The Yogi Project: Software Property Checking via Static Analysis and Testing

Aditya V. Nori[1], Sriram K. Rajamani[1], SaiDeep Tetali[1], and Aditya V. Thakur[2]

[1] Microsoft Research India
{adityan,sriram,v-saitet}@microsoft.com
[2] University of Wisconsin-Madison
adi@cs.wisc.edu

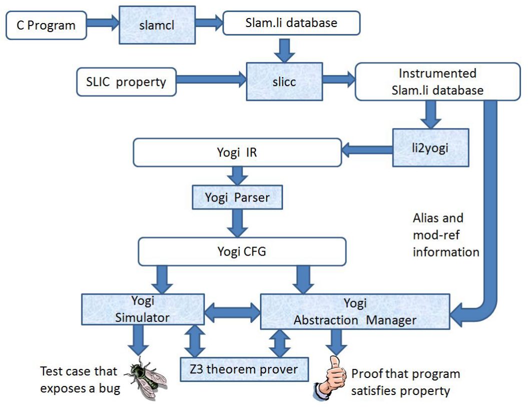
Abstract. We present Yogi, a tool that checks properties of C programs by combining static analysis and testing. Yogi implements the Dash algorithm which performs verification by combining directed testing and abstraction. We have engineered Yogi in such a way that it plugs into Microsoft's Static Driver Verifier framework. We have used this framework to run Yogi on 69 Windows Vista drivers with 85 properties. We find that the new algorithm enables Yogi to scale much better than Slam, which is the current engine driving Microsoft's Static Driver Verifier.

1 Introduction

Static analysis and testing have always had complementary strengths and weaknesses. With static analysis, we can obtain very good coverage and analyze program paths that are hard to exercise using testing, but we are forced to deal with scalability issues and false errors. With runtime testing, we can obtain only partial coverage, but the approach scales to large programs and every error that is reported is indeed realizable. Thus, attempting to combine the complementary strengths of static analysis and runtime testing is natural.

For the past few years, we have been investigating methods for combining static analysis in the style of counter-example driven refinement ala Slam [1], with runtime testing and automatic test case generation approaches in the style of concolic execution ala Dart [5]. Our first attempt in this direction was the Synergy algorithm [6], which handled single procedure programs with only integer variables. Then, we proposed Dash [3], which had new ideas to handle pointer aliasing and procedure calls in programs. Throughout this evolution, Yogi has been our implementation vehicle to realize and evaluate these algorithms. Currently, Yogi implements the Dash algorithm. We have spent over 3 person-years of engineering to make the tool robust and usable – Yogi has been run over several hundreds of thousands of lines of C code, with several properties.

We describe the design and engineering of Yogi in this paper. The Synergy and Dash algorithms themselves are described in [6,3]. Section 2 outlines the



(a) Architecture of YOGI

```
state {
   enum { Unlocked=0, Locked=1 }
     state = Unlocked;
}

KeAcquireSpinLock.return {
  if (state == Locked)
    abort;
  else
    state = Locked;
}

KeReleaseSpinLock.return {
  if (state == Unlocked)
    abort;
  else
    state = Unlocked;
}
```

(b) An example SLIC specification

Fig. 1.

architecture and various components in YOGI. Section 3 describes empirical results from running YOGI on 69 Windows Vista device drivers with 85 properties and Section 4 concludes the paper by discussing current status of YOGI.

2 Architecture

As shown in Figure 1(a), YOGI takes two inputs: (1) a C program, and (2) a safety property specified in the SLIC specification language [2]. A sample SLIC specification for a locking protocol (`KeAcquireSpinLock` and `KeReleaseSpinLock` occur in strict alternation) is shown in Figure 1(b). YOGI uses SLAM's front-end (called slamcl) to parse C programs and SLAM's property instrumentor (called slicc) to instrument the property into the program. The resulting program with the property instrumented is in SLAM's internal binary format called li. We have developed a translator called LI2YOGI that converts the li format to YOGI's intermediate form called yogi-ir. The yogi-ir is a textual format that represents the program at the level of basic blocks with instructions. Each instruction is one of three types: an assignment, assume statement or a procedure call. Once a program has been converted to the yogi-ir format, it is read by the YOGIPARSER to produce an internal inter-procedural control flow graph.

The two main components of YOGI are: (1) YSIM, a simulator which can perform both concrete execution with concrete values and symbolic execution, and (2) YABSMAN, an abstraction manager, which manages proofs.

The YSIM simulator code is polymorphic over the type of the values it operates. Thus, the same simulation code does both concrete and symbolic execution. During concrete execution, the simulator uses a model of memory where concrete values of appropriate type are stored in locations. During symbolic execution, the simulator stores symbols and formulas in locations. It uses the Z3 theorem prover [4] to reason about consistency of the formulas and to generate test cases as satisfiable models of formulas. The YABSMAN abstraction manager maintains a region graph abstraction of the program. For each control point in a program YabsMan maintains a finite partition over the set of states. Each partition is represented by a predicate, which is a Z3 formula.

Yogi implements the Dash algorithm [3]. The Dash algorithm simultaneously maintains a forest of test runs and a region-graph abstraction of the program. Tests are used to find bugs and abstractions are used to prove their absence. During every iteration, if a concrete test has managed to reach the error region, a bug has been found. If no path in the abstract region graph exists from the initial region to the error region, a proof of correctness has been found. If neither of the above two cases are true, then we have an abstract counterexample, which is a sequence of regions in the abstract region graph, along which a test can be potentially driven to reveal a bug. The Dash algorithm crucially relies on the notion of a *frontier* [6,3], which is the boundary between tested and untested regions along an abstract counterexample that a concrete test has managed to reach. In every iteration, the algorithm first attempts to extend the frontier using test case generation techniques similar to Dart. If test case generation fails, then the algorithm refines the abstract region graph so as to eliminate the abstract counterexample.

Yogi performs modular verification. For function calls, Yogi uses an initial abstraction that is based on locations that the procedure modifies. A conservative alias analysis is used to get an overapproximation to the set of locations modified by the procedure and this is used to build an initial summary for each function. If a procedure call occurs at the frontier, then the summary so computed is first used to see if a refinement can rule out the abstract counterexample. If this is not possible, Yogi tries to generate a test case through the procedure, and see if the test case extends the frontier. If the test case so generated does not extend the frontier, then Yogi descends into the called procedure and analyzes the procedure in detail [3].

3 Empirical Results

We have integrated Yogi with Microsoft's Static Driver Verifier framework. We have tested Yogi with the Static Driver Verifer's integration test pass suite, which contains 69 device drivers and 85 properties, a total of 5865 driver-property pairs. The largest driver in this pass has over 30K lines of code, and the total size of all the drivers is over 300K lines of code.

At the time of this writing Yogi finishes on 95% of the runs on the integration test pass. It is able to both prove properties correct and find bugs in the driver code. In comparison with Slam there are 129 runs where Slam either times out or spaces out, where Yogi is able to give a result. The total time taken by Yogi to run over all the 5865 runs is about 32 hours on an 4 core machine, compared to over 69 hours taken by Slam.

A comparison of Yogi with Slam on 16 representative drivers is shown in Table 1. Every row of this table shows the driver, its number of lines of code, the number of properties checked and the time (in minutes) taken by Slam and Yogi along with the number of time-outs (set to 30 minutes).

Table 1. Empirical evaluation of YOGI on 16 device drivers

Program	Lines	Properties	SLAM		YOGI	
			Time-outs	Time (min)	Time-outs	Time (min)
`parport`	34196	19	1	**91.2**	0	**26.1**
`serial1`	32385	21	3	**142.4**	0	**21.5**
`serial`	31861	21	3	**203.9**	0	**28.1**
`fdc_fail`	9251	50	0	**117.6**	0	**8**
`kbdclass1`	7426	38	2	**124.9**	0	**115**
`kbdclass`	7132	36	2	**125.5**	0	**90.4**
`serenum`	6011	38	1	**95.6**	0	**10.9**
`pscr`	5680	37	0	**55**	0	**26.4**
`modem`	3467	19	0	**18**	0	**22.3**
`1394Vdev`	2757	22	2	**90.7**	0	**72.9**
`1394Diag`	2745	23	3	**121.4**	0	**68.8**
`diskperf`	2351	31	0	**36.8**	1	**100**
`incomplete1`	1558	29	0	**16**	0	**6.3**
`toastmon1`	1539	32	0	**13.5**	0	**8.4**
`toastmon`	1505	32	0	**16.6**	0	**7.6**
`daytona`	565	29	1	**106.9**	0	**77.4**

4 Current Status

YOGI is a stable and robust tool that has been run over several hundreds of thousands of lines of C code. At the tool demonstration, we will show YOGI running on small programs and demonstrate its ability to find bugs and prove programs correct. We will also present the results from running YOGI on the 5865 runs from Static Driver Verifier's integration test pass (a subset of these results are shown in Table 1).

References

1. Ball, T., Rajamani, S.K.: Automatically validating temporal safety properties of interfaces. In: Dwyer, M.B. (ed.) SPIN 2001. LNCS, vol. 2057, pp. 103–122. Springer, Heidelberg (2001)
2. Ball, T., Rajamani, S.K.: SLIC: A specification language for interface checking of C. Technical Report MSR-TR-2001-21, Microsoft Research (2001)
3. Beckman, N.E., Nori, A.V., Rajamani, S.K., Simmons, R.J.: Proofs from tests. In: ISSTA 2008: International Symposium on Software Testing and Analysis, pp. 103–122. ACM Press, New York (2008)
4. de Moura, L., Bjorner, N.: Z3: An efficient SMT solver. In: TACAS 2008: Tools and Algorithms for the Construction and Analysis of Systems (2008)
5. Godefroid, P., Klarlund, N., Sen, K.: DART: Directed automated random testing. In: PLDI 2005: Programming Language Design and Implementation, pp. 213–223. ACM Press, New York (2005)
6. Gulavani, B.S., Henzinger, T.A., Kannan, Y., Nori, A.V., Rajamani, S.K.: SYNERGY: A new algorithm for property checking. In: FSE 2006: Foundations of Software Engineering, pp. 117–127. ACM Press, New York (2006)