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 faults, 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 faults 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 faults in real programs.

Random testing sessions must last several hours to maximize fault-finding effectiveness [1], [2]. During testing, all generated objects are usually kept in a pool for later reuse. 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 faults 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.

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. 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 [4]. 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 a fault 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.

We implemented stateful testing within our AutoTest [1] framework for random testing of object-oriented Eiffel applications. 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 data structure classes. AutoTest exposed 95 faults in the classes, and inferred hundreds of 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 faults (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.

II. Examples

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

$$\textit{merge}\ (\textit{other}:\ \textit{TWO\_WAY\_SORTED\_SET})$$

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*:

$$\textit{pre\_1}:\ \textbf{Current}.\textit{disjoint}\ \ (\textit{other})\ ,$$

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:

$$\begin{array}{ll} (1) & \textit{o1.type}\ = \textit{TWO\_WAY\_SORTED\_SET}\ , \\ (2) & \textit{o2.type}\ = \textit{TWO\_WAY\_SORTED\_SET}\ , \\ (3) & \textbf{not}\ \textit{o1.}\ \textit{disjoint}\ \ (\textit{o2})\ . \end{array}$$

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.

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*:

$$\begin{array}{l} \textit{post\_2}:\ \textbf{old}\ \textbf{Current}.\textit{is\_equal}\ \ (\textit{other}) \\ \qquad\qquad\qquad \textbf{implies}\ \textbf{Current}.\textit{is\_empty}\ . \end{array}$$

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*.

III. How Stateful Testing Works

Figure 1 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 database*.
3) AutoInfer [3], the dynamic contract inference component of AutoTest, summarizes the behavior of the test cases 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 faults* in the code under test, and can reveal which of the inferred *contracts* are *incorrect* and should be discarded.

A. Random Test-case Generation

Stateful testing first applies random testing to generate a test suite for classes under test. During random testing, stateful testing keeps all objects in serialized form along with their abstract state predicate evaluations. After random testing, stateful testing stores all the objects and their abstract states in an *object database*, implemented using a relational database techinque.

B. Dynamic Contract Inference

To get a concise characterization of the test suite in terms of class features, stateful testing performs *contract inference* with dynamic techniques. The implementation uses AutoInfer, the inference component of the AutoTest framework.

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 [5]. 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 section.

C. Reduction

After building the object 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

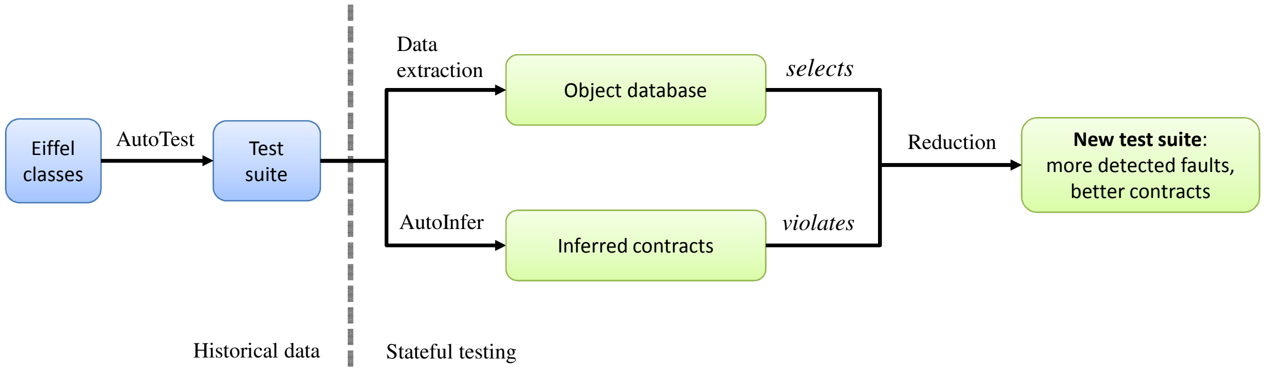

Fig. 1. Overview of how stateful testing works.

at least one test case. This is clearly 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 database for tuples of objects $\langle o_0, o_1, \dots, o_m\rangle$ that satisfy $\clubsuit^r_p$ (expressed as a conjunction of elementary expressions). For each tuple $\langle o_0, o_1, \dots, o_m\rangle$ retrieved in the search, it constructs the new test case

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

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 precondition p is unsound and removed from $pre(r)$; if t^{new} is failing, a fault 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.

2) 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:

$$q : \quad \mathbf{old}(A) \implies C\,.$$

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 objects $\langle o_0, \dots, o_m\rangle$ that satisfy $\spadesuit^r_{p,q}$, and generate the new test case t^{new} that calls r on $\langle o_0, \dots, o_m\rangle$ (as in Section III-C1). If t^{new} is valid and passing (with respect to r's programmer-written contracts only) but C is false after executing it, the postcondition q is unsound and is removed from $post(r)$; if t^{new} is failing (again with respect to r's programmer-written contracts, which are always assumed correct), it also shows a fault.

IV. Evaluation

A. Experimental Setup

The experiments targeted 13 data structure classes from the libraries EiffelBase and Gobo. Table I lists the size of each class in lines of code (LOC) and public routines (#R).

1) Random Testing: To generate the original test suite—upon which stateful testing builds—AutoTest ran 30 sessions of random testing for each of the 13 classes. A session lasts 80 minutes. This 520 hours of testing generated a test suite revealing 95 distinct faults[1] (column #E of Table I).

2) Stateful Testing Running Time: AutoInfer processed the test suite 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

[1] Two faults are distinct if they violate two different contract clauses.

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**	**$367'$**

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. Constructing the object database from the test suite took 5 hours. The database contains about 3.5 million objects and 18.4 million predicate evaluations. Notice that querying the object 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 faults in the classes under test, corresponding to a 68.4% improvement over the number of faults 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 faults 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.

1) Unsound Preconditions: 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-C1. 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 faults (column #E$_p$).

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-C2. 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 faults (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.

V. 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 faults and invalidates many of the unsound contracts inferred dynamically from the original test suite. An extended version of this paper, including analysis of related work, is available at:

http://arxiv.org/abs/1108.1068

Acknowledgments. Nadia Polikarpova provided suggestions and comments. Work partially funded by SNF projects 200021-117995 and 200021-134976 and by the Hasler foundation on related projects; the experiments used the facilities of the Swiss National Supercomputing Centre.

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

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