PROGRAMS THAT TEST THEMSELVES

Bertrand Meyer, *ETH Zurich and Eiffel Software*
Arno Fiva, Ilinca Ciupa, Andreas Leitner, and Yi Wei, *ETH Zurich*
Emmanuel Stapf, *Eiffel Software*

The AutoTest framework automates the software testing process by relying on programs that contain the instruments of their own verification, in the form of contract-oriented specifications of classes and their individual routines.

Modern engineering products—from planes, cars, and industrial plants down to refrigerators and coffee machines—routinely test themselves while they operate. The goal is to detect possible deficiencies and to avoid incidents by warning the users of needed maintenance actions. This self-testing capability is an integral part of the design of such artifacts.

The lesson that their builders have learned is to *design for testability*. This concept was not always understood: With cars, for example, we used to have no clue (save for the oil gauge) that major mechanical trouble might be imminent; if we wanted to know more, we would take our car to a mechanic who would check every component from scratch, not knowing what actually happened during operation. Today's cars, in contrast, are filled with sensors and gauges that perform continuous testing and gather data for maintenance.

While software does not physically degrade during operation, its development requires extensive testing (and other forms of verification); yet software design usually pays little attention to testing needs. It is as if we had not learned the lessons of other industries: Software construction and software verification are essentially separate activities, each conducted without much consideration of the other's needs. A consequence is that testing, in spite of improved tools, remains a labor-intensive activity.

AUTOTEST

AutoTest is a collection of tools that automate the testing process by relying on programs that contain the instruments of their own verification, in the form of *contracts*—specifications of classes and their individual routines (methods). The three main components of AutoTest address complementary aspects:

- *Test Generation*: automatically creates and runs test cases, without any human input such as manually prepared test cases and test oracles.
- *Test Extraction*: automatically produces test cases from execution failures. The observation behind Test

Extraction is that some of the most important test cases are not devised as such: They occur when a developer tries the program informally during development, but then it's execution fails. The failure is interesting, in particular for future regression testing, but usually it is not remembered: The developer fixes the problem and moves on. From such failures, Test Extraction automatically creates test cases, which can be replayed in subsequent test campaigns.

- *Integration of Manual Tests*: supports the development and management of manually produced tests. Unlike Test Generation and Test Extraction, this functionality relies on state-of-the-art techniques and includes no major innovation, but it ensures a smooth interaction of the automatic mechanisms with existing practices by ensuring all tests are managed in the same way regardless of their origin—generated, extracted, or manual.

These mechanisms, initially developed for research purposes at ETH Zurich, have now been integrated into the EiffelStudio environment and are available both as an open source download (http://eiffelstudio.origo.ethz.ch) and commercially. Research continues on the underlying theory and methods (http://se.ethz.ch/research/autotest).

Our working definition of testing focuses on one essential aspect: To test a program is to try to make it fail.[1] Other definitions include more lofty goals, such as "provid[ing] information about the quality of the product or service" (http://en.wikipedia.org/wiki/Software_testing). But in practice, the crucial task is to uncover *failures* of execution, which in IEEE-standard terminology[2] reflect *faults* in the program, themselves the result of *mistakes* in the developer's thinking. AutoTest helps provoke failures and manage information about the corresponding faults.

'AUTOMATED TESTING'

"Automated testing" is a widely used phrase. To understand what it entails, it is necessary to distinguish several increasingly ambitious levels of automation.

What is best automated today is *test execution*. In a project that has generated thousands of test cases, running them manually would be tedious, especially as testing campaigns occur repeatedly—for example, it is customary to run extensive tests before every release. Traditionally, testers wrote scripts to run the tests. The novelty is the spread of frameworks such as JUnit (www.junit.org) that avoid project-specific scripts. This widely influential development has markedly improved testing practice, but it only automates a specific task.

A related goal, also addressed by some of today's tools, is *regression testing*. It is a common phenomenon of software development that some corrected faults reappear in later versions, indicating that the software has partly "regressed." A project should retain any test that failed at any stage of its history, then passed after the fault was corrected; test campaigns should run all such tests to spot cases of regression.

Automated tools should provide *resilience*. A large test suite is likely to contain some test cases that, in a particular execution, crash the program. Resilience means that the process may continue anyway with the remaining cases.

One of the most tedious aspects of testing is *test case generation*. With modern computers we can run very large numbers of test cases. Usually, developers or testers have to devise them; this approach, limited by people's time, does not scale up. The AutoTest tools complement such manual test cases with automatic tests exercising the software with values generated by algorithms. Object-oriented programming increases the difficulty because it requires not only elementary values such as integers but also objects.

AutoTest helps provoke failures and manage information about the corresponding faults.

Test oracles represent another challenge. A test run is only useful if we know whether it passed or failed; an oracle is a mechanism to determine this. Here too a manual process does not scale up. Approaches such as JUnit include oracles in test cases through such instructions as "assert (success_criterion)," where "assert" is a general mechanism that reports failure if the success_criterion does not hold. This automates the application of oracles, but not their preparation: The tester must still devise an assert for every test. AutoTest's approach removes this requirement by relying on contracts already present in the code.

Another candidate for automation is *minimization*. It is desirable to retain and replay any test that ever failed. The failure may, however, have happened after a long execution exercising many instructions that are irrelevant to the failures. Retaining them would make regression testing too slow. Minimization means replacing a test case, whenever possible, with a simplified one producing a failure that evidences the same fault.

Commonly used frameworks mostly address the first three goals: test execution, regression testing, and resilience. They do not address the most labor-intensive tasks: preparing test cases, possibly in a minimized form, and interpreting test results. Without progress on these issues, testing confronts a paradox: While the growth of computing power should enable us to perform ever more

DESIGN BY CONTRACT

Design by Contract[1] is a mechanism pioneered by Eiffel that characterizes every software element by answering three questions:

- **What does it expect?**
- **What does it guarantee?**
- **What does it maintain?**

Answers take the form of preconditions, postconditions, and invariants. For example, starting a car has the precondition that the ignition is turned on and the postcondition that the engine is running. The invariant, applying to all operations of the class CAR, includes such properties as "dashboard controls are illuminated if and only if ignition is on."

With Design by Contract, such properties are not expressed in separate requirements or design documents but become part of the software; languages such as Eiffel and Spec#, and language extensions such as JML, include syntax—keywords such as require, ensure, and invariant—to state contracts.

Applications cover many software tasks: analysis, to make sure requirements are precise yet abstract; design and implementation, to obtain software with fewer faults since it is built to a precise specification; automatic documentation, through tools extracting the contracts; support for managers, enabling them to understand program essentials free from implementation details; better control over language mechanisms such as inheritance and exceptions; and, with runtime contract monitoring, improvements in testing and debugging, which AutoTest takes further.

Reference

1. **B. Meyer, "Applying 'Design by Contract,'" *Computer*, Oct. 1992, pp. 40-51.**

exhaustive tests, these manual activities dominate the process; they limit its practical effectiveness and prevent scaling it up.

The AutoTest framework includes traditional automation but particularly innovates on test case generation, oracles, and minimization. It has already uncovered many faults in released software and routinely finds new ones when given classes to analyze.

CONTRACTS AS ORACLES

AutoTest exercises software as it is, without instrumentation. In particular, its approach does not require writing oracles.

What makes this possible is that the software under test consists of classes with contracts: Routines may include *preconditions* and *postconditions*; classes may include *invariants*. In contract-supporting languages such as Eiffel, contracts are Boolean expressions of the underlying programming language, and hence can be evaluated during execution; this provides the basis of the contract-based approach to testing. The "Design by Contract" sidebar describes the use of contracts in more detail.

In the traditional Eiffel process, developers write programs annotated with contracts, then manually run these programs, relying on the contracts to check the executions' correctness. AutoTest's Test Generation component adds many more such executions by generating test cases automatically.

Execution will, on entry to a routine *r*, evaluate *r*'s precondition and the class invariant; on exit, it evaluates *r*'s postcondition and the invariant. For correct software, such evaluations always yield true, with no other consequence; but an evaluation to false, known as a contract violation, signals a flaw:[3]

- A precondition violation signals a possible fault in the client (the routine that called *r*).
- A postcondition or invariant violation signals a possible fault in the supplier (*r* itself).

If the call is a result of automatic test generation, the interpretation of the first case is more subtle:

- If the tool directly issued the call to *r*, this is a problem with the tool's generation strategy, not the software under test; the test case should be ignored. Testing strategies should minimize such spurious occurrences.
- If another routine performed the call, the caller did not observe *r*'s specification, signaling a fault in that routine.

The benefit of using contracts as oracles is that the software is tested as it is. Other tools using contracts often require software that has been specially prepared for testing. With Eiffel or Spec# (http://research.microsoft.com/SpecSharp)—and JML, the Java Modeling Language, if used to write code rather than to instrument existing Java code—contracts are there from the start.

In practice, no special skill is required of programmers using Design by Contract. Although the approach can be extended to full formal specifications, most contracts in common usage state simple properties: A variable is positive, two references point to the same object, a field is not void. In addition, contracts are not just a theoretical possibility; programmers use them. Analysis of a large body of Eiffel code, proprietary and open source, indicates widespread contract use, accounting for 1.5 to 7 percent of lines.[4]

In such a context, writing simple contracts becomes as natural as any other programming task.

Not all failures result from explicit contract violations; another typical case is arithmetic overflow. AutoTest records all failures in the same way. Unlike many static analysis tools, AutoTest produces no false alarms: Every violation it reports reflects a fault in either the implementation or the contract.

TEST GENERATION

There has been considerable research on test generation from specifications. The "Using Specifications for Test Case Generation: A Short Survey" sidebar highlights some key aspects of this research.

The Test Generation part of AutoTest is a push-button testing framework. The only information it requires is a set of classes to be tested. The tool takes care of the rest by automating three of the key tasks cited earlier:

- To generate tests, it creates instances of the classes and calls their routines with various arguments.
- To determine success or failure, AutoTest uses the classes' contracts as oracles.
- The tool produces minimized versions of failed tests for regression testing.

An important property for users is that the environment will treat all tests in the same way, regardless of their origin (generated, manual, or extracted); this applies in particular to regression testing.

Figure 1 shows the principal steps for testing a set of classes:

- Generate instances of the classes under test.
- Select some of these objects for testing.
- Select arguments for the features to be called.
- Run the tests.
- Assess the outcome: pass or fail, applying the contracts as oracles.
- Log results and failure-reproducing test cases.
- Construct a minimized form of every logged test and add it to the regression suite.

The test-generation strategies involve numerous choices controlled by parameters to AutoTest. Extensive experimentation has produced default values for all these parameters.

Obtaining objects and other values

The unit of testing is a routine call of the form *target.routine (arguments)*. It requires at least one object, the target; the arguments may include other objects and primitive values.

To obtain test inputs, AutoTest maintains an object pool. Whenever it needs an object of a type *T*, it decides whether to create a new instance of *T* or draw from the pool. Creation is necessary if the pool does not contain an instance of *T*; but even if it does, AutoTest will, with a preset frequency (one of the tool's parameters), create an object and add it to the pool. An effective strategy needs both possibilities:

USING SPECIFICATIONS FOR TEST CASE GENERATION: A SHORT SURVEY

The goal of automating testing based on specification is an active research topic.

- **Robert V. Binder (*Testing Object-Oriented Systems: Models, Patterns and Tools*, Addison-Wesley, 1999) emphasizes contracts as oracles.**
- **Dennis Peters and David Parnas ("Using Test Oracles Generated from Program Documentation," *IEEE Trans. Software Eng.*, Mar. 1998, pp. 161-173) use oracles derived from specifications, separate from the program.**
- **The jmlunit script pioneered some of the ideas described in this article, in particular, postconditions as oracles and the observation that a test that directly violates a precondition does not signal a fault. In jmlunit as described by Yoonsik Cheon and Gary T. Leavens ("A Simple and Practical Approach to Unit Testing: The JML and JUnit Way," *ECOOP 2002—Object-Oriented Programming*, LNCS 2374, Springer, 2002, pp. 1789-1901), test cases remain the user's responsibility.**
- **Korat (C. Boyapati, S. Khurshid, and D. Marinov, "Korat: Automated Testing Based on Java Predicates," *Proc. 2002 ACM SIGSOFT Int'l Symp. Software Testing and Analysis*, ACM Press, 2002, pp. 123-133) is an automated testing framework that uses some of the same concepts as AutoTest; to generate objects it does not use creation procedures but fills object fields and discards the result if it violates the invariant. Using creation procedures seems preferable.**
- **DSD-Crasher (C. Csallner and Y. Smaragdakis, "DSD-Crasher: A Hybrid Analysis Tool for Bug Finding," *ACM Trans. Software Eng. and Methodology*, Apr. 2008, vol. 17, no. 2, art. 8) infers contracts from executions, then statically explores paths under the resulting restricted input domain, and generates test cases to verify the results.**
- **Debra Richardson, Owen O'Malley, and C. Tittle ("Approaches to Specification-Based Testing," *ACM SIGSOFT Software Eng. Notes*, Dec. 1989, pp. 86-96) emphasize extending existing implementation-based testing to use specifications.**
- **Alexandre K. Petrenko ("Specification Based Testing: Towards Practice," *Perspectives of System Informatics, LNCS 2244*, Springer, 2001, pp. 287-300) surveys existing approaches.**
- **A. Jefferson Offutt, Yiwei Xiong, and Shaoying Liu ("Criteria for Generating Specification-Based Tests," *Proc. 5th Int'l Congress Eng. of Complex Computer Systems*, IEEE CS Press, 1999, pp. 119-129) discuss generating test inputs from state-based specifications.**

- New objects *diversify* the pool.
- Creating a new object every time would restrict tests to youthful object structures. For example, a newly created list object represents a list with zero elements or one element; realistic testing needs lists with many elements, obtained by creating a list then repeatedly calling insertion procedures.

When the decision is to create an object, this object should satisfy the class invariant. AutoTest relies on the

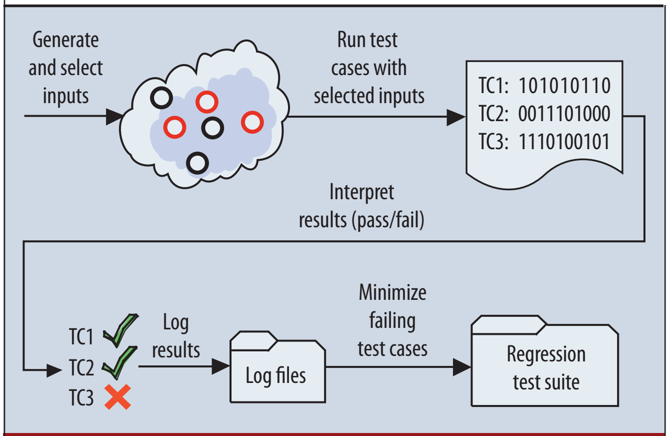

Generate and select inputs
Run test cases with selected inputs
TC1: 101010110
TC2: 0011101000
TC3: 1110100101
Interpret results (pass/fail)
TC1
TC2
TC3
Log results
Log files
Minimize failing test cases
Regression test suite

Figure 1. Test Generation's automated testing process.

normal mechanism for creating instances, satisfying the invariant: creation procedures (constructors). The steps are as follows:

- Choose a creation procedure (constructor).
- Choose arguments, if needed, with the strategies defined below for routine calls. Some of these arguments may be objects, requiring recursive application of the strategy (selection from pool or creation).
- Create the object and call the procedure.

Any object this algorithm creates at any stage is added to the pool, contributing to diversification. Any failure of these operations is logged, even if the operation is not explicitly part of the requested test. The purpose of testing is to cause failures; it does not matter how: The end justifies the means.

Besides objects, a call may need primitive values of types such as INTEGER or CHARACTER. The current strategy uses

- distinguished values preset for each type such as, for integers: 0, minimum and maximum integers, ±1, and so on; and
- other values from the range, selected at random.

This approach may appear simplistic. We are indeed investigating more advanced policies. We have learned, however, that in devising testing strategies sophisticated ideas do not necessarily outperform simpler approaches.[1] The main measure of effectiveness for a testing strategy—at least if we do not rank faults by risk level, but treat all faults as equally important—is the fault count function fc (*t*), the number of faults found in *t* seconds of testing. A "smart" strategy's ability to find more faults or find them faster can be outweighed by a longer setup time. It is essential to submit any idea, however attractive, to objective evaluation.

Adaptive random testing and object distance

To improve on purely random strategies, *adaptive random testing* (ART)[5] attempts to space out values evenly across their domains. This applies in particular to integers. In object-oriented programming, many interesting inputs are objects, with no immediate notion of "evenly spaced out." We introduced *object distance*[6] to extend ART by ensuring that a set of objects is representative. The distance between objects *o*1 and *o*2 is a normalized weighted sum of three properties:

- distance between the types, based on their distance in the inheritance graph and the number of distinct features;
- distance between the immediate values of the objects (primitive values or references); and
- for matching fields, object distance computed recursively with an attenuation factor.

Our measurements show that ART with object distance uncovers new faults but generally does not find faults faster than the basic random strategy, and misses some faults found by this strategy. It thus complements rather than replaces the basic random strategy.

Minimization

AutoTest preserves all failed tests, automatic or manual, for replay in regression testing.

Preserving the entire original scenario is generally impractical, since the execution may involve many irrelevant instructions. AutoTest's minimization algorithm attempts to derive a shorter scenario that still triggers the failure. The idea is to retain only the instructions that involve the target and arguments of the failing routine. Having found such a candidate, AutoTest executes it to check that it reproduces the failure; if it does not, AutoTest retains the original. While theoretically not complete, the algorithm is sound since its resulting scenario always triggers the same failure. In practice it is near-complete, often reducing scenario size by several orders of magnitude.[7]

Boolean queries

A promising strategy, comparable to techniques used for model checking, follows from the observation that classes often possess a set of argument-less Boolean-valued queries on the state: "is_overdraft" for a bank account; "is_empty" for any container structure; "after," stating that the cursor is past the last element, for a list with cursors. We investigated a *Boolean query conjecture*:[8] The argument-less Boolean queries of a well-written class yield a partition of the corresponding object state space that helps testing strategies.

The rationale for this conjecture is that such queries characterize the most important divisions of an object's possible states: An account is overdraft or not, it is open or closed, it bears interest or not. Combining them yields a representative partition of the space set, containing dramatically fewer elements. With a typical class, considering all possible instance states is intractable, but combining n Boolean queries yields 2^n possibilities, or *abstract query states*; in our experience, n is seldom more than 10—for example, only 25 percent of the 217 classes in the EiffelBase 6.4 library have more than 10 argument-less Boolean queries. The algorithm may limit this number further by considering only combinations that satisfy the invariant.

The conjecture suggests looking for a test suite that maximizes *Boolean query coverage* (BQC): the percentage of abstract states exercised. While this strategy is not yet a standard component of AutoTest, our experiments suggest that it may be useful. It involves trimming abstract query states through a constraint solver, then using a theorem prover for clauses involving noninteger queries. In experiments so far, the strategy yields a BQC close to 100 percent with minimal invariant adaptation; routine coverage increases from about 85 percent for basic AutoTest to 99 or 100 percent, and the number of faults found increases significantly.

Test Generation results

Table 1 shows results of applying Test Generation (no BQC) to the EiffelBase[9] and Gobo (www.gobosoft.com) data structure and algorithm libraries, widely used in operational applications, and to an experimental library providing complete specifications.

These results are typical of many more experiments. As the tested classes have different semantics and sizes in terms of various code metrics, the experiments appear representative of many problem domains. Since AutoTest is a unit testing tool and was used for this purpose in the experiments, we do not claim that these results are representative of the performance of contract-based random testing for entire applications or software systems.

Table 1. Test Generation results.

Tested library	Faults	Percent failing routines	Percent failed tests
EiffelBase	127	6.4 (127/1,984)	3.8 (1,513/39,615)
Gobo libraries	26	4.4 (26/585)	3.7 (2,928/79,886)
Specification library	72	14.1 (72/510)	49.6 (12,860/25,946)

TEST EXTRACTION

During development, programmers routinely execute the program to check that it proceeds as expected. They generally do not think of these executions as formal test cases. If results are wrong or the execution otherwise fails, they fix the problem and return to development; off goes a potentially interesting test, which could have benefited future regression testing. The programmers could create a test case, but most of the time they will not find the task worth the time—after all, they did correct the problem, or at least they addressed the symptoms.

Test Extraction will create the test for developers and give it the same status as any other manual or generated test. Figure 2 provides an example.

Figure 2a shows the state after a failure in a bank account class, with an incorrect implementation of "deposit" causing a postcondition violation when a user attempts to withdraw $100 from an account with a balance of $500. The lower part of the figure shows the source code of the routine "withdraw," containing an erroneous postcondition tagged "withdrawn": The plus should have been a minus. Execution causes the postcondition violation shown at the top part of the figure. The message is the normal EiffelStudio reaction to a postcondition violation, with the debugger showing the call stack.

Test Extraction's innovation is to turn this failure automatically into a test case. Figure 2b shows an example of an extracted test, including the different components necessary to reproduce the original exception: "test_withdraw" calls the routine "withdraw," and "context" describes the target object's state.

Subsequent test executions will display the status of the extracted test, which initially fails, as shown in Figure 2c. Once the postcondition has been corrected, the test will pass and the status will turn green.

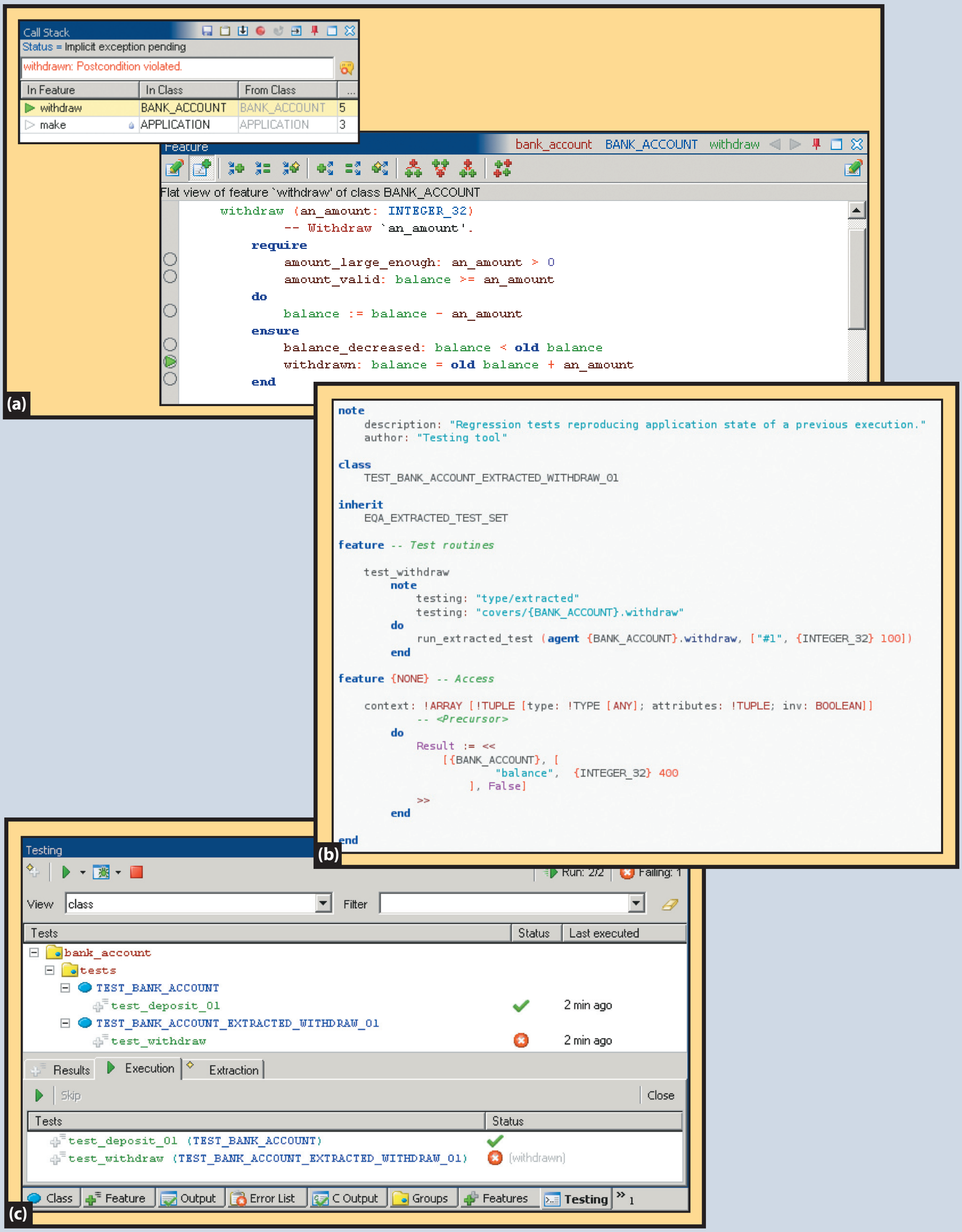
Minimization allows AutoTest to record and replay many such violations. The key idea is that it is not necessary to replay the program execution as it actually happened; as any failure is the result of calling a routine on a certain object in a certain object structure, it suffices to record that structure and, when replaying, to call the routine on the target object.

As software evolves, a test may become inapplicable. To address this situation Test Extraction will check, before replaying the test, that both the object's invariant and the routine's precondition hold. If either does not, it would make no sense to run the test; Test Extraction marks it invalid.[10]

EXAMPLE SESSION WITH AUTOTEST

Originally an independent tool, AutoTest is now simply the testing part of the EiffelStudio environment. To start the following example session, just launch EiffelStudio. While the functionalities are the same across all supported platforms, the user interface, shown for Windows in the screenshots in Figure 3, will have a different look and feel on, for example, Linux, Solaris, or Mac OS X.

To perform automatic tests on the application class BANK_ACCOUNT and the library classes STRING and



Status = Implicit exception pending

withdrawn: Postcondition violated.

In Feature	In Class	From Class	...
withdraw	BANK_ACCOUNT	BANK_ACCOUNT	5
make	APPLICATION	APPLICATION	3

Flat view of feature `withdraw' of class BANK_ACCOUNT

```
withdraw (an_amount: INTEGER_32)
        -- Withdraw `an_amount'.
    require
        amount_large_enough: an_amount > 0
        amount_valid: balance >= an_amount
    do
        balance := balance - an_amount
    ensure
        balance_decreased: balance < old balance
        withdrawn: balance = old balance + an_amount
    end
```

```
note
    description: "Regression tests reproducing application state of a previous execution."
    author: "Testing tool"

class
    TEST_BANK_ACCOUNT_EXTRACTED_WITHDRAW_01

inherit
    EQA_EXTRACTED_TEST_SET

feature -- Test routines

    test_withdraw
        note
            testing: "type/extracted"
            testing: "covers/{BANK_ACCOUNT}.withdraw"
        do
            run_extracted_test (agent {BANK_ACCOUNT}.withdraw, ["#1", {INTEGER_32} 100])
        end

feature {NONE} -- Access

    context: !ARRAY [!TUPLE [type: !TYPE [ANY]; attributes: !TUPLE; inv: BOOLEAN]]
        -- <Precursor>
        do
            Result := <<
                [{BANK_ACCOUNT}, [
                        "balance",   {INTEGER_32} 400
                    ], False]
            >>
        end

end
```

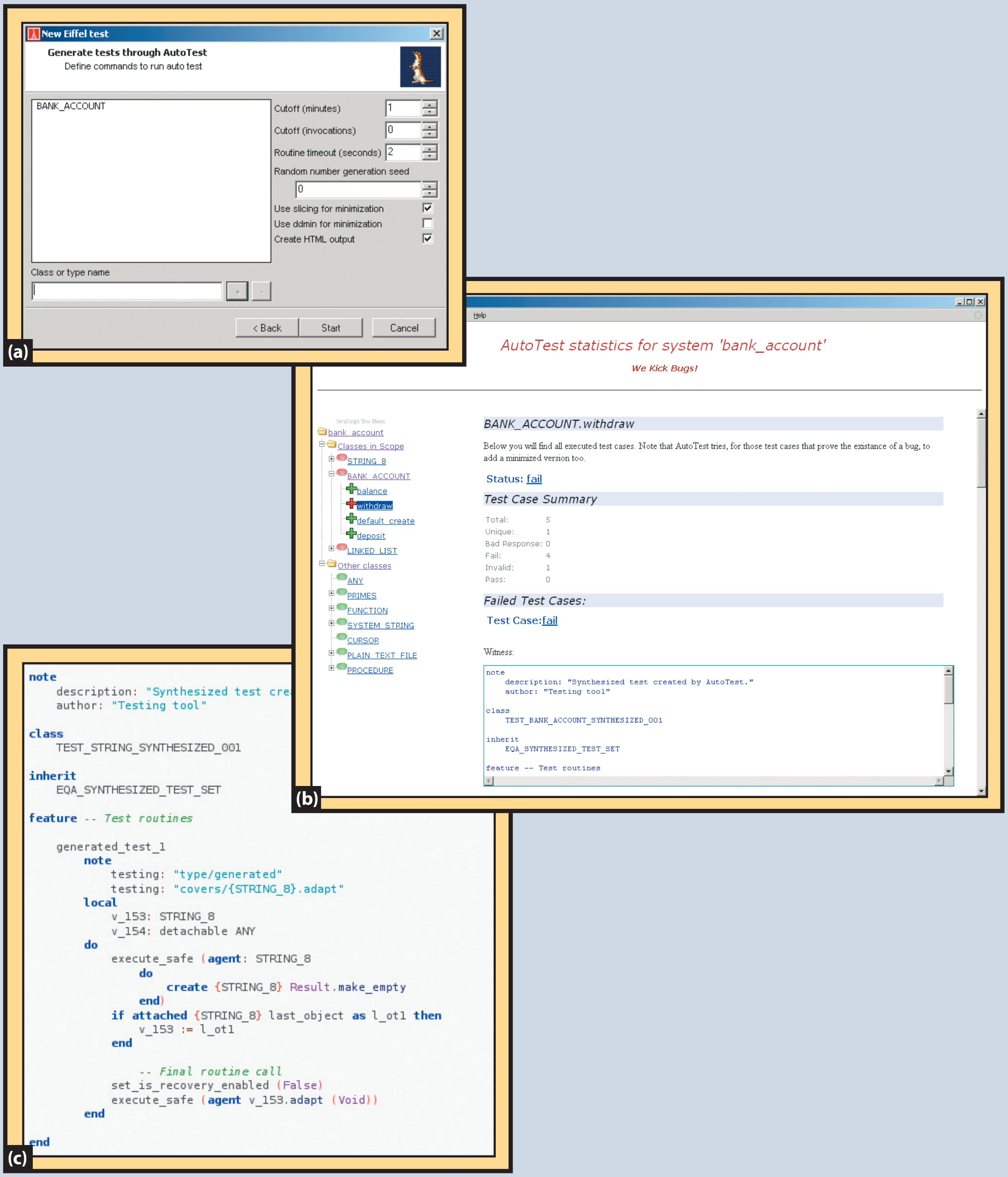
View: class

Filter:

Tests	Status	Last executed
bank_account		
tests		
TEST_BANK_ACCOUNT		
test_deposit_01	✓	2 min ago
TEST_BANK_ACCOUNT_EXTRACTED_WITHDRAW_01		
test_withdraw	✗	2 min ago

Tests	Status
test_deposit_01 (TEST_BANK_ACCOUNT)	✓
test_withdraw (TEST_BANK_ACCOUNT_EXTRACTED_WITHDRAW_01)	✗ (withdrawn)

Figure 2. Test Extraction example: (a) catching a contract violation, (b) turning this failure automatically into a test case, and (c) using the extracted test to reproduce the original exception.



Generate tests through AutoTest
Define commands to run auto test

BANK_ACCOUNT

Cutoff (minutes) 1
Cutoff (invocations) 0
Routine timeout (seconds) 2
Random number generation seed 0
Use slicing for minimization [✓]
Use ddmin for minimization [ ]
Create HTML output [✓]

Class or type name

AutoTest statistics for system 'bank_account'

We Kick Bugs!

- bank_account
  - Classes in Scope
    - STRING_8
    - BANK_ACCOUNT
      - balance
      - withdraw
      - default_create
      - deposit
    - LINKED_LIST
  - Other classes
    - ANY
    - PRIMES
    - FUNCTION
    - SYSTEM_STRING
    - CURSOR
    - PLAIN_TEXT_FILE
    - PROCEDURE

BANK_ACCOUNT.withdraw

Below you will find all executed test cases. Note that AutoTest tries, for those test cases that prove the existance of a bug, to add a minimized version too.

Status: fail

Test Case Summary

```
Total:         5
Unique:        1
Bad Response:  0
Fail:          4
Invalid:       1
Pass:          0
```

Failed Test Cases:

Test Case:fail

Witness:

```
note
    description: "Synthesized test created by AutoTest."
    author: "Testing tool"

class
    TEST_BANK_ACCOUNT_SYNTHESIZED_001

inherit
    EQA_SYNTHESIZED_TEST_SET

feature -- Test routines
```

```
note
    description: "Synthesized test cre
    author: "Testing tool"

class
    TEST_STRING_SYNTHESIZED_001

inherit
    EQA_SYNTHESIZED_TEST_SET

feature -- Test routines

    generated_test_1
        note
            testing: "type/generated"
            testing: "covers/{STRING_8}.adapt"
        local
            v_153: STRING_8
            v_154: detachable ANY
        do
            execute_safe (agent: STRING_8
                do
                    create {STRING_8} Result.make_empty
                end)
            if attached {STRING_8} last_object as l_ot1 then
                v_153 := l_ot1
            end

                -- Final routine call
            set_is_recovery_enabled (False)
            execute_safe (agent v_153.adapt (Void))
        end

end
```

Figure 3. Example session using AutoTest: (a) "New Eiffel test" wizard, (b) sample AutoTest statistics, and (c) minimized witness.

LINKED_LIST, launch the "New Eiffel test" wizard, as shown in Figure 3a. In the first pane, choose the radio button labeled "Synthesized test using AutoTest." The last wizard window will ask you to specify AutoTest parameters, such as the classes to be tested and how long random testing should be performed. AutoTest will test the classes listed and any others on which they depend directly or indirectly.

By default, AutoTest will report result statistics, contract violations, and other failures in HTML, as in Figure 3b. All three classes under test are marked in red, indicating that at least one feature test triggered a failure in each. Expanding the tree node shows the offending features: For BANK_ACCOUNT, "default_create," "balance," and "deposit" were successful (green), but "withdraw" had failures. Clicking it displays the failure details. This includes a *witness* for each failure: a test scenario, generated by the tool, which triggers the failure. Scrolling shows the first witness's offending instructions.

The witness reproduces the postcondition failure (resulting from a fault planted for illustration) encountered using Test Extraction. This means that AutoTest found the same erroneous postcondition as the manual process.

Figure 3c shows a real fault, in the routine "adapt" of the library class STRING. The seldom-used "adapt" serves to initialize a string from a manifest value "Some Characters," as an instance not of STRING but of some descendant MY_STRING. The witness reveals that "adapt" is missing a precondition requiring a nonvoid argument. Without it, "adapt" accepts a void but passes it on to "share," which demands a nonvoid argument. The fault, since corrected, was first uncovered by AutoTest.

We have used the AutoTest framework to perform large-scale experiments,[11-13] totaling tens of thousands of hours of CPU time, that investigate such questions as: How does the number of faults found by random testing evolve over time? Are more faults uncovered as contract violations or through other exceptions? How predictable is random testing? Are there more faults in the contracts or in the implementation? How do uncovered faults compare to those found by manual testing and by software users?

The AutoTest tools provide significant functional help. In addition, they yield a better understanding of the challenges and benefits of tests. Testing will never be an exact science; it is an imperfect approach that becomes useful when more ambitious techniques such as static analysis and proofs let us down. If we cannot guarantee the absence of faults, we can at least try to find as many as possible, and make the findings part of a project's knowledge base forever, replaying the scenarios before every new release to ensure old faults don't creep back in. While more modest than full verification, this goal is critical for practical software development. ■

Acknowledgments

The original idea for AutoTest came from discussions with Xavier Rousselot. Karine Bezault started the first version. Per Madsen provided useful suggestions on state partitioning. The development benefited from discussions with numerous people, in particular Gary Leavens, Peter Müller, Manuel Oriol, Alexander Pretschner, and Andreas Zeller. Bernd Schoeller suggested the use of Boolean queries to reduce state spaces, which Lisa (Ling) Liu studied experimentally. Test Extraction, as developed by Andreas Leitner, was originally called CDD (Contract-Driven Development). We presented an earlier version of this article, on Test Generation only, at SOFSEM 2007: B. Meyer et al., "Automatic Testing of Object-Oriented Software, *Proc. 33rd Conf. Current Trends in Theory and Practice of Software Development*, LNCS 4362, Springer, 2007, pp. 114-129. Design by Contract is a trademark of Eiffel Software.

References

1. B. Meyer, "Seven Principles of Software Testing," *Computer*, Aug. 2008, pp. 99-101.
2. IEEE Std. 610.12-1990, *IEEE Standard Glossary of Software Eng. Terminology*, IEEE, 1990.
3. B. Meyer, *Object-Oriented Software Construction*, 2nd ed., Prentice Hall, 1997.
4. P. Chalin, "Are Practitioners Writing Contracts?," *Rigorous Eng. Fault-Tolerant Systems*, LNCS 4157, Springer, 2006, pp. 100-113.
5. T.Y. Chen, H. Leung, and I. Mak, "Adaptive Random Testing," *Proc. 9th Asian Computing Science Conf.* (Asian 04), LNCS 3321, Springer, 2004, pp. 320-329.
6. I. Ciupa et al., "ARTOO: Adaptive Random Testing for Object-Oriented Software," *Proc. 30th Ann. Conf. Software Eng.* (ICSE 08), ACM Press, 2008, pp. 71-80.
7. I. Ciupa et al., "On the Predictability of Random Tests for Object-Oriented Software," *Proc. 2008 Int'l Conf. Software Testing, Verification, and Validation* (ICST 08), IEEE CS Press, 2008, pp. 72-81.
8. I. Ciupa et al., "Experimental Assessment of Random Testing for Object-Oriented Software," *Proc. 2007 Int'l Symp. Software Testing and Analysis* (ISSTA 07), ACM Press, 2007, pp. 84-94.
9. B. Meyer, *Reusable Software: The Base Object-Oriented Component Libraries*, Prentice Hall, 1994.
10. A. Leitner, "Contract Driven Development = Test Driven Development - Writing Test Cases," *Proc. 6th Joint Meeting of the European Software Eng. Conf. and the ACM SIGSOFT Symp. the Foundations of Software* (ESEC-FSE 07), ACM Press, 2007, pp. 425-434.
11. L. Liu, B. Meyer, and B. Schoeller, "Using Contracts and Boolean Queries to Improve the Quality of Automatic Test Generation," *Tests and Proofs*, LNCS 4454, Springer, 2007, pp. 114-130.
12. A. Leitner et al., "Efficient Unit Test Case Minimization," *Proc. 22nd IEEE/ACM Int'l Conf. Automated Software Eng.* (ASE 07), ACM Press, 2007, pp. 417-420.
13. I. Ciupa et al., "Finding Faults: Manual Testing vs. Random+ Testing vs. User Reports," *Proc. 19th Int'l Symp. Software Reliability Eng.* (ISSRE 08), IEEE Press, 2008, pp. 157-166.

***Bertrand Meyer** is Professor of Software Engineering at ETH Zurich (Swiss Federal Institute of Technology), Zurich, Switzerland, and cofounder and Chief Architect of Eiffel Software, based in Santa Barbara, Calif. His latest book is* Touch of Class: An Introduction to Programming Well

(Springer, 2009), based on the introductory programming course at ETH. He is a Fellow of the ACM and president of Informatics Europe. Contact him at bertrand.meyer@inf.ethz.ch.

***Arno Fiva** was an engineer at Eiffel Software and is now completing his MS at the Chair of Software Engineering at ETH Zurich. His research focuses on automated software testing. Contact him at fivaa@student.ethz.ch.*

***Ilinca Ciupa** was a research assistant at the Chair of Software Engineering at ETH Zurich, where she received a PhD in automated software testing. She is now an automation engineer at Phonak in Switzerland. Contact her at ilinca.ciupa@inf.ethz.ch.*

***Andreas Leitner** was a researcher at the Chair of Software Engineering at ETH Zurich, where he received a PhD in automated software testing, and is now an engineer at Google's Zurich office. Contact him at andreasleitner@google.com.*

***Yi Wei** was an engineer at Eiffel Software and is now a research assistant at the Chair of Software Engineering at ETH Zurich, where he is working toward his PhD in automated software testing and self-repairing programs. Wei received an MS in engineering from Wuhan University in China. Contact him at yi.wei@inf.ethz.ch.*

***Emmanuel Stapf** is a senior software developer at Eiffel Software, where he leads the EiffelStudio development team. His research interests include compiler development, integrated development environments, and testing. Stapf received an engineer's degree from ENSEEIHT in Toulouse, France. Contact him at manus@eiffel.com.*