Program Repair at Arbitrary Fault Depth

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

Matias Martinez
University of Valenciennes
Valenciennes, France
matias.sebastian.martinez@gmail.com

Ali Mili
New Jersey Institute of Technology
Newark, NJ, USA
mili@njit.edu

***Abstract*—Program repair has been an active research area for over a decade and has achieved great strides in terms of scalable automated repair tools. In this paper we argue that existing program repair tools lack an important ingredient, which limits their scope and their efficiency: a formal definition of a fault, and a formal characterization of fault removal. To support our conjecture, we consider GenProg, an archetypical program repair tool, and modify it according to our definitions of fault and fault removal; then we show, by means of empirical experiments, the impact that this has on the effectiveness and efficiency of the tool.**

I. Introduction: The Missing Ingredient

A. Program Repair without Relative Correctness

As a field of research and development, the discipline of program repair has made great strides over the past decade, producing a continuous stream of increasingly effective, scalable tools [2, 4, 5, 7, 11, 12, 16–18, 21, 23–28]. In this paper we argue that a key ingredient has been missing in the study of program repair so far; and we illustrate, through a limited empirical study, how integration of this ingredient into a program repair tool brings about great enhancements in its effectiveness (through broader scope) and efficiency (through shorter response time).

The missing ingredient in question is the concept of *relative correctness*, i.e. the property of a program to be more-correct than another with respect to a specification [9]. Whereas traditional (absolute) correctness distinguishes between only two classes of candidate programs for a given specification, namely *correct* programs and *incorrect programs*, relative correctness ranks candidate programs on a partial ordering whose maximal elements are the (absolutely) correct programs. We argue that repairing a program means making it more-correct, but does not necessarily mean making it absolutely correct; as a repaired program may itself still be incorrect. Hence in the same way that absolute correctness is the criterion by which we judge the derivation of a program from a specification, relative correctness ought to be the criterion by which we judge the process of repairing a program with respect to a specification. Consequently, we argue that relative correctness ought to be an integral part of the study of program repair; in section I-B, we discuss the insights and advances that relative correctness may contribute to the practice of program repair; in particular, we use relative correctness to define faults and fault removals.

This works is partly supported by the US National Science Foundation through grant DGE 1565478.

In order to illustrate the usefulness of relative correctness in practice, we consider *Astor*, a publicly available program repair library for Java, which adapts to Java and integrates several pre-existing program repair packages, including jGenProg, jKali and jMutRepair [19]; Astor offers researchers ways to modify these repair approaches in a modular fashion through a set of extension points. Taking advantage of this capability, we consider jGenProg, we re-write its validation step to reflect ideas of relative correctness, and we test the performance of the new tool by comparison with the original version. Our empirical observations support our claim that the improved validation process enhances the effectiveness of jGenProg by enabling it to repair multiple faults in benchmark programs by tackling them in sequence, one a time; the original version of jGenProg (as well as most program repair tools, to the best of our knowledge) can only repair programs that have a single fault. The improved validation process also enhances the efficiency of jGenProg, by enabling it to repair several faults in significantly less time than it takes the original jGenProg to repair the first fault.

In section I-B we discuss the motivation for integrating relative correctness in the study of program repair by highlighting the issues that this concept may enable us to elucidate. If program repair is the art of locating and removing faults then defining faults and related concepts ought to be of some importance; in section II we use the concept of relative correctness to define faults, fault removals, fault density and fault depth. Once we understand the contrast between relative correctness and absolute correctness, it is easy to imagine that program repair algorithms ought to be variations of the following theme: enhance relative correctness until we achieve absolute correctness; in section III we discuss a generic program repair algorithm based on this broad theme. In section IV we discuss how we instantiate this generic algorithm by combining the patch generation function of jGenProg with an original patch validation function derived from oracles based on absolute and relative correctness. In section V we present the empirical results we obtain with the tool produced in section IV on routine benchmark programs and faults, and compare our results with those obtained by applying the original jGenProg tool to the same data.

B. Program Repair as Correctness Enhancement

Given that the discipline of program repair has achieved great strides in the past decade, it is legitimate to raise the question of why we need a theory of this discipline (since we are doing very well without it). In this section, we briefly discuss some of the insights and practical advances that we envision to gain from such a theory. For the sake of argument, we limit our discussion in this section (and this paper) to program repair methods that follow the generate-and-validate principle, even though we feel that relative correctness is relevant to other methods as well.

- *Expanding the Scope*. In the absence of a definition of relative correctness, program repair methods test repair candidates for absolute correctness. This limits their scope to programs that are within striking distance of absolute correctness; in other words, they can only be applied to repair the last fault of a program (or a sufficiently small set of remaining faults to be combinatorially tractable), which is clearly a severe limitation. It is useful to expand the scope of program repair methods to more realistic circumstances, where the repaired program, while being more-correct than the original, still falls short of being absolutely correct.
- *Program Repair as Correctness Enhancement*. Using the definition of relative correctness, we equate program repair (aka *fault removal*) with any transformation of the program's source code that makes the program more-correct.
- *Focusing on Faults*. There is a crucial difference between *fault removal* and *failure remediation* in a program. Most program repair methods operate by remedying a failure, in the following sense: upon observing a failure in the program, they resolve to generate variants of this program and test them in turn until they find one that remedies the observed failure while preserveing the correct behavior of the program. The trouble with this approach is that, when we initiate the search for a successful variant, we have no idea how many faults are conspiring to cause the observed failure; if this number, call it N, is large, then we need to find a combination of N simultaneous fault removals that remedy the observed failure; this, clearly, is a receipe for unbounded combinatorial explosion. We argue that it is more judicious to let fault removal, rather than failure remediation, drive the effort: we consider that the program is repaired whenever we make it incrementally more-correct; if we keep enhancing its correctness (by removing more and more faults) then eventually whatever failure we have observed will be remedied.
- *Preserving Correctness vs Preserving Correct Behavior*. Many program repair methods perform patch validation by checking that candidate repairs rectify the incorrect behavior of the base program while preserving its correct behavior. But most specifications we encounter in practice are (vastly) non-deterministic, hence correct behavior is not unique, and it is possible for a candidate repair to be correct without imitating the correct behavior of the base program. Testing for the preservation of correct behavior leads to a loss of recall: patch generation may well produce a valid repair, but patch validation fails to recognize it as such.
- *Approximating Absolute Correctness*. In the absence of a definition of relative correctness, program repair methods resort to approximations of absolute correctness, including *fitness functions*, which reward candidate repairs for correct behavior and penalize them for incorrect behavior. We find that such functions are approximations of program reliability, and their use leads to a loss of precision: they may select repair candidates that are not actually more-correct than the base program.

II. What is a Fault, and Why does it Matter?

Given that program repair is the process of diagnosing and removing faults from programs, it is sensible to expect that a definition of faults may give us some insights into this process. In this section, we revisit the definition of faults given in [10, 22] and present its main properties; for the sake of readability, we do not dwell on theoretical details, but keep the focus of our discussions at an intuitive level; the interested reader may refer to [10, 22] for further details.

A. Relative Correctness

The *space* (aka *state space*) of a program (usually denoted by S) is the set of values that its variables may take; For example, if the program has two variables x and y of type integers, then its space is the set of pairs of integers; a *state* (usually denoted by lower case s) of a program is an element of its space. Relative correctness, like traditional (absolute) correctness [13], is defined with respect to a specification; for the purposes of our discussion, specifications are represented by binary relations on a state space; also, programs are represented by functions that map initial states onto final states. Following [10, 22], we let the *competence domain* of a program P with respect to a specification R be the set of initial states that P maps onto correct (with respect to R) final states. As a very simple example, let space S, specification R and program P be defined as follows:

$S = \{0, 1, 2, 3, 4, 5, 6\}$.
$R = \{(1, 0), (1, 2), (2, 1), (2, 3), (3, 2), (3, 4), (4, 3), (4, 5), (5, 4)\}$.
$P = \{(0, 1), (1, 2), (2, 3), (3, 4), (4, 5), (5, 6)\}$.

The competence domain of P is:

$CD = \{1, 2, 3, 4\}$

since these are the initial values for which P behaves as indicated by R. See Figure 1, where the competence domain of P is highlighted with the oval.

Given a specification R and two candidate programs P and P', we let CD and CD' be the competence domains of (resp.) P and P' with respect to R, and we say that P' is *more-correct* than P with respect to R if and only if $CD \subseteq CD'$; also, we say that P' is *strictly more-correct* than P with respect to R if and only if $CD \subset CD'$.

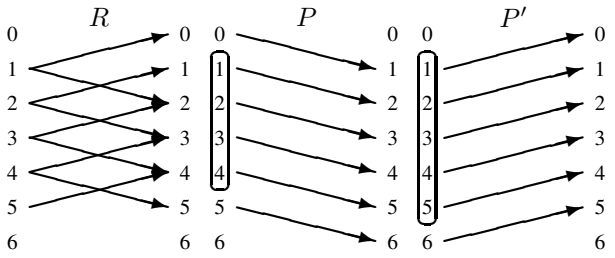

Fig. 1. P' is more-correct than P with respect to R

See Figure 1; the competence domains of P and P', which are highlighted in the figure, are, respectively, $CD = \{1, 2, 3, 4\}$ and $CD' = \{1, 2, 3, 4, 5\}$. Note that even though P' is more-correct than P, it does not mimic the correct behavior of P; rather it has a different correct behavior.

To illustrate in what way relative correctness defines a partial ordering among candidate programs, we consider the state space S defined by two integer variables x and y, and we let R be the following specification:

$$R = \{(\langle x, y\rangle, \langle x', y'\rangle) | x^2 \leq x'y' \leq 2x^2\}.$$

We consider the following candidate programs, denoted P_0 through P_7. Next to each program P_i, we represent its competence domain (CD_i); we leave it to the reader to check that for each P_i, the associated competence domain CD_i is indeed the set of inputs for which the program produces outputs that satisfy the specification. We do not show how competence domains are computed; interested readers may look at [10] for details. 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. Programs P_5, P_6 and P_7 are absolutely correct with respect to R; programs P_0... P_4 are incorrect, but there are degrees of (in)correctness, as depicted in Figure 2.

P_0: {x=1; y=-1;}. $CD_0 = \{\}$.
P_1: {x=2*x; y=0;}. $CD_1 = \{s | x = 0\}$.
P_2: {x=x*x; y=0;}. $CD_2 = \{s | x = 0\}$.
P_3: {x=2*x; y=1;}. $CD_3 = \{s | 0 \leq x \leq 2\}$.
P_4: {x=2*x; y=2;}. $CD_4 = \{s | x = 0 \vee 2 \leq x \leq 4\}$.
P_5: {x=2*x; y=x/2;}. $CD_5 = S$.
P_6: {y=x/2; x=2*x;}. $CD_6 = S$.
P_7: {x=x*x; y=2;}. $CD_7 = S$.

B. Faults and Fault Removals

As we attempt to define faults, we must acknowledge that any definition of fault must refer to a level of granularity at which we want to isolate faults. At an extreme scale, we may consider the whole program as the level of granularity, which would mean that we either have no fault (if the program is absolutely correct) or one fault (if it is incorrect); but that is clearly unhelpful, as we want to define faults at a scale that enables us to identify the parts of the source code that require our attention. Typical levels of granularity include the line of code, the elementary statement, the expression, the operator, or the operand, etc. We use the term *syntactic feature* (or: *feature*) to refer to a syntactic element of the source code of P, at the appropriate level of granularity; for the sake of generality, we do not mandate that a feature be contiguous (so a feature may, e.g. involve two statements at two distinct locations in the source code).

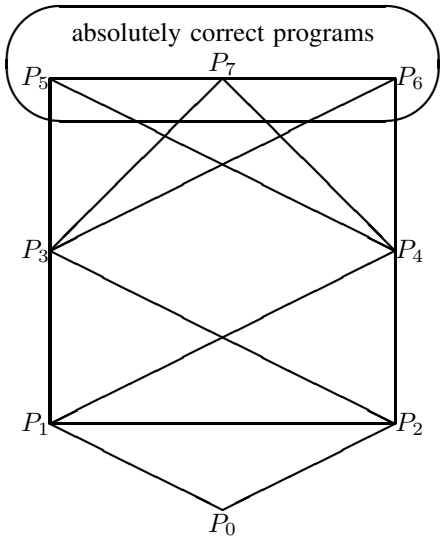

Fig. 2. Ordering Candidate Programs by Relative Correctness

Given a specification R, a program P, and a feature f in P, we say that f is a *fault* in P with respect to R if and only if there exists a feature f' such that the program P' obtained from P by replacing f by f' is strictly more-correct than P with respect to R. The pair (f, f') is referred to as a *fault removal* of P with respect to R.

Of course, all the substitutions we talk about in this paper are assumed to produce a syntactically correct program. An unspoken assumption of these definitions is that f and f' are, broadly speaking, of the same level of granularity (if f is an assignment statement one would not expect f' to be a 5 KLOC block of code); it is impossible to integrate this condition in the definition, but it is sensible to assume it.

C. Elementary Faults

We consider a specification R and a program P, and we let f_1 and f_2 be two features in P; let f_1' and f_2' be two features such that if we replace f_1 by f_1' and f_2 by f_2' we obtain a program P' that is strictly more-correct than P with respect to R. By the definition above, the aggregate $\langle f_1, f_2\rangle$ is a fault in P with respect to R. The question that arises in this case is: do we have a single fault, which happens to span multiple sites, or do we have two single-site faults? The answer to this question depends on whether both substitutions are needed to produce a strictly more-correct program, or one substitution is sufficient. Whence the following definition: any single-site fault is said to be an *elementary fault*; a multi-site fault is said to be an *elementary fault* if and only if no part of it is a fault.

Why is this concept important? Because if we are going to talk about faults in the plural, a good way to start is to tell the difference between one fault and two faults. If we cannot

tell the difference between one fault and two faults, we cannot possibly assign any meaning to *three faults*, *four faults*, etc. Still, we discuss in the next section why counting faults is not as meaningful in fact as counting fault removals.

D. Fault Density and Fault Depth

Given a specification R and a program P, we use the term *fault density* to refer to the number of elementary faults in P with respect to R. A first observation about this concept is that it is not a meaningful measure of program faultiness: If P has, say two (elementary) faults with respect to R, e.g. f_1 and f_2, and we find a substitute for f_1 (say, f_1') to produce a program P' that is strictly more-correct than P with respect to R, we have no reason to believe that f_2 will be a fault in P'; conversely, a feature f_3 that is not a fault in P may well become a fault in P' if it was masked by f_1 and is made observable by the repair; hence removing a fault in a program does not necessarily reduce the fault density by one; it may reduce it by more than one (if the removal of one fault cancels another fault), but may also increase it (if the removed fault exposes other faults that were heretofore masked).

Whence the following definition: The *fault depth* of a program P with respect to a specification R is the minimal number of elementary fault removals (i.e. fault removals of elementary faults) that separate P from a (absolutely) correct program.

A simple way to discuss the contrast between fault density and fault depth is to consider an example: we have taken the `tcas` component of the Siemens Benchmark [3], applied eight modifications to its source code (provided in the benchmark), then attempted to repair the resulting program by iterative fault removals. Fault removals are performed by generating mutants of the current program until we find a mutant that is strictly more-correct; this process is repeated until we reach a mutant that is absolutely correct or all maximal nodes admit no mutants that are strictly more-correct. The mutant generator that we used to this effect is Proteum, due to [6]; this mutant is fairly exhaustive in terms of the range of mutation operators that it applies.

The result is shown in the graph of Figure 3; each node represents a mutant of `tcas` (obtained after any number of mutations); and each arc represents a strict relative correctness relationship (the upper node is strictly more-correct than the lower node). Because each pair of nodes linked by an arc results from a mutation (the upper node is obtained by applying a mutation operator to the lower node), in fact each arc represents an elementary fault removal. The bottom of the graph represents the faulty version of `tcas`, whereas the top represents a correct version of `tcas`.

If we assume, for the sake of argument, that the mutant generator is exhaustive (in the sense that if a strictly more-correct mutant exists, the mutant generator will find it) and that the test data we use is adequate (i.e. whatever property we observe on test data T holds for all input data) then:

- *Fault Density*. The fault density of each node of the graph is equal to the outgoing degree of the node, since each outgoing arc represents a fault removal.
- *Fault Depth*. The fault depth of each node of the graph is equal to the distance from the node to the top of the graph (i.e. the number of arcs in the shortest path from the node to the top).

As we can see on this graph, the fault depth decreases by one with each fault removal (this is true in general as long as the fault removal is on a shortest path to a correct program). By contrast, the fault density evolves erratically as faults are removed: in some instances it does decrease by one; in several instances it remains unchanged; there are cases where it actually increases after a fault removal. Generally speaking, fault depth is a better measure of faultiness than fault density: Note that the node at the bottom of the graph, which represents the original version of `tcas` in which we have seeded eight modifications, shows only four faults, yet is at a fault depth of seven (we need seven fault removals to obtain a correct version).

III. A Generic Algorithm for Program Repair

Using the ideas discussed so far, we propose a generic program repair algorithm (whose original version is due to [15]), which has the following properties:

- It is generic in the sense that it outlines a general process for selecting repair candidates, but does not specify how repair candidates are generated; hence it can be instantiated for any given patch generation method.
- The algorithm basically says: enhance relative correctness until you achieve absolute correctness.
- Because we do not specify patch generation, we have no way to ensure that the algorithm will always achieve absolute correctness; hence in fact the algorithm iterates until either it achieves absolute correctness or it determines that it can no longer enhance relative correctness (due to inadequate patch generation).
- The algorithm applies to programs of arbitrary fault depth, because it does not test for absolute correctness, but rather tests for relative correctness over the base program.
- It is based on an elementary routine that performs a unitary increment of correctness enhancement; this routine removes one elementary fault at a time. Because elementary faults may be multi-site, it attempts to enhance correctness by single-site features, then double-site features, etc, until it succeeds or reaches a user-supplied threshold.
- The algorithm requires on input:
  - The specification R with respect to which correctness is judged. This takes the form of a Boolean function between initial states and final states.
  - The faulty program, P.
  - The test data T that we use to test for absolute correctness and relative correctness.

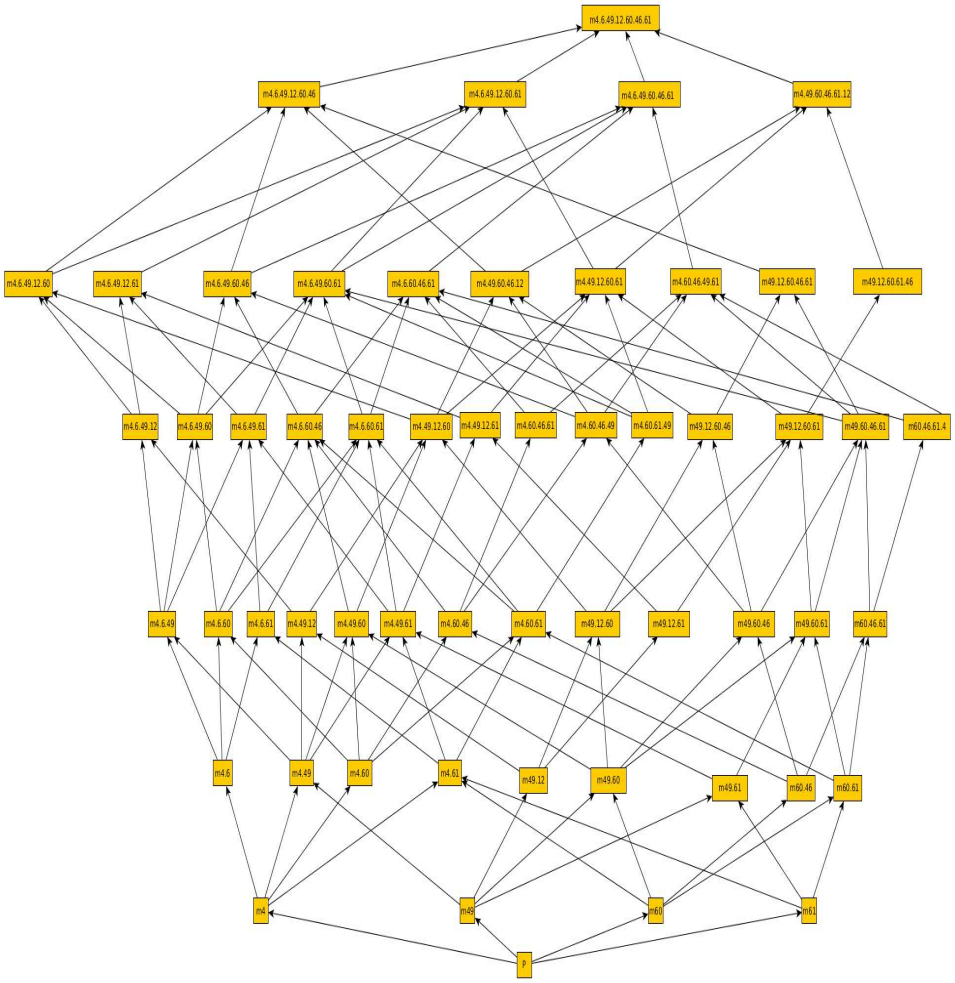

Fig. 3. Stepwise Repair of `tcas` Faults

- The threshold of multiplicity ($MaxM$) that we are willing to consider for multi-site elementary faults; if we restrict our search to single-site faults, then we let $MaxM$ be 1. It is seldom that we need to set $MaxM$ much greater than 2.
- While in theory the domain of R can be inferred from R, this is difficult in practice; hence in addition to providing R (as we discuss above) we also mandate providing the domain of R in the form of a Boolean function on initial states.

The outline of our program repair algorithm can be written as:

```
void ProgramRepair(programa P, specification R,
                   testdata T, int MaxM)
   {bool incremented=true;
    while (incremented && not abscor(P))
       {P = UnitIncCor(P, R, MaxM)}}
```

where `UnitIncCor(P, R, MaxM)` takes a program P and attempts to perform a unitary increment of relative correctness on it to return a program P' that is strictly more-correct than P with respect to R, and stems from P by the removal of an elementary fault whose multiplicity does not exceed $MaxM$. The algorithm `ProgramRepair(P,R,T,MaxM)` aims to enhance the (relative) correctness of program P with respect to specification R as far as the (implicit) patch generation allows by repeatedly performing unitary increments of correctness of multiplicity less than or equal $MaxM$ until we reach an absolutely correct program or fail to further enhance relative correctness; test data T is used to determine absolute correctness and relative correctness of candidate programs. The oracle `abscor(P)` tests P for absolute correctness on test data T with respect to R, and variable `incremented` is a Boolean variable that remains true as long as `ProgramRepair` keeps enhancing the correctness of the program through successive calls to `UnitIncCor()`, whose algorithm follows:

```
programtype UnitIncCor(programtype P,          //1
                specification R, int MaxM)//2
   {int mult=1; incremented = false;           //3
    while (not incremented && mult <= MaxM)    //4
       {programtype Pp=P; initPatches(mult);//5
        while (not smc(Pp,P) &&                //6
               MorePatches(P,mult))            //7
           {Pp = NextPatch(P, mult);}          //8
        if smc(Pp,P)                           //9
           {incremented = true; return Pp;}//10
        else                                   //11
           {mult = mult+1;}}}                  //12
```

We assume that patch generation avails us of three functions, namely:

- A Boolean function `MorePatches(P,mult)` which returns true if and only if more patches of P are available, stemming from P by modifying `mult` sites across P.
- A function `NextPatch(P,mult)`, which returns the next patch of P of multiplicity `mult` (in an order determined by the patch generator).
- A function `InitPatches(mult)`, which initializes the generation of patches of multiplicity `mult`.

In line (3) we initializse the correctness enhancement process by setting `mult` to 1 (i.e. we want to enhance the correctness of P by single-site fault removals, if we can) and setting `incremented` to false (we have not enhanced correctness yet). The loop in lines 5 through 8 invokes the patch generator as long as we have not yet found a patch Pp of P that is strictly more-correct than P and there are more patches to review. If, upon exiting the loop, we find thet Pp is strictly more-correct than P then we terminate the search and return Pp, else we increment the multiplicity of patches by 1 and try again, until we hit $MaxM$.

IV. Adapting jGenProg

In order to instantiate the generic algorithm depicted above, we propose to combine it with an existing patch generation tool. To this effect, we consider *Astor* [19, 20], a program repair framework that contains five state-of-the-art program repair approaches implemented in Java, and provides extension points for:

- Overriding the default behavior of the implementations that are already included.
- Creating new repair approaches by defining and combining new components (e.g. new repair operators, new search space navigation strategies, new patch validation techniques).

The five program repair methods that are aggregated in Astor are all implementations of the generate-and-validate repair strategy, and include: jGenProg, jKali, jMutataion, DeepRepair, and Cardumen. Astor provides twelve extension points that form the design space of program repair; novel repair approaches can be implemented by choosing an original component for each extension point. Among the main design decisions that Astor enables one to make, we cite:

- The repair operators that can be used to define the search space.
- The strategies that can be deployed to navigate the search space.
- The criteria that can be used to select candidate repairs.

For the purposes of our research, we choose to adapt jGenProg; to this effect, we adopt the following features of jGenProg:

- *Fault Localisation*: Astor/ jGenProg uses an existing fault localisation technique called GZoltgar [1].
- *Scale*: jGenProg analyzes programs at the statement level.
- *Navigation*: Whereas jGenProg uses the evolutionary optimization technique of search space navigation to prune the search space, we adopt the exhaustive navigation approach, so as to preserve recall. Candidate patches are generated by considering modification points (features) and applying to them all relevant repair operators from the relevant repair operator space. The number of modification points is determined by the user according to the suspected fault depth of the program; the modification points are ranked according to the weight assigned by the fault localisation step.
- *Patch Generation*: we adopt three types of code transformation, namely: removing statements, inserting statements, and replacing statements.
- *Patch Validation*: Astor provides an extension point named *EP PV*, to specify the patch validation process. Whereas Astor performs patch validation by testing candidate patches for correct execution on the test suite, we override this process using the oracles of absolute correctness and strict relative correctness referenced in section III. If the patch is found to be absolutely correct (on the test data provided) then the search concludes and the patch is returned as a valid repair. Else if the patch is found to be strictly more-correct than the base program then the patch is taken as the new base program and the repair is reinitialized. Else if the current multiplicity is less than the maximum multiplicity, then we increase the multiplicity and reinitialize the repair. If all of these options fail (the patch is not absolutely correct, nor strictly more-correct thana the base and the maaximum mutiplicity is reached) then we declare that patch generation fails to find a patch.

V. Empirical Observations

In order to test/ validate our tool, which we call *RCFix* (*RC* stands for *relative correctness*), we resolve to run it on benchmark programs with benchmark faults. Most program repair experiments we are aware of take a benchmark program, seed it with a benchmark fault, and test whether the proposed repair method can locate and remove the seeded fault. The advantage of relative correctness is that it enables us to repair a faulty program P into a program P' that is still faulty, while being more-correct than P. In order to highlight this advantage, we take benchmark programs, seed them with three benchmark faults, and show how RCFix retrieves them one by one; then we execte jGenProg on the same program, and observe its performance. We briefly present our experiments below; they are all applied to the *Math* project of *Defects4J*, albeit with different sets of faults [14].

A. MathI1, MathI2, Math70

The Selected Faults and their patches are:

- *Math70*. Fault: `return solve(min,max).` Patch: `return solve(f,min,max).`
- *MathI1*. Fault: `return 0`. Patch: `return result`.
- *MathI2*. Fault: `double real = 4.0*real;`. Patch: `double real = 2.0*real;`.

We deploy RCFix with the following (fault localisation and patch generation) parameters:

- Fault localisation threshold: 0.5.
- Number of modification points: 12.
- Maximum Mutiplicity of elementary faults, $MaxM$: 1. In this particular experiment, because each one of the selected modifications represents a fault (i.e. we find a substitute for it that enhances the correctness of the program), taking a greater value for $MaxM$ would make no difference.
- Maximum execution time: 9 hours.

We find the following results, in the order in which faults were repaired:

Fault	Number of Variants	Time (secs)	Time (mins)
MathI2	7	47	0.78
MathI1	91	648	10.8
Math70	1	24	0.40

We run jGenProg on the same program seeded with all three faults, using the following parameters:

- Maximum number of generations: 500.
- Population size: 40.
- Fault localisation threshold: 0.5.
- Maximum Execution time: 9 hours.

Execution of jGenProg times out after 9 hours without repairing any of the three faults; one possible explanation is that, because it is using absolute correctness, jGenProg can only repair the three faults simultaneously, and that the combinatorics of doing so are such that 9 hours is not sufficient. Also, note that RCFix deploys fault localisation after each fault removal; hence whenever it removes one fault, it gains better information to locate the following faults, whereas jGenProg operates with the initial fault localisation information throughout its search.

B. Math70, Math73, Math85

The Selected Faults and their patches are:

- *Math70*. Fault: `return solve(min,max).` Patch: `return solve(f,min,max).`
- *Math73*. Fault: A Missing Statement. Patch: `if (yMin*yMax>0) {throw Math.RuntimeException. createIllegalArgumentException (NON BRACKETING MESSAGE, min, max, yMin, yMax)}.`
- *Math85*. Fault: `if (fa*fb>=0);.` Patch: `if (fa*fb>0);.`

We deploy RCFix with the following (fault localisation and patch generation) parameters:

- Fault localisation threshold: 0.5.
- Number of modification points: 12.
- Maximum Mutiplicity of elementary faults, $MaxM$: 1.
- Maximum execution time: 9 hours.

We find the following results:

Fault	Number of Variants	Time (secs)	Time (mins)
Math70	1	24	0.40
Math85	77	370	6.16
Math73	1	23	0.38

We run jGenProg on the same program seeded with all three faults, using the following parameters:

- Maximum number of generations: 500.
- Population size: 40.
- Fault localisation threshold: 0.5.
- Maximum Execution time: 9 hours.

Execution of jGenProg times out after 9 hours without repairing any of the three faults; we try running it again by changing several of its parameters, increasing the maximum number of generations, to no avail.

C. Math80, Math84, Math95

The Selected Faults and their patches are:

- *Math80*. Fault: `int j=4*n-1`. Patch: `int j=4*(n-1).`
- *Math84*. Fault: A Missing Statement. Patch: `final RealConvergenceChecker... if (converged)...`
- *Math95*. Fault: `if (fa*fb>=0.0);.` Patch: `if (fa*fb>0.0);.`

We deploy RCFix with the following (fault localisation and patch generation) parameters:

- Fault localisation threshold: 0.5.
- Number of modification points: 12.
- Maximum Mutiplicity of elementary faults, $MaxM$: 1.
- Maximum execution time: 9 hours.

Our tool repairs the three faults in the following order:

Fault	Number of Variants	Time (secs)	Time (mins)
Math80	714	4817	80.28
Math84	218	1196	19.93
Math95	112	797	13.28

We run jGenProg on the same program seeded with all three faults, using the following parameters:

- Maximum number of generations: 500.
- Population size: 40.
- Fault localisation threshold: 0.5.
- Maximum Execution time: 9 hours.

Execution of jGenProg times out after 9 hours without repairing any of the three faults; we try running it again by changing several of its parameters, increasing the maximum number of generations, to no avail.

VI. Concluding Remarks

In this paper, we conjecture that to repair a program means to make it more-correct than it is, from which we infer that relative correctness ought to be an integral part of the study of program repair. Also, we propose a generic program repair algorithm whose main idea is to enhance relative correctness until it achieves absolute correctness, or stalls, i.e. fails to

enhance relative correctness if patch generation is inadequate. We derive an instance of this generic algorithm by combining its patch validation policy with the patch generation of jGenProg. Because it enhances correctness in a stepwise manner, this algorithm can theoretically repair programs at arbitrary fault depths by removing faults in sequence, one at a time.

To illustrate this property, we deploy our tool (RCFix) on benchmark programs seeded with benchmark faults, and find that it succeeds in locating and removing the faults sequentially, with execution times that range from a few second to about an hour. By contrast, jGenProg times out on the same program and sets of faults, probably because it attempts to repair all the faults simultaneously, since it tests repair candidates for absolute correctness.

We anticipate that RCFix would work even better under the following conditions:

- *Non Deterministic Specifications*. In the experiments reported in this paper, we use the base program as the specification of correct behavior, to be compliant with the benchmark;yet the definition of relative correctness shows that correctness with respect to non-deterministic specification does not require preserving correct behavior; using non-deterministic specifications enhances recall by weakening the oracle of relative correctness.
- *Large Negative Test Data Sets*. In the experiments reported in this paper, we use the test data set provided by the *Defects4J* benchmark; these data sets often include very few negative data points; having a large set of negative test data is useful because it allows us to monitor correctness enhancement with greater accuracy.
- *Non Deterministic Programs*. In this paper we have restricted our study of relative correctness to deterministic programs, and we have designed our oracles accordingly; yet in [8] we introduce a definition of relative correctness for non-deterministic programs, which remains heretofore unexplored for practical applications; we anticipate that using this broader concept would enhance the scope and effectiveness of our approach. A definition of relative correctness for non-deterministic programs enables us to reason about relative correctness between programs without having to refer to their function in all its detail.

These matters are currently under investigation.

REFERENCES

[1] Rui Abreu. Gzoltar: A toolset for automatic test suite minimization and fault identification. In *International Workshop on the Future of Debugging*, Lugano, Switzerland, July 2013.

[2] Martinez M. and Monperrus M. Mining software repair models for reasoning on the search space of automated program fixing. *Empirical Software Engineering*, 2013.

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

[4] Kim D., Nam J., Song J., and Kim S. Automatic patch generation learned from human-written patches. In *ICSE 2013*, pages 802–811, 2013.

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

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

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

[8] J. Desharnais, N. Diallo, W. Ghardallou, M. F. Frias, A. Jaoua, and A. Mili. Relational mathematics for relative correctness. In *RAMICS, 2015*, volume 9348 of *LNCS*, pages 191–208, Braga, Portugal, September 2015. Springer Verlag.

[9] Jules Desharnais, Nafi Diallo, Wided Ghardallou, Marcelo F. Frias, Ali Jaoua, and Ali Mili. Relational mathematics for relative correctness. In *Relational and Algebraic Methods in Computer Science, 2015*, volume 9348 of *Lecture Notes in Computer Science*, pages 191–208, Braga, Portugal, September 2015. Springer Verlag.

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

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

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

[13] C.A.R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–583, October 1969.

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

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

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

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

[18] Claire LeGoues, M. Dewey Vogt, Stephanie Forrest, and Wesley Weimer. A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each. In *Proceedings, ICSE 2012*, pages 3–13, 2012.

[19] Matias Martinez and Martin Monperrus. Astor: A program repair library for java. In *Proceedings, ISSTA 2016*, pages 441–444, Saarbrucken, Germany, July 2016.

[20] Matias Martinez and Martin Monperrus. Astor: Exploring the design space of generate-and-validate program repair beyond genprog, 2018.

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

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

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

[24] Hoang Duong Thien Nguyen, DaWei Qi, Abhik Roychoudhury, and Satish Chandra. Semfix: Program repair via semantic analysis. In *Proceedings, ICSE*, pages 772–781, 2013.

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

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

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

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