Reconciling Manual and Automated Testing: the AutoTest Experience

Andreas Leitner, Ilinca Ciupa, Bertrand Meyer
Chair of Software Engineering, Department of Computer Science, ETH Zurich
CH-8092 Zürich
{Firstname.Lastname}@inf.ethz.ch

Mark Howard
AXA Rosenberg Investment Management LLC
Orinda, California 94563
mhoward@axarosenberg.com

Abstract

Software can be tested either manually or automatically. The two approaches are complementary: automated testing can perform a large number of tests in little time, whereas manual testing uses the knowledge of the testing engineer to target testing to the parts of the system that are assumed to be more error-prone.

Despite this complementarity, tools for manual and automatic testing are usually different, leading to decreased productivity and reliability of the testing process.

AutoTest is a testing tool that provides a "best of both worlds" strategy: it integrates developers' test cases into an automated process of systematic contract-driven testing. This allows it to combine the benefits of both approaches while keeping a simple interface, and to treat the two types of tests in a unified fashion: evaluation of results is the same, coverage measures are added up, and both types of tests can be saved in the same format.

1. Introduction

A testing strategy can be manual or automated. With a manual strategy, the more traditional approach, testers prepare test suites that they think will best exercise the program. An automated testing strategy tries to remove the tediousness of the process by relying on a software tool that generates test cases from the program's specification (black box) or its actual text (white box).

Automated and manual strategies are often thought of as completely distinct, and usually supported by different tools.

In fact they are complementary, since each has weaknesses that the other addresses. Manual tests are good for capturing deep or special cases, which automated tests might not guess; but they cannot yield extensive coverage because of the sheer number of test cases this requires. Automated tests are good at breadth but much less at depth.

The AutoTest tool is, at its core, an automated testing framework that produces systematic tests from contracts [1] of object-oriented programs. AutoTest is the successor of a series of tools we built, that all implement contract-based testing [2-4]. We have developed the latest version, which is described in this paper, in a way so that many aspects (e.g. the actual testing strategy) can be plugged in as needed. The complete source of AutoTest can be downloaded from the tool homepage [5]. "Automatic", when applied to AutoTest, should be understood in the full "push-button" sense of the term: all a user must specify is the set of classes that he wants to test; then AutoTest will test these classes automatically, without requiring any intervention of the user, such as preparing test cases.

But AutoTest also supports manual testing, in particular the inclusion of any extra test that a developer finds relevant for any reason, supporting the important rule that any test that has uncovered a bug – whether the test case was generated through automated

or manual means – should remain part of the regression test database of the project.

One of the novelties of AutoTest is the close integration of its manual and automated testing parts. A manual test is represented by a class, distinguished only by its inheritance from a specific library class. The automated testing framework is able to detect such classes and then runs them first, reserving any remaining time for the generation and execution of as many automated test cases as timing constraints will permit. In incremental mode, the framework can be used to run test cases, manual or automated, that pertain only to parts of the software that have been modified since the last run.

Also contributing to the close integration of the two approaches is the use of a single mechanism — contracts — as test oracle.

AutoTest is a released tool that has already served to uncover bugs in production libraries and systems. AutoTest currently targets Eiffel code, because Eiffel is one of the few languages that integrate executable specification. A large base of contracted code for testing our tool is available (which is not the case for the recent additions of contracts to such languages as Java and C#). Selecting relevant test cases from a test scope is applicable to object oriented languages in general. The integration with automated contract-based testing is only feasible for languages that support contracts either natively or via extensions such as JML [6].

The main contribution of this paper lies in the mechanisms that we provide to integrate the manual and automated testing strategies. This integration has the following advantages:

- The overall testing process benefits from the strengths of both manual and automated testing;
- Support for regression testing: any automatically generated tests that uncover bugs can be saved in the same format as manual tests and stored in a regression testing database;
- The measures of coverage (code, dataflow, specification) will be computed for the manual and automated tests as a whole;
- The interface is kept consistent and simple: AutoTest only requires a user to specify the classes that he wants to test. Manual unit test cases that are not relevant for any of those classes are automatically filtered out.

The paper is organized as follows: the next section contains a general presentation of the manual and automated testing strategies and motivates why they should be combined. Section 3 describes the architecture of AutoTest. Section 4 describes how the two strategies can be unified and how this was accomplished in AutoTest. Section 5 gives an evaluation of our approach. Section 6 provides related work and Section 7 presents ideas for future work and draws conclusions.

2. Testing strategies

In this section we introduce the two strategies unified by our tool, manual testing and automated testing, then an analysis of the advantages and disadvantages of each, and the rationale for integrating them.

2.1 Manual testing

Manual unit testing has established itself as an integral part in modern software development. It only reached a respectable state with the introduction of adequate tool support (the xUnit family of tools, e.g. jUnit for Java, sUnit for Smalltalk, pyUnit for Python, and Gobo Eiffel Test for Eiffel). Such frameworks are typically small but they provide significant practical benefits.

Manual unit testing frameworks automate test case execution. The test cases themselves (including input data generation and test result verification) need to be created by hand.

To add a new test case, the user must create a new class that inherits from an abstract test case class (often called `TEST_CASE` or equivalent). Typically the goal of a test case class is to exercise one class from the system under test. The developer can put as many testing routines[1] into this new test case class as desired. The goal of such a routine is to test a certain scenario. With most frameworks, the names of these routines are required to start with *test*.

Testing frameworks use reflection to find the set of test case classes: membership in this set is determined by checking if a class inherits from the abstract test case class. During test case execution, all routines that start with *test* are invoked in sequence.

The test routines themselves have two responsibilities:

- They trigger the execution of the system, by creating objects and invoking routines on them.
- They verify whether the output and status of the system under test after the execution is correct. This is done by calling an *assert* routine in the test routines. This routine requires a boolean argument

1 Routines are also called “methods”.

that signals whether an assumption held or was violated.

Over time, software projects typically acquire a large number of manual unit tests. The execution of a whole test suite is usually time-consuming. Testing frameworks hence offer the ability to run test cases in isolation or run just a subset of the available test cases for incremental development.

2.2 Automated testing

In contrast to manual testing, automated testing automates not only test case execution, but also test case generation and test result verification. A fully automated testing system is able to test software as-is, without any user intervention.

Contract-based testing achieves full automation through the use of contracts as oracle. Contracts are executable preconditions, postconditions, and invariants embedded in the software text [1].

Contracts are part of Eiffel [7] and Spec# [8], and are available as add-ons for Java using for example JML [6], iContract [9], or OCL [10].

Preconditions serve to filter out invalid input; postconditions serve to detect failures in the system under test.

Recently, contract-based testing [11, 12] has been the subject of much research, taking advantage of such techniques as: constraint solving [13], state pruning by monitoring read accesses [14], integration with static verifiers [15], evolutionary test case generation [16, 17], and synthesis through a planning system [18].

When using fully automated testing systems, users do not choose what test cases to execute. Instead, they typically provide a *test scope*: a set of classes that should be tested; in other words, they only have to specify what to test and not how.

AutoTest contains a fully automated contract-based testing strategy that creates test cases using random input data. This strategy receives the test scope as input and creates test cases for each routine of every class in the scope.

2.3 Manual vs. automated testing

Automated testing requires less effort on the developer's side, but it cannot fully replace manual unit testing: developers are better at setting up complex input data and at finding *interesting* test cases (where "interesting" means "more likely to uncover a bug").

Nevertheless, automated testing retains strong advantages. A developer might misunderstand the real input domain of a routine. For example, he might not think of certain borderline cases, and may write an implementation that does not work as he would expect in those cases. Since his understanding of the input will be just as flawed when he tests the system, he is unlikely to write tests that exercise the erroneous cases; the bug will not be uncovered. An automated testing system does not try to guess the intended semantics of a system, hence it does not exhibit this weakness.

In this paper we present a way to integrate manual and automated testing that retains the advantages of both approaches:

- By testing both automatically and manually we uncover bugs that one strategy alone might miss.
- We retain a simple tool interface: the user only specifies what classes to test, not how. In particular this means that we have to automatically select those manual test cases from the complete set of manual test cases that are relevant with regards to the current testing goal.
- A single set of results is produced. Coverage data reflects coverage achieved by both manual and automated testing.
- Generated test cases that reveal a bug are saved in the same format as manual test cases. They can hence be easily added to the manual test suite.

3. AutoTest architecture

AutoTest is a framework for fully automated software testing. It allows for arbitrary testing strategies to be plugged in and is not hard coded to a certain testing strategy. The pluggable testing strategy is only concerned with determining exactly how and with what inputs the system under test should be invoked. The actual execution is a task of the framework.

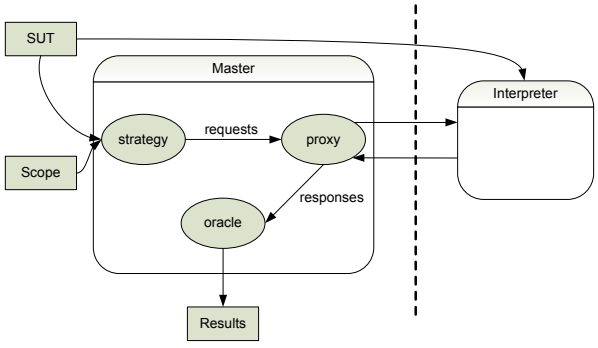

Figure 1: AutoTest architecture

As shown in Figure 1, the main parts of AutoTest are:

- **Testing strategy**: pluggable component that determines what instructions should be executed on the system under test. A testing strategy receives the AST of the system under test and the test scope, i.e., the set of classes that should be tested. It then uses this information to synthesize test cases which it gives to the proxy. The strategy provided by default creates test cases that use random input to exercise the classes under test. In addition to the default strategy, we have developed an experimental *forward class testing* strategy [19], an experimental planner-based strategy and a strategy that handles manually written unit tests. The last strategy and its integration with automated testing are the main contribution of this paper and will be described in Section 4.
- **Interpreter:** Executes instructions on the system under test. The interpreter lives in a separate process to increase robustness. Typical instructions for the interpreter are: *create object*, *invoke routine*, and *assign result*.
- **Proxy:** Component that handles inter-process communication. The proxy receives execution requests from the strategy and forwards them to the interpreter. The execution results are then sent to the oracle.
- **Oracle:** The oracle is based on the idea of contract-based testing as further described in Section 4.4. It receives execution results and determines the outcome of the execution. The oracle then writes the testing results in the form of HTML documents to the hard disk.

4. Integrating manual and automated testing

The two testing strategies described above can have several incarnations. In this section, we describe in detail their particular implementations in the AutoTest tool and how we obtain their seamless integration.

4.1 Test scope selection

The first step in any testing strategy is to decide on the test scope. AutoTest is specifically designed for unit testing, where the scope is usually one or several classes of the system under test. AutoTest supports the incremental testing of software as it is being developed. Our tool requires no user intervention. A possible scenario is for software to be tested in the background, receiving the test scope from the IDE in the form of a list of recently changed classes, or to be tested as soon as the code is committed to a version control system. In both cases testing can be limited to the part of the software that has changed.

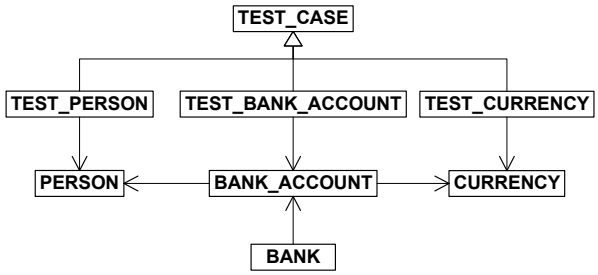

Figure 2: Example class diagram

Consider the class diagram depicted in Figure 2, which we will use as a running example. The classes *PERSON*, *CURRENCY*, *BANK* and *BANK_ACCOUNT* make up the system under test. The classes *TEST_BANK_ACCOUNT*, *TEST_CURRENCY* and *TEST_PERSON* form the corresponding test suite. They are all descendants of class *TEST_CASE*. Class *TEST_BANK_ACCOUNT* is a client of *BANK_ACCOUNT*, class *TEST_PERSON* is a client of *PERSON*, and class *TEST_CURRENCY* is a client of *CURRENCY*.

AutoTest only requires an input file describing the system under test (similar to a make or ant file), and the list of classes that must be tested (the test scope). The following invocation instructs AutoTest to test the classes `PERSON` and `CURRENCY`:

```
auto_test system.ace PERSON CURRENCY
```

The file `system.ace` is the standard build file for the system under test. Note that no part of the system under test (including the build file) has to be modified for testing.

Once AutoTest knows the test scope, it will perform all other steps necessary for test case generation, execution, and result evaluation in a completely automated manner.

4.2 The intuition behind the selection of relevant test cases

Since AutoTest knows that the user is interested in testing classes *PERSON* and *CURRENCY*, it will not only test them automatically, but also detect the manual test cases that apply to them. The relevant test cases are detected through the two fundamental relations of object-oriented programming as follows:

- Inheritance is used to mark which classes are manual unit tests
- Client (or association) is used to determine which manual unit tests apply to the classes in the test scope.

The inheritance relation is commonly used in manual unit testing frameworks. Our use of the client-relationship to reduce the number of relevant manual unit tests is novel. In an incremental development setting where development is constantly interleaved with testing it is important that test execution be fast. Since the execution of the whole test suite might take too long, this reduction is a great improvement.

We detect the complete set of manual test cases (in the example `TEST_BANK_ACCOUNT`, `TEST_CURRENCY` and `TEST_PERSON`) using the inheritance relation. To reduce the number of test cases we only select those test cases that are clients of a class in the scope: `TEST_CURRENCY` and `TEST_PERSON`.

In most unit test suites a given test case class is dedicated to testing one class from the system under test. Any such test case class is obviously a direct client of its class under test; we will call it *immediately relevant*. The notion of immediate relevance is naturally extended by including tests that are directly or indirectly clients of a class under test. This extension, called *recursive relevance*, can capture subtle interactions between classes that may not be caught by examining only immediate clients. In our example the set of recursively relevant test cases would also include class `TEST_BANK_ACCOUNT` because it is a client of class `BANK_ACCOUNT`, which is a client of class `CURRENCY`, which is in the test scope. Test case selection based on recursive relevance ensures that all test cases that may exercise a class from the scope are selected. In contrast, a selection made by a human may not include all test cases relevant to a change that has been made.

We now formalize these two notions of relevance and our strategy for test case selection.

4.3 Formal description of the selection of relevant test cases

This section describes the notion of immediately relevant and recursively relevant test cases and shows how they can be implemented efficiently.

Let C be the set of classes making up the system under test and $tc \in C$ be the abstract test case class from which every manual test case class inherits. We need notations for the inheritance and client-of relations:

Definition 1 (inheritance). Let ***inh*** be the inheritance relation such that for any two classes $a,b \in C$, a ***inh*** b iff a is a direct descendant (subclass) of b.

The set of manual test cases T is defined as
$T := \{t \in C \mid t \; \boldsymbol{inh}^+ \; tc\}$

where $\boldsymbol{inh}^+$ denotes transitive closure of the relation ***inh***. Hence, T is the set of all manual test cases from the system under test.

Definition 2 (client-of). Let ***co*** be the client relation such that for any two classes $a,b \in C$, a ***co*** b iff a is a *direct client* of b. Furthermore, a is an *indirect client* of b iff $a \; \boldsymbol{co}^* \; b$ where $\boldsymbol{co}^*$ denotes the reflexive, transitive closure of the relation ***co.***

We denote the inverse of the client-of relation, called supplier-of, by ***so***.

Let $S \subseteq C$ be the given test scope. The sets of immediately and recursively relevant test cases are defined as:

- $T_{immediate} := \{t \in T \mid \exists \; s \in S: \; t \; \boldsymbol{co} \; s\}$
- $T_{recursive} := \{t \in T \mid \exists \; s \in S: \; t \; \boldsymbol{co}^* \; s\}$

Given a class s in the test scope, computing all test cases that are clients of s requires the traversal of all classes in C. However, to compute the suppliers of a test case t that are in the test scope, it is sufficient to traverse t and all its direct and indirect parents. Consequently, in our implementation we use the supplier-of relation to compute $T_{immediate}$.

The set $T_{immediate}$ can be calculated efficiently using the supplier-of relation. To obtain $T_{recursive}$ we need to compute the transitive closure of the suppliers of all test cases, which involves finding all indirect suppliers of these test cases. In large and highly interconnected systems this can lead to a significant overhead.

To improve performance when computing $T_{recursive}$, we use a *reasonable* approximation of the relation $\boldsymbol{so}^*$. We define this approximation based on some observations about constants:

- Constants have types that are defined in a core library of the language.
- Core libraries are self-contained, meaning they do not depend on classes external to the library.
- Core libraries do not need testing except from the compiler provider.

Thus, when computing so^* it is sufficient to consider classes outside of the core library.

Let so_{nc} be the relation between classes such that for $a, b \in C$, $a\ so_{nc}\ b$ iff a is a supplier of b that does not occur in a core library. Given a class b, the direct and indirect suppliers of b that do not occur in a core library are the classes a such that $a\ so^*_{nc}\ b$.

The computation of the relation so^*_{nc} is also expensive, since it depends on the semantics of every class involved. Note that this is also true for the relations introduced above. There exists an over-approximation of so^*_{nc}, based on purely lexical analysis, which makes it cheap to compute:

- Given a class a mark all type names occurring in the text of a as belonging to the result set.
- Process all type names occurring in the text of a recursively.

This over-approximation can be used to compute $T_{recursive}$ more efficiently.

4.3 Execution

AutoTest allows to specify the duration of testing: users can provide a number of minutes (or use the default of 10) representing how long they would like AutoTest to test their classes. When the time has elapsed, AutoTest stops testing and displays the results.

AutoTest executes relevant manual test cases first. The remaining time is used for automatic testing of classes in the scope. The reason for this scheduling is that the existence of manual tests indicates that the user is most interested in those tests: hence we execute them in the beginning. Any time that remains may be used for generating and running as many automated tests as possible.

4.4 Oracle

The oracle is notoriously difficult to automate: AutoTest uses the contracts embedded in the software for this purpose. These contracts come in the form of routine pre- and postconditions, class invariants, loop variants and invariants, and `check`[2] instructions.

Contracts contain the specification of the software and can be monitored at runtime. Except for the case when a generated test case directly violates the precondition of the routine under test (and hence this is an invalid test case), any contract violation signals a mismatch between the implementation and the specification. Hence, whenever it encounters a contract violation (with the exception of the case mentioned above), AutoTest signals a bug and the test case that triggered the contract violation is accordingly marked as failed. If all contracts are fulfilled during the execution of a test case, the test case is marked as pass. The same result verification process can be applied in the case of manual tests. The user need not write any result checking code; the contracts are the oracle, hence, just as above, any contract violation will cause the manual test case to fail and lack of contract violations will mean that the manual test case has passed. To add supplementary checks on the results, it suffices (as shown in the example below) to embed them in a `check` instruction, a regular contract perfectly integrated in the contract-based oracle system. The following example shows a manual test case that uses such a `check` instruction.

```
class TEST_BANK_ACCOUNT
inherit TEST_CASE
feature -- Tests
  test_creation is
      -- Check that bank accounts are created
      -- with an initial balance of 0.
    local
      b: BANK_ACCOUNT
      p: PERSON
    do
      create p.make ("John Doe", 30)
      create b.make (p)
      check
          bank_account_empty: b.balance = 0
      end
    end
```

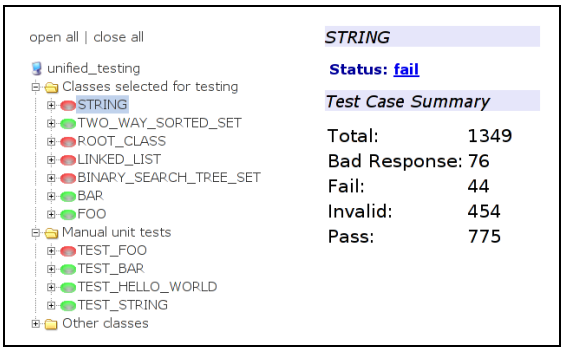

Figure 3: Screenshot of AutoTest results

Unified results. A great advantage of the integration of automated and manual tests is the unification of their results. After all manual and automatically generated test cases have been run, AutoTest displays their results in the same setting, as shown in Figure 3. This is very convenient for the user of the tool, because for him it is not important what kind of test case uncovered the bug, but only that the

2 In C++ and Java, the equivalent of the `check` instruction is the `assert` mechanism.

bug was found (or that all test cases passed). In addition, AutoTest provides a bug-reproducing witness.

A major drawback of performing manual and automated tests through separate tools is that the coverage measures will be computed separately too. Obviously, adding the two resulting measures will not give the overall coverage of the manual and automated tests. This problem disappears in AutoTest: as test cases are executed in the same framework, their coverage is also computed together. Hence we will have only one measure, representing the coverage achieved by manual and automated tests together.

Another advantage of the integration of manual and automated tests is that automatically generated tests that fail can be saved in the format of manual tests (classes inheriting from *TEST_CASE*) and stored in a regression testing database, together with any failing manual tests.

5. Evaluation

As noted in Section 1, automatically generated and manually written tests have different strengths. An automatic strategy can generate and run a much greater number of test cases than a human could run in the same time. Table 1 shows some results obtained by the automatic strategy when testing some widely used Eiffel libraries and applications.

Table 1: Results obtained by the automatic strategy when testing some Eiffel libraries and applications

Library/ Application	Failed tests / Total tests	Buggy routines / total tested
EiffelBase (base)	1513/39615	127/1984
base.structures	1143/21242	88/1400
Gobo math	16/1539	9/144
DoctorC	1283/8972	15/33

For illustration purposes, we provide an example of a bug that was found by AutoTest in the EiffelBase library. The bug is located in routine `has` of generic class `BOUNDED_STACK [G]`, which checks if the argument that it receives is an element of the bounded stack. The following test case generated by AutoTest found a bug in this routine:

```
create {BOUNDED_STACK [ANY]} v_75.make (8)
v_76 := Void
v_77 := v_75.has (v_76)
```

The first instruction creates an empty bounded stack with at most 8 elements. AutoTest signals a bug because the postcondition of `has` is violated:

```
not_found_in_empty: Result implies not is_empty
```

Since no element has been pushed on the stack no element should be contained in it. Nevertheless **`Result`** is set to true and the implementation of routine `has` is wrong. Upon creation the stack already allocates space for all 8 elements that it is able to store. The space of these elements is by default initialized with **`Void`**. The bug in `has` is that it traverses those empty cells even though it should know that they are not in use yet.

Although, as shown above, AutoTest can find many bugs even in production-quality code, manually written test cases benefit from the knowledge that the tester has about the system under test, and hence can uncover bugs that an automatic strategy might not find given limited time. We provide two examples of such bugs here. They are both located in the EiffelBase library mentioned in Table 1.

Class `STRING` from cluster `base.kernel` contains a routine `is_integer` returning a boolean result which indicates whether the string represents an integer. It does so by checking that each character in the string is a digit, except for the first one, which can also be a plus or a minus sign. AutoTest did not find any bug in this routine when testing it automatically, but the following manually written test case did:

```
create {STRING} s.make_filled ('2', 100)
check not s.is_integer end
```

This test case creates a string that is 100 characters long, each character being '2'. Function `is_integer` returns true, but the number that the string represents is much greater than the maximum integer. Therefore, when this routine is called in the precondition of procedure `to_integer` (for example) it will return true for the given string, then `to_integer` will try to convert it to an integer number and will fail.

Another bug that was found only by a manual test case and not by AutoTest appears in routine `occurrences` of generic class `BOUNDED_STACK [G]`. This routine should return the number of times an element occurs in the bounded stack. The manual test case that uncovered the bug is:

```
create {BOUNDED_STACK [ANY]} bs.make (9)
check bs.occurrences (Void) = 0 end
```

The first instruction creates an empty bounded stack with an initial capacity of 9 elements. The second tries to compute how many Void elements there are in this bounded stack, and `occurrences` wrongly returns 9, because the empty slots in the structure are counted as Void elements. The automatic strategy also tried to call `occurrences` with a void argument, but the postcondition of this routine only states that the result should be greater than or equal to 0, so the routine passes its automatically generated tests.

These two examples illustrate two issues that the automatic strategy has:

- In the first case, generation of input values;
- In the second case, incomplete oracle (because of under-specified contracts).

Another problem that the current random strategy for generating input values has is that it cannot fulfill strong preconditions in a limited time. Routines with many arguments and preconditions on each of them are particularly problematic. Naturally, manually written test cases do not suffer from the same drawback, and are thus necessary for testing these routines that the automatic approach leaves untested.

6. Related work

Support for manual unit testing has been greatly improved with the advent of the xunit family of tools. Some of its members are JUnit [20] for Java, SUnit [21] for Smalltalk, or Gobo Eiffel Test [22] for Eiffel. The idea of using the client relation to detect relevant manual test cases was first implemented in Rose Studio, an in-house IDE developed at AXA Rosenberg. Rose Studio tightly integrates with manual unit testing but does not cover automated testing.

All these tools function by the same principle: they provide an automated test driver, but the user still has to write the test case to be executed: input values, code for calling the routines under test, and code for comparing the expected result to the actual one. Despite the amount of manual work involved, automatic execution brings a big improvement over fully manual testing, so these tools have become very popular and are still the de-facto standard for manual unit testing. They allow testers to exercise the inputs and parts of the code that the testers think are most likely to expose bugs. Because of the amount of work involved, the size of a manual test suite cannot compare to the size of an automatically generated one, so testers have to pay particular attention to how they invest their effort and always try the combinations of inputs that they think will bring the most information about the system under test.

The research community has invested a lot of effort during recent years into developing tools for completely automatic testing. AutoTest [23] is part of this effort. The Korat [14] tool can also perform automatic testing of contracted code; it provides full coverage of a bounded subset of the input domain by creating all non-isomorphic inputs that satisfy a boolean predicate up to a given size. This strategy for generating input values is especially useful when dealing with predicates on the structure of the input, but is not as efficient for arithmetic expressions. DART [24] is another tool that can perform fully automatic testing; however, it is designed to work at the level of the whole, integrated system, while AutoTest specifically targets unit testing.

A tool very similar in concept to AutoTest is Jartege [25] (Java Random Test Generator), which performs automatic testing of Java programs equipped with JML [6] contracts. This tool also uses random generation of test inputs, and for routines with strong preconditions the user must write generators for the parameters.

Another approach to input value generation relies on symbolically executing the routine under test to find inputs that cover a certain path [26, 27]. Although code coverage is an important measure of test quality, achieving full path coverage in no way guarantees that all faults are detected in the tested software.

Other strategies go into the direction of trying to improve the performance of random testing, while keeping its simplicity. Adaptive Random Testing (ART) [28] go into this direction: they provide ways of generating and selecting inputs based on a random strategy, but use a notion of "distance" between the inputs in order to run tests with values that are "far" away from each other in the input domain. This strategy improves over the efficiency of random testing for non-point types of failure patterns. The distribution of failure-causing values in the input domain is important for the efficiency of ART; this strategy is based on the idea that failure-causing inputs are clustered into regions.

Despite the fact that all tools cited here provide great advantages by the degree of automation that they offer, the implemented automatic testing strategies cannot make up for the lack of specialized knowledge that a human tester has. When manual and automatic testing are integrated, each of them can benefit from the advantages offered by the other.

Parasoft's Jtest tool [29] automatically generates and runs unit tests on Java classes, and tests code for compliance with development rules. Contracts can be

added to Java code by using the same company's Jcontract tool. Developers can also add their own unit tests. However, Jtest does not use the transitive closure of the client relation for manually written unit test cases to determine which test cases are relevant for which class, as is the case in our work. Instead, when the user selects a certain class for testing, Jtest will only run the unit tests that were specifically designed for that particular class.

Agitator [30] is a commercial tool that integrates dynamic invariant discovery [31] with other recent ideas from the testing research community into an Eclipse extension. It uses several strategies of automatic testing to infer likely invariants that then can be promoted to real assertions by the developer. In contrast to our work it does not assume the presence of contracts. Agitator can include manual unit tests in its harness, but does not detect relevant test cases automatically.

Augmenting manual unit tests with automatically generated ones has also been investigated [32, 33]. In the former, operational abstractions are generated from the execution of manual tests and then any automatic tests that violate these abstractions are candidates for inclusion in the test suite. In the latter, an operational model of the behavior of the classes under test is also inferred from a set of correct executions. Automatic test cases whose executions produce models different from the inferred ones are considered likely to identify faults. Both these techniques use dynamic discovery of program properties, against which program behavior is checked afterwards. Our technique assumes the presence of contracts in code, and these contracts are not modified in any way during testing.

7. Future work and conclusions

We have shown that the seamless integration between manual and fully automated tests has several advantages, such as combining the strengths of both approaches, the support it provides for regression testing, and the unification of coverage data. We have shown how the interfaces to an automated testing strategy and a manual one can be unified: the user provides the set of classes to be tested; these classes directly drive the automated strategy and are input for a selection process of manual unit tests.

The already existing features of AutoTest open the way towards *continuous testing* (or *testing in the background*), based on the integration of AutoTest in an integrated development environment (IDE). Because the tool can test software completely autonomously, it can be constantly run in the background while the developer is writing the code. As soon as a bug is found, the corresponding routine can be marked (in an un-intrusive way), to draw the developer's attention that the current implementation of the routine is not correct with respect to its specification. A further advantage of integrating AutoTest into an IDE is that the test scope selection is automated too, since the IDE can keep track of classes that have changed.

Moreover, AutoTest can be used in a test-driven development process [34]: the manual tests (which are written first in such a process) will be continuously executed by AutoTest in the background; as long as they fail, the corresponding routines will be highlighted (similar to syntax and type check errors), and these warning signs will only disappear once the test cases pass.

8. Acknowledgements

We thank Vijay D'silva, Manuel Oriol, Bernd Schoeller, Lisa Liu, and Piotr Nienaltowski for their valuable feedback, and Eric Bezault both for fruitful discussions and the Gobo Eiffel framework.

9. References

[1] B. Meyer, *Object-Oriented Software Construction*: Prentice Hall PTR, 1997.

[2] I. Ciupa, "Test Studio: An environment for automatic test generation based on Design by Contract," ETH Zurich, 2004.

[3] I. Ciupa and A. Leitner., "Automatic Testing Based on Design by Contract," in *Proceedings of Net.ObjectDays 2005*, 2005, pp. 545-557.

[4] A. Leitner, "Strategies to Automatically Test Eiffel Programs," Graz University of Technology, 2004.

[5] A. Leitner and I. Ciupa, "AutoTest," http://se.ethz.ch/people/leitner/auto_test, 2006.

[6] E. P. G. T. Leavens, C. Clifton, Y. Cheon, C. Ruby, D. Cok, P. Müller, J. Kiniry, *JML Reference Manual*: (draft), 2005.

[7] "Eiffel Analysis, Design and Programming Language," *ECMA Standard*, vol. 367, 2005.

[8] M. Barnett, K. R. M. Leino, and W. Schulte, "The Spec# programming system: An overview.," in *CASSIS 2004: Construction and Analysis of Safe, Secure and Interoperable Smart devices*: Springer, 2004.

[9] R. Kramer, "iContract - The Java(tm) Design by Contract(tm) Tool," in *TOOLS '98: Proceedings of the Technology of Object-Oriented Languages and Systems*. Washington, DC, USA: IEEE Computer Society, 1998, pp. 295.

[10] M. Richters and M. Gogolla, "On Formalizing the UML Object Constraint Language OCL," in *Proc.*

17th International Conference on Conceptual Modeling (ER), vol. 1507, M. L. Lee, Ed.: Springer-Verlag, 1998, pp. 449-464.

[11] B. K. Aichernig, "Contract-Based Testing," *Lecture Notes in Computer Science*, vol. 2757, 2003.

[12] Y. Cheon and G. T. Leavens, "A Simple and Practical Approach to Unit Testing: The JML and JUnit Way," in *Proceedings of the 16th European Conference on Object-Oriented Programming*, *Proceedings of the 16th European Conference on Object-Oriented Programming*, 2002, pp. 231-255.

[13] D. Marinov and 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.

[14] C. Boyapati, S. Khurshid, and D. Marinov, "Korat: automated testing based on Java predicates," in *Proceedings of the 2002 International Symposium on Software Testing and Analysis (ISSTA)*. Rome: ACM Press, 2002.

[15] C. Csallner and Y. Smaragdakis, "Check 'n' crash: combining static checking and testing," in *ICSE '05: Proceedings of the 27th international conference on Software engineering*. St. Louis, MO, USA: ACM Press, 2005, pp. 422-431.

[16] Y. Cheon, M. Y. Kim, and A. Perumandla, "A Complete Automation of Unit Testing for Java Programs," in *Proceedings of the 2005 International Conference on Software Engineering Research and Practice (SERP '05)*, *Proceedings of the 2005 International Conference on Software Engineering Research and Practice (SERP '05)*. Las Vegas, 2005, pp. 290-295.

[17] P. Tonella, "Evolutionary testing of classes," in *International symposium on Software testing and analysis (ISSTA'04)*. Boston, Massachusetts, USA: ACM Press, 2004, pp. 119-128.

[18] A. E. Howe, A. v. Mayrhauser, and R. T. Mraz, "Test Case Generation as an AI Planning Problem," *Automated Software Engineering*, vol. 4, pp. 77-106, 1997.

[19] L. Liu, A. Leitner, and J. Offut, "Using Contracts to Automate Forward Class Testing," *submitted to Elsevier Science*, 2006.

[20] "JUnit," http://www.junit.org/, 2004.

[21] "SUnit," http://www.xprogramming.com/software.htm.

[22] E. Bezault, "Gobo Eiffel Project," http://www.gobosoft.com, 2003.

[23] I. Ciupa and A. Leitner, "Automatic Testing Based on Design by Contract," in *Proceedings of Net.ObjectDays 2005*, 2005, pp. 545-557.

[24] N. K. Patrice Godefroid, Koushik Sen, "DART: directed automated random testing," presented at PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation, 2005.

[25] C. Oriat, "Jartege: a Tool for Random Generation of Unit Tests for Java Classes," Centre National de la Recherche Scientifique, Institut National Polytechnique de Grenoble, Universit\u017de Joseph Fourier Grenoble I, 2004.

[26] W. Visser, C. S. Pasareanu, and S. Khurshid, "Test Input Generation with Java PathFinder," in *International Symposium on Software Testing and Analysis (ISSTA'04)*, 2004.

[27] T. Xie, D. Marinov, W. Schulte, and D. Notkin, "Symstra: A Framework for Generating Object-Oriented Unit Tests using Symbolic Execution," in *Proceedings of the 11th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 05)*. Edinburgh, UK, 2005, pp. 365-381.

[28] T. Y. Chen, H. Leung, and I. K. Mak, "Adaptive Random Testing," in *Advances in Computer Science - ASIAN 2004: Higher-Level Decision Making. 9th Asian Computing Science Conference. Proceedings*, M. J. Maher, Ed.: Springer-Verlag GmbH, 2004.

[29] Parasoft, "Jtest," http://www.parasoft.com, 2006.

[30] B. Marat, D. Roongko, and S. Alberto, "From daikon to agitator: lessons and challenges in building a commercial tool for developer testing," in *Proceedings of the 2006 international symposium on Software testing and analysis*. Portland, Maine, USA: ACM Press, 2006.

[31] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin, "Dynamically Discovering Likely Program Invariants to Support Program Evolution," in *International Conference on Software Engineering*, 1999, pp. 213-224.

[32] C. Pacheco and M. D. Ernst, "Eclat: Automatic generation and classification of test inputs," in *ECOOP 2005 -- Object-Oriented Programming, 19th European Conference*. Glasgow, Scotland, 2005.

[33] T. Xie and D. Notkin, "Tool-assisted unit test selection based on operational violations," in *18th IEEE International Conference on Automated Software Engineering*: IEEE Computer Society, 2003.

[34] K. Beck, *Test Driven Development: By Example*: Addison-Wesley Professional, 2002.

10. Copyright forms and reprint orders

Not needed for initial submission.