Automatic Testing Based on Design by Contract™

Ilinca Ciupa Andreas Leitner

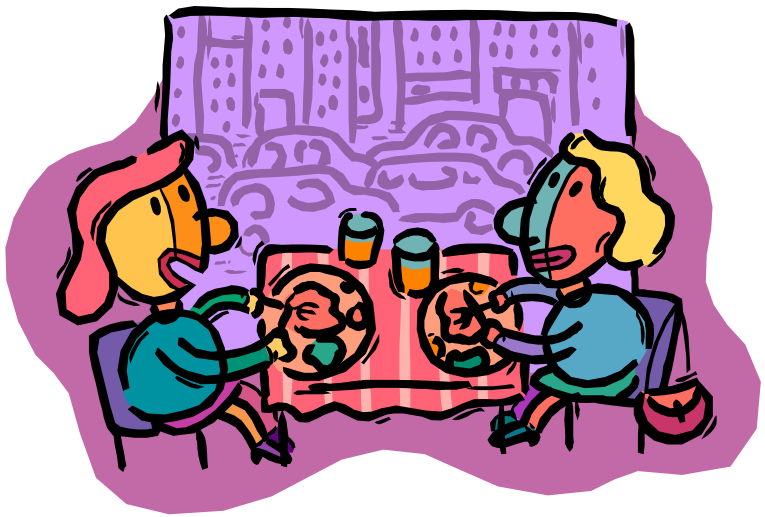
Chair of Software Engineering, ETH Zürich
(Swiss Federal Institute of Technology Zurich)

SOQUA Developer Track 2005

September 22, 2005

The important things in life

… taking a lunch break



Outline

1. Problems addressed and our solution: contract-based testing
2. AutoTest framework
   - Test process configuration
   - Input value generation
   - Test execution
   - Test results
3. Results
4. Conclusions

Problems with existing testing strategies

- They require too much work from the user
- Those that are automatic, work only in the most basic cases
- They are targeted at only one particular language
- Some of them require some kind of modification / addition to the tested software
- They are often tested by their authors only on small, artificial examples

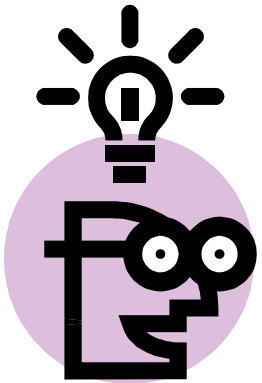
Our goal

We want to develop a testing strategy that:

- Finds *the real bugs* in real systems
- Is *fully automatic*
- Is applicable to *several* programming *languages*
- Tests software *as it is*
- Can be tested on *full-fledged, industrial-scale applications*

The idea

As long as we know what the software is supposed to do, we do not need any human intervention to test it.



Contracts contain the specification of the software!

⇩

Contract-based testing: using contracts as defined in Design by Contract™ as an automatic, freely-available testing oracle (and for input value generation)

Contract-based testing

When testing a certain method:

- We try to satisfy its *precondition* (so that we can execute it)
- We hope it will not fulfill its *postcondition* => BUG

```
class ARRAYED_LIST [G] ...
    put (v: like item) is
            -- Replace current item by `v'.
            -- (Synonym for `replace')
        require
            extendible: extendible
        do
            ...
        ensure
            item_inserted: is_inserted (v)
            same_count: count = old count
        end
```

precondition

body

postcondition

So practically...

- Our testing strategy: fully automatic, random
- Tool: AutoTest
- Programming language: Eiffel
- Found bugs:
  - Yes – many!
  - In production-quality libraries

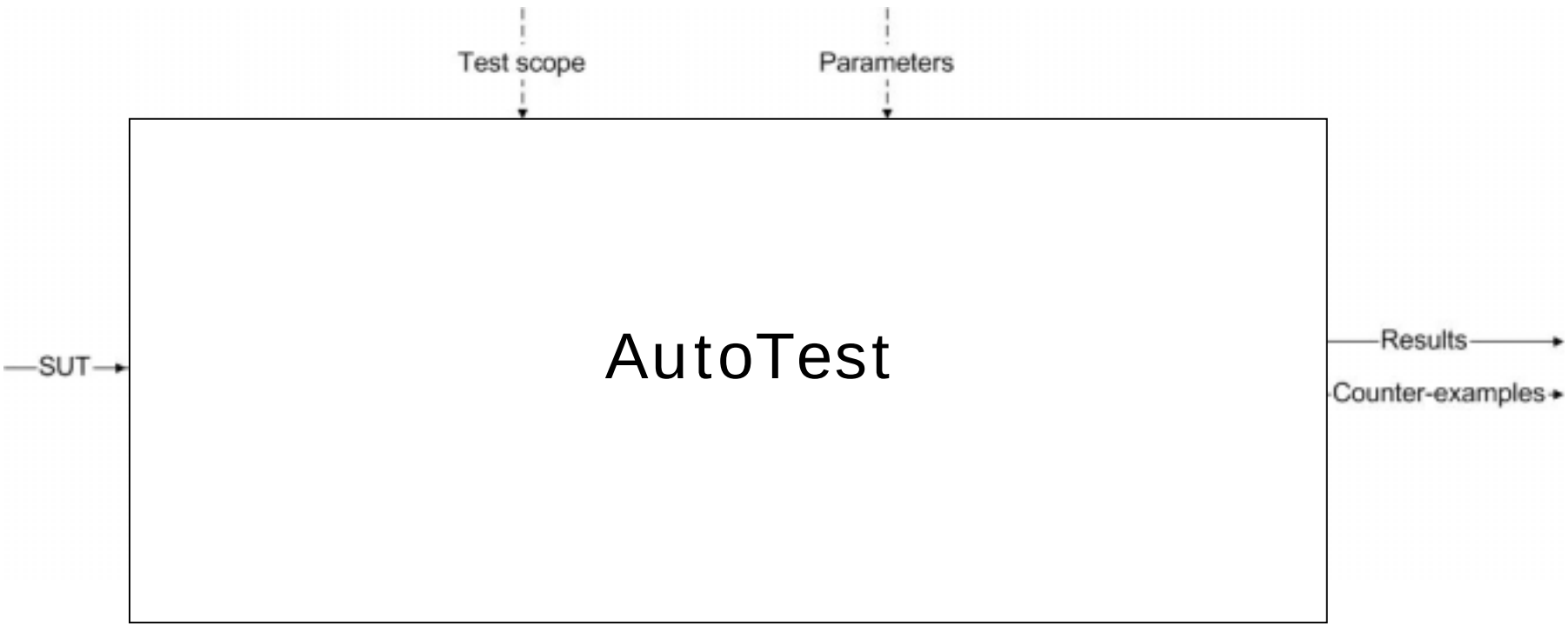
Outline

1. Problems addressed and our solution: contract-based testing
2. AutoTest framework
   - Test process configuration
   - Input value generation
   - Test execution
   - Test results
3. Results
4. Conclusions

Issues

- Generation of input data
- Tailorability of the testing process
- Interpretation and representation of results
- Fault isolation
- Failure-reproducing examples
- Estimation of the quality of the test suites

AutoTest


Test scope
Parameters
SUT
AutoTest
Results
Counter-examples

Test scope

- *Methods*
- For convenience, user can also specify scope at:
  - Class level – all its methods included
  - Package level – all its classes included

- Suppose user wants to test method **m** of class **A**.
  Issues:
  - Should we also test **m** in descendants of **A**?
  - If **A** is generic, which instantiations of the generic parameter/s do we use?

Configuration parameters

- Stress level:
  - Number of calls to tested methods
  - Testing in descendants
- Testing order
- Level of support for genericity
- Values used for primitive types
- Instances for non-primitive types

Input value generation

- Currently *random strategy*
  - Random object creation
  - Random object modification
- We keep a pool of objects that we enhance and diversify as testing proceeds
- Algorithm

  For each call to each method under test (MUT):

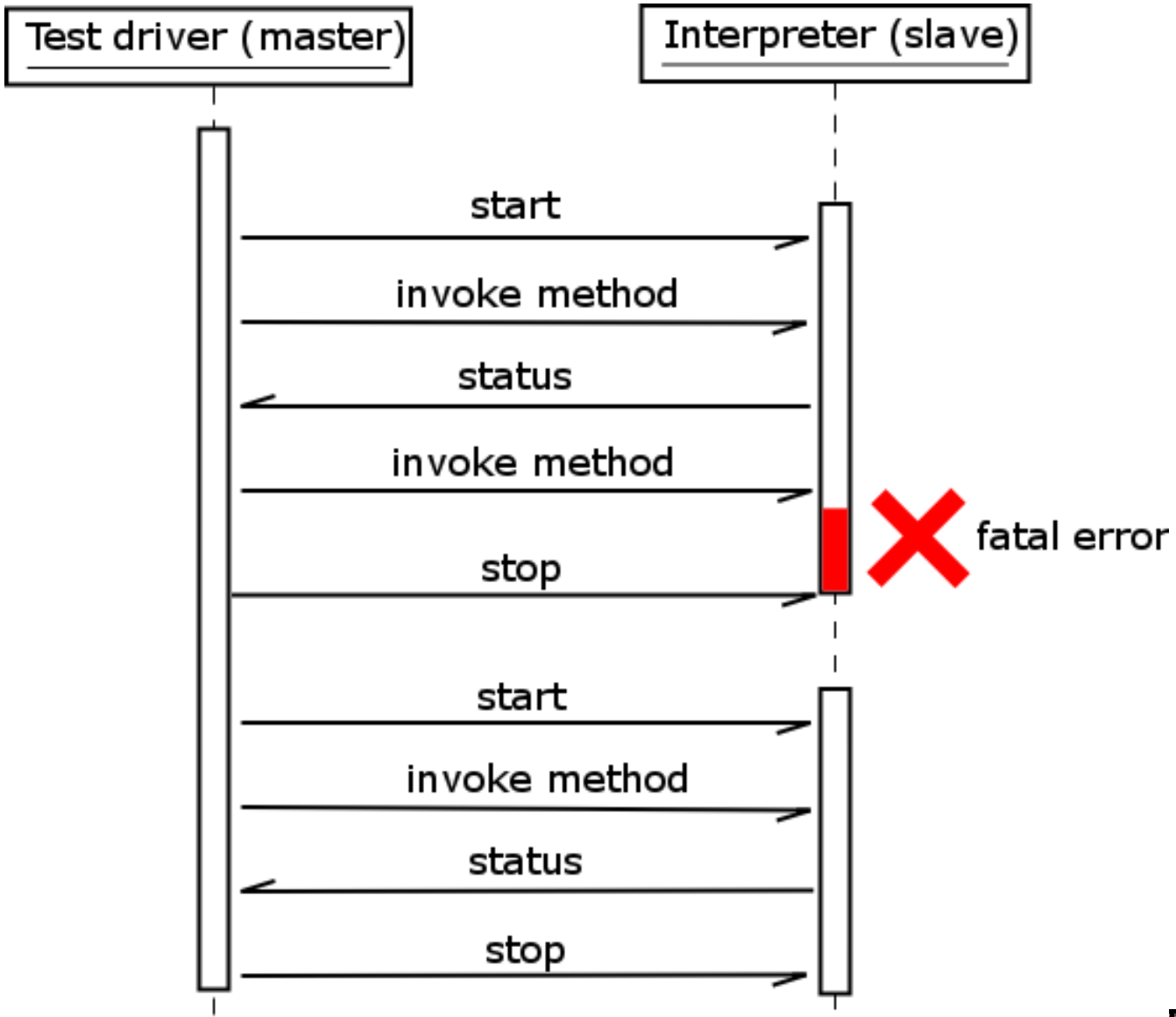
  1. Create objects needed for the next call
  2. Store these objects in the pool
  3. Modify an arbitrary object from the pool
  4. Call MUT with objects randomly selected from the pool

Preconditions

- Generated test cases (TCs) that don't satisfy the tested method's precondition are useless.
- The *stronger the preconditions*, the harder it is for the random strategy to satisfy them. (Plus, random strategy has no notion of coverage.)
- So how do we automatically generate valid TCs?

- Our solution: *planning* augmented with learning

Execution of test code



Test results

- Evaluation straightforward due to the presence of contracts
- Classification:
  - Pass – all OK /
  - Invalid – MUT was not executed
  - Fail – contract violation => bug
  - Bad response – interpreter returned unexpected response

Outline

1. Problems addressed and our solution: contract-based testing
2. AutoTest framework
   - Test process configuration
   - Input value generation
   - Test execution
   - Test results
3. Results
4. Conclusions

Results

- Main advantage of our approach: we can test it on *existing large-scale applications and libraries*
- Some figures from testing EiffelBase (the most widely used Eiffel library, that has been in use for more than 10 years):
  - Class ARRAYED_LIST [G]: tests failed for *13* out of 98 public methods
  - Class STRING: test failed for *25* out of 157 public methods

Example of found bug

```
class ARRAYED_LIST [G] ...
    put (v: like item) is
            -- Replace current item by `v'.
            -- (Synonym for `replace')
        require
            extendible: extendible
        do
            replace (v)
        ensure
            item_inserted: is_inserted (v)
            same_count: count = old count
        end
    replace (v: like first) is
            -- Replace current item by `v'.
        require
            writable: writable
        do
            put_i_th (v, index)
        ensure
            item_replaced: item = v
        end
    extendible: BOOLEAN is True
```

1 <= index <= count

(there exists a current element)

NOT ALWAYS TRUE!

Conclusions

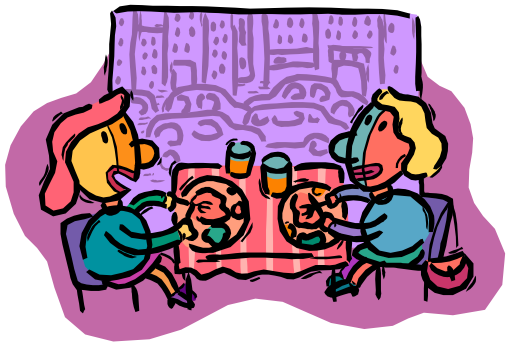
- We focus on *functional unit testing*
- Main contribution: *practical*
- Tool: *AutoTest* (http://se.inf.ethz.ch/research/tests.html)
- *Strengths*:
  - Fully automatic
  - Approach tested on full-fledged, industrial-scale applications
  - Approach can be extended to any programming language with support for assertions
  - No need to modify the software in order to test it
  - Found (an impressive number of) real bugs in real libraries

Future work

- Perform a thorough *evaluation* of the random strategy
- *Generation of input values*
  - Improve planning strategy
  - Try alternatives
- Minimization of *failure-reproducing examples*
- Incorporating the *knowledge of the human tester* in the process: integration with manual unit tests
- Does the use of *contracts* for automatic testing
  - decrease novice developers' reluctance towards them?
  - improve the quality of the contracts written by developers?

Final point

"Over-lunch testing" concept already in operation



Future work: "over-coffee-break testing"

TO DO