Automated workarounds from Java program specifications based on SAT solving

Marcelo Uva[1] · Pablo Ponzio[1,3] · Germán Regis[1] · Nazareno Aguirre[1,3] · Marcelo F. Frias[2,3]

Abstract

The failures that bugs in software lead to can sometimes be bypassed by the so-called *workarounds*: when a (faulty) routine fails, alternative routines that the system offers can be used in place of the failing one, to circumvent the failure. Existing approaches to workaround-based system recovery consider workarounds that are produced from *equivalent method sequences*, automatically computed from user-provided abstract models, or directly produced from user-provided equivalent sequences of operations. In this paper, we present two techniques for computing workarounds from Java code equipped with formal specifications, that improve previous approaches in two respects. First, the particular state where the failure originated is actively involved in computing workarounds, thus leading to repairs that are more *state specific*. Second, our techniques automatically compute workarounds on concrete program state characterizations, avoiding abstract software models and user-provided equivalences. The first technique uses SAT solving to compute a sequence of methods that is equivalent to a failing method on a specific failing state, but which can also be generalized to *schemas* for workaround reuse. The second technique directly exploits SAT to circumvent a failing method, building a state that mimics the (correct) behaviour of a failing routine, from a specific program state too. We perform an experimental evaluation based on case studies involving implementations of collections and a library for date arithmetic, showing that the techniques can effectively compute workarounds from complex contracts in an important number of cases, in time that makes them feasible to be used for run-time repairs. Our results also show that our state-specific workarounds enable us to produce repairs in many cases where previous workaround-based approaches are inapplicable.

Keywords Runtime recovery · Workarounds · SAT Solving

1 Introduction

Even in software systems that are built with high-quality standards using rigorous software development techniques, bugs still make it through to deployment. Various issues contribute to this situation: the intrinsic complexity of software, the constant adaptation and extension that software systems undergo during maintenance, and the increasing pressure to shorter time to market, among other factors. These circumstances, combined with demands for availability on software, make techniques that help systems tolerate bug-related failures highly relevant. A mechanism that has been useful for bypassing failures led to by program bugs is the so-called *workaround*: when a call to a (faulty) routine leads to a failure, alternative routines or combinations of routines that the software system offers can be used in place of the failing one, to circumvent the failure. Previous works have exploited the workaround notion to automat-

✉ Nazareno Aguirre
naguirre@dc.exa.unrc.edu.ar

Marcelo Uva
uva@dc.exa.unrc.edu.ar

Pablo Ponzio
pponzio@dc.exa.unrc.edu.ar

Germán Regis
gregis@dc.exa.unrc.edu.ar

Marcelo F. Frias
mfrias@itba.edu.ar

[1] Universidad Nacional de Río Cuarto, Río Cuarto, Argentina

[2] Instituto Tecnológico de Buenos Aires (ITBA), Buenos Aires, Argentina

[3] Consejo Nacional de Investigaciones Científicas y Técnicas (CONICET), Buenos Aires, Argentina

ically recover from run-time failures in some application domains, notably web applications [5]. However, while existing approaches compute workarounds automatically, they do so from an abstract, state machine like model of the software being considered [5,8], that needs to be manually provided, or require the user to provide equivalent alternative sequences of operations [4], from which workarounds are computed, diminishing the automation of workaround-based system recovery. In both cases, workarounds are computed from *equivalent method sequences*, either because these are produced from abstract state machine models where different paths connecting the same states are equivalent sequences [5,8], or because these are directly produced from user-provided equivalent sequences [4].

In this paper, we propose two techniques that, through the use of state-of-the-art SAT-based technology, can automatically compute workarounds directly from formal specifications accompanying Java source code in the form of JML *contracts*, thus avoiding the need for more abstract, manually built software models or user-provided alternatives to system routines. Furthermore, our techniques make the particular state in which the failure took place an active part of the workaround computation, leading to more *state-specific* workarounds. Thus, many of our computed workarounds can be used for run-time repair only in a particular situation in which a failure occurred.

Our techniques have similar requirements for their application, but differ in the actual mechanism to compute, and provide, workarounds. The first technique employs SAT solving to compute workarounds that, as usual, exploit the intrinsic redundancy of the module holding the failing routine; while this technique produces state-specific repairs, they can be generalized to produce *schemas*, which can be used to speed up the search for workarounds in other failing states. Workaround schemas are predefined sequences of methods that need to be instantiated with concrete inputs to be transformed into state-specific workarounds. Workaround schemas exploit the observation that a same sequence of methods can be useful in many different repair situations, if we vary the inputs passed to the methods. The advantage of using schemas is that, in case a schema can be applied, the underlying decision procedure (a SAT solver) only has to find inputs for the (fixed) sequence of methods, avoiding the need to perform a nondeterministic search for method sequences to employ in building workarounds. Furthermore, finding inputs for schemas can be done in parallel with the "traditional" workaround finding procedure, and returning the first workaround found by any of the running processes. As shown in our experimental assessment, schemas allow us to significantly reduce the time required to find a workaround in many cases.

The second technique directly exploits SAT solving to circumvent the failing method, automatically building a state that mimics the (correct) behaviour of this failing routine. This second technique is then closer to work on constraint-based repair, e.g. [32,36,37], although it differs in the approaches used to improve scalability (see the Sect. 6 for a comparison with these techniques).

In order to assess the applicability of the presented techniques, we develop a number of case studies based on contract-equipped collection classes and a Java library for date arithmetic. For these case studies, we produce randomly generated program state scenarios, where methods are assumed to fail, and workarounds for them, of the two kinds just described, are computed. These case studies show that the techniques can effectively compute workarounds from complex contracts in an important number of concrete state situations, in times that makes them feasible to be used for run-time repairs. Our results also show that our produced state-specific workarounds allow us to produce repairs in many cases where previous approaches are inapplicable, i.e. where no workaround based on sequence of methods equivalent to the failing routine is possible, and that workaround schemas derived from identified "concrete" workarounds can significantly improve the efficiency in finding method workarounds in new failing situations.

The remainder of this article is organized as follows. In Sect. 2, we describe some preliminaries, in particular the concept of workaround, and the Alloy and DynAlloy languages, and their associated SAT-based analyses, that will be central to our techniques. Section 3 presents our first technique that computes traditional workarounds from contracts of Java programs. Section 4 introduces our second technique, whose corresponding workarounds are obtained by directly manipulating program states, as opposed to doing so indirectly via available methods of the module being treated. Section 5 is dedicated to an experimental evaluation of the presented techniques, including an assessment of the results and the identification of potential threats to validity. Section 6 compares our presented techniques with related work, and finally, in Sect. 7, we draw some conclusions and propose some lines for further work.

2 Background

Workarounds and run-time repair The concept of *workaround* was initially defined in the context of self-healing systems [8]. Intuitively, a workaround exploits the implicit redundancy present in system modules in order to overcome a fault in the module. Given an initial state S_i, a routine m (failing when invoked in state S_i), and a desired final state S_f, a *workaround* is a procedure P composed of a sequence of other routines in the module that contains m, that leads from S_i to S_f. If the intended behaviour of a given system module is captured through a finite state machine abstraction, then a

method or routine failing in a specific state is represented by a particular transition from a source state (the initial state) to the desired target state. Workarounds composed of sequences of other routines can be systematically explored by traversing the state machine, from the initial state, attempting to reach the final state without traversing through transitions labelled with the failing routine. This is in fact the process employed for automated workaround computation presented in [6,8].

Other approaches employing workarounds (although not computing them automatically) have been developed in the context of self-healing systems. A distinguishing approach is that presented in [4], where an architecture for self-healing systems, composed of a mechanism to monitor system execution and automatically recover via rollbacks and the application of (user provided) workarounds, is introduced. The concept of workaround has been successfully applied in real software systems through the above-described approaches, with demonstrating case studies involving complex software systems such as Google Maps and Flickr [5]. Moreover, further experimental analyses have been performed, showing that the redundancy exploited by the workarounds mechanism is actually inherent to many component-based systems [7].

It is worth remarking that the mentioned workaround-based run-time recovery approaches [4–6,8] compute workarounds from equivalent method sequences. When these are produced from abstract state machine models, the level of abstraction of such models makes different paths connecting the same states to be equivalent sequences [6,8]. Other works directly require users to provide such equivalent sequences to be used as part of workarounds [4]. This implies that computed workaround sequences can be used in *any* program state, in place of the failing routine.

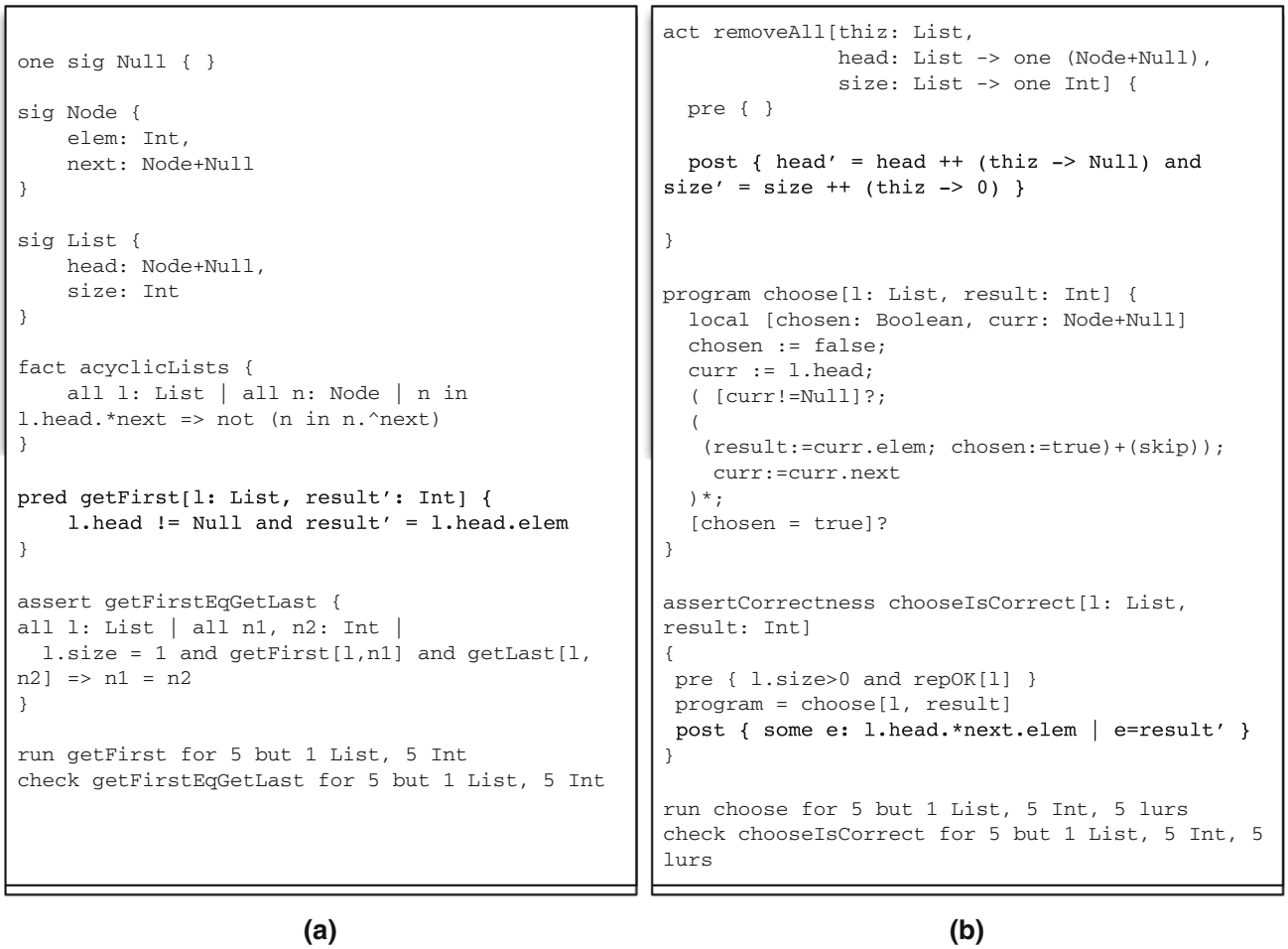
The Alloy and DynAlloy modelling languages In Alloy [20], data types are defined by *signatures*. For instance, assuming that we want to model the behaviour of linked lists, their structure can be defined through signatures `Null`, `Node` and `List` in Fig. 1a. Int (integers) is the only predefined signature. Every signature defines a set of atoms, i.e. a domain. The modifier `one` forces the corresponding signatures to have exactly one element, i.e. to be singletons, which is useful to define constants to be used in specifications. (In our case, `Null` is such a "constant".) Signatures can have fields. For instance, signature `List` has two fields, `head` and `size`. Field `head` is in fact a relation (more precisely, a function) from `List` atoms to `Node` atoms or `Null`.

Alloy also features *facts*, *predicates* and *assertions*. Facts define properties assumed to be true in the models and are written in relational logic (first-order logic with relational operators, including transitive and reflexive–transitive closures). For instance, if one would want to restrict analysis to *acyclic* lists, one may impose acyclicity via fact `acyclicLists` in Fig. 1a. In this fact, dot (`.`) is relational composition (which can be intuitively seen as a navigational operator); `*` and `^` represent reflexive–transitive and transitive closures; so, the formula expresses that, for every list l and every node n reachable from the list's head, n cannot be reached from n navigating through (one or more) "next" links.

Predicates are formulas with potentially free variables and can be used to express properties, and in particular to capture operations. For instance, predicate `getFirst` in Fig. 1a captures the "get first" operation on lists. Finally, assertions are *intended* properties, i.e. properties that should be implied by facts, but must be checked for. For instance, one may check that, when lists have size one, `getFirst` and `getLast` return the same value, expressed in assertion `getFirstEqGetLast` in Fig. 1a.

Both predicates and assertions can be subject to automated analysis using Alloy Analyzer, a tool that employs off-the-shelf SAT solvers to build satisfying instances of predicates or violating instances for assertions, under user-provided scopes. Figure 1a shows some sample commands running Alloy Analyzer. These will use SAT solving to build instances involving at most one list, five nodes and using integers with bit width 5, that satisfy `getFirst`, and violate `getFirstEqGetLast`, respectively. In the first case, it will serve as a sample execution of `getFirst`. In the second case, if a violation is found, it exhibits a problem regarding a property that the user thought it would be valid; if on the other hand no counterexample is found, it helps gaining confidence on the correctness of the model and the validity of the property (although it is clearly not a proof of validity).

Alloy is a convenient, simple and expressive language for building *static* models of software. Dealing with *dynamic* models, i.e. models that capture system execution elements such as state change, is less straightforward. DynAlloy [13] is an extension of Alloy that incorporates convenient constructs to easily capture state change. DynAlloy's syntax and semantics is based on dynamic logic. The language extends Alloy with *basic actions*, *programs* and *partial correctness assertions*. Basic actions are *defined* through pre- and post-conditions. For instance, an action that removes all elements of a list can be defined as `removeAll` in Fig. 1b. This atomic action updates the head and size of the list, using relational overriding (++). A few things are worth noticing. First, action `removeAll` has `List`'s fields `head` and `size` as explicit parameters, instead of being attributes of argument `thiz` (we avoid `this` because it is a reserved word in Alloy). This is a necessary part of our mutable model of the heap (see [14] for details). Second, as opposed to Alloy predicates, which require parameters for poststate variables, these are implicit in DynAlloy's actions. Indeed, notice that the postcondition refers to primed variables `head'` and `size'`, which are not listed explicitly as action arguments. Moreover, when a primed variable is not mentioned in the postcondition, it is



```
one sig Null { }

sig Node {
    elem: Int,
    next: Node+Null
}

sig List {
    head: Node+Null,
    size: Int
}

fact acyclicLists {
    all l: List | all n: Node | n in
l.head.*next => not (n in n.^next)
}

pred getFirst[l: List, result': Int] {
    l.head != Null and result' = l.head.elem
}

assert getFirstEqGetLast {
all l: List | all n1, n2: Int |
  l.size = 1 and getFirst[l,n1] and getLast[l,
n2] => n1 = n2
}

run getFirst for 5 but 1 List, 5 Int
check getFirstEqGetLast for 5 but 1 List, 5 Int
```

```
act removeAll[thiz: List,
              head: List -> one (Node+Null),
              size: List -> one Int] {
  pre { }

  post { head' = head ++ (thiz -> Null) and
size' = size ++ (thiz -> 0) }

}

program choose[l: List, result: Int] {
  local [chosen: Boolean, curr: Node+Null]
  chosen := false;
  curr := l.head;
  ( [curr!=Null]?;
  (
   (result:=curr.elem; chosen:=true)+(skip));
    curr:=curr.next
  )*;
  [chosen = true]?
}

assertCorrectness chooseIsCorrect[l: List,
result: Int]
{
 pre { l.size>0 and repOK[l] }
 program = choose[l, result]
 post { some e: l.head.*next.elem | e=result' }
}

run choose for 5 but 1 List, 5 Int, 5 lurs
check chooseIsCorrect for 5 but 1 List, 5 Int, 5
lurs
```

(a) **(b)**

Fig. 1 Alloy and DynAlloy specifications for linked lists

assumed to be left unchanged by the action; that is, variable `thiz` (the list object to which `removeAll` is applied) is not changed by this atomic action. DynAlloy programs are built using assignment (`:=`), skip, tests (guarded skip actions, `[expr]?`) and atomic actions as base cases, combined using sequential composition (`;`), nondeterministic choice (+) and iteration (*). A sample program that nondeterministically returns some element of a linked list is program `choose` in Fig. 1b. DynAlloy programs can be equipped with partial correctness assertions. For instance, one may specify the intended behaviour of the `choose` program as a partial correctness assertion, as illustrated in Fig. 1b, where we assume `repOK` to be a provided Alloy predicate characterizing the representation invariant of lists (e.g. acyclicity). DynAlloy programs are subject to SAT-based analysis, via a translation into Alloy [13]. They can be run (i.e. producing instances that correspond to program executions), and when they are equipped with partial correctness assertions, they can be verified against their specifications. For instance, the first command in Fig. 1b produces an execution of `choose` on a list with at most five nodes with at most five iterations; the second command checks whether *every* terminating execution of at most five iterations of `choose`, on valid and nonempty lists with at most five nodes and integers of bit width 5, returns an element of the list.

Alloy and DynAlloy are sufficiently expressive to capture Java programs and JML specifications and have been used as intermediate languages for various analyses, including bounded verification and test generation of JML-annotated Java programs [2,15,16] (although the SAT-based analysis of Alloy/DynAlloy is intrinsically incomplete). Our translation is based on [15,16] and relies on symmetry breaking and *tight field bounds* as optimizations. More precisely, we use the symmetry breaking technique introduced in [15,16], which automatically builds predicates that force canonical orderings in heap allocated structures, allowing the analysis to remove structures which are isomorphic to others already considered. Tight field bounds, on the other hand, are used to reduce the number of variables and clauses in the propositional encodings of the memory heap, for Java program analysis [15,16]. They are automatically computed from *assumed* properties, such as preconditions and invari-

ants, and are employed to restrict structures in states that are assumed to satisfy such properties. These optimizations are crucial to our analysis' efficiency, especially because we use the encoding for numerical data types originally introduced in [2] (extended to support some Alloy functions, notably cardinality), enabling us to support increased precision in numerical characterizations of Java basic data types. We refer the reader to [2,15,16] for further details.

3 Computing workarounds from program specifications

Let us now turn our attention to our first technique for computing automated workarounds for Java program specifications, employing the SAT-based automated analysis described in the previous section. The approach exploits the translation of JML contracts of Java programs into DynAlloy, and the bounded iteration (*) and nondeterministic choice (+) operators from this language, to build a partial correctness assertion involving a (nondeterministic) program, whose counterexamples correspond to workarounds.

The overall approach works as follows. Let C be a class, and $m_1, m_2, \ldots, m_k$ the public methods in C. Each method m_i is accompanied by its pre- and postcondition in JML, say pre_{m_i} and $post_{m_i}$, respectively. Notice that, as explained in the previous section, from the JML formulas corresponding to the contract of m_i, we can obtain corresponding Alloy formulas, using the translation embedded in TACO [16]. This process leads to Alloy formulas $pre^A_{m_i}$ and $post^A_{m_i}$. According to DynAlloy's syntax, we can, with these formulas, define a DynAlloy atomic action

$$a_i : \texttt{act}\ a_i\ \left\{\texttt{pre}\ \left\{pre^A_{m_i}\right\}\ \texttt{post}\ \left\{post^A_{m_i}\right\}\right\}.$$

Notice that the behaviour of DynAlloy atomic action a_i is *defined* by its pre- and postcondition, i.e. it is assumed that a_i behaves exactly as its specification prescribes. Now, given actions $a_1, a_2, \ldots, a_k$, corresponding to the translation of methods $m_1, m_2, \ldots, m_k$ into DynAlloy, we can build the DynAlloy program

$$(a_1 + a_2 + \cdots + a_k) *.$$

According to the semantics of nondeterministic choice and iteration, this program represents *all* sequential compositions of actions $a_1, a_2, \ldots, a_k$, and consequently, of methods $m_1, m_2, \ldots, m_k$.

Now, let us suppose that method m_i fails at run-time, in a concrete program state s_i. Again, we can capture state s_i as an Alloy predicate s^A_i, as shown in the previous section. Thus, we have all the elements to construct the following partial correctness assertion:

$$\left\{s^A_i\right\}\ (a_1 + a_2 + \cdots + a_{i-1} + a_{i+1} + \cdots + a_k) *\ \left\{\neg post^A_{m_i}\right\}$$

which can be automatically analysed using DynAlloy Analyzer. A counterexample of the above assertion would consist of a sequence of Alloy states $s_{A_0}, \ldots, s_{A_j}$ such that:

- s_{A_0} is state s^A_i;
- there is a sequence $a_{p(1)}; a_{p(2)}; \ldots; a_{p(j)}$ of operations such that $\langle s_{A_i}, s_{A_{i+1}}\rangle$ are related by $a_{p(i)}$ transition relation; and
- s_{A_j} is a state s^A_f that does *not* satisfy $\neg post^A_{m_i}$, i.e. that satisfies $post^A_{m_i}$.

Taking into account that s^A_i and $post^A_{m_i}$ are Alloy representations of state s_i and the postcondition of method m_i, respectively, such counterexample is indeed a workaround: it provides a sequence of actions, representing methods of class C, that take the system from state s_i to a state that satisfies $post_{m_i}$. Moreover, if DynAlloy Analyzer does not find a counterexample to the above assertion, within a provided scope, it is guaranteed that there are no workarounds in that scope (with workarounds understood as simple sequences of other methods, not more complex programs). It is important to notice the key role that state s^A_i plays in this workaround computation approach. This "pre" state drives the search for a sequence of methods that leads to a state satisfying the postcondition of interest. In fact, having this pre-state fixed improves the scalability of our DynAlloy analysis (checking programs with a high degree of nondeterminism such as the above is costly, especially for preconditions admitting many possible initial states). Moreover, this mechanism makes our workarounds *state specific*, in the sense that they allow us to recover from a failure of m_i in state s^A_i and not necessarily in other program states. As we will show later on, this more general kind of workaround enables us to recover in many situations in which workarounds based on equivalent sequences are unavailable.

Dealing with parameterized methods When looking for a workaround involving methods that receive parameters, we have an additional problem, namely how to choose appropriate values to pass as parameters so that these lead to workarounds. To do so, we define atomic actions that nondeterministically assign a value to a variable. For integer-typed variables such an action is defined as follows:

```
act nonDetAssign[x: Int] {
    pre { }
    post { x' in Int }
}
```

Then, if a method `m(int i)` is involved when attempting to build workarounds for another method, it will participate

in the iteration of nondeterministic choice of methods, as program: `nonDetAssign[i] ; m[i]`. Notice that this nondeterministic assignment is inside the iteration `*`, to allow for the possibility of using `m[i]` more than once, with different parameters. Also, in this example we are using Alloy's `Int` signature, for illustration purposes. In our case studies, we use the custom-built signatures for Java precision integers defined in [2].

An Example Consider a simple Java implementation of tuples, with methods:

- `setFirst(int value)`,
- `setSecond(int value)` and
- `swap()` (swaps first and second elements of a tuple).

Suppose that method `setFirst(3)` fails on a tuple object `t` with values `t.first: 4` and `t.second: 3`. Then, the DynAlloy program that is built to produce workarounds from is the following:

```
assertCorrectness computeWorkaround[t: Tuple+Null,
                                    first: Tuple -> one Int,
                                    second: Tuple -> one Int ]
{
  pre { t!=Null and t.first=4 and t.second=3 }
  program { local i: Int;
      ( t.swap() + (nonDetAssign[i] ; t.setSecond[i]) )*
  }
  post { !(t.first'=3 and t.second'=t.second)}
}
```

For this program, the analysis would return, for instance, the following workaround:

```
swap();
nonDetAssign(i);
setSecond(i)
```

where `nonDetAssign` assigned 3 to variable `i`. (These values can be recovered from the counterexample instance built by DynAlloy Analyzer.) The minimum scope to provide to find such workaround is two loop unrolls, one tuple and two 32-bit integers.

It is important to notice that in the above-described approach to compute workarounds, methods are seen as *atomic*, i.e. we do not take into account the *code* of method implementations, only their specifications. This simplification is made for scalability reasons, since there is no technical limitation in translating methods as programs (rather than doing so as atomic actions, as in our case).

As we mentioned previously, the technique presented in this section allows us to compute *state-specific*, or as we will call them later, "transient", workarounds. This is in contrast to workarounds involved in related techniques (see Sect. 6), which are in general *state independent*. However, as we discuss in the remainder of this section, some workarounds computed by our technique can be generalized, or "lifted", to workaround schemas, that may be reused in many different repair scenarios.

3.1 Workaround schemas

While the workarounds computed by the technique presented in this section are *state specific*, we observed that many of the obtained workarounds can be easily (and efficiently) adapted to be applied to many different repair situations, by changing the values taken by some of their inputs. Consider, for example, the `java.util.TreeSet` API given in Table 1. Assuming that the `pollFirst` method fails to establish its postcondition when the smallest element in the set is a negative number, it will fail for instance at the state `s={-1,3,7}`. A feasible workaround (discovered by our technique) to circumvent this problem is shown below:

```
// Initial state s={-1,3,7}
    int res = s.first();
    s.remove(-1);
    return res;
```

Notice that a faulty method is likely to fail again in the future, when executed with similar inputs. For example, `pollFirst` will also fail for input `s={-2,4,9}`. This time, a workaround to substitute `pollFirst` and keep this class's client running, may be the following:

```
// Initial state s={-2,4,9}
    int res = s.first();
    s.remove(-2);
    return res;
```

Observe the similarities between both workarounds above. Basically, they are both an instance of what we call a workaround schema, given below:

```
// X: int
    int res = s.first();
    s.remove(X);
    return res;
```

where `X` is an integer value (`-1` in the first case, `-2` in the second). This schema can be applied to any input set `s`, although this is not necessarily true of all schemas. Also notice that a schema is not a permanent workaround [18], as it has some variables that have to be instantiated appropriately for the given initial state.

As another example, the schema:

```
// X: int
    s.add(X);
    int res = s.remove(X);
    return res;
```

can be employed to build a transient workaround for the `contains` method, only in cases where the element we are searching for does belong to the involved set. For example,

Table 1 Excerpt of the `java.util.TreeSet` API

Return type	Method and description
Boolean	add(E e)
	Adds the specified element to this set if it is not already present
E	ceiling(E e)
	Returns the least element in this set greater than or equal to the given element, or null if there is no such element
void	clear()
	Removes all of the elements from this set
Boolean	contains(E e)
	Returns true if this set contains the specified element
E	first()
	Returns the first (lowest) element currently in this set
E	floor(E e)
	Returns the greatest element in this set less than or equal to the given element, or null if there is no such element
E	higher(E e)
	Returns the least element in this set strictly greater than the given element, or null if there is no such element
Boolean	isEmpty()
	Returns true if this set contains no elements
E	last()
	Returns the last (highest) element currently in this set
E	lower(E e)
	Returns the greatest element in this set strictly less than the given element, or null if there is no such element
E	pollFirst()
	Retrieves and removes the first (lowest) element, or returns null if this set is empty
E	pollLast()
	Retrieves and removes the last (highest) element, or returns null if this set is empty
Boolean	remove(Object o)
	Removes the specified element from this set if it is present.
NavigableSet<E>	Subset(E fromElement, boolean fromInclusive , E toElement, boolean toInclusive)
	Returns a view of the portion of this set whose elements range from fromElement, inclusive, to toElement, exclusive
SortedSet<E>	subset(E fromElement, E toElement)
	Returns a view of the portion of this set whose elements range from fromElement to toElement

the following is a workaround for a faulty invocation of the `contains` method, with inputs `s={-2,3,9}` and `e=3` (which we also denote by `{-2,3,9}.contains(3)`):

```
// s={-2,3,9}, e=3
    s.add(4);
    int res = s.remove(4);
    return res;
```

Notice that this workaround consists of adding an element to a set that does not already contain it, so that removing the element from the set (and restoring the set to its former state) and returning true have the same behaviour as that expected for `{-2,3,9}.contains(3)`.

In order to improve workaround discovery using our technique presented above, we propose to lift the transient workarounds discovered by our approach for a faulty method to *schemas*. These schemas can be used to circumvent failures produced at a later time by the same method (see Sect. 3.2). As shown in our experimental results, the use of schemas allows our approach to compute workarounds faster in such situations, as instantiating schemas to build workarounds involves querying a SAT solver on how to choose the appropriate nonprimitive variables (the `X`s in the schemas above). This is a much easier problem to solve, compared to solving the general workaround finding problem presented in the beginning of Sect. 3, as the latter involves a significant degree of nondeterminism in choosing the right actions in the right order (the `+` and `*` in the partial correctness assertion built by our approach).

3.2 Creating and exploiting workaround schemas

Creating a schema from a transient workaround calls for a generalization process of some kind. In our case, we choose a very simple process that involves replacing all the primitive constants by variables of the corresponding type. The reason is that generating schemas from workarounds must be very efficient, to make it worthwhile as a mechanism for saving time in producing new workarounds.

As an example of our schema generation process, consider the following transient workaround, used previously as a recovery for `{-1,3,7}.pollFirst()`:

```
// Initial state s={-1,3,7}
    int res = s.first();
    s.remove(-1);
    return res;
```

This workaround can be lifted to the following schema:

```
// X: int
    s.add(X);
    bool res = s.remove(X);
    return res;
```

In order to repair a failure of `pollFirst` for another input, say `{-2,4,9}.pollFirst()`, we encode the above schema as the following DynAlloy program:

```
assertCorrectness instanceSchema[s: TreeSet,
                                 ...
                                 res: Int,
                                                ] {
  pre { prePollFirst[s] and s={-2,4,9} }
  program { local X: Int;
    nonDetAssign[X];
    add[s,X];
    remove[s,X,res]
  }
  post { !postPollFirst[s,res] }
}
```

and use a SAT solver to fill in the X's. As seen in the previous section, an execution of the above program could set X to -2, which results in the workaround below:

```
// Initial state s={-2,4,9}
    int res = s.first();
    s.remove(-2);
    return res;
```

Notice how solving the above program is much cheaper than solving the general program for finding a transient workarounds:

$$\{s_i^A\}\ (a_1 + a_2 + \cdots + a_{i-1} + a_{i+1} + \cdots + a_k) * \{\neg post_{m_i}^A\}$$

as the solver only has to choose appropriate values for the primitive variables. On the other hand, the latter program contains a high degree of nondeterminism in the selection of actions and a bounded iteration that is much more expensive to represent as a propositional formula because it encodes a significant number of feasible intermediate states.

Now, clearly different schemas may be collected as generalizations of previously obtained workarounds. When a set $sc_1, sc_2, \ldots, sc_j$ of schemas is available for a given method, we can capture the nondeterministic choice of all of them as a DynAlloy program:

$$\{s_i^A\}\ sc_1 + sc_2 + \cdots + sc_j\ \{\neg post_{m_i}^A\}$$

Finding workarounds using this program is still much cheaper for the SAT solver than the general program for finding transient workarounds. This is mainly due to two reasons. Firstly, this simpler program does not involve bounded iteration. Secondly, in practice we can restrict the set of available schemas to build workarounds for a given action to just a few, leading to less nondeterminism (i.e. $j \ll k$).

Schemas are incorporated into the process of finding workarounds as shown in Fig. 2. We store previously discovered schemas for each method in a schema database. When a method fails for a given input, two tasks are run in parallel: (i) one attempts to produce a workaround from previously stored schemas using the nondeterministic program above

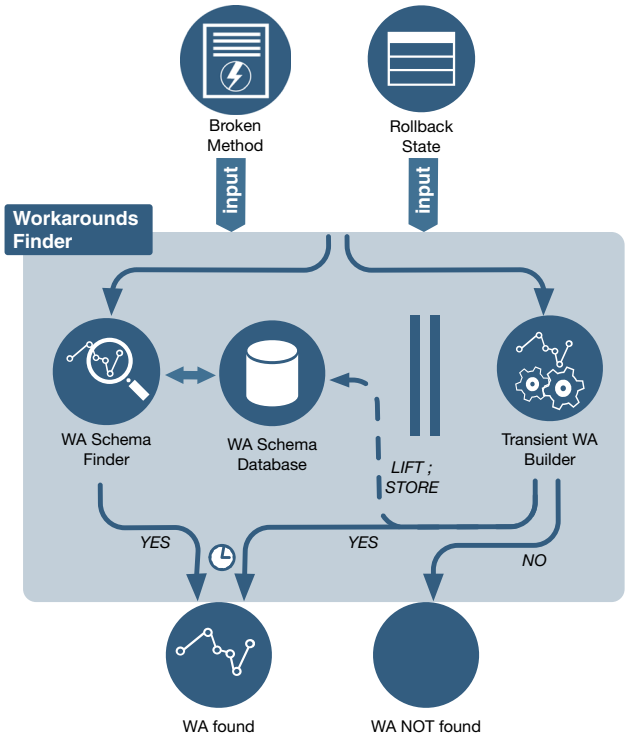


Fig. 2 Our approach for finding workarounds

and (ii) the other runs the general transient workaround finding program. If task (i) finishes first producing a workaround as an instance of a schema, then we stop the other job and return the instantiated schema as a recovery. If on the other hand (ii) finishes first, then we stop the other job, return the produced workaround as the proposed recovery and *promote* this workaround to a schema, as described in the previous subsection (as long as this schema was not already present in the database).

As described later on in this paper, the use of schemas allowed us to repair many faulty scenarios with a significant speedup with respect to the transient workaround finding approach (see Sect. 5.3).

3.3 Applying workarounds in practice

The workarounds computed by our approach can be used straightforwardly in the context of previous self-healing approaches, like the one presented in [4]. The approach put forward therein, called ARMOR, proposes instrumenting calls to the programs that are being subject to workarounds. The instrumentation works by storing the system state before every call to a method of a module for which workarounds are going to be provided, and checks whether the state after the call satisfies the method postcondition (in our case, JML postconditions are automatically translated to run-time assertions). In case, it does not, ARMOR rollbacks to the system

state before the call. At this point, our approach can be invoked to find a feasible workaround for the method in the rollback state that is then passed to ARMOR for it to execute the workaround. The result of executing the workaround produces a valid state that can be passed to the system to continue its execution as if the failure never occurred.

Of course, using ARMOR produces an overhead in run-time, 194% in the worst case reported in [4], for a system that terminates in a few seconds. We believe that the benefits of using ARMOR combined with our approach to compute workarounds to repair run-time failures greatly outweigh the run-time overhead of the approaches.

The technique that we will introduce in Sect. 4 tackles the workaround computation problem in a different way, by resorting to the use of SAT solving to directly build a recovery program state, rather than a recovery sequence of methods.

4 Program state repair using SAT

The technique in the previous section computes standard workarounds, and differs from other workaround approaches in that it applies to contract specifications at the level of detail of source code, and it computes workarounds fully automatically. In this section, we present a different approach, which attempts to repair the failing routine by directly producing the expected post state using the specification of the routine and SAT solving.

While this technique has in principle the same constraints as the previous one, i.e. that contracts must be available for the programs being subject to the analysis, it can be better explained (and exploited) through the use of *abstraction functions*. Data representations often attempt to capture more abstract models. For instance, binary search trees are often used as an implementation of sets of elements. The abstraction function is part of a data representation specification, which indicates how concrete data representation instances map to the corresponding abstract elements. Going back to our example of binary search trees, the abstraction function would indicate, for each binary search tree, which is the set is represents (i.e. it essentially returns the set of values held in the AVL). Contract languages such as JML [9] support the definition of model variables and abstraction functions; abstraction functions can also be captured directly in Java, as shown in [24]. In our case, to simplify the presentation, we will use Alloy to express abstraction functions. For instance, the abstraction function of binary search trees, we just referred to, is captured in Alloy (in this case, using a predicate) as follows:

```
pred absFunction[thiz: Tree,
                 root: Tree -> one (Node+Null),
                 left: Node -> one (Node+Null),
                 right: Node -> one (Node+Null),
                 key: Node -> one Int,
                 result: set Int] {
    result = thiz.root.*(left+right).key
}
```

So, let us assume that, besides the pre- and postconditions for all class methods, and the class invariant, we have the Alloy specification of the abstraction function. (This may be given in JML and then translated to Alloy.) Now, as in the previous technique, assume that method m_i breaks at run-time in a concrete program state s_i. We would want to recover from this failure, reaching a state s_f that satisfies the postcondition $post_{m_i}(s_i, s_f)$. (Notice that the postcondition in languages such as JML and DynAlloy is actually a postcondition *relation* that indicates the relationship between precondition states and postcondition states.) We can build a formula that characterizes these "recovery" states, as follows:

```
pred recoveryStates[s_f: State] {
    some x, y | alpha[s_i, x] and alpha[s_f, y] and
                post_m_i [x, y] and repOK[s_f]
}
```

where `repOK` is the class invariant translated to Alloy, `post_m_i` is the postcondition relation of method m_i, translated to Alloy from JML, and `alpha` is the abstraction function. Finding satisfying instances of this predicate will produce valid poststates, in the sense that they satisfy the class invariant, that mimic the execution of method m_i.

An Example Consider a binary search tree representation of sets. Assume that the JML invariant for binary search trees and the JML postcondition of method `remove` have already been translated into Alloy predicates `repOK` and `post_rem`, respectively. These would look as follows:

```
pred repOK[thiz: Tree,
           root: Tree -> one(Node+Null),
           left: Node -> one(Node+Null),
           right: Node -> one (Node+Null),
           key: Node -> one Int] {
all n : Node | n in thiz.root.*(left + right) implies
(n.key != null and
(no (((n.left).*(left+right)&(n.right).*(left+right))-Null))
and (n !in n.^(left+right)) and
(all m: Node | m in n.left.*(left+right) implies
 n.key>m.key)  and
(all m: Node | m in n.right.*(left+right) implies
 m.key>n.key)
)
}

pred post_rem[elems, elems': set Int, elem: Int] {
    elem in elems and elems' = elems - elem
}
```

Now, consider the left-hand side binary search tree in Fig. 3, and suppose that method `remove(x)` failed on this tree, for `x = 3`. By looking for models of the following Alloy predicate:

```
pred recoveryStates [thiz: Tree,
                     root,root': Tree -> one (Node+Null),
                     left,left': Node -> one (Node+Null),
                     right,right': Node -> one(Node+Null),
                     key,key': Node -> one Int ] {
thiz = T0 and root = (T0->N0) and
left = (N0->N1)+(N1->N3)+(N2->Null)+(N3->Null)+(N4->Null)
and right = ... and ... key = ... and ...
some x, y : set Int |
     absFunction[thiz,root,left,right,key,x] and
     absFunction[thiz',root',left',right',key',y] and
     post_rem[x, y, 3]
}
```

we will be searching for a valid binary search tree that represents the set resulting from removing 3 from the left-hand side tree of Fig. 3. The right-hand side binary tree in Fig. 3 is

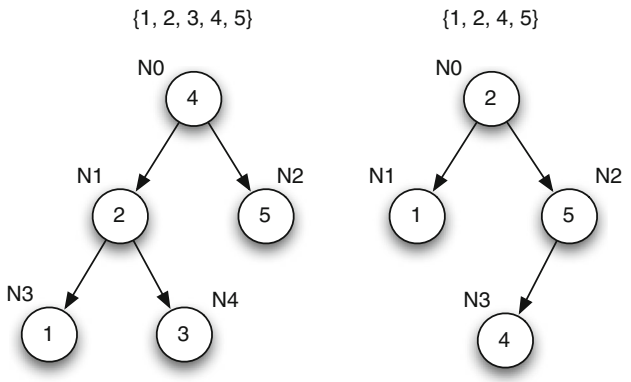


Fig. 3 Two binary search trees, and the sets they represent

an instance satisfying the predicate. Notice how this returned structure does not perform the expected change that a removal method, of a leaf in this case, would produce. But as far as the abstract data type instance that the structure represents, this resulting structure is indeed a valid result of removing key 3.

Predicate `recoveryStates` above makes some simplifications, for presentation purposes. First, it uses Alloy `Int` signature, whereas in our experiments we use a Java precision integer specification. Second, notice the use of higher-order existential quantification (`some x, y: set Int`). Such quantifications are *skolemized* for analysis (a "one" signature declares `x` and `y` as `set Int` fields, which are then used directly in predicate `recoveryStates`), a standard mechanism to deal with existential higher-order quantification in Alloy, since Alloy Analyzer does not directly support it (see [20] for more details). Finally, and more importantly, two elements are also part of `recoveryStates`, though not explicitly mentioned in the predicate. One is the addition of an automatically computed *symmetry breaking* predicate, as put forward in [15,16], which forces a canonical ordering in the structures and has a substantial impact in analysis. Second, we use *tight bounds* [15,16] computed from class invariants (these reduce propositional state representations by removing propositional variables that represent field values deemed infeasible by the invariants) to constrain postcondition states, since these states are assumed to satisfy the corresponding invariants, as shown in the above predicate.

Again, notice how the specific state where the failure originated is an important part of our workaround computation (lines 6 and 7 of predicate `recoveryStates` "fix" the pre-state for the workaround computation). In fact, for this particular technique a fixed pre-state is a *necessary* part, since the solver needs it to be fixed to attempt to find a repairing poststate.

5 Evaluation

Our evaluation consists of an experimental assessment of the effectiveness of the two presented techniques for automatically computing workarounds and repairing faulty states, and an analysis of how our state-specific workarounds improve previous works in terms of repairability. The evaluation of the effectiveness of the techniques is based on the following benchmark of collection implementations (accompanied by their corresponding JML contracts including requires/ensures clauses, loop variant functions and class invariants):

- two implementations of interface `java.util.List`, one based on singly linked lists, taken from [15], the other a circular doubly linked list taken from `AbstractLinkedList` in Apache Commons.Collections;
- three alternative implementations of `java.util.Set`, one based on binary search trees taken from [34], another based on AVL trees taken from [3], and the red–black trees implementation `TreeSet` from `java.util`; and
- one implementation of `java.util.Map`, based on red–black trees, taken from class `TreeMap` in `java.util`.

This benchmark is complemented with the analysis of a Java library, namely library `JodaTime` for date arithmetic.

The comparison of our state-specific workarounds with previous works in terms of repairability is made on classes from `java.util` and `Graphstream`, based on the experiments presented in [18]. More precisely, we extend the set of classes employed in the evaluation of [18], namely `Stack` from `java.util` and the six classes of `Graphstream`, with a class from `java.util` (`TreeSet`) and a class from `Graphstream` (`FixedArrayList`).

All the experiments were run on a PC with 3.40 GHz Intel(R) Core(TM) i5-4460 CPU, with 8 GB of RAM. We used GNU/Linux 3.2.0 as the OS. The workaround repair prototypes together with the specifications used for the experiments can be found in [1]. Experiments can be reproduced following the instructions provided therein. Also, further experimental data are presented in [1].

5.1 Effectiveness of the techniques

In order to assess the effectiveness of our workaround techniques, we artificially built repair situations, i.e. situations in which it was assumed that a method *m* has failed. These situations were randomly and automatically constructed, using Randoop [27]. For each data structure interface, we ran Randoop for 1 h, producing 116,000 list traces, 136,000 set

traces and 138,000 map traces, leading to the same number of instances of the corresponding data structure. We sampled one every 1000 structures (number 1000, number 2000, number 3000, etc., since Randoop tends to produce structures of increasing size due to its feedback driven generation policy based on randomly extending previously obtained sequences [27]), obtaining 116 lists, 136 sets and 138 maps. We proceeded in a similar way for class `TimeOfDay` of `JodaTime`, producing 50 scenarios. For each method *m* in the corresponding class, we assumed it failed on each of the structures, and attempted a workaround-based repair using the remaining methods. So, for instance, for method `removeLast` from `List`, we attempted its workaround repair using the remaining 32 methods of the class, in 116 different repair situations. Notice that for the first technique, and since workarounds are computed at the interface level from method specifications (not implementations), we have one experiment per interface (e.g. AVL and TreeSet set implementations are equivalent from the specification point of view, so computing workarounds for one implementation also work for the others). For the second technique, on the other hand, each implementation leads to different experiments, since the technique depends on the structure implementation.

We summarize the experimental results of the evaluation of the first technique in columns Lists, Sets and Maps of Tables 2, 3 and 4. Tables report:

- method being fixed (the fix is computed from the iteration of nondeterministic choice of remaining methods);
- total time, the time spent in fixing *all* 100 faulty situations; time is reported in h:mm:ss format;
- average repair time, i.e. the time that in average it took to repair *each* faulty situation; again, time is reported in h:mm:ss format;
- average workaround length, i.e. number of routines that the found workaround had, in average; and
- number of timeouts, i.e. faulty situations that could not be repaired within 10 min.

It is important to remark that, in the tables, we only count the repairs that actually ended within the timeout, to compute the total and average repair times. Also, each table reports, for the corresponding structure, the minimum, maximum and average size for the randomly generated structures (see table headings).

Regarding the second technique, we evaluated its performance on producing recovery structures on the same scenarios as the first technique. Recall that scenarios were produced using, for all implementations of the same data type, the same interface, so these are shared among different implementations of the same data type. The timeout is set in 10 min. Results are reported in the remaining columns of Tables 2, 3 and 4. Notice that for this technique we do not report workaround size, since it "repairs" the failing method by directly building a suitable postexecution state. Regarding the results of both techniques on the `JodaTime` date arithmetic library, these are summarized in a single table (Table 5), for varying bit widths in numeric data types.

Assessment Notice that our first technique performed very well on the presented experiments. Many methods could be repaired within the timeout limit of 10 min (see the very small number of timeouts in the tables) and with small traces; in fact, the great majority could be repaired by workarounds of size 1 (i.e. by calling only one alternative method), and some with workarounds of size up to 3, confirming the observations in [7]. It is important to observe that some methods are difficult to repair. For instance, method `clear`, that removes all elements in the corresponding collection, cannot be solved alternatively by *short* workarounds. In fact, this method requires performing as many element removals as the structure holds, which went beyond the 10-min timeout in all cases. This technique also performed well on our arithmetic-intensive case study. Notice that, as bit width is increased, analysis becomes slightly more expensive, but more workarounds arise (since some workarounds are infeasible with smaller bit widths). Our second technique features even more impressive experimental results. Most of the repair situations that we built with Randoop were repaired using this technique. This included repairing methods that, from many program states, could not be repaired by the first technique.

These techniques scaled for the evaluated classes beyond some SAT-based analysis techniques, e.g. for test generation or bounded verification [2,16]. The reason for this increased scalability might at first sight seem obvious, since the analysis starts from a concrete program state (a distinguishing characteristic of our computed workarounds). However, the nondeterminism of the (DynAlloy) program used in the computation of the workarounds, formed by an iteration of a nondeterministic choice of actions (representing methods), makes the analysis challenging and the obtained results, in terms of efficiency, relatively surprising. A technical detail that makes the results interesting is the fact that the translation from Java into Alloy and DynAlloy that we use encodes numerical data types with Java's precision. That is, integers are encoded as 32-bit integers (in the case of `JodaTime`, where arithmetic is heavily used, we assessed our techniques with different bit widths), as opposed to other works that use Alloy integers (very limited numerical ranges). The approach is that presented in [2], extended to make some Alloy functions, notably cardinality (#), work on these numerical characterization of Java basic data types.

As we have stated in the paper, our approach requires formal specifications. Our case studies demanded different specification overheads. Data structure implementations

Table 2 Workaround computation for lists

`Lists`: 116 structs.; min. size: 6; max. size: 25; avg. size: 14.85										
`Singly Lkd Lists`: 116 structs.; min. size: 6; max. size: 25; avg. size: 14.85										
`Abst. Lkd Lists`: 116 structs.; min. size: 6; max. size: 25; avg. size: 14.85										
Method to fix	`Lists`				`Singly Lkd Lists`			`Abst. Lkd Lists`		
	Total time	Avg rep. time	Avg wa.	# TOs	Total time	Avg rep. time	# TOs	Total time	Avg rep. time	# TOs
add	0:36:06	0:0:18	1	0	0:08:01	0:0:04	0	0:11:18	0:0:05	0
addfirst	0:36:05	0:0:18	1	0	0:08:09	0:0:04	0	0:11:20	0:0:05	0
clear	19:20:00	–	–	116	0:07:50	0:0:04	0	0:10:22	0:0:05	0
contains	0:36:16	0:0:18	1	0	0:07:44	0:0:04	0	0:10:36	0:0:05	0
get	0:36:25	0:0:18	1	0	0:07:42	0:0:04	0	0:11:19	0:0:05	0
getfirst	0:36:14	0:0:18	1	0	0:09:52	0:0:05	0	0:11:11	0:0:05	0
indexof	0:35:18	0:0:18	1	0	0:09:20	0:0:04	0	0:12:23	0:0:06	0
isempty	0:36:05	0:0:18	1	0	0:07:40	0:0:04	0	0:11:12	0:0:05	0
lastindexof	0:35:29	0:0:18	1	0	0:09:00	0:0:04	0	0:12:23	0:0:06	0
offer	0:36:18	0:0:18	1	0	0:08:16	0:0:04	0	0:11:17	0:0:05	0
peek	0:36:35	0:0:18	1	0	0:08:00	0:0:04	0	0:11:23	0:0:05	0
poll	0:36:14	0:0:18	1	0	0:08:32	0:0:04	0	0:11:31	0:0:05	0
pop	0:36:07	0:0:18	1	0	0:08:24	0:0:04	0	0:10:28	0:0:05	0
push	0:36:25	0:0:18	1	0	0:08:33	0:0:04	0	0:11:26	0:0:05	0
remove	0:36:05	0:0:18	1	0	0:08:40	0:0:04	0	0:10:58	0:0:05	0
removem	1:34:34	0:0:48	1732	0	0:11:18	0:0:06	0	0:11:12	0:0:05	0
setelement	1:48:38	0:0:56	1948	0	0:07:54	0:0:04	0	0:10:42	0:0:05	0
size	0:36:24	0:0:18	1	0	0:08:04	0:0:04	0	0:10:39	0:0:05	0

Table 3 Workaround computation for sets and trees

`Sets`: 136 structs.; min. size: 11; max. size: 22; avg. size: 13.17													
`TreeSet`: 136 structs.; min. size: 11; max. size: 22; avg. size: 13.17													
`AVL Tree`: 136 structs.; min. size: 11; max. size: 22; avg. size: 13.17													
`Search Tree`: 136 structs.; min. size: 11; max. size: 22; avg. size: 13.17													
Method to fix	`Sets`				`TreeSet`			`AVL Tree`			`Search Tree`		
	Total time	Avg rep. time	Avg. wa.	# TOs	Total time	Avg rep. time	# TOs	Total time	Avg rep. time	# TOs	Total time	Avg rep. time	# TOs
add	20:40:24	0:00: 42	1	124	2:30:13	0:1:02	1	1:18:07	0:0:30	1	0:49:11	0:0:17	1
ceiling	1:37:28	0:00: 43	1	0	0:19:15	0:0:08	0	0:19:27	0:0:08	0	0:18:48	0:0:08	0
clear	1:39:44	0:00: 44	1	0	0:10:19	0:0:04	0	0:11:33	0:0:05	0	0:10:48	0:0:04	0
contains	1:39:44	0:00: 44	1	0	0:10:33	0:0:04	0	0:11:44	0:0:05	0	0:10:51	0:0:04	0
first	1:03:28	0:00: 28	1	0	1:33:50	0:0:37	1	0:57:40	0:0:21	1	0:49:26	0:0:13	2
floor	1:37:28	0:00: 43	1	0	2:01:34	0:0:49	1	1:00:19	0:0:22	1	0:27:46	0:0:12	0
higher	1:01:12	0:00: 27	1	0	1:55:17	0:0:42	2	0:58:13	0:0:21	1	1:10:47	0:0:27	1
isEmpty	0:49:52	0:00: 22	1	0	1:42:15	0:0:41	1	1:00:26	0:0:22	1	1:14:17	0:0:24	2
last	1:08:00	0:00: 30	1	0	1:19:27	0:0:30	1	0:53:02	0:0:19	1	0:35:47	0:0:15	0
lower	0:49:52	0:00: 22	1	0	1:29:02	0:0:35	1	0:57:28	0:0:21	1	1:11:47	0:0:18	3
pollFirst	1:39:44	0:00: 44	1	0	1:23:57	0:0:37	0	0:51:05	0:0:22	0	0:47:43	0:0:12	2
remove	21:56:15	0:01: 15	2	131	1:41:19	0:0:40	1	0:51:38	0:0:22	0	0:34:28	0:0:15	0

Table 4 Workaround computation for maps

`Maps`: 138 structs.; min. size: 11; max. size: 22; avg. size: 13.68							
`Tree Maps`: 138 structs.; min. size: 11; max. size: 22; avg. size: 13.68							
Method to fix	`Maps`				`Tree Maps`		
	Total time	Avg rep. time	Avg wa.	# TOs	Total time	Avg rep. time	# TOs
ceilingkey	0:48:38	0:0:21	1	0	0:34:38	0:0:15	0
clear	23:00:00	–	–	138	0:29:53	0:0:12	0
containsvalue	0:47:01	0:0:20	1	0	0:30:45	0:0:13	0
firstentry	0:51:09	0:0:22	1	0	0:31:29	0:0:13	0
get	23:00:00	–	–	138	0:29:37	0:0:12	0
higherentry	1:17:04	0:0:33	1	0	0:32:44	0:0:14	0
isempty	1:20:19	0:0:34	1	0	0:27:16	0:0:11	0
lastkey	1:20:10	0:0:34	1	0	0:30:02	0:0:13	0
lowerentry	1:17:03	0:0:33	1	0	0:32:57	0:0:14	0
polllastentry	7:26:46	0:3:14	1	0	7:54:33	0:2:55	10
put	23:00:00	–	–	138	16:36:33	0:5:55	44
remove	23:00:00	–	–	138	9:27:27	0:3:03	21

have clear abstract mathematical models, and thus, methods for these classes can be formally specified rather straightforwardly, with the aid of suitable abstraction functions. For the case of `JodaTime`, the overhead was significantly larger: we needed to manually examine the code of this case study's classes to understand the behaviour of methods, and logically capture such behaviours. The most serious overhead was in understanding (intended) software behaviour, not in writing the specifications themselves.

5.2 State-specific versus traditional workarounds

We now turn our attention to the analysis of our state-specific computed workarounds, compared to the traditional workarounds based on equivalent method sequences. As mentioned earlier, the evaluation is based on the experiments presented in [18], extended with one class for each of the involved case studies (`TreeSet` from `java.util` and `FixedArrayList` from `Graphstream`). Let us first refer to the classes evaluated in [18], i.e. `Stack` from `java.util`, and classes `Vector2`, `Vector3`, `Path` and `Edge` from `Graphstream`. For class `Stack`, we considered 141 randomly and automatically generated scenarios, following the same approach as for our previous experiments, using Randoop. For each method of the class, we assumed it failed in each scenario, and ran our first technique, to attempt to produce a state-specific method-sequence workaround. We report the number of repairs found (column # transient), total time as well as average time. We also indicate, for each method, whether it has a state-independent workaround identified by the technique in [18] (column permanent). We call our workarounds *transient* in the sense that these repair the corresponding method only in a specific state, as opposed to *permanent* workarounds, that can be used in place of the failing method in any situation. Results are given in Table 6.

We proceeded in a similar way for classes `Vector2`, `Vector3`, `Path` and `Edge`, with 200 randomly generated scenarios for `Vector2` and `Vector3`, 160 scenarios for `Path` and 150 scenarios for `Edge`. We report the results of our first technique in Tables 7, 8, 9 and 10. Again, each method analysis is also accompanied by the *permanent* information, indicating whether the corresponding method has a state-independent workaround identified by the technique in [18].

As mentioned above, we extended the classes evaluated in [18] with two further classes of the same case studies. These were assessed following the same procedure as for the other classes. For `TreeSet`, we considered the same 116 structures generated for our effectiveness evaluation; for `FixedArrayList`, we randomly generated 300 structures. The evaluation on `TreeSet` is reported in Table 11, while the evaluation on class `FixedArrayList` is reported in Table 12. Again, as for the other classes, for each method we indicate in column *permanent* whether the method has a workaround based on equivalent sequences.

Assessment The evaluation of transient workarounds compared to permanent workarounds shows that, in a significant number of cases, one is able to compute transient workarounds (i.e. state-specific workarounds) when no permanent workaround (workarounds based on equivalent sequences) is possible. Even in classes with a high degree of redundancy such as `Stack` (where almost every functionality is provided by at least two methods), there are some methods with no permanent workarounds for which our first technique is able to effectively produce state-specific repairs. About 50% of methods in classes

Table 5 Workaround computation for jodatime

Method to fix	Technique 1						Technique 2					
	`Int.16 bits`			`Int.32 bits`			`Int.16 bits`			`Int.32 bits`		
	# wa.	Total time	Avg rep.	# wa.	Total time	Avg rep.	# wa.	Total time	Avg rep.	# wa.	Total time	Avg rep.
minusHours	48	0:08:23	0:00:10	48	0:13:12	0:00:16	48	0:01:18	0:00:01	48	0:02:32	0:00:03
minusMillis	1	7:50:09	0:00:09	48	1:21:53	0:01:42	1	0:01:46	0:00:02	48	0:05:27	0:00:06
minusMinutes	9	6:31:30	0:00:10	46	1:00:00	0:00:52	9	0:01:30	0:00:01	48	0:05:37	0:00:07
minusPeriodHours	48	0:08:41	0:00:01	48	0:13:12	0:00:16	48	0:01:18	0:00:01	48	0:02:31	0:00:03
minusPeriodMillis	1	7:50:09	0:00:09	45	1:48:05	0:01:44	1	0:01:45	0:00:02	48	0:05:26	0:00:06
minusPeriodMinutes	9	6:31:32	0:00:10	46	1:01:53	0:00:54	9	0:01:34	0:00:01	48	0:04:38	0:00:05
plusHours	48	0:08:57	0:00:11	48	0:12:38	0:00:15	48	0:01:18	0:00:01	48	0:02:32	0:00:03
plusMillis	1	7:50:12	0:00:12	47	1:12:01	0:01:19	1	0:01:39	0:00:02	48	0:04:55	0:00:06
plusMinutes	29	3:13:21	0:00:09	48	0:13:00	0:00:16	29	0:01:30	0:00:01	48	0:02:51	0:00:03
plusPeriodHours	48	0:08:45	0:00:01	48	0:12:50	0:00:16	48	0:01:17	0:00:01	48	0:02:31	0:00:03
plusPeriodMillis	1	7:50:12	0:00:12	47	1:06:41	0:01:12	1	0:01:44	0:00:02	48	0:05:01	0:00:06
plusPeriodMinutes	29	3:14:41	0:00:09	48	0:12:42	0:00:15	29	0:01:31	0:00:01	48	0:02:53	0:00:03
withHourOfDay	48	0:09:21	0:00:11	48	0:13:18	0:00:16	48	0:01:07	0:00:01	48	0:02:22	0:00:02
getHourOfDay	0	8:00:00	–	0	8:00:00	–	48	0:01:07	0:00:01	48	0:02:41	0:00:03
getMillisOfSecond	0	8:00:00	–	0	8:00:00	–	48	0:01:06	0:00:01	48	0:02:39	0:00:03

Table 6 Workaround computation for stack

`Stacks`: 141 structs.; min. size: 5, max. size: 19, avg. size: 12.19						
Method to fix	Total time	Avg rep. time	Avg. wa.	# TOs	# transient	Permanent?
stack_contains	1:43:24	0:00: 44	1	1	141	NO
stack_empty	1:34:00	0:00: 40	1	1	141	NO
stack_firstElement	1:43:24	0:00: 44	1	1	141	YES
stack_peek	1:43:24	0:00: 44	1	1	141	YES
stack_pop	1:56:00	0:00: 51	1	1	141	YES
stack_push	4:51:24	0:02: 05	1	1	141	YES
vector_add	1:45:45	0:00: 45	2	1	141	YES
vector_addElement	1:52:48	0:00: 48	1	1	141	YES
vector_addIndexItem	1:45:45	0:00: 45	1	1	141	YES
vector_clear	1:34:00	0:00: 40	1	1	141	YES
vector_elementAt	1:45:45	0:00: 45	1	1	141	YES
vector_get	1:45:45	0:00: 45	1	1	141	YES
vector_get first	1:43:24	0:00: 44	1	1	141	YES
vector_insertElementAt	1:45:45	0:00: 45	1	1	141	YES
vector_lastElement	1:43:24	0:00: 44	1	1	141	YES
vector_remove	1:45:45	0:00: 45	1	1	141	YES
vector_removeAllElements	1:45:45	0:00: 45	1	1	141	YES
vector_removeElement	1:48:06	0:00: 46	1	24	117	NO
vector_removeElementAt	1:45:45	0:00: 45	1	1	141	YES
vector_removeIndex	1:45:45	0:00: 45	1	1	141	YES
vector_set	1:45:45	0:00: 45	1	1	141	YES
vector_setElement	1:45:45	0:00: 45	1	1	141	YES

Table 7 Workaround computation for Vector2

`Vector2s`: 200 structs.; min. size: NA, max. size: NA, avg. size: NA (always two components)						
Method to fix	Total time	Avg rep. time	Avg. wa.	# TOs	# transient	Permanent?
copy	00:05:05	00:00:01	1	0	200	YES
equals	00:05:16	00:00:01	1	0	200	NO
fill	00:05:10	00:00:01	1	0	200	YES
iszero	00:05:13	00:00:01	1	0	200	NO
setxy	00:05:07	00:00:01	1	0	200	YES
validcomponent	00:05:07	00:00:01	1	0	200	NO
x	00:05:15	00:00:01	1	0	200	YES
y	00:05:14	00:00:01	1	0	200	YES

`Vector2`, `Vector3`, `Path` and `Edge` do not have permanent workarounds, and we can provide transient workarounds in most of the evaluated scenarios. Classes `TreeSet` and `FixedArrayList` provide less redundancy, and thus, fewer permanent workarounds are available; most of the cases (all of them in the case of `TreeSet`, actually) can be repaired with transient workarounds. These experiments show that our state-specific workarounds have better repair capability than permanent workarounds, in the sense that these enable run-time fixes that workarounds based on equivalent sequences are unable to handle.

From the point of view of efficiency, however, a very important point to notice is that permanent workarounds are *reusable*. That is, once a state-independent workaround for a method is found, it can be used in *any* situation, so its computation cost is *amortized* as more failing scenarios arise. In fact, if we consider the work in [18], for some methods it takes in average the same amount of time to compute a per-

Table 8 Workaround computation for Vector3

`Vector3s`: 200 structs.; min. size: NA, max. size: NA, avg. size: NA (always three components)						
Method to fix	Total time	Avg rep. time	Avg. wa.	# TOs	# transient	Permanent?
copy	00:05:07	00:00:01	1	0	200	YES
equals	00:05:13	00:00:01	1	0	200	NO
fill	00:05:07	00:00:01	1	0	200	YES
iszero	00:05:09	00:00:01	1	0	200	NO
setxyz	00:05:12	00:00:01	1	0	200	YES
validcomponent	00:05:21	00:00:01	1	0	200	NO
x	00:05:26	00:00:01	1	0	200	YES
y	00:05:24	00:00:01	1	0	200	YES
z	00:05:26	00:00:01	1	0	200	YES

Table 9 Workaround computation for path

`Paths`: 160 structs.; min. size: 2, max. size: 15, avg. size: 8.93						
Method to fix	Total time	Avg rep. time	Avg. wa.	# TOs	# transient	Permanent?
contains_edge	00:06:37	00:00:02	1	0	160	NO
contains_node	00:06:38	00:00:02	1	0	160	NO
equals	00:06:39	00:00:02	1	0	160	NO
getedgecount	00:06:30	00:00:02	1	0	160	YES
getnodecount	00:06:30	00:00:02	1	0	160	YES
isempty	00:06:35	00:00:02	1	0	160	NO
pathsize	00:06:30	00:00:02	1	0	160	YES

Table 10 Workaround computation for edge

`Edges`: 150 structs.; min. size: NA, max. size: NA, avg. size: NA (always join two vertices)						
Method to fix	Total time	Avg rep. time	Avg. wa.	# TOs	# transient	Permanent?
addattribute	00:05:37	00:00:02	1	0	150	YES
addattributes	00:11:54	00:00:02	2	0	150	NO
changeattribute	00:05:32	00:00:02	1	0	150	YES
clearattributes	00:41:50	00:00:04	3.478	58	92	NO
getattribute	00:05:42	00:00:02	1	0	150	YES
getnode0	00:05:30	00:00:02	1	0	150	YES
getnode1	00:05:31	00:00:02	1	0	150	YES
getopposite	00:31:48	00:00:02	1	120	30	NO
getsourcenode	00:05:27	00:00:02	1	0	150	YES
gettargetnode	00:05:31	00:00:02	1	0	150	YES
isdirected	00:23:57	00:00:02	1	81	69	NO
isloop	00:23:54	00:00:02	1	81	69	NO
removeattribute	00:05:33	00:00:02	1	0	150	NO
setattribute	00:05:34	00:00:02	1	0	150	YES

manent workaround, that it takes our technique to compute a transient, state-specific one.

5.3 Assessment of workaround schemas

In this section, we experimentally assess the advantages of exploiting workaround schemas with respect to transient workarounds. Table 13 shows examples of transient workarounds found by our approach (second column), the schemas obtained from the workarounds (third column) and an informal description of when the schemas can be applied. We chose the `java.util.TreeSet` case study because it allows us to illustrate the different kinds of transient workarounds and schemas that can be discovered using our

Table 11 Workaround computation for TreeSet

`TreeSets`: 136 structs.; min. size: 11, max. size: 22, avg. size: 13.16						
Method to fix	Total time	Avg rep. time	Avg. wa.	# TOs	# transient	Permanent?
add	20:40:24	0:00: 42	1	124	12	NO
ceiling	02:38:59	00:00:43	1	0	136	NO
clear	02:39:47	00:00:44	2	0	136	YES
contains	02:40:04	00:00:44	1	0	136	NO
first	02:35:51	00:00:42	1	0	136	YES
floor	02:39:38	00:00:43	1	0	136	NO
higher	02:39:36	00:00:44	1	0	136	NO
is_empty	02:40:22	00:00:44	1	0	136	YES
last	02:40:12	00:00:44	1	0	136	YES
lower	02:38:31	00:00:43	1	0	136	NO
poll_first	02:40:37	00:00:44	1	0	136	NO
poll_last	02:40:45	00:00:44	1	0	136	NO
remove	21:56:15	0:01: 15	2	131	5	NO
subseta	06:00:53	00:01:20	1	0	136	NO
subsetb	06:02:19	00:01:19	1	0	136	NO

Table 12 Workaround computation for FixedArrayList

`FixedArrayLists`: 300 structs.; min. size: 1, max. size: 53, avg. size: 23.03						
Method to fix	Total time	Avg rep. time	Avg. wa.	# TOs	# transient	Permanent?
contains	00:34:02	00:00:06	1	0	300	NO
get	00:33:50	00:00:06	1	0	300	NO
getlastindex	08:02:31	00:00:20	3.11	45	255	NO
getnextaddindex	04:32:57	00:00:06	1	82	218	NO
isempty	00:34:29	00:00:06	1	0	300	YES
realsize	02:55:00	00:00:07	1	64	236	NO
remove	14:39:05	00:00:04	1	286	14	NO
size	02:58:17	00:00:06	1	63	237	NO
unsafeget	00:33:27	00:00:06	1	0	300	NO

approach. We purposefully omit the return values of methods and workarounds in Table 13 to simplify the presentation. The detailed relevant method signatures are presented in Table 1.

For space reasons, we use a slightly different, more concise, notation in Table 13, compared to the notation we employed to introduce our approaches in Sect. 3. Here, we often replace formal by actual parameters in method invocations. For instance, in the first row, second column of Table 13 we denote by {1,7,10}.add(10) the invocation to the add method with parameters s = {1, 7, 10} and e = 10. In the second column, we denote state-specific workarounds by $F \mapsto W$, where F is a particular invocation of a failing method using specific inputs and W is a sequence of methods that serves as a fix for that scenario. For example, in row one we have that the call {1,7,10}.add(10) (F) can be repaired by executing {1,7,10}.contains(−1) (W). The rationale is that {1,7,10}.add(10) should not modify its input set and return the boolean false (10 is already in s; hence, it is not added again), and exactly the same result (false as a return value, an unchanged input set) is obtained by executing {1,7,10}.contains(−1).

In Table 13, we denote schemas by $F \mapsto S$ (third column) together with the condition C that must be satisfied for our approach to produce a successful repair using the schema (fourth column). This means that if F is invoked using inputs I satisfying condition C, a SAT solver can be queried to produce inputs to instantiate schema S, resulting in a state-specific workaround for the execution of F on I. In this way, the schema in the first row of the table can be read as follows: for any set s and element X such that X ∈ s (C, the condition that needs to be satisfied), a call to s.add(X) (F)

Table 13 Examples of transient WA and schemas found by our approach for `java.util.TreeSet`

Method	Transient WAs	WA Schemas	Conditions
add(e)	{1,7,10}.add(10) ↦ {1,7,10}.contains(-1)	s.add(X) ↦ s.contains(Y)	X ∈ s
ceiling(e)	{1,7,10}.ceiling(10) ↦ {1,7,10}.floor(10)	s.ceiling(X) ↦ s.floor(Y)	True
	{1,7,10}.ceiling(8) ↦ {1,7,10}.floor(11)		
	{1,7,10}.ceiling(20) ↦ {1,7,10}.floor(-1)		
contains(e)	{2,4}.contains(5) ↦ {2,4}.isEmpty()	s.contains(X) ↦ s.isEmpty()	X ∉ s, s ≠ ∅
	{2,4}.contains(4) ↦ {2,4}.add(5);{2,4,5}.remove(5)	s.contains(X) ↦ s.add(Y);s.remove(Z)	X ∈ s
first()	{2,3,4}.first() ↦ {2,3,4}.floor(2)	s.first() ↦ s.floor(X)	True
floor(e)	{1,7,3,10}.floor(10) ↦ {1,7,3,10}.ceiling(10)	s.floor(X) ↦ s.ceiling(Y)	True
	{1,7,3,10}.floor(8) ↦ {1,7,3,10}.ceiling(5)		
	{1,7,3,10}.floor(-1) ↦ {1,7,3,10}.ceiling(100)		
higher(e)	{2,3,5}.higher(4) ↦ {2,3,5}.floor(5)	s.higher(X) ↦ s.floor(Y)	True
	{2,3,5}.higher(-10) ↦ {2,3,5}.floor(-1)		
isEmpty()	{2,3,4}.isEmpty() ↦ {2,3,4}.contains(10)	s.isEmpty() ↦ s.contains(X)	s ≠ ∅
	{}.isEmpty() ↦ {}.add(1);{1}.remove(1)	s.isEmpty() ↦ s.add(X);s.remove(Y)	s = ∅
last()	{2,3,4}.last() ↦ {2,3,4}.floor(4)	s.last() ↦ s.floor(X)	True
lower(e)	{-1,3,4}.lower(2) ↦ {-1,3,4}.floor(-1)	s.lower(X) ↦ s.floor(Y)	True
	{1,3,24}.lower(-1) ↦ {1,3,24}.floor(-2)		
pollFirst()	{2,4,7}.pollFirst() ↦ {2,4,7}.first();{2,4,7}.remove(2)	s.pollFirst() ↦ s.first();s.remove(X)	True
pollLast()	{2,4,7}.pollLast() ↦ {2,4,7}.last();{2,4,7}.remove(7)	s.pollLast() ↦ s.last();s.remove(X)	True
remove(e)	{2,5,9}.remove(2) ↦ {2,5,9}.pollFirst()	s.remove(X) ↦ s.pollFirst()	X = min(s)
	{2,5,9}.remove(9) ↦ {2,5,9}.pollLast()	s.remove(X) ↦ s.pollLast()	X = max(s)

can be replaced by a call to s.contains(Y) (*S*), and the SAT solver will find a value for Y such that the latter call produces exactly the same result as the former (i.e. if s = {1,7,10} and X = 10, the solver might return −1 as an answer). The other rows of the table share the same format.

While transient workarounds are very specific to a particular state, the schemas that are derived from them feature different degrees of generality. On the one hand, we have schemas that are always useful to repair the given action. Examples of these are the schemas discovered for the actions `first`, `floor`, `higher`, `last`, `lower`, `pollFirst` and `pollLast`. In such cases, our approach is much faster, as shown by the experimental analysis discussed below. Notice that in none of these cases the schemas correspond to permanent workarounds, as the variables of the schemas have to be instantiated with different values depending on the input sets.

On the other hand, other schemas are only applicable under particular conditions on the inputs of the actions. For example, `add` can only be repaired when the element to be added is already in the set; `remove` can be repaired only if the element being removed is the smallest or the largest of the set; and the second schema for `isEmpty` only works for the empty set. While some of these workarounds are rather naive, in some cases they can still be very useful. (The schema-based workarounds for `remove` are good examples.)

With respect to the performance of our approach, Tables 14 and 15 show the time required to find a workaround when schemas are used and when they are not, for the `Stack` and `TreeSet` case studies, respectively. Again, we used many randomly generated inputs for each method, assumed the method failed for the input, and tried to produce a repair with and without schemas. The result of this experiment indicates that schemas are significantly faster as a vehicle for producing repairs than the general transient workaround discovery approach, 11.12 times faster for `Stack` and 8.48 times faster for `TreeSet`, and they should be used if available (recall that schemas are derived from transient workarounds, and can be used when the same method fails after the schema is created).

It is important to remark that our prototypical tool for workaround discovery is not as efficient as it can be in generating workarounds from schemas, as it encodes schemas as part of a DynAlloy program, which is run each time a new schema-based workaround is searched for. Optimizing the run-time of schema-based workaround search is feasible, for instance, by storing the CNF formula corresponding to the schema instantiation program, instead of building this formula for each call, as our prototype does.

Table 14 Speedup achieved by using schemas in the `Stack` case study

`Stacks`: 141 structs.; min. size: 5, max. size: 19, avg. size: 12.19			
Method to fix	Avg rep. time without templates	Avg rep. time with templates	Speed up
stack_contains	0:00:44	0:00: 03	14.67
stack_ empty	0:00:40	0:00: 04	10
stack_firstElement	0:00:44	0:00: 04	11
stack_peek	0:00:44	0:00: 04	11
stack_pop	0:00:51	0:00: 05	10.2
stack_push	0:02:05	0:00: 04	31.25
vector_add	0:00:45	0:00: 04	11.25
vector_addElement	0:00:48	0:00: 04	12
vector_addIndexItem	0:00:45	0:00: 05	9
vector_clear	0:00:40	0:00: 04	10
vector_elementAt	0:00:45	0:00: 04	11.25
vector_get	0:00:45	0:00: 04	11.25
vector_get_first	0:00:44	0:00: 04	11
vector_insertElementAt	0:00:45	0:00: 05	9
vector_lastElement	0:00:44	0:00: 04	11
vector_remove	0:00:45	0:00: 20	2.25
vector_removeAllElements	0:00:45	0:00: 04	11.25
vector_removeElement	0:00:46	0:00: 20	2.3
vector_removeElementAt	0:00:45	0:00: 04	11.25
vector_removeIndex	0:00:45	0:00: 04	11.25
vector_set	0:00:45	0:00: 04	11.25
vector_setElement	0:00:45	0:00: 04	11.25

Table 15 Speedup achieved by using schemas in the `TreeSet` case study

`Sets`: 136 structs.; min. size: 11, max. size: 22, avg. size: 13.16			
Method to fix	Avg rep. time without templates	Avg rep. time with templates	Speed up
add	0:00:42	0:00: 03	14
ceiling	0:00:43	0:00: 13	3.31
clear	0:00:44	0:00: 02	22
contains	0:00:44	0:00: 02	22
first	0:00:28	0:00: 06	4.67
floor	0:00:43	0:00: 12	3.6
higher	0:00:27	0:00: 13	2.1
is_empty	0:00:22	0:00: 02	11
last	0:00:30	0:00: 06	5
lower	0:00:22	0:00: 07	3.14
poll_first	0:00:44	0:00: 05	8.8
remove	0:00:15	0:00:07	8.5

5.4 Threats to validity

Our experimental evaluation involved implementations accompanied by corresponding abstract data types. When available, these were taken from previous work, that used them in a benchmark for automated analysis. We did not formally verify that these implementations and specifications are correct, and they may contain errors that affect our results (notice that even for our second part of the evaluation, transient vs. permanent, we had to equip the evaluated classes with JML specifications in our to be able to run one of our techniques). We manually checked that the workarounds obtained using our techniques (using both of them) were correct, confirming that, as far as our techniques required, the specifications were correct.

Our experiments involved randomly generated scenarios (program states), from which workaround computations were launched. Different randomly picked scenarios may of course lead to different results. We attempted to build a sufficiently varied set of such program states, while keeping the size of the sample manageable. In all cases, we performed workaround computations, for each method under analysis, on more than 100 scenarios. These were selected following an even distribution, and taking into account how Randoop (the random testing tool used to produce the scenarios) performed the generation, reporting our results as an average. We took as many measures as possible to ensure that the selection of the cases did not particularly favour our techniques. Our workaround computation tools make use of optimizations, such as tight bounds [15,16]. These may introduce errors, e.g. making the exploration for workarounds not bounded exhaustive. We experimentally checked consistency of our prototypes with/without these optimizations, to ensure these did not affect the outcomes.

When comparing with other techniques, especially in the second part of our evaluation, we relied in the results and running times reported in the literature [18], since replicating the experiments in our own infrastructure led to worse running times than those reported in [18].

The authors of [28] state that strong contracts allowed them to find twice as many bugs than weaker contracts, using testing. We believe that it is much more important to have stronger contracts in program repair techniques (like the one presented here) than in other approaches like verification or software testing, since a repair that is not correct results in new failures that do not contribute to improving software reliability (in fact it decreases reliability). The problem of incorrect repairs with weaker specifications happened too frequently in our experiments that we ended up discarding them.

6 Related work

Existing approaches to workaround computation are among the closest work related to our first technique. We identify two lines, one that concentrates in *computing* workarounds, as in [6,8,18], and another that focuses on *applying* workarounds [4]. Our work is closer to the former, since we do not study in this paper the run-time application of workarounds. As opposed to [6,8], requiring a state transition system abstraction, our workarounds are computed directly from source code contracts. Our workarounds are also *transient*, as we have previously emphasized in the paper, as opposed both to [6,8] and [18], that produce permanent workarounds, with a diminished repairability as evaluated in the previous section. The work in [18] produces equivalent method sequences (workarounds) directly at the level of source code. The approach differs from ours in the fact that they do not employ formal specifications; it instead resorts to a two-phase procedure based on evolutionary computation that uses: (i) test cases in the first phase, to produce candidate equivalent sequences that behave as the method of interest in the test scenarios, and (ii) the method of interest (that for which an equivalent sequence is being computed) in the second phase, to attempt to compute a test that differentiates the routines [18]. This second phase needs to assume that the method of interest is correct, a hypothesis that prevents the use of the technique for workaround computation.

Workarounds of the kind used in [4] are alternative equivalent programs to that being repaired. Thus, workarounds can be thought of as automated program repair strategies. In this sense, the work is related to the works on automated program repair, e.g. [10,22,35]. Again, as we mentioned, the workarounds that we compute can repair a program *in a specific state*, i.e. they are workarounds as in the original works [6,8], that do not constitute "permanent" program repairs, but "transient" ones, i.e. that only work on specific situations. Program repair techniques often use tests as specifications and thus can lead to spurious fixes (see [29,33] for detailed analyses of this problem).

Our second technique for workarounds directly manipulates program states, as opposed to trying to produce these indirectly via method calls. This technique is closely related to constraint-based and contract-based structure repair approaches, e.g. [11,19,21], in particular the approach of Khurshid and collaborators to repair complex structures, reported in [36,37]. While Khurshid et al. compute a kind of structure "frame" (the part of the structure that the failing program modified) and then try to repair structures by only modifying the frame, we allow modifications on the whole structure. Also, in [36,37], Alloy integers are used, instead of integers with Java precision. Thus, a greater scalability can be observed in their work (in that work the authors can deal with bigger structures, compared to our approach), whereas in our case the program state characterization is closer to the actual Java program states. Moreover, our technique can repair structures that the approach in [36] cannot. A thorough comparison cannot be made, because the tool and experiments from [36] are not available. Nevertheless, we have followed that paper's procedure, and attempted to repair some of the randomly produced structures of our experiments. For instance, in cases where a rotation is missing (in a balanced tree), the approach in [36] cannot produce repairs, since the fields that are allowed to change are restricted to those visited by the program, and since the rotation is mistakenly prevented, the technique cannot modify fields that are essential for the repair. If, instead, we allow the approach in [36] to modify the whole structure, then the approach is similar to ours without the use of tight bounds and symmetry breaking, which we already discussed in the previous section. The

approaches are, however, complementary, in the sense that we may restrict modifiable fields as proposed in [36], and they could exploit symmetry breaking predicates and tight bounds, as in our case. Our work uses tight field bounds to improve analysis. Tight bounds have been exploited in previous work, to improve SAT-based automated bug finding and test input generation, e.g. in [2,15,16,30], and in symbolic execution-based model checking, to prune parts of the symbolic execution search tree constraining nondeterministic options, in [17,31].

Both our techniques require formal specifications for run-time repair; more precisely, we require that class invariants, preconditions and postconditions for the methods involved in the workaround discovery process, to be provided. While this requirement may seem very demanding for developers, we conjecture that there is no other effective way to guarantee that the run-time repairs produced by approaches such as the ones presented in this paper are indeed correct. This is in fact an issue that affects automated program repair approaches in general, not only "run-time" repairs as the ones aimed at in this paper. As a hint of the seriousness of this issue, some of the authors of this paper have recently conducted a study [38] that shows that most of the program repairs produced by state-of-the-art automated repair tools that do not use formal specifications are in fact *spurious* (i.e. are not correct repairs). This issue, known as overfitting and also acknowledged by various researchers [26,33], evidences that, while tests may be suitable as a way of approximating program verification, these are "too partial" as behaviour specifications to be used in more complex settings, such as program repair ones.

To reduce the burden of writing specifications, modern formal modelling languages (like JML) provide different features. Notably, model-based contracts [28] allow one to write invariants, preconditions and postconditions over an abstract model of the software, in a simple and concise way. For instance, by using model-based contracts one can relate the states of a TreeSet structure (implemented with a red–black tree) to a simple sets and specify the pre- and postconditions of methods at the set level; one can express the postcondition of a `TreeSet`'s `add` method by saying that the obtained set after the execution of `add` is equal to the union of the former set plus the new added element. The authors of [28] state that strong contracts allowed them to find twice as many bugs than weaker contracts, using testing. Other authors have also recently recognized the relative simplicity of formal specification compared to implementation in some domains, as well as the need (and usefulness) of formal specification for complex activities such as repair or synthesis [25].

Since our techniques require formal specifications, one may wonder why not perform automated verification rather than performing run-time recoveries. One reason is *scalability*. In fact, automated SAT-based bounded verification techniques (e.g. the tool TACO [15]) have hard problems scaling up to structures with half the size of those supported by our approach. For example, for bounded verification of a single method of `Binary Search Tree`, using structures with up to 10 nodes, TACO requires more than 30 min, and it does not terminate in 10 h for structures with 12 nodes (similar tools, like JForge [12], exhibit even worse performance). In contrast, our approach can find run-time repairs for the same case in less than a minute, for structures twice as large (11 to 22 nodes). There are two reasons for this. First, our approach works at the specification level, which as explained above is usually much more abstract than the source code level. In this way, our approach does not care about implementation details (i.e. for Binary Search Trees our approach works with simple sets, in contrast to TACO and JForge that work with the detailed structure of program heaps conforming Binary Search Trees). Second, our approach exploits the fact that it has to consider only one initial state (the state where the faulty action failed), instead of the whole set of (bounded) initial states that satisfy the precondition of a method needed to perform bounded verification. This greatly reduces the work of the SAT solver in finding run-time repairs. This means that our approach can still be useful when the software is well tested or verified with bounded approaches like TACO/JForge, as testing might miss faults, and for the reasons mentioned above the scalability of our approach can be superior to that of similar verification approaches.

Compared to theorem proving based techniques, like Dafny [23], our approach sacrifices full correctness guarantees, but requires significantly less work from the user. Proving program correctness typically involves a deep understanding of the low-level details of the program being verified, such as pointer aliasing, frame conditions and loop invariants, that are not needed by our approaches, which only depend on higher level method pre/postconditions and class invariants. Moreover, theorem provers often require the user to devise and prove auxiliary lemmas to complete verification, as well as mastering proof techniques like induction, reductio ad absurdum, etc., that are not required by our approaches.

Notice that our techniques disregard nonfunctional properties such as resource efficiency and thus cannot handle bugs affecting these aspects of software.

7 Conclusions and future work

The intrinsic complexity of software, the constant adaptation/extension that software undergoes and other factors make it very difficult to produce software systems maintaining high quality throughout their whole lifetime. This

fact, combined with increasingly stronger pressures to have constant availability in software, makes techniques that help systems tolerate bug-related failures highly relevant. In this paper, we have presented two techniques that contribute to tolerate run-time bug-related failures. These techniques propose the use of SAT-based automated analysis to automatically compute workarounds, i.e. alternative mechanisms offered by failing modules to achieve a desired task, and automated program state repair. These techniques apply directly to formal specifications at the level of detail of program contracts, which are exploited for workaround and state repair computations. Our program state characterizations are closer to the actual concrete program states than some related approaches and can automatically deal with program specifications at the level of detail of source code, as opposed to alternatives that require the engineer to manually produce high level state machine program abstractions. Also, our produced workarounds are *state specific*, in the sense that these can be used in place of a failing routine only in a particular program state, that is used as part of the search for the alternative execution path. We have performed an experimental evaluation that involved various contract-equipped implementations (including arithmetic-intensive ones) and showed that our techniques can circumvent run-time failures by automatically computing workarounds/state repairs from complex program specifications, in a number of randomly produced execution scenarios. The experiments also show that our techniques can compute state-specific workarounds in many situations in which workarounds based on equivalent method sequences are unavailable.

Our presented work leads to various lines for further work. On one hand, further experimental evaluation, in particular evaluating the techniques' performance in software other than our case studies, is important, especially taking into account the previously identified target for workarounds [8]. Also, developing more sophisticated optimization techniques, for instance, further exploiting tight bounds, is a constant concern in our research. Finally, the repairs produced by our workarounds are "transient", in the sense that they only repair a failing program in a specific state situation. However, many of the computed workarounds are in fact instances of more "permanent" workarounds, i.e. they may be generalized to produce alternative program paths to *permanently* circumvent program failures. Our notion of workaround schema is a step in the direction of automatically producing generalizations from transient workarounds, but as we mentioned previously, do not constitute permanent workarounds per se (schema variable instantiation still demands calls to a SAT solver). We are nevertheless, through schemas, a step closer to permanent workarounds. We plan to study mechanisms to *promote* schemas into permanent workarounds as part of our future work.

References

1. Replication Package for Automated Workarounds from Java Program Specifications Based on SAT Solving. http://dc.exa.unrc.edu.ar/staff/naguirre/sat-workarounds/. Accessed 30 July 2018
2. Abad, P., Aguirre, N., Bengolea, V.S., Ciolek, D., Frias, M.F., Galeotti, J.P., Maibaum, T., Moscato, M.M., Rosner, N., Vissani, I.: Improving test generation under rich contracts by tight bounds and incremental SAT solving. In: Sixth IEEE International Conference on Software Testing, Verification and Validation (ICST 2013), Luxembourg, 18–22 Mar 2013, pp. 21–30. IEEE Computer Society (2013)
3. Belt, J., Xianghua, D.: Sireum/topi LDP: a lightweight semi-decision procedure for optimizing symbolic execution-based analyses. In: van Vliet H., Issarny V. (eds.) Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering 2009, Amsterdam, 24–28 Aug 2009, pp. 355–364. ACM (2009)
4. Carzaniga, A., Gorla, A., Mattavelli, A., Perino, N., Pezzè, M.: Automatic recovery from runtime failures. In: Notkin D., Cheng B.H.C., Pohl K. (eds.) 35th International Conference on Software Engineering (ICSE '13), San Francisco, CA, pp. 782–791, 18–26 May 2013. IEEE Computer Society (2013)
5. Carzaniga, A., Gorla, A., Perino, N., Pezzè, M.: Automatic workarounds for web applications. In: Roman G.-C., van der Hoek A. (eds.) Proceedings of the 18th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2010, Santa Fe, NM, USA, 7–11 Nov 2010, pp. 237–246. ACM (2010)
6. Carzaniga, A., Gorla, A., Perino, N., Pezzè, M.: RAW: runtime automatic workarounds. In: Kramer J., Bishop J., Devanbu P.T., Uchitel S. (eds.) Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering (ICSE 2010), Cape Town, vol. 2, pp. 321–322, 1–8 May 2010. ACM (2010)
7. Carzaniga, A., Gorla, A., Perino, N., Pezzè, M.: Automatic workarounds: exploiting the intrinsic redundancy of web applications. ACM Trans. Softw. Eng. Methodol. **24**(3), 16-1–16-42 (2015)
8. Carzaniga, A., Gorla, A., Pezzè, M.: Self-healing by means of automatic workarounds. In: Cheng B.H.C., de Lemos R., Garlan D., Giese H., Litoiu M., Magee J., Müller H.A., Taylor R.N. (eds.) 2008 ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems, SEAMS 2008, Leipzig, Germany, 12–13 May 2008, pp. 17–24. ACM (2008)
9. Chalin, P., Kiniry, J.R., Leavens, G.T., Poll, E.: Beyond assertions: advanced specification and verification with JML and esc/Java2. In: de Boer F.S., Bonsangue M.M., Graf S., de Roever W.P. (eds.) Formal Methods for Components and Objects, 4th International Symposium, FMCO 2005, Amsterdam, 1–4 Nov 2005, Revised Lectures, volume 4111 of Lecture Notes in Computer Science, pp. 342–363. Springer (2005)
10. Debroy, V., Wong, W.E.: Using mutation to automatically suggest fixes for faulty programs. In: Third International Conference on Software Testing, Verification and Validation (ICST 2010), Paris, 7–9 Apr 2010, pp. 65–74. IEEE Computer Society (2010)
11. Demsky, B., Rinard, M.C.: Automatic detection and repair of errors in data structures. In: Crocker R.S. Jr., Guy L. (eds.) Proceedings of the 2003 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA 2003), 26–30 Oct 2003, Anaheim, CA, pp. 78–95. ACM (2003)
12. Dennis, G., Chang, F.S.H., Jackson, D.: Modular verification of code with SAT. In: Pollock L.L., Pezzè M. (eds.) Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2006), Portland, Maine, pp. 109–120, 17–20 July 2006. ACM (2006)

13. Frias, M.F., Galeotti, J.P., Pombo, C.L., Aguirre, N.: Dynalloy: upgrading alloy with actions. In: Roman G.-C., Griswold W.G., Nuseibeh B. (eds.) 27th International Conference on Software Engineering (ICSE 2005), 15–21 May 2005, St. Louis, Missouri, pp. 442–451. ACM (2005)
14. Galeotti, J.P., Frias, M.F.: Dynalloy as a formal method for the analysis of java programs. In: Sacha K. (ed.) Software Engineering Techniques: Design for Quality (SET 2006), 17–20 Oct 2006, Warsaw, Poland, volume 227 of IFIP, pp. 249–260. Springer (2006)
15. Galeotti, J.P., Rosner, N., Pombo, C.G.L., Frias, M.F.: TACO: efficient sat-based bounded verification using symmetry breaking and tight bounds. IEEE Trans. Softw. Eng. **39**(9), 1283–1307 (2013)
16. Galeotti, J.P., Rosner, N., Pombo, C.L., Frias, M.F.: Analysis of invariants for efficient bounded verification. In: Tonella P., Orso A. (eds.) Proceedings of the Nineteenth International Symposium on Software Testing and Analysis (ISSTA 2010), Trento, 12–16 July 2010, pp. 25–36. ACM (2010)
17. Geldenhuys, J., Aguirre, N., Frias, M.F., Visser, W.: Bounded lazy initialization. In: Brat G., Rungta N., Venet A. (eds.) 5th International Symposium NASA Formal Methods (NFM 2013), Moffett Field, CA, 14–16 May 2013. Proceedings, volume 7871 of Lecture Notes in Computer Science, pp. 229–243. Springer (2013)
18. Goffi, A., Gorla, A., Mattavelli, A., Pezzè, M., Tonella, P.: Search-based synthesis of equivalent method sequences. In: Cheung S.-C., Orso A., Storey M.A.D. (eds.) Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE-22), Hong Kong, 16–22 Nov 2014, pp. 366–376. ACM (2014)
19. Hussain, I., Csallner, C.: Dynamic symbolic data structure repair. In: Kramer J., Bishop J., Devanbu P.T., Uchitel S. (eds.) Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering (ICSE 2010), Cape Town, vol. 2, pp. 215–218, 1–8 May 2010. ACM (2010)
20. Jackson, D.: Software Abstractions: Logic, Language, and Analysis. MIT Press, Cambridge (2006)
21. Khurshid, S., García, I., Suen, Y.L.: Repairing structurally complex data. In: Patrice G. (ed.) Model Checking Software, 12th International SPIN Workshop, San Francisco, CA, 22–24 Aug 2005. Proceedings, volume 3639 of Lecture Notes in Computer Science, pp. 123–138. Springer (2005)
22. Kim, D., Nam, J., Song, J., Kim, S.: Automatic patch generation learned from human-written patches. In: Notkin D., Cheng B.H.C., Pohl K. (eds.) 35th International Conference on Software Engineering (ICSE '13), San Francisco, CA, pp. 802–811, 18–26 May 2013. IEEE Computer Society (2013)
23. Leino, K.R.M.: Dafny: an automatic program verifier for functional correctness. In: Clarke E.M., Voronkov A. (eds.) Logic for Programming, Artificial Intelligence, and Reasoning—16th International Conference, LPAR-16, Dakar, April 25–May 1 2010, Revised Selected Papers, volume 6355 of Lecture Notes in Computer Science, pp. 348–370. Springer (2010)
24. Liskov, B., Guttag, J.V.: Program Development in Java: Abstraction, Specification, and Object-Oriented Design. Addison-Wesley, Reading (2001)
25. Loncaric, C., Ernst, M.D., Torlak, E.: Generalized data structure synthesis. In: Chaudron M., Crnkovic I., Chechik M., Harman M. (eds.) Proceedings of the 40th International Conference on Software Engineering (ICSE 2018), Gothenburg, May 27–June 03 2018, pp. 958–968. ACM (2018)
26. Long, F., Rinard, M.: Staged program repair with condition synthesis. In: Di Nitto E., Harman M., Heymans P. (eds.) Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2015), Bergamo, pp. 166–178, Aug 30–Sept 4 2015. ACM (2015)
27. Pacheco, C., Lahiri, S.K., Ernst, M.D., Ball, T.: Feedback-directed random test generation. In: 29th International Conference on Software Engineering (ICSE 2007), Minneapolis, MN, 20–26 May 2007, pp. 75–84. IEEE Computer Society (2007)
28. Polikarpova, N., Furia, C.A., Pei, Y., Wei, Y., Meyer, B.: What good are strong specifications? In: Notkin D., Cheng B.H.C., Pohl K. (eds.) 35th International Conference on Software Engineering (ICSE '13), San Francisco, CA, pp. 262–271, 18–26 May 2013. IEEE Computer Society (2013)
29. Qi, Z., Long, F., Achour, S., Rinard, M.C.: An analysis of patch plausibility and correctness for generate-and-validate patch generation systems. In: Young M., Xie T. (eds.) Proceedings of the 2015 International Symposium on Software Testing and Analysis (ISSTA 2015), Baltimore, MD, 12–17 July 2015, pp. 24–36. ACM (2015)
30. Rosner, N., Bengolea, V.S., Ponzio, P., Khalek, S.A., Aguirre, N., Frias, M.F., Khurshid, S.: Bounded exhaustive test input generation from hybrid invariants. In: Black A.P., Millstein T.D. (eds.) Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA 2014), Part of SPLASH 2014, Portland, OR, 20–24 Oct 2014, pp. 655–674. ACM (2014)
31. Rosner, N., Geldenhuys, J., Aguirre, N., Visser, W., Frias, M.F.: BLISS: improved symbolic execution by bounded lazy initialization with SAT support. IEEE Trans. Softw. Eng. **41**(7), 639–660 (2015)
32. Samimi, H., Aung, E.D., Millstein, T.D.: Falling back on executable specifications. In: D'Hondt T. (ed.) ECOOP 2010—Object-Oriented Programming, 24th European Conference, Maribor, Slovenia, 21–25 June 2010. Proceedings, volume 6183 of Lecture Notes in Computer Science, pp. 552–576. Springer (2010)
33. Smith, E.K., Barr, E.T., Le Goues, C., Brun, Y.: Is the cure worse than the disease? Overfitting in automated program repair. In: Di Nitto E., Harman M., Heymans P. (eds.) Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2015), Bergamo, pp. 532–543, Aug 30–Sept 4 2015. ACM (2015)
34. Visser, W., Pasareanu, C.S., Pelánek, R.: Test input generation for java containers using state matching. In: Pollock L.L., Pezzè M. (eds.) Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2006), Portland, Maine, pp. 37–48, 17–20 July 2006. ACM (2006)
35. Weimer, W., Nguyen, T., Le Goues, C., Forrest, S.: Automatically finding patches using genetic programming. In: 31st International Conference on Software Engineering (ICSE 2009), 16–24 May 2009, Vancouver, Proceedings, pp. 364–374. IEEE (2009)
36. Zaeem, R.N., Gopinath, D., Khurshid, S., McKinley, K.S.: History-aware data structure repair using SAT. In: Flanagan C., König B. (eds.) Tools and Algorithms for the Construction and Analysis of Systems—18th International Conference (TACAS 2012), Held as Part of the European Joint Conferences on Theory and Practice of Software (ETAPS 2012), Tallinn, Estonia, Mar 24–Apr 1 2012. Proceedings, volume 7214 of Lecture Notes in Computer Science, pp. 2–17. Springer (2012)
37. Zaeem, R.N., Khurshid, S.: Contract-based data structure repair using alloy. In: D'Hondt T. (ed.) ECOOP 2010—Object-Oriented Programming, 24th European Conference, Maribor, Slovenia, 21–25 June 2010. Proceedings, volume 6183 of Lecture Notes in Computer Science, pp. 577–598. Springer (2010)
38. Zemín, L., Brida, S.G., Godio, A., Cornejo, C., Degiovanni, R., Regis, G., Aguirre, Na., Frias, M.F.: An analysis of the suitability of test-based patch acceptance criteria. In: 10th IEEE/ACM International Workshop on Search-Based Software Testing (SBST@ICSE 2017), Buenos Aires, Argentina, 22–23 May 2017, pp. 14–20. IEEE (2017)