The Bane of Generate-and-Validate Program Repair: Too Much Generation, Too Little Validation

Anonymous, Pending Review

Keywords: absolute correctness, relative correctness, patch generation, patch validation, program repair, Cardumen tool

Abstract: To repair a program does not mean to make it (absolutely) correct; it only means to make it more-correct than it was originally. Yet, in the absence of a concept of relative correctness, program repair methods and tools have performed patch validation using various approximations of absolute correctness. In this paper, we show by means of analytical and empirical arguments that an approach to program repair based on relative correctness can reduce the search space of patch generation without reducing recall, and can enhance the precision of patch validation while maintaining adequate performance.

1 Introduction: Generation and validation in Program Repair

To repair a program does not necessarily mean to make it (absolutely) correct; it only means to make it more-correct, in some sense, than it was originally. This is not a mere academic distinction: Given that software products typically have a dozen faults per KLOC, it is important for program repair methods and tools to be geared towards mapping incorrect programs into incorrect, albeit more-correct, programs. In the absence of a formal concept of relative correctness (i.e. the property of a program $P'$ to me more-correct than $P$ with respect to a specification $R$), program repair methods have resorted to approximations of absolute correctness, including the preservation of correct behavior, and the enhancement of reliability. But as we discuss in this paper, the preservation of correct behavior is sufficient but an unnecessary condition of relative correctness (at least as we define it), and reliability enhancement is a necessary but an insufficient condition of relative correctness. Hence using the preservation of correct behavior in the retrieval of repair candidates is prone to cause the loss of recall, and using reliability enhancement is prone to cause the loss of precision; in the empirical part of this paper, we use a test oracle that is certified to test for strict relative correctness, and we show by means of experiments that it yields significantly better results than current tools.

As a field of research and development, the discipline of program repair has achieved great strides over the past decades, producing a continuous stream of increasingly sophisticated engineering solutions spanning several programming languages and several categories of faults (Mechtaev et al., 2016; Monperrus, 2014; D. et al., 2013; and Monperrus M., 2013; LeGoues et al., 2013; Nguyen et al., 2013; Debroyn and Wong, 2013; Qi et al., 2015; DeMarco et al., 2014; LeGoues et al., 2012; Goues et al., 2012; Gupta et al., 2017; Soto and Goues, 2018; Wen et al., 2018; Xin and Reiss, 2017; Le et al., 2017; Wen et al., 2018; Gazzola et al., 2019; Xiong et al., 2017; Jiang et al., 2018; Xuan and Monperrus, 2014; Saha et al., 2019; Goues et al., 2019). Program repair methods are usually divided into two broad categories: Generate-and-Validate methods, which proceed by generating a large set of candidate patches, then testing them in sequence until a satisfactory patch is found; Constraint-Based methods, which differ from Generate-and-Validate methods by the fact that they integrate validation concerns into the generation phase, to control the size of the search space of candidate patches. We focus our discussion on the Generate-and-Validate family of methods, though some of what we say applies also to Constraint-Based methods, since they too rely on absolute correctness for patch validation.

In the Generate-and-Validate methods, the first phase is to generate candidate repairs using various resources, such as selected mutation operators, excerpts from the source code, code segments from patch repositories, etc. In the absence of a concept of relative correctness, candidate repairs are tested for absolute correctness, typically with respect to some sample of input/output test data. A repair is successful if a (absolutely) correct program is generated in the patch generation phase. Now, any definition of relative correctness ought to be such that absolute correctness logically implies relative correctness: i.e. if a
program $P$ is absolutely correct with respect to some specification $R$, then $P$ is more-correct-than or as-correct-as any other candidate program. This means that in any given set of repair candidates for program $P$ with respect to specification $R$, there would be more programs that are more-correct-than $P$ with respect to $R$ than programs that are absolutely correct with respect to $R$. Hence if our criterion for successful repair is based on absolute correctness, then we have to generate a (much) larger set of repair candidates before we encounter a candidate that satisfies the criterion. This is the basis of our claim that using absolute correctness as the criterion for success of program repair forces us to generate larger search spaces.

To address the challenge of large search spaces, program repair methods have sometimes applied pruning techniques; regardless of how much care one exercises in applying pruning techniques, it is virtually impossible to avoid the risk of loss of recall. Another way to control the computational cost that stems from large search spaces is to shorten validation by using small test data sets; this in turn raises the risk of loss of precision. In the absence of any guarantees of precision and recall, program repair methods are typically validated by testing them on shared benchmarks of programs and faults, rather than by any verifiable analytical claim. Also, with the possible exception of (Saha et al., 2019), all program repair methods are tested using one fault at a time (still, even the method of Saha et al. (Saha et al., 2019) applies in very special cases). So long as we get to choose how many faults we seed into a program, we can seed one at a time; but real software typically contains a dozen faults per KLOC, hence program repair methods and tools ought to be geared towards mapping an incorrect program into a possibly incorrect program. This can be done if we have a concept of relative correctness, i.e. the property of a program $P$ to be more-correct than a program $P$ with respect to a specification $R$. In the same way that (traditional) absolute correctness serves as the criterions against which we validate the derivation of a program from a specification $R$, relative correctness ought to serve as the criterion against which we validate the transformation of an incorrect program $P$ into a program $P'$ that is more-correct than $P$ with respect to $R$ (yet possibly still incorrect with respect to $R$).

In this paper we discuss a definition of relative correctness, and study how it can be used as a basis for a disciplined approach to program program. Once we have adopted the premise that to repair a program means to make it more-correct, then it is easy to see that any program repair algorithm ought to be a variation of the generic pattern:

```

enhance relative correctness until we achieve absolute correctness
```

Adopting this discipline affords us several attributes that we do not see in the program repair methods and tools that are available in the literature today:

- **The ability to specify and verify correctness properties of program repair algorithms.** The ontology and logic of relative correctness that we use in this paper enables us to formulate specifications of repair algorithms (in the form of precondition/postcondition pairs), and to verify the correctness of the algorithm with respect to the specification, using Hoare logic (Hoare, 1969). This is in contrast with existing program repair methods and tools, which are validated solely by testing them on shared benchmark programs and faults.

- **The ability to repair programs with more than one fault.** A relative correctness-based approach enables us to remove several faults in sequence, by enhancing the relative correctness of the program in a stepwise fashion, until absolute correctness is achieved; we illustrate this claim by means of experiments involving benchmark programs in which we seed two faults. This will be illustrated in section 5.

- **A stepwise process of program repair.** Typically, program repair methods are triggered by an observation of failure, and are geared towards changing the program to remedy the observed failure; so that when the program fails for input $x$, they resolve to alter the program to make it run successfully for $x$, while preserving the program’s correct behavior. Given that the observed failure may be due to the combination of an unknown/ unbounded number of faults, this goal may prove elusive. We argue for a stepwise approach which proceeds one fault at a time, rather than one failure at a time. Hence when we observe that program $P$ fails at input $x$, we only conclude that $P$ is incorrect (not that it fails specifically at $x$). Also, we do not ask: *How can we make the program succeed for input $x$?* Rather we only ask: *How can we make the program more-correct?* If we keep enhancing the correctness of $P$, then it will eventually succeed for input $x$, but letting fault repair, rather than failure remediation, drive the process means that we let the program expose its faults in the order it determines, and we remove them as they appear. See Figure 1. So long as we test program repair tools on programs that have a single fault, the difference between the two policies cannot be observed; but realistic programs typically have multiple faults, and in section 5 we show that...
when we seed benchmark programs with several faults, the policy of one-fault-at-a-time will prove to be more effective. We conjecture that the emphasis on failure remediation rather than fault repair may be the cause of overfitting in program repair, alluded to in (Smith et al., 2015).

- The Need for a Large Test Data Set. Test data used in program repair is usually divided into two categories: positive test data, which the original program is known to pass and we want to maintain; and negative test data, which the original program fails and we want the repaired program to pass. In this paper we argue for the need to have a large test data set, for the sake of precision, and within that, a large set of negative test data, so as to better monitor/ control the evolution of the program towards absolute correctness. We will discuss in section 5 why the benefit in terms of precision far exceeds the cost overhead.

In section 2 we briefly discuss the definition of relative correctness that we adopt, along with its implications for program repair. In section 3 we discuss a generic algorithm for program repair, and in section 4 we use the ontology and axiomatization of relative correctness to formulate specifications of this algorithm, then we outline how to prove the correctness of this algorithm. In section 5 we create an instance of the generic algorithm by invoking the patch generation function of Cardumen (Martínez and Monperrus, 2018b), a component of the Astor framework (Martínez and Monperrus, 2018a); we refer to the resulting tool as CRCFix (CRC: Cardumen + Relative Correctness). We illustrate the performance of CRCFix by running it on benchmark programs seeded with several faults, and comparing its results to those of the original Cardumen tool.

2 Absolute and Relative Correctness

2.1 Relational Mathematics

We assume the reader familiar with elementary relational algebra (Brink et al., 1997), hence this section is not a tutorial on relations as much as it is an introduction to some terminology and notations. We represent sets in a program-like notation by writing variable names and associated data types; if we write $S$ as: $x: X; y: Y$; 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 $\emptyset = \{\}$. Operations on relations include the set theoretic operations of union, intersection, difference and complement; they also include the domain of a relation defined by $\text{dom}(R) = \{(s,s') | (s,s') \in R\}$, and the product of two relations $R$ and $R'$ defined by: $R \times R' = \{(s,s')(s',s'') | (s,s') \in R \land (s',s'') \in R'\}$; when no ambiguity arises, we may write $R R'$ for $R \times R'$. The pre-restriction of relation $R$ to set $T$ is the relation denoted by $T \setminus R$ and defined by: $T \setminus R = \{(s,s') | s \in T \land (s,s') \in R\}$. The converse of relation $R$ is the relation denoted by $\hat{R}$ and defined by $\hat{R} = \{(s,s') | (s',s) \in R\}$.

A relation $R$ is said to be reflexive if and only if $I \subseteq R$, symmetric if and only if $R = \hat{R}$, antisymmetric if and only if $R \cap \hat{R} \subseteq I$, and transitive if and only if $RR \subseteq R$. A relation $R$ is said to be deterministic (or: a function) if and only if $RR \subseteq I$. A relation $R$ is said to be a vector if and only if $RL = R$; vectors have the form $R = A \times S$ for some subset $A$ of $S$; we use them as relational representations of sets. In particular, note that $RL$ can be written as $\text{dom}(R) \times S$; we use it as a relational representation of the domain of $R$. We may sometimes, for the sake of convenience, use the same symbol to represent a set (say $T$) and the vector $(T \times S)$ that represents the same set, in relational form. Hence, for example, the restriction of relation $R$ to set $T$ can be written as $T \cap R$, where we interpret $T$ as a vector. We admit without proof (a well known property of functions) that if $F$ and $G$ are functions then $F = G$ if and only if $F \subseteq G$ and $GL \subseteq FL$.

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 set of pairs $(s,s')$ such that if program $p$ starts execution in state $s$ it terminates in state $s'$; we may, when no ambiguity arises, refer to a program and its function by the same name, $P$.

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

Intuitively, this definition means that $R'$ has a larger domain than $R$ and that on the domain of $R$, $R'$ assigns fewer images to each argument that does $R$. It is the relational equivalent of the traditional definition that equates refinement with a weaker precondition and/or a stronger postcondition (Gries, 1981; Dijkstra, 1976).
Before we propose a definition of relative correctness, we consider the question: What constitutes a sound definition of relative correctness? We have identified four properties we feel relative correctness ought to satisfy.

- **Reflexivity and Transitivity, but not Antisymmetry.** It is clear why we want reflexivity and transitivity; we do not want antisymmetry because we want to admit that two programs be equally correct but distinct.

- **Culmination in Absolute Correctness.** We want an absolutely correct program to be more-correct than (or as correct as) any program.

- **Relative Correctness and Reliability.** If \( P' \) is more-correct than \( P \) with respect to specification \( R \), then we want \( P' \) to be more reliable than \( P \) for any operational profile (Musa, 1993) (i.e. probability distribution over \( \text{dom}(R) \)).

- **Relative Correctness and Refinement.** Refinement is a binary relation between two programs that means essentially that one program is more capable than another: \( P' \) refines \( P \) if and only if whatever \( P \) can do, \( P' \) does as well or better. By contrast, relative correctness is a tri-parite relation that means: \( P' \) is better than \( P \) for the purposes of a particular specification \( R \). Given these two interpretations, we want to think that \( P' \) refines \( P \) if and only if \( P' \) is more-correct than \( P \) with respect to any specification \( R \).

### 2.3 Criteria for Relative Correctness

**Proposition 1.** Due to (Mills et al., 1986). Given a specification \( R \) and a deterministic program \( P \), program \( P \) is correct with respect to \( R \) if and only if \( (R \cap P)L = RL \).

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

**Definition 3.** 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. \( (R \cap P')L \supset (R \cap P)L \)).

We denote relative correctness (res. strict relative correctness) with respect to \( R \) by \( P \sqsubseteq_R P' \) (resp. \( P' \sqsupseteq_R P \)). To contrast relative correctness with correctness, we may refer to the latter as absolute correctness. Though we offer no proof, this definition does satisfy all the criteria set forth in section 2.3.

For an illustration of relative correctness, see Figure 2. Specification \( R \) is shown in the middle. To the left, we show two programs, \( Q \) and \( Q' \), such that \( Q' \) is more-correct than \( Q \) with respect to \( R \); to the right, we show two programs \( P \) and \( P' \) such that \( P' \) is more-correct than \( P \) with respect to \( R \); the competence domain of each program is indicated by the ovals. Notice that \( Q' \) is more-correct than \( Q \) by virtue of imitating the correct behavior of \( Q \), whereas \( P' \) is more-correct than \( P \) by virtue of a different correct behavior.

### 2.4 Definitions

For the sake of simplicity, we limit our discussions of relative correctness to deterministic programs, i.e. programs that map every initial state into no more than one final state.

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

The following Proposition sets the context for definition 3.
• An atomic change in program \( P \) is a pair of source code fragments \( (a, a') \) such that \( a \) is a syntactic atom in \( P \) and \( a' \) is a code fragment that we can substitute for \( a \) without violating the syntactic correctness of \( P \).

We use the term feature in a program \( P \) to refer to one or more syntactic atoms in \( P \); this concept is needed because some faults may involve more than one atom.

**Definition 4.** Given a program \( P \) on space \( S \) and a specification \( R \) on \( S \), a feature \( f \) of \( P \) is said to be a fault if \( f \) is strictly more-correct than \( P \) with respect to specification \( R \) if and only if there exists a substitute \( f' \) derived from \( f \) by atomic changes such that program \( P' \) obtained from \( P \) by replacing \( f \) with \( f' \) is strictly more-correct than \( P \) with respect to \( R \). The pair \((f, f')\) is said to be a fault repair of \( f \) in \( P \) with respect to \( R \).

Note that according to this definition, if \( f_1 \) and \( f_2 \) are faults in \( P \) with respect to \( R \), then (due to the transitivity of relative correctness) so is the aggregate \( \langle f_1, f_2 \rangle \). This motivates us to introduce the following definition.

**Definition 5.** A fault \( f \) in program \( P \) with respect to specification \( R \) is said to be an elementary fault if and only if it includes a single atom, or it includes more than one atom but no subset of the atoms is a fault. The number of atoms in an elementary fault is called the multiplicity of the fault.

### 2.6 Fault Density and Fault Depth

**Definition 6.** Given a program \( P \) and a specification \( R \) on space \( S \), the fault density of \( P \) with respect to \( R \) is the number of elementary faults in \( P \); the fault depth of \( P \) with respect to \( R \) is the minimal number of elementary repairs that separate \( P \) from a correct program.

Below are noteworthy remarks about these metrics:

- These two metrics are distinct; just because we have \( N \) elementary faults does not mean we need to perform \( N \) elementary fault repairs to obtain a correct program.
- If program \( P \) has \( N \) faults, say \( \{f_1, f_2, ..., f_N\} \) and we repair one of them, say \( f_1 \), to obtain a new program \( P' \), there is no reason to assume that \( \{f_2, ..., f_N\} \) are faults in \( P' \), since \( P \) and \( P' \) are different programs. Also, \( P' \) may have faults that were not faults in \( P \).
- These two metrics are subject to different rules. For example, if \( P' \) stems from \( P \) by an elementary fault repair then we have the equation:
  \[
  \text{depth}(P') \leq \text{depth}(P) + 1.
  \]
  Equality holds if \( P' \) is on a minimal path from \( P \) to a correct program. By contrast, fault density varies arbitrarily after an elementary fault repair.
- We find that fault depth is a more meaningful measure of faultiness than fault density.

### 3 A Generic Algorithm for Program Repair

Now that we have definitions of absolute correctness and relative correctness, we propose an algorithm of program repair whose main idea is: enhance relative correctness until you achieve absolute correctness. In order to put this idea into practice, we need to make some provisions:

- **Patch validation:** We must derive oracles that test candidate programs for absolute correctness, relative correctness, and strict relative correctness.
- **Patch Generation:** We must provide a mechanism to generate candidate programs; we assume the availability of a generator of atomic changes. Correctness enhancement is attempted with single atomic changes, then double atomic changes, then triple atomic changes, etc until it succeeds or a maximum multiplicity is reached.
- **Exception Handling:** We make provisions for the case where the patch generation is unable to enhance correctness within the range of allowed multiplicities.

#### 3.1 Absolute Correctness

An oracle takes the form of a binary predicate that refers to the initial state and the final state of the program, as in the following generic pattern:

\[
\text{oracle}(\langle Q, P \rangle) \Rightarrow \langle R \rangle
\]
We propose the following oracle for absolute correctness.

**Definition 7.** Given a specification $R$ on space $S$, the oracle of absolute correctness derived from $R$ is denoted by $\Omega(s,s')$ and defined by:

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

The following proposition asserts that this definition is sound; due to lack of space, we present it without proof.

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

Based on this proposition, we derive the following oracle:

```c
bool absoluteCorrectness(testData T) // 1
{statetype inits, s; bool relcor=true;// 2
 while (moretestdata(T)) // 3
 {inits=gettestdata(T); //load test datum 4
 s = inits;p(); // modifies s, not inits 5
 bool absoracle = absoracle(inits,s);// //6
 return absoracle}// 7
 bool absoracle(statetype s, sprime) // 8
 {return (!abscor||absoracle(inits,s));}// 9
 return relcor}// 10
```

Predicate $\text{absoracle}$ represents the absolute correctness of $P$ on $\text{inits}$ and predicate $\text{absoracle}(\text{inits},s)$ represents the absolute correctness of $P()$ on the same input. Line 9 represents the formula of $\omega()$ and line 8 cumulates this formula over set $T$.

### 3.3 Strict Relative Correctness

We are given a specification $R$ on space $S$ and a program $P$ on $S$. We consider a program $P'$ on $S$ and we want to write an oracle that checks whether a program $P'$ is strictly more-correct than $P$ with respect to $R$. $P'$ is strictly more-correct than $P$ if and only if it is more-correct than $P$ and there exists at least one input $s$ on which $P$ fails and $P'$ succeeds; whence the following formula.

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

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

The following proposition justifies this definition; due to lack of space, we present it without proof.

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

The test driver for strict relative correctness is written as:

```c
bool relativeCorrectness(testData T) // 1
{statetype inits, s; bool relcor=true;// 2
 while (moretestdata(T)) // 3
 {inits=gettestdata(T); //load test datum 4
 s = inits; p(); // modifies s, not inits 5
 bool absoracle = absoracle(inits,s);// //6
 return absoracle}// 7
 return relcor}// 10
```
The purpose of this section is to write code that performs a unitary increment of relative correctness; as we have seen in section 2.5, this amounts to repairing an elementary fault. To this effect, we assume that candidate patches are organized as a set of patch streams of increasing multiplicity, starting at 1, and not exceeding a user specified maximum, \( M \). For the sake of argument, we assume that candidate patches are organized as a set of patch streams of increasing multiplicities, which we name, respectively, \( PS(1) \), \( PS(2) \), \( \ldots \), \( PS(M) \). Each patch stream \( PS(m) \) is an ordered sequence, to which we may apply sequence operators \( \text{head()} \) and \( \text{tail()} \), referring respectively to the first element, and the remainder of the sequence. We assume that the patch generator puts at our disposal the following functions:

- \( \text{MorePatches}(P, m) \), a Boolean function that returns true if and only if there remains patches of \( P \) of multiplicity \( m \).
- \( \text{NextPatch}(P, m) \), which returns the next element of \( PS(m) \), for \( 1 \leq m \leq M \).

Using these functions, and assuming that specification \( R \) and test data \( T \) are global variables, we write the following code:

```c
void UnitIncCor(programType P, int M, //1 bool& inc, programType& Pp) //2 {
    // tries to return in Pp a corr. enhancement of P, using m atomic changes,
    // inc=true iff Pp more correct than P
    int m=1; inc=false; Pp=P; //6
    while (not inc && m<=M) //7
        { //increase correctness with m changes
            while (! smc(Pp,P) && MorePatches(P,m)) //9
                { //smc: Pp strictly more corr. than P
                    Pp = NextPatch(P, m); //11
                    if smc(Pp,P) {inc=true;} //12
                    else {m=m+1;} //higher multiplicity
                }
            // tries to return in Pp a corr. enhancement
            while (mortestdata())// 5
                {statetype inits, s;// 2
                    s = inits; Pprime(); //9
                    relcor = relcor && // 10
                        (!abscor || absoracle(inits,s)); //11
                    strict = strict || // 12
                        (!abscor && absoracle(inits,s)); //13
                    return (strict && relcor)\} //14
            Line 11 reflects the condition of relative correctness, and line 10 reflects the universal quantification over \( T \); line 13 reflects the condition of strict relative correctness, and line 12 reflects the existential quantification over \( T \). The oracles derived in the previous section give us means to recognize valid repairs, but do not give us means to generate valid repairs. Hence all we can do for now is to derive an algorithm that, given a function to generate plausible repair candidates, can select those that are indeed valid.

3.4 Elementary Fault Repair

The purpose of this section is to write code that performs a unitary increment of relative correctness; as we have seen in section 2.5, this amounts to repairing an elementary fault. To this effect, we assume the availability of a patch generator that applies a set of atomic change operators, and we seek to enhance correctness by attempting to identify faults of atomic changes to \( P \). We write, in turn, some sample specifications for partial correctness and termination properties of the generic algorithm with respect to specifications formulated as precondition/postcondition pairs.

4 Proof of Correctness

Using Propositions 2, 3 and 4, we resolve to prove partial correctness and termination properties of the generic algorithm with respect to specifications formulated as precondition/postcondition pairs.

4.1 Specifications

We write, in turn, some sample specifications for
We propose to prove the partial correctness of UnitIncCor(P,M,inc,Pp) which attempts to apply a unitary increment of correctness (whence the name UnitIncCor()) to P using at most M atomic changes, returning the result in Pp and signaling in inc whether it was successful.

ProgramRepair(P,R,T,M), which attempts to enhance the correctness of P with respect to R using unitary increments of correctness that involve M atomic changes or fewer.

For UnitIncCor(), we present the following sample of specifications, where we represent T as a set of specifications, where we represent T.

\[
\exists m: 1 \leq m \leq M : \exists Q \in PS(m) : Q \supseteq R.
\]

\[
\text{Postcondition: } P \supseteq R.
\]

- **Precondition:** true.
- **Postcondition:** P \(\supseteq\) R P.

If the patch generator can generate a strictly more-correct program with M atomic changes or fewer, then UnitIncCor() will enhance correctness.

\[
\exists m: 1 \leq m \leq M : \exists Q \in PS(m) : Q \supseteq R.
\]

\[
\text{Postcondition: } P \supseteq R.
\]

- **Precondition:** inc is true, then Pp is strictly more correct than P with respect to R.
- **Postcondition:** (inc \(\iff\) Pp \(\supseteq\) R).

For ProgramRepair(), we present the following sample:

- **Precondition:** true.
- **Postcondition:** P = P0.

- **Precondition:** P \(\supseteq\) R P0.
- **Postcondition:** P \(\supseteq\) R P0.

If the patch generator can generate an absolutely correct program with M atomic changes or fewer, then ProgramRepair() will return an absolutely correct program.

\[
\exists m: 1 \leq m \leq M : \exists Q \in PS(m) : Q \supseteq R'.
\]

\[
\text{Postcondition: } P \supseteq R'.
\]

### 4.2 Partial Correctness

We propose to prove the partial correctness of UnitIncCor() with respect to the second specification given above for it. This amounts to proving that the following Hoare formula is a theorem in Hoare's deductive logic (Hoare, 1969):

\[
v: \{\exists m: 1 \leq m \leq M : \exists Q \in PS(m) : Q \supseteq P\}
\]

m=1; inc=false; Pp=P;
while (! inc && m<=M)
{while (! smc(Pp,P) && MorePatches(P,m))
(Pp = NextPatch(P,m);)
if smc(Pp,P) (inc=true;)
else {m=m+1;}//try higher multiplicity
}

{Pp \(\supseteq\) P}.

Size limitations preclude us from including the detailed proof of this formula; hence we merely include the creative steps of this proof, i.e. the invariant assertions of, respectively, the outer loop then the inner loop; the interested reader may retrieve the whole proof from these. For the outer loop, we propose the following loop invariant \(\forall b\):

\[
\text{UnitIncCor(m) \(\land\) (inc \(\land\) Pp \(\supseteq\) R)}
\]

\[
\text{Postcondition: } P \supseteq R.
\]

\[
\text{Precondition: (}\exists h: m \leq h \leq M : \exists Q \in PS(h) : Q \supseteq R\text{)},
\]

where \(\text{UnitIncCor(m)}\) (stands for: in bounds) is shorthand for: 1 \(\leq\) m \(\leq\) M. For the inner while loop, we consider the following loop invariant, \(\forall b\):

\[
\text{UnitIncCor(m) \(\land\)}
\]

\[
\text{Postcondition: } P \supseteq R \(\forall\) (\exists h: m \leq h \leq M : \exists Q \in PS(h) : Q \supseteq R).
\]

### 4.3 Termination

A while loop of the form \(\{\text{while } t\} \{b\}\) on space S can be proved to terminate for precondition true if we can prove that the transitive closure of \((T \cap B)\) is well-founded, i.e. it admits no infinite chains. This, in turn, can be proved by finding a superset of the transitive closure of \((T \cap B)\) that is well founded. One way to do so is to find a variant function \(f\) on space S that satisfies the following condition:

\[
t(s) \land s \in dom(B) \Rightarrow f(B(s)) < f(s),
\]

for some well founded relation < on S.

In light of this, we find that function UnitIncCor() terminates for precondition true for the following reason: the inner loop can be proved to terminate using the function \(f() = length(PS(m))\), which decreases by 1 at each iteration and is bounded by 0. The outer loop can be proved to terminate using the function \(g() = \langle M - m, inc\rangle\), where each iteration either increases m (hence decreases \(M - m\)) or preserves m but decreases inc (if we consider true < false).

As for function ProgramRepair(), we have no assurance that it terminates in general, but we do have a simple sufficient condition: if \(\text{dom}(R)\) is finite then each iteration of the loop of ProgramRepair() reduces the function

\[
f() = \text{dom}(R) \setminus \text{dom}(R \cap P),
\]

by at least one element, and is bounded by the empty set.
5 CRCFix: An Instance of the Generic Algorithm

In addition to proving the correctness of the generic algorithm, we resolve to also test it, by creating an instance thereof using the patch generation function of an existing program repair tool. To this effect, we choose Cardumen (Martinez and Monperrus, 2018b), which is an element of the Astor framework of Martinez and Monperrus (Martinez and Monperrus, 2018a); we refer to the resulting tool as CRCFix (pronounced: C R C Fix). The Astor framework gives us the opportunity to integrate our algorithm with Cardumen functionality in a fairly modular fashion. The CRCFix tool can be accessed at (Anonymous, 2020b) and the data presented in this section can be accessed at (Anonymous, 2020a).

5.1 The Program and Specification

The program on which we ran our experiments is a binary search program, whose space is defined by an array \( a \), a variable \( x \) of the same data type as the array, and an index \( k \) into the array. The specification is:

\[
R = \{(s,s′)|\text{sorted}(a) \land a[k] = x\}.
\]

The domain of this relation is:

\[
\text{dom}(R) = \{s|\text{sorted}(a) \land x \in a\}.
\]

\( R \) and \( \text{dom}(R) \) are represented by, respectively, a binary predicate and unary predicate, which are then used to derive oracles as per definitions 7, 8 and 9. The program, due to (Lin et al., 2017), is written as:

```c
int lo=0; int hi=a.length; int k=-1; //1
while (lo<hi) //2
  {int mid=(lo+hi)/2; //3
   if (x==a[mid]&&(mid==0||x!=a[mid-1]))//4
     {k=mid; break;} //5
   else //6
     if (x>a[mid]) {hi=mid;} //7
     else {lo=mid+1;} //8
return k;} //9
```

Now that we have a definition of faults (Definition 4), we are reluctant to use the term fault if we are not sure that a program feature meets our definition; without meaning to be hair-splitting we refer to source code modifications merely as modifications; for all we know, a modification to source code may very well have no impact on the function of the program, or may be part of a multi-site fault. We seed this program with a number of modifications, then we run CRCFix on it, using the oracles derived from \( R \), and compare the outcome to that of running the original Cardumen tool. All the modifications we seed are taken from the QuixBugs benchmark (Lin et al., 2017), although we seed more than one fault at a time, and we use \( R \) as the specification (rather than the deterministic behavior of the program prior to the modifications). We use the test data provided in the benchmark, and we show, for each experiment, the passing test data and the failing test data. Interestingly, the number of passing and failing test data is not the same for CRCFix and Cardumen, because CRCFix uses specification \( R \) to judge correctness, whereas Cardumen uses the behavior of the program prior to seeding the modifications (remember the distinction between the criterion of preserving correctness, adopted by CRCFix and the criterion of preserving correct behavior, adopted by Cardumen: see Figure 2). In all these experiments (except the last), we limit Cardumen to 500 generations and a run-time of 120 minutes (in the last, the number of generations is limited to 700).

5.2 Two Modifications

In this example we make two modifications to the source code:

- Mod 1: Change \( k=\text{mid} \) in line 5 to \( k=\text{mid}-1 \).
- Mod 2: Change \( \text{hi}=\text{mid} \) in line 7 to \( \text{hi}=\text{mid}-1 \).

The outcome of the executions of CRCFix and Cardumen is given in Figure 3. Of course, the fault repairs that CRCFix has done are not what we would want, but we must remember: CRCFix is not using \( R \) as a target specification, rather it is using \( R′ \), which is the pre-restriction of \( R \) to the test data set \( T \). So that the repaired program is guaranteed to be absolutely correct with respect to \( R′ \subset R \), but not necessarily with respect to \( R \). With a small test data set \( T \) of size 7, there is much scope for loss of precision, which is why we argue for large data sets, including large sets of failing tests.

Since Cardumen applies single modifications to the original program, we did not really expect that it would be able to repair this program; yet this experiment shows that if it were seeking to achieve relative correctness rather than absolute correctness, then Cardumen would have enhanced the relative correctness of the program. Also, if it were applied repeatedly (which is what CRCFix does), then it would have achieved absolute correctness with respect to \( R′ \) (which is what CRCFix did). So that Cardumen is good at generating valid patches, but fails to recognize them as valid, because it is testing for absolute correctness.

5.3 Preserving Correctness

The purpose of this example is to illustrate the difference between preserving correct behavior (which is
what most program repair methods aim to achieve), and merely Preserving correctness (which is what CRCFix aims to achieve). When we take a non-deterministic specification as a reference (which is the case for $R$), then we can preserve correctness without preserving correct behavior, since correct behavior is not unique; see Figure 2, where $Q$ preserves correct behavior whereas $P$ merely preserves correctness. In this example we make two modifications to the source code:

- **Mod 1**: Change $\text{mid}=(\text{lo}+\text{hi})/2$ in line 3 to $\text{mid}=\text{h}-1$.
- **Mod 2**: Change $\text{hi}=$mid in line 7 to $\text{hi}=$mid-1.

The outcome of the executions of CRCFix and Cardumen is given in Figure 4. Again, we may not like the resulting program, but CRCFix only guarantees correctness with respect to $R'$, not with respect to $R$; if we want to increase the likelihood of absolute correctness with respect to $R$, we need to make $T$ larger/ more diverse. As for why CRCFix worked but Cardumen did not, we have a simple plausible explanation: The new program (let’s call it $P'$) is more correct than $P$ with respect to $R'$, but it does not duplicate the correct behavior of $P$. Indeed, specification $R$ (and $R'$) dictate the final value of $k$, but do not specify the final values of the other variables ($\text{lo}$, $\text{hi}$, mid). Hence CRCFix selected $P'$ because $P'$ returns the proper value for $k$, but Cardumen rejected $P'$ because Cardumen only selects the programs that duplicate the behavior of $P$ on all the variables ($k$, but also $\text{lo}$, $\text{hi}$, mid). So Cardumen maya well have encountered the repair candidate retrieved by CRCFix, but excluded it because it does not preserve the correct behavior of $P$.

The following example is an instance of the same situation, where the non-determinism of $R$ enables CRCFix to find a repair while Cardumen fails to find a repair because it is looking for a repair that duplicates the correct behavior of the original program. In this example we make two modifications to the text of the program:

- **Mod 1**: Change $k=$mid in line 5 to $k=$mid-1.
- **Mod 2**: Change $\text{return }k$ in line 7 to $\text{return }k-1$.

The outcome of the executions of CRCFix and Cardumen is given in Figure 5.

### 5.4 Large Data Set

One of the arguments we make in this paper is that we ought to generate fewer repair candidates, but test them more thoroughly. In this example we show how increasing the size and diversity of the test data set $T$ enhances the precision of the search, and in fact produces a solution that is absolutely correct with respect to $R$, in addition to being (provably) absolutely correct with respect to $R'$. In this example we make one modification to the text of the program:

- **Mod 1**: Change $x<=a[\text{mid}]$ in line 7 to $x<=a[\text{mid}-1]$.

The outcome of the executions of CRCFix and Cardumen is given in Figure 6. Both Cardumen and CRCFix have found patches that are absolutely correct with respect to $R'$, though CRCFix did so faster and with fewer generations. Interestingly, when we increase the size of $T$ from 7 to 100, generating additional test data, CRCFix converges faster, with a few fewer generations, and produces a better solution; the solution it produces is in fact absolutely correct with respect to $R$! Cardumen produced 700 generations and aborted (note: Cardumen does not test all the candidate repairs it generates, it tests only a randomly selected subset). See Figure 7.

### 6 Concluding Remarks

The thesis of this paper can be summed up in the following premises:

- To repair a program does not mean to make it absolutely correct; it only means to make it more-correct than it was originally.
<table>
<thead>
<tr>
<th>Patch</th>
<th>CRCFix</th>
<th>Cardumen</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pass. tests</td>
<td>5</td>
<td>3</td>
</tr>
<tr>
<td>Fail. tests</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>Patches</td>
<td>1 Iteration of UnitIncCor Mod 2: mid → hi-1 Aborted, Max Generations</td>
<td></td>
</tr>
<tr>
<td># Generations</td>
<td>25</td>
<td>500</td>
</tr>
<tr>
<td>Time (s)</td>
<td>121</td>
<td>1338.6</td>
</tr>
</tbody>
</table>

Figure 4: Preserving Correctness

<table>
<thead>
<tr>
<th>Patch</th>
<th>CRCFix</th>
<th>Cardumen</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pass. tests</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>Fail. tests</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>Patches</td>
<td>3 Iterations of UnitIncCor Mod 1: mid → mid+mid Mod 1: mid+mid → (hi+hi)/2 Aborted, Max Generations Mod 2: (hi+hi)/2 → mid+1</td>
<td></td>
</tr>
<tr>
<td># Generations</td>
<td>3+3+159=196</td>
<td>500</td>
</tr>
<tr>
<td>Time (s)</td>
<td>10+90+542=642</td>
<td>1384.2</td>
</tr>
</tbody>
</table>

Figure 5: Non Determinacy

<table>
<thead>
<tr>
<th>Patch</th>
<th>CRCFix</th>
<th>Cardumen</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pass. tests</td>
<td>6</td>
<td>5</td>
</tr>
<tr>
<td>Fail. tests</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>Patches</td>
<td>1 Iteration of UnitIncCor Mod 1: (hi+lo)/2 → k+hi Mod 2: x&lt;=a[(mid-1), x&gt;a[ mid-1], (x&lt;hi), (x&lt;hi), (x&lt;hi), (x&lt;hi), (x&lt;hi), (x&lt;hi) Aborted, Max Generations</td>
<td></td>
</tr>
<tr>
<td># Generations</td>
<td>195</td>
<td>484</td>
</tr>
<tr>
<td>Time (s)</td>
<td>747</td>
<td>1400</td>
</tr>
</tbody>
</table>

Figure 6: Large Data Set

<table>
<thead>
<tr>
<th>Patch</th>
<th>CRCFix</th>
<th>Cardumen</th>
</tr>
</thead>
<tbody>
<tr>
<td>Passing Tests</td>
<td>92</td>
<td>68</td>
</tr>
<tr>
<td>Failing Tests</td>
<td>8</td>
<td>32</td>
</tr>
<tr>
<td>Patch(es)</td>
<td>1 Iteration of UnitIncCor (x&lt;=a[(lo+hi)/2]) Aborted, Max Generations</td>
<td></td>
</tr>
<tr>
<td># Generations</td>
<td>19</td>
<td>700</td>
</tr>
<tr>
<td>Time (s)</td>
<td>82</td>
<td>2759.6</td>
</tr>
</tbody>
</table>

Figure 7: Larger Data Set
• Consequently, the study of program repair ought to be based on concepts of relative correctness.
• Relative correctness ought to play for program repair the role that absolute correctness plays for program derivation.

These premises have practical implications for the discipline of program repair:
• It takes significantly less patch generation to find a relatively correct program (over a program \( P \) with respect to a specification \( R \)) than it takes to find an absolutely correct program (with respect to \( R \)).
• Relative correctness enables us to repair programs with multiple faults (re: section 5.2).
• Relative correctness enables us to distinguish between repairing one fault at a time, versus remediating one failure at a time; and shows that the former policy is superior to the latter policy (re: section 5.2).
• Relative correctness enables us to distinguish between preserving correctness and preserving correct behavior; and shows that the former policy is superior (re: section 5.3).
• We further show, in section 5.4, that using a large test data set can yield gains in terms of execution time, (smaller) number of repair candidates, and quality of the repair; in particular, it makes it less likely that a repair candidate is correct with respect to \( R' \) but not with respect to \( R \).
• Perhaps most important of all, relative correctness provides us with the ontology and the axiomatization that we need to formulate specifications of program repair algorithms, and prove their correctness using Hoare logic. As far as we know, no other method or tool has been proven correct.

We feel that the discipline of program repair has achieved great strides in terms of patch generation, and that if adopted, relative correctness can put all this patch generation power on steroids to produce methods and tools that are both more effective and more efficient. Also, we argue that a theory of relative correctness will enable researchers to formulate and prove specific claims about program repair methods and tools, rather than merely judge them by their performance on shared benchmarks.

Threats to validity of our approach include the usual scalability and applicability concerns that arise with any attempt to introduce formal methods into a successful engineering discipline; we argue that relative correctness scales as much as (or as little as) absolute correctness, and that it may play for program repair the role that absolute correctness plays for program derivation. While we use a specific definition of relative correctness in this paper, we do not claim that it is the only possible definition, nor the best definition.

Assuming that one agrees with our basic premises, we argue that anyone who proposes a method of program repair ought to tell us what definition of relative correctness they adopt, and in what way their method does enhance relative correctness as defined.

REFERENCES


