Automated Workarounds from Java Program Specifications based on SAT Solving

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

[1] Universidad Nacional de Río Cuarto, Río Cuarto, Argentina.
{uva, pponzio, gregis, naguirre}@dc.exa.unrc.edu.ar
[2] Instituto Tecnológico de Buenos Aires (ITBA), Buenos Aires, Argentina.
mfrias@itba.edu.ar
[3] Consejo Nacional de Investigaciones Científicas y Técnicas (CONICET), Argentina

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. Previous works have exploited this workarounds notion to automatically recover from runtime failures in some application domains. However, existing approaches that compute workarounds automatically either require the user to manually build an abstract model of the software under consideration, or to provide equivalent sequences of operations from which workarounds are computed, diminishing the automation of workaround-based system recovery.

In this paper, we present two techniques that automatically compute workarounds from Java code equipped with formal specifications, avoiding abstract software models and user provided equivalences. These techniques employ SAT solving to compute workarounds on concrete program state characterizations. The first employs SAT solving to compute *traditional* workarounds, while the second directly exploits SAT solving to circumvent a failing method, building a state that mimics the (correct) behaviour of this failing routine. Our experiments, based on case studies involving implementations of collections and a library for date arithmetic, enable us to show 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.

1 Introduction

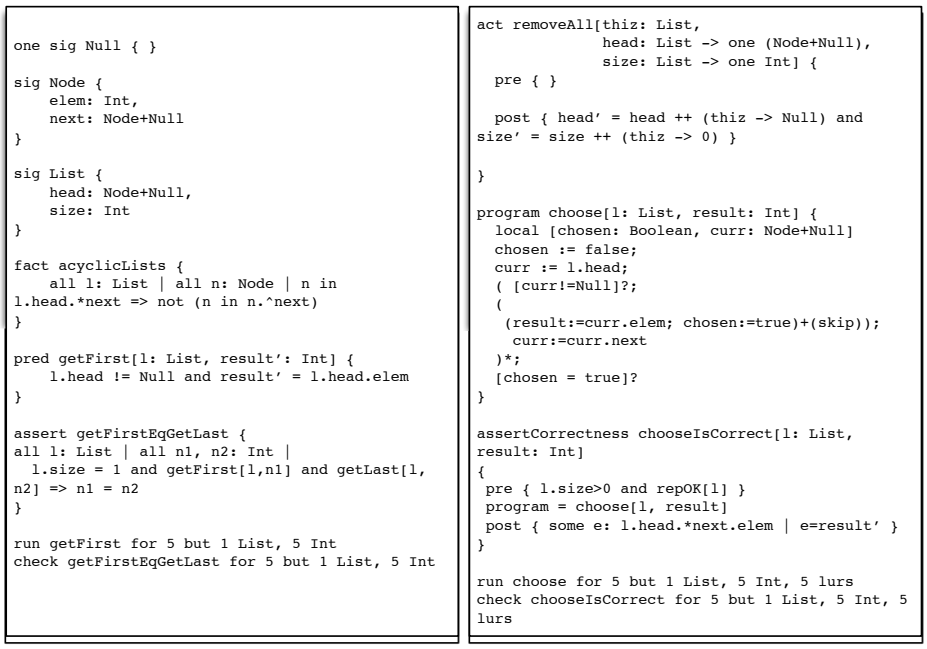
Even in software systems that are built with high quality standards using rigorous software development techniques, bugs still make it through to deployment. Various issues contribute to this situation: the intrinsic complexity of software, the constant adaptation and extension that software systems 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 automatically recover from runtime 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 [4, 5], that needs to be manually provided, or require the user to provide equivalent alternative sequences of operations [7], from which workarounds are computed, diminishing the automation of workaround-based system recovery.

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. These 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 *traditional* workarounds, in the sense that these exploit the intrinsic redundancy of the module holding the failing routine. 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., [26, 30, 31], although it differs in the approaches used to improve scalability. 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, combined with randomly generated program state scenarios for these classes, where methods of these 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.

2 Background

Workarounds and Run Time Repair. The concept of *workaround* was initially defined in the context of self-healing systems [4]. 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



```
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
```

(a)

```
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
```

(b)

Fig. 1: Alloy and DynAlloy specifications for linked lists.

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 labeled with the failing routine. This is in fact the process employed for automated workaround computation presented in [4, 6].

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 [7], 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 [8].

The Alloy and DynAlloy Modeling Languages. In Alloy [18], datatypes 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 Figure 1(a). 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 of 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. 1(a). 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. 1(a) 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. 1(a). 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. Fig. 1(a) shows some sample commands running Alloy Analyzer. These will use SAT solving to build instances involving at most 1 list, 5 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 [12] 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 postconditions. For instance, an action that removes all elements of a list can be defined as `removeAll` in Fig. 1(b). 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. This is a necessary part of our mutable model of the heap (see [13] for details). Second, as opposed to Alloy predicates, which require parameters for post-state 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 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 and atomic actions as base cases, combined using sequential composition (;), nondeterministic choice (+) and iteration ($\star$). A sample program that nondeterministically returns some element of a linked list is program choose in Fig. 1(b). 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. 1(b), 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 [12]. 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. 1(b) produces an execution of choose on a list with at most 5 nodes with at most 5 iterations; the second command checks whether *every* terminating execution of at most 5 iterations of choose, on valid and non-empty lists with at most 5 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 [14, 15, 2] (although the SAT-based analysis of Alloy/DynAlloy is intrinsically incomplete). Our translation is based on [14, 15], and relies on symmetry breaking and *tight field bounds* as optimizations. More precisely, we use the symmetry breaking technique introduced in [14, 15], 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 [14, 15]. They are automatically computed from *assumed* properties, such as preconditions and invariants, 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 datatypes originally introduced in [2] (extended to support some Alloy functions, notably cardinality), enabling us to support increased precision in numerical characterizations of Java basic datatypes. We refer the reader to [14, 15, 2] 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 non-deterministic 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 [14]. 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: $\texttt{act}\, a_i \{\texttt{pre}\{\ pre^A_{m_i}\ \}\texttt{post}\{\ post^A_{m_i}\ \}\}$. 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:

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

which can be automatically analyzed 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: *(i)* s_{A_0} is state s^A_i; *(ii)* 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 *(iii)* 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).

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 instance, 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 nondeteministic choice of methods, as program: `nonDetAssign[i] ; m[i]`. Notice that this nondeterministic assignment is inside the iteration $\star$, 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 2 loop unrolls, 1 tuple and 2 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).

The technique that we introduce in the following section 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 specifica-

tions 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, that 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 [21]. 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 post-conditions 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 post-states, 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 Figure 2, 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 Figure 2. The right-hand side binary tree in Figure 2 is 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 datatype 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 the recoveryStates predicate), a standard mechanism to deal with existential higher-order quantification in Alloy, since Alloy Analyzer does not directly support it (see [18] 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 [14, 15], which forces a canonical ordering in the structures and has a substantial impact in analysis. Second, we use *tight bounds* [14, 15] computed from class invariants (these reduce propositional state representations by removing propositional variables that represent field values deemed infeasible by the invariants) to constrain post-condition states, since these states are assumed to satisfy the corresponding invariants, as shown in the above predicate.

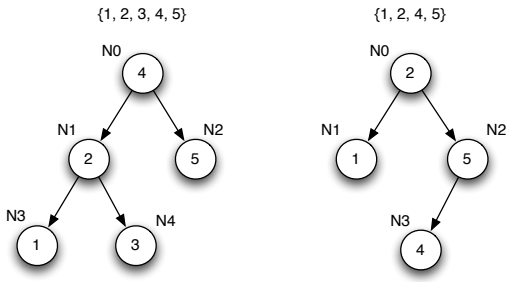


Fig. 2: Two binary search trees, and the sets they represent.

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, respectively. The evaluation 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): *(i)* two implementations of interface `java.util.List`, one based on singly linked lists, taken from [15], the other a circular double linked list taken from `AbstractLinkedList` in Apache Commons.Collections; *(ii)* three alternative implementations of `java.util.Set`, one based on binary search trees taken from [28], another based on AVL trees taken from [3], and the red-black trees implementation `TreeSet` from `java.util`; and *(iii)* 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. All the experiments were run on a PC with 3.40Ghz Intel(R) Core(TM) i5-4460 CPU, with 8GB 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].

In order to assess 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 [22]. For each data structure interface, we ran Randoop for 1 hour, producing 116000 list traces, 136000 set traces, and 138000 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 [22]), 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

Table 1: 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										
	`Lists`				`Singly Lkd Lists`			`Abst. Lkd Lists`		
Method to fix	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	1,732	0	0:11:18	0:0:06	0	0:11:12	0:0:05	0
setelement	1:48:38	0:0:56	1,948	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

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 1, 2 and 3. Tables report: *(i)* method being fixed (the fix is computed from the iteration of nondeterministic choice of remaining methods); *(ii)* total time, the time spent in fixing *all* 100 faulty situations; time is reported in h:mm:ss format; *(iii)* 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; *(iv)* average workaround length, i.e., number of routines that the found workaround had, in average; and *(v)* number of timeouts, i.e., faulty situations that could not be repaired within 10 minutes. 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

Table 2: 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													
	Sets				TreeSet			AVL Tree			Search Tree		
Method to fix	Total Time	Avg. Rep.T.	Avg. wa.	# TOs	Total Time	Avg Rep.T.	# TOs	Total Time	Avg Rep.T.	# TOs	Total Time	Avg Rep.T.	# TOs
add	3:03:48	0:1:21	2	0	2:30:13	0:1:02	1	1:18:07	0:0:30	1	0:49:11	0:0:17	1
ceiling	1:00:11	0:0:26	1	0	0:19:15	0:0:08	0	0:19:27	0:0:08	0	0:18:48	0:0:08	0
clear	22:40:00	0:0:00	-	136	0:10:19	0:0:04	0	0:11:33	0:0:05	0	0:10:48	0:0:04	0
contains	22:40:00	0:0:00	-	136	0:10:33	0:0:04	0	0:11:44	0:0:05	0	0:10:51	0:0:04	0
first	1:04:50	0:0:28	1	0	1:33:50	0:0:37	1	0:57:40	0:0:21	1	0:49:26	0:0:13	2
floor	1:05:06	0:0:28	1	0	2:01:34	0:0:49	1	1:00:19	0:0:22	1	0:27:46	0:0:12	0
higher	0:51:24	0:0:22	1	0	1:55:17	0:0:42	2	0:58:13	0:0:21	1	1:10:47	0:0:27	1
isEmpty	1:13:01	0:0:22	1	2	1:42:15	0:0:41	1	1:00:26	0:0:22	1	1:14:17	0:0:24	2
last	1:09:14	0:0:30	1	0	1:19:27	0:0:30	1	0:53:02	0:0:19	1	0:35:47	0:0:15	0
lower	0:52:03	0:0:22	1	0	1:29:02	0:0:35	1	0:57:28	0:0:21	1	1:11:47	0:0:18	3
pollFirst	1:09:52	0:0:30	1	0	1:23:57	0:0:37	0	0:51:05	0:0:22	0	0:47:43	0:0:12	2
remove	1:00:29	0:0:26	1	0	1:41:19	0:0:40	1	0:51:38	0:0:22	0	0:34:28	0:0:15	0

Table 3: 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							
	Maps				Tree Maps		
Method to fix	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

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 datatype. The timeout is set in 10 minutes. Results are reported in the remaining columns of Tables 1, 2 and 3. Notice that for this technique we do not report workaround size, since it "repairs" the failing method by directly building a suitable post-execution state. Regarding the results of both techniques on the `JodaTime` date arithmetic library, these are summarized in a single table (Table 4) due to space restrictions, for varying bitwidths in numeric datatypes.

Assessment. Notice that our first technique performed very well on the presented experiments. Many methods could be repaired within the timeout limit of 10 minutes (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 [8]. 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

Table 4: Workaround Computation for JodaTime.

	Technique 1						Technique 2					
	Int.16 bits			Int.32 bits			Int.16 bits			Int.32 bits		
Method to fix	# 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

removals as the structure holds, which went beyond the 10-minute 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, 14]. The reason for this increased scalability might at first sight seem obvious, since the analysis starts from a concrete program state. 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 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 datatypes 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 datatypes.

Threats to Validity. Our experimental evaluation involved implementations accompanied by corresponding abstract datatypes. 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. We manually checked that the obtained workarounds were correct, confirming that, as far as our techniques required, the specifications were correct. Our experiments involved ran-

domly 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 at the same time keeping the size of the sample maneagable. 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 favor our techniques. Our workaround computation tools make use of optimizations, such as tight bounds [14, 15]. 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.

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 [4, 6], and another that focuses on *applying* workarounds [7]. Our work is closer to the former. As opposed to [4, 6], requiring a state transition system abstraction, our workarounds are computed directly from source code contracts. Workarounds of the kind used in [7] 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, 20, 29]. The workarounds that we compute can repair a program *in a specific state*, i.e., they are workarounds as in the original works [4, 6], 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 [23, 27] 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, 17, 19], in particular the approach of Khurshid and collaborators to repair complex structures, reported in [30, 31]. 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 [30, 31], 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 [31] cannot. A thorough comparison cannot be carried out, because the tool and experiments from [31]

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 [31] 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 [31] 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 [31], 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 [14, 15, 2, 24], and in symbolic execution based model checking, to prune parts of the symbolic execution search tree constraining nondeterministic options, in [16, 25].

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 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. 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.

As future work, we plan to evaluate the techniques' performance in software other than our case studies, as well as to develop more sophisticated optimization techniques, e.g., by further exploiting tight bounds. Moreover, while the repairs produced by workarounds are in principle "transient", many of the computed workarounds are instances of "permanent" workarounds; we plan to study ways to automatically produce "permanent" workarounds from "transient" candidates, as a proposal of a program repair technique.

References

1. Replication package for *Automated Workarounds from Java Program Specifications based on SAT Solving*, available at http://dc.exa.unrc.edu.ar/staff/naguirre/sat-workarounds/
2. P. Abad, N. Aguirre, V. Bengolea, D. Ciolek, M. Frias, J. Galeotti, T. Maibaum, M. Moscato, N. Rosner and I. Vissani, *Improving Test Generation under Rich Contracts by Tight Bounds and Incremental SAT Solving*, in Proceedings of 6th IEEE International Conference on Software Testing, Verification and Validation ICST 2013, Luxembourg City, Luxembourg, IEEE, 2013.
3. J. Belt, Robby and X. Deng, *Sireum/Topi LDP: a lightweight semi-decision procedure for optimizing symbolic execution-based analyses*, in Proceedings of the 7th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering ESEC/FSE 2009, ACM, 2009.
4. A. Carzaniga, A. Gorla and M. Pezzè, *Self-healing by means of automatic workarounds*, in Proceedings of 2008 ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems SEAMS 2008, Leipzig, Germany, May 12-13, ACM, 2008.
5. A. Carzaniga, A. Gorla, N. Perino and M. Pezzè, *Automatic Workarounds for Web Applications*, in Proceedings of the 18th ACM SIGSOFT International Symposium on Foundations of Software Engineering FSE 2010, Santa Fe (NM), USA, ACM, 2010.
6. A. Carzaniga, A. Gorla, N. Perino and M. Pezzè, *RAW: runtime automatic workarounds*, in Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering ICSE 2010, New York (NY), USA, ACM, 2010.
7. A. Carzaniga, A. Gorla, A. Mattavelli, N. Perino and M. Pezzè, *Automatic recovery from runtime failures*, in Proceedings of the 35th International Conference on Software Engineering ICSE 2013, San Francisco (CA), USA, IEEE/ACM, 2013.
8. A. Carzaniga, A. Gorla, N. Perino and M. Pezzè, *Automatic Workarounds: Exploiting the Intrinsic Redundancy of Web Applications*, ACM Trans. Softw. Eng. Methodol. 24(3), ACM, 2015.
9. P. Chalin, J. R. Kiniry, G. T. Leavens and E. Poll, *Beyond Assertions: Advanced Specification and Verification with JML and ESC/Java2*, in Proceedings of 4th International Symposium on Formal Methods for Components and Objects FMCO 2005, LNCS 4111, Springer, 2005.
10. Debroy V. and Wong W.E., *Using Mutation to Automatically Suggest Fixes to Faulty Programs*. ICST 2010, pp. 65–74.
11. B. Demsky and M. Rinard, *Static Specification Analysis for Termination of Specification-Based Data Structure Repair*, in Proceedings of Proceedings of the 2003 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications OOPSLA 2003, ACM, 2003.
12. M. Frias, J. Galeotti, C. López Pombo and N. Aguirre, *DynAlloy: Upgrading Alloy with Actions*, in Proceedings of International Conference on Software Engineering ICSE 2005, St. Louis, Missouri, USA, ACM, 2005.
13. J. Galeotti and M. Frias, *DynAlloy as a Formal Method for the Analysis of Java Programs*, in Proceedings of Software Engineering Techniques SET 2006: Design for Quality, Warsaw, Poland, IFIP 227, Springer, 2006.
14. J.P. Galeotti, N. Rosner, C. López Pombo and M. Frias, *Analysis of Invariants for Efficient Bounded Verification*, in Proceedings of the Nineteenth International

Symposium on Software Testing and Analysis, ISSTA 2010, Trento, Italy, July 12-16, ACM, 2010.
15. J.P. Galeotti, N. Rosner, C. López Pombo and M. Frias, *TACO: Efficient SAT-Based Bounded Verification Using Symmetry Breaking and Tight Bounds*, IEEE Transactions on Software Engineering 39(9), IEEE, 2013.
16. J. Geldenhuys, N. Aguirre, M. F. Frias and W. Visser, *Bounded Lazy Initialization*, in Proceedings of the 5th International NASA Formal Methods Symposium NFM 2013, LNCS 7871, Springer, 2013.
17. I. Hussain and C. Csallner, *Dynamic symbolic data structure repair*, in Proceedings of Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering ICSE 2010, ACM, 2010.
18. D. Jackson, *Software Abstractions: Logic, Language, and Analysis*, The MIT Press, 2006.
19. S. Khurshid, I. Garcia and Y. Suen, *Repairing Structurally Complex Data*, in Proceedings of 12th International SPIN Workshop SPIN 2005, LNCS, Springer, 2005.
20. Kim D., Nam J., Song J. and Kim S., *Automatic patch generation learned from human-written patches*. ICSE 2013: pp. 802–811.
21. B. Liskov and J. Guttag, *Program Development in Java: Abstraction, Specification and Object-Oriented Design*, Addison-Wesley, 2000.
22. C. Pacheco, S. K. Lahiri, M. D. Ernst and T. Ball, *Feedback-Directed Random Test Generation*, in Proceedings of International Conference on Software Engineering ICSE 2007, IEEE, 2007.
23. Z. Qi, F. Long, S. Achour, and M.C. Rinard. *An analysis of patch plausibility and correctness for generate-and-validate patch generation systems*. In Proceedings of the 2015 International Symposium on Software Testing and Analysis, ISSTA 2015, Baltimore, MD, USA, July 12-17, 2015, pages 24?36, 2015.
24. N. Rosner, V. Bengolea, P. Ponzio, S. Khalek, N. Aguirre, M. Frias and S. Khurshid, *Bounded Exhaustive Test Input Generation from Hybrid Invariants*, in Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages & Applications OOPSLA 2014, ACM, 2014.
25. N. Rosner, J. Geldenhuys, N. Aguirre, W. Visser and M. Frias, *BLISS: Improved Symbolic Execution by Bounded Lazy Initialization with SAT Support*, IEEE Transactions on Software Engineering 41(7), IEEE, 2015.
26. H. Samimi, E. D. Aung and T. Millstein, *Falling Back on Executable Specifications*, in Proceedings of 24th European Conference on Object-Oriented Programming ECOOP 2010, LNCS 6183, Springer, 2010.
27. E.K. Smith, E. Barr, C. Le Goues, and Y. Brun, *Is the Cure Worse Than the Disease? Overfitting in Automated Program Repair*, Symposium on the Foundations of Software Engineering (FSE), 2015.
28. W. Visser, C. Pasareanu and R. Pelánek, *Test input generation for java containers using state matching*, in Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis ISSTA 2006, ACM, 2006.
29. Weimer W., Nguyen T., Le Goues C. and Forrest S., *Automatically finding patches using genetic programming*. ICSE 2009: pp. 364–374.
30. R. Zaeem and S. Khurshid, *Contract-Based Data Structure Repair Using Alloy*, in Proceedings of 24th European Conference on Object-Oriented Programming ECOOP 2010, LNCS 6183, Springer, 2010.
31. R. Zaeem, D. Gopinath, S. Khurshid and K. McKinley, *History-Aware Data Structure Repair Using SAT*, in Proceedings of 18th International Conference on Tools and Algorithms for the Construction and Analysis of Systems TACAS 2012, LNCS 7214, Springer, 2010.