Automatic Structural Testing with Abstraction Refinement and Coarsening*

Mauro Baluda
Faculty of Informatics
University of Lugano, Switzerland
mauro.baluda@usi.ch

ABSTRACT

White box testing, also referred to as structural testing, can be used to assess the validity of test suites with respect to the implementation. The applicability of white box testing and structural coverage is limited by the difficulty and the cost of inspecting the uncovered code elements to either generate test cases that cover elements not yet executed or to prove the infeasibility of the elements not yet covered.

My research targets the problem of increasing code coverage by automatically generating test cases that augment the coverage of the code or proving the infeasibility of uncovered elements, and thus eliminating them from the coverage measure to obtain more realistic values. Although the problem is undecidable in general, the results achieved so far during my PhD indicate that it is possible to extend the test suites and identify many infeasible elements by suitably combining static and dynamic analysis techniques, and that it is possible to manage the combinatorial explosion of execution models by identifying and remove elements of the execution models when not needed anymore.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging—*Testing Tools (E.G., Data Generators, Coverage Testing)*

General Terms

Algorithms, Verification

1. PROBLEM STATEMENT

Testing is the most popular quality assurance technique in the software industry. One of the challenges of testing is to evaluate the adequacy of a test suite, that is, establishing whether a test suite exercises the software to a sufficient extent, or must be augmented with other test cases. One of the ways proposed by researchers to quantify the adequacy of testing is to measure code coverage, defined as the amount of code elements of a given type (for example, statements, branches or data flow uses) exercised by a test suite, with respect to the number of those elements in the program [12].

Despite the large body of scientific literature on code coverage, only simple coverage metrics find some practical interest, and their industrial application is still limited. The main problem that reduces the applicability of coverage metrics amounts to the high effort needed for manually devising test cases that execute specific code elements, especially if the coverage domain is huge and includes infeasible elements, that is, elements that belong statically to the code but cannot be executed for any possible input. Infeasible elements both divert the testing effort, since testers may waste resources to prove that they cannot be executed, and weaken the meaningfulness of the coverage metrics, since achieving high coverage rates can be impossible when the portion of undetected infeasible elements is large, as is often the case when considering advanced coverage metrics (for example data-flow criteria) and complex code (for example code that include OTS components and exploit defensive programming).

The problem of augmenting test suites and improving code coverage is attracting a lot of interest. Recent work proposes techniques for either generating test cases by exploring the program paths in some order, or proving the reachability of specific faulty statements, but no proposal addresses both goals in a synergistic fashion yet [10, 11, 5, 9, 13].

My PhD research focuses on synergistic solutions that increase code coverage, while identifying as many infeasible code elements as possible. Identifying infeasible elements can both speed up the convergence of the analysis, by avoiding to be stuck in the impossible attempt of covering an infeasible element, and strengthen the precision of the coverage measurements, by pruning the infeasible elements out of the coverage counts.

2. RELATED WORK

So far researchers have investigated automatic test generation and code reachability as separate problems. Test generation has been addressed with approaches based on input space search and symbolic execution. Code reachability has been explored in the domain of software model checking.

Random testing samples the input space randomly. It can quickly produce large test suites that cover part of the code, but typically fails to cover code that depends on corner val-

*This research is supervised by Mauro Pezzè, University of Lugano, Switzerland (mauro.pezze@usi.ch).
During my research I coauthored two publications that present the initial results of my PhD work [1, 2].

ESEC/FSE'11, September 5–9, 2011, Szeged, Hungary.

ues or rare behaviors. More elaborate search strategies proposed in literature, for example based on genetic algorithms, may outperform random testing in some circumstances, but share similar problems in the general case [10].

Symbolic execution computes logic formulas that characterize the inputs that drive the execution along sets of program paths, and finds inputs that execute given paths by satisfying these formulas by means of automatic solvers [11]. Concolic execution exploits the interplay between symbolic execution and dynamic concrete analysis to overcome the limitations of theorem provers and progress the analysis. Popular tools that exploit concolic execution, like DART [5] and PEX [9], proceed through a depth-first analysis of program paths, which may lead to explore only a small portion of the program state space. Different search heuristics have been proposed, but none of them seems to perform consistently better then the others. Neither Random/genetic searches nor symbolic execution account for infeasible code.

Techniques for proving reachability properties of programs have been developed in the area of software model checking. Counter-Example Guided Abstraction Refinement (CEGAR) addresses the unreachability of faulty statements by conservatively over-approximating a program state space to a finite model, and then refining the model incrementally up to the precision needed for the proof [7].

The approaches closest to my research are the one proposed by Beckman et al., who combine CEGAR and testing in a verification approach that generates test cases and triggers suitable abstraction refinement when the test generation fails [6, 3], and McMillan, who proposes a similar procedure that exploits an interpolating prover [8]. Both approaches can prove the reachability of specific faulty statements, but do not apply well to the problem of identifying infeasible elements within large coverage domains, since this problem requires to simultaneously solve many reachability problems, becoming intractable even for small size programs, as demonstrated experimentally when we tried to apply CEGAR based techniques to cover the code coverage problem.

3. RESEARCH OBJECTIVES

My PhD research tackles the problem of both increasing code coverage and identifying unreachable code elements by combining static and dynamic techniques and complementing analysis with a new technique that we call *coarsening* and that contrasts the state explosion of CEGAR by unrolling the refinement steps as soon as the analysis satisfies any intermediate reachability goal [1, 2].

I plan to meet the main goal of my thesis by exploring the following research directions:

- Assessing existing and novel analysis techniques to generate test cases that exercise code elements not yet covered and identify unreachable ones.
- Studying how techniques for generating test cases, which typically reason at the level of the control flow of the program under analysis, can be adapted and extended to efficiently address coverage domains based on different program abstractions, such as, data-flow models.
- Conducting extensive experimental campaigns to identify the best balance between precision and efficiency of the proposed solutions. This is of paramount importance for the approach to scale to practical software, and then be useful in practice.

4. TECHNICAL CHALLENGES

The main technical challenges relate to the research directions outlined above: suitably combining and extending static and dynamic analysis techniques to efficiently generate test cases and rule out infeasible elements, tackling advanced coverage metrics, and instantiating the approach into an efficient analysis framework.

Symbolic and concolic execution are well known techniques for generating test cases, and are implemented in several robust tools. The main bottleneck for the applicability of symbolic and concolic executors is the underlying constraint solvers. Automatic solvers can reason efficiently about basic arithmetic constructs, but do not perform well when tackling non linear operations, and cannot deal with the complexity of floating point representation. The first challenge of my PhD work is to leverage on existing techniques for overcoming the limitation of current constraint solvers in the scope of the problem of interest.

The performance of current approaches for proving code reachability depends on the specific problem instance. For example, the relative qualities of popular approaches to devise the state predicates needed for abstraction refinement, for instance weakest precondition calculus and Craig interpolants [3, 8], are still unclear. My research will address the challenge of identifying an effective combination of analysis techniques by empirically investigating the performance of different approaches, to find a suitable adaptation to the specificity of structural testing. In the first part of my PhD, I experimentally investigated the applicability of CEGAR and I found out that CEGAR performs well when analyzing the reachability of single program statements, but does not perform equally well when analyzing the reachability of a whole coverage domain as required by our problem. The performance problems derive from the complexity of the models maintained during analysis that easily goes out of memory. In the first part of my PhD, I proposed a novel technique (called coarsening) to control memory consumption while CEGAR progresses [1]. Tackling the state explosion problem by leveraging the specificity of test case generation is a crucial challenge to my research.

Generating test cases for a wide range of coverage criteria challenges the path exploration strategies available in the literature. Current symbolic test generation techniques are control flow directed and do not target complex criteria like data flow based structural coverage, which would greatly benefit from the approach, since a large portion of the coverage domain is typically infeasible with such criteria.

The availability of an efficient analysis tool can show the potential of the approach in supporting fault detection and quality assessment activities for software of industrial interest. A remarkable engineering and optimization effort is required to exploit the characteristics of modern machines like complex memory hierarchies and parallelism in the implementation of the tool.

5. CURRENT STATUS

In the first part of my PhD I focused mostly on the challenge of evaluating, combining and complementing existing analysis techniques to deal with branch coverage. I refer to the approach as *ARC (abstraction refinement and coarsening)*, and to the prototype tool that works for branch coverage as (ARC-B).

The ARC approach works on what we call the *frontier* between covered and not-covered code elements, the frontier includes the transitions that connect elements of the two groups in the execution tree of the program under test. The frontier plays a key role in exploiting the interplay of static and dynamic analysis techniques effectively.

ARC identifies the current frontier by referring to the **coverage domain model**, which is a state transition system that represents all the elements of the coverage domain as abstract states (the coverage targets), and overapproximates the possible program flows as transitions between the abstract states. ARC combines in a unitary framework four static and dynamic analysis components:

Coverage frontier monitoring: a dynamic analysis technique that traces sets of executions against the coverage domain model, measures the current coverage, and identifies the *frontier* between the reached and the unreached abstract states. The frontier indicates promising program flows to be explored to reach coverage targets not yet covered.

Coverage frontier analysis: an analysis technique that reasons on the frontier of the coverage domain model. This analysis aims to identify new test cases that can traverse the frontier towards uncovered targets, reach abstract states not reached yet and possibly increase the code coverage. The current ARC-B prototype exploits symbolic execution techniques as a support for coverage frontier analysis. When the coverage frontier analysis fails in generating a test case that traverses the frontier, it triggers the coverage frontier refinement analysis technique that refines the model to identify and eliminate unreachable flows.

Coverage frontier refinement: an abstraction refinement analysis technique that increases the level of precision of the coverage domain model with respect to a set of frontier transitions. This analysis aims to transform (possibly recompute) the current model either to better characterize the set of executions that can reach a given frontier transition, or to remove such transition from the model when the analysis can conclude that no program execution may reach that transition. The increased precision can allow test generation to progress, while removing transitions can detect unreachable abstract states, and possibly identify infeasible coverage targets. The current ARC-B prototype exploits CEGAR model checking as a support for coverage frontier refinement.

Coarsening: a maintenance analysis technique that contrasts the state explosion problem caused by reiterated abstraction refinement. While coverage frontier analysis and refinement alternate to explore the execution space of the program and move the frontier further in the coverage domain model, many elements created during refinement become irrelevant. Coarsening alleviates the state explosion problem by eliminating redundant elements from the coverage domain model. Coarsening tracks the relation between the refinements due to the analysis of the transitions leading to a given abstract state and the abstract state itself, and deletes the information produced by the refinement step when the corresponding abstract state is either reached or removed from the model. Coarsening is triggered by the frontier monitoring component upon reaching abstract states, and by the frontier refinement component upon removing unreachable abstract states.

The prototype tool ARC-B helped me in collecting a preliminary set of empirical data on the validity of the approach for branch coverage. In ARC-B, the abstract states of the coverage domain model correspond to the branches in the program execution. ARC-B initializes the coverage domain model based on the control-flow graph of the program, and while the test generation progresses it augments the abstract states with constraints on the values of the program variables, to assist the reachability analysis. ARC-B implements frontier monitoring evaluating the abstract states against the program execution during the test runs. ARC-B implements frontier analysis in the style of concolic execution, that is, by performing lightweight symbolic execution on a concrete path up to a frontier, and then using a solver to compute a test case for traversing the frontier. ARC-B implements frontier refinement in the style of template-based abstract refinement [3], that is, by augmenting the pre-frontier abstract states with the weakest precondition to reach the post-frontier abstract states. By doing this, it incrementally improves the precision of the analysis done by the test generation stage, and it can eventually detect contradictory preconditions and thus determine the possibility of removing transitions. Finally, ARC-B includes an implementation of coarsening that removes the conditions generated by the refinements upon traversing or removing the frontier transitions.

The empirical data collected with ARC-B on 12 sample C programs indicate that ARC-B generated test cases that cover most branches (543 out of 588), and identified most unreachable branches (34 out of at most 45). The coverage obtained with ARC-B spans from 96% to 100% of the branches in the programs and outperforms state of the art tools. Additional details of the experiments are reported in [1].

6. RESEARCH PLAN

My research activity is conceived as an incremental process driven by experimental evaluation of the proposed solutions with respect to the research objectives. The need to experiment with existing analysis techniques to evaluate their effectiveness in practical settings suggested the design of a general architectural framework to simplify their integration and evaluation. The proposed framework identifies a small number of analysis steps that can be embodied by different combinations of static and dynamic techniques allowing innovative combinations and their empirical evaluation.

My future plan is articulated in two main phases. In the first phase I plan to extend the initial results obtained so far to increase branch coverage of large programs. The results obtained with ARC-B show that the approach is potentially interesting and useful, but are limited to small size programs, and thus do not generalize yet. To understand the applicability and the limits of the analysis techniques embedded in ARC-B, I plan to work experimentally, and I need to both extend the approach and the prototype tool, to deal with a larger subset of the C language, and ameliorate the constraint solver support that seems to be the bottleneck of the current prototype.

Currently the template-based Coverage frontier refinement implemented in ARC-B treats pointer aliases with imprecision (making the detection of infeasible elements reliable only for programs that do not contain aliases) and is limited to intra-procedural analysis. I plan to investigate the use of recently proposed algorithms to precisely account for aliases

during symbolic execution and abstraction refinement [3, 4]. I also plan to integrate support for inter-procedural analysis by extending the current coverage domain model according to the program call graph, and designing call-return context sensitiveness in the current algorithms. I finally plan to integrate other techniques for abstraction refinement, and in particular I will study the applicability of interpolant based predicate refinement [8].

To improve the constraint solving power, I need to compare the performances of the different kinds of solvers in the context of the target problem. I plan to work experimentally by designing a parametric interface to integrate different kinds of solvers in the ARC-B prototype tool and compare the solvers experimentally in the filed. I am currently introducing an homogeneous interface for SMT solvers and implementing ad hoc techniques to treat specific nonlinearities like the ones arising from the application of modulo and division operators between integer variables and constants.

The improvement in the technique and in the prototype tool will enable the collection of further experimental data on the efficiency of the approach, while the identification of possible bottlenecks will indicate the elements of the analysis that require further refinement. A key element for the scalability of the approach is the new coarsening technique that I introduced in the first part of my PhD, and that succeeded in removing redundancy from the model and controlling the state explosion issue. I believe that the core technique can be further improved by introducing strategies to prioritize and parallelize the selection of the frontier transitions to be traversed. An aggressive preprocessing step to the analysis of the software could also improve the approach considerably, in particular I will consider precise program dependence analysis as a means to reduce the number of transitions in the coverage domain model.

This first phase will result in an approach to increase branch coverage up to approximating 100% coverage. In the second and last phase, I plan to extend the approach towards finer grain coverage criteria. In particular, I aim to cover data flow criteria, since I expect the ARC approach to be particularly suitable for these criteria that suffer greatly from the presence of infeasible elements. Extending the approach to a wider class of coverage criteria would require only the definition of different coverage domain models and the adaptation of the coverage frontier monitor to the new models. Coverage frontier analysis, refinement and coarsening depend only on the coverage frontier monitor and are independent from the coverage criteria. Thus this second phase should not be too demanding.

A continuous validation effort will highlight strengths and weaknesses of the different analysis combinations and new directions of improvement. The possibility to compare results from different analysis techniques on the same subjects will increase confidence in the correctness of their implementation. I will evaluate the effectiveness of ARC against increasingly large programs from open source domains (e.g., common UNIX utilities). Through all experiments, I will evaluate the failure detection ability of the test suites computed by the ARC framework, as the objective of any novel testing method is to be effective in failure detection.

In the final evaluation I will try to quantify the testing effort and effectiveness induced by ARC, and I will compare the results both with other automated techniques (for instance, random testing) and manual test-design strategies.

7. REFERENCES

[1] M. Baluda, P. Braione, G. Denaro, and M. Pezzè. Structural coverage of feasible code. In *Proceedings of the 5th Workshop on Automation of Software Test*, AST '10, pages 59–66, New York, NY, USA, 2010.

[2] M. Baluda, P. Braione, G. Denaro, and M. Pezzè. Structural coverage of feasible code. In *Software Quality Journal*, 2011, to appear.

[3] N. E. Beckman, A. V. Nori, S. K. Rajamani, and R. J. Simmons. Proofs from tests. In *Proceedings of the 2008 international symposium on Software testing and analysis*, ISSTA '08, pages 3–14, New York, NY, USA, 2008. ACM.

[4] B. Elkarablieh, P. Godefroid, and M. Y. Levin. Precise pointer reasoning for dynamic test generation. In *Proceedings of the eighteenth international symposium on Software testing and analysis*, ISSTA '09, pages 129–140, New York, NY, USA, 2009. ACM.

[5] P. Godefroid, N. Klarlund, and K. Sen. Dart: directed automated random testing. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '05, pages 213–223, New York, NY, USA, 2005. ACM.

[6] B. S. Gulavani, T. A. Henzinger, Y. Kannan, A. V. Nori, and S. K. Rajamani. Synergy: a new algorithm for property checking. In *Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering*, SIGSOFT '06/FSE-14, pages 117–127, New York, NY, USA, 2006. ACM.

[7] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '02, pages 58–70, New York, NY, USA, 2002. ACM.

[8] K. McMillan. Lazy annotation for program testing and verification. In T. Touili, B. Cook, and P. Jackson, editors, *Computer Aided Verification*, volume 6174 of *Lecture Notes in Computer Science*, pages 104–118. Springer Berlin / Heidelberg, 2010.

[9] N. Tillmann and J. De Halleux. Pex: white box test generation for .net. In *Proceedings of the 2nd international conference on Tests and proofs*, TAP'08, pages 134–153, Berlin, Heidelberg, 2008. Springer-Verlag.

[10] P. Tonella. Evolutionary testing of classes. In *Proceedings of the 2004 ACM SIGSOFT international symposium on Software testing and analysis*, ISSTA '04, pages 119–128, New York, NY, USA, 2004. ACM.

[11] W. Visser, C. S. Păsăreanu, and S. Khurshid. Test input generation with java pathfinder. In *Proceedings of the 2004 ACM SIGSOFT international symposium on Software testing and analysis*, ISSTA '04, pages 97–107, New York, NY, USA, 2004. ACM.

[12] E. J. Weyuker. The evaluation of program-based software test data adequacy criteria. *Communications of The ACM*, 31:668–675, June 1988.

[13] Z. Xu, Y. Kim, M. Kim, G. Rothermel, and M. B. Cohen. Directed test suite augmentation: techniques and tradeoffs. In *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering*, FSE '10, pages 257–266, New York, NY, USA, 2010. ACM.