

# Automatic Program Repair by Fixing Contracts<sup>\*</sup>

Yu Pei, Carlo A. Furia, Martin Nordio, and Bertrand Meyer

Chair of Software Engineering, ETH Zurich, Switzerland  
firstname.lastname@inf.ethz.ch

**Abstract.** While most debugging techniques focus on patching implementations, there are bugs whose most appropriate corrections consist in fixing the specification to prevent invalid executions—such as to define the correct input domain of a function. In this paper, we present a fully automatic technique that fixes bugs by proposing changes to contracts (simple executable specification elements such as pre- and postconditions). The technique relies on dynamic analysis to understand the source of buggy behavior, to infer changes to the contracts that emend the bugs, and to validate the changes against general usage. We have implemented the technique in a tool called *SpeciFix*, which works on programs written in Eiffel, and evaluated it on 44 bugs found in standard data-structure libraries. Manual analysis by human programmers found that *SpeciFix* suggested repairs that are deployable for 25% of the faults; in most cases, these contract repairs were preferred over fixes for the same bugs that change the implementation.

## 1 Introduction

A software *bug* is the manifestation of a discrepancy between specification and implementation: program behavior (implementation) deviates from expectations (specification). Correcting a bug may thus require changing implementation, specification, or both. In fact, there is a significant number of bugs [3] whose most appropriate correction is changing the specification to rectify the expectations about what the implementation ought to do. For example, a function `max` computing the maximum value of a set of integers is undefined if the set is empty; we could change `max`'s implementation to return a special value when called on an empty set, but the best thing to do is disallowing such calls altogether by specifying them invalid. However, since specifications are often informal or implicit at best, debugging techniques normally modify implementations rather than specifications. In particular, fully automatic fixing—which has made substantial progress in recent years [13, 20, 21] (see Section 5 for more references)—has focused on suggesting repairs to implementations, thus failing to provide the best corrections in cases where the ultimate source of failure is incorrect specification.

This paper presents a fully automatic technique that fixes bugs by rectifying specifications. Our technique targets programs with *contracts*—simple specification elements in the form of executable assertions. A program execution that violates some contract reveals a bug; to fix it, the technique suggests changes to the contracts that prevent the violation from being triggered. We have prototyped the technique in a tool called

---

<sup>\*</sup> Work partially supported by ERC grant CME/291389; by SNF grants LSAT/200020-134974 and ASII/200021-134976; and by Hasler-Stiftung grant #2327.

SpeciFix, which works on programs with contracts written in Eiffel. (However, the same technique is implementable in any language supporting some form of contracts.) SpeciFix is completely automatic: its only required input are programs with simple contracts. In an experimental evaluation, we applied SpeciFix to 44 bugs of Eiffel’s standard data-structure libraries. SpeciFix suggested repairs for 42 of these bugs; more significant, 11 of the bug repairs are genuine corrections of quality sufficient to be deployable. A small trial with human programmers confirmed this assessment and often found the fixes produced by SpeciFix preferable to fixes for the same bugs that modified the implementation rather than the contracts.

Fixing contracts relies on extracting specification elements based on the actual behavior of the implementation. This is superficially similar to the problem of *inferring* (or mining) specifications—a well-established research area that produced numerous landmark results (e.g., [4, 8]; see Section 5 for more references). While SpeciFix uses inference techniques as one of its components, suggesting changes to an existing specification to correct a bug is more delicate business than just inferring specifications. Changing contracts is changing the design of an API as experienced by its clients. In the example of `max`, adding a precondition that requires that the set be non empty makes all client code of `max` responsible for satisfying the requirement upon calling `max`. Therefore, we must make sure that the suggested contract changes have a limited impact on a potentially infinite number of clients.

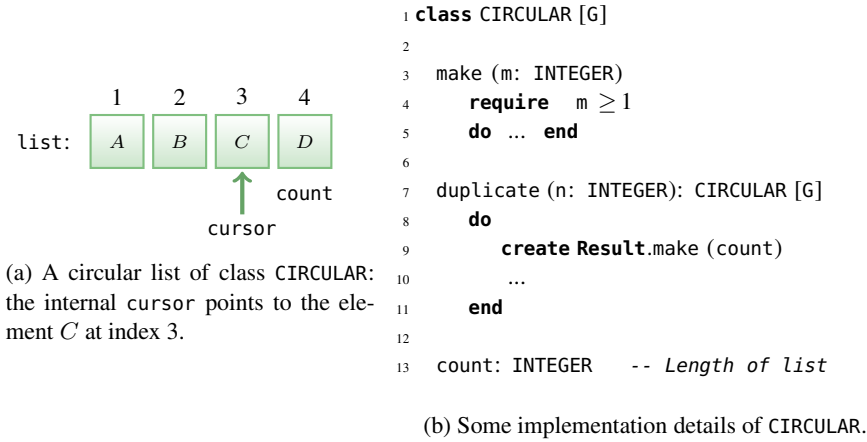
The SpeciFix technique presented in this paper uses a combination of heuristics to validate possible specification fixes with respect to their impact on client code. It discards fixes that invalidate previously passing test cases; to avoid overfitting, it runs every candidate fix through a regression testing session that generates (completely automatically, using our testing framework AutoTest) new executions; and it ranks all fixes that pass regression by preferring those that are the least restrictive. The empirical evaluation in Section 4 indicates that these heuristics work well in practice for the bugs we considered. Notably, there is a significant fraction of bugs whose appropriate fix is a change to the specification; in those cases, SpeciFix can often generate useful fixes.

Section 2 demonstrates the idea of fixing specifications by means of an actual example from the standard Eiffel implementation of array-based circular lists. Section 3 presents the technique implemented in SpeciFix, starting with an overview of its components (Figure 3) followed by a detailed description of each of them. For brevity, we use the name “SpeciFix” to denote both the fixing technique presented in this paper and its prototype implementation. The evaluation in Section 4 presents experiments where we applied SpeciFix to 44 faults in standard data-structure libraries. Section 5 discusses the essential related work; and Section 6 concludes and outlines future work.

## 2 SpeciFix in Action

Let us briefly demonstrate how SpeciFix works using an example from the experimental evaluation of Section 4. The example targets a bug of routine (method) `duplicate` in class `CIRCULAR`, which is the standard Eiffel library implementation of circular array-based lists.

To understand the bug, Figure 1 illustrates a few details of `CIRCULAR`’s API. Lists are numbered from index 1 to index count (an attribute denoting the list length), and



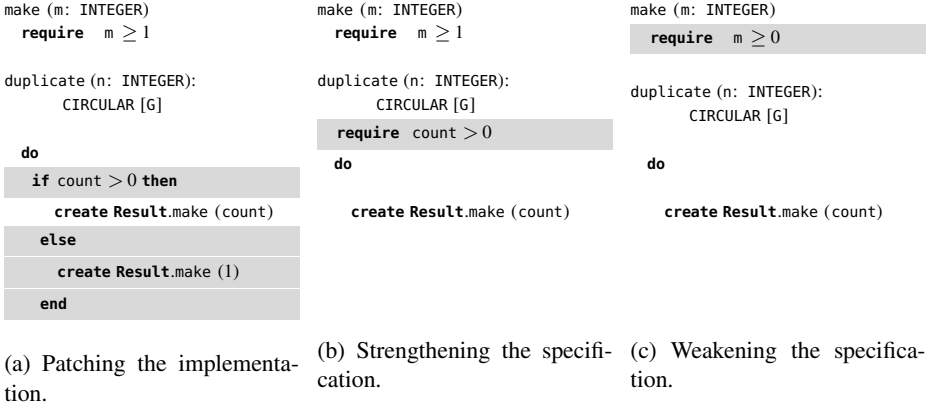
**Fig. 1.** Example and some API details of circular lists in Eiffel

include an internal cursor that may point to any element of the list. Routine `duplicate` takes a single integer argument *n*, which denotes the number of elements to be copied; called on a list object `list`, it returns a new instance of `CIRCULAR` with at most *n* elements copied from `list` starting from the position pointed to by `cursor`. Since we are dealing with circular lists, the copy wraps over to the first element. For example, calling `duplicate (3)` on the `list` in Figure 1a returns a fresh list with elements  $\langle C, D, A \rangle$  in this order.

The implementation of `duplicate` is straightforward: it creates a fresh `CIRCULAR` object `Result` (line 9 in Figure 1b); it iteratively copies *n* elements from the current list into `Result`; and it finally returns the list attached to `Result`. The call to the creation procedure (constructor) `make` on line 9 allocates space for a list with `count` elements; this is certainly sufficient, since `Result` cannot contain more elements than the list that is duplicated. However, `CIRCULAR`'s creation procedure `make` includes a precondition (line 4 in Figure 1b) that only allows allocating lists with space for at least one element (**require**  $m \geq 1$ ). This sets off a bug when `duplicate` is called on an empty list: `count` is 0, and hence the call on line 9 triggers a violation of `make`'s precondition. Testing tools such as `AutoTest` detect this bug automatically by providing a concrete test case that exposes the discrepancy between implementation and specification.

How should we fix this bug? Figure 2 shows three different possible repairs, all of which we can generate completely automatically. An obvious choice is patching `duplicate`'s implementation as shown in Figure 2a: if `count` is 0 when `duplicate` is invoked, allocate `Result` with space for *one* element; this satisfies `make`'s precondition in all cases. Our `AutoFix` tool [17, 20] targets fixes of *implementations* and in fact suggests the patch in Figure 2a.

The fix that changes the implementation is acceptable, since it makes `duplicate` run correctly, but it is not entirely satisfactory: `CIRCULAR`'s implementation looks perfectly adequate, whereas the ultimate source of failure seems to be incorrect or inadequate *specification*. A straightforward fix is then adding a precondition to `duplicate` that



**Fig. 2.** Three different fixes for the bug of Figure 1. Changed or added lines are highlighted.

only allows calling it on non-empty lists. Figure 2b shows such a fix, which *strengthens* `duplicate`'s precondition thus invalidating the test case exposing the bug. The strengthening fix has the advantage of being textually simpler than the implementation fix, and hence also probably simpler for programmers to understand. However, both fixes in Figures 2a and 2b are partial, in that they remove the source of faulty behavior in `duplicate` but they do not prevent similar faults—deriving from calling `make` with  $m = 0$ —from happening. A more critical issue with the specification-strengthening fix in Figure 2b is that it may break clients of `CIRCULAR` that rely on the previous weaker precondition.<sup>1</sup> There are cases—such as when computing the maximum of an empty list—where strengthening produces the most appropriate fixes; in the running example, however, strengthening arguably is not the optimal strategy.

A look at `make`'s implementation (not shown in Figure 1b) would reveal that the creation procedure's precondition  $m \geq 1$  is unnecessarily restrictive, since the routine body works as expected also when executed with  $m = 0$ . This suggests a fix that *weakens* `make`'s precondition as shown in Figure 2c. This is arguably the most appropriate correction to the bug of `duplicate`: it is very simple, it fixes the specific bug as well as similar ones originating in creating an empty list, and it does not invalidate any clients of `CIRCULAR`'s API. The Specifix tool described in this paper generates both specification fixes in Figures 2b and 2c but ranks the weakening fix higher than the strengthening one. More generally, Specifix outputs specification-strengthening fixes only when they do not introduce bugs in available tests, and it always prefers the least restrictive fixes among those that are applicable.

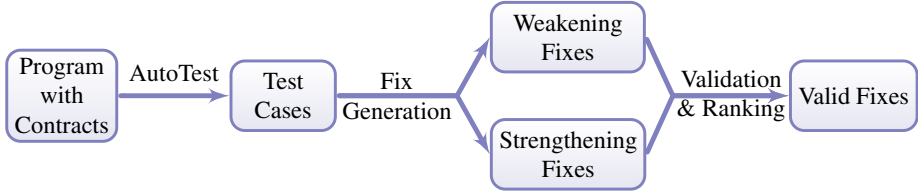
### 3 How Specifix Works

Specifix works completely automatically: its only input is an Eiffel program annotated with simple contracts (pre- and postconditions and class invariants) which constitute its

<sup>1</sup> Note that this strengthening does not introduce new bugs; it just shifts the responsibility for the fault from `duplicate` to its clients.

specification. After going through the steps described in the rest of this section, Specifix's final output is a list of fix suggestions for the bugs in the input program.

Figure 3 gives an overview of the components of the Specifix technique. Specifix is based on dynamic analysis, and hence it characterizes correct and incorrect behavior by means of passing and failing *test cases* (Sections 3.1 and 3.2). To provide full automation, we use the random testing framework AutoTest to generate the tests used by Specifix. The core of the *fix generation* algorithm applies two complementary strategies (Section 3.3): weaken (i.e., relax) a violated contract if it is needlessly restrictive; or strengthen an existing contract to rule out failure-inducing inputs. Specifix produces *candidate fixes* using both strategies, possibly in combination (Section 3.4). To determine whether the weaker or stronger contracts remove all faulty behavior in the program, Specifix runs candidate fixes through a *validation* phase (Section 3.5) based on all available tests. To avoid overfitting, some tests are generated initially but used only in the validation phase (and not directly to generate fixes). If multiple fixes for the same fault survive the validation phase, Specifix outputs them to the user *ordered* according to the strength of their new contracts: weaker contracts are more widely applicable, and hence are ranked higher than more restrictive stronger contracts (Section 3.5).



**Fig. 3.** An overview of how Specifix works. Running AutoTest on an input Eiffel program with contracts produces a collection of test cases that characterize correct and incorrect behavior. With the goal of correcting faulty behavior, the fix generation algorithm builds candidate fixes using two strategies: weakening and strengthening the existing contracts. The candidate fixes enter a validation phase where they must pass all valid test cases; valid fixes are ranked—the weaker the new contracts the higher the ranking—and presented as output.

### 3.1 Test Cases

A *test case* (or just “test”)  $t$  consists of a call of some routine  $r$  with actual arguments  $a_1, \dots, a_n$  on a target object  $a_0$ , written  $t : a_0.r(a_1, \dots, a_n)$ ; we refer to  $r$  as  $t$ 's *outermost routine*. For instance, if `list` is the list of Figure 1a and `emp` is an instance of empty CIRCULAR list, `list.duplicate(3)` and `emp.duplicate(1)` are two tests.

Let  $\mathcal{S}$  be a set of program states. The execution of a test  $t$  starts with routine  $r$ 's body executing from an initial state  $s_0 \in \mathcal{S}$ . In general,  $r$ 's body may call another routine  $r_1$  from a state  $s_1$ , which in turn calls another  $r_2$  from a state  $s_2$ , and so on until the test terminates.<sup>2</sup> Therefore, a test  $t$  uniquely defines a *trace*  $\rho_t$  as the sequence

$$\rho_t = s_0 r_0 s_1 r_1 \cdots s_{n-1} r_{n-1} s_n r_n \quad (1)$$

<sup>2</sup> To avoid dealing with nonterminating programs, we forcibly terminate tests that are still running after a timeout.

of state snapshots when nested routines are called or return. Precisely, for  $j = 0, \dots, n$ , a pair  $s_j r_j$  denotes either that routine  $r_j$  begins execution from state  $s_j$ , that is  $s_j$  is the *pre-state* of a nested call; or that routine  $r_j$  returns to the caller from state  $s_j$ , that is  $s_j$  is the *post-state* of a nested call. Since  $t$  is a call to  $r$  at the outermost level,  $r_0 = r$ ; call traces ignore intermediate states other than pre- and post-states. The sequence  $\kappa_t = r_0 r_1 \dots r_{n-1} r_n$  containing only routine names in  $\rho_t$  is the *call sequence* determined by  $t$ . For example, the test `emp.duplicate(1)` determines the trace  $x_0$  duplicate  $x_1$  make where  $x_0$  is the initial state and  $x_1$  is the state when calling make on line 9 in Figure 1b; the test terminates then with a contract violation. The other test `list.duplicate(3)` determines the trace  $y_0$  duplicate  $y_1$  make  $y_2$  make  $y_3$  duplicate where  $y_0$  is the initial state,  $y_1$  is the state when calling make,  $y_2$  is the state when make returns, and  $y_3$  is the state when duplicate and the whole test terminates.

In SpeciFix, we generate test cases automatically using AutoTest—Eiffel’s random test generator. However, if manually-written test cases are available, they can also be supplied to SpeciFix to supplement the automatically generated tests; the extra input may improve the quality of the final output.

### 3.2 Contracts, Correctness, and Faults

Contracts are simple specification elements made of assertions including preconditions (**require**), postconditions (**ensure**), and class invariants (**invariant**). We denote by  $P_r$  and  $Q_r$  the pre- and postcondition of a routine  $r$ . In this work, we focus on changing pre- and postconditions only; thus, we use the term *specification* to collectively denote pre- and postconditions, and use the terms “specification” and “contracts” as synonyms.

Given an assertion  $A$  (pre- or postcondition) and a program state  $s \in \mathcal{S}$ , we say that  $A$  *holds* at  $s$  (or, equivalently, that  $s$  satisfies  $A$ ) if  $A$  evaluates to **True** under state  $s$ ; if this is the case, we write  $s \models A$ . Since contracts are executable, we can evaluate any assertion at any program state reached during a concrete execution.

Contracts provide an operational criterion to classify test cases into invalid, passing, and failing. A test case  $t$  is *valid* if the initial state  $s_0$  of the trace  $\rho_t$  is such that it satisfies  $r$ ’s precondition, that is  $s_0 \models P_r$ ; otherwise  $t$  is *invalid*. An invalid test case for routine  $r$  does not tell us anything about  $r$ ’s correctness, since every invocation of  $r$  should satisfy  $r$ ’s precondition to be acceptable. A valid test case  $t$  is *passing* if, for every  $j = 1, \dots, n$ , state  $s_j$  in  $t$ ’s trace  $\rho_t$  satisfies the following: if  $s_j$  is the pre-state of a call to  $r_j$  then  $s_j \models P_{r_j}$ ; and if  $s_j$  is the post-state of a call to  $r_j$  then  $s_j \models Q_{r_j}$ . In words, every nested call performed during the computation of  $r$  starts in a state that satisfies the called routine’s precondition and terminates in a state that satisfies the called routine’s postcondition when it returns. A valid test case is *failing* if it is not passing, that is if it eventually reaches a state that violates some pre- or postcondition; the violation terminates test case execution. The test `list.duplicate(3)` is passing because the call to duplicate terminates without violating any contract (and produces the correct result). The other test `emp.duplicate(1)` is valid but failing: the nested call to make does not satisfy make’s precondition  $m \geq 1$  on line 4 in Figure 1b because `count = 0 < 1` in an empty list.

A failing test case  $t$  reveals a *fault* (informally called bug in the introduction) in routine  $r$ , namely a discrepancy between implementation and specification (the violated

contract). Conversely, a passing test case documents a legitimate usage of routine  $r$  with respect to its specification. Two failing test cases  $t_1, t_2$  identify the *same fault* if their call sequences  $\kappa_1, \kappa_2$  are the same (and hence they violate the same assertion).

### 3.3 Weakening vs. Strengthening

Let  $t$  be a failing test case with trace  $\rho_t$  as in (1);  $r = r_0$  is the outermost routine of  $t$ , and  $r_n$  is the routine whose contract violation triggers the fault. Assuming the implementation of all routines  $r_0, \dots, r_n$  is correct, we should change the contracts of  $r_0, r_1, \dots, r_n$  to fix the fault exposed by  $t$ . There are two ways to do that:

**Strengthening:** strengthen  $r$ 's precondition to disallow  $t$ 's input. Strengthening makes  $t$  invalid and thus prevents the call sequence that led to the violation of  $r_n$ 's contract.

**Weakening:** weaken  $r_n$ 's contract to allow  $t$ 's execution to continue past  $r_n$ . If the execution can continue without triggering other errors, weakening makes  $t$  passing.

If applicable, weakening is in principle preferable to strengthening, because the former does not risk breaking clients by introducing more stringent conditions for correctly calling  $r$ . Strengthening is, however, always applicable, whereas weakening may not work if  $r_n$ 's correct execution depends on the weakened contract. Even in the cases where weakening makes  $t$  passing without triggering any new fault, it may be that the absence of new faults is just a result of the rest of the specification being inaccurate or incomplete. For example, weakening the precondition of a function `max` to work on lists of any size (including empty lists) may not trigger any faults simply because `max` has no postcondition, and hence there is no automatic way of finding out that the value returned for empty lists is inconsistent.

In practice, Specifix prefers the least restrictive fixes (i.e., weakening) but always tries both weakening and strengthening in combination. Another observation is that strengthening only the outermost routine's precondition often is too *ad hoc*, since it corresponds to a partial change of API assumptions which may be inconsistent with the way other routines are used. Therefore, Specifix tries to collectively strengthen all routines  $r_0, \dots, r_{n-1}$  to disallow fault-inducing input at every call site. Indeed, the experiments of Section 4 show that strengthening leads to many useful and correct fixes in practice.

### 3.4 Fix Generation

A run of Specifix targets a specific fault of some routine  $r$ . This is characterized by a set  $\mathcal{F}_r$  of failing test cases all of which have  $r$  as outermost routine and identify the same fault—the violation of contract  $A_n$  (pre- or postcondition) of routine  $r_n$ . To characterize correct behavior, Specifix also inputs a set  $\mathcal{P}_r$  of passing test cases which have  $r$  as outermost routine. Based on this, Specifix builds a set  $\Phi$  of candidate fixes through the following steps, illustrated on the running example.

**Build weakening assertions  $\Omega$  for  $r_n$ .** Let  $\tilde{r}_n$  be  $r_n$  with  $A_n$  relaxed to **True**. Generate fresh sets  $\tilde{\mathcal{P}}$  and  $\tilde{\mathcal{F}}$  of passing and failing test cases for  $\tilde{r}_n$ . Based on them, determine the sets  $\mathcal{I}^{\tilde{\mathcal{P}}}$  and  $\mathcal{I}^{\tilde{\mathcal{F}}}$  of dynamic *invariants* respectively holding in all passing

tests  $\tilde{\mathcal{P}}$  and in all failing tests  $\tilde{\mathcal{F}}$  (Section 3.6 describes the dynamic invariant detection process). Let  $\Omega = \{\omega \mid \omega \in \mathcal{I}^{\tilde{\mathcal{P}}} \text{ and } \neg\omega \in \mathcal{I}^{\tilde{\mathcal{F}}}\}$  be a set of weakening assertions, which characterize the minimal requirements for a test of  $\tilde{r}_n$  to be passing and not failing. In the example, `make` works without errors when  $m \geq 0$ , whereas it fails when  $m < 0$ ; thus  $\Omega = \{m \geq 0\}$ .

**Build weakening fixes  $W$ .** For each  $\omega \in \Omega \cup \{\mathbf{False}\}$ , build the weakening fix  $f$  obtained by replacing  $A_n$  with  $A_n \vee w$  in  $r_n$ . Add  $f$  to the set  $W$  of weakening fixes. Adding **False** to  $\Omega$  determines a dummy fix which is used to build purely strengthening fixes in the next step. In the example,  $W$  contains a weakening fix  $f_w$  corresponding to the one in Figure 2c, and a dummy fix  $f_0$  where `make`'s precondition has been “weakened” with **False** (hence it is unchanged).

**Validate weakening fixes.** For each  $f \in W$ , if  $f$  passes all tests in  $\mathcal{P}_r \cup \mathcal{F}_r$  then add  $f$  to the set  $\Phi$  of candidate fixes without modifications, and remove it from  $W$ . In the example,  $f_w$  passes validation and is added to  $\Phi$ .  $f_0$  is instead the unchanged program in Figure 1b, and hence it stays in  $W$ .

**Build strengthening assertions  $\Sigma_k$  for  $r_k$ .** For each  $f \in W$  that did not pass validation, determine the sets  $\mathcal{I}_k^{\mathcal{P}}$  and  $\mathcal{I}_k^{\mathcal{F}}$  of dynamic *invariants* currently holding in all pre-states of the calls to  $r_k$  respectively in the passing tests  $\mathcal{P}_r$  and in the failing tests  $\mathcal{F}_r$ ;  $k$  ranges over the subset of  $\{0, \dots, n-1\}$  for which  $s_k$  is a pre-state ( $s_k r_k$  appears in the traces). Let  $\Sigma_k = \{\sigma \mid \sigma \in \mathcal{I}_k^{\mathcal{P}} \text{ and } \neg\sigma \in \mathcal{I}_k^{\mathcal{F}}\}$  be the corresponding sets of strengthening assertions, which characterize the minimal additional requirements for a test to pass through  $r_k$  without failing. In the example, duplicate correctly calls `make` precisely when `count > 0`; thus,  $\Sigma_0 = \{\text{count} > 0\}$ .

**Build strengthening fixes.** For each combination  $\langle \sigma_0, \dots, \sigma_{n-1} \rangle \subseteq \Sigma_0 \times \dots \times \Sigma_{n-1}$  of strengthening assertions, build the strengthening fix  $\phi$  obtained by replacing each precondition  $P_{r_k}$  of routine  $r_k$  with  $P_{r_k} \wedge \sigma_k$ , for all applicable  $k$ . Add  $\phi$  to the set  $\Phi$  of candidate fixes. In the example, the dummy fix  $f_0$  is turned into a valid fix  $\phi_0$  by strengthening `duplicate`'s precondition as `count > 0`.

**Candidates.** The output of the fix generation phase is a set  $\Phi$  of fix candidates. The candidates are filtered and ranked as explained in the following section.

### 3.5 Fix Validation and Ranking

**Validation.** The purpose of the *validation* phase is to ascertain which of the candidate fixes in  $\Phi$  remove the fault under analysis. To this end, Specifix runs every fix candidate  $f \in \Phi$  through all available tests for  $r$ ;  $f$  is *valid* if it still passes all originally passing tests, and it also passes all originally failing tests that have not become invalid.

The dual risk of unsoundness for validation based on a finite number of test cases is *overfitting*: a fix may pass validation but be unusable in a general context, because it introduces specification changes that harm usages of the API different from those exercised by the test cases used to generate the fix. To reduce the risk of overfitting, Specifix uses only half of the originally generated test cases to generate the candidate fixes. Then, the validation phase uses *all* available tests for the routine under analysis, not only those in  $\mathcal{P}_r$  and  $\mathcal{F}_r$  used to generate fixes. This increases the likelihood that the validated fixes are applicable beyond the specific cases that drove fix generation.



**Ranking.** Not all valid fixes are equally desirable: all else being equal, we prefer those that introduce the least changes to the specification, and that make invalid the fewest test cases. Specifix ranks valid fixes to reflect these criteria, and only reports the top five fixes for each fault. This approach is a good compromise between the contrasting needs of exposing programmers to a limited number of fixes—which they have to understand and validate—and of retaining fixes that fall behind in the ranking even if they are of high quality, due to the imperfect precision of the ranking heuristics.

The ranking heuristics is based on two elements: number of invalidated tests and the strength of the new contracts. A fix  $f$  consists of a collection  $\langle A_0, \dots, A_n \rangle$  of new contracts for the routines  $r_0, \dots, r_n$ ; each  $A_k$  ( $0 \leq k \leq n$ ) is either a pre- or a postcondition and may be weaker, stronger, or unchanged with respect to the original program. Given two valid fixes  $f_1, f_2$ , let  $A_k^1, A_k^2$  be their new contracts for the same routine  $r_k$ . We say that  $A_k^1$  is *not stronger than*  $A_k^2$ , written  $A_k^1 \preceq A_k^2$ , if  $A_k^1$  holds whenever  $A_k^2$  holds; precisely, we determine strength based on executing all available tests for  $r$ :  $A_k^1 \preceq A_k^2$  iff every test that is valid for  $A_k^1$  (i.e., a test that leads to executions where  $A_k^1$  is evaluated and holds) is also valid for  $A_k^2$  (i.e.,  $A_k^2$  is evaluated and holds). This generalizes to an ordering between fixes by lexicographic generalization of  $\preceq$  on tuples  $\langle A_0, \dots, A_n \rangle$ . The ordering is partial because the sets of valid test cases for  $f_1$  and for  $f_2$  may be non-comparable. The final ranking orders fixes according to the  $\preceq$  relation and, for incomparable fixes, ranks higher those that determine the higher number of valid (and hence passing) tests.

In the running example, the weakening fix in Figure 2c ranks higher than the strengthening fix in Figure 2b: all test cases with count  $> 0$  are equivalent for the two fixes, but the test cases with count  $= 0$  are valid only for the weakening fix.

### 3.6 Dynamic Invariants and State Abstraction

Specifix infers invariants at program states dynamically by observing the behavior during concrete executions. Dynamic invariant inference (see Section 5) has become a standard technique of dynamic analysis. Using the notation of Section 3.1, we can define an invariant at the entry of routine  $r_k$  as an assertion  $I$  such that  $s_k \models I$  for every test  $t$  whose trace  $\rho_t$  includes the snapshot  $s_k r_k$  where  $s_k$  is a pre-state; the invariant at routine exit is defined similarly with respect to post-states.

Invariant inference in Specifix must cater to the specific needs of fixing contracts. To this end, we abstract the concrete program state by a number of predicates that include public queries (i.e., routines or attributes giving a value characterizing object state) as well as any subexpressions of the available contracts.

## 4 Experimental Evaluation

We performed a preliminary evaluation of the behavior of Specifix by applying it to 44 bugs of production software. The overall goal of the evaluation is corroborating the expectation that, for bugs whose “most appropriate” correction is fixing the specification, Specifix can produce repair suggestions of good quality. A more detailed evaluation taking into account aspects such as robustness and readability of the produced fixes belongs to future work.

#### 4.1 Experimental Setup

We selected 10 of the most widely used data-structure classes of the EiffelBase (rev. 92914) and Gobo (rev. 91005) libraries—the two major Eiffel standard libraries. While these are the same classes used in the experimental evaluation of AutoFix [20], we did not attempt a direct comparison for different reasons. First, some of the bugs used in AutoFix have been fixed in the latest library versions, and hence they are not reproducible. Second, AutoFix and Specifix are complementary approaches: our experience with AutoFix suggested that there is a substantial fraction of bugs whose most appropriate correction is fixing the specification, and it is precisely on those that we expect Specifix to work successfully. Third, running Specifix on the very same input as AutoFix would limit the generalizability of the evaluation results; instead, we want to evaluate the behavior of Specifix in standard conditions and avoid overfitting.

All the experiments ran on a Windows 7 machine with a 2.6 GHz Intel 4-core CPU and 16 GB of memory. We ran AutoTest for one hour on each of the 10 classes in Table 4. This automatic testing session found 44 unique faults consisting of pre- or postcondition violations. We ran Specifix on each of these faults individually, using only half of the test cases (randomly picked among those generated for each fault in the one-hour session) to generate the fixes and all of them in the validation phase (Section 3.5). The right-hand side of Table 4 reports, for each class, the total number of test cases used by Specifix, and the total time for testing (the initial one-hour sessions plus additional calls to AutoTest to generate tests for relaxed routines used to infer the weakening assertions  $\Omega$ , as described in Section 3.4) and fixing. The average figures *per fault* are: 106.4 minutes of testing time and 7.2 minutes of fixing time (minimum: 4.1 minutes, maximum: 30 minutes, median 6.2 minutes). The testing time dominates since AutoTest operates randomly and thus generates many test cases that will not be used (such as passing tests of routines without faults).

**Table 4.** Classes used in the experiments; for each class we report: lines of code LOC, number #R of routines, number #P of assertions in preconditions, number #Q of assertions in postconditions, and number #C of assertions in the class invariant. In the right-hand side, we report the number #F of faults targeted by the experiments, the total number of test cases (passing #P and #F failing) used by Specifix, the  $T_t$  minutes spent running AutoTest on routines of the class, and the  $T_f$  minutes spent running Specifix (net of testing time) on faults of the class.

CLASS	LOC	#R	#P	#Q	#C	#F	#P	#F	$T_t$	$T_f$
ACTIVE_LIST	2165	139	91	121	25	2	212	210	240	23
ARRAY	1474	101	70	110	10	9	850	555	900	72
ARRAYED_CIRCULAR <sup>3</sup>	1907	133	80	92	23	3	320	234	360	17
ARRAYED_SET	2346	146	118	131	26	6	554	432	720	34
DS_ARRAYED_LIST	2862	168	219	173	15	3	132	89	240	15
DS_HASH_SET	3159	171	154	140	20	1	14	60	120	5
DS_LINKED_LIST	3497	162	207	166	13	3	360	25	360	25
LINKED_LIST	1995	109	70	91	23	0	—	—	60	—
LINKED_SET	2347	122	99	101	26	4	416	70	480	22
TWO_WAY_SORTED_SET	2856	141	118	118	31	13	1260	655	1260	106
TOTAL	24608	1392	1226	1243	212	44	4118	2330	4680	319

<sup>3</sup> Shortened to CIRCULAR in Section 2.

## 4.2 Results

Evaluating the effectiveness of repairs that modify contracts is a somewhat subtle issue, since it ultimately involves what is a design choice: changing API specification. Related work on automatic repair (see Section 5) has rarely, if ever,<sup>4</sup> assessed the quality and *acceptability* for human programmers of the produced fixes beyond running standard regression test suites. To this end, in previous work [17,20] we introduced the notions of *valid* and *proper* fix: any fix that passes all the available tests is valid (and hence every fix output by Specifix is valid), but only those that manual inspection reveals to satisfactorily remove the real source of failure without introducing other bugs are classified as *proper*. Even if the line between proper and improper might be fuzzy in some corner cases, we could normally confidently classify fixes into proper and improper based on our familiarity with the code base under analysis.

We use the same classification criterion in the evaluation of fixes produced by Specifix: Table 5 lists the total number of faults for which Specifix generated valid or proper fixes (and ranked them in the top 5 positions: we ignore fixes that rank lower).

*For 25% of the faults, Specifix produced fixes that manual inspection revealed to satisfactorily remove the real source of failure.*

**Table 5.** Fixes built by Specifix. For each TYPE of fault, the left-hand side of the table reports the number #F of faults of that type input to Specifix, and for how many of those faults Specifix built (at least one) VALID or PROPER fixes. The right-hand side reports the total number of *fixes* produced in each category; the same fault may have multiple valid or proper fixes. Columns ALL list all fixes in each category, followed by a breakdown into purely weakening (WEAK), purely strengthening (STRONG), and mixed (involving BOTH strengthening of some contract and weakening of some other).

TYPE OF FAULT	#F	VALID	PROPER	VALID FIXES				PROPER FIXES			
				ALL	WEAK	STRONG	BOTH	ALL	WEAK	STRONG	BOTH
Precondition violation	22	22	7	77	23	30	24	13	1	12	0
Postcondition violation	22	20	4	71	56	13	2	7	3	4	0
TOTAL	44	42	11	148	79	43	26	20	4	16	0

The percentage of proper fixes (25% of faults) is similar to that obtained in the work with AutoFix; but the high percentage of valid fixes (over 90%) requires some explanation. Obtaining valid contract fixes is easy if only poor-quality tests are available. One can always strengthen preconditions to invalidate failing test cases (or, conversely, weaken failing postconditions to trivially pass tests): since Specifix validates fixes based on the available test cases, which in turn are only as good as the contracts of the class (beyond those directly targeted by the fix), such straightforward fixes yield valid repairs for classes equipped with very weak and incomplete contracts. This does not mean that such fixes are always improper; in fact, 80% of all proper fixes strengthen preconditions: it is only when it is combined with very poor specification (especially class invariants) that fixing may lead to improper fixes. Furthermore, despite being not

<sup>4</sup> The only exception we are aware of is [13].

directly deployable, the valid but improper fixes produced by Specifix are still very valuable as debugging aids, since they clearly highlight the failure-inducing inputs.

**Acceptability Trial.** In order to get more confidence in the capability of Specifix to produce proper, acceptable fixes from a programmer’s perspective, we conducted a small trial involving 4 PhD students (henceforth, the “subjects”) in our group. The subjects were quite familiar with the Eiffel language and its standard libraries, but had not been involved in the work on Specifix or AutoFix. To keep the workload small, we randomly selected only 8 out of the 11 faults for which Specifix produced proper fixes, and submitted them to the subjects: for each fault, we produced one failing test case (randomly picked among those produced by AutoTest) and up to 3 fixes produced by Specifix. In order to compare the acceptability of specification and implementation fixes, we also included up to 2 proper implementation fixes for each of 5 faults (out of 8) produced using AutoFix. For each fault, the subjects: (1) declared which fixes they considered acceptable (i.e., they “correct the fault while not introducing new faults”, as in our definition of “proper”); and (2) ordered the fixes in decreasing order of quality.

See the extended version of this paper for detailed results. The highlights: all subjects but one agreed with our assessment of proper fixes; the subjects unanimously preferred a contract fix over an implementation fix for 3 of the 5 faults that had both kinds of fix. The subject who disagreed about proper fixes still agreed that the contract fixes for 6 out of 8 faults are proper. With the proviso that its small scale does not warrant arbitrary generalizations, the trial demonstrates substantial agreement with our assessment of proper fixes; and suggests that, if a fault can be fixed with a contract fix, Specifix has a chance of building a high-quality one.

*Programmers found most proper fixes produced by Specifix acceptable and often preferable to fixes for the same bugs that change the implementation.*

### 4.3 Limitations and Threats to Validity

*Limitations.* The main limitation to the applicability of Specifix is that it requires contracts. On the one hand, it requires a language where contracts are expressible; this is an obvious consequence of the technique’s goals and is not severely restrictive since many languages support some form of notation for contracts (e.g., JML for Java and CodeContracts for C#). On the other hand, Specifix works well only on classes that come already equipped with *some* contracts of decent quality. Class invariants (which Specifix does not change but only assumes) are particularly useful to ensure that the test cases generated represent reasonable usage, so that validation (Section 3.5) is precise. Despite being often weak and largely incomplete, the kinds of contracts Eiffel programmer write have been sufficient to get good experimental results; but in future work we will investigate how Specifix performance improves if it is given more expressive contracts [18].

*Threats to Validity.* The most significant threat to *external* validity—concerning the generalizability of our experimental results—comes from limiting the experiments to data-structure classes. This is a limitation partly inherited from the usage of AutoTest to

generate test cases; AutoTest is meant for unit testing and hence works more easily with classes with a clearly defined interface such as data structures. In future work, we plan to experiment with other kinds of program (as we already did successfully with AutoFix [17]) and possibly with manually-written test cases. Another threat comes from the small number of subjects used in the trial (Section 4.2), and the fact that they all were graduate students. We acknowledge that the trial only gives a preliminary assessment, and more user studies are needed to ensure generalizability.

Threats to *internal* validity—concerning the proper execution of our experiments—include repeatability. Since Specifix uses AutoTest to generate test cases, and the performance of AutoTest is affected by chance, different runs may yield different results. Based on our previous extensive experience with using AutoTest’s test cases for dynamic analysis [17–20], we expect AutoTest behavior to be predictable over the testing time allotted in our experiments; therefore, this threat is unlikely to be significant. Since Specifix produces many valid but not proper fixes, an issue is how much effort is required to identify the improper fixes. While we have no hard evidence about this, even improper fixes succinctly characterize the failure-inducing inputs, and hence they are still useful as debugging aids. Furthermore, contract fixes are normally quite simple, arguably easier to read than implementation fixes; all subjects in the trial spent on average around two minutes to classify each contract fix, which seems to indicate an acceptable overhead. More experiments are also needed to determine the sensitivity of Specifix to what fraction of the tests are used for generation vs. validation.

## 5 Related Work

Specifix is a novel technique in the recently emerging area of automatic program repair, whose most important contributions we briefly review below. Dynamic invariant inference is one of the specific techniques used in Specifix; we also discuss fundamental related work in this area.

**Automatic Program Repair.** Source-code repair aims to remove buggy behavior from a program by changing its implementation. GenProg [21] is one of the first and most successful techniques for source-code repair. It uses genetic programming to mutate a faulty program into one that satisfies a given set of test cases. GenProg has been evaluated [14] using various open-source programs, showing that it can produce many non-trivial fixes. GenProg works on programs without annotations; however, it requires a regression test suite as part of its input.

Other work has applied different techniques to the problem of source-code repair, with the goal of improving the applicability and acceptability of the produced repairs; for example, by deploying machine-learning techniques [2, 12, 13], constraint-based approaches [10, 16], and finite-state abstractions [7]. These techniques also normally require a regression test suite as part of their input.

In previous work, we developed AutoFix [17, 20], an automatic tool that suggests fixes of implementations written in Eiffel and annotated with simple contracts. Contracts dispense with the need for a regression test suite, as one can be generated as needed through automatic testing. Specifix’s technique can be seen as the dual to AutoFix’s: the latter assumes contracts correct to fix implementations, whereas the former assumes implementations correct to fix contracts.

**Invariant Inference.** Invariant inference techniques learn assertions that hold for a given implementation. These techniques are naturally classified in *static* and *dynamic*. Static techniques analyze the source code to infer specification elements. Since inferring all but the simplest classes of properties is undecidable, static techniques are usually sound but incomplete. Abstract interpretation is a fundamental framework for static invariant inference [4], which has been applied in many different contexts.

SpeciFix relies instead on *dynamic* techniques for invariant inference. These summarize properties that are invariant over multiple runs of a program; their advantage over static techniques is that dynamic approaches do not require a sophisticated analytical framework and are applicable to the whole programming language: they work on anything that can be executed. While dynamic techniques provide no guarantees of soundness or completeness, they work quite well in practice. Dynamic invariant inference has been pioneered by the Daikon tool [8]. Daikon uses a pre-defined set of templates describing common relations among program variables. Much work has been done to extend and improve the Daikon approach; for example to support object-oriented features [5], and to infer complex and often complete postconditions [19]. The dynamic approach has also been applied to other kinds of specifications such as finite-state behavioral specifications [1, 6, 15, 22] and algebraic specifications [9, 11].

## 6 Conclusions and Future Work

We presented an automatic technique that fixes programming bugs by rectifying *specifications* in the form of simple contracts (pre- and postconditions). In an experimental evaluation, we ran Specifix on 44 bugs of Eiffel standard data-structure libraries. An evaluation by human programmers indicates that Specifix produced fixes of quality sufficient to be deployed for 25% of the bugs.

We now have complementary techniques to fix bugs either by changing the implementation (such as in our previous work on AutoFix [17, 20]) or by changing the specification (using Specifix presented in this paper). Therefore, the main goal of future work is to apply both fixing approaches in combination, and in particular to develop automatic heuristics to decide whether the “best” fix for a given bug involves changing implementation, specification, or both.

**Availability.** The Specifix source code, and all data and results cited in this article, are available at: <http://se.inf.ethz.ch/research/specifix/>.

**Acknowledgments.** Thanks to Alexey Kolesnichenko, Nadia Polikarpova, Andrey Rusakov, and Julian Tschannen for participating in the trial (Section 4.2).

## References

1. Ammons, G., Bodík, R., Larus, J.R.: Mining specifications. In: POPL, pp. 4–16 (2002)
2. Arcuri, A.: Evolutionary repair of faulty software. *Applied Soft Computing* 11(4), 3494–3514 (2011)
3. Ciupa, I., Pretschner, A., Oriol, M., Leitner, A., Meyer, B.: On the number and nature of faults found by random testing. *Softw. Test., Verif. Reliab.* 21(1), 3–28 (2011)

4. Cousot, P., Halbwachs, N.: Automatic discovery of linear restraints among variables of a program. In: POPL, pp. 84–96 (1978)
5. Csallner, C., Smaragdakis, Y.: Dynamically discovering likely interface invariants. In: ICSE, pp. 861–864 (2006)
6. Dallmeier, V., Lindig, C., Wasylkowski, A., Zeller, A.: Mining object behavior with ADABU. In: WODA, pp. 17–24 (2006)
7. Dallmeier, V., Zeller, A., Meyer, B.: Generating fixes from object behavior anomalies. In: ASE, pp. 550–554. IEEE (2009)
8. Ernst, M.D., Cockrell, J., Griswold, W.G., Notkin, D.: Dynamically discovering likely program invariants to support program evolution. *IEEE TSE* 27(2), 99–123 (2001)
9. Ghezzi, C., Mocci, A., Monga, M.: Synthesizing intensional behavior models by graph transformation. In: ICSE, pp. 430–440 (2009)
10. Gopinath, D., Malik, M.Z., Khurshid, S.: Specification-based program repair using SAT. In: Abdulla, P.A., Leino, K.R.M. (eds.) TACAS 2011. LNCS, vol. 6605, pp. 173–188. Springer, Heidelberg (2011)
11. Henkel, J., Reichenbach, C., Diwan, A.: Discovering documentation for Java container classes. *IEEE TSE* 33(8), 526–543 (2007)
12. Jeffrey, D., Feng, M., Gupta, N., Gupta, R.: BugFix: a learning-based tool to assist developers in fixing bugs. In: ICPC, pp. 70–79. IEEE (2009)
13. Kim, D., Nam, J., Song, J., Kim, S.: Automatic patch generation learned from human-written patches. In: ICSE, pp. 802–811. IEEE (2013)
14. Le Goues, C., Dewey-Vogt, M., Forrest, S., Weimer, W.: A systematic study of automated program repair: Fixing 55 out of 105 bugs for \$8 each. In: ICSE, pp. 3–13. IEEE (2012)
15. Lorenzoli, D., Mariani, L., Pezzè, M.: Automatic generation of software behavioral models. In: ICSE, pp. 501–510 (2008)
16. Nguyen, H.D.T., Qi, D., Roychoudhury, A., Chandra, S.: SemFix: program repair via semantic analysis. In: ICSE, pp. 772–781. IEEE (2013)
17. Pei, Y., Wei, Y., Furia, C.A., Nordio, M., Meyer, B.: Code-based automated program fixing. In: ASE, pp. 392–395. ACM (2011)
18. Polikarpova, N., Furia, C.A., Pei, Y., Wei, Y., Meyer, B.: What good are strong specifications? In: ICSE, pp. 257–266. ACM (2013)
19. Wei, Y., Furia, C.A., Kazmin, N., Meyer, B.: Inferring better contracts. In: ICSE, pp. 191–200. ACM (2011)
20. Wei, Y., Pei, Y., Furia, C.A., Silva, L.S., Buchholz, S., Meyer, B., Zeller, A.: Automated fixing of programs with contracts. In: ISSTA, pp. 61–72. ACM (2010)
21. Weimer, W., Nguyen, T., Le Goues, C., Forrest, S.: Automatically finding patches using genetic programming. In: ICSE, pp. 364–374. IEEE (2009)
22. Xie, T., Martin, E., Yuan, H.: Automatic extraction of abstract-object-state machines from unit-test executions. In: ICSE, pp. 835–838. IEEE (2006)