Practical Symbolic Verification of Regular Properties

Hengbiao Yu
College of Computer, National University of Defense Technology
Changsha, China
hengbiaoyu@nudt.edu.cn

ABSTRACT

It is challenging to verify regular properties of programs. This paper presents *symbolic regular verification* (SRV), a dynamic symbolic execution based technique for verifying regular properties. The key technique of SRV is a novel synergistic combination of property-oriented path slicing and guiding to mitigate the path explosion problem. Indeed, slicing can prune redundant paths, while guiding can boost the finding of counterexamples. We have implemented SRV for Java and evaluated it on 16 real-world open-source Java programs (totaling 270K lines of code). The experimental results demonstrate the effectiveness and efficiency of SRV.

CCS CONCEPTS

• **Software and its engineering → Software verification and validation**;

KEYWORDS

Regular property; Verification; Slicing; Guiding

ACM Reference Format:
Hengbiao Yu. 2017. Practical Symbolic Verification of Regular Properties. In *Proceedings of 2017 11th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, Paderborn, Germany, September 4–8, 2017 (ESEC/FSE'17),* 3 pages.
https://doi.org/10.1145/3106237.3121275

1 INTRODUCTION

A *regular property* is a property that can be specified by a finite state machine (FSM) [10]. In terms of property specification, regular properties are widely used in software analysis and verification (*e.g.*, model-based testing [17], typestate analysis [9], model checking [4], and performance analysis [16]). It is challenging to verify regular properties for real-world programs in software engineering.

Existing works for verifying regular properties can be divided into two categories: *static* verification and *dynamic* verification. Static methods (such as [7–9]) abstract the programs *soundly*, and carry out the verification on the abstracted models. Static verification usually achieves high code coverage, but is limited by the false alarms. In contrast, dynamic methods (such as [1, 3]), perform verification along with the program's concrete execution. Hence, every reported violation by dynamic verification is real. However, dynamic verification can only verify the program's behavior under the given inputs.

Symbolic execution [2, 11, 14] runs the program using symbolic values. It can systematically explore the program's path space through forking states or re-executing the program when encountering a branch. Given a regular property φ and program P, a transition *event* in φ's FSM corresponds to the execution of one or more statements of P. We use $Seq(p)$ to denote the generated event sequence of path p. p is a *relevant path* if $Seq(p)$ is not empty; otherwise, p is *irrelevant*. Property φ is checked along with the path exploration. If there exists a path p that $Seq(p)$ can drive the FSM of $\neg\varphi$ to an accepted state, a counterexample path is found; otherwise, P satisfies φ. Compared with static and dynamic approaches, symbolic execution achieves better precision or coverage, respectively.

To mitigate the path explosion problem — the path space increases exponentially with the number of branches in the program, we propose a scalable dynamic symbolic execution (DSE) [11, 21] based verification technique, called *symbolic regular verification* (SRV). The main intuition behind SRV lies in two aspects: (1) *w.r.t.* the regular property φ, there usually exist a large number of irrelevant paths, and many of the relevant paths are equivalent; (2). only the relevant paths with specific event sequences can violate the property φ. SRV integrates property-oriented slicing based on path slicing [12] with property guiding [23]. Slicing prunes irrelevant and equivalent relevant paths during DSE, and guiding steers DSE to find counterexample paths quickly.

We have implemented SRV for Java based on a regular property guided symbolic execution engine [23] and a dynamic slicer Javaslicer [5]. We evaluate SRV on 16 real-world open-source Java programs against representative regular properties, and the experimental results are promising.

2 SRV: SYMBOLIC REGULAR VERIFICATION

The main framework of SRV is shown in Algorithm 1. The input consists of a program P, an FSM $M_{\neg\varphi}$ corresponding to the negation of the regular property φ and an initial input I_0 to P. The candidate branches to be explored and accepted event sequences are stored in $worklist$ and $\mathcal{X}$, respectively. PC denotes the path condition, and I is the input to DSE.

SRV performs a two-staged analysis. In the first stage, Program P is statically analyzed *w.r.t.* $M_{\neg\varphi}$ through a backward data flow analysis [20] to calculate the future behavior (denoted as $Postset$) for every program location (Line 3). The $Postset$ of a location loc contains the states of $M_{\neg\varphi}$ that can reach an accepted state of $M_{\neg\varphi}$ through executing the program after loc. In the second stage, The property is checked during DSE (Lines 4-14). Additionally, SRV uses the runtime information to calculate the history information

ESEC/FSE'17, September 4–8, 2017, Paderborn, Germany

ACM ISBN 978-1-4503-5105-8/17/09…$15.00
https://doi.org/10.1145/3106237.3121275

Algorithm 1: DSE-based Regular Property Verification

```
SRV(P, M_¬φ, I_0)
Data: program P, FSM M_¬φ and an initial input I_0
1  begin
2      worklist, 𝒳 ← ∅; PC ← true; I ← I_0;
3      ComputePostset(P, M_¬φ);
4      while true do
5          (PC, p_c) ← runAndMonitor(I, M_¬φ);
6          if Seq(p_c) is accepted by M_¬φ then
7              𝒳 ← 𝒳 ∪ {LSeq(p_c)};
8              Report a counterexample path;
9          S ← Slice(P, p_c, M_¬φ);
10         saveAndPrune(worklist, S, PC);
11         if worklist = ∅ ∨ Timeout then
12             exit;
13         PC ← Select(worklist);
14         I ← Solve(PC);
```

(denoted as $Preset$) for the candidate branches. The $Preset$ of a branch b contains the states of $M_{\neg\varphi}$ that can be reached via the path from the beginning of the program to b.

Violation Detection. Property φ is checked along with path exploration (Line 5). Specifically, for an explored path p_c, we check whether $Seq(p_c)$ is accepted by $M_{\neg\varphi}$. If accepted, we add the event sequence with program location [18] information (denoted as $LSeq(p_c)$) to $\mathcal{X}$ and report p_c (Lines 6-8).

Slicing. Once a path is completed, SRV invokes the property-oriented slicing Slice to prune branches along the path (Line 9). In addition to the control and data dependence analysis used in path slicing [12], the property-oriented slicing exploits the $Preset$ and $Postset$ information to prune additional branches. Specifically, we proved that when property φ is only parametric with objects, and the sensitive objects are not data-dependent on the inputs[1], both $Preset$ and $Postset$ are sound. Hence, for such properties, SRV can also prune a branch b if one of the following conditions is satisfied: (1). the intersection of b's $Preset$ and $Postset$ is empty; (2). all the possible accepted event sequences by concatenating the event sequence before b and the possible event sequences after b belongs to $\mathcal{X}$. After slicing, $saveAndPrune$ will save the candidate branches remained in the slicing result S to the $worklist$ (Line 10).

Guiding. The size of the intersection of $Preset$ and $Postset$ is used as the main heuristic value of each branch. Select selects the branch with larger heuristic value to generate the path condition for the next iteration (Line 13). Then, A backend SMT solver (Line 14) solves the path condition to generate the inputs for the next iteration. We enhance the guiding technique [23] to support multi-object properties.

Guiding and slicing [22] are two orthogonal techniques for the verification. Slicing prunes redundant paths, while guiding helps to find counterexample paths quickly. In addition to the compatibility, they strengthen each other: slicing can boost the efficiency of guiding through pruning equivalent relevant paths, and the guiding information can help slicing to prune additional paths.

[1] A large number of regular properties satisfy this condition, such as all the properties used in the experiments.

3 EVALUATION

To evaluate SRV, we apply it to verify 16 real-world open source Java programs (shown in Table 1) *w.r.t.* representative regular properties. The properties can be classified into two categories: (1). typestate properties, such as a reader/writer cannot read/write a closed stream, the Iterator/Enumeration should call hasNext/hasMoreElement before next/nextElement, and A collection cannot be modified while being iterated[2]; and (2). user-defined properties, such as the property defined for `htmlparser` requires the input string to be the JSP format, *i.e.*, " $<\%\ldots\%>$ ". Since most programs are violation free, to further evaluate SRV, we generate three mutants [13] for the first 6 programs in Table 1 through injecting an event to a randomly selected branch. The time threshold of verification is 24 hours.

Table 1: Programs in the experiments

Program	LOC	Brief Description
rohino-a	19799	Javascript interpreter
schroeder	11092	Sampled audio editor
soot-c	32358	Static analysis editor
jlex	4400	Lexical analyzer
bloat	45357	Java bytecode optimization
bmpdecoder	531	BMP file decoder
ftpclient	2436	FTP client in Java
pobs	5488	Java parser objects
jpat	3254	Java string parser
jericho	25657	Jericho HTML Parser
nano-xml	3317	Non-validating XML parser
htmlparser	21830	HTML parser in Java
xml	5138	XML parser in Java
fastjson	20223	JSON library from alibaba
jep	42868	Mathematics library
udl	26896	UDL language library
Total	**270734**	**16 open source programs**

A verification task comprises the verified program (or the program's mutant) and the given regular property. For the total 47 verification tasks, SRV can complete[3] 39 tasks, while the default DFS, pure guiding and pure slicing can complete 30, 30 and 32 tasks, respectively. Compared with them, SRV achieves the improvements 30%, 30% and 22%, respectively. For the successfully verified 39 tasks, SRV has an average 249X, 248X, and 206X time speedups, over DFS, pure guiding and pure slicing, respectively. For the unsuccessfully verified 8 tasks, the reason is that the programs have complex control flows, causing very few paths can be pruned and the slicing is time-consuming.

4 RELATED WORK

ESP [7] is a static verifier, and achieves good scalability by merging symbolic states. The typestate verifier [9] is based on parametric

[2] The property involves multiple objects, *i.e.*, collection and iterator.
[3] A verification task completes means all the path space has been explored.

abstract domain, and adopts a staged analysis to reduce false alarms. Compared with them, SRV ensures the completeness with the help of DSE. JavaMOP [3] and Tracematches [1] are two representative runtime verification [15] methods for Java programs. While SRV is a DSE based method that can obtain higher code coverage and find more bugs. YOGI [19] integrates model checking with DSE to find real counterexamples faster. In comparison, SRV is light-weight and scalable. Woodpecker [6] uses path slicing to prune redundant paths for verifying system rules via symbolic execution. Compared with Woodpecker, SRV can prune more paths and find counterexample paths faster.

5 CONCLUSION

We have presented SRV, a practical DSE-based technique for verifying regular properties. To improve the scalability, we have introduced a synergistic combination of property-oriented slicing and guiding. Slicing prunes redundant paths, while guiding helps find counterexample paths quickly. We have implemented SRV for Java and the experimental results are promising. Future work lies in several directions: (1) techniques to further reduce the overhead of slicing (*e.g.*, better guiding strategies to generate shorter paths earlier) and (2) further improvements to our tool's usability and feasibility for releasing to and benefiting the community.

ACKNOWLEDGMENTS

This work was supported by National 973 (2014CB340703) and NSFC (61472440, 61632015, 61690203, 61532007) of China.

REFERENCES

[1] C. Allan, P. Avgustinov, A. S. Christensen, L. J. Hendren, S. Kuzins, O. Lhoták, O. de Moor, D. Sereni, G. Sittampalam, and J. Tibble. Adding trace matching with free variables to AspectJ. In *OOPSLA*, pages 345–364, 2005.

[2] C. Cadar, D. Dunbar, and D. Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, pages 209–224, 2008.

[3] F. Chen and G. Rosu. MOP: an efficient and generic runtime verification framework. In *OOPSLA*, pages 569–588, 2007.

[4] E. M. Clarke, O. Grumberg, and D. Peled. *Model checking*. MIT press, 1999.

[5] Clemens Hammacher, Martin Burger, and Valentin Dallmeier. JavaSlicer. https://www.st.cs.uni-saarland.de/javaslicer/, 2008.

[6] H. Cui, G. Hu, J. Wu, and J. Yang. Verifying systems rules using rule-directed symbolic execution. In *ASPLOS*, pages 329–342, 2013.

[7] M. Das, S. Lerner, and M. Seigle. ESP: path-sensitive program verification in polynomial time. In *PLDI*, pages 57–68, 2002.

[8] D. R. Engler, B. Chelf, A. Chou, and S. Hallem. Checking system rules using system-specific, programmer-written compiler extensions. In *OSDI*, pages 1–16, 2000.

[9] S. J. Fink, E. Yahav, N. Dor, G. Ramalingam, and E. Geay. Effective typestate verification in the presence of aliasing. In *ISSTA*, pages 133–144, 2006.

[10] A. Gill et al. Introduction to the theory of finite-state machines. 1962.

[11] P. Godefroid, N. Klarlund, and K. Sen. DART: directed automated random testing. In *PLDI*, pages 213–223, 2005.

[12] R. Jhala and R. Majumdar. Path slicing. In *PLDI*, pages 38–47, 2005.

[13] R. Just, D. Jalali, L. Inozemtseva, M. D. Ernst, R. Holmes, and G. Fraser. Are mutants a valid substitute for real faults in software testing? In *FSE*, pages 654–665, 2014.

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

[15] M. Leucker and C. Schallhart. A brief account of runtime verification. *J. Log. Algebr. Program.*, 78(5):293–303, 2009.

[16] Y. Liu, C. Xu, and S.-C. Cheung. Characterizing and detecting performance bugs for smartphone applications. In *ICSE*, pages 1013–1024, 2014.

[17] G. J. Myers, C. Sandler, and T. Badgett. *The art of software testing*. John Wiley & Sons, 2011.

[18] F. Nielson, H. R. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer, 2015.

[19] A. V. Nori, S. K. Rajamani, S. Tetali, and A. V. Thakur. The YOGI project: Software property checking via static analysis and testing. In *TACAS*, pages 178–181, 2009.

[20] T. Reps, S. Horwitz, and M. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *POPL*, pages 49–61, 1995.

[21] K. Sen, D. Marinov, and G. Agha. CUTE: a concolic unit testing engine for C. In *FSE*, pages 263–272, 2005.

[22] X. Yang, J. Wang, and X. Yi. Slicing execution with partial weakest precondition for model abstraction of C programs. *Comput. J.*, pages 37–49, 2010.

[23] Y. Zhang, Z. Chen, J. Wang, W. Dong, and Z. Liu. Regular property guided dynamic symbolic execution. In *ICSE*, pages 643–653, 2015.