Relative Correctness: A Bridge Between Testing and Proving

Nafi Diallo[1], Wided Ghardallou[2], and Ali Mili[1]

[1] New Jersey Institute of Technology, Newark, NJ, USA
[2] University of Tunis El Manar, Tunis, Tunisia
ncd8@njit.edu, wided.ghardallou@gmail.com, ali.mili@njit.edu

Abstract. Relative correctness is the property of a program to be more-correct than another with respect to a specification. Whereas traditionally we distinguish between two categories of candidate programs, namely correct programs and incorrect programs, relative correctness arranges candidate programs on a partial ordering structure, whose maximal elements are the correct programs. Also, whereas traditionally we deploy proof methods on correct programs to prove their correctness and we deploy testing methods on incorrect programs to detect and remove their faults, relative correctness enables us to bridge this gap by showing that we can deploy static analytical methods to an incorrect program to prove that while it may be incorrect, it is still more-correct than another. We are evolving a technique, called *debugging without testing*, in which we can remove a fault from a program and prove that the new program is more-correct than the original, all without any testing (and its associated uncertainties/ imperfections). Given that there are orders of magnitude more incorrect programs than correct programs in use nowadays, this has the potential to expand the scope of proving methods significantly. Also, relative correctness has other broad implications for testing and proving, which we briefly explore in this paper.

Keywords

absolute correctness, relative correctness, program testing, program proving, debugging without testing, programming without refining, testing for relative correctness.

1 Blurring Traditional Distinctions

Relative correctness is the property of a program to be more-correct than another with respect to a given specification; intuitively, a program P' is more-correct than a program P with respect to a specification R if and only if P' behaves according to R more often than P, and violates R less egregiously (i.e. in fewer ways) than P. Whereas traditionally we distinguish between two categories of candidate programs for a given specification R, namely correct programs and incorrect programs, relative correctness enables us to arrange candidate programs over a partial ordering structure, whose maximal elements are the correct programs, and all non-maximal elements are incorrect.

Also, traditionally, proving methods and testing methods have been used on different sets of programs:

- Proving methods are deployed on correct programs to prove their correctness; they are of limited use when deployed on incorrect programs because even when a proof fails, we cannot always conclude that the program is incorrect, since we cannot tell whether the proof failed because the program is incorrect or because it was improperly documented (re: invariant assertions, intermediate assertions, etc). Some methods of program analysis can identify sources of faults when the correctness proof fails, but their scope is limited.
- Testing methods are deployed on incorrect programs to detect, locate and remove their faults; they are useless when deployed on correct programs, because no matter how often a program runs failure-free under test, we can never (in practice) conclude with certainty that it is correct.

We argue that consideration of relative correctness has the potential to alter the practice of proving methods and testing methods:

- Once we have a formal definition of relative correctness, we can deploy proving methods to an incorrect program to prove that while it may be incorrect, it is still more-correct than another. Given that there are orderes of magnitude more incorrect programs than correct programs, the ability to apply proving methods to incorrect programs expands the scope of these methods significantly. This approach is discussed in section 4.
- Relative Correctness can also alter the practice of software testing by recognizing the difference between testing for relative correctness and testing for absolute correctness. When we remove a fault from a program, we ought to test it for relative correctness rather than absolute correctness, unless we have reason to believe (how do we ever?) that the fault we have just removed is the last fault of the program. This matter is discussed in section 5.
- It has long been a cornerstone of software engineering wisdom that programs should not be developed then checked for correctness, but should instead be developed hand-in-hand along with their proof, with the proof leading the way [8]; echoing David Gries, Carrol Morgan talks about developing programs by calculation from their specification, in the same way that a mathematician solves an equation by computing its root [22]. The prevailing paradigm for developing programs from specifications is that of refinement, whereby a program is derived from a specification through a sequence of correctness-preserving transformations based on refinement. In section 6 we present an alternative paradigm based on relative correctness, illustrate it with a simple example, and briefly compare it to related work.

In section 2 we introduce the mathematical background that is needed to carry out our discussions, and in section 3 we introduce our definition of relative correctness for deterministic and non-deterministic programs. Also, we conclude in section 7 by summarizing our findings, discussing related work, and sketching future directions of research.

2 Background

2.1 Relational Mathematics

In this paper we use relations to represent specifications and programs, hence we devote this section to discussing some operations and properties of relations. We assume the reader familiar with elementary relational algebra [2], hence this section is not a tutorial on relations as much as it is an introduction to some relevant definitions and notations. We represent sets in a program-like notation by writing variable names and associated data types (sets of values); if we write S as:

```
   x: X; y: Y;
```

then we mean to let S be the cartesian product $S = X \times Y$; elements of S are usually denoted by s and the X- (resp. Y-) component of s is denoted by $x(s)$ (resp. $y(s)$). When no ambiguity arises, we may write x for $x(s)$, and x' for $x(s')$, etc. A *relation* R on set S is a subset of $S \times S$. Special relations on S include the *universal relation* $L = S \times S$, the identity relation $I = \{(s,s)|s \in S\}$ and the empty relation $\phi = \{\}$. Operations on relations include the set theoretic operations of union ($\cup$), intersection ($\cap$), difference ($\backslash$) and complement ($\overline{R}$); they also include the *converse* of a relation R defined by $\widehat{R} = \{(s,s')|(s',s) \in R\}$, the *domain* of a relation defined by $dom(R) = \{s|\exists s' : (s,s') \in R\}$, the *range* of a relation defined by $rng(R) = dom(\widehat{R})$, and the product of two relations R and R' defined by: $R \circ R' = \{(s,s')|\exists s'' : (s,s'') \in R \wedge (s'',s') \in R'\}$; when no ambiguity arises, we may write RR' for $R \circ R'$.

A relation R is said to be reflexive if and only if $I \subseteq R$, symmetric if and only if $R = \widehat{R}$, antisymmetric if $R \cap \widehat{R} \subseteq I$, asymmetric if and only if $R \cap \widehat{R} = \phi$ and transitive if and only if $RR \subseteq R$. A relation R is said to be *total* if and only if $I \subseteq R\widehat{R}$ and *deterministic* if and only if $\widehat{R}R \subseteq I$. A relation R is said to be a *vector* if and only if $RL = R$; vectors have the form $R = A \times S$ for some subset A of S; we use them as relational representations of sets. In particular, note that RL can be written as $dom(R) \times S$; we use it as a relational representation of the domain of R.

2.2 Program Semantics

Given a program `p` on space S written in a C-like notation, we define the function of `p` (denoted by P) as the function that `p` defines on S, i.e. the set of pairs (s,s') such that if program `p` starts execution in state s it terminates in state s'; we may, when no ambiguity arises, refer to a program and its function by the same name, P. Because our discussion of correctness and relative correctness refers to a notion of refinement, we give here a definition of this property.

Definition 2.1. *Given two relations R and R', we say that R'* refines *R (abbrev: $R' \sqsupseteq R$ or $R \sqsubseteq R'$) if and only if $RL \cap R'L \cap (R \cup R') = R$.*

Intuitively, this definition means that R' has a larger domain than R and that on the domain of R, R' assigns fewer images to each argument that does R. See Figure 1. We use refinement to define correctness.

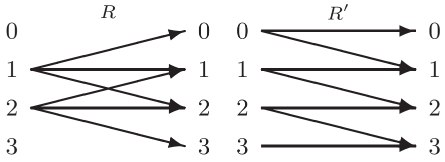

Fig. 1. $R' \sqsupseteq R$

Definition 2.2. *A program* `p` *on space S is said to be* correct *with respect to specification R on S if and only if its function P refines R.*

This definition is identical (modulo differences of notation) to traditional definitions of total correctness [8, 9, 19, 22].

3 Relative Correctness

3.1 Relative Correctness: Deterministic Programs

Proposition 3.1. *Due to [21]. Given a specification R and a program P, program P is correct with respect to R if and only if $(R \cap P)L = RL$.*

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

Definition 3.2. *Due to [20]. Given a specification R and two deterministic programs P and P', we say that P' is* more-correct *(resp.* strictly more-correct*) than P with respect to R if and only if $(R \cap P')L \supseteq (R \cap P)L$ (resp. $(R \cap P')L \supset (R \cap P)L$).*

To contrast relative correctness with the definition of correctness given in Definition 2.2, we may refer to the latter as *absolute correctness.* We refer to the domain of $(R \cap P)$ as the *competence domain* of P with respect to R; for deterministic programs, to be more-correct simply means to have a larger competence domain. See Figure 2; note that P' is more-correct than P but does not duplicate the correct behavior of P.

In [20], we find that relative correctness satisfies the following properties:

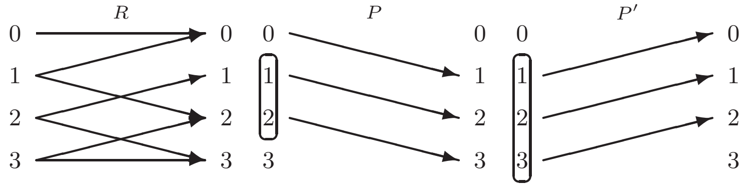

Fig. 2. $P' \sqsupseteq_R P$, Deterministic Programs

- *Ordering Properties.* Relative correctness is reflexive and transitive, but not antisymmetric (i.e. two candidate programs could be equally correct, yet compute distinct functions). As a very simple example, consider R defined by $R = \{(0,0),(1,0),(1,2)\}$ and let P and P' be defined by $P = \{(0,1),(1,0)\}$, $P' = \{(0,2),(1,2)\}$.
- *Relative correctness culminates in absolute correctness.* A (absolutely) correct program is more-correct than (or as correct as) any candidate program.
- *Relative Correctness Implies Enhanced Reliability.* If P' is more-correct than P with respect to R, then it is more reliable than P; but more-reliable is not equivalent to more-correct, as the latter is a logical property whereas the former is a stochastic property.
- *Relative Correctness and Refinement.* Program P' refines program P if and only if P' is more-correct than P with respect to *any* specification R. We write this property as:

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

3.2 Relative Correctness: Non-Deterministic Programs

In this section we generalize the definition of relative correctness to non-deterministic programs, as provided by [3]. One may want to ask why we want to discuss relative correctness of non-deterministic programs, when the programming language we use is deterministic. The answer is that we want to reason about the relative correctness of C-like programs without having to compute their function in all its detail; for example, if program P manipulates variables x, y and z and program P' is more correct than P with respect to R by virtue of dealing better than P with variables x and y (more in keeping with R), then we want to reach that conclusion by focusing exclusively on the behavior of P and P' on variables x and y. But doing so means that we do not need to determine how P and P' affect variable z; hence we will deal with non-deterministic representations of P and P'. We have the following definition, due to [3].

Definition 3.3. *Given a specification R and two programs P and P'. We say that P' is* more-correct *than P with respect to R if and only if: $(R \cap P')L \supseteq (R \cap P)L$ and $(R \cap P)L \cap \overline{R} \cap P' \subseteq P$. Also, we say that P' is* strictly more-correct *than P with respect to R if and only if at least one of the inequalities in this definition is strict.*

Interpretation: The first clause provides that P' has a larger competence domain than P. To understand the second clause, consider that the left-hand side of this clause represents the set of (s, s') pairs such that s is in the competence domain of P (re: $(R \cap P)L$) and s' is an image of s by P' that violates specification R (re: $\overline{R} \cap P'$). What this clause is providing is that any such (s, s') pair is in P; so that on the competence domain of P, P' does not violate specification R unless P does (but P may violate R in ways that P' does not). In other words: P' is more-correct than P with respect to R if and only if P' has a larger competence domain (first clause), and violates R in fewer ways (second clause). See Figure 3: The competence domain of P is $\{1, 2\}$ and that of P' is $\{1, 2, 3\}$. Program P' violates R by assigning 1 to 1 and 2 to 2; but P is guilty of the same misdeed, in addition to also assigning 0 to 2 and 3 to 1, in violation of specification R. Hence P' is more-correct than P with respect to R.

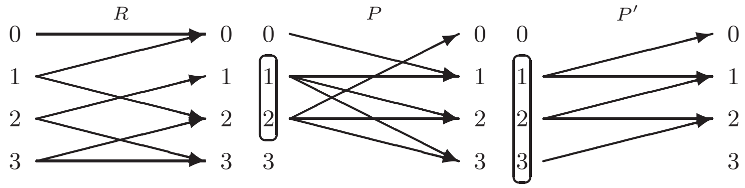

Fig. 3. $P' \sqsupseteq_R P$, Non-deterministic Specifications

4 Debugging Without Testing

4.1 What is a Fault?

Our study of relative correctness came about when we attempted to answer the question: what is a *fault* in a program? and when can we say that we have removed a fault from a program? The first matter we have to consider is that any definition of a fault must refer to a level of granularity at which we want to isolate faults. The coarsest level of granularity would be to consider the whole program as a possible fault, but that is clearly unhelpful as far as diagnosing and removing faults; more typical levels of granularity include a line of code, a lexeme, an expression, an assignment statement, elementary programming statements, etc. We use the term *feature* to refer to any part of the source code at an appropriate level of granularity, including non-contiguous parts.

Definition 4.1. *Due to [20]. Given a specification R and a program P, a* fault *in program P is any feature f that admits a substitute f' such that the program P' obtained from P by replacing f with f' is strictly more-correct than P. A* fault removal *in P is a pair of features (f, f') such that f is a fault in P and P' obtained from P by replacing f with f' is strictly more-correct than P.*

Using this definition, we can in principle remove a fault from a program without testing it; all we need to do is prove that the new program obtained by replacing f with f' is strictly more-correct than P. But in practice it is very difficult to use this definition, because it is difficult to compute the functions of P and P', then the competence domains of P and P', then compare them.

4.2 Proving Correctness and Incorrectness of Loops

The analysis of while loops by means of invariant relations provides a way to infer the relative correctness of iterative programs from partial semantic information. In [17] we discuss how to use invariant relations to prove the correctness or incorrectness of an iterative program of the form w: `while (t) {b;}`. If we let B be the function of the loop body (`{b;}`) and we let T be the vector that represents the loop condition (i.e. $T = \{(s, s')|t(s)\}$) then an invariant relation of the while loop w is a reflexive transitive superset of $(T \cap B)$. We are evolving a tool that can generate invariant relations for common combinations of code patterns; this tool matches an internal representation of the loop against prestored code patterns (called the *recognizers*) for which it has patterns of invariant relations; actual invariant relations are generated by instantiating the patterns of invariant relations with actual program variables. The following table shows, for the sake of illustration, two sample recognizers dealing with numeric variables; more information on this tool can be found at `https://selab.njit.edu/tools/fxloops.php`.

ID	Data	Statements	Invariant Relation Pattern

1R1	int x;
	const int a>0;
		x=x+a	$\{(s, s') \mid x \mod a = x' \mod a\}$

2R1	float x, y;
	const float a, b;
		x=x+a,
		y=y+b
			$\{(s, s') \mid ay - bx = ay' - bx'\}$.

Invariant relations are important for our purposes because they enable us to determine whether a loop is correct or not with respect to a specification, as shown in the following two propositions (which are due to [17]). Even though correctness is defined in terms of the program function, invariant relations enable us to rule on correctness or incorrectness long before we have collected all the necessary information to compute the loop function.

Proposition 4.2. Necessary condition of correctness. *Let w be a while loop of the form w =* `while (t) {b}` *that terminates for all states in S, let V be an invariant relation for w, and let R be a specification on S. If w is correct with respect to R then $(R \cap V)\overline{T} = RL$.*

This proposition provides, in effect, that any while loop that admits an invariant relation V that fails to meet this condition could not possibly be correct with respect to R. We say about such an invariant relation V that it is *incompatible* with the specification R. Any invariant relation that is not incompatible with the specification is said to be *compatible*.

Proposition 4.3. Sufficient condition of correctness. *Given a while loop w of the form* `while (t) {b}` *that terminates for all states in its space S, and given a specification R on S, if an invariant relation V of w satisfies the condition*

$$V\overline{T} \cap RL \cap (R \cup V \cap \widehat{\overline{T}}) = R$$

then w is correct with respect to R.

This proposition provides, in effect, that if an invariant relation V meets this condition, then it contains sufficient information to subsume the specification, and to prove the correctness of the loop with respect to R.

4.3 Proving Relative Correctness of Loops

While in the previous section we show how invariant relations can be used to prove the (absolute) correctness or incorrectness of a loop with respect to a specification, in this section we explore how they can be used to prove relative correctness of a loop over another with respect to a given specification. A proposition given in [6] provides an intuitive result to the effect that if we alter a while loop in such a way as to migrate an invariant relation from the incompatible column to the compatible column, while preserving all the compatible invariant relations, we obtain a while loop that is strictly more-correct; in other words, whatever alteration we make to the while loop as described above can be considered as a monotonic fault removal. This proposition can be used to diagnose and remove faults in a while loop:

- *Evidence that a fault exists.* The first step in fault removal is to have evidence that a fault does indeed exist. In our case, the existence of an incompatible invariant relation proves that the program has a fault.
- *Locating the fault.* Among all the incompatible invariant relations, we select that which involves the smallest number of program variables; this enables us to focus our attention on those statements of the program that involve the variables in question.
- *Removing the fault.* We must modify the selected variable(s), while ensuring that all the compatible invariant relations are preserved. The constraint of preserving the compatible invariant relations is used for guidance in deciding how to change the selected variables.
- *Verifying that the fault has been removed.* For each modification that is generated in the previous step, we deploy the invariant relations generator and check whether the incompatible invariant relation identified in the second step has been replaced by a compatible invariant relation. Then we are sure that the new loop is strictly more-correct than the original loop (and the fault has been removed).

As an illustration of this approach, we consider the following example, which we borrow from [7]. The space of the specification is defined by the following variable declarations: `char q[]; int let, dig, other, i, l; char c;`.
The specification R that we use for relative correctness is:
$R = \{(s,s')|q \in list\langle\alpha_A \cup \alpha_a \cup \vartheta \cup \sigma\rangle \wedge let' = let + \#_a(q) + \#_A(q) \wedge dig' =$

$dig + \#_{\vartheta}(q) \wedge other' = other + \#_{\sigma}(q)\}$
where $list\langle T\rangle$ denotes the set of lists of elements of type T, $\#_A$, $\#_a$, $\#_\vartheta$ and $\#_\sigma$ the functions that to each list q assign (respectively) the number of upper case alphabetic characters, lower case alphabetic characters, numeric digits and symbols. The (faulty) program that we consider is w:

```
{i=0; let=0; dig=0; other=0; l=strlen(q);
    while (i<l) {c = q[i];i++;
        if (A<=c && Z>=c) let=let-1;
        else if (a<=c && z>=c) let=let-1;
             else if (0>c && 9>=c) dig=dig+1;
                  else other=other+1;}}
```

We find the following invariant relations of this while loop, where $\sigma 1$ and $\sigma 2$ designate the set of characters whose Ascii codes are less than, and (respectively) greater than equal to, the code of '0':

- $V_0 = \{(s,s')|q = q'\}$
- $V_1 = \{(s,s')|i \leq i'\}$
- $V_2 = \{(s,s')|dig \leq dig'\}$
- $V_3 = \{(s,s')|other \leq other'\}$
- $V_4 = \{(s,s')|let \geq let'\}$
- $V_5 = \{(s,s')|let - \#_{a\cup A}(q[i..l-1]) = let' - \#_{a\cup A}(q'[i'..l-1])\}$
- $V_6 = \{(s,s')|dig + \#_{\sigma 1}(q[i..l-1]) = dig' + \#_{\sigma 1}(q'[i'..l-1])\}$
- $V_7 = \{(s,s')|other + \#_{\sigma 2\cup\vartheta}(q[i..l-1]) = other' + \#_{\sigma 2\cup\vartheta}(q'[i'..l-1])\}$

The following table shows which of these invariant relations are compatible, and which are incompatible.

Compatible Invariant Relations	Incompatible Invariant Relations
V_0, V_1, V_2, V_3	V_4, V_5, V_6, V_7

Becuse the *incompatible* column is non-empty, we conclude that the program is incorrect with respect to R, hence we must enhance its correctness. To this effect, we select the incompatible invariant relation V_4 for remediation, which leads us to focus on variable let for fault removal. Preservation of the compatible invariant relations mandates that let be modified under the following condition: $let \leq let'$. We propose: `let=let+1;`. The generation of invariant relations of the new loop yields the following table:

Compatible Invariant Relations	Incompatible Invariant Relations
V_0, V_1, V_2, V_3, V_4', V_5'	V_6, V_7

Application of the same process one more time yields the following program:

```
{i=0; let=0; dig=0; other=0; l=strlen(q);
    while (i<l) {c = q[i];i++;
        if (A<=c && Z>=c) let=let+1;
        else if (a<=c && z>=c) let=let+1;
             else if (0<=c && 9>=c) dig=dig+1;
                  else other=other+1;}}
```

Analysis of this program produces 8 invariant relations, which are all compatible.

Compatible Invariant Relations	Incompatible Invariant Relations
V_0, V_1, V_2, V_3, V_4', V_5', V_6', V_7'	

This does not prove that the program is now correct, all it proves is that we have no evidence (in the forms of an incompatible invariant relation) that it is incorrect. To establish correctness, we must ensure that the intersection of all the compatible invariant relations satisfies the sufficient condition provided by Proposition 4.3, which it does. All the faults have been removed; we now have a correct program.

5 Testing for Relative Correctness

The usual process of software debugging proceeds as follows: We observe a failure of the program; we analyze the failure and formulate a hypothesis on its cause; we modify the source code on the basis of our hypothesis; and finally we test the new program to ensure that it is now correct. But there is a serious flaw in this process: when we remove a fault from an incorrect program, we have no reason to expect the new program to be correct, unless we know (how do we ever?) that the fault we have just removed is the last fault of the program; hence when a fault is removed from a program, the new program ought to be tested for relative correctness over the original program, rather than for absolute correctness. Of course, regression testing is supposed to ensure monotonicity of fault removal, but regression testing is essentially a test data selection matter, whereas the difference between testing for relative correctness and testing for absolute correctness involves other aspects. We argue that testing a program for relative correctness has an impact on three aspects of testing, namely test data selection, test oracle design, and test coverage assessment.

- *Test data selection.* The problem of test data selection can be summarized as follows: We are given a large or infinite test space S, and we must select a small subset thereof T such that the behavior of candidate programs on T is a faithful predictor of their behavior on S. The difference between absolute correctness and relative correctness is that for absolute correctness with respect to specification R, the test space S is $dom(R)$ whereas for relative correctness over P with respect to R the test space S is $dom(R \cap P)$.
- *Test oracle design.* Let $\Omega(s, s')$ be the test oracle for absolute correctness derived from specification R. Because relative correctness over program P tests a candidate program P' for Ω only for those states on which P is successful, the oracle for relative correctness $\omega(s, s')$ can be written as:
  $\omega(s, s') \equiv (\Omega(s, P(s)) \Rightarrow \Omega(s, s'))$.
- *Test Coverage Assessment.* It is not sufficient to know that some program P' has executed successfully on a test data set of size N using oracle $\omega(s, s')$; it is also necessary to know what percentage of the test data set satisfy the precondition $\Omega(s, P(s))$ (i.e. is P' more-correct than P because P' is very good or because P is very bad?).

To illustrate the difference between absolute correctness and relative correctness, we consider the same program as section 4, and we resolve to remove its faults not by static analysis, as we did there, but by testing for relative correctness after each fault removal. To this effect, we proceed iteratively as follows, starting from the original program:

1. Using muJava [18], we generate mutants of the program, and submit each mutant to three tests:
   - A test for absolute correctness, using oracle $\Omega(s, s')$ derived from specification R.
   - A test for relative correctness, using oracle $\omega(s, s')$ derived from $\Omega(s, s')$.
   - A test for strict relative correctness, which in addition to relative correctness also ensures that there is at least one state on which the mutant satisfies Ω whereas the base program fails it.
2. We select those mutants which prove to be strictly more-correct than the base program, make each one of them a base program on which we apply recursively the same procedure, starting from step 1 above.

We invoke muJava with the option of mutating statements and conditions and we test every mutant for relative correctness, strict relative correctness and absolute correctness using randomly generated test data of size 1000. Every invokation of muJava generates exactly 64 mutants, which we label by indices 1 through 64; hence for example $m4.53.8$ is mutant 8 of mutant 53 of mutant 4 of the original program. The outcome of this experiment is shown in Figure 4. The arcs represent relative correctness relationships; at the bottom of this graph is the original program, and at the top is the corrected version of the program. Note that the test for absolute correctness kept coming empty-handed every time except whenever muJava produced the correct program P'. i.e. six times. The test for relative correctness returned *true* for every arc in Figure 4 i.e. 25 times; it enabled us to remove faults one at a time, until we reach a correct program. Note also that many mutations prove to be perfectly commutative; such is the case for 4, 8 and 53. Note further that, if we assume for the sake of argument that our test is exhaustive, then the number of arcs emerging from each program represents the number of faults in that program. For example, program P has four faults even though it is three fault removals away from being correct; we say that P has a *fault density* of 4 and a *fault depth* of 3.

6 Programming Without Refinement

For all its interest, program verification is really a dubious bargain: the idea that we write programs in an informal/ approximate manner then try to prove their correctness is not very sound. What is more rational is to seek means to write programs that are certified to be correct by construction [8, 9, 22, 23]. Given a specification, we can do so in one of two ways:

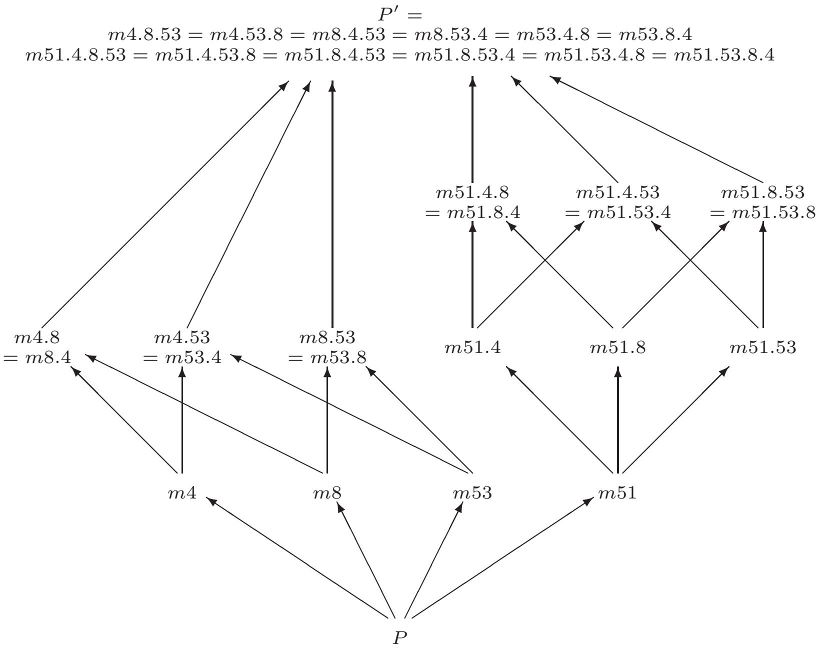

Fig. 4. Program Repair by Stepwise Correctness Enhancement

- *Preserve correctness, achieve program quality.* Starting from the specification, we transform it progressively to make it more and more program-like, while preserving correctness through refinement. This is the traditional process of program construction by successive refinements.
- *Preserve program quality, achieve correctness.* Starting from the trivial program *abort*, we transform it progressively to make more and more-correct with respect to R until it is correct. In [5], we present this model as an alternative program derivation paradigm, and discuss some of its advantages; we refer to this approach as *programming without refinement*.

We illustrate this process on a program that computes the number of alphabetic characters, numeric characters, and special symbols in an input string. We consider the same space and relation as those given in Section 4, and we present a sequence of programs P_0 ... P_4 that are increasingly more-correct, though not increasingly more-refined. We do not have a methodology for how to do this in general, hence we merely present the programs and their competence domains.

P_0 `{abort}`. $CD_0 = \phi$.

P_1
```
{i=0; let=0; dig=0; other=0; l=strlen(q);
 while (i<l) { c = q[i]; i++;
    if (’A’<=c && ’Z’>=c) let+=1;}}
```
$CD_2 = \{s|q \in list\langle\alpha_A\rangle\}$.

P_2
```
{i=0; let=0; dig=0; other=0; l=strlen(q);
 while (i<l) {c = q[i]; i++;
```

```
       if (’A’<=c && ’Z’>=c) let+=1;
       else if (’a’<=c && ’z’>=c) let+=1;}}
```

$CD_2 = \{s|q \in list\langle \alpha_A \cup \alpha_a \rangle\}$.

P_3
```
{i=0; let=0; dig=0; other=0; l=strlen(q);
 while (i<l) {c = q[i]; i++;
    if (’A’<=c && ’Z’>=c) let+=1;
    else  if (’a’<=c && ’z’>=c) let+=1;
    else  if (’0’<=c && ’9’>=c) dig+=1;}}
```

$CD_3 = \{s|q \in list\langle \alpha_A \cup \alpha_a \cup \nu \rangle\}$.

P_4
```
{i=0; let=0; dig=0; other=0; l=strlen(q);
 while (i<l) {c = q[i]; i++;
    if (’A’<=c && ’Z’>=c) let+=1;
    else if (’a’<=c && ’z’>=c) let+=1;
    else if (’0’<=c && ’9’>=c) dig+=1;
    else other+=1;}
```

$CD_4 = \{s|q \in list\langle \alpha_A \cup \alpha_a \cup \nu \cup \sigma \rangle\}$.

Clearly, we do have $P_0 \sqsubseteq_R P_1 \sqsubseteq_R P_2 \sqsubseteq_R P_3 \sqsubseteq_R P_4$; also, we find $CD_4 = dom(R)$, hence (by Proposition 3.1) $P_4 \sqsupseteq R$, i.e. P_4 is correct with respect to R.

This process bears a striking resemblance to test driven derivation (TDD) [10, 24], in the following sense: if we let Q_1, Q_2, Q_3, ... Q_n be the successive test data samples (in the form of (input, output) pairs) used in successive iterations of TDD, and if we let R_i be defined as $R_i = \bigcup_{k=1}^{i} Q_i$, then we argue that TDD is nothing but an instance of programming without refinement, where the specification is $R = R_n$ and the successive competence domains are $CD_i = dom(R_i)$. Clearly, the competence domains are increasingly large, by construction of R_i, hence we do have $P_1 \sqsubseteq_R P_2 \sqsubseteq_R P_3 \sqsubseteq_R ... \sqsubseteq_R P_{n-1} \sqsubseteq_R P_n$, where P_i is the program obtained after considering the i^{th} test data sample Q_i.

7 Concluding Remarks

In [1, 12–14] Laprie et al. discuss various aspects of dependability, and argue that faults are at the center of the study of dependability;and yet, they provide a rather vague definition of a fault: A *fault* is the adjudged or hypothesized cause of an error [1]. In [3, 4, 20] we attempt to give a formal definition of a fault, and find that a way (the only way?) to do so is to introduce the concept of relative correctness. Beyond defining faults, this concept has many implications, of which we discuss three in this paper, as they pertain to program proving and program testing.

– *Impact on Proving.* Whereas traditionally proving methods are deployed exclusively on correct programs to prove their correctness, we argue that relative correctness enables us to apply proving methods to an incorrect program to prove that despite being incorrect, it is still more correct than another. Given that there are orders of magnitude more incorrect programs than correct programs, this points to a potential expansion of the scope of static analysis methods.

- *Impact on Testing.* We argue that when we remove a fault from a program we ought to test it for relative correctness rather than absolute correctness; of course regression testing is an attempt to test for relative correctness through the selection of targeted test data, but we argue that testing for relative correctness has an impact not only on test data selection, but also on oracle design and test coverage assessment.
- *Impact on Program Derivation.* Whereas the traditional paradigm of program derivation is to proceed by successive correctness-preserving transformations on the basis of refinement, we argue that it is also possible to proceed by successive correctness-enhancing transformations on the basis of relative correctness. One of the main advantages of our paradigm is that it models not only the derivation of programs from scratch, but also many aspects of software evolution. Given that more software is produced by evolution than from scratch, this approach carries significant potential in practice.

Other authors have introduced similar-sounding but distinct concepts of relative correctness [11, 15, 16] in the context of software testing and program repair. Their work differs from ours in terms of its specification format (executable assertions, vs relations), its program semantics (execution traces, vs. program functions), its definition of correctness (all assertions are true, vs refinement), its definition of relative correctness (more valid traces, fewer invalid traces vs larger competence domain and fewer violations), and its goals (fault removal, vs proving, testing, derivation and evolution).

Whereas most other authors approximate while loops by unrolling them a number of times, we approximate them by means of invariant relations. We view the contrast between unrolling a loop and capturing its behavior by invariant relations as a choice between capturing all the functional details of a few iterations, and capturing some functional detail of all the iterations. We argue that capturing all the functional details is typically unnecessary (not all the loop's functional properties are worthy / relevant), and modeling a limited number of iterations is typically insufficient (the behavior of the loop for a limited number of iterations may not indicate its behavior for an arbitrary number thereof).

Acknowledgement The authors are very grateful to the anonymous reviewers for their thoughful, valuable feedback.

References

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

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

[3] Jules Desharnais, Nafi Diallo, Wided Ghardallou, Marcelo Frias, Ali Jaoua, and Ali Mili. Mathematics for relative correctness. In *Relational and Algebraic Methods in Computer Science, 2015*, pages 191–208, Lisbon, Portugal, September 2015.

[4] Nafi Diallo, Wided Ghardallou, and Ali Mili. Correctness and relative correctness. In *Proceedings, 37th International Conference on Software Engineering*, Firenze, Italy, May 20–22 2015.

[5] Nafi Diallo, Wided Ghardallou, and Ali Mili. Program derivation by correctness enhancements. In *Proceedings, Refinement 2015*, Oslo, Norway, June 2015.

[6] Wided Ghardallou, Nafi Diallo, Ali Mili, and Marcelo Frias. Debugging without testing. In *Proceedings, International Conference on Software Testing*, Chicago, IL, April 2016.

[7] Alberto Gonzalez-Sanchez, Rui Abreu, Hans Gerhart Gross, and Arjan J.C. van Gemund. Prioritizing tests for fault localization through ambiguity group reduction. In *proceedings, Automated Software Engineering*, Lawrence, KS, 2011.

[8] David Gries. *The Science of programming*. Springer Verlag, 1981.

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

[10] David Janzen and Hossein Saiedian. Test driven development: Concepts, taxonomy and future direction. *IEEE Computer*, 38(9):43–50, September 2005.

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

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

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

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

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

[16] Francesco Logozzo, Shuvendu Lahiri, Manual Faehndrich, and Sam Blackshear. Verification modulo versions: Towards usable verification. In *Proceedings, PLDI*, page 32, 2014.

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

[18] Yu Seung Ma, Jeff Offutt, and Yong Rae Kwon. Mujava: An automated class mutation system. *Software Testing, Verification and Reliability*, 15(2):97–133, June 2005.

[19] Zohar Manna. *A Mathematical Theory of Computation*. McGraw Hill, 1974.

[20] Ali Mili, Marcelo Frias, and Ali Jaoua. On faults and faulty programs. In Peter Hoefner, Peter Jipsen, Wolfram Kahl, and Martin Eric Mueller, editors, *Proceedings, RAMICS: 14th International Conference on Relational and Algebraic Methods in Computer Science*, volume 8428 of *Lecture Notes in Computer Science*, pages 191–207, Marienstatt, Germany, April 28–May 1st 2014. Springer.

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

[22] Carrol C. Morgan. *Programming from Specifications*. International Series in Computer Sciences. Prentice Hall, London, UK, 1998.

[23] Jose N. Oliveira. Programming from metaphorisms. *Journal of Logic and Algebraic Programming*, 2016.

[24] Daniel Perelman, Sumit Gulwani, Dan Grossman, and Peter Provost. Test driven synthesis. In *Proceedings, 35th ACM SIGPLAN Conference, PLDI*, volume 49, pages 408–418, Edinburgh, UK, 2014.