Cite this paper as follows: Bertrand Meyer, Ilinca Ciupa, Andreas Leitner and Lisa (Ling) Liu, *Automatic Testing of Object-Oriented Software*, in SOFSEM 2007 (Current Trends in Theory and Practice of Computer Science, Harrachov, Czech Republic, 20-26 January 2007), ed. Jan van Leeuwen, to appear in Lecture Notes in Computer Science, Springer-Verlag, 2007.

Automatic testing of object-oriented software

Bertrand Meyer, Ilinca Ciupa, Andreas Leitner, Lisa (Ling) Liu

Chair of Software Engineering, ETH Zurich, Switzerland
http://se.ethz.ch

{Firstname.Lastname}@inf.ethz.ch

Abstract. Effective testing involves preparing test oracles and test cases, two activities which are too tedious to be effectively performed by humans, yet for the most part remain manual. The AutoTest unit testing framework automates both, by using Eiffel contracts — already present in the software — as test oracles, and generating objects and routine arguments to exercise all given classes; manual tests can also be added, and all failed test cases are automatically retained for regression testing, in a "minimized" form retaining only the relevant instructions. AutoTest has already detected numerous hitherto unknown bugs in production software.

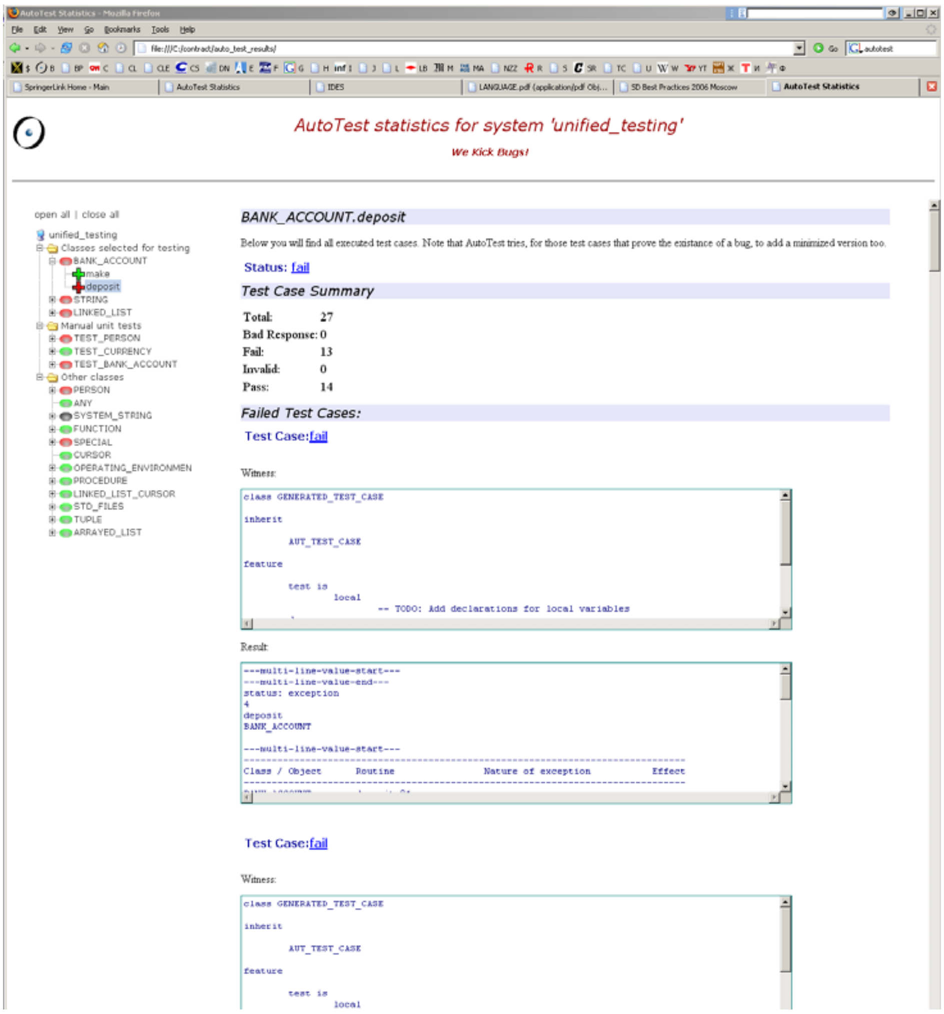
Keywords: automated software engineering, automatic testing, testing frameworks, regression testing, constraint satisfaction, Eiffel, Design by Contract.

1 Overview: a session with AutoTest

Testing remains, in the practice of software development, the most important part of quality assurance. It's not a particularly pleasant part, especially because of two tasks that consume much of the effort: preparing test *cases* (the set of values chosen to exercise the program) and test *oracles* (the criteria to determine whether a test run has succeeded). Since these tasks are so tedious, testing is often not thorough enough, and as a consequence too many bugs remain. A survey by the US National Institute of Standards and Technology [22] added a quantitative touch to this observation by boldly assessing the cost of inadequate software testing for 2002 at $59.5 billion, or 0.6% of the US GNP. One of the principal obstacles is the large amount of manual work still involved in the testing process. Recent years have seen progress with the advent of testing frameworks such as JUnit [13] which, however, only automate test management and execution, not the most delicate and tedious parts. Further improvements to the effectiveness of testing require more automation.

The testing framework described in this article, AutoTest, permits completely automated unit testing, applied to classes (object-oriented software components). The only input required of a user of AutoTest is the list of classes to be tested. This is possible because of two properties of object technology as realized in Eiffel [18]:

- Each class represents a set of possible run-time *objects* and defines the set of *features* applicable to them; the testing framework can (thanks to the power of today's computers) generate large numbers of sample objects and perform large numbers of feature calls on them. This takes care of test case generation.
- Classes are equipped with *contracts* [17] [20], specifying expected behavior, which can be monitored during execution. This takes care of test oracles [3].



AutoTest statistics for system 'unified_testing'

We Kick Bugs!

BANK_ACCOUNT.deposit

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:	27
Bad Response:	0
Fail:	13
Invalid:	0
Pass:	14

Failed Test Cases:

Test Case:fail

Witness:

```
class GENERATED_TEST_CASE

inherit

        AUT_TEST_CASE

feature

        test is
                local
                        -- TODO: Add declarations for local variables
```

Result:

```
---multi-line-value-start---
---multi-line-value-end---
status: exception
4
deposit
BANK_ACCOUNT

---multi-line-value-start---
-------------------------------------------------------------------------------
Class / Object      Routine                    Nature of exception          Effect
-------------------------------------------------------------------------------
```

Test Case:fail

Witness:

```
class GENERATED_TEST_CASE

inherit

        AUT_TEST_CASE

feature

        test is
                local
```

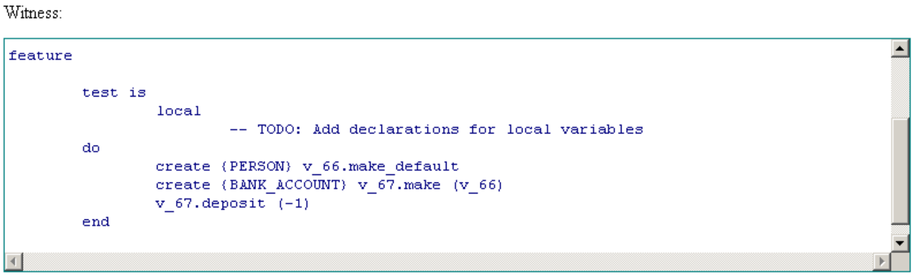
Figure 1: Sample AutoTest output

A typical use of AutoTest (from the command line) is

auto_test *time_limit compilation_options BANK_ACCOUNT STRING LINKED_LIST* where *time_limit* is the maximum time given to AutoTest for exercising the software, *compilation_options* is the control file (generated by the EiffelStudio session and thus already present) governing compilation of a system, and the remaining arguments are names of classes to be tested. AutoTest will also exercise any other classes on which these classes depend directly or indirectly, for example library components.

This information suffices to specify an AutoTest session which will, in the prescribed time, test the given classes and their features, using various heuristics to maximize testing effectiveness; it will evaluate the contracts and log any contract violations or other failures. At the end of the session, AutoTest reports all failures, using by default the HTML format illustrated in figure 1. Most failures reflect bugs in the software.

In the example results, some classes are marked in red[1]: all three classes under test, some classes with manual tests (as explained next), and two of the supporting classes (*PERSON* and *SPECIAL*). Such a mark indicates that a test case for one or more features triggered a failure. Expanding the tree node shows the offending features: for *BANK_ACCOUNT* feature *make* was OK (green), but *deposit* had failures. Clicking the name of this feature displays, on the right side of the figure, the details of these failures. For each failed test, this includes a **witness**: a test scenario, automatically generated by AutoTest, which triggered the failure. The beginning of the first witness appears in figure 1, as part of a class *GENERATED_TEST_CASE* (the context is visible in figure 1) generated by AutoTest. Scrolling shows the offending instructions:


Witness:

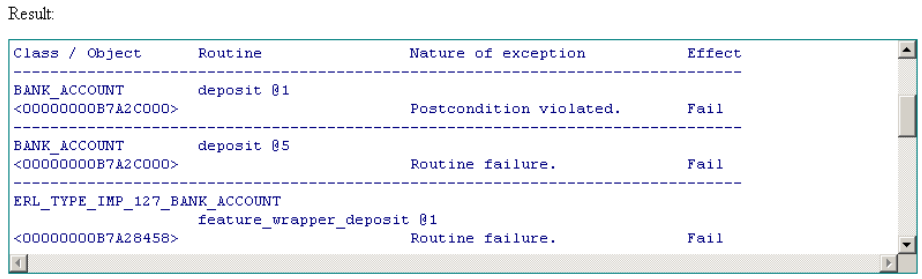
```
feature

        test is
                local
                        -- TODO: Add declarations for local variables
        do
                create {PERSON} v_66.make_default
                create {BANK_ACCOUNT} v_67.make (v_66)
                v_67.deposit (-1)
        end
```

Figure 2: An automatically generated test witness leading to a failure

This automatically generated code creates an instance of *PERSON* and another of *BANK_ACCOUNT* (whose creation refers to the person, *v_66*, as the owner of the account), then deposits a negative amount — causing a precondition violation in a called routine, as detailed in the next subwindow, which (after scrolling) shows the failure trace:

[1] In a B&W printout these six classes appear with darker ellipses. All others except one have green ellipses, appearing lighter; this indicates that no failure occurred. *SYSTEM_STRING* has a gray ellipse indicating that no test was performed.



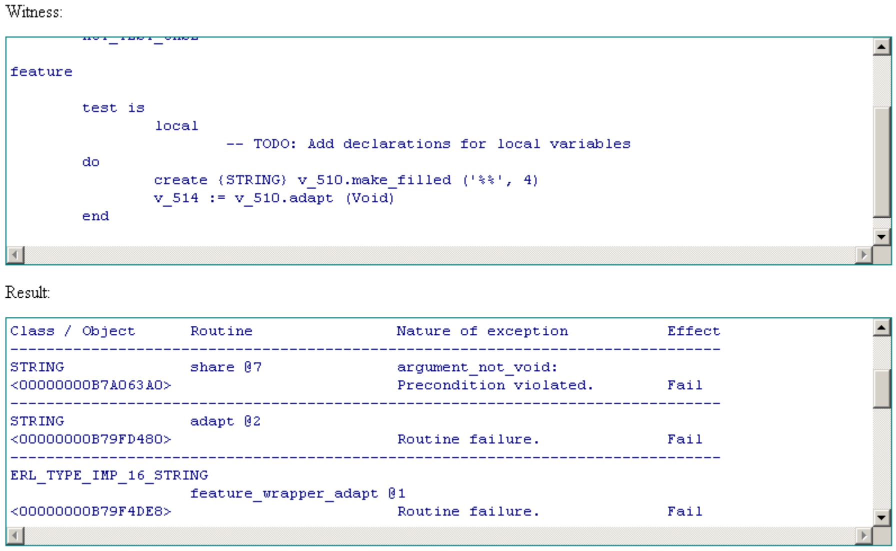
Result:

```
Class / Object      Routine                 Nature of exception           Effect
--------------------------------------------------------------------------------
BANK_ACCOUNT        deposit @1
<00000000B7A2C000>                          Postcondition violated.       Fail
--------------------------------------------------------------------------------
BANK_ACCOUNT        deposit @5
<00000000B7A2C000>                          Routine failure.              Fail
--------------------------------------------------------------------------------
ERL_TYPE_IMP_127_BANK_ACCOUNT
                    feature_wrapper_deposit @1
<00000000B7A28458>                          Routine failure.              Fail
```

Figure 3: Failure trace

Failure witnesses, as in figure 2, appear in a **minimized** form. AutoTest may originally have detected the failure through a longer sequence of calls, but will then compute a minimal sequence that leads to the same result. This helps AutoTest users understand and correct the bug by reducing it to a simpler case. Minimization is also critical for regression testing: AutoTest records all failure scenarios and re-tests them in all subsequent test runs; but since the path to a failure often goes through many irrelevant detours, it is essential for the efficiency of future test runs to remove these detours, keeping only the instructions that take part in causing the failure. AutoTest uses a *slicing* technique (see e.g. [25]) for witness minimization.

In the case of *deposit* in class *BANK_ACCOUNT* the bug was planted, for purposes of illustration. But AutoTest routinely finds real, unexpected bugs in production software. This is indeed apparent on the example where *STRING* and *LINKED_LIST*, basic library classes, appear in red. The corresponding bugs, since corrected, were unknown until AutoTest was applied to these classes. In each case they affected a single feature. For *STRING* the feature is *adapt*:



Witness:

```
feature

        test is
                local
                        -- TODO: Add declarations for local variables
        do
                create {STRING} v_510.make_filled ('%%', 4)
                v_514 := v_510.adapt (Void)
        end
```

Result:

```
Class / Object      Routine                 Nature of exception           Effect
--------------------------------------------------------------------------------
STRING              share @7                argument_not_void:
<00000000B7A063A0>                          Precondition violated.        Fail
--------------------------------------------------------------------------------
STRING              adapt @2
<00000000B79FD480>                          Routine failure.              Fail
--------------------------------------------------------------------------------
ERL_TYPE_IMP_16_STRING
                    feature_wrapper_adapt @1
<00000000B79F4DE8>                          Routine failure.              Fail
```

Figure 4: Test witness for bug in STRING class

(Bug explanation: *adapt* is is a little-used operation, applicable when a programmer has defined a descendant *MY_STRING* of the basic library class *STRING*: it is invalid to assign a manifest string written in the usual notation "*Some Explicit Characters*", of type *STRING*, to a variable of type *MY_STRING*, but *adapt* will yield an equivalent object of type *MY_STRING*. Figure 4 shows the test witness, revealing a bug: *adapt* should include a precondition requiring a non-void argument. Without it, *adapt* accepts a void argument but passes it on to a routine *share* that demands non-void.)

While AutoTest, as illustrated, provides extensive mechanisms for automated test generation, the framework also supports *manual* tests. The two approaches are complementary, one providing breadth, the other depth: automatic tests are good at exercising components much more extensively than a human tester would ever accomplish; manual tests can take advantage of domain knowledge to test specific scenarios that an automatic mechanism would not have the time to reach. AutoTest closely combines the two strategies:

- The framework records manual tests and re-runs them in every session, along with the manual tests.
- *Any* failed automatic test becomes part of the manual test suite, to be re-run in the future for regression testing. This reflects what we take as a software engineering principle: any failed execution is a significant event of a project's history, and must become an indelible element of the project's memory.

These are the basic ideas behind AutoTest as seen by its users. To make them practically useful, AutoTest provides a full testing infrastructure. In particular:

- The architecture is multi-process: the master AutoTest process is only in charge of starting successive test runs in another process. This allows AutoTest to continue if any particular test crashes.
- For automatic tests, AutoTest uses a combination of strategies to increase the likelihood of finding useful contract violations. Strategies include adaptive random testing [6], which we have generalized to objects by introducing a notion of "object distance", the reliance on boolean queries to extract significant abstract object state, and techniques borrowed from program proving and constraint satisfaction to avoid spurious test cases (those violating a routine precondition). These techniques are detailed in the following sections.

Two limitations of the current state of the AutoTest work should be mentioned. First, the framework focuses on the testing of individual program components; it does not currently address such issues as GUI testing. Second, although this paper includes some experimental results, we do not profess to have decisive quantitative evidence of AutoTest's effectiveness at finding bugs, or of its superiority to other approaches. Our efforts so far have been directed at building the framework and making it practical. More systematic collection of empirical results is now in progress.

As a compensation for these limitations it is important to note that AutoTest is not an exploratory prototype but a directly usable system, built with the primary objective of helping practicing developers find bugs early. The tool has already found numerous previously unknown bugs in libraries used in deployed production systems.

This is possible because what AutoTest examines is software as it actually exists, without any special instrumentation. Other tools using some of the same general ideas (see section 8 for references on previous work) generally require software that has been especially prepared for testing; for example, Java code as it is commonly written does not include contracts, and so must be extended with JML (Java Modeling Language) assertions, or special "repOk" routines representing class invariants, to lend itself to contract-based testing; in the case of "model-based testing", test generation is only possible if someone has produced a detailed model of the intended behavior, based for example on Statecharts. It is not easy in practice to impose such extra work on projects. It is even harder to guarantee, assuming a model was initially produced, that the project will keep it up to date as the software changes. AutoTest in contrast takes existing software, which in Eiffel typically has the contracts. Of course these contracts are often not exhaustive, and the quality of AutoTest's results will improve with the quality of the contracts; but even with elementary contracts, experience shows that AutoTest finds significant bugs.

The following sections describe the concepts behind AutoTest.

To apply AutoTest to their own software, and to reproduce, criticize or extend our results, readers can download AutoTest both as a binary and (under an open-source license) in source form from the AutoTest page [1].

2 Automated testing

The phrase "automated testing" is widely used, to describe techniques which automate various aspects of the process. It is important to clarify the terminology. The following components of the testing activity can be the target of automation:

1. **Test management and execution**: even if test data and oracles are known, just setting up a test session and running the tests can be a labor-intensive task for a large system or library. Test execution frameworks, working from a description of the testing process, can perform that process automatically.
2. **Failure recovery**: this is an important addition to the previous step, allowing testing to continue even if a test case causes execution to fail — as is bound to happen with a large set of test cases and a large system. Producing failures is indeed among the intended outcomes of the testing process, as a way to uncover bugs, but this shouldn't stop the testing process itself. Automating this aspect requires using at least two communicating processes, one driving the test plan and the other performing the tests. The first one should not fail; the second one may fail, but will then be restarted after results have been logged.

3. **Regression testing**: after a change, re-running an appropriate subset of earlier tests is a fundamental practice of a good testing process. It can be automated if an infrastructure is in place for recording test cases and test results.
4. **Script-driven GUI testing**: after recording a user's inputs during an interactive session, run the session again without human intervention.
5. **Test case minimization**: after a failed test run, devise the shortest possible test run that produces the same failure. This facilitates debugging and is, as noted, particularly important for regression testing.
6. **Test case generation**.
7. **Test oracle generation**: determining whether a test run has passed or failed.

As used most commonly in the literature "automated testing" denotes some combination of applications 1 to 3 in this list. They are a prerequisite for more advanced applications, and indeed supported by AutoTest; but AutoTest also addresses 5, 6 and 7 which, as noted, correspond to the most delicate and time-consuming of testing, and hence are of particular importance.

Because AutoTest is directed at the testing of software components through their API, script-driven GUI testing (4) is, as noted, beyond its current scope.

3 The testing process

Automatic component testing involves several steps:

- Generating inputs; for the testing of O-O software components this will mean generating objects.
- Selecting a subset of these objects for actual testing.
- Selecting arguments for the features to be called on these objects.
- Running the tests.
- Assessing the outcome, pass or fail.
- Logging relevant outcomes, in particular failures.

The following describes the strategies used for the most delicate of these steps.

3.1 Creating target objects

Testing a class means, for AutoTest, testing a number of *routine calls* on instances of the class. Constructing such a call, the basic unit of AutoTest execution, requires both a target object and, if the routine takes arguments, values for these arguments. We first consider the issue of generating objects.

AutoTest will generate objects and retain them in an **object pool** for possible later reuse. The general strategy, when an object of type *T* is needed — as target of a call, but also possibly as argument to a routine — is the following, where some steps involve pseudo-random choices based on preset frequencies.

The first step is to decide whether to create a new instance of *T*. This is necessary if the pool does not contain any object of a type conforming to *T*, but may also happen, based on pseudo-random choice, if such objects are available; the aim in this case is to **diversify** the object pool.

If the decision is to create an object, the algorithm will:

- Choose one of the creation procedures (constructors) of the class.
- Choose argument values for this procedure, if needed, using the strategies defined below. Note that some of these arguments may represent objects, in which case the algorithm will call itself recursively.
- Call the creation procedure with the selected arguments. This might cause a failure (duly logged by AutoTest), but normally will produce an object.
- Choose for the test an object from the pool — not necessarily the one just created.

To achieve further diversification, AutoTest chooses after certain test executions — again with a preset frequency — to call a modifier feature on a randomly selected object, with the sole purpose of creating one or more new objects for the pool. This is in addition to the diversification that occurs already as part of performing the tests, since all objects produced by feature calls are added to the pool.

Various settable parameters control the choices involved in the algorithm.

The approach just described differs from strategies used in earlier work. For example Korat [4] directly sets object fields to create all non-isomorphic inputs, up to a certain bound, for a routine under test. This may lead to the creation of meaningless objects (not satisfying the class invariant), requiring filtering, or of objects that actual executions would not normally create. In contrast, AutoTest obtains objects by calling creation procedures of the class, then its routines; hence it will only create objects similar to those resulting from normal runs of the application.

3.2 Selecting routine arguments

Many routine calls require arguments. These can be either object references or basic values; in Eiffel the latter are instances of “expanded” types such as *INTEGER*, *BOOLEAN*, *CHARACTER*, *INTEGER* and *REAL*.

For objects, the strategy is as described above: get an object from the pool, or create a new object.

For basic values, the current strategy is to select randomly from a set of values preset for each type. For example the preset values for *INTEGER* include 0, the minimum and maximum integer values, +/−1, +/−2, +/−10, +/−100, and a few others.

Benefits expected from this approach include: ease of implementation and understanding; lack of bias; and speed of execution, which ultimately translates into an increased number of test runs.

While these techniques have given good results so far, we intend to conduct a more systematic and quantitative assessment of the choices involved, using two complementary criteria: which choices actually uncover hitherto unknown bugs; and, as an estimate of the likelihood of finding new bugs in the future, how fast they uncover known bugs.

3.3 Adaptive random testing and object distance

The strategies just described for choosing objects and other values are random. Adaptive random testing (ART) has been proposed by Chen et al. [6] to improve on random selection through techniques that spread out the selected values over the corresponding intervals. For example, integers should be evenly spaced. Experimental results show that ART applied to integer arguments does yield more effective tests.

Work on ART has so far only considered inputs of primitive types such as integers, for which this notion of evenly spaced values has an immediate meaning: the inputs belong to a known interval with a total order relation. But in object-oriented programming many of the interesting inputs are objects or object references, for which there is no total order. How do we guarantee that a set of objects is “representative” of the available possibilities in the same way as equally spread integers?

To address this issue we have defined [9] a notion of *object distance* to determine how “far” an object is from another, and use it as a basis for object selection strategies. The object distance is a normalized weighted sum of the following three properties, for two objects *o1* and *o2*:

- The distance between their types, based on the length of the path from one type to the other in the inheritance graph, and the number of non-common fields.
- The distance between the immediate (non-reference) values of their matching fields, using a simple notion of distance for basic types (difference for integers, Levenshtein distance for strings).
- For matching fields involving references to other objects, their object distances, computed recursively.

Although the released version of AutoTest (leading to the results reported below) does not yet use adaptive random testing, we have added ART, with the object distance as defined, for inclusion in future releases. The first evaluations appear to indicate that ART with object distance does bring a significant improvement to the testing process.

3.4 Contracts as oracles

Test oracles decide whether a test case has passed or failed. Devising oracles can, as noted, be one of the most delicate and time-consuming aspects of testing. In testing Eiffel classes, we do not write any separate description for oracles, but simply use the contracts already present in Eiffel.

Contracts state what conditions the software must meet at certain points of the execution and they can be evaluated at runtime. They include:

- The **precondition** of a routine, stating the conditions to be established (by the callers) *before* any of its executions.
- The **postcondition** of a routine, stating conditions to be established (by the routine) *after* execution.
- The **invariant** of a class, stating conditions that instances of the class must fulfill upon creation (after execution of a creation procedure) and then before and after executions of exported routines.

The Design by Contract approach [17] [20] does not require a fully formal specification; contracts can be partial. In practice, preconditions tend to be exhaustive, because without them routines could be called incorrectly; postconditions and class invariants are more or less extensive depending on the developer's style. But as Chalin's analysis [5] of both public-domain and commercial software shows, Eiffel programmers do use contracts (accounting overall for 4.4% of the code); this provides AutoTest with a significant advantage over approaches where specifications of the software's intent must be added before testing can proceed.

Contracts take the form of boolean expressions augmented, in the case of postconditions, by "**old**" expressions to express the relationship between the final state of a routine's execution to its initial state, as in a postcondition clause of the form *counter* = **old** *counter* + 1. Because boolean expressions can rely on function calls, contracts can express arbitrarily complex properties.

Because contracts use valid expressions of the programming language, they can be evaluated during execution. This property — which has always been central in the Eiffel approach to debugging — enables AutoTest to use contracts as oracles. To understand this more precisely, it is necessary to consider the relationship of contract violations to bugs:

- Since establishing a precondition is the responsibility of a routine's client (caller), its violation signals a possible bug in the client.
- Conversely, a postcondition or invariant violation signals a possible bug in the supplier.

(In both cases the violation establishes a "possible" bug because the error can occasionally be due to an inappropriate contract rather than to an inappropriate implementation.) This means that if we consider the execution of AutoTest as a game aimed at finding as many bugs as possible we must distinguish between favorable and unfavorable situations:

1. A postcondition or invariant violation is a *win* for AutoTest: it has uncovered a possible bug in the routine being called.
2. A precondition violation, for a routine, called directly by AutoTest as part of its strategy, is a *loss*: the object and argument generation strategy has failed to produce a legitimate call. The call will be aborted; AutoTest has wasted time.
3. If, however, a routine *r* legitimately called by AutoTest, directly or indirectly, attempts to call another routine with its precondition violated, this is evidence of a problem in *r*, not in AutoTest: we are back to a *win* as in case 1.

AutoTest's strategy must be, as a consequence, to minimize occurrences of direct precondition violations (case 2), and to maximize the likelihood of cases 1 and 3. All violations matching these cases will be logged, together with details about the context: exact contract clause being violated, full stack trace. This information is very useful for interpreting the results of an AutoTest session: determining whether the violation signals a bug, and if it is (the most usual case), correcting the bug.

3.5 Test case minimization

For regression testing, it is important to record any scenario that has been found — at any time in the life of a project — to produce a failure. The naïve solution of preserving the original scenario is, as noted, generally impractical, since the sequence leading to the bug could be very long, involving many irrelevant instructions.

AutoTest includes a minimization algorithm, which attempts to derive a scenario made of a minimal set of instructions that still triggers the bug. The basic idea is to retain only the instructions that involve the inputs (target object and arguments) of the routine where the bug was found. Having found such a candidate minimum, AutoTest executes it to check that it indeed reproduces the bug; if not, it retains the original. While this process is not guaranteed to yield a minimum, it is guaranteed to yield a scenario that triggers the failure. Our experiments show that in practice the algorithm reduces the size of bug-reproducing examples by several orders of magnitude.

4 Integration of manual and automatic tests

No human input can match the power of an automatic strategy to generate thousands or millions of test cases, relentlessly exercising software components. In some cases, however, humans just know what kind of inputs to look for. A typical example is a parsing component taking as output a long string representing a program or program part. Automatic argument generation strategies are unlikely to generate interesting inputs in such a case.

AutoTest is not just a test case generation tool but a general testing framework, and closely integrates manual tests with automatic ones. The node labeled “Manual unit tests” at the top level of the tree on the left of figure 1 can be expanded to show details of manual tests.

AutoTest, as already noted, turns any failed test scenario into a manual test, and runs all manual tests at the beginning of a session. During test execution, no distinction is made between the two kinds; for example, manual tests contribute to the object pool and participate in object diversification as described in section 3. [14] contains more details on the integration of the two strategies.

5 Experimental results

Table 1 shows some results of applying AutoTest (without manually added tests) to a number of libraries and applications.

EiffelBase is the basic library of data structures and algorithms [19]; Gobo [2] is another set of libraries, partly covering the same ground as EiffelBase, and partly complementary. Both are widely used, including in commercial applications. Results for these libraries are shown both for each library as a whole and for some of its clusters (sub-libraries).

EWG (Eiffel Wrapper Generator), available for several years, is an open-source application for producing Eiffel libraries by wrapping C libraries.

The other two are much more recent, which probably explains the higher number of failures. One is a set of classes for specifying high-level properties, using the Perfect Developer proof system. DoctorC is an open-source application.

The figures in the last two columns of the table were extracted directly from the information that AutoTest outputs at the end of every testing session.

On the other hand the “number of bugs” (second column) can only result from human interpretation, to determine whether each failure really corresponds to a bug. It should be noted, however, that so far there are essentially no false alarms — such as affect, for example, the results of many static analyzers — in AutoTest’s results: almost every violation reflects either an implementation bug or a contract that does not express the intended effect of the implementation (and hence is also a bug, although it has no effect on execution).

A significant proportion of the bugs currently found by AutoTest have to do with “void” issues: failure of the code to protect itself against feature calls on void (null) targets. This issue should go away with the next version of Eiffel, which will handle it statically as part of the type system [21]; but in the meantime the prevalence of such bugs in the results highlights the importance of the problem, and AutoTest enables developers to remove many potential run-time crashes resulting from void calls.

Tested library/ application	Number of bugs found	Routines causing failures / Total tested routines	Failed tests/ total tests
EiffelBase: all	127	6.40% (127/1984)	3.8% (1513/39615)
EiffelBase: kernel	16	4.6% (16/343)	1.3% (204/15140)
EiffelBase: Support	23	10.7% (23/214)	5.1% (166/3233)
EiffelBase: Data structures	88	6.3% (88/1400)	5.4% (1143/21242)
Gobo: all	26	4.4% (26/585)	3.7% (2928/79886)
Gobo: XML	17	3.8% (17/441)	3.7% (2912/78347)
Gobo: Math	9	6.2% (9/144)	1% (16/1539)
Specification Library for Perfect Developer	72	14.1% (72/510)	49.6% (12860/25946)
DoctorC	15	45.4% (15/33)	14.3% (1283/8972)
EWG	8	10.38% (8/77)	1.32% (43/3245)

Table 1. AutoTest results for some libraries and applications

6 Optimization

The effectiveness of the automatic testing strategy described above is highly dependent on the quality of the object pool. We are currently investigating *abstract-query partitioning* (AQP) as a way to improve this quality. The following is an outline of AQP; see [16] for more details.

A **partitioning** strategy divides the space of possible object states, for a certain class, into a set of disjoint sub-spaces, and directs automatic testing to pick objects from every sub-space. For this be useful, the sub-spaces must be representative of the various cases that may occur during executions affecting these objects.

AQP takes into account the special nature of object-oriented software where, typically, a class is characterized (among other features) by a set of boolean-valued queries returning information on the state. Examples are *is_overdraft* for a bank account class, *is_empty* for a list or other data structure class, and *after*, expressing that the cursor is after the last element, for a class representing lists or other structures with cursors [19]. We only consider such queries if they have no argument (for simplicity) and if they are exported (so that they are part of the abstract properties of the object state as available to clients).

The intuition behind AQP is expressed by the following conjecture:

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.

Argumentless queries indeed seem to play a key role in the understanding and use of many classes; this suggests they may also be useful for partitioning object states to obtain more representative test objects.

An **abstract object state** for a class will be determined by the values, true or false, of all such queries. If a class has *n* queries, the number of abstract object states is 2^n. This is smaller, by many orders of magnitude, than the size of the full object state; typical values of *n*, for classes we have examined, are less than a dozen. Still, 2^n may still be too high in practice, but we need only consider abstract states that satisfy the class invariant; this generally reduces the size to a tractable value.

We have experimentally implemented AQP through a two-step strategy:

- Using a constraint solver, currently SICStus [23], generate a first set of abstract object states. This set is already filtered by the invariant, but using only the invariant clauses that only involve abstract queries, since these are the only ones that the constraint solver can handle; for example an invariant clause of the form is_*empty* **implies** *after* will take part in this step, but not *count* < *capacity*, involving integer-valued queries *count* (current size of the structure) and *capacity* (maximum size).
- Then, trim the abstract space further by reintroducing the invariant clauses initially ignored and, with the help of a theorem prover, currently Simplify [11], discarding states that would violate these clauses.

For a *FIXED_LIST* class close to the actual version in EiffelBase, the number of abstract queries is 9, resulting in an abstract space with 512 elements. Constraint solving reduces this number to 122, and theorem proving brings it down to 22.

This strategy suggests a new criterion for test coverage: **boolean query coverage**. A set of tests for a class satisfies this criterion if and only if their execution covers all the abstract object states that satisfy the class invariant. To achieve boolean query coverage, we have developed a *forward exploration* process which:

- Creates some objects through various creation procedures to generate an initial set of abstract object states.
- Executes exported routines in these states to produce more states.
- Repeats this step until it either finds no new abstract object states or reaches a predefined threshold (of number of calls, or of testing time).

Although AQP and this forward testing strategy are not yet part of the released version of AutoTest, our experiments make the approach look promising. Applying the ideas to a number of classes similar to classes of EiffelBase (but not yet identical due to current limitations of the AQP implementation regarding inheritance) yields an initial boolean query coverage of 80% or higher. Manual inspection of the results then enables us quickly to uncover missing properties in invariants and, after filling them in, to achieve 100% coverage. Running AutoTest with this improved strategy yields significant improvements on the examples examined so far, as shown by Table 2.

Tested Class	Random Testing (current AutoTest)		AutoTest extended with AQP	
	Routine Coverage	Bugs Found	Routine Coverage	Bugs Found
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

Table 2. Comparison of boolean query coverage with random testing

We are continuing our experiments and hope to include the AQP strategy in the standard version of AutoTest.

8 Previous work

This section does not claim exhaustivity but simply cites some earlier work that we have found useful in developing our ideas.

Binder [3] emphasizes the importance of using contracts as test oracles. Peters and Parnas [24] use oracles derived from specifications, separate from the program.

The jmlunit tool [7, 8] pioneered some of the ideas used here, in particular the use of postcondition contracts as oracles, and made the observation that a test that directly causes a precondition violation does not signal a bug. In jmlunit as described in these references, test suites remain the user's responsibility.

The Korat system [4] is based on some of the same concepts as the present work; to generate objects it does not use constructors but fills object fields and discards the result if it does not satisfy the invariant. Using the actual constructors of the class seems a more effective strategy.

9 Further development

Work is proceeding to improve the effectiveness of AutoTest in the various directions cited earlier, in particular adaptive random testing and abstract query partitioning. We are also performing systematic empirical evaluations of the effectiveness of various strategies used by AutoTest or proposed for future extensions. In parallel, we are also exploring the integration of testing techniques with other approaches, in particular proofs.

A significant part of our current work is devoted to the usability of AutoTest for ordinary software development. While AutoTest is currently a separate tool, developers would benefit greatly if it were integrated into the standard IDE; such efforts have now been made possible by the open-sourcing of the EiffelStudio environment [12]. Such efforts are consistent with our goal of developing AutoTest not only as a research vehicle, but more importantly as an everyday resource for practicing developers determined to eradicate bugs from their software before they have had the opportunity to harm.

Acknowledgments

The original idea for AutoTest came out of discussions with Xavier Rousselot. The first version (then called TestStudio) was started by Karine Arnout. Per Madsen provided useful suggestions on state partitioning. The development also benefited from discussions with numerous people, in particular Gary Leavens, Manuel Oriol, Peter Müller, Bernd Schoeller and Andreas Zeller.

Design by Contract is a trademark of Eiffel Software.

References

1. AutoTest page at se.ethz.ch/research/autotest/.
2. Eric Bezault et al.: Gobo library and tools, at www.gobosoft.com.
3. Robert V. Binder: *Testing Object-Oriented Systems: Models, Patterns and Tools*, Addison-Wesley, 1999.
4. C. Boyapati, S. Khurshid and D. Marinov: *Korat: Automated Testing Based on Java Predicates*, in 2002 ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2002), Rome, Italy, 2002.
5. Patrice Chalin: *Are Practitioners Writing Contracts?*, in Proceedings of the Workshop on Rigorous Engineering of Fault-Tolerant Systems (REFT 2005), Technical Report CS-TR-

915, eds. Michael Butler, Cliff Jones, Alexander Romanovsky, and Elena Troubitsyna, University of Newcastle upon Tyne, 2005.

6. TY Chen, H. Leung and I. Mak: *Adaptive random testing*, in M. J. Maher (ed.), Advances in Computer Science – ASIAN 2004: Higher-Level Decision Making, 9th Asian Computing Science Conference, Springer-Verlag, 2004.
7. Yoonsik Cheon and Gary T. Leavens: *A Simple and Practical Approach to Unit Testing: The JML and JUnit Way*, in ECOOP 2002 (Proceedings of European Conference on Object-Oriented Programming, Malaga, 2002), ed. Boris Magnusson, Lecture Notes in Computer Science 2374, Springer Verlag , 2002, pages 231-255.
8. Yoonsik Cheon and Gary T. Leavens: *The JML and JUnit Way of Unit Testing and its Implementation*, Technical Report 04-02, Computer Science Department, Iowa State University, at archives.cs.iastate.edu/documents/disk0/00/00/03/27/00000327-00/TR.pdf.
9. Ilinca Ciupa, Andreas Leitner, Manuel Oriol and Bertrand Meyer: *Object Distance and Its Application to Adaptive Random Testing of Object-Oriented Programs*, in Proc. of First International Workshop on Random Testing (RT 2006), Portland, Maine, USA, July 2006.
10. 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.
11. D. Detlefs, G. Nelson, and J. B. Saxe: *Simplify: A theorem prover for program checking*, Technical Report HPL-2003-148, HP Labs, 2003, available at research.compaq.com/SRC/esc/Simplify.html.
12. EiffelStudio open-source development site at eiffelsoftware.origo.ethz.ch.
13. JUnit pages at www.junit.org/index.htm.
14. Andreas Leitner, Mark Howard, Bertrand Meyer and Ilinca Ciupa: *Reconciling manual and automated testing: the TestApp experience*, to appear in HICSS 2007 (Hawaii International Conference on System Sciences), Hawaii, January 2007.
15. Lisa Liu, Andreas Leitner and Jeff Offutt: *Using Contracts to Automate Forward Class Testing*, submitted for publication.
16. Lisa Liu, Bertrand Meyer and Bernd Schoeller: *Using Contracts and Boolean Queries to Improve the Quality of Automatic Test Generation*, to appear in Proceedings of TAP (Tests And Proofs), Zurich, February 2007, Lecture Notes in Computer Science, Springer Verlag, 2007.
17. Bertrand Meyer: *Applying "Design by Contract"*, in Computer (IEEE), 25, 10, October 1992, pages 40-51.
18. Bertrand Meyer: *Eiffel: The Language*, revised printing, Prentice Hall, 1991.
19. Bertrand Meyer: *Reusable Software: The Base Object-Oriented Libraries*, Prentice Hall, 1994.
20. Bertrand Meyer: *Object-Oriented Software Construction*, 2nd Edition, Prentice Hall, 1997.
21. Bertrand Meyer: *Attached Types and their Application to Three Open Problems of Object-Oriented Programming*, in ECOOP 2005 (Proceedings of European Conference on Object-Oriented Programming, Edinburgh, 25-29 July 2005), ed. Andrew Black, Lecture Notes in Computer Science 3586, Springer Verlag, 2005, pages 1-32.
22. NIST (National Institute of Standards and Technology): *The Economic Impacts of Inadequate Infrastructure for Software Testing*, Report 7007.011, available at www.nist.gov/director/prog-ofc/report02-3.pdf.
23. SICStus Prolog User's Manual, http://www.sics.se/sicstus/docs/latest/pdf/sicstus.pdf.
24. Dennis K. Peters and David L. Parnas: *Using Test Oracles Generated from Program Documentation*, in IEEE Transactions on Software Engineering, vol. 24, no. 3, March 1998, pages 161-173.
25. Andreas Zeller: *Why Programs Fail: A Guide to Systematic Debugging*, Morgan-Kaufmann, 2005.