RGSE: A Regular Property Guided Symbolic Executor for Java

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

Zhenbang Chen*
College of Computer, National University of Defense Technology
Changsha, China
zbchen@nudt.edu.cn

Yufeng Zhang
Jiangnan Institute of Computing Technology
Wuxi, China
yuffonzhang@163.com

Ji Wang
HPCL, College of Computer, National University of Defense Technology
Changsha, China
wj@nudt.edu.cn

Wei Dong
College of Computer, National University of Defense Technology
Changsha, China
wdong@nudt.edu.cn

ABSTRACT

It is challenging to effectively check a regular property of a program. This paper presents RGSE, a regular property guided dynamic symbolic execution (DSE) engine, for finding a program path satisfying a regular property *as soon as possible*. The key idea is to evaluate the candidate branches based on the history and future information, and explore the branches along which the paths are more likely to satisfy the property in priority. We have applied RGSE to 16 real-world open source Java programs, totaling 270K lines of code. Compared with the state-of-the-art, RGSE achieves two orders of magnitude speedups for finding the first target path. RGSE can benefit many research topics of software testing and analysis, such as path-oriented test case generation, typestate bug finding, and performance tuning. The demo video is at: https://youtu.be/7zAhvRIdaUU, and RGSE can be accessed at: http://jrgse.github.io.

CCS CONCEPTS

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

KEYWORDS

Regular property; Dynamic symbolic execution; RGSE

ACM Reference Format:
Hengbiao Yu, Zhenbang Chen, Yufeng Zhang, Ji Wang, and Wei Dong. 2017. RGSE: A Regular Property Guided Symbolic Executor for Java. 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),* 5 pages.
https://doi.org/10.1145/3106237.3122830

*Corresponding Author.

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

ACM ISBN 978-1-4503-5105-8/17/09…$15.00
https://doi.org/10.1145/3106237.3122830

1 INTRODUCTION

Regular properties are widely used in software engineering to describe program properties. In program analysis, it is a common challenge to *check whether there exists a program path satisfying a regular property*. Usually, we use finite state machines (FSMs) [11] to specify regular properties. Checking a program *w.r.t.* a regular property equals to check whether there exists a path that drives the property's the FSM to an accepted state.

Dynamic symbolic execution (DSE) [12, 25] runs a program both concretely and symbolically. DSE first uses the initial inputs to run the program, and collects the encountered constraints on symbolic variables simultaneously, called *path condition*. Then, DSE will select a branch of the path condition to negate for getting a new path constraint, which will be fed into an SMT solver to generate the inputs for the next iteration. The next run is supposed to explore the path along the selected direction. In this way, DSE can systematically explore the program's path space. DSE can be used for automatically finding the program paths satisfying a regular property. However, DSE is inherent to the path explosion problem, *i.e.*, the exponential increase of paths *w.r.t.* the number of conditional statements.

This paper presents RGSE, a regular property guided DSE engine for finding program paths satisfying a regular property *as soon as possible*. RGSE is an implementation of the technique in [31] for Java programs. The key idea is to guide symbolic execution based on the history and future information of branches, which can be calculated through dynamic analysis and static analysis, respectively. Then, the branches are prioritized *w.r.t.* the possibility of generating target paths. RGSE will select the branch with the highest priority to explore next.

We have applied RGSE to many real-world open source Java programs against representative regular properties. Compared with pure DSE and a state-of-the-art work [8] using path slicing [16] for pruning paths during symbolic execution, RGSE achieves an average >179X and >130X time speedups for finding the first target path, respectively. The experimental results demonstrate RGSE's effectiveness, efficiency, feasibility and stability.

2 REGULAR PROPERTY GUIDED DSE

Given a program $\mathcal{P}$ and a regular property φ's corresponding FSM M_φ, a transition *event* in M_φ represents the execution of one or

more statements in $\mathcal{P}$. If an execution path p cannot generate any event, we call p an *irrelevant path*, otherwise p is a *relevant path*. If p generates an event sequence accepted by M_φ, p is an *accepted path*. The key insight of regular property guided DSE is that a large portion of $\mathcal{P}$'s path space corresponds to irrelevant paths, and even for the relevant paths, only the ones with specific event sequences can be accepted by M_φ. Therefore, it is desirable for DSE to delay the exploration of irrelevant paths and relevant paths not accepted by M_φ, and explore the rest earlier.

The key idea of RGSE is to evaluate the possibility of a branch for generating an accepted path *w.r.t.* to M_φ, and explore the branches with higher possibilities in priority. RGSE uses a set of states of M_φ, called *Postset*, to denote the future behavior information of a branch. A state q in the *Postset* of a branch b indicates that q can reach an accepted state of M_φ through executing the program after b. The history behavior information of a branch is also a set of states of M_φ, called *Preset*. A state q in *Preset* of branch b means executing the path from the program entry to b can drive M_φ from the initial state to state q. RGSE uses the size of the intersection of *Preset* and *Postset* as the main heuristic value of a branch for guiding DSE, *i.e.*, exploring the branches having larger size of intersection in priority.

The *Preset* information is calculated during DSE based on the runtime information. The idea of monitoring in runtime verification [17] is used. We use (I_s, q) to represent a monitor, where I_s is the identity set of the monitored objects, and q is a state of M_φ. The size of I_s depends on the number of related objects *w.r.t.* property φ, *e.g.*, I_s only contains one element when φ is a single object property. RGSE updates the status of the monitor according to M_φ during path exploration. *Preset* is represented by a set of monitors. Note that once the status of a monitor becomes M_φ's final state, an accepted path is found. On the other hand, RGSE computes *Postset* by a static typestate analysis [3] of $\mathcal{P}$ *w.r.t.* M_φ. The details of how to calculate and utilize *Preset* and *Postset* can be referred to our prior work [31].

The method of guided DSE can benefit many research topics. For detecting typestate bugs [10][19], we can express bug patterns as regular properties and use guided DSE to find the accepted paths, *i.e.*, bugs. We can also use guided DSE to do path-oriented test case generation by expressing the targeting paths as an FSM. Besides, for verifying a regular property φ, we can use guided DSE to guide symbolic execution *w.r.t.* $\neg\varphi$ for finding counter-examples faster.

3 DESIGN AND IMPLEMENTATION

Figure 1 shows the architecture of RGSE. The inputs of RGSE are the Java bytecodes under analysis and the FSM specifying a regular property. The output is the analysis report, which contains the information including the accepted paths, the relevant paths, *etc.*

The static analyzer will first analyze the bytecode program (denoted by $\mathcal{P}$) *w.r.t.* the regular property (denoted by φ) for computing the *Postset* information. Then, $\mathcal{P}$ will be run using JPF [21], *i.e.*, a Java virtual machine (JVM). When running $\mathcal{P}$, RGSE collects the path constraint of the current path. At the same time, the dynamic analyzer uses the runtime information to calculate the *Preset* information *w.r.t.* φ. When the current path terminates, we insert the path constraint to the constraint manager, which maintains the constraint tree of $\mathcal{P}$. Then, the guided searcher will use the information produced by static and dynamic analyzers to select an unexplored branch in the constraint manager. The searcher invokes the constraint solver to solve the path constraint of the branch, generating the inputs of $\mathcal{P}$ for the next run. If there is no unexplored branch or timeout, the whole analysis is terminated and the reporter generates an analysis report. Next, we introduce each part of RGSE.

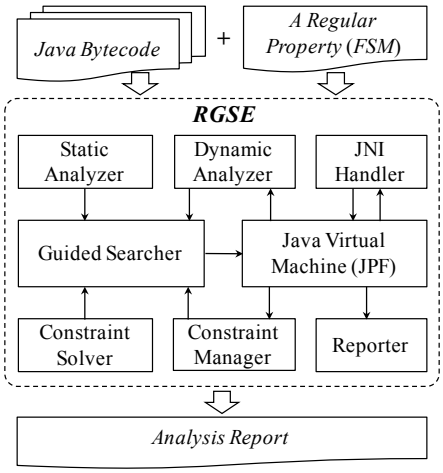

Figure 1: The architecture of RGSE.

3.1 Static Analyzer

Inspired by Clara [3], we calculate *Postset* information using a *backward* dataflow analysis. To improve the precision, we make the analysis inter-procedural based on the IDFS framework [24]. We implement the static analyzer for Java bytecode program using WALA [13], which provides a convenient IDFS framework for inter-procedural dataflow analyses and context-sensitive pointer analysis support. We can control the precision of *Postset* by the call string length of inter-procedural control flow graph. The length is exposed as a parameter of RGSE. RGSE stores the calculated *Postset* information in a cache in order for a fast access during DSE.

3.2 Core Components

The core components are the JVM, the constraint manager and the guided searcher. The JVM is in charge of running the program and collecting path constraints. We implemented the JVM based on JPF-JDart [14], which is a concolic execution engine built on JPF [21]. For each bytecode instruction, its interpretation implementation needs to include not only the concrete execution, but also the symbolic manipulations and the constraint collections of symbolic variables. In addition, for efficient multiple runs, we implemented a mechanism for state saving and restoring.

The constraint manager maintains the constraint tree of the program, which is a *binary* tree that abstracts the execution tree of the program. Each node in the tree contains the constraint of a branch statement. When a path condition PC is inserted into the tree, a node is created for the negation branch of each constraint in PC, and the node is set to be unexplored. At the original version of JPF-JDart [14], the constraint tree is stored using an array with

a fixed length, which is infeasible for real-world programs. We replaced the array with a binary tree data structure.

The guided searcher drives the path exploration of RGSE. After each run, the searcher will select an unexplored node from the constrain manager and invoke the constraint solver to solve the path constraint of the node, targeting the input generation of the next run. To improve the efficiency, we use a highly efficient sorted list [15] to store the unexplored nodes, each of which is ordered by its value. Then, the searcher can directly pick the header of the list. By assigning different values for a node, the searcher can be configured to adopt different styles of exploration, such as depth first search (DFS) and breadth first search (BFS). When the searcher is guided, the *Preset* and *Postset* information will be used for calculating a node's value.

3.3 Dynamic Analyzer

The dynamic analyzer calculates the *Preset* information using the idea of monitoring in runtime verification. We implemented the analyzer using the listener mechanism in JPF. The analyzer creates a monitor for each object whose class or interface is specified by the property. A monitor may change its state *w.r.t.* the property when one of the monitored object's methods is invoked. Once an accepted state is reached, an accepted path is found. The monitors are stored in a global set that can be accessed by the searcher.

3.4 Utility Components

The utility components are the constraint solver, the Java native interface (JNI) handler and the reporter. RGSE uses the bit-vector module in Z3 for constraint solving. We have developed the bit-vector solver adaptor for Z3 based on the constraint encoding and solving framework of SPF [22]. To improve the feasibility, RGSE utilizes JPF-nHandler [27] to alleviate the environment modeling problem. The native interface invocations are executed by the underlying JVM. In practice, with the help of the JNI handler, RGSE can analyze real-world programs including network operations, file operations, *etc.* The reporter is in charge of reporting analysis results, including the numbers of different kinds of paths (*e.g.*, accepted paths and relevant paths), the status of each path *w.r.t.* the regular property, the time for static analysis, *etc.* For accepted paths, RGSE can also report the inputs that can generate the paths. Besides, RGSE uses JFLAP [28] as the library for manipulating FSMs.

4 USAGE

To use RGSE for checking a program *w.r.t.* a regular property, we need to write an analysis driver for RGSE. The driver gives the entry point of analysis, and consists of two main modules: property specification and run configuration.

Property specification. We specify the regular property to be checked as a finite state machine (FSM). To define the property's FSM, the user need to create the states, add the transitions, and build the relationship between transition events and program statements. For example, suppose S_0 and S_1 are two created states in the FSM of file's reader property, *i.e.*, a file cannot be read while closed. Then trans.add(newFSATransition(S_0, S_1, "C")) adds a transition, *i.e.*, executing event C can drive state S_0 to state S_1, and the invocation Monitor.addMethodNameChar("close", "C") specifies that event C corresponds to the execution of file close.

Run configuration. The configuration parameters set the mode of RGSE. Specifically, the most important parameter *arguments* contains seven elements, and they represent call string bound, guiding flag, refinement flag, maximum iterations, time threshold, name of result file, and slicing flag, respectively. For example, args=newString[]{"1","1","0","−1","1, 5, 30, 0","*result.txt*","0"} indicates that RGSE will run in the guiding mode with no iteration threshold and 1 hour 5 minutes 30 seconds time threshold. Users can adjust the running mode through changing the value of arguments. More details can be found from RGSE's website.

5 EVALUATION

To demonstrate the effectiveness, efficiency and applicability of RGSE, we apply it to analyze 16 real-world open source Java programs (shown in Table 1) against representative regular properties. The properties we used can be classified into two categories: *bug properties* (shown in Table 2) and *user-defined properties*. Since most programs are violation free, we mutate the first 6 programs in Table 1 as follows: 1) collect all the conditional statements along DSE; 2) inject an event's statements, *e.g.*, a `close` operation for the `Reader` property, to a branch of a randomly selected conditional statement. We generate 3 mutants for each program. The time threshold of each analysis task is set to be 24 hours.

Table 1: Programs in the experiments

Program	LOC	Brief Description
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
rohino-a	19799	Javascript interpreter
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**

Bug properties in Table 2 are widely used in typestate analysis [10] for finding the invalid usage of resources. Note that, properties with superscript $*$ are multi-object properties. In addition, we also specify application specific properties for the last 9 programs in Table 1. For example, the user-defined property for `htmlparser` requires that the input string is in the JSP format, *i.e.*, "$<\%\ldots\%>$".

Table 2: Regular properties in the experiments

Property	Meaning
`Enumeration`	Call hasMoreElements before nextElement
`Iterator`	Call hasNext before next
	Do not update the collection while iterating*
`Reader`	Do not read a closed stream
	No read if dependent input stream closed*
`Writer`	Do not write a closed stream
	No write if dependent output stream closed*
`Socket`	Do not use a closed socket

5.1 Experimental Results

We collect the time for finding the first accepted path. In order to further evaluate RGSE, we also implemented the path slicing technique [16] used by Woodpecker [8].

Table 3: Experiment results of analysis time

Program (Property)	Type	First Violation Time(s)		
		D	S	G
schroeder (Reader)	bug_1	3.58	3.73	10.64
	bug_2	173.69	411.45	13.14
schroeder (Reader*)	bug_1	4580.73	5191.91	14.8
	bug_2	167.11	394.79	223.27
soot-c (Writer)	bug_1	18.77	85.01	103
	bug_2	17.06	81.12	107.82
	bug_3	12.79	85.17	107.94
soot-c(Writer*)	bug_1	16.85	77.25	98.8
bloat(Iterator)	O	10.93	35.04	29.26
bloat (Iterator*)	bug_1	23.77	40.14	52.05
	bug_2	28.77	40.76	44.18
bmp(Reader)	bug_1	6.82	16.82	8.96
ftpclient (Socket)	bug_1	7.11	31.47	28.55
	bug_2	8.81	32.39	28.73
jlex (Reader)	bug_1	11.66	446.35	21.79
	bug_2	NO	NO	11.64
jlex (Reader*)	bug_1	NO	NO	13.99
	bug_2	190.07	14914.13	37.01
	bug_3	33.46	1816.1	46.17
rohino-a(Enum)	O	NO	7346.56	279.32
jpat(UD)	O	NO	NO	15.53
nano-xml(UD)	O	NO	NO	14.45
pobs(UD)	O	NO	17.43	12.67
jericho(UD)	O	NO	62.44	16.11
fastjason(UD)	O	NO	NO	NO
jep(UD)	O	1429.23	21326.2	29.88
htmlparser(UD)	O	17.01	82.63	20.85
udl(UD)	O	NO	NO	2963.01
xmlparser(UD)	O	NO	NO	17.67

Table 3 lists the detailed experimental results. For clarity, we omit the tasks having no accepted path. The first column gives the analysis tasks, including the names of programs and the checked properties. Column **Type** indicates whether the program is mutated, where O denotes the original program, and bug_i represents the ith mutant. The time for finding the first accepted path is shown in column **First Violation Time(s)**, where NO means no accepted path is found after 24 hours. It is worth noting that the overload of static analysis used by slicing and guiding may make them perform worse than DFS, *e.g.*, `soot-c`. The tasks that RGSE performs best are in gray background. Compared with the pure DFS mode (D) and the pure path slicing mode (S), RGSE achieves an average 179X and 130X time speedups, respectively.

5.2 Detected Bugs and Usability

Our tool automatically detected two known bugs in `bloat` and `rohino-a`. One is a bug violating `Enumeration` property, and the other one violates `Iterator` property. Besides, most of the randomly injected typestate bugs can be successfully detected by RGSE. In addition, through analyzing the programs in Tables 1, we also find 17 runtime bugs, including array index out-of-bound, null pointer exception, division by zero, infinite loop and format exception.

To evaluate RGSE's usability, we invited a final year undergraduate student of computer science to use RGSE to analyze open-source Java programs. After a tutorial of background and usage, the student successfully analyzed six Java library programs *w.r.t.* user-defined regular properties.

6 RELATED WORK

Static analysis and dynamic analysis are two effective approaches for checking regular properties. Static analysis [3, 9, 10] enjoys high code coverage, but suffers from false positives. Dynamic analysis [1, 6] ensures the completeness, but can only cover a small portion of path space. Compared with static and dynamic approaches, RGSE employs dynamic symbolic execution, and achieves better precision or coverage.

Many approaches have been proposed to guide symbolic execution targeting different goals, such as improving the coverage [4, 5, 18, 26, 29], reaching a program location [2, 23, 30], and covering specific path space [7, 20, 23]. Different from those approaches, RGSE focuses on finding the paths satisfying a regular property as soon as possible.

The closest related work is Woodpecker [8], which employs path slicing during symbolic execution for pruning redundant paths. Woodpecker is implemented on KLEE [5] for verifying C programs *w.r.t.* system rules. We have implemented Woodpecker's method and performed the comparison in the experiments. RGSE outperforms Woodpecker significantly in finding the first accepted path.

7 CONCLUSION AND FUTURE WORK

In this paper, we present RGSE, a DSE based tool for finding program paths satisfying a given regular property. The behind technique of RGSE is from [31]. We applied RGSE to 16 real-world open source Java programs against representative regular properties. For finding the first accepted path, RGSE has on average two orders of magnitude time speedups compared with pure DSE and the DSE with path slicing. Next, we will develop new functions to improve RGSE's scalability, feasibility and usability further.

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] D. Babić, L. Martignoni, S. McCamant, and D. Song. Statically-directed dynamic automated test generation. In *ISSTA*, pages 12–22. ACM, 2011.

[3] E. Bodden. Efficient hybrid typestate analysis by determining continuation-equivalent states. In *ICSE*, pages 5–14, 2010.

[4] J. Burnim and K. Sen. Heuristics for scalable dynamic test generation. In *ASE*, pages 443–446, 2008.

[5] 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.

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

[7] M. Christakis, P. Müller, and V. Wüstholz. Guiding dynamic symbolic execution toward unverified program executions. In *ICSE*, pages 144–155, 2016.

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

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

[10] 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.

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

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

[13] IBM. T.J. Watson Libraries for Analysis (WALA). http://wala.sf.net/.

[14] K. Jayaraman, D. Harvison, V. Ganesh, and A. Kiezun. jFuzz: A concolic whitebox fuzzer for Java. In *NASA Formal Methods*, pages 121–125, 2009.

[15] Jesse Wilson. Glazed Lists Library. http://www.glazedlists.com/.

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

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

[18] Y. Li, Z. Su, L. Wang, and X. Li. Steering symbolic execution to less traveled paths. In *OOPSLA*, pages 19–32, 2013.

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

[20] P. D. Marinescu and C. Cadar. make test-zesti: A symbolic execution solution for improving regression testing. In *ICSE*, pages 716–726, 2012.

[21] C. Păsăreanu, P. Mehlitz, D. Bushnell, K. Gundy-Burlet, M. Lowry, S. Person, and M. Pape. Combining unit-level symbolic execution and system-level concrete execution for testing nasa software. In *ISSTA*, pages 15–26. ACM, 2008.

[22] C. S. Pasareanu and N. Rungta. Symbolic pathfinder: symbolic execution of java bytecode. In *ASE*, pages 179–180, 2010.

[23] S. Person, G. Yang, N. Rungta, and S. Khurshid. Directed incremental symbolic execution. In *PLDI*, pages 504–515, 2011.

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

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

[26] H. Seo and S. Kim. How we get there: a context-guided search strategy in concolic testing. In *FSE*, pages 413–424, 2014.

[27] N. Shafiei and F. van Breugel. Automatic handling of native methods in java pathfinder. In *SPIN*, pages 97–100, 2014.

[28] Susan H. Rodger and Thomas W. Finley. JFLAP - Official Site. http://www.jflap.org/.

[29] T. Xie, N. Tillmann, J. de Halleux, and W. Schulte. Fitness-guided path exploration in dynamic symbolic execution. In *DSN*, pages 359–368, 2009.

[30] C. Zamfir and G. Candea. Execution synthesis: a technique for automated software debugging. In *EuroSys*, pages 321–334, 2010.

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