Invariant Discovery Guided by Symbolic Execution

Lingming Zhang*, Guowei Yang*, Neha Rungta†, Suzette Person‡, Sarfraz Khurshid*
*Department of Electrical and Computer Engineering, University of Texas, Austin
Email: zhanglm@utexas.edu, {gyang,khurshid}@ece.utexas.edu
†NASA Ames Research Center, Email: neha.s.rungta@nasa.gov
‡NASA Langley Research Center, Email: suzette.person@nasa.gov

***Abstract*—Program invariants are useful for software implementation, testing, and maintenance activities. However, invariants are difficult to discover, and can take intensive manual effort to validate. A large body of research has focused on automated invariant discovery. The most widely used techniques for invariant discovery (e.g., Daikon) are based on dynamic test execution traces. Although these techniques can efficiently generate a large number of invariants for real-world programs, there are two main issues: (1) they may generate a number of incorrect or imprecise invariants; and (2) they may miss correct invariants. In this paper, we describe preliminary work on *iDiscovery*, an iterative approach that uses symbolic execution results to guide the invariant generation process. More precisely, iDiscovery uses symbolic execution to refute incorrect invariants, refine imprecise invariants, and characterize new behaviors to help achieve more accurate invariant discovery.**

I. INTRODUCTION

Program invariants are useful for software implementation, testing, and maintenance activities. For example, they can be encoded as assertions for runtime checking or regression test oracles. They can also be used for inference of state-based behavior models and for specification recovery. However, invariants are also difficult to discover, and often require intensive manual effort to validate. A large body of research has focused on automated invariant discovery, e.g., Daikon [5]. These tools infer likely invariants based on dynamic test execution traces. Although these techniques can efficiently generate a large number of invariants for real-world programs, there are two main issues: (1) they may generate a number of incorrect or imprecise invariants; and (2) they may miss correct invariants. Dynamically inferred invariants are generated based on observed program executions which means they may miss relevant invariants if the test suite used to generate invariants does not cover relevant program executions. Moreover, there are no formal assurances that the generated invariants are correct even though these tools only produce invariants when there is some statistical confidence that their occurrence is not accidental. Thus, when a large number of invariants is generated, the process of distinguishing correct from incorrect invariants – a process that generally requires human intervention – is time consuming and error prone.

In this paper, we present preliminary work on *iDiscovery*, an approach that uses symbolic execution to help discover likely invariants. More precisely, iDiscovery treats the invariant discovery process as an iterative process using symbolic execution results as feed-back to the dynamic invariant discovery process. During each iteration, iDiscovery infers invariants based on the results of checking the last set of invariants using symbolic execution. Symbolic execution is used to invalidate incorrect invariants, refine imprecise invariants, and characterize new behaviors to help achieve more accurate invariant discovery. iDiscovery stops when the whole process reaches a fix point, i.e., no old invariants are deleted and no new invariants are discovered.

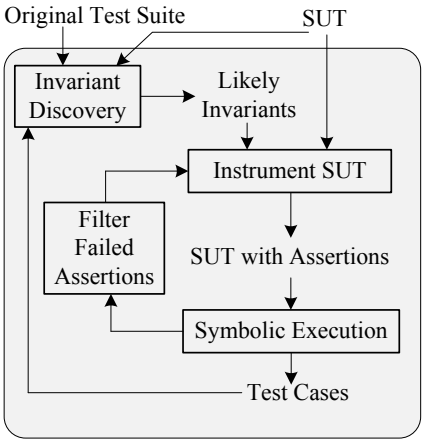

Fig. 1: iDiscovery Overview.

II. APPROACH

Our iDiscovery approach is a feed-back directed technique that combines dynamic invariant detection with symbolic execution [1], [9]. Automated techniques for dynamic invariant detection, e.g., Daikon [5], rely heavily on the quality and completeness of the input test suite and on internal techniques to minimize irrelevant invariants. Symbolic execution helps offset these limitations by systematically checking the code under analysis, exploring program execution traces that may not be covered by the input test suite, and generating counter-examples for invalid invariants. The generated test cases, including counter-examples for failed assertions and additional tests exposing new program behaviors, can be used to augment the test suite used for dynamic invariant detection. The list of invariants that cause the SUT to fail characterizes invariants that are likely invalid, and can be used to filter the invariants

generated in subsequent iterations of iDiscovery, and to ultimately reduce the number of invalid invariants considerable in the final set of results. During the generation of various tests to break existing invariants, some tests may also expose new program behaviors, and thus help with discovering new valid invariants. The process stops when a fix point is reached, e.g., no new invariants are generated and no old invariants are deleted.

The inputs to iDiscovery, shown in Figure 1 are: 1) the software under test (SUT), and 2) an initial test suite. The invariant discovery tool used by iDiscovery is Daikon [5]. Daikon uses the test suite to generate execution traces for the SUT. For each method in the program, Daikon generates one set of pre-conditions and one set of post-conditions for the method as a whole, and one set of post-conditions for each exit point (`return` statement) in the method. Each set of pre-conditions contains one or more constraints on the program inputs; each set of post-conditions contain one or more constraints on the return value and final state of the program under test. We transform the invariants generated by Daikon to Java `assert` statements and add the assertions to the appropriate location in the SUT using a custom application. To enable compilation and execution of the resulting code, we introduce fresh variables in the corresponding method. These variables are used to store the pre-state and the value of the expression in the `return` statement at the control point of interest, and to handle post-state expressions and $\texttt{arg}_k$ argument references.

Daikon's invariant generation relies heavily on the input test suite and Daikon's own internal collection of invariants, thus some of the invariants generated by Daikon may be invalid or irrelevant [13], [14]. To mitigate this factor, we use Symbolic PathFinder (SPF) [11] to symbolically execute the SUT annotated with assertions to check the invariants. The path conditions computed by SPF are solved to create test cases to augment the test suite provided to Daikon in the subsequent iterations. We use both passing test cases (execution traces that do not violate any of the assertions) and failing test cases (counter-examples) to augment the test suite provided to Daikon. Both types of test cases provide information that enables Daikon to compute more accurate invariants. We directly feed the execution traces of those additional failing and passing tests to Daikon. Based on the high-quality traces, Daikon will automatically filter out the invalid invariants and augment new valid invariants.

III. Preliminary Study

A. Study Setup

We perform the preliminary study on our iDiscovery approach using two versions of WBS program and three versions of the Tcas program. The Wheel Brake System (WBS), is a synchronous reactive component from the automotive domain. The Java model was implemented based on a Simulink model derived from the WBS case example [8], and consists of one class and 231 source lines of code. We applied iDiscovery on the `update` method in WBS. This method determines how much braking pressure to apply based on the environment. We randomly selected two versions of WBS from the variants WBS programs created in our previous work [12]. Trafc Anti-Collision Avoidance System (Tcas) is a system to avoid air collisions. We randomly selected three versions of Tcas from the Software Infrastructure Repository (SIR) [4], and manually converted the code to Java. The translated Java version has approximately 150 lines of code. Note that we inlined all the functionalities of Tcas into the `alt_sep_test` method, and applied iDiscovery on that method.

For both programs, we used 50 tests from their original test suites as the initial tests to evaluate iDiscovery. The 50 tests for WBS was randomly selected from the 1000 random tests created by our previous work [12]. The 50 tests for Tcas was randomly selected from its original test suite in SIR, which consists of 1608 tests.

Table I shows the results for iDiscovery on WBS and Tcas. The first column lists the artifact name and version. Column 2 lists the number of iterations of iDiscovery applied for each artifact. Note that we did not feed all the invariants generated by Daikon to the symbolic execution component, because some Daikon invariants cannot be directly translated into Java assertions. Thus, we present the results for both used and all invariants separately. For each iteration, Columns 3-5 list the number of used pre-conditions from the set of invariants computed by Daikon, the number of deleted used pre-invariants compared with the first iteration, and the number of added used pre-invariants compared to the first iteration. Similarly, Columns 6-14 present the statistics for the used pos-invariants, all pre-invariants, and all post-invariants for the programs under test. Columns 15 and 16 list the instruction coverage and branch coverage for the tests in each iteration (including both the total number of instructions/branches and the coverage ratio). Columns 17 and 18 list the number of additional tests generated based symbolic execution, and the generation time for the additional tests[1]. Note that we did not show the remaining iterations for the programs when iDiscovery reaches at the fix point.

B. Results and Discussion

Based on Table I, we have the following findings:

First, except the second version of WBS, the number of all types of invaraints becomes smaller after each interation until iDiscovery reaches a fix point. For example, iDiscovery successfully falsified 84.6% (121 out of the 143) of precondition invariants for all the three versions of Tcas. Manual inspection shows that all of the deleted (filtered) invariants are incorrect or imprecise. This demonstrates the effectiveness of our iDiscovery approach in invalidating incorrect or imprecise invariants.

Second, except the second version of WBS, iDiscovery also adds additional invariants for all the types of invariants. Some added invariants even keep valid after the fix point is

[1]The symboloc execution time for two versions of WBS is 0, because the time is too small to be caught by the SPF timer.

TABLE I: Initial experimental results for iDiscovery

Subjects	Iter.	UsedPreInvs			UsedPostInvs			AllPreInvs			AllPostInvs			Instr.	Branch	Symbc-	Symbc-
		Num	Del	New	Num	Del	New	Num	Del	New	Num	Del	New	Coverage	Coverage	TestNum	Time
wbs-v1	1	7	0	0	9	0	0	7	0	0	10	0	0	316(59.18%)	90(48.89%)	9	00:00:00
	2	4	4	1	9	2	2	4	4	1	9	3	2	316(80.06%)	90(68.89%)	25	00:00:00
	3	3	4	0	9	2	2	3	4	0	9	3	2	316(80.06%)	90(68.89%)	24	00:00:00
wbs-v2	1	4	0	0	8	0	0	4	0	0	8	0	0	316(59.18%)	90(48.89%)	4	00:00:00
	2	4	0	0	8	0	0	4	0	0	8	0	0	316(65.51%)	90(55.56%)	4	00:00:00
	3	4	0	0	8	0	0	4	0	0	8	0	0	316(65.51%)	90(55.56%)	4	00:00:00
tcas-v1	1	143	0	0	195	0	0	143	0	0	201	0	0	212(98.11%)	72(88.89%)	131	00:37:18
	2	30	117	4	71	131	7	30	117	4	77	131	7	212(98.58%)	72(93.06%)	137	00:03:03
	3	24	119	0	62	133	0	24	119	0	68	133	0	212(98.58%)	72(93.06%)	70	00:01:57
	4	22	121	0	60	135	0	22	121	0	66	135	0	212(98.58%)	72(93.06%)	68	00:01:41
tcas-v2	1	143	0	0	195	0	0	143	0	0	201	0	0	208(98.56%)	70(91.43%)	178	00:59:35
	2	31	117	5	72	131	8	31	117	5	78	131	8	208(98.56%)	70(92.86%)	331	00:08:26
	3	24	119	0	62	133	0	24	119	0	68	133	0	208(98.56%)	70(97.14%)	170	00:04:47
	4	22	121	0	60	135	0	22	121	0	66	135	0	208(98.56%)	70(97.14%)	168	00:04:11
tcas-v3	1	143	0	0	195	0	0	143	0	0	201	0	0	202(96.53%)	68(89.71%)	333	02:10:33
	2	31	117	5	72	131	8	31	117	5	78	131	8	202(98.51%)	68(94.12%)	723	00:17:42
	3	24	119	0	62	133	0	24	119	0	68	133	0	202(98.51%)	68(97.06%)	370	00:09:32
	4	22	121	0	60	135	0	22	121	0	66	135	0	202(98.51%)	68(97.06%)	368	00:07:59

reached for some program (e.g., the first version of WBS). Manual inspection show those additional invariants are correct or more precise than the deleted invariants. This demonstrate the effectiveness of our iDiscovery approach in finding new correct or more precise invariants.

Third, iDiscovery reaches a fix point in a relatively few number of iterations. In addition, both the instruction coverage and branch coverage grow and reach the peak before the number of invariants reaches the fix point. For WBS, both the instruction and branch coverage reaches a peak immediately after the first iteration. For Tcas, the instruction coverage reaches a peak of 98.5% after the first iteration, while the branch coverage sometimes requires one additional iteration to reach the peak. This demonstrates the stability of our iDiscovery approach.

IV. Related Work

Daikon pioneered the idea of dynamic detection of likely invariants and has been used in a number of applications, ranging from test suite minimization to fault tolerance [5]. The quality of Daikon's invariants depends largely on the collection of invariants in its invariant repository and the user-provided test suite, and as shown in a couple of recent studies, the quality can vary [13], [14].

DySy [3] presents the first application of forward symbolic execution [1], [9] to invariant discovery. Specifically, it uses symbolic execution over paths that are executed by a given test suite to compute path conditions of interest, which form a part of likely invariants generated. DySy does not use integrate with Daikon or use symbolic execution to validate its invariants.

While several projects [6], [10] have used static verification tools, e.g., ESC/Java [7] to check the invariants that are synthesized or dynamically discovered, the checking results are not used as feedback to further improve the discovery of invariants as is done in this work.

Xie and Notkin [15] are the first to connect invariant discovery and test input generation via a feedback-loop: generate tests using Jtest [2] based on the likely invariants discovered by Daikon, and run Daikon with the tests generated by Jtest. The feedback-loop is iterated to improve the quality of the invariants discovered as well as the tests generated. They do not use symbolic execution to generate tests or validate Daikon's output invariants.

References

[1] L. A. Clarke. A program testing system. In *Proceedings of the 1976 annual conference*, ACM '76, pages 488–491, 1976.

[2] P. Corporation. Jtest manuals version 4.5 october 23 (2002). http://www.parasoft.com/, 2002.

[3] C. Csallner, N. Tillmann, and Y. Smaragdakis. Dysy: Dynamic symbolic execution for invariant inference. In *ICSE*, pages 281–290, 2008.

[4] H. Do, S. Elbaum, and G. Rothermel. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *Empirical Software Engineering*, 10(4):405–435, 2005.

[5] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao. The daikon system for dynamic detection of likely invariants. *Sci. Comput. Program.*, 69(1-3):35–45, 2007.

[6] C. Flanagan and K. R. M. Leino. Houdini, an annotation assistant for ESC/Java. In *FME*, pages 500–517, 2001.

[7] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for Java. In *PLDI*, pages 234–245, 2002.

[8] A. Joshi and M. P. Heimdahl. Model-based safety analysis of simulink models using scade design verifier. In *Computer Safety, Reliability, and Security*, pages 122–135. Springer, 2005.

[9] J. C. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976.

[10] J. W. Nimmer and M. D. Ernst. Static verification of dynamically detected program invariants: Integrating Daikon and ESC/Java. In *Proceedings of RV'01, First Workshop on Runtime Verification*, Paris, France, July 23, 2001.

[11] C. S. Păsăreanu and N. Rungta. Symbolic Pathfinder: symbolic execution of Java bytecode. In *ASE*, pages 179–180, 2010.

[12] S. Person, G. Yang, N. Rungta, and S. Khurshid. Directed incremental symbolic execution. *ACM SIGPLAN Notices*, 47(6):504–515, 2012.

[13] N. Polikarpova, I. Ciupa, and B. Meyer. A comparative study of programmer-written and automatically inferred contracts. In *ISSTA*, pages 93–104, 2009.

[14] M. Staats, S. Hong, M. Kim, and G. Rothermel. Understanding user understanding: Determining correctness of generated program invariants. In *ISSTA*, pages 188–198, 2012.

[15] T. Xie and D. Notkin. Mutually enhancing test generation and specification inference. In *In Proc. 3rd International Workshop on Formal Approaches to Testing of Software, volume 2931 of LNCS*, pages 60–69, 2003.