Stateful Testing: Finding More Errors in Code and Contracts

Yi Wei · Hannes Roth · Carlo A. Furia · Yu Pei · Alexander Horton · Michael Steindorfer · Martin Nordio · Bertrand Meyer
Chair of Software Engineering, ETH Zurich, Switzerland
{yi.wei, carlo.furia, yu.pei, martin.nordio, bertrand.meyer}@inf.ethz.ch {haroth, ahorton, msteindorfer}@student.ethz.ch

***Abstract*—Automated random testing has shown to be an effective approach to finding faults but still faces a major unsolved issue: how to generate test inputs diverse enough to find many faults and find them quickly. Stateful testing, the automated testing technique introduced in this article, generates new test cases that improve an existing test suite. The generated test cases are designed to violate the dynamically inferred contracts (invariants) characterizing the existing test suite. As a consequence, they are in a good position to detect new errors, and also to improve the accuracy of the inferred contracts by discovering those that are unsound.**

Experiments on 13 data structure classes totalling over 28,000 lines of code demonstrate the effectiveness of stateful testing in improving over the results of long sessions of random testing: stateful testing found 68.4% new errors and improved the accuracy of automatically inferred contracts to over 99%, with just a 7% time overhead.

***Keywords*-random testing, dynamic analysis, automation**

I. Introduction

Drawing inputs at random may sound like a desultory approach to testing, since it ignores any information about the structure of the system under test. This intuition, however, turns out to be largely flawed: there is now a compelling amount of evidence—both empirical [1] and analytical [2]—showing that random testing is a quite effective testing technique that can uncover many subtle errors in real programs.

When the tested software is equipped with *contracts* (pre and postconditions) random testing even becomes a completely *automated* technique: preconditions help select valid inputs and postconditions provide oracles to check if a test case exposes unexpected behavior that does not conform to specification. The applications of random input generation are not limited to testing but extend to other software dynamic analysis techniques, such as inference of contracts [3] (improving and completing those written by programmers) and even automated program correction [4].

Constructing random inputs is straightforward for primitive types, such as integers and characters, where it boils down to drawing pseudo-random numbers. Constructing random objects of arbitrary classes is more involved, because objects can only be created and modified using a class' routines (methods). To approach this problem, random input generation algorithms for object-oriented languages maintain an *object pool*, which stores all objects randomly generated during the current testing session. The pool is populated either with fresh objects, built from scratch by creation procedures (constructors), or with objects returned by random routine calls on objects of appropriate type, randomly drawn from the pool. Routines and creation procedures with arguments are handled by recursively drawing from the pool conforming objects to be used as arguments. A *test case* is the combination of any target object in the pool with a routine applied to it.

Random testing sessions must last several hours to maximize error-finding effectiveness [1], [2]. A drawback of this necessity is that the object pool grows to contain a large number of objects, even when duplicates are pruned. Therefore, the probability of generating at random test cases that would expose new bugs significantly decreases over time: the objects needed to generate the "missing" test cases may already be in the object pool, but they are unlikely to be drawn at random because they constitute only a small fraction of the whole pool.

This paper presents *stateful testing*, a dynamic analysis technique that builds on top of random testing and magnifies its effectiveness. Stateful testing takes over where random testing gives up: after long sessions of random test case generation, the number of errors found reaches a plateau or grows sluggishly, and the object pool contains thousands of objects. At this point, stateful testing populates a database with the content of the pool stored as serialized objects; the database is searchable for objects that satisfy given predicates. For example, we can look up an object n of class *INTEGER* such that $n > 0$, or an object s of class *SET* that satisfies **not** $s\,.\,is\_empty$ (that is, the set contains at least one element).

After populating the database, stateful testing runs dynamic contract inference [3] on all *passing test cases* generated during random testing; the result of this step is a collection of pre and postcondition clauses that summarize the properties of the test cases. Dynamic contract inference characterizes the passing test cases with pre and postconditions based on *templates*, which capture recurring usage patterns that lend themselves to "meaningful" generalization. For object-oriented programs, the set of public *queries* (functions) of a class often provides a valuable collection of predicates to be combined in templates; *is_empty* in the example above is a public query that often appears in contracts (inferred and programmer-written). Since the inference is based on a finite number of observations and on heuristics in the form of templates, some of the inferred contracts can be *unsound*: they merely are a reflection of the test cases that have been exercised.

Stateful testing combines the information stored in the database of objects and the inferred contracts, with the goal of mutually enhancing the test suite and the contracts, along the lines of Xie and Notkin's proposal [5]. Stateful testing proceeds by systematically searching the database for objects that *violate* some of the inferred contracts and therefore enable the creation of *new* test cases. A new test case that executes successfully shows that an inferred contract can be violated without compromising execution, hence the contract is unsound and should be discarded. A new test case that triggers a failure exposes an error overlooked in the previous testing session, corresponding to an input never tried before. Either way, the new test cases improve over the previous testing session by reaching out regions of the object space previously unexplored. Take, for example, a routine *wipe_out* of class *SET*, which removes all the elements contained in the set. If *wipe_out* has always been called on empty sets, dynamic contract inference suggests the precondition *is_empty*. Then, select an object *s* that violates the precondition, that is such that **not** *s.is_empty*. If the call *s.wipe_out* succeeds, it shows that the inferred precondition *s.is_empty* is unsound and should be removed. If the call triggers a failure, it exposes an error in the routine's implementation, which does not handle correctly sets that are not already empty.

We implemented stateful testing within our AutoTest [6] framework for random testing of object-oriented Eiffel applications; the implementation is integrated in EVE [7], the freely available research branch of the EiffelStudio development environment. In an extensive set of experiments described in the paper, we applied stateful testing to the historical data generated by running AutoTest for 520 hours on 13 classes from the EiffelBase [8] and Gobo [9] data structure collections. Both libraries have a long development history and are widely used in the Eiffel community. AutoTest generated 149,293 distinct test cases, exposed 95 errors in the libraries, and inferred hundreds of new contracts. We applied stateful testing for 36 hours on this massive data set. In this relatively limited amount of time, stateful testing exposed 65 new errors (68.4% improvement) and invalidated 39.3% of the inferred contracts; manual inspection reveals that almost all the retained contracts are sound. These figures are promising and demonstrate that stateful testing is an effective technique to boost the effectiveness of random testing and dynamic analysis.

The rest of the paper is organized as follows: Section II gives an overview of stateful testing with a few examples; Section III describes the details of the technique; Section IV outlines the design of the relational database used to store the results of the initial dynamic analysis; Section V reports the experimental evaluation of stateful testing; Section VI discusses limitations and future work; Section VII presents related work; Section VIII concludes.

II. Examples

This section presents three detailed examples that demonstrate the applicability of stateful testing; the examples are from the libraries EiffelBase and Gobo.

A. Unsound preconditions

The first example shows how stateful testing can generate tests with a better coverage and detect unsound preconditions. Class *TWO_WAY_SORTED_SET* is the standard Eiffel implementation of sets with ordered elements. The class includes a public routine

merge (*other*: *TWO_WAY_SORTED_SET*)

which inserts all elements of *other* into the **Current** set (*this* in Java or C#). After running for 40 hours, AutoTest reports a dynamically inferred precondition for *merge*:

pre_1: **Current**.*disjoint* (*other*) ,

indicating that it has only been called on disjoint sets: **Current** $\cap$ *other* = $\emptyset$, hence the functionality of *merge* has not been tested thoroughly.

Stateful testing takes over from this situation and tries to generate new test cases that cover the deficiency. To this end, it looks up the database—filled with data from hours of random testing—for objects of suitable type that *violate pre_1*; namely, it searches for two objects *o1*, *o2* such that:

(1) *o1.type* = *TWO_WAY_SORTED_SET* ,
(2) *o2.type* = *TWO_WAY_SORTED_SET* ,
(3) **not** *o1. disjoint* (*o2*) .

Even if AutoTest never drew such objects during the 40-hour session, there are several pairs satisfying the three constraints (1–3) in the database. For every such pair of objects, stateful testing generates the new test case *o1.merge* (*o2*).

Executing the new test cases improves the coverage of routine *merge*; it also reveals that the inferred precondition *pre_1* is unsound and must be *reduced*, hence removing an error in the inferred contracts. In our experiments, the new test cases did not expose any error.

B. Unsound postconditions

The second example shows how stateful testing can detect unsound dynamically inferred *postconditions*. Routine *merge_left* (*other*: *LINKED_LIST*) in class *LINKED_LIST* merges the content of *other* into the **Current** list. Extensive dynamic analysis reports, among others, the following postcondition for *merge_left*:

post_2: **old Current**.*is_equal* (*other*)
implies Current.*is_empty* .

That is, whenever **Current** and *other* contain the same elements (they are *equal*), they are actually empty lists. *post_2* is unsound, as it merely reflects the fact that the test suite never ran *merge_left* on lists that are equal but not empty.

Stateful testing targets the antecedent in the implication *post_2*, which refers to the state *before* executing *merge_left* by means of the **old** notation. The antecedent suggests to exercise the routine on objects *o1*, *o2* where **old** *o1.is_equal* (*o2*) is the case, but **not** *o1.is_empty*, with the hope of showing that *post_2*'s consequent does not hold after the call. Stateful testing creates a new test case *o1.merge_left* (*o2*) for every pair of objects in the database that satisfy the criteria. Since *merge_left* does not remove any element from the target *o1*, **not** *o1.is_empty* still holds after executing the test cases, thus invalidating *post_2* and increasing the coverage of *merge_left*.

C. Constructing new objects

The third example shows how stateful testing can generate new objects by mutating other objects serialized in the database. The example targets class *TWO_WAY_TREE*, an implementation of trees with arbitrary number of branches at each level. An object of type *TWO_WAY_TREE* encapsulates a tree's node; each node includes a list of references to its *children*—empty if the node is a leaf—and a *cursor*. The cursor is an iterator over the list of children, pointing to an element in the list or being *off* the list. Given two nodes n_1, n_2, we can merge n_2's children into n_1's by calling $n1$.*merge_tree_after* ($n2$): n_2's list merges into n_1's after the position marked by n_1's cursor, as shown in Figure 1 where an arrow ⇑ marks the position of the cursor, when it is not *off*. The position "after the cursor" is not defined if the cursor is *off*; developers wrote a precondition (**require** clause) to *merge_tree_after* to enforce this constraint on the input:

```
merge_tree_after (other: TWO_WAY_TREE)
    require not off
```

where *off* is a Boolean query that holds when the cursor of the **Current** node is *off* (such as for nodes n_0 and n_2 in Figure 1).

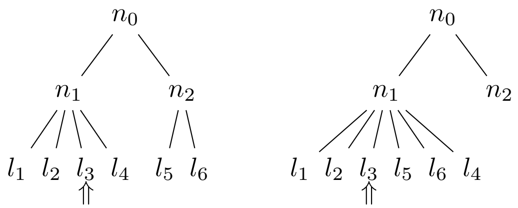
n_0
n_1
n_2
l_1 l_2 l_3 l_4 l_5 l_6
⇑
n_0
n_1
n_2
l_1 l_2 l_3 l_5 l_6 l_4
⇑

Fig. 1. Calling n_1*merge_tree_after* (n_2) on the left tree results in the tree shown on the right.

Dynamic analysis with AutoTest reports the dynamically inferred precondition for *merge_tree_after*:

pre_3: **not Current**. *is_sibling* (*other*) .

pre_3 reveals that *merge_tree_after* has never been tested with *sibling* nodes, that is nodes at the same level of the tree (e.g., n_1 and n_2 in Figure 1). Correspondingly, stateful testing looks up the database for objects violating *pre_3*, suitable to generate new test cases: two objects *o1*, *o2* of type *TWO_WAY_TREE* that satisfy *o1*. *is_sibling* (*o2*) and **not** *o1*. *off*—the latter constraint is *merge_tree_after*'s programmer-written precondition.

Unfortunately, no pair of objects in the database satisfies all these constraints: there are several trees with sibling nodes, but all of them have their cursor *off*, hence *merge_tree_after* cannot be applied. In such situations, stateful testing selects available objects that satisfy *some* of the requirements and searches for *routines* that can mutate the object state to satisfy the missing requirements. The database also includes information on the behavior of routines, collected during dynamic analysis.

In the running example, stateful testing searches for a routine of class *TWO_WAY_TREE* that can change a node where *off* is **False** to one where it is **True**. Routine *start* moves the cursor to the first child node (if the child list is not empty), hence it satisfies the search criteria. With this routine, stateful testing generates a new test case for *merge_tree_after* as follows. It selects two serialized objects *o1*, *o2* of type *TWO_WAY_TREE* that are siblings; the test case consists of two consecutive calls:

o1. *start* ; *o1*. *merge_tree_after* (*o2*) .

In our experiments, this new test case triggered a failure, showing that *merge_tree_after* does not work correctly on sibling nodes. This error went undetected in the random testing session, but stateful testing readily exposed it.

III. How Stateful Testing Works

This section starts with an overview of how stateful testing works (Section III-A), and then describes the details of the technique: what are the products of random testing (Section III-B), how stateful testing processes and organizes them (Section III-C), the role of dynamically inferred contracts (Section III-D), and their reduction to produce new test suites (Section III-E).

A. Overview

Figure 2 provides a bird's eye view of how stateful testing works. Stateful testing is a fully automated technique that produces new test cases from an existing test suite:

1) Running AutoTest, the automatic random testing framework for Eiffel, for several hours produces a large pool of *objects*, and a *test suite* based on those objects.
2) Stateful testing selects and extracts information from the object pool and the test suite and stores it in a relational database: the *object/transition database*.
3) AutoInfer, the dynamic contract inference component of AutoTest, summarizes the behavior of the test cases in the test suite in the form of *dynamically inferred contracts*.
4) The *reduction* phase extracts objects from the database that violate some of the inferred contracts. The extracted objects support the generation of a *new test suite*, which exercises the classes under tests differently than in the original test suite.
5) Executing the new test suite can uncover *new errors* in the code under test, and reveal which of the inferred *contracts* are *incorrect* and should be discarded.

B. Preliminaries: test cases and objects

A *test case* t is the call of a routine r on a target object a_0 with actual arguments $a_1, \ldots, a_m$ that returns an object b, denoted as:

$$t \quad = \quad a_0.r\,(a_1, \ldots, a_m) : b\,.$$

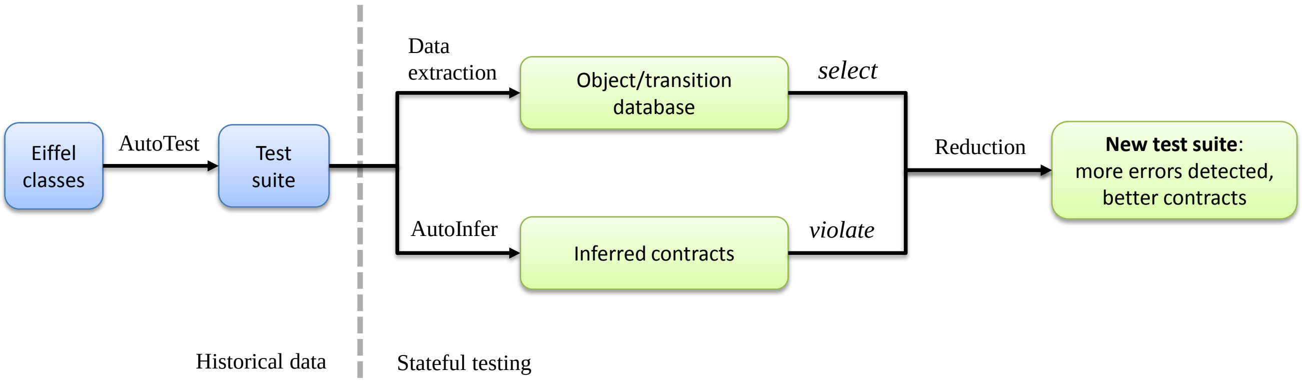

Fig. 2. Overview of how stateful testing works.

If r is a *command* (procedure), which does not return any value, replace b with the dummy object ϵ; if r is a *creation procedure* (constructor), which returns a fresh object, replace the target a_0 with ϵ.

Contracts are annotations using the same syntax as programming language Boolean expressions; they specify the behavior of routines through *preconditions* and *postconditions*. The precondition of a routine r is a predicate that r's target and arguments satisfy before the call; for example, *pre_1* in Section II-A declares that the **Current** list (i.e., the target) and the *other* list (i.e., the argument) are disjoint, for every call of *merge*. The postcondition of a routine r is a predicate over r's result (if any), as well as r's target and arguments; postconditions can refer to targets and arguments both in the post-state (i.e., after the call) and in the pre-state (i.e., before the call, with the **old** keyword). For example, *post_2* in Section II-B specifies that, if the target list and the *other* list contained the same elements before a call to *merge*, then the target is empty after the call. In Eiffel, programmers can annotate routines with pre (**require** clause) and postconditions (**ensure** clause); stateful testing includes a contract inference phase that supplements the contracts written by programmers with inferred contracts.

Contracts provide a criterion to determine if a test case is *passing* or *failing* completely automatically. A routine's test case $a_0.r\,(a_1, \ldots, a_m) : b$ is *valid* if its target and arguments $a_0, a_1, \ldots, a_m$ satisfy r's precondition, and is *invalid* otherwise. Executing a valid test case t changes the target and arguments into the *post-state* $a'_0, a'_1, \ldots, a'_m$, denoted

$$t \rightsquigarrow \langle a'_0, a'_1, \ldots, a'_m \rangle \,.$$

t is *passing* if executing the test case triggers no exceptions, and the post-state $\langle a'_0, a'_1, \ldots, a'_m \rangle$, the pre-state $\langle a_0, a_1, \ldots, a_m \rangle$ and the returned object b satisfy r's postcondition; otherwise, t is *failing*.

Stateful testing builds upon an existing test suite that exercises a set of classes. A test suite is a collection $\mathcal{T} = \{t_1, t_2, \ldots\}$ of test cases; it induces the set $\mathcal{O} = \{o_1, o_2, \ldots\}$ of all objects mentioned in $\mathcal{T}$'s test cases or in the post-state of passing test cases; $\mathcal{O}$ is the *object pool*. Stateful testing works independently of how the object pool $\mathcal{O}$ and the test suite $\mathcal{T}$ are generated. Its implementation in the AutoTest framework, however, generates them completely automatically from a set of Eiffel classes with random testing.

Listing 1. Routines of class *LIST* with contracts.

```
make: LIST                 -- Create an empty list
    ensure Result.is_empty

wipe_out                   -- Remove all elements
    ensure is_empty

extend (v: ANY)            -- Add 'v' to the end
    ensure has (v)

append (other: LIST)       -- Append 'other' to the end
    require other ≠ Void

has (v: ANY): BOOLEAN      -- Does the list include 'v'?

is_empty: BOOLEAN          -- Is the list empty?
```

Example. The class *LIST* implements dynamic lists; it is modeled after real Eiffel classes, but is simplified for clarity. Listing 1 shows the signatures of *LIST*'s routines with programmer-written contracts. Consider the test suite T:

$$\begin{aligned}
t_1&: \ \epsilon.\mathit{make} : l_1 \rightsquigarrow \langle \epsilon \rangle\\
t_2&: \ l_1.\mathit{wipe\_out} : \epsilon \rightsquigarrow \langle l_1 \rangle\\
t_3&: \ l_1.\mathit{append}\ (l_1)\colon \epsilon \rightsquigarrow \langle l_1, l_1 \rangle\\
t_4&: \ l_1.\mathit{extend}\ (l_1)\colon \epsilon \rightsquigarrow \langle l_2, l_2 \rangle\\
t_5&: \ l_1.\mathit{is\_empty}\colon b_3 \rightsquigarrow \langle l_1 \rangle\\
t_6&: \ \epsilon.\mathit{make} : l_4 \rightsquigarrow \langle \epsilon \rangle
\end{aligned}$$

where all test cases are passing. For simplicity, we do not introduce new duplicate objects in T when they are unchanged in the post-state with respect to the pre-state; for example, t_4 denotes a call to *extend* with l_1 as target and argument, and l_2 is the name given to the list after extending it, whereas t_5 does not change the target l_1 which is then repeated in the post-state. T induces the object pool $O = \{l_1, l_2, b_3, l_4\}$, where l_1, l_4 are empty lists, l_2 has one element (a reference to l_2 itself), and b_3 is the Boolean **True**.

C. Object/transition database

The object/transition database contains detailed information about the objects in the object pool $\mathcal{O}$. Section IV details how the database is implemented with relational database technology; the current section describes how stateful testing selects and extracts the information to store in the database.

Abstract object states and transitions. The object/transition database stores all objects in the pool $\mathcal{O}$ in serialized form. On top of the serialized objects, the database stores their *abstract state*, expressed in terms of the public *queries* (functions) of the objects. In the running example, class *LIST* has two public queries: *is_empty* and *has*. We would like to have information as extensive as possible in the database: for every combination of objects in the pool, evaluate every public query that is applicable. This is clearly unfeasible for object pools of non-trivial size, hence stateful testing uses a heuristic based on the usage of objects in the test suite $\mathcal{T}$. For an object $o \in \mathcal{O}$, consider the set $\rho(o)$ of objects *reachable* from references in o, including o itself; because of how the object pool is defined, $\rho(o)$ is a subset of $\mathcal{O}$. Extend the notation to objects reachable from a set of objects: $\rho(O) = \bigcup_{o \in O} \rho(o)$. For every test case $t = a_0.r\,(a_1, \ldots, a_m) : b \rightsquigarrow \langle a'_0, \ldots, a'_m \rangle$, the database stores *all the applicable public queries* q:

$$\alpha_0.q\,(\alpha_1, \ldots, \alpha_n) : \beta \qquad (1)$$

$$\omega_0.q\,(\omega_1, \ldots, \omega_n) : \psi \qquad (2)$$

where $\alpha_0, \alpha_1, \ldots, \alpha_n$ range over the set $\rho(a_0, a_1, \ldots, a_m)$ of object reachable in t's pre-state, and $\omega_0, \omega_1, \ldots, \omega_n$ range over the set $\rho(b, a'_0, a'_1, \ldots, a'_m)$ of object reachable in t's post-state. Precisely, for every call of the form (1) or (2), the database adds the objects β, ψ in serialized form and includes a tuple with q's signature, and references to the serialized objects $\alpha_0, \ldots, \alpha_n, \omega_0, \ldots, \omega_n, \beta, \psi$ stored in the database. The current tool implementation supports queries of generic return type; for simplicity, the presentation in this paper only considers queries that return Boolean values.

The database also stores information about *transitions*: each transition associates the routine r with several pairs of query evaluations; the first element of the pair evaluates a query in the pre-state (1), and the second evaluates it in the post-state (2). Then, the transition represents the fact that calling r when the pre-state holds can drive the object to the post-state.

Continuing the running example (Listing 1), the test cases t_1, t_2, t_3 only mention the list object l_1, which produces the queries l_1.*is_empty*: **True** and l_1.*has* (l_1): **False**. t_4 introduces the object l_2, hence the new queries l_2.*is_empty*: **False**, l_1.*has* (l_2): **False**, l_2.*has* (l_1): **False**, l_2.*has* (l_2): **True**. t_6 introduces two more queries on l_4: l_4.*is_empty*: **True** and *l4*.*has* (*l4*): **False**. Finally, t_4 induces the only non-trivial transitions from a pre-state where *is_empty* evaluates to **True** and *has* to **False**, to a post-state where both queries change their returned value when evaluated on the changed target.

Public branch and path conditions. To increase the precision of the abstract states stored in the database, stateful testing includes the value of several Boolean expressions extracted from the program text. For every test case t exercising a routine r, collect all the Boolean expressions $e_1, e_2, \ldots$ that appear as *branch conditions* or as *path conditions* in r's control-flow graph, and that only reference *public features* (members) of r's containing class. The rationale for storing branch and path conditions is that they often offer "interesting" partitions of the input states. The database stores the evaluations of these expressions for each applicable combination of objects reachable in the pre-state and in the post-state of every test t.

D. Dynamic contract inference

To get a concise characterization of the test suite $\mathcal{T}$ in terms of class features, stateful testing performs *contract inference* with dynamic techniques. The implementation uses AutoInfer [3], the inference component of the AutoTest framework.

Contract inference only considers the passing test cases from the suite $\mathcal{T}$ and produces, for each routine r exercised in the test suite, a list $pre(r)$ of preconditions and a list $post(r)$ of postconditions. These inferred contracts summarize r's behavior with the test cases in $\mathcal{T}$: for every passing test $t = a_0.r\,(a_1, \ldots, a_m) : b \rightsquigarrow \langle a'_0, \ldots, a'_m \rangle$ in $\mathcal{T}$, the arguments $a_1, \ldots, a_m$ and the target satisfy all preconditions in $pre(r)$, and the result b (if any) and post-state $a'_0, a'_1, \ldots, a'_m$ satisfy all postconditions in $post(r)$.

In the running example (Listing 1), *wipe_out* is always invoked on an empty list, hence *is_empty* is an inferred precondition in pre(*wipe_out*); *append* is invoked once on an empty list which is still empty after the call, hence **old** *is_empty* **implies** *is_empty* is a postcondition in $post$(*append*), and *other*.*is_empty* is a precondition in pre(*append*).[1]

The inferred contracts are typically different than those programmers write: the former tend to be more detailed and numerous than the latter, especially in the case of postconditions, which programmers neglect but dynamic analysis is effective at reporting [3], [10]. Furthermore, dynamically inferred contracts have no guarantee of being correct: since they are based on a finite number of observations, they may merely be a reflection of a not sufficiently varied test suite, such as the two examples discussed in the previous paragraph.

E. Reduction

After building the object/transition database and collecting the inferred contracts, stateful testing generates a new test suite by *precondition reduction*. The basic idea is partitioning the input space: a predicate p defines two regions, one where p holds and one where it doesn't; a comprehensive test suite should cover every region, for every combination of "interesting" predicates, with at least one test case. This is clearly

[1] Dynamic inference does not really infer contracts based on so few test cases because they are statistically insignificant; the example is only for illustration purposes.

unfeasible, because the predicates are too many; precondition reduction is a heuristic technique that considers a reduced number of partitions based on the inferred preconditions.

1) Precondition reduction: The *precondition reduction* of a routine r generates new inputs to test r by trying to invalidate r's inferred preconditions. Suppose r has m arguments, and let $\mathbf{require}(r)$ denote r's programmer-written preconditions. Select a dynamically inferred precondition p from the set $pre(r)$ and build the predicate:

$$\clubsuit^r_p : \quad \neg p \;\wedge\; \mathbf{require}(r)\,.$$

$\clubsuit^r_p$ characterizes objects that satisfy r's programmer-written preconditions but violate the inferred p, hence they can be used to test r in a way not covered by the existing test suite.

Stateful testing searches the object/transition database for tuples of objects $\langle o_0, o_1, \ldots, o_m\rangle$ that satisfy $\clubsuit^r_p$ (expressed as a conjunction of elementary expressions). In the running example (Listing 1), *wipe_out*'s inferred precondition *is_empty* suggests to search for objects of type *LIST* satisfying **not** *is_empty* (*wipe_out* has no programmer-written precondition); l_2 satisfies the search criterion.

For each tuple $\langle o_0, o_1, \ldots, o_m\rangle$ retrieved in the search, stateful testing constructs the new test case

$$t^{\text{new}} = o_0.r\,(o_1, \ldots, o_m)\,.$$

In practice, there is a cut-off on the number of retrieved tuples (if they are too many, only a few are tried) and a time-out on the time spent searching the database (if no tuple is found by the time-out, we move to the next reduction). If t^{new} is passing, then the precondition p is unsound and removed from $pre(r)$; if t^{new} is failing, an error is found (and p is also unsound). Since the information stored in the database is incomplete, t^{new} may also be invalid, in which case it is simply discarded. In the running example, the list l_2 is not empty and the test case l_2.*wipe_out* exercises *wipe_out* in ways not tested before.

2) Using transitions: If the search for objects satisfying $\clubsuit^r_p$ fails, stateful testing tries to retrieve objects satistying a weaker predicate than $\clubsuit^r_p$, and then it searches for a transition that drives the objects to match the desired $\clubsuit^r_p$. To this end, put $\clubsuit^r_p$ in conjunctive normal form $c_1 \wedge \cdots \wedge c_n$ and select $1 \leq d < n$ clauses to drop; without loss of generality, we drop $c_1, \ldots, c_d$ (D) and we keep $c_{d+1}, \ldots, c_n$ (K):

$$\clubsuit^r_p \equiv \quad \underbrace{c_1 \wedge \cdots c_d}_{D} \wedge \underbrace{c_{d+1} \wedge \cdots \wedge c_n}_{K}\,.$$

For any tuple of objects $\langle o_0, \ldots, o_m\rangle$ satisfying K, search the object/transition database for *transitions* that can transform a tuple $\langle o_0, \ldots, o_m\rangle$ satisfying $\neg D$ into a tuple $\langle o'_0, \ldots, o'_m\rangle$ satisfying D. Every such transition consists of a routine s and a mapping $\mu : [0..n] \to [0..m]$, where s has $n \geq 0$ arguments. μ binds the objects $o_0, \ldots, o_m$ to s's target and arguments: the i-th argument is instantiated with $o_{\mu(i)}$. For every such transition, construct the new test case

$$t^{\text{new}} = o_{\mu(0)}.s\,(o_{\mu(1)}, \ldots, o_{\mu(n)})\;;\; o_0.r\,(o_1, \ldots, o_m)\,,$$

consisting of two consecutive calls.

In the *TWO_WAY_TREE* example in Section II-C, the dropped clause D is **not Current**.*off* and the kept clause K is **Current**. *is_sibling* (*other*). Two objects *o1*, *o2* in the database satisfy *o1*. *is_sibling* (*o2*), and a transition suggests that routine *start* can change *o1* from *o1*.*off* to **not** *o1*.*off*.

The search for transitions is heuristic: since the information about transitions in the database is incomplete in general, the routine s may be inapplicable to the objects $\langle o_0, \ldots, o_m\rangle$, or it may not drive them in a state satisfying $\clubsuit^r_p$—for example, because it invalidates K as a side-effect of satisfying D. In practice, the heuristic search is reasonably successful when there are objects whose state is close to satisfying $\clubsuit^r_p$; correspondingly, the current implementation drops at most one clause ($d = 1$), and does not build sequences of transitions with more than two calls.

3) Detecting unsound postconditions: Inferred postconditions can be unsound, too, but we cannot directly select objects that violate postconditions, because we do not have direct control over post-states. Precondition reduction, however, can also help to invalidate inferred postconditions, while testing routines more thoroughly. Consider an inferred postcondition q in $post(r)$ in the form:

$$q : \quad \textbf{old}(A) \Longrightarrow C\,.$$

We focus on postconditions in this form, because q naturally expresses many postconditions where a property C of the post-state is a consequence of a property A of the pre-state (**old**). Invalidating the implication q means producing test cases that start in a pre-state where A holds and reach a post-state where $\neg C$ holds. The existing test suite does not include such test cases, otherwise P would not be a dynamically inferred postcondition.

The inferred *preconditions*, however, help select pre-states that may challenge the validity of q. To this end, consider the set $pre(r|A)$ of r's dynamically inferred preconditions that hold when A also holds. Select a $p \in pre(r|A)$ among these preconditions and build the predicate:

$$\spadesuit^r_{p,q} : \quad A \;\wedge\; \clubsuit^r_p\,.$$

Then, select (or build with transitions) objects $\langle o_0, \ldots, o_m\rangle$ that satisfy $\spadesuit^r_{p,q}$, and generate the new test case t^{new} that calls r on $\langle o_0, \ldots, o_m\rangle$ (as in Section III-E1). If t^{new} is valid but C is false after executing it, the postcondition q is unsound and is removed from $post(r)$; if t^{new} is failing, it also shows an error.

In the example of Listing 1, stateful testing targets the inferred postcondition **old** *is_empty* **implies** *is_empty* of routine *append*, which is in the form q: A is *is_empty*; for the same routine, *other*.*is_empty* is a precondition inferred when A also holds. Hence, stateful testing looks for two lists, one empty and one not; l_1, l_2 satisfy the criterion and yield the new test case l_1.*append* (l_2).

IV. Object/transition database

Section III-C describes what kind of information the object/transition database stores; the present section details how the database is implemented with relational technology.

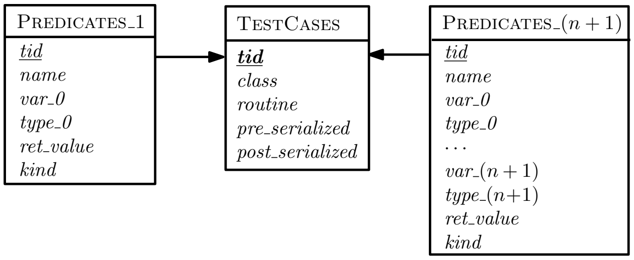

Fig. 3. Relational schema of the object/transition database.

A. Relational schema

Figure 3 shows the most significant parts of the object/transition database's relational schema. The database is centered around the test cases in the test suite $\mathcal{T}$: for each test case $a_0.r\,(a_1,\ldots,a_m) : b \rightsquigarrow \langle a'_0,\ldots,a'_m\rangle$ table TESTCASES stores: (1) a unique identifier (attribute *tid*), (2) r's class (*class*), (3) r's name (*routine*), (4) the list $a_0, a_1, \ldots, a_m, a_{m+1}, \ldots, a_n$ of all serialized objects in the pre-state followed by all objects reachable from them (*pre_serialized*), (5) the list $a'_0, a'_1, \ldots, a'_m, b, a'_{m+1}, \ldots, a'_{n'}$ of all serialized objects in the post-state followed by all objects reachable from them (*post_serialized*). We do not discuss the straightforward details of how lists of serialized objects are encoded as sequences of characters with separators.

The other tables PREDICATES_1, PREDICATES_2, ..., PREDICATES_9 store information about the abstract state of objects, in the form of predicates over 1, 2, ..., 9 objects. Consider an atomic Boolean predicate q evaluated over $n+1$ objects, in the form $o_0.q\,(o_1,\ldots,o_n) : v$, that holds for the pre-state of a test case with identifier x. Table PREDICATES_$(n+1)$ stores an entry with: (1) a reference to the test case x (attribute *tid*), (2) the normalized textual form of the predicate, obtained by replacing every reference to objects with '\$' placeholders as in $\$.q\,(\$,\ldots,\$)$ (attribute *name*), (3) for each object o_i, $0 \leq i \leq n+1$, an integer k_i such that the k_i-th element of the list in the *pre_state* attribute of the test case with *tid* x contains o_i (attribute *var_i*), (4) for each object o_i, $0 \leq i \leq n{+}1$, its dynamic type (*type_i*), (5) the Boolean value v returned (attribute *ret_value*), (6) the constant *pre* to denote that q is evaluated over the pre-state (attribute *kind*). For predicates evaluated in the post-state, attribute *kind* stores the constant *post*, and everything else is like for pre-states.

Consider, for example, the test case t_4 in the running example (Section III-B): $l_1.\mathit{extend}(l_1) : \epsilon \rightsquigarrow \langle l_2, l_2\rangle$. Table TESTCASES stores a tuple $\langle id_4, \mathit{LIST}, \mathit{extend}, \pi, \Pi\rangle$, where id_4 is the unique identifier, π is the list of (serialized) objects in the pre-state: $\pi = [l_1, l_1]$, and Π is the list of objects in the post-state: $\Pi = [l_2, l_2]$. t_4 induces, among others, the evaluation of the query $l_1.\mathit{has}(l_1)$ in the pre-state, which table PREDICATES_2 stores as the tuple

$$\langle id_4, \$.\mathit{has}(\$), 0, \mathit{LIST}, 0, \mathit{LIST}, \mathbf{False}, \mathit{pre}\rangle$$

where the entries 0 in attributes *var_0* and *var_1* refer to the first element in the 0-indexed list π of serialized objects (i.e., object l_1 in serialized form).

B. Quering the database

The translation of predicates into SQL queries to the object/transition database is straightforward:

- Objects become variables in the **SELECT** clause;
- These variables are joined with the PREDICATES and TESTCASES tables in the **FROM** clause;
- The **WHERE** clause encodes the constraints on the individual predicates, and SQL Boolean operators map the Boolean connectives in the translated predicate.

Let us demonstrate the creation of SQL queries with the example at the end of Section III-E3, where we search for two objects l_1, l_2 of type *LIST* such that l_1.*is_empty* and **not** l_2.*is_empty*. We create the SQL query in Listing 2 that searches for such objects in pre-states (the query for post-states is all similar). The SQL query returns a tuple *objs1*, *objs2*, *idx1*, *idx2* such that *objs1*, *objs2* are collections of serialized objects and *idx1*, *idx2* are integer indices: the *idx1*-th element in collection *objs1* is an empty list, and the *idx2*-th element in collection *objs2* is a non-empty list.

Listing 2. An SQL query searching for two lists.

```
SELECT
 t1. pre_serialized as objs1, t2. pre_serialized as objs2,
 p1.var_0 as idx1, p2.var_0 as idx2
FROM
 Predicates_1 p1 join TestCases t1 on p1.tid = t1.tid,
 Predicates_1 p2 join TestCases t2 on p2.tid = t2.tid
WHERE
 p1.name = '$.is_empty'                 AND
 p1.type_0 = 'LIST'                     AND
 p1. ret_value AND p1.kind = 'pre'      AND

 p2.name = '$.is_empty'                 AND
 p2.type_0 = 'LIST'                     AND
 NOT (p2.ret_value) AND p2.kind = 'pre'
```

V. EVALUATION

This section presents the results of an experimental evaluation, summarized in Table I: the leftmost part of the table contains statistics about random testing, the middle part shows the performance of stateful testing with preconditions and the rightmost part with postconditions.

A. Experimental setup

The experiments targeted 13 Eiffel classes implementing data structures from the libraries EiffelBase [8] (revision 506) and Gobo [9] (revision 6665). Table I lists the size of each class in lines of code (LOC) and public routines (#R).

Each session in the preparation of the original test suite with random testing ran on a Linux node with a 2.53 GHz Intel Nehalem quad-core CPU and 8 GB of memory. The

TABLE I
CLASSES UNDER TEST AND RESULTS.

RANDOM TESTING				STATEFUL TESTING WITH PRECONDITIONS						STATEFUL TESTING WITH POSTCONDITIONS					
CLASS	LOC	#R	#E	#T$_p$	#U$_p$	#V$_p$		#E$_p$	#M$_p$	#T$_q$	#U$_q$	#V$_q$		#E$_q$	#M$_q$
ARRAY	1466	65	9	111	23	23	(100%)	2	52′	14	1	1	(100%)	0	5′
ARRAYED_QUEUE	1064	40	0	17	13	13	(100%)	0	7′	19	0	0	N/A	0	9′
ARRAYED_SET	2343	46	9	55	18	18	(100%)	1	25′	141	0	0	N/A	0	10′
BOUNDED_QUEUE	1130	40	0	20	16	16	(100%)	0	7′	22	0	0	N/A	0	9′
DS_ARRAYED_LIST	2760	104	5	178	107	107	(100%)	4	92′	170	16	11	(69%)	0	154′
DS_HASH_SET	3074	82	1	279	173	173	(100%)	2	40′	51	3	3	(100%)	0	5′
DS_LINKED_LIST	3432	100	5	196	120	120	(100%)	2	106′	129	1	0	(0%)	1	88′
DS_LINKED_STACK	934	28	0	39	38	38	(100%)	0	4′	4	0	0	N/A	0	1′
HASH_TABLE	2032	58	1	117	88	87	(99%)	1	16′	63	10	10	(100%)	0	30′
LINKED_LIST	1998	72	1	53	46	46	(100%)	0	9′	149	13	13	(100%)	1	22′
LINKED_SET	2366	80	13	91	47	47	(100%)	4	33′	176	15	15	(100%)	1	28′
TWO_WAY_SORTED_SET	2866	92	29	221	120	120	(100%)	15	49′	25	7	7	(100%)	0	2′
TWO_WAY_TREE	3316	107	22	364	203	198	(98%)	26	75′	10	3	0	(0%)	5	4′
Total	**28781**	**914**	**95**	**1741**	**1012**	**1006**	**(99.4%)**	**57**	**515′**	**973**	**68**	**60**	**(88.2%)**	**8**	**367′**

other experiments (contract inference and stateful testing) ran on an Ubuntu machine with a 1.73 GHz Intel Core i7 CPU and 8 GB of memory. The average speed of random testing, contract inference, and stateful testing on the two architectures is comparable.

1) Random testing: To generate the original test suite $\mathcal{T}$—upon which stateful testing builds—AutoTest ran 30 sessions of random testing for each of the 13 classes. A session lasts 80 minutes and initializes the pseudo-random number generator with a new seed. The 30 sessions totaled 520 hours of testing and generated a test suite with 149,293 distinct test cases. The test suite $\mathcal{T}$ revealed 95 distinct errors[2] (column #E of Table I).

2) Stateful testing running time: **Dynamic contract inference.** AutoInfer processed the test suite $\mathcal{T}$ for 16 hours and reported 1741 preconditions and 973 postconditions expressible as implications, shown in column #T$_p$ and #T$_q$ in Table I. Manual inspection revealed that 1012 (58%) of the inferred preconditions and 68 (7%) of the inferred postconditions are unsound. Columns #U$_p$ and #U$_q$ respectively report the number of unsound pre and postconditions for each class.

Object/transition database construction. Constructing the object/transition database from $\mathcal{T}$ took 5 hours. The database contains about 3.5 million objects, 18.4 million predicate evaluations, and 68.8 thousand transitions, and occupies 3.4 GB on disk.

Reduction. Notice that querying the object/transition database gives predictable results, hence the reduction is deterministic and needs to run only once. Stateful testing ran for 15 hours trying to violate the inferred pre and postconditions. The times (in minutes) spent on the pre and postconditions in each class are shown in columns #M$_p$ and #M$_q$ of Table I. In the experiments, every query times out after one minute.

B. Experimental results

In all, stateful testing discovered 65 new errors in the classes under test, corresponding to a 68.4% improvement over the number of errors found by random testing, with only a 7% time overhead (36/520 hours). Columns #E$_p$ and #E$_q$ in Table I respectively show the number of new errors detected while trying to violate the inferred pre and postconditions in each class. The performance in terms of number of unsound preconditions and postconditions detected is given below.

Building upon random testing, stateful testing detected 68.4% new errors in a fraction of the time.

1) Unsound preconditions: Table II gives an account of the most common structures of the inferred preconditions targeted in the experiments. Stateful testing tried to invalidate the 1741 inferred preconditions for 8.2 hours (i.e., about 18 seconds per precondition), following the technique in Section III-E1. It successfully invalidated 1006 (99.4%) of the unsound preconditions (column #V$_p$ of Table I, which also report the percentages relative to column #U$_p$), while exposing 57 new errors (column #E$_p$).

TABLE II
STRUCTURE OF INFERRED PRECONDITIONS.

STRUCTURE	EXAMPLE	#T
Reference equality	$o1 = o2$	154
Object equality	$o1.is\_equal(o2)$	329
Voidness check	$o \neq$ **Void**	7
Integer equality	$o.count = 0$	377
Boolean query with arguments	$o.has(v)$	483
Boolean query without arguments	$o.is\_empty$	356
Other (complex)	$o.full$ **or** $i < l.count$	35
Total		**1741**

2) Unsound postconditions: Stateful testing tried to invalidate the 973 inferred postconditions in implication form for 6 hours (i.e., about 23 seconds per postcondition), following the technique in Section III-E3. It successfully invalidated 60 (88.2%) of the unsound postconditions (column #V$_q$ of Table I, which also report the percentages relative to column #U$_q$), while exposing 8 new errors (column #E$_q$).

3) Undetected unsound contracts: Stateful testing only failed to detect 6 unsound preconditions (0.6% of the total) and 8 unsound postconditions (11.8%). In all such cases, no serialized objects were in a state violating the contract (or sufficiently close to it), or the predicates provided an abstraction of the object state that was too coarse-grained for the desired objects to be identifiable.

[2] Two errors are distinct if they violate two different contract clauses.

Stateful testing increased the soundness of inferred contracts from 60.2% to 99.5%.

VI. Limitations and future work

Stateful testing, and its current implementation, has some limitations to be addressed in future work.

- As it is customary in random testing [11], we have evaluated stateful testing on classes implementing data structures. This was also useful for comparison against out previous experience with AutoTest [1]. Further experiments will target different types of classes.
- Stateful testing can start from a test suite $\mathcal{T}$ generated manually or with any technique, all our experiments used test cases automatically generated. Further experiments will determine if the performance of stateful testing is affected by how the original test suite is generated.
- The current implementation of stateful testing adopts the following heuristics when retrieving objects and transitions from the database: (1) search for objects and consider the first 45 results; (2) if none of the 45 retrieved objects work, search for transitions and call them on the 45 result objects; (3) if none of the transitions work, give up and move to the next reduction. This heuristic worked quite well in the experiments, but further experience will determine if it can be improved and how robust the results are with respect to this heuristic.
- Future experiments will try to iterate the infer/reduce process on the new test suite generated with stateful testing. This will challenge the state of the art in dynamic invariant inference and is likely to suggest improvements to the techniques used in the process.

VII. Related Work

Xie and Notkin [5] first suggested a framework that combines test-case generation and dynamic specification inference with the goal of mutually enhancing their results.

Dallmeier et al. [12] implement Xie and Notkin's ideas for typestate specifications (finite-state automata describing abstract object states and transitions), and report an evaluation showing that their technique builds more accurate specification and finds more errors injected in Java applications than traditional dynamic analysis techniques [13]. Stateful testing is based on the same principles—applied to contract specifications—and extends them with the usage of a database to improve the reuse of previous testing sessions and to build new test cases. Typestates provide object-state abstractions—based on argumentless Boolean queries and simple integer partitioning—that are coarser-grained than the one deployed in the present paper; consequently, building a typestate model by exhaustive exploration is feasible, whereas our more detailed model requires heuristics and an efficient search of serialized objects to be built. We also provide an implementation and experimental evaluation.

Stateful testing combines diverse techniques of program analysis; the rest of this section summarizes some representative work involving these techniques. More comprehensive references are available in the bibliography of the cited work.

A. Automated test-case generation

Automated random testing [6] is now a well-understood technique which, in spite of the simplicity of its underlying ideas, is quite effective and can find subtle bugs [1]. Arcuri et al.'s analysis of random testing [2] analytically confirms the experimental results, and suggests that more sophisticated test-case generation techniques are best deployed *after* random testing exhausts its potential. Stateful testing is indeed applied following random testing sessions, to reuse the objects generated and find new inputs and errors.

Search-based test-case generation refines random testing with goal-driven searches in the space of test cases; McMinn [14] and Ali et al. [15] survey the state of the art in search-based techniques. Test suite augmentation [16] uses search-based techniques driven by coverage criteria, with the purpose of adapting a regression test suite to changed code. Genetic algorithms are a recurring choice to search for test inputs; Tonella [17] first suggested the idea, and Andrews et al. [18] show how to use genetic algorithms to optimize the performance of standard random testing. Stateful testing is also search-based, but the search takes place among previously generated objects, and it is guided by contracts that characterize an existing test suite to be improved.

Other refinements of random testing combine it with white-box techniques such as symbolic execution (e.g., [19]), or leverage the availability of formal specifications in various forms (see Hierons et al. [20] for a survey). Stateful testing also makes extensive usage of specifications in the form of contracts, both inferred and written by programmers.

In previous work [21], we developed *precondition satisfiaction*, a search strategy that improves the selection of objects to test routines with complex preconditions. This technique is included in AutoTest, and all the random testing session that preceded stateful testing (in the experiments of Section V) deployed precondition satisfaction.

B. Dynamic specification inference

Daikon [22] pioneered the dynamic inference of specifications and program invariants, and showed that assertions "guessed" based on a finite number of runs are often sound with respect to generic executions. Since the first Daikon release, dynamic inference has been applied to other specification models (e.g., typestates [23]) and has improved its accuracy (such as in our own AutoInfer [3]).

Gupta and Heidepriem [24] suggest to improve the quality of inferred contracts by using different test suites (i.e., based on code coverage and invariant coverage), and by retaining only the contracts that are inferred with both techniques. Fraser and Zeller [25] simplify and improve test cases based on mining recurring usage patterns in code bases; the simplified tests are easier to understand and focus on common usage. Other approaches to improve the quality of inferred contracts combine static and dynamic techniques (e.g., [26], [27]).

Stateful testing leverages dynamic contract inference techniques, and tries to violate inferred contracts to explore new regions of the input state space; this not only improves the test suite, but it also detects many unsound contracts. Stateful testing also includes the results of some lightweight static analysis (based on the branching structure of routines) to gather more information about the available test suite.

C. Search-based techniques

The idea of constructing "semantic" databases, with a uniform search interface to retrieve programs [28], program elements [29], [30], or test cases [31] with specific characteristics has recently been deployed, mostly to help developers organize their code and reuse products written by others.

The object capture technique [32] stores serialized objects, created during program executions, and reuses them as new test inputs to reach uncovered branches. Stateful testing also stores serialized objects and reuses them to create new test cases; the object capture framework, however, only supports searching for objects based on their types, whereas stateful testing stores rich information about the objects' abstract states, as well as transitions between states. Another significant difference is that stateful testing targets the mutual improvement of test suites and inferred contracts, whereas object capture is only concerned with improving branch coverage.

VIII. Conclusions

This paper presented *stateful testing*, a completely automated testing technique which generates new test cases from an existing test suite. Stateful testing works by trying to *reduce* (i.e., invalidate) the inferred contracts that characterize the existing test suite. Extensive experiments show that stateful testing is quite effective: it generates tests that uncover new errors and invalidates many of the unsound contracts inferred dynamically from the original test suite.

Stateful testing is part of the automated testing framework AutoTest; the source code of AutoTest—including the basic testing infrastructure, dynamic contract inference, and the stateful testing implementation—detailed experimental results and instructions to reproduce the experiments are available at:

http://se.inf.ethz.ch/research/autotest

Acknowledgments. Nadia Polikarpova suggested the idea—and the name—of precondition reduction, and provided valuable comments on a draft of this paper. This work has been partially funded by the Swiss National Science Foundation under the projects SATS and ASII (SNF 200021-117995 and 200021-134976); it also benefited from funding by the Hasler foundation on related projects. The facilities of the Swiss National Supercomputing Centre (CSCS) helped to generate the large-scale test suite used in the experiments.

References

[1] I. Ciupa, A. Pretschner, M. Oriol, A. Leitner, and B. Meyer, "On the number and nature of faults found by random testing," *Softw. Test., Verif. Reliab.*, vol. 21, no. 1, pp. 3–28, 2011.

[2] A. Arcuri, M. Z. Iqbal, and L. Briand, "Formal analysis of the effectiveness and predictability of random testing," in *ISSTA*. ACM, 2010.

[3] Y. Wei, C. A. Furia, N. Kazmin, and B. Meyer, "Inferring better contracts," in *ICSE'11*. ACM, 2011.

[4] Y. Wei, Y. Pei, C. A. Furia, L. S. Silva, S. Buchholz, B. Meyer, and A. Zeller, "Automated fixing of programs with contracts," in *ISSTA*. ACM, 2010, pp. 61–72.

[5] T. Xie and D. Notkin, "Mutually enhancing test generation and specification inference," in *FATES*, ser. LNCS, vol. 2931, 2003, pp. 60–69.

[6] B. Meyer, A. Fiva, I. Ciupa, A. Leitner, Y. Wei, and E. Stapf, "Programs that test themselves," *IEEE Software*, pp. 22–24, 2009.

[7] http://eve.origo.ethz.ch/.

[8] http://freeelks.svn.sourceforge.net.

[9] http://sourceforge.net/projects/gobo-eiffel/.

[10] N. Polikarpova, I. Ciupa, and B. Meyer, "A comparative study of programmer-written and automatically inferred contracts," in *ISSTA*, 2009, pp. 93–104.

[11] R. Sharma, M. Gligoric, A. Arcuri, G. Fraser, and D. Marinov, "Testing container classes: Random or systematic?" in *FASE*, ser. LNCS, vol. 6603, 2011, pp. 262–277.

[12] V. Dallmeier, N. Knopp, C. Mallon, S. Hack, and A. Zeller, "Generating test cases for specification mining," in *ISSTA*, 2010, pp. 85–96.

[13] V. Dallmeier, A. Zeller, and B. Meyer, "Generating fixes from object behavior anomalies," in *ASE*, 2009, pp. 550–554.

[14] P. McMinn, "Search-based software test data generation: a survey," *Softw. Test., Verif. Reliab.*, vol. 14, no. 2, pp. 105–156, 2004.

[15] S. Ali, L. C. Briand, H. Hemmati, and R. K. Panesar-Walawege, "A systematic review of the application and empirical investigation of search-based test case generation," *IEEE Trans. Software Eng.*, vol. 36, no. 6, pp. 742–762, 2010.

[16] Z. Xu, Y. Kim, M. Kim, G. Rothermel, and M. B. Cohen, "Directed test suite augmentation: techniques and tradeoffs," in *FSE*. ACM, 2010, pp. 257–266.

[17] P. Tonella, "Evolutionary testing of classes," in *ISSTA*, 2004, pp. 119–128.

[18] J. H. Andrews, T. Menzies, and F. C. H. Li, "Genetic algorithms for randomized unit testing," *IEEE Trans. Software Eng.*, vol. 37, no. 1, pp. 80–94, 2011.

[19] R. Majumdar and K. Sen, "Hybrid concolic testing," in *ICSE*, 2007, pp. 416–426.

[20] R. M. Hierons, K. Bogdanov, J. P. Bowen, R. Cleaveland, J. Derrick, J. Dick, M. Gheorghe, M. Harman, K. Kapoor, P. Krause, G. Lüttgen, A. J. H. Simons, S. A. Vilkomir, M. R. Woodward, and H. Zedan, "Using formal specifications to support testing," *ACM Comput. Surv.*, vol. 41, no. 2, 2009.

[21] Y. Wei, S. Gebhardt, M. Oriol, and B. Meyer, "Satisfying test preconditions through guided object selection," in *ICST*, 2010, pp. 303–312.

[22] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin, "Dynamically discovering likely program invariants to support program evolution," *IEEE Trans. on Soft. Eng.*, vol. 27, no. 2, pp. 99–123, 2001.

[23] V. Dallmeier, C. Lindig, A. Wasylkowski, and A. Zeller, "Mining object behavior with ADABU," in *WODA '06*. ACM, 2006, pp. 17–24.

[24] N. Gupta and Z. V. Heidepriem, "A new structural coverage criterion for dynamic detection of program invariants," in *ASE*, 2003, pp. 49–59.

[25] G. Fraser and A. Zeller, "Exploiting common object usage in test case generation," in *ICST*, 2011.

[26] C. Csallner, N. Tillmann, and Y. Smaragdakis, "DySy: Dynamic symbolic execution for invariant inference," in *ICSE*, 2008, pp. 281–290.

[27] N. Tillmann, F. Chen, and W. Schulte, "Discovering likely method specifications," in *ICFEM*, ser. LNCS, vol. 4260. Springer, 2006, pp. 717–736.

[28] M. Grechanik, C. Fu, Q. Xie, C. McMillan, D. Poshyvanyk, and C. Cumby, "A search engine for finding highly relevant applications," in *ICSE*. ACM, 2010, pp. 475–484.

[29] M. Würsch, G. Ghezzi, G. Reif, and H. C. Gall, "Supporting developers with natural language queries," in *ICSE*. ACM, 2010, pp. 165–174.

[30] S. P. Reiss, "Semantics-based code search," in *ICSE*, 2009, pp. 243–253.

[31] A. Holzer, C. Schallhart, M. Tautschnig, and H. Veith, "How did you specify your test suite," in *ASE*. ACM, 2010, pp. 407–416.

[32] H. Jaygarl, S. Kim, T. Xie, and C. K. Chang, "OCAT: Object capture based automated testing," in *ISSTA*, 2010.