Cite this paper as follows: Lisa (Ling) Liu, Bertrand Meyer and Bernd Schoeller, *Using contracts and boolean queries to improve the quality of automatic test generation*, in proceedings of *TAP: Tests And Proofs*, ETH Zurich, 5-6 February 2007, eds. Yuri Gurevich and Bertrand Meyer, Lecture Notes in Computer Science, Springer-Verlag, 2007, to appear.

Using contracts and boolean queries to improve the quality of automatic test generation

Lisa (Ling) Liu Bertrand Meyer Bernd Schoeller

Chair of Software Engineering,
ETH Zurich, Switzerland
{ling.liu, Bertrand.Meyer, bernd.schoeller}@inf.ethz.ch

Abstract. Since test cases cannot be exhaustive, any effective test case generation strategy must identify the execution states most likely to uncover bugs. The key issue is to define criteria for selecting such interesting states.

If the units being tested are classes in object-oriented programming, it seems attractive to rely on the *boolean queries* present in each class, which indeed define criteria on the states of the corresponding objects, and — in contract-equipped O-O software — figure prominently in preconditions, postconditions and invariants. As these queries are part of the class specification and hence relevant to its clients, one may conjecture that the resulting partition of the state space is also relevant for tests.

We explore this conjecture by examining whether relying on the boolean queries of a class to extract abstract states improves the results of black-box testing. The approach uses proof techniques to generate objects that satisfy the class invariants, then performs testing by relying on postconditions as test oracles.

1 Overview

Unlike other approaches to improving program quality, in particular proofs, program testing focuses not on guaranteeing the absence of bugs but on uncovering bugs. This is by itself a very interesting goal since any bug removed is a significant improvement to a program.

The effectiveness of a testing strategy is, as a result, defined by how likely it is to uncover bugs. We present a testing strategy for classes — object-oriented program units — that takes advantage of two of their distinctive properties: the presence of *boolean queries* as part of the interface of a class, and in some programming formalisms, the use of *contracts* to specify abstract properties of classes.

The topic of this paper is, as a consequence, simple. We state a conjecture: that using contracts and queries will improve the effectiveness of testing strategies. Then we assess the validity of that conjecture by applying contract- and query-based testing through our automatic test environment, AutoTest [C2], and measuring whether this improves AutoTest's effectiveness in finding bugs.

A characteristic of our testing work is that (rather than artificial examples, although one will be used to illustrate the concepts) it applies testing strategies and in particular the AutoTest tool to actual production software, in particular the EiffelBase library of fundamental data structures and algorithms, used daily in mission-critical production environments. Testing for us is then not just an academic pursuit but also a very practical attempt to find bugs in actual software. Along with the concepts we propose, the main concrete result of the study reported here is that it has enabled us to find and correct real bugs in software components that are in actual use, and hence provide a tangible benefit to the users of those components.

1.1 Correctness and contracts

The correctness of a program element is not an absolute property but is always defined with respect to a certain specification. In the "Design by Contract" approach [M2], the specification is present in the text of classes (the program units of object-oriented programming) in the form of invariants for classes, and preconditions and postconditions for routines[1]. Ascertaining the correctness of a class in languages that natively support such mechanisms —Eiffel [M3] or Spec# [B1] —, or in contract add-ons to Java (such as JML [L1, L2], iContract [K1]) or UML (Object Constraint Language [R1, H1]), means ascertaining that the implementations are consistent with the contracts: specifically, that every creation procedure (constructor) yields an object satisfying the invariant of its class, and that every exported routine, started in a state satisfying the invariant and the precondition, terminates in a state satisfying the invariant and the postcondition.

Using testing we cannot prove such correctness for any realistic program, but we can uncover correctness violations — bugs — by finding inputs that will cause routine executions to violate an invariant or postcondition.

1.2 Testing and program states

Because the set of possible program execution states is inexhaustible, any practical testing strategy must identify a subset of interesting states, where "interest" is defined — in the negative mindset that characterizes the work of the tester, whose reward is to prove software *in*correct — as likelihood to uncover bugs. Usually this is achieved through a *partitioning* approach which, using appropriate criteria, divides the state space into disjoint parts, then picks one state (or a few) from each such part, with the expectation that each selected state is somehow representative of that part.

A common approach for such partitioning is to use white-box tests, based on an analysis of the implementation's control flow, such as "path coverage" and "branch coverage". This has two disadvantages. First, the tester needs access to the implementation, which may be an unrealistic requirement in the presence of information hiding. Second, the approach requires possibly complex computation of weakest preconditions to exercise specific branches or paths.

1.3 Query-based testing

The approach described here relies instead on a black-box testing strategy, based on contracts. Specifically:

- Instead of relying on the implementation of a class, it uses its contracts and its boolean queries to partition the state space.
- The partitioning is aided by an insight into the structure of good contracts, the *Boolean Query Conjecture*, defined below.

[1] "Routines" are called "methods" in Java and C++. This paper uses Eiffel terminology and notation.

- Techniques from boolean constraint solving and program proving help reduce the resulting state space further.
- Then we develop a test strategy - boolean query coverage to achieve complete test coverage based on the outcomes of this reasoning.

The main contributions of this paper are the following:

1. New application of Design by Contract techniques to improve the testing process.
2. A new method for partitioning program state, applied here to testing but (we think) with potential applications here, for example in model checking.
3. The experimental validation of that method on concrete examples.
4. New techniques for improving test coverage.
5. The integration of constraint-solving and program-proving techniques in a testing framework.
6. A technique for taking advantage test results to improve not only test coverage but also class designs (through stronger invariants).
7. Concretely, as noted, the detection through an automatic procedure (and subsequent correction), in actual production libraries, of real bugs, until now unsuspected and not found by any previous technique, manual or automatic.

Section 2 presents the notion of boolean query and introduces the conjecture behind this work's approach to testing, as well as the method for assessing the conjecture. Section 3 explains the overall strategy based on contracts, the notion of abstract state space, constraint satisfaction techniques, proof techniques, and the AutoTest framework. Section 4 describes the experimental study applying this strategy to a set of actual classes, and analyzes the result. Section 5 discusses related work, and section 6 discusses future work.

2. The role of boolean queries

The central issue of test case generation is, as noted above, to maximize the likelihood of uncovering bugs. If we are testing object-oriented software we should take advantage of the distinctive structure of O-O programs.

2.1 Classes and object states

A class is an implementation of an abstract data type, providing all the operations, or "features", on a certain type of run-time objects. These features are of two kinds [M2]:

- **Commands** modify the corresponding object: withdraw money (for a class representing bank accounts), open (for a class representing files), increase indent (for a class representing paragraphs in a text).
- **Queries** return information about an object: current balance, number of characters, margin size.

Both commands and queries can be exercised on a particular object through a "feature call" written, in most object-oriented languages, through dot notation, as in

*my_account***.***withdraw* (500)
b := *my_account***.***balance*

2.2 Argumentless boolean queries

Among queries, *boolean* queries are of particular interest, especially boolean queries without arguments. Examination of object-oriented libraries such as EiffelBase [M1] and others indicates that argumentless queries play a prominent role in classes. Examples include:

- In a bank account class, *is_overdraft*.
- In a paragraph class, *is_justified*.
- In data structure classes, *is_empty* and (if the representation has limited capacity) *is_full*.
- In a list class where lists have cursors indicating a current position of interest, *is_before*, *is_after*, *off*, *is_first*.

The recommended Eiffel convention, whose very existence reflects the ubiquity and importance of such queries, is to give them names starting with *is_*.

Such argumentless queries are generally part of the official interface of the corresponding classes. They intuitively seem, for a well-designed class, to reflect fundamental, *qualitative* properties of the state. For example a list may, or not, be empty; and it may, or not, have twenty-five elements. While the corresponding classes will typically have a query *is_empty* they will not, in general, offer *has_twenty_five_elements*. This is because the designer of the class intuitively thought of the second property (if he considered it at all) to reflect a circumstantial possibility for the state of a list, but understood the distinction between empty and non-empty lists as a critical division of the set of possible list states.

Observation of well-written O-O software reinforces this intuition about the importance of argumentless queries, both externally (as part of the interface of classes) and internally (as part of their implementation):

- Externally, boolean queries often serve as preconditions and appear in invariants. For example, the precondition of a routine to remove an item from a list is **not** *is_empty*; and the invariant will include properties such as *is_before* **implies** *off*.
- Internally, the implementation of a routine to add an item to a list will proceed differently depending on whether the list is initially empty or not and (in an implementation based on an array but dynamically resizable) whether the current implementation is full or not.

All this suggests that the distinction may also be useful when it comes to dividing the state space for purposes of testing the software.

In particular, it follows from the last comment — about features being internally relevant to the implementation — that argumentless boolean queries may be our best bet when we are doing black-box testing and trying to guess the kind of properties actually used in decision branches of the implementation. A query such as *is_empty* is, in the end, nothing else than a predicate — a boolean expression — as used by the control structure of programs to select between branches of conditional expressions and to decide whether to terminate loops. Since testing strategies must partition the state space into representative categories, they use such predicates for the partitioning; for example white-box testing relies on predicates used in tests, such as *c* in **if** *c* **then** *a* **else** *b* **end**, to generate a test with *c* true to exercise *a* and one with *c* false to exercise *b*. If our intuition is correct that boolean queries reflect qualitatively important properties of the object state, then it may be useful to use them, rather than arbitrary predicates, to partition the state space. This possibility is particularly

attractive in black-box testing, where we don't have access to the internal structure of the code, and cannot, as a result, directly know which boolean expressions, such as *c* above, actually appear in tests governing the control structure. In light of the above observations, argumentless queries are our best bet.

2.3 The conjecture

The preceding observations lead to the conjecture behind the present work:

> *Boolean Query Conjecture:* The argumentless boolean queries of a well-written class yield a partition of the corresponding object state space that helps the effectiveness of testing strategies.

"Well-written" is a subjective term, but we will assume the following:

- The class indeed includes boolean queries reflecting important abstract properties of the corresponding objects.
- Routines are equipped with contracts, in particular preconditions. Our main experimentation target is the EiffelBase libraries [M1], which indeed is equipped with contracts.
- The contracting style is on the "demanding" side [M2]: routines try to limit their functionality to the required minimum by enforcing reasonableness conditions on their clients.

2.4 Assessing the conjecture

The Boolean Query Conjecture is of a heuristic nature and, as such, not amenable to a formal proof. To assess its validity, we simply:

- Extend an existing tool for automatic test generation, AutoTest, to take advantage of partitioning based on argumentless boolean queries.
- Compare the effectiveness of the resulting testing strategy — how many bugs it finds, and the quality of its routine coverage — with the effectiveness of the original AutoTest using a random strategy for black-box testing.

3. Using contracts and proof techniques

3.1 Basic definitions

In the rest of this discussion the term **query** will be used as a shorthand for "exported argumentless boolean query", since these are the only kinds of queries of interest for the discussion. The following definitions will be useful.

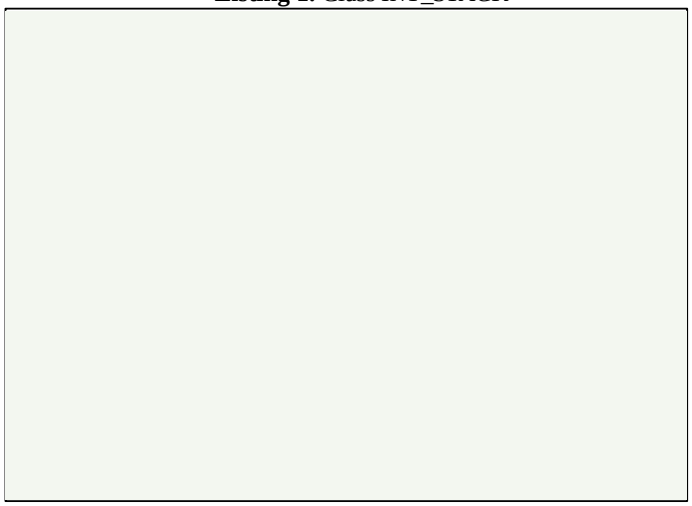
Boolean abstraction function: A *boolean abstraction function* is a vector $<q_1, q_2, \ldots, q_n>$ of queries.

Abstract object state: An *abstract object state* is the vector $<v_1, v_2, \ldots, v_n>$ containing the result of evaluating the queries of a boolean abstraction function $<q_1, q_2, \ldots, q_n>$ in a concrete state s of a particular object, with $v_i = q_i(s)$ for all $i \in 1..n$.

If a class has *n* queries, the number of abstract object states for an instance of the class is 2^n. Note that usually only a subset of these possible abstract states makes sense, since a useful state should satisfy the class invariant.

As a simple example of these concepts, consider the following Eiffel class, adapted from actual (generic) stack classes in EiffelBase:

Listing 1. Class *INT_STACK*



The features "*not_empty*" and "*not_full*" are queries. The vector <*not_empty*, *not_full*> makes up the boolean abstraction function for the class. The set of abstract object states is {<0, 0>, <0, 1>, <1, 0>, <1, 1>}) (using 0 for False and 1 for True).

Such an abstract state space will usually be too large to be practically tractable. With a language supporting the inclusion of class invariants, and classes that take advantage of this mechanism, we can reduce that size significantly by excluding states that do not satisfy the invariant. For example a stack cannot (with *capacity*> 0) as also ensured by the invariant) be both empty and full, so we can remove <0, 0> from the above state space. The following definition generalizes this observation:

Reachable abstract object state. An abstract object state is *reachable* if it satisfies the class invariant.

3.2 Query-based testing

The general strategy for query-based testing, represented by figure 1, will involve the following elements, detailed in subsequent sections:

- Find the exported argumentless boolean queries.

- (Section 3.3 below.) Use a boolean constraint solver (SICStus) to generate all possible abstract object states that satisfy the clauses of the class invariant involving only these queries — ignoring any invariant clauses involving other features of the class, such as the integer attributes *count* and *capacity* in the above example, since this is beyond the reach of a boolean constraint solver.
- (3.4) Use a theorem prover (Simplify) to prune abstract object states that do not satisfy the invariant (including the previously ignored clauses, such as those involving *count* and *capacity* in the example.
- (3.5) Use a forward testing approach (part of the AutoTest tool), attempt to cover all the resulting abstract object states. In this process, any routine execution that violates a contract element uncovers a bug and hence marks a success of the strategy.
- (3.6) All the previous steps are automatic. After they have been run, it is useful to perform a manual inspection to determine how many of the abstract object states have been covered. For each state that has *not* been covered, you should inspect the specification to determine whether each the state makes sense or not. If not, this may lead, if you have access to the original class or may make suggestions to its developers, to strengthening its invariant. On the other hand if you find out that the state is logically meaningful, you may have to adapt the testing strategy, adding manual tests if necessary, to extend coverage.

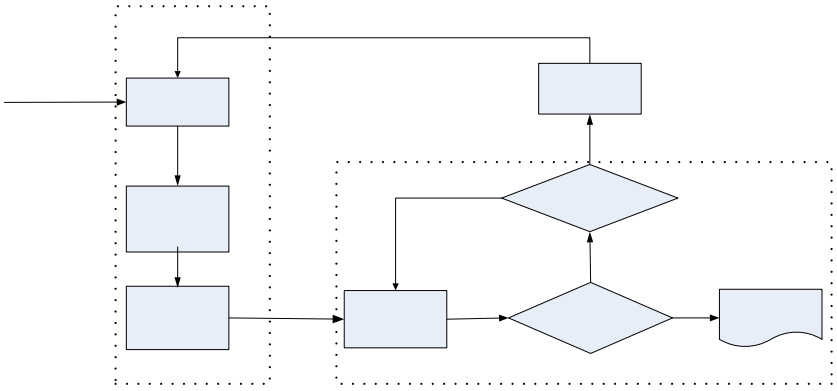


Fig. 1. Overview of class testing procedure

3.3 Generating abstract state through boolean constraint solving

Acquiring all reachable abstract object states requires the support of a boolean constraint solver and theorem prover. As noted above, the first step is to collect all the exported, argumentless boolean queries from the class interface; this can be done in several ways (parsing of the class of just its official interface documentation, reflection, or data from the IDE). The next step is to strip down the class invariant to those clauses that only involve these queries, temporarily dropping any other clauses, for example those involving *count* or *capacity*.

This allows feeding the resulting simplified invariant into a boolean constraint solver. We have chosen the SCIStus solver [S1] for that purpose. The result is to

obtain all possible abstract object states; in the simple example above we would get { <0, 1>, <1, 0>, <1, 1>}, with only two queries, *not_empty* and *not_full*. Note that <0, 0> is not a member of that set since the constraint solver takes advantage of the invariant clause *not_empty* **or** *not_full* to remove it as inconsistent.

For an actual class, *FIXED_LIST*, the number of applicable queries is 9, resulting in an abstract state space with 512 elements. Contraint solving reduces this number considerably, to 122.

3.4 Pruning the state space through theorem proving

The resulting abstract state space may still includes states that do not make sense. This is not the case in the simple *INT_STACK* example, since the three states that survive the previous step are all reachable, but often happens in larger cases; for example, in the *FIXED_LIST* class of EiffelBase, boolean constraint solving does not eliminate a state in which "**not** *before* **and not** *after* **and** *off*" holds.

To prune the state space from such spurious cases violating the invariant, the strategy next applies theorem proving. The theorem prover reintroduces the invariant clauses ignored by the previous step to reduce the state of the state space. The proof tool we use is **Simplify** [D1]. Adding this step is quite effective: for example, in the *FIXED_LIST* case, it reduces the state space from 122 elements to 22.

3.5 Forward testing

The previous steps give us a set of abstract states that can be used as a criterion for test coverage according to the following definition:

> **Boolean query coverage**. A set of tests for a class satisfies *boolean query coverage* if and only if the execution of these tests can cover all the reachable abstract object states for that class.

This sets the stage for the testing effort: try to achieve boolean query coverage by covering as many as possible of the abstract object states determined through application of the preceding techniques.

For the testing effort we rely on AutoTest [C2], a "push-button" testing tool that uses contracts to perform automatic test generation and bug detection. AutoTest uses a *forward testing* [L4] process.

The forward testing process attempts to explore all abstract object states. The process first creates some objects via different creation procedures to generate the initial abstract object states. Starting from these initial abstract object states, it executes all exported routines in these abstract object states to explore more abstract states. It repeats this step until it either finds no new abstract object states or reaches a predefined threshold (of number of calls, or testing time). Listing 2 describes the procedure more precisely.

Listing 2. Forward Testing

To formalize this process we may rely on the following notion (adapted from [L3]:

> **Object state machine.** Consider a class C; let EC be its set of exported commands and S be the set of corresponding object states. The *object state machine* for C is defined by the subset I $\subseteq$ S of initial object states (as produced by creation procedures) and the transition function t: S×EC $\rightarrow$ S describing the effect of C's commands.

We can talk of abstract or concrete object state machines, based on this definition, by choosing S to be the set of abstract or concrete states.

The class testing procedure records all exercised abstract object states and transitions. This means that developers can examine the result of a test campaign to determine if the class under testing exhibits unexpected behavior, or to assess the completeness of a test suite.

For INT_STACK, the extracted abstract object state machine is as follows.

Queries:
1. not_empty
2. not_full

Set of states S:
{<0, 1>, <1, 1> , <1, 0>}

Initial states I:
<0, 1>

Command set EC:
pop, push

Transition function t:

<0, 1>	pop	<0, 1>
<0, 1>	push	<1, 1>
<1, 1>	pop	<0, 1>
<1, 1>	push	<1, 0>
<1, 0>	pop	<1, 1>
<1, 0>	push	<1, 0>

Applying AutoTest's forward testing to class *INT_STACK* will cover all reachable abstract object states. This may seem to be because of the simple nature of this academic example, but in fact a very encouraging result of our experiments is that AutoTest's automated strategy yields a very high initial coverage, 80% or higher, of the abstract object state space for all the actual (production) library classes we have tried. As described in the next section, we then perform a manual inspection of the results to examine uncovered states, and improve the invariants as a result of this inspection; in all of our experiments so far this has enabled us in the end to reach 100% state space coverage.

3.6 Inspecting the specification

At the end of the process it is useful to inspect the results, in particular to examine abstract state coverage. If an abstract state has not been covered, possible actions are:

- Add manual tests that will exercise the corresponding states. (AutoTest has the possibility of including manual tests along the automatically generated ones.)
- If it appears that the states are not possible, reinforce the class invariants to exclude them.

As noted earlier, our experiments so far have yielded excellent coverage of the abstract state. But as an example of the second case, we found that in class *FIXED_LIST* 10 states, out of the 22 remaining from previous reductions of the abstract state space, seemed unreachable because a particular property relative to the query *prunable* seems to be missing. Adding the corresponding invariant clause achieves total coverage.

4. Experimental setup and study results

4.1 Choice of library

To examine the Boolean Query Conjecture with the above strategy, we performed a number of tests of classes from the EiffelBase library. EiffelBase is particularly interesting in several respects:

- It is not an academic example but a production library, used — in its successive incarnations since its first version almost twenty years ago — in numerous applications, in particular, currently, in large, mission-critical systems handling (for example) billions of dollars of investments or large-scale missile simulations.
- In spite of this background it still (regrettably) has bugs.
- These bugs (fortunately) arise only in remote, uncommon cases, and are only found through systematic testing by AutoTest, which has taken EiffelBase as one of its primary experimental targets. Obviously, all EiffelBase bugs found so far by AutoTest, including the ones uncovered by present study, have now been corrected.
- EiffelBase is a showcase of object-oriented techniques and in particular makes extensive use of contracts.

4.3 Choice of target classes

For the present study, we used *INT_STACK*, our toy example (for reference purposes), and four important classes of the EiffelBase library: *LINKED_LIST*, *BINARY_TREE* , *ARRAYED_SET* and.

The size of these classes, in terms of number of routines (and ignoring attributes) is as follows:

- *LINKED_LIST*: 93 routines.
- *BINARY_TREE*: 99 routines.
- *ARRAYED_SET*: 69 routines.
- *FIXED_LIST*: 87 routines.

Of these, 27 come from the top-level class *ANY*, which is the only ancestor of the classes given. (All Eiffel classes inherit from *ANY*). AutoTest tests all routines, whether defined in the class itself or inherited. Indeed, as the classes given are pretty deep in the inheritance hierarchy, many of their routines are inherited.

4.2 The testing environment

The AutoTest tool, the centerpiece of our testing work and responsible for the forward testing step (3.5), is a push-button testing environment which takes care of both test case generation and test oracles. Test cases are generated by systematically calling all the routines of the selected classes and any classes on which they rely; test oracles (the mechanisms to determine whether a test is successful) are entirely provided by routine postconditions and invariants. More precisely:

- A precondition violation for a routine directly called by AutoTest indicates that the test is not interesting; AutoTest minimizes such occurrences through constraint solving and proof techniques as used in this article.
- If a routine gets executed (its precondition was satisfied), any violation of the class invariant, the routine's postcondition, or the precondition of another routine that it calls indicates a bug to be added to the output of the AutoTest run.

In the last case, AutoTest performs a *minimization* step that finds, if possible, a shorter sequence leading to the same incorrect result; this enables using the shorter sequence, and hence maximizing efficiency, for debugging, and for later regression testing.

AutoTest has a sophisticated testing architecture making it possible to perform a large number of such automatic routine executions, recovering if any of them fails, and presenting the test results in convenient HTML format. When detecting a bug — a sequence of execution that leads to a violation of a postcondition or other contract element

Although primarily an automatic testing tool, AutoTest is also a general testing environment supporting the addition of manually selected test cases, and automating the testing process, in particular regression testing. AutoTest is being more closely integrated with the EiffelStudio environment so that in the future, for example, users will have the choice, when an execution fails, of having the faulty call sequence automatically integrated, after minimization, in the regression test suite.

4.4 Study results

We applied AutoTest to the result of performing the constraint solving and theorem proving steps described above on the selected classes. We also applied plain AutoTest, not taking advantage of these steps, to the same class, and compared the results for *number of bugs found* and *routine coverage* (the number of routines exercised). The following table shows the results.

Table 1. Comparison of Forward Testing with Random Testing

Tested Class	Random Testing		Boolean Query Coverage	
	Routine Coverage	Bugs Found	Routine Coverage	Bugs Found
INT_STACK	33% (1/3)	1	100% (3/3)	2
LINKED_LIST	85% (79/93)	1	99% (92/93)	7
BINARY_TREE	88% (87/99)	5	100% (99/99)	11
ARRAYED_SET	84% (58/69)	1	100% (69/69)	6
FIXED_LIST	93% (81/87)	12	99% (86/87)	12

4.5 Evaluation

The number of classes to which we have applied the strategy is still too small to warrant statistically significant conclusions, but the number of bugs found thanks to the strategy is remarkable on the current examples: doubling in two cases, and, in another, going from 1 to 7.

We do not yet have figures taking timing costs into account. Clearly, performing the extra steps of constraint solving and proof adds time to the overall testing process. While this time has been small in the experiments we have performed so far, we need more precise measurements of the overhead before making any final assessment of the approach.

At this point, however, this limitation is not critical for us. “Time to first bug”, while relevant to other test work, is not our primary concern. We are studying production-grade software; any bug identified is a major result and, as long as the time to find it is reasonable, it does not really matter whether it is 30 seconds or 10 minutes — especially since the procedure to find it is (apart from the optional manual steps that can be added as described in 3.6) entirely automatic, so that we can make machines, rather than human testers, do the work for us.

In this respect the techniques described here have already proved their worth by enabling us to detect and correct heretofore unsuspected bugs, and hence improve the reliability of real software systems.

5 Related Work

The following work is relevant to the discussion of the testing strategy presented in this paper.

5.1 Construction of Abstract States

Queries and boolean predicates have been used to generalize concrete states [B2, X1, Y1]. Xie et al. gave a black-box abstraction method that uses public observers that return non-void values to generalize concrete object state machine into observable object state machine and infer this abstract machine through unit testing [X1]. This approach cannot bound concrete object states to a finite abstract object states, as a result, cannot achieve abstract state coverage in testing. Ball et al. presented a white-box boolean predicate abstraction approach that uses all predicates appearing in program to generalize concrete program states into a set of abstract program states, and gave the upper bound and lower bound of these abstract states. This approach cannot infer all abstract states of a program that satisfy its specification since it is a white-box method. Therefore, it cannot statically decide the exact bound of satisfiable abstract states. Yorsh et al. make use of the boolean predicate abstraction approach to find a proof for a program rather than detecting real errors.

Our object state abstraction approach is a black-box method and uses contracts and proof techniques to infer all abstract object states that satisfy class contracts. This abstraction process is independent of testing and can be done statically. Moreover, it also provides a way to inspect class contracts. Because our abstraction approach maps concrete object states to finite reachable abstract states, we can direct our class testing procedure to completely explore these states.

5.2 Black-Box Test Coverage Criteria

Category-Partition (CP) [O1] is a common black-box test strategy. Each category defines a major property of the parameter or condition of a function/routine and partitioned into a series of distinct choices. A set of choices from all the categories is combined into a test frame, where each category contributes with, at most, one choice. These test frames are templates used to derive test cases. To apply CP, we need to consider the approach to combining choices. There are three combing approaches: *all combination*, *base choice* and *each choice*, where *all combination* derives all combinations of choices as test frame. Hence *all combination* partitions the whole input domain and is the most expensive and effective combining approach. boolean query coverage is essentially a Category-Partition strategy used for generating object states. This strategy takes every boolean query as a category and defines all possible combinations among the values of these boolean queries. Therefore, it partitions the whole object sate space and is a most effective CP strategy for generating object states.

Because of the easiness of automation, random testing [D2, H2, H3] is practically widely adopted black-box test strategy. The studies in [D2, H2] show that random testing could be more cost-effective than partition testing (assuming that its cost is lower than that of partition testing) with respect to the probability of detecting at least

one failure. Comparing to random testing, boolean query coverage can also be implemented automatically and detects more object state related bugs.

5.3 Test Case Generation and Automatic Testing

To cover all reachable abstract object state space, we mainly use the forward testing and complement this process with random testing and manual testing. All of these testing strategies have been implemented in Eiffel automatic unit testing tool AutoTest.

Automatic class testing is more practical when class specification are embedded into the program as formal or semi-formal contracts. TestEra[M4] is a contract-based software test tool targeting Java source code and specification written in Alloy [J1] (a structural modeling language based on first-order logic). Due to the impedance mismatch between the specification and the implementation language the testing process is not fully automatic and there is a higher barrier for the developer to provide the specification since he has to learn a new language. This automatic testing tool does not adopt object state abstraction approach, while uses model checking technique to generate the test inputs that satisfy a function's precondition.

The Korat tool [B3] uses a function's precondition on its input to automatically generate all (nonisomorphic) test cases up to a given small size. Korat constructs test cases by setting the field values directly not by invoking routines as done in our forward testing strategy.

Another tool, Check'n'Crash [C1], does not use specifications but uses an external static verifier (ESC/Java2) to calculate a precondition to describe the conditions that might result in a failure. It then uses a constraint solver to generate instances that satisfy this precondition. Since their approach assumes no specifications, they use a heuristic to filter expected failures from unexpected ones.

AutoTest [C2] implements fully automatic class testing based on contracts. Without intervention from a user, AutoTest generates tests, executes tests and verifies test results. This testing tool is configurable. Testers can configure the testing strategies (random, forward and manual), then AutoTest can execute these selected testing strategies automatically.

Our testing procedure includes two fully automatic testing processes. The first is using forward testing to explore most abstract object states. If there are some abstract object states that cannot be covered then tester complement some test cases encoded in manual test case form and execute AutoTest to cover all abstract object states and construct abstract object state machine. The second is an automatic test oracle that uses contracts embedded in the class under test.

6. Future Work

The results presented here are particularly promising but require further work, in particular:

- Application to many more example classes. Potentially we should process all EiffelBase classes.

- Application to software that is more representative of user programs: EiffelBase is a general-purpose library, but we must also apply the approach to typical commercial software in various application areas.
- Closer evaluation of the results, in particular with respect to the time needed to find bugs (for the whole strategy, including testing but also the preparatory stages of constraint solving and proof), not just the number of bugs eventually found.
- Integration of the techniques, to the extent that will appear justified, in the AutoTest framework, so that it can take advantage of the best combination of various software reliability techniques, from constraint solving and model checking to proofs as well as tests.

Acknowledgements

We thank Joseph N. Ruskiewicz for his help with Simplify and constructive comments. We also thank Stephanie Balzer, Andreas Leitner, Ilinca Ciupa and Manuel Oriol for their feedback and many invaluable technical discussions. We also thank Eric Bezault for providing Gobo Eiffel which served us as a great platform to build our tools on.

References

[B1] M. Barnettl, K. Rustan, M. Leinol, W. Schultel, The Spec# programming system: An overview, in: M. H. Gilles Barthe, J.-L. Lanet, T. Muntean (Eds.), Construction and Analysis of Saft, Secure, and Interoperable Smart Devices: International Workshop, CASSIS 2004, Springer Berlin / Heidelberg, Marseille, France, 2004.

[B2] T. Ball, A theory of predicate-complete test coverage and generation, in: 3rd International Symposium on Formal Methods for Components and Objects, 2004, pp. 1-22.

[B3] C. Boyapati, S. Khurshid, D. Marinov, Korat: Automated testing based on Java predicates, in: Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA'02), ACM Press, 2002, pp. 123-133.

[C1] C. Csallner, Y. Smaragdakis, Check 'n' crash: combining static checking and testing, in: ICSE'05: Proceedings of the 27th international conference on Software Engineering, ACM Press, New York, NY, USA, 2005, pp. 422-431.

[C2] I. Ciupa, A. Leitner, Automatic testing based on design by contract, in: Proceedings of Net.ObjectDays 2005 (6th Annual International Conference on Object-Oriented and Internet-based Technologies, Concepts and Applications for a Networked World), 2005, pp. 545-557.

[D1] D. Detlefs, G. Nelson, and J. B. Saxe, Simplify: A theorem prover for program checking. Technical Report HPL-2003-148, HP Labs, 2003. http://research.compaq.com/SRC/esc/Simplify.html.

[D2] J. Duran and S. Ntafos, An evaluation of random testing, IEEE Transactions on Software Engineering, , July 1984, SE-10:438 – 444.

[H1] A. Hamie, Towards verifying Java realization of OCL-constrained design models using JML, in: Proceedings of 6th IASTED International Conference on Software Engineering and Applications, ACTA Press, MIT, Cambridge, MA, USA, 2002.

[H2] D. Hamlet and R. Taylor, Partition testing does not inspire confidence, IEEE Transac tions on Software Engineering, December 1990, 16 (12):1402–1411.

[H3] R. Hamlet, Random testing, in: J. Marciniak, editor, Encyclopedia of Software Engineering, Wiley, 1994, pp. 970-978.

[J1] D. Jackson, Alloy: Alightweight object modeling notation, ACM Trans. Soft. Eng. Methodology, 11(2) (2002) pp. 256-290.
[K1] R. Kramer, iContract - the Java ™ design by contract ™ tool, in: Proceedings of Object-Oriented Language and Systems, IEEE Computer Society, Washington, DC, USA, 1998, pp. 295-307.
[L1] G. T. Leavens, A. L. Baker, Enhancing the pre- and postcondition technique for more expressive specifications, in: World Congress on Formal Methods, 1999, pp.1087-1106.
[L2] G. T. Leavens, Y. Cheon, C. Clifton, C. Ruby, D. R. Cok, How the design of jml accommodates both runtime assertion checking and formal verification, in: FMCO 2002, pp. 262-284.
[L3] D. Lee and M. Yannakakis, Principles and methods of testing finite state machines - A survey, in: Proc. The IEEE, 1996, pp. 1090-1123.
[L4] L. Liu, A. Leitner and J. Offutt, Using contracts to automate forward class testing, submitted to Journal of System and Software.
[M1] B. Meyer, Reusable Software: The Base Object-Oriented Libraries, Prentice Hall, 1994.
[M2] B. Meyer, Object-Oriented Software Construction, 2nd Edition, Prentice Hall, 1997.
[M3] B. Meyer, Eiffel: The Language, Prentice Hall, 1991, revised edition in progress at http://se.ethz.ch/~meyer/ongoing/etl/, 2006..
[M4] D. Marinov, S. Khurshid, TestEra: A novel framework for automated testing of Java programs, in: Proc. 16th IEEE International Conference on Automated Software Engineering (ASE), 2001, pp. 22-34.
[N1] J. W. Nimmer and M. D. Ernst, Invariant inference for static checking: An empirical evaluation, in: FSE 2002, pp. 11-20.
[O1] T. J. Ostrand and M. J. Balcer, The Category-Partition method for specifying and generating functional test, Comm. ACM, vol. 31, no. 6, pp. 676-686, 1988.
[R1] M. Richtersl, M. Gogolla, On formalizing the UML object constraint language OCL, in: M. L. L. Tok Wang Ling, Sudha Ram (Eds.), 17 International Conference on Conceptual Modeling (ER), Springer Berlin / Heidelberg, Singapore, 1998.
[S1] SICStus Prolog User's Manual, http://www.sics.se/sicstus/docs/latest/pdf/sicstus.pdf.
[W1] J. Whaley, M. C. Martin and M. S. Lam, Automatic extraction of object-oriented component interface, ISSTA 2002, pp. 218-228.
[X1] T. Xie and D. Notkin, Automatic extraction of object-oriented observer abstractions from unit-test executions, in: ICFEM 2004, pp. 290-305.
[Y1] G. Yorsh, T. Ball and M. Sagiv, Testing, abstraction, theorem proving: better together! ISSTA 2006, pp. 145-156.