality nor inequality holds between $density(p)$ and $density(p')$, as we showed above.

In summary, the contrast between fault density and fault depth reflects the difference between the two statements: Program p has N elementary faults (density); program p needs N elementary fault removals (depth). If faults were like black balls, then these two statements would be equivalent, but they are not. We argue that depth is a more meaningful measure of faultiness than density.

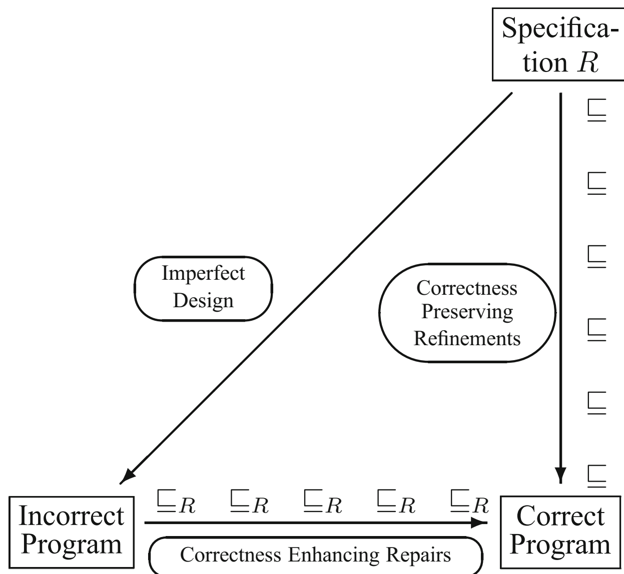

Fig. 4 A framework for monotonic fault removal

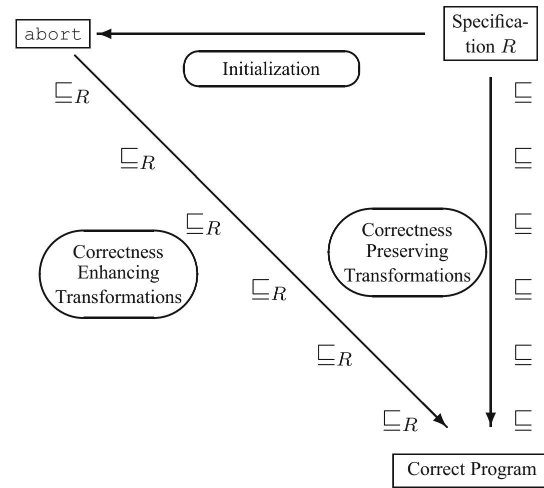

Fig. 5 Program derivation by correctness enhancement

5.2 Monotonic fault removal

As programmers, we may experience the frustration of trying to remove faults from a program, only to find that we are running in circles, patching the program at one end only to break it down at another; as teachers, we often see our students go through the same frustration. This would not happen if we restricted program transformations to provably monotonic fault removals; with such a discipline, we are assured that with each transformation, the program becomes more-correct. Of course, ensuring that a program transformation qualifies as a monotonic fault removal is generally a non-trivial exercise; we postpone the discussion of how to do this to Sect. 6. Here we simply argue that in the same way that stepwise refinement provides a logical framework for software design, which proceeds monotonically from a specification to a program through *correctness-preserving* transformations, relative correctness provides a logical framework for stepwise fault removal, which starts from an incorrect program and proceeds monotonically toward a correct program through *correctness-enhancing* transformations. This process is illustrated in Fig. 4. The concept of relative correctness ought to play for software fault removal the same role that refinement plays for software design: first, as a logical framework for reasoning about faults and fault removal; second, as an ideal process to be followed scrupulously when the stakes warrant it; and third, as a yardstick against which large-scale methods and tools can be evaluated.

5.3 Software design

In Sect. 4.2.4, we have found that program p' refines program p if and only if p' is more-correct than p with respect to *any* specification. This sheds new light on program derivation by successive refinements, which requires that at each stage of this process, we transform a program into a more-refined program. According to our discussion of Sect. 4.2.4, this process requires that at each stage, we transform a program, say p, into a program p' that is more-correct than p with respect to *any* specification. But this raises the question: Why should p' be more-correct than p with respect to *any* specification when we are only interested in specification R? Is it possible that the requirement of refinement is too strong? To explore this venue, we revisit the process depicted in Fig. 4 and imagine that instead of designing a (possibly incorrect) program then proceeding with correctness-enhancing transformations toward a correct program, we start with the (trivially incorrect) `abort` program and transform it into increasingly more-correct (rather than more-refined) programs until we find a correct program. This process is depicted in Fig. 5, which can be viewed as a variation on the process depicted in Fig. 4, where we merely short circuit the *imperfect design* step, and replace it by the trivial initialization to `{abort}`.

As an illustration of this process, we briefly present an example borrowed from [9] (to which the interested reader is referred for further details). We let S be the space defined by natural variables x, y and n, and we let R be the following specification (known as Fermat's factorization):

$$R = \{(s, s')|((n \textbf{ mod } 2 = 1) \vee (n \textbf{ mod } 4 = 0)) \\ \wedge n = x'^2 - y'^2 \wedge 0 \leq y' \leq x'\}.$$

To find a program that is correct with respect to this specification, we consider increasingly complex configurations of x and y and derive the corresponding Fermat factorization; this yields the following sequence of programs, which are ranked by relative correctness with respect to R (though not

by refinement) and culminate in a program that is absolutely correct with respect to R.

```
p0: abort.
p1: {int  r;  x=0;  y=0;  r=0;
    while  (r<n)  {r=r+2*x+1;  x=x+1;}}
p2: {int  r;  x=0;  r=0;
    while (r < n) {r = r + 2 * x + 1; x = x + 1;}
    if  (r>n)  {y=0; while  (r>n)  {r=r-2
    *y - 1; y = y + 1;}}}
p3: {int  r;  x=0;  r=0;
    while  (r<n)  {r=r+2*x+1;  x=x+1;}
    while  (r>n)  {int rsave=r;  y=0;
    while  (r>n)  {r=r-2*y-1;  y=y+1;}
    if  (r<n)  {r=rsave+2*x+1;  x=x+1;}}}
```

Imagine a scenario where our goal is not necessarily to produce a correct program, but rather to produce a sufficiently reliable program, for a pre-specified reliability threshold. Now, consider that, according to Proposition 4, relative correctness logically implies higher reliability. Hence the programs that we generate in this sequence are more and more reliable; if we can estimate the reliability of each program that we generate in this sequence, then we can imagine a scenario where this stepwise transformation concludes, not when we obtain a correct program, but rather when we obtain a program whose reliability equals or exceeds the pre-specified reliability threshold. While we have not yet proven the viability of this approach, it certainly sounds like a worthwhile venue to pursue; as an exercise, we have found that under the hypothesis of uniform probability distribution of the inputs, the reliability of the sequence of programs given above (p_0, p_1, p_2, p_3) is, respectively, (0.0, 0.0133, 0.1328, 1.0).

5.4 A software testing life cycle

The traditional life cycle of software testing is triggered by an observation of failure and proceeds by analyzing the failure, tracing it back to a hypothetical fault, removing the fault, then testing the program for correctness. We argue that it is wrong to test the program for correctness at the end of this process, unless we have reason to believe that the fault we have just removed is the last fault of the program. Given that in general we have no way to check such an assumption, there is no reason we should expect the program to be correct, even if we assume that the fault was properly removed. Instead, the most we can hope for is that the new program is more-correct than the original, and we should be testing it for relative correctness rather than absolute correctness. This raises the question: How do we test for relative correctness? and how is this different from testing for absolute correctness? We argue that testing a program for relative correctness rather than absolute correctness affects two separate aspects of testing, namely test data generation and oracle design.

- *Test Data Generation* The essence of test data generation is to approximate an infinite or very large input space by a small representative test data set; clearly, what input space we are trying to approximate influences what test data we select, regardless of the selection criterion that we apply. When we test a program for absolute correctness with respect to a specification R, the relevant input space is $dom(R)$. By contrast, when we test a program for relative correctness over program p with respect to specification R, the relevant input space is the competence domain of P with respect to R, i.e., $dom(R \cap P)$.
- *Oracle Design* Let $\omega(s, s')$ be the oracle that we use to test a program for absolute correctness with respect to specification R. To test a program p' for relative correctness over program p, we need to check that oracle $\omega(s, s')$ holds only for those inputs s on which program p runs successfully. Hence the oracle of relative correctness, $\Omega(s, s')$, should be written as follows:

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

  This formula shows how to derive the oracle of relative correctness (Ω) from the oracle of absolute correctness (ω); in [32] we discuss how to derive the oracle of absolute correctness ($\omega(s, s')$) from specification R.

5.5 Software repair

Program repair has been an active research area for over a decade, offering increasingly sophisticated tools and methods and producing higher and higher levels of accuracy and scale [2,6,14,15,36,38]. We argue that relative correctness ought to play for program repair the role that absolute correctness plays for program derivation; also, we find the practice of program repair may benefit from insights offered by relative correctness. In particular,

- Current practice of program repair fails to acknowledge the concept of elementary fault removal; as a result, it is prone to cause combinatorial explosion because it makes no distinction between addressing a multi-site elementary fault and multiple single-site faults. Indeed, there is no reason to attempt to remove multiple faults simultaneously; with a proper oracle of relative correctness, it is more efficient to remove faults one at a time. If each generation of patches produces N candidates, then removing k faults in the program takes $k \times O(N)$ operations, which is an $O(N)$ process, if we remove them one at a time. But if we attempt to remove them all at once, this pro-

cess takes $O(N^k)$ operations, a prohibitively expensive proposition for high and unbounded fault depth (k).
- In current practice, the patch validation phase proceeds by applying regression testing to candidate patches; in [19] we argue that regression tests are a sufficient but an unnecessary condition of relative correctness. This means that regression tests are prone to cause a loss of recall of the patch validation phase.
- Another alternative that is used to select candidate patches in the patch validation phase is the use of fitness functions; we find in [19] that fitness functions are an approximation of reliability. And we find in Sect. 4.2.3 that reliability is a necessary but insufficient condition of relative correctness, so that using fitness functions is prone to cause a loss of precision in the patch validation phase.

For all these reasons, we argue that the practice of program repair may benefit from considering relative correctness as part of its theoretic basis.

5.5.1 Illustration

To illustrate the distinction between program repair by absolute correctness and by relative correctness, we consider the Fermat decomposition program that we derived in Sect. 5.3, in which we introduce three changes. We rewrite this program (which we call p') as:

```
void fermatFactorization() {
   int n, x, y; // input/output variables
   int r; // work variable
   x = 0; r = 0;
   while (r < n) {r = r + 2 * x + 1; x = x + 1; }
   while (r > n) {int rsave; y = 0; rsave = r;
      while (r > n) {r = r - 2 * y - 1; y = y + 1; }
      if (r < n) {r = rsave + 2 * x + 1; x = x + 1; }}}
```

We introduce three changes to this program, as shown below; we do not call them faults yet because we do not know whether they meet our definition of a fault (Definition 5). A given number of changes (re: three in this case) can lead to fewer faults (if some changes cancel each other, or if one or more changes have no effect on the function of the program); also, a given number of changes (three in this case) can also lead to a larger number of faults (the same change can be remedied either by reversing the change or by altering the program elsewhere to cancel the change). We revisit this discussion in the next section. We let p be the program obtained after introducing the changes to p':

```
void basep(int& n, int& x, int& y) {
   int r; x = 0; r = 0;
   while (r < n) {r = r + 2 * x - 1; /*change in r*/ x =x+1;}
   while (r > n) {int rsave; rsave = r; y = 0;
      while (r > n) {r =r-2*y+1; /*change in r*/ y =y+1;}
      if (r < n) {r =rsave+2*x-1; /*change in r*/ x =x+1;}}}
```

Most program repair methods proceed by generating patches of the base program and testing them for absolute correctness; all we are advocating in this paper is that instead of testing patches for absolute correctness, we ought to test them for relative correctness. To illustrate our approach, we generate mutants of program p, test them for absolute correctness and show that none of them are (absolutely) correct. If absolute correctness were our only criterion, then this would be the (unsuccessful) end of the experiment. But we find that while none of the mutants are absolutely correct, some are strictly more-correct than p; hence, the transition from p to these mutants represents a fault removal (by Definition 5). If we take these mutants as our base programs and apply the mutation generator to them, then test them for strict relative correctness, we can iteratively remove the faults of the program in a stepwise manner, climbing the relative correctness ordering until we reach a (absolutely) correct program.

Specifically, we start from program p and apply muJava to generate mutants using the single mutation option with the AORB operator (Arithmetic Operator Replacement, Binary). Whenever a set of mutants are generated, we subject them to three tests:

- A test for absolute correctness, using the oracle $\omega(s, s')$.
- A test for relative correctness, using the oracle $\Omega(s, s')$.
- A test for strict relative correctness, which in addition to relative correctness checks the presence of at least one state in the competence domain of the mutant that is not in the competence domain of the base program.

The mutants that are found to be strictly more-correct than the base program are used as new base programs, and the process is iterated again until at least one mutant is found to be absolutely correct; we select this mutant as the repaired version of the original program p. The main iteration of the test driver is given below. All the details of our experiment are posted online at https://selab.njit.edu/programrepair/.

```
int main ()
 {for (int mutant =1; mutant<= nbmutants; mutant++)
  {// test mutant vs spec. R for abs and rel correctness
   bool cumulabs=true; bool cumulrel=true; bool cumulstrict=false;
   while (moretestdata)
    {int n,x,y; int initn,initx,inity; //initial, final states
     bool abscor, relcor, strict;
     initn=td[tdi]; tdi++;  // getting test data
     n=initn; x=initx; y=inity;   // saving initial state
     callmutant(mutant, n, x, y);
     abscor = absoracle(initn, initx, inity, n, x, y);
     cumulabs = cumulabs && abscor;
     n=initn; x=initx; y=inity;  //  re-initializing
     basep(n, x, y);
     relcor = ! absoracle(initn, initx, inity, n, x, y) || abscor;
     strict = ! absoracle(initn, initx, inity, n, x, y) && abscor;
     cumulrel = cumulrel && relcor;
     cumulstrict = cumulstrict || strict;
 }}}
bool R (int initn, int initx, int inity, int n, int x, int y)
   {return ((initn%2==1) || (initn%4==0)) && (initn==x*x-y*y);}
bool domR (int initn, int initx, int inity)
   {return ((initn%2==1) || (initn%4==0));}
bool absoracle (int initn,int initx,int inity,int n,int x,int y)
   {return  (! (domR(initn, initx, inity))
             || R(initn, initx, inity, n, x, y));}
```

The main program includes two nested loops; the outer loop iterates over mutants, and the inner loop iterates over test data. For each mutant and test datum, we execute the mutant and the base program on the test datum and test the mutant for absolute correctness (`abscor`), relative correctness (`relcor`) and strict relative correctness (`strict`); these Boolean results are cumulated for each mutant in variables `cumulabs`, `cumulrel` and `cumulstrict` and are used to diagnose the mutant. As for the Boolean functions `R`, `domR` and `absoracle`, they stem readily from the definition of R and from the oracle definitions given in Sect. 5.4.

5.5.2 Experimental results

Starting with program p, we apply muJava repeatedly to generate mutants, taking mutants which are found to be strictly more-correct as base programs and repeating until we generate a correct program. This proceeds as follows:

- When muJava is executed on program p, it produces 48 mutants, of which two ($m12$ and $m44$) are found to be strictly more-correct than p, and none are found to be absolutely correct with respect to R; we pursue the analysis of $m12$ and $m44$.

- *Analysis of* $m44$. When we apply muJava to $m44$, we find 48 mutants, none of them prove to be absolutely correct, nor relatively correct, nor strictly relatively correct.
- *Analysis of* $m12$. We find by inspection that $m12$ reverses one of the modifications we had applied to p' to find p; since $m12$ is strictly more-correct than p with respect to R, we conclude that the feature in question was in fact a fault in p with respect to R. When we apply muJava to $m12$, it generates 48 mutants, three of which prove to be strictly more-correct than $m12$: We name them $m12.19$, $m12.20$ and $m12.28$. All the other mutants are found to be neither absolutely correct with respect to R, nor more correct than $m12$.
  - *Analysis of* $m12.19$. When we apply muJava to $m12.19$, it generates 48 mutants, none of which is found to be absolutely correct nor strictly more-correct than $m12.19$, but one ($m12.19.24$) proves to be identical to $m12.20$ and is more-correct than (but not strictly more-correct than, hence as-correct-as) $m12.19$.
  - *Analysis of* $m12.20$. When we apply muJava to $m12.20$, it generates 48 mutants, none of which is found to be absolutely correct nor strictly more-correct than $m12.20$, but one ($m12.20.24$) proves to be identical to $m12.19$ and is more-correct than (but not strictly more-correct than, hence as-correct-as) $m12.20$.
  - *Analysis of* $m12.28$. We find by inspection that $m12.28$ reverses a second modification we had applied to p' to obtain p; since $m12.28$ is strictly more-correct than $m12$, this feature is a fault in $m12$; whether it is a fault in p we have not checked, as we have not compared $m12.28$ and p for relative correctness. When we apply muJava to $m12.28$, we find a single mutant, namely $m12.28.44$ that is absolutely correct with respect to R, more-correct than $m12.28$ with respect to R and strictly more-correct than $m12.28$ with respect to R.
    - *Analysis of* $m12.28.44$. We find by inspection that $m12.28.44$ is nothing but the original Fermat decomposition program we have started out with: p'.

The results of this analysis are represented in Fig. 6. Note that $m12$ and $m44$ are strictly more-correct than p with respect to R; hence (according to Definition 5) the mutations that produced these programs from p constitute fault removals; whence we can say that p has at least two faults, which we write as $faultDensity(p) \geq 2$. On the other hand, this experiment shows that we can generate a correct program (p') from p by means of three fault removals; if we let the *Fault Depth* of a program be the minimal number of fault removals that separate it from a correct program, then we can write: $faultDepth(p) \leq 3$.

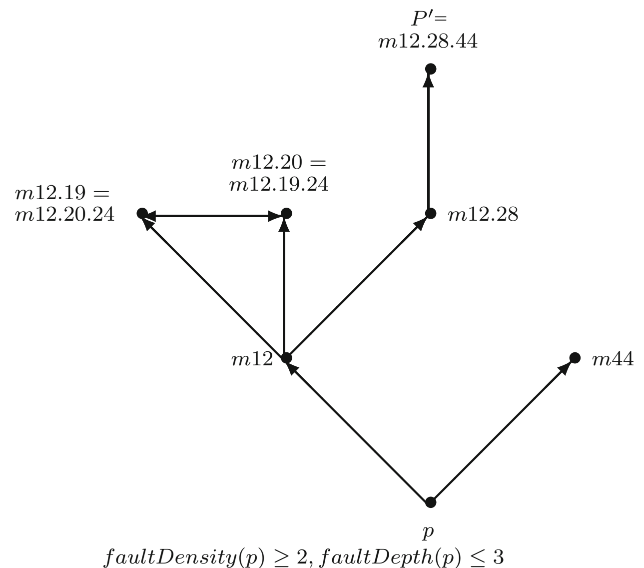

Fig. 6 Relative correctness-based repair: stepwise fault removal

5.6 Multiple mutation

Debroy and Wong [36] use a single muJava mutation in order to generate fix candidates. A clear limitation of such an approach is that many faults will not be fixed; this happens in the case of multi-site faults (that span through more than one program location), as well as whenever the program under analysis has multiple faults. The natural alternative is to apply multiple mutations. This is the case in tools such as those presented in [14] and [38]. The impact of relative correctness on multiple mutation testing depends on the reason for deploying multiple mutations; we see two possible scenarios, which we will discuss in turn.

- *Multiple mutations are deployed to repair multiple faults*. When one uses a test of absolute correctness to assess the validity of program repairs, one has to remove all faults at once in order for the test to be meaningful. Multiple mutation proceeds by applying mutation operators at different places in the program and then testing the resulting program for absolute correctness. We argue that with relative correctness, it is no longer necessary to consider several faults at once, since we can characterize fault removals one fault at a time. Managing faults one at a time offers many advantages: First and foremost, it spares us the massive combinatorial explosion that stems from applying several simultaneous mutations through the program; second, it spares us the trouble of dealing with many fault

removals at once, when we do not know how each fault removal affects others.

- *Multiple mutations are deployed to repair multi-site elementary faults*. In this case, it is sensible to deploy multiple mutations, but note that the multiplicity of the mutation is not the estimated number of faults we are trying to repair simultaneously but rather the multiplicity of the multi-site elementary faults we are trying to repair individually, usually a much smaller number.

6 Proving relative correctness

Given a specification R and two candidate programs p and p', how can we prove that p' is more-correct than p with respect to specification R? Relying on the definition of relative correctness is impractical because it requires that we compute the functions P and P', which is usually a very difficult proposition. Hence we rely on inductive approaches: Sect. 6.1 proceeds by induction on the structure of the program, whereas Sect. 6.2 proceeds by induction on the structure of the specification.

6.1 Induction on the program structure

In this section, we discuss some preliminary results that enable us to prove the relative correctness of a program over another, not by computing their respective functions, but rather by reasoning about their structure. We consider a while loop w on space S, of the form `{while (t) {b}}`, and we denote by B the function of the loop body b and by T the vector that represents the loop condition $T = \{(s, s')|t(s)\}$. An *invariant relation* of loop w is a reflexive transitive superset of $(T \cap B)$; the interested reader is referred to [34] for more details on invariant relations. The following proposition (due to [28]) shows how we can use invariant relations to prove the correctness or the incorrectness of a loop with respect to a specification.

Proposition 6 *Let R be a specification on space S, and let w be a while statement on S of the form w:* `{while (t) {b}}`*, which terminates normally for any state in S, and let V be an invariant relation of w.*

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

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

Intuitive interpretation The sufficient condition of correctness means in effect that the invariant relation V captures enough information about the loop to subsume the specification R; the necessary condition of correctness means that no loop that admits an invariant relation that violates this condition can possibly be correct with respect to R. If we encounter an invariant relation V that does not satisfy the necessary condition of correctness, we conclude that the loop w is not correct with respect to specification R. For the sake of argument, we introduce the following definition.

Definition 10 Let R be a specification on space S, let w be a while statement on S of the form w: `{while (t) {b}}`, which terminates normally for any state in S, and let V be an invariant relation of w. We say that V is *incompatible* with specification R if and only if V fails to satisfy the necessary condition of correctness $(R \cap V)\overline{T} = RL$.

When a relation does satisfy the necessary condition of correctness, we say about it that it is *compatible* with the specification, even though *not incompatible* is a better characterization of such a relation.

Given a while loop w and a specification R, we generate all the invariant relations of w and we divide them into two classes: compatible relations and incompatible relations. If at least one relation (say Q) is incompatible with specification R, then we conclude that the loop is incorrect, and we prepare to repair it; the following proposition provides the basis for doing so.

Proposition 7 *Let R be a specification on space S and let w be a while loop on S of the form, w:* `{while (t) {b}}` *which terminates for all s in S. Let Q be an invariant relation of w that is incompatible with R, and let C be the largest invariant relation of w such that $W = (C \cap Q) \cap \widehat{\widehat{T}}$. Let w' be a while loop that has C as an invariant relation, terminates for all s in S and admits an invariant relation Q' that is compatible with R and satisfies the condition $W' = (C \cap Q') \cap \widehat{\widehat{T}}$. Then w' is strictly more-correct than w.*

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

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

$(R \cap Q)\overline{T} \neq RL$

$\Rightarrow$ {by set theory$(R \cap Q)\overline{T} \subseteq (R \cap Q)L \subseteq RL$}

$(R \cap Q)\overline{T} \subset RL$

$\Rightarrow$ {By hypothesis, $Q'$is compatible}

$(R \cap Q)\overline{T} \subset (R \cap Q')\overline{T}$

$\Rightarrow$ {Taking the intersection with C on both sides}

$(R \cap Q \cap C)\overline{T} \subset (R \cap Q' \cap C)\overline{T}$

$\Rightarrow$ {For any vector v and relation R, $Rv = (R \cap \widehat{v})L$}

$(R \cap Q \cap C \cap \widehat{\overline{T}})L \subset (R \cap Q' \cap C \cap \widehat{\overline{T}})L$

$\Rightarrow$ {associativity}

$(R \cap (Q \cap C \cap \widehat{\overline{T}}))L \subset (R \cap (Q' \cap C \cap \widehat{\overline{T}}))L$

$\Rightarrow$ {substitution}

$(R \cap W)L \subset (R \cap W')L.$

A fault removal action proceeds through four steps (viz., observation of failure, fault localization, fault removal, validation); we discuss below how we perform each step, using Propositions 6 and 7.

- *Observation of Failure* If one of the invariant relations (say, Q) is incompatible with R, then the loop is incorrect; hence there is a fault.
- *Fault Localization* We focus on the variables that are referenced by relation Q.
- *Fault Removal* We must change the statements that affect the identified variables without altering the compatible invariant relations (C). Let x_1, x_2, x_3,...,x_n be the variables of the program, and let us assume that only x_1 and x_2 are involved in the definition of Q. In order to know how to modify x_1 and x_2 in the loop, we write the following condition on x_1, x_2, x_1', x_2':

$$\exists x_3, x_4, \ldots, x_n, x_3', x_4', \ldots, x_n' : \begin{pmatrix} x_1 & x_1' \\ \langle \, x_2 \, \rangle, & \langle \, x_2' \, \rangle \\ \ldots & \ldots \\ x_n & x_n' \end{pmatrix} \in C,$$

  where C is the intersection of all the compatible invariant relations.
- *Validation* Once we change variables x_1 and x_2, we recompute the new invariant relation Q' involving these variables; if Q' is compatible with R, then the new loop is strictly more-correct than the original loop, and a fault has been removed.

This process enables us to remove a fault and prove that the fault has been removed, all by static analysis rather then execution; we refer to this process as *Debugging without Testing* [12].

6.1.1 Illustration

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

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

We consider the following specification, which we are judging the loop against:

$$R = \Big\{(s, s')|b < a < 1 \wedge x' = x \wedge w' = w - z \times \frac{1-(1+a)^{t'-t}}{a} \wedge k' = k + 1000 \times (t' - t) \wedge t \leq t' \wedge 0 < l \leq l' \wedge z > 0 \wedge l \times (1+b)^{-t} = l' \times (1+b)^{-t'}\Big\}.$$

Analysis of this loop by an invariant relations generator [28] derives fourteen invariant relations, of which five are found to be incompatible with the specification. We select the following incompatible invariant relation for remediation:

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

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

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

We focus our attention on variable z and consider the possible mutations of the statement `{z=(1+a)+z}` that preserve

the equation $z' \geq z$; for each mutant of this statement, we recompute the new invariant relation that substitutes for Q and check whether it is compatible with R. We find that the statement `{z=(1+a)*z}` produces a compatible invariant relation, and conclude, by virtue of Proposition 7, that the following loop is more-correct with respect to R than the original loop.

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

We have removed a fault from w and shown that the new program wm is strictly more-correct than the original program w. In order to illustrate the difference between absolute correctness and relative correctness, we ran this program on randomly generated test data using the oracle of absolute correctness derived from R; the program fails at the third test execution. But its failure does not mean that our fault removal was wrong; rather, it means that while wm is more-correct than w, it is not yet absolutely correct. When we run this loop on randomly generated test data using an oracle that tests for relative correctness rather than absolute correctness (see Sect. 5.4), it runs for over eight hundred thousand test data without failure.

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

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

6.2 Induction on the specification structure

In the previous section we have discussed how to prove relative correctness of p' over p with respect to some specification R without having to compute p and p' (as they may be too complex). In this section we turn our attention to the other potential source of complexity, which is specification R.

Proposition 8 *Let p and p' be two programs on space S, and let R and Q be two specifications on S. If p' is more-correct than p with respect to R and with respect to Q, then it is more-correct than p with respect to $(R \sqcup Q)$.*

Proof We introduce a lemma that will be useful for our proof:

$$\widehat{P}P \subseteq I \wedge Q \subseteq P \Rightarrow (R \cap P)L \cap Q = R \cap Q.$$

To this effect, we write:

$(R \cap P)L \cap Q = R \cap Q$

$\Leftrightarrow$ $\{(R \cap P)L \cap Q \subseteq Q, R \cap Q \subseteq (R \cap P)L, R \cap Q \subseteq Q\}$

$(R \cap P)L \cap Q \subseteq R$

$\Leftarrow$ {Dedekind, [5]}

$(R \cap P \cap QL)(L \cap \widehat{(R \cap P)}Q) \subseteq R$

$\Leftarrow$ {hypothesis: $Q \subseteq P$}

$(R \cap P)\widehat{(R \cap P)}P \subseteq R$

$\Leftarrow$ {monotonicity of intersection}

$R\widehat{P}P \subseteq R$

$\Leftarrow$ {monotonicity of product}

$\widehat{P}P \subseteq I$

$\Leftarrow$ {hypothesis: $\widehat{P}P \subseteq I$}

true.

Using this lemma, we now show the proposition:

$P' \sqsupseteq_{Q \sqcup R} P$

$\Leftrightarrow$ {definition of relative correctness}

$((Q \sqcup R) \cap P)L \subseteq ((Q \sqcup R) \cap P')L$

$\Leftrightarrow$ {definition of $\sqcup$}

$(((\overline{RL} \cap Q) \cup (\overline{QL} \cap R) \cup (R \cap R)) \cap P)L$

$\subseteq (((\overline{RL} \cap Q) \cup (\overline{QL} \cap R) \cup (R \cap R)) \cap P')L$

$\Leftrightarrow$ {factoring L on both sides}

$(\overline{RL} \cap (Q \cap P)L) \cup (\overline{QL} \cap (R \cap P)L) \cup (Q \cap R \cap P)L$

$\subseteq (\overline{RL} \cap (Q \cap P')L) \cup (\overline{QL} \cap (R \cap P')L) \cup (Q \cap R \cap P')L$
$\Leftrightarrow$ {boolean algebra, $P' \sqsupseteq_R P$ and $P' \sqsupseteq_Q P$}
$(Q \cap R \cap P)L \subseteq (Q \cap R \cap P')L$
$\Leftarrow$ {for any relations A, B, $(A \cap B)L \subseteq AL \cap BL$}
$(Q \cap P)L \cap (R \cap P)L \subseteq (Q \cap R \cap P')L$
$\Leftarrow$ {$(Q \cap P)L \subseteq (Q \cap P')L$ and $(R \cap P)L \subseteq (R \cap P')L$}
$(Q \cap P')L \cap (R \cap P')L \subseteq (Q \cap R \cap P')L$
$\Leftarrow$ {rewriting the first L as LL and factoring L}
$((Q \cap P')L \cap (R \cap P'))L \subseteq (Q \cap R \cap P')L$
$\Leftarrow$ {we apply the lemma above to P' and $(R \cap P')$}
$(Q \cap R \cap P')L \subseteq (Q \cap R \cap P')L$
$\Leftrightarrow$ {tautology}
true. □

This proposition is interesting in practice, for the following reason: We had found in [4] that complex specifications can be composed from simpler specifications by means of the join operator; this proposition provides that in order to prove that a program p' is more-correct than a program p with respect to a complex specification $R = R' \sqcup R''$, it is sufficient to prove that p' is more-correct than p with respect to each component of R.

7 Concluding remarks

7.1 Summary

In this paper we have studied the concept of relative correctness, used it to propose a definition for program faults, then explored the implications of these two concepts on a variety of aspects of testing and fault removal. Among the most salient contributions of this paper, we cite the following:

- A definition of relative correctness, and an analysis of the proposed definition to ensure that it meets all the properties that one wants to see in such a concept.
- A definition of fault and fault removal, and the analysis of monotonic fault removal, as a process that transforms a faulty program into a correct program by a monotonic sequence of correctness-enhancing transformations.
- An analysis of program repair, highlighting that when repair candidates are evaluated by testing them for absolute correctness rather than relative correctness, one runs the risk of selecting programs that are not adequate repairs, and rejecting programs that are.
- A critique of the concept of fault density, and the introduction of fault depth as perhaps a more meaningful measure of the degree of imperfection of a faulty program; also the observation that for a given fault depth, the higher the fault density the better (which is the opposite of what fault density purports to represent).
- An analysis of techniques for testing that a program is more-correct than another with respect to a specification and discussion of the difference between testing a program for relative correctness and testing it for absolute correctness.
- A study of techniques for proving, by static analysis, that a program is more-correct than another with respect to a given specification, as well as techniques for decomposing a proof of relative correctness with respect to a compound specification into proofs of relative correctness with respect to its building components.

7.2 Related work

In [27] Logozzo et al. introduce a technique for extracting and maintaining semantic information across program versions: Specifically, they consider an original program P and a variation (version) P' of P, and they explore the question of extracting semantic information from P, using it to instrument P' (by means of executable assertions) and then pondering what semantic guarantees they can infer about the instrumented version of P'. The focus of their analysis is the condition under which programs P and P' can execute without causing an abort (due to attempting an illegal operation), which they approximate by sufficient conditions and necessary conditions. They implement their approach in a system called VMV (*Verification Modulo Versions*) whose goal is to exploit semantic information about P in the analysis of P' and to ensure that the transition from P to P' happens without regression; in that case, they say that P' is *correct relative to* P. The definition of relative correctness of Logozzo et al. [27] is different from ours, for several reasons: Whereas [27] talk about relative correctness between an original program and a subsequent version in the context of adaptive maintenance (where P and P' may be subject to distinct requirements), we talk about relative correctness between an original (faulty) software product and a revised version of the program (possibly still faulty yet more-correct)

in the context of corrective maintenance with respect to a fixed requirements specification; whereas [27] use a set of assertions inserted throughout the program as a specification, we use a relation that maps initial states to final states to specify the standards against which absolute correctness and relative correctness are defined; whereas [27] represent program executions by execution traces (snapshots of the program state at assertion sites), we represent program executions by functions mapping initial states into final states; finally, whereas Logozzo et al. define a successful execution as a trace that satisfies all the relevant assertions, we define it as an initial state/final state pair that falls within the relational specification.

In [21] Lahiri et al. introduce a technique called *Differential Assertion Checking* for verifying the relative correctness of a program with respect to a previous version of the program. Lahiri et al. explore applications of this technique as a trade-off between soundness (which they concede) and lower costs (which they hope to achieve). Like the approach of Logozzo et al. [27] (from the same team), the work of Lahiri uses executable assertions as specifications, represents executions by traces, defines successful executions as traces that satisfy all the executable assertions and targets abort-freedom as the main focus of the executable assertions. Also, they define relative correctness between programs P and P' as the property that P' has a larger set of successful traces and a smallest set of unsuccessful traces than P; they introduce relative specifications as specifications that capture functionality of P' that P does not have. By contrast, we use input/output (or initial state/final state) relations as specifications, we represent program executions by functions from initial states to final states, we characterize correct executions by initial state/final state pairs that belong to the specification, and we make no distinction between abort-freedom (a.k.a. safety, in [21]) and normal functional properties. Indeed, for us the function of a program is the function that the program defines between its initial states and its final states; the domain of this function is the set of states for which execution terminates normally and returns a well-defined final state. Hence execution of the program on a state s is abort free if and only if the state is in the domain of the program function; the domain of the program function is part of the function rather than being an orthogonal attribute; hence we view abort-freedom as a special form of functional attribute, rather than being an orthogonal attribute. Another important distinction with [21] is that we do not view relative correctness as a compromise that we accept as a substitute for absolute correctness; rather, we argue that in many cases, we ought to test programs for relative correctness rather than absolute correctness, regardless of cost. In other words, whereas Lahiri et al. argue in favor of relative correctness on the grounds that it optimizes a quality vs. cost ratio, we argue in favor on the grounds that it optimizes quality.

In [26], Logozzo and Ball introduce a definition of relative correctness whereby a program P' is correct relative to P (*an improvement* over P) if and only if P' has more good traces and fewer bad traces than P. Programs are modeled with trace semantics, and execution traces are compared in terms of executable assertions inserted into P and P'; in order for the comparison to make sense, programs P and P' have to have the same (or similar) structure and/or there must be a mapping from traces of P to traces of P'. When P' is obtained from P by a transformation, and when P' is provably correct relative to P, the transformation in question is called a *verified repair*. Logozzo and Ball introduce an algorithm that specializes in deriving program repairs from a pre-defined catalog that is targeted to specific program constructs, such as: contracts, initializations, guards, floating point comparisons. Like the work cited above ([21,27]), Logozzo and Ball model programs by execution traces and distinguish between two types of failures: contract violations, when functional properties are not satisfied; and run-time errors, when the execution causes an abort; for the reasons we discuss above, we do not make this distinction and model the two aspects with the same relational framework. Logozzo and Ball deploy their approach in an automated tool based on the static analyzer cccheck and assess their tool for effectiveness and efficiency.

In [35], Nguyen et al. present an automated repair method based on symbolic execution, constraint solving and program synthesis; they call their method SemFix, on the grounds that it performs program repair by means of semantic analysis. This method combines three techniques: fault isolation by means of statistical analysis of the possible suspect statements; statement-level specification inference, whereby a local specification is inferred from the global specification and the product structure; and program synthesis, whereby a corrected statement is computed from the local specification inferred in the previous step. The method is organized in such a way that program synthesis is modeled as a search problem under constraints, and possible correct statements are inspected in the order of increasing complexity. When programs are repaired by SemFix, they are tested for (absolute) correctness against some pre-defined test data suite; as we argue throughout this paper, it is not sensible to test a program for absolute correctness after a repair, unless we have reason to believe that the fault we have just repaired is the last fault of the program (how do we ever know that?). By advocating to test for relative correctness, we enable the tester to focus on one fault at a time and ensure that other faults do not interfere with our assessment of whether the fault under consideration has or has not been repaired adequately.

In [37], Weimer et al. discuss an automated program repair method that takes as input a faulty program, along with a set of positive tests (i.e., test data on which the program is known to perform correctly) and a set of negative tests (i.e., test data on which the program is known to fail) and returns a set of

possible patches. The proposed method proceeds by keeping track of the execution paths that are visited by successful executions and those that are visited by unsuccessful executions, and using this information to focus the search for repairs on those statements that appear in the latter paths and not in the former paths. Mutation operators are applied to these statements, and the results are tested again against the positive and negative test data to narrow the set of eligible mutants.

In [25] Le Goues et al. survey existing technology in automated program repair and identify open research challenges; among the criteria for automated repair methods, they cite applicability (extent of real-world relevance), scalability (ability to operate effectively and efficiently for products of realistic size), generality (scope of application domain, types of faults repaired) and credibility (extent of confidence in the soundness of the repair tool). Among the research issues they identify, they cite mining specifications for extant software, introducing formal methods to improve repair quality and user trust and modeling monotonic fault removal.

In [20] Kim and Smidts review several definitions of terms pertaining to safety and reliability (including faults), lament several discrepancies and shortcomings within and between these definitions, and make recommendations on how to remedy this situation. Whereas we are focused exclusively on faults, and particularly on software faults, Kim and Smidts discuss the broader terminology pertaining to safety, reliability and fault tolerance and do so for the broader context of digital systems. Also, whereas we are concerned with correctness and relative correctness as formally defined properties, Kim and Smidts are more interested in the ontological aspects of the debate than the formal semantics aspects.

7.3 Assessment and prospects

The research presented in this paper is clearly in its infancy; we have merely introduced some new definitions of old concepts and shown the ramifications that stem from these definitions. Yet we feel that in doing so, we have opened up many new venues of investigation, which we envision to explore:

- *Debugging without Testing* Traditionally, it is so inconceivable to debug a program without testing it that these two words are often used interchangeably; yet Sect. 6.1 shows precisely that this can be done, albeit (so far) in a special context; we envision to broaden the scope of this line of research.
- *Programming without Refinement* In Sect. 5.3, we argue that while refinement-based program derivation is a sufficient condition for producing correct programs, it may be viewed as unnecessarily strong; as a substitute, we show how we can derive a program by successive correctness-enhancing transformations rather than the traditional process of successive correctness-preserving transformations. We envision to elaborate on this idea.
- *Mutation Testing with Relative Correctness* In light of the discussions of Sect. 5.5, it appears that if we deploy mutation testing with relative correctness rather than absolute correctness, we may significantly improve the precision and recall of the technique; we envision to test this conjecture in practice.
- *Measuring Faultiness with Fault Depth* The discussions of Sect. 5.1.3 appear to show that fault depth is a better measure of product imperfection (failure rate, repair effort) than fault density; we envision to test this hypothesis.
- *Testing for Relative Correctness* We envision to investigate test data generation strategies that are appropriate for relative correctness and to generate broadly applicable conditions of relative correctness in the style of Proposition 7.

Acknowledgements The authors are grateful to the anonymous reviewers for their thoughtful, insightful feedback; the paper has been greatly enhanced on the basis of their comments and suggestions.

References

1. IEEE Std 7-4.3.2-2003 (2003) Ieee standard criteria for digital computers in safety systems of nuclear power generating stations. Technical report, The Institute of Electrical and Electronics Engineers
2. Arcuri A, Yao X (2008) A novel co-evolutionary approach to automatic software bug fixing. In: CEC. pp 162–168
3. Avizienis A, Laprie JC, Randell B, Landwehr CE (2004) Basic concepts and taxonomy of dependable and secure computing. IEEE Trans Dependable Secur Comput 1(1):11–33
4. Boudriga N, Elloumi F, Mili A (1992) The lattice of specifications: applications to a specification methodology. Form Asp Comput 4(6):544–571
5. Brink Ch, Kahl W, Schmidt G (1997) Relational methods in computer science. Springer, Berlin
6. Kim D, Nam J, Song J, Kim S (2013) Automatic patch generation learned from human-written patches. ICSE 2013:802–811
7. Desharnais J, Diallo N, Ghardallou W, Frias MF, Jaoua A, Mili A (2015) Relational mathematics for relative correctness. In: RAMICS, 2015, volume 9348 of LNCS, September 2015. Springer, Braga, Portugal, pp 191–208
8. Diallo N, Ghardallou W, Mili A (2015) Correctness and relative correctness. In: Proceedings of 37th international conference on software engineering, NIER track, Firenze, Italy, 20–22 May
9. Diallo N, Ghardallou W, Mili A (2015) Program derivation by correctness enhancements. In: Refinement 2015, Oslo, Norway, June 2015
10. Dijkstra EW (1976) A discipline of programming. Prentice Hall, Upper Saddle River
11. Gaertner FC (1999) Fundamentals of fault tolerant distributed computing in asynchronous environments. ACM Comput Surv 31:1–26
12. Ghardallou W, Diallo N, Mili A, Frias M (2016) Debugging without testing. In: Proceedings of international conference on software testing, Chicago, IL, April 2016

13. Gonzalez-Sanchez A, Abreu R, Gross HG, van Gemund AJC (2011) Prioritizing tests for fault localization through ambiguity group reduction. In: Proceedings of automated software engineering, Lawrence, KS
14. Gopinath D, Malik MZ, Khurshid S (2011) Specification based program repair using sat. In: Proceedings of TACAS, pp 173–188
15. Le Goues C, Nguyen T, Forrest S, Weimer W (2012) Genprog: a generic method for automated software repair. IEEE Trans Softw Eng 31(1):54–72
16. Gries D (1981) The science of programming. Springer, Berlin
17. Hehner ECR (1992) A practical theory of programming. Prentice Hall, Upper Saddle River
18. Hoare CAR (1969) An axiomatic basis for computer programming. Commun ACM 12(10):576–583
19. Khaireddine B, Zakharchenko A, Mili A (2017) A generic algorithm for program repair. In: Proceedings of FormaliSE, Buenos Aires, Argentina
20. Kim MC, Smidts CS (2015) Three suggestions on the definition of terms for the safety and reliability analysis of digital systems. Reliab Eng Syst Saf 135:81–91
21. Lahiri SK, McMillan KL, Sharma R, Hawblitzel C (2013) Differential assertion checking. In: Proceedings of ESEC/FSE, pp 345–355
22. Laprie JC (1991) Dependability: basic concepts and terminology: In: English, French, German. Italian and Japanese. Springer, Heidelberg
23. Laprie JC (1995) Dependability—its attributes, impairments and means. In: Predictably dependable computing systems. Springer, Berlin, pp 1–19
24. Laprie JC (2004) Dependable computing: concepts, challenges, directions. In: Proceedings of COMP-SAC
25. LeGoues C, Forrest S, Weimer W (2013) Current challenges in automatic software repair. Softw Qual J 21(3):421–443
26. Logozzo F, Ball T (2012) Modular and verified automatic program repair. In: Proceedings of OOPSLA, pp 133–146
27. Logozzo F, Lahiri S, Faehndrich M, Blackshear S (2014) Verification modulo versions: towards usable verification. In: Proceedings of PLDI, pp 294–304
28. Louhichi A, Ghardallou W, Bsaies K, Jilani LL, Mraihi O, Mili A (2014) Verifying loops with invariant relations. Int J Crit Comput Based Syst 5(1):78–102
29. Manna Z (1974) A mathematical theory of computation. McGraw-Hill, New York
30. Mili A, Desharnais J, Mili F, Frappier M (1994) Computer program construction. Oxford University Press, Oxford
31. Mili A, Frias M, Jaoua A (2014) On faults and faulty programs. In: Hoefner P, Jipsen P, Kahl W, Mueller ME (ed) Proceedings of RAMICS 2014, volume 8428 of LNCS. pp 191–207
32. Mili Ali, Tchier Fairouz (2015) Software testing: operations and concepts. Wiley, Hoboken
33. Mills HD, Basili VR, Gannon JD, Hamlet DR (1986) Structured programming: a mathematical approach. Allyn and Bacon, Boston
34. Mraihi O, Louhichi A, Jilani LL, Desharnais J, Mili A (2013) Invariant assertions, invariant relations, and invariant functions. Sci Comput Program 78(9):1212–1239
35. Nguyen HDT, Qi D, Roychoudhury A, Chandra S (2013) Semfix: program repair via semantic analysis. In: Proceedings of ICSE, pp 772–781
36. Debroy V, Wong WE (2010) Using mutation to automatically suggest fixes to faulty programs. In: Proceedings of ICST. pp 65–74
37. Weimer W, Nguyen T, Le Goues C, Forrest S (2009) Automatically finding patches using genetic programming. In: Proceedings of ICSE. pp 364–374
38. Zemín L, Guttiérrez S, Perez de Rosso S, Aguirre N, Mili A, Jaoua A, Frias M (2015) Stryker: Scaling specification-based program repair by pruning infeasible mutants with sat. Technical report, ITBA, Buenos Aires, Argentina