Applying Search in an Automatic Contract-Based Testing Tool

Alexey Kolesnichenko, Christopher M. Poskitt, and Bertrand Meyer

ETH Zürich, Switzerland

Abstract. Automated random testing has been shown to be effective at finding faults in a variety of contexts and is deployed in several testing frameworks. AutoTest is one such framework, targeting programs written in Eiffel, an object-oriented language natively supporting executable pre- and postconditions; these respectively serving as test filters and test oracles. In this paper, we propose the integration of search-based techniques—along the lines of Tracey—to try and guide the tool towards input data that leads to violations of the postconditions present in the code; input data that random testing alone might miss, or take longer to find. Furthermore, we propose to minimise the performance impact of this extension by applying GPU programming to amenable parts of the computation.

1 Introduction

Automated random testing has become widely used, in part because it is inexpensive to run, relatively simple to implement, and most importantly has been demonstrated to be effective at finding bugs in programs; the technique having been implemented in several testing frameworks including AutoTest [7], JCrasher [3], and Randoop [8]. Using random testing comes with the cost of only using straightforward strategies, and in particular, not leveraging information from previous executions or specifications that—if provided—might otherwise help guide towards test data revealing undiscovered bugs.

The AutoTest framework targets programs written in the object-oriented language Eiffel, which natively supports *contracts* [6], i.e. executable pre- and postconditions for routines (although the framework could be adapted to other contract-equipped languages such as Java with JML). Contracts go hand-in-hand with random testing, with preconditions serving as *filters* for test data, and postconditions serving as *test oracles*. Furthermore, there are techniques to guide the selection of inputs towards ones that satisfy preconditions, e.g. the precondition-satisfaction strategy of AutoTest [12]. We claim in this short paper however that there is still more to be gained from contracts in automated testing. In particular, we propose that contracts are ideal for integration with search-based techniques for test data generation [5]; passing tests could be "measured" against how close they are to violating the postconditions, with *fitness functions* then favouring input data that optimises this measure. This strategy exploits ready-to-use contracts to focus the generation of test data towards objects that get closer to violating postconditions, hence possibly revealing bugs that random testing alone might miss, or take longer to find.

Applying search to test data generation is of course a computationally more expensive task—whilst the envisaged techniques might reveal individual bugs that random testing might miss, the approach quickly loses appeal if the ratio of bugs encountered over time suffers. Hence, we propose to investigate how to implement the computationally expensive parts on modern GPU devices, following on from previous work in the search community (e.g. [13]).

The rest of the paper is structured as follows: Section 2 provides an overview of AutoTest and how search-based techniques might be applied; Section 3 speculates on measuring how "close" input data is to deriving outputs that violate postconditions; Section 4 discusses the application of GPU computation to search; and Section 5 outlines our plans and concludes.

2 Extending the AutoTest Workflow

Having introduced the idea of AutoTest in the previous section, we illustrate its workflow via a simple example. Consider the square root routine in Listing 1; implementation details are not given, but we provide its contract. The precondition, given after **require**, expresses that the input is non-negative; the postcondition, given after **ensure**, expresses the same. Recall that in AutoTest, preconditions are filters and postconditions are oracles. Hence in this example, `sqrt` is only tested on non-negative inputs, and any negative output indicates that its implementation does not meet its specification.

```
sqrt (a : DOUBLE) : DOUBLE
require
  a >= 0
ensure
  Result >= 0
end
```

Listing 1: Square root contracts

The current workflow of AutoTest is roughly as follows: firstly, random inputs satisfying the routine's precondition are generated. Secondly, the routine is executed on the generated data. If there is a postcondition violation, then the test fails and is recorded. Otherwise, the test passes. The overall picture is shown in Figure 1.

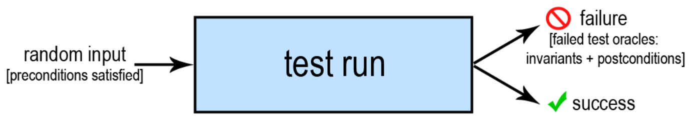

Fig. 1. High-level overview of the AutoTest workflow

This workflow however does not take into account information from *successful* test cases. Currently, we can say that they satisfied the test oracle; but more interestingly, can we measure how “close” they came to failing it, and use this information in search to derive input data that gets even closer? A high-level picture of the proposed extension to AutoTest is shown in Figure 2. We discuss the issues of search and measuring “closeness” in the following section.

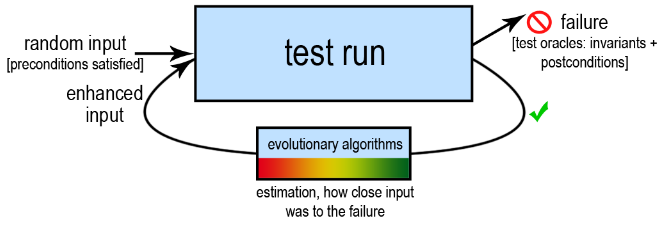
random input
[preconditions satisfied]
test run
failure
[test oracles: invariants + postconditions]
enhanced input
evolutionary algorithms
estimation, how close input was to the failure

Fig. 2. High-level overview of the proposed extension to the AutoTest workflow

We remark that search was applied to AutoTest previously in [10]; however, their approach differs in that they use genetic algorithms to generate an efficient testing strategy by evolving random ones. Other authors [2] have suggested that condition coverage on postconditions—in a testing tool for Java programs equipped with contracts—seems promising in generating test data, but as far as we know, did not publish implementations or evaluations of such algorithms.

3 Optimising Postcondition Violations

A key part of our proposal involves evaluating how “close” input data is to deriving a postcondition violation. We propose to follow Tracey [11], who defined *objective functions* for logical connectives and relations over data of primitive types. The concepts can be applied to similarly simple contracts, so we illustrate with an example (based on the counter algorithm presented in [11,5]). A *faulty* wrapping counter is shown in Listing 2. It is supposed to take an integer between 0 and 10 (see the precondition), returning that integer incremented by 1 if it was less than 10, and 0 otherwise (see the postcondition).

We can negate the postcondition, and add it in conjunction to the precondition to form a constraint only satisfiable by input data that derives a fault; for example:

$$(n \geq 0 \vee n \leq 10) \wedge \neg((n = 10 \rightarrow \mathit{Result} = 0) \wedge (n \neq 10 \rightarrow \mathit{Result} = n + 1)).$$

We can apply Tracey’s objective functions to the relations and connectives of the constraint, yielding a value indicating the “closeness” to satisfying it (smaller values indicating that we are closer). For example, for relations $a = b$, we can define $\mathrm{obj}(a = b)$ to return the absolute difference between the values of a and b. Other relational predicates can be measured in a similar fashion. Objective functions are defined inductively for logical connectives. For example, consider

```
cyclic_increment (n : INTEGER) : INTEGER
require
  n >= 0 and n <= 10
do
  if (n > 10) then
    Result := 0
  else
    Result := n + 1
  end
ensure
  (n = 10 implies Result = 0) and (n /= 10 implies Result = n + 1)
end
```

Listing 2: Faulty wrapping counter

the formula $a \vee b$. A suitable definition of $\text{obj}(a \vee b)$ would be $\min(\text{obj}(a), \text{obj}(b))$, i.e. the minimum value of the two parameters. With a fitness function including such a measure, we hope to apply metaheuristic search techniques [4], optimising the search towards input data that gets closer to violating postconditions (i.e. revealing bugs).

In our wrapping counter example, the smallest output of the objective function should be yielded for $n = 10$, since the implementation incorrectly increments the counter for this value.

For real object-oriented programs, we encounter the challenge of hidden states: objects tend to conceal their implementation details. For example, a routine of a bounded buffer might assert in its postcondition that the buffer is not full. Expressed as `buffer.count` < `buffer.size`, we can apply objective functions as we have described. However, the postcondition might instead be expressed as **not** `buffer.is_full`, i.e. a Boolean query. Boolean queries are not informative for metaheuristic search because of their "all or nothing" nature. In this example, we cannot distinguish between a buffer that is completely empty and another that is close to being full. Postconditions containing Boolean queries should be transformed into equivalent postconditions that are more amenable for testing. A solution proposed by [1] is to "expand" Boolean queries using their specifications; an approach compatible with our contract-based one. For example, the postcondition of `is_full` might express that the result is true if and only if `count` >= `size`; this being an assertion to which objective functions can be applied more successfully.

4 Performance Considerations

The ideas described in the previous sections do not come for free. Additional computation increases the running time of the tool, and may adversely affect the ratio of bugs found over time. In order to keep the performance as reasonable as possible, we propose to apply GPU computing to speed-up the computationally intensive aspects of the search. Consider for example the family of genetic algorithms (GAs). The population can be encoded as a numerical vector, and the fitness function as a vector function $f \colon \mathbb{R}^n \to \mathbb{R}$.

Essentially, the GA input can be represented as a matrix $m \times n$, where m is the population size, and n is the size of the chromosome vector. Thus, to evaluate a fitness function, one just needs to apply some vector function to each matrix row. A mutation operation (changing chromosomes of some species subset) can also be performed row-wise. Crossover operation is also essentially row-based.

GPUs are very different to conventional CPUs. Whereas CPUs are designed as general-purpose computing devices, with lots of optimisations like branch prediction, multi-level caches, etc., the processing units of GPUs are much simpler. A GPU's processing unit cannot handle the processing of arbitrary data as efficiently as CPUs can; they do not possess sophisticated hardware optimisations. However, there are many more processing cores on a GPU device, compared to the CPU. GPUs are tuned for data parallelism, implementing the SIMD (Single Instruction - Multiple Data) processing model, allowing the execution of thousands of threads in parallel. GPUs have proven to be extremely efficient with matrix-style computations [9], providing a convincing speed-up of 2-3 orders of magnitude.

GPU acceleration was used for the problem of minimising test suite size in [13], and demonstrated that speed-ups of over 25x are possible using GPU computing.

5 Research Plans and Conclusion

The proposed ideas—namely implementing search-based techniques to improve the fault discovery rate of a contract-based random testing tool, and using GPU acceleration to limit the performance hit—need to be carefully evaluated. While we believe that search will enhance the quality of inputs in AutoTest, there are several risks and challenges to be dealt with along the way to confirming or disproving this hypothesis.

A first challenge is the previously mentioned implementation hiding in object-oriented languges, that makes guided search ineffective without first transforming Boolean queries into postconditions that are better suited for objective functions. A second challenge: one should never forget that the goal of testing tools is to reveal as many faults as possible. That is why the enhanced tool needs to be tested against previously successful strategies, e.g. the precondition-satisfaction strategy [12] of AutoTest.

Thirdly, some thought needs to be given as to the particular type of search algorithm to apply (e.g. a genetic algorithm), and how to best encode the objects and data on which these algorithms operate. The final choice will be determined by the quality of inferred data and amenability to GPU-style computations. Finally, one needs to take into account, that GPU acceleration may be overwhelmed by the additional overhead of copying data from main memory to the GPU, and vice versa. Thus, GPU computing should only be applied to computationally intensive parts of the proposed AutoTest workflow.

Acknowledgments. The research leading to these results has received funding from the European Research Council under the European Union's Seventh Framework Programme (FP7/2007-2013) / ERC Grant agreement no. 291389, the Hasler Foundation, and ETH (ETHIIRA). The authors would like to thank Simon Poulding for helpful discussions.

References

1. Y. Cheon and M. Kim. A specification-based fitness function for evolutionary testing of object-oriented programs. In *Proc. Genetic and Evolutionary Computation Conference (GECCO 2006)*, pages 1953–1954. ACM, 2006.
2. Y. Cheon, M. Kim, and A. Perumandla. A complete automation of unit testing for Java programs. In *Proc. International Conference on Software Engineering Research and Practice (SERP 2005)*, pages 290–295. CSREA Press, 2005.
3. C. Csallner and Y. Smaragdakis. Jcrasher: an automatic robustness tester for java. *Software: Practice and Experience*, 34(11):1025–1050, 2004.
4. M. Harman. The current state and future of search based software engineering. In *Proc. Future of Software Engineering (FOSE 2007)*, pages 342–357. IEEE, 2007.
5. P. McMinn. Search-based software test data generation: a survey. *Software Testing, Verification and Reliability*, 14(2):105–156, 2004.
6. B. Meyer. *Object-Oriented Software Construction.* Prentice Hall, 2nd edition, 1997.
7. B. Meyer, A. Fiva, I. Ciupa, A. Leitner, Y. Wei, and E. Stapf. Programs that test themselves. *IEEE Computer*, 42(9):46–55, 2009.
8. C. Pacheco, S. K. Lahiri, and T. Ball. Finding errors in .NET with feedback-directed random testing. In *Proc. International Symposium on Software Testing and Analysis (ISSTA 2008)*, pages 87–96. ACM, 2008.
9. S. Ryoo, C. I. Rodrigues, S. S. Baghsorkhi, S. S. Stone, D. B. Kirk, and W. W. Hwu. Optimization principles and application performance evaluation of a multithreaded GPU using CUDA. In *Proc. Symposium on Principles and Practice of Parallel Programming (PPOPP 2008)*, pages 73–82. ACM, 2008.
10. L. S. Silva, Y. Wei, B. Meyer, and M. Oriol. Evotec: Evolving the best testing strategy for contract-equipped programs. In *Proc. Asia Pacific Software Engineering Conference (APSEC 2011)*, pages 290–297, 2011.
11. N. J. Tracey. *A Search-Based Automated Test-Data Generation Framework for Safety-Critical Software.* PhD thesis, The University of York, 2000.
12. Y. Wei, S. Gebhardt, M. Oriol, and B. Meyer. Satisfying test preconditions through guided object selection. In *Proc. International Conference on Software Testing, Verification and Validation (ICST 2010)*, pages 303–312. IEEE, 2010.
13. S. Yoo, M. Harman, and S. Ur. Highly scalable multi objective test suite minimisation using graphics cards. In *Proc. International Symposium on Search-Based Software Engineering (SSBSE 2011)*, volume 6956 of *LNCS*, pages 219–236. Springer, 2011.