Poster: Testing Heap-Based Programs with Java StarFinder

Long H. Pham* Quang Loc Le† Quoc-Sang Phan‡ Jun Sun* Shengchao Qin†

* Singapore University of Technology and Design, SG † Teesside University, UK ‡ Fujitsu Labs. of America, US

ABSTRACT

We present Java StarFinder (JSF), a tool for automated test case generation and error detection for Java programs having inputs in the form of complex heap-manipulating data structures. The core of JSF is a symbolic execution engine that uses *separation logic* with existential quantifiers and inductively-defined predicates to precisely represent the (unbounded) symbolic heap. The feasibility of a heap configuration is checked by a satisfiability solver for separation logic. At the end of each feasible path, a concrete model of the symbolic heap (returned by the solver) is used to generate a test case, e.g., a linked list or an AVL tree, that exercises that path.

We show the effectiveness of JSF by applying it on non-trivial heap-manipulating programs and evaluated it against JBSE, a state-of-the-art symbolic execution engine for heap-based programs. Experimental results show that our tool significantly reduces the number of invalid test inputs and improves the test coverage.

KEYWORDS

Symbolic execution, separation logic, test input generation

ACM Reference Format:

Long H. Pham* Quang Loc Le† Quoc-Sang Phan‡ Jun Sun* Shengchao Qin† . 2018. Poster: Testing Heap-Based Programs with Java StarFinder. In *ICSE '18 Companion: 40th International Conference on Software Engineering Companion, May 27-June 3, 2018, Gothenburg, Sweden.* ACM, New York, NY, USA, 2 pages. https://doi.org/10.1145/3183440.3194964

1 INTRODUCTION

Symbolic execution [8] is a popular automatic approach to test input generation, error detection and security vulnerability discovery. In essence, this approach takes program inputs as symbols, instead of concrete values, and computes the effects of program statements as expressions over those input symbols. This, however, is not straightforward when the program inputs are dynamically-allocated linked data structures, such as lists and trees, and thus testing and bug finding of programs with heap inputs remain a challenge.

We present Java StarFinder (JSF), a tool that aims to address the aforementioned problem. JSF is a test case generation tool for *heap-based* programs, i.e., programs with inputs in the form of complex heap-based data structures. JSF takes Java bytecode programs as inputs. It performs symbolic execution of the program, and generates JUnit test cases that achieve high coverage. The generated test cases are valid data structure instances, such as doubly linked lists or red-black trees, that satisfy the invariant predicates, often called *repOK*, of the corresponding data structures.

Related tools. The state-of-the-art approaches to handling data structures in symbolic execution are based on lazy initialization [7], which is a brute-force algorithm that considers all possible cases of a reference variable. Lazy initialization and its variants, e.g., [4, 5], do not take into account the shape of the input data structures, and thus generate too many invalid test cases. Recently, authors in [3] introduced JBSE with preconditioned lazy initialization. In particular, JBSE uses the so-called HEX logic as a specification language to describe the input data structures, and prunes off the *invalid* initialization when the specification is violated. However, we found that the HEX logic is not expressive enough to describe data structures. In this work, we present JSF, which addresses this gap with separation logic. A more detailed comparison between JBSE and JSF can be found in [11].

2 TOOL DESCRIPTION

JSF is a preconditioned symbolic execution engine that uses separation logic as the specification language to describe the input data structures. JSF combines three new features. Firstly, to express the execution of heap objects, JSF uses *separation logic* [6, 12] combined with *existential quantifiers* and *inductive definitions* to precisely represent the input data structures and the symbolic states with unbounded heap. Secondly, JSF applies lazy initialization such that uninitialized variables/fields are instantiated only when they are accessed, e.g., assigned to another variable or heap accessed, i.e., de-referenced. Especially, instead of brute-force enumeration of all possible heap objects, our initialization is *context-sensitive*; JSF only enumerates those values that satisfy the preconditions. Lastly, JSF exploits recent advances in satisfiability checking of separation logic [9, 10], which enable generating a model for each feasible symbolic heap configuration. These models are then used to generate test inputs. An in-depth discussion of technical details of JSF can be found in a companion paper [11]. JSF is a freely available open-source project: https://github.com/star-finder/jpf-star.

The architecture of JSF is depicted in Figure 1. It consists of four components: `jpf-core`, `jpf-star`, `starlib` and `S2SAT` where our current focuses are `jpf-star` and `starlib`.

jpf-core is the core of NASA's Java PathFinder (JPF) model checking platform [2]. In essence, it is a customized JVM that (concretely) executes Java bytecode. Different from a standard JVM, it allows defining non-deterministic choices during the execution, e.g., in multi-threaded programs. When there are non-deterministic choices, JPF will search all possible executions using depth-first search (by default) or other heuristics.

ICSE '18 Companion, May 27-June 3, 2018, Gothenburg, Sweden

ACM ISBN 978-1-4503-5663-3/18/05.

https://doi.org/10.1145/3183440.3194964

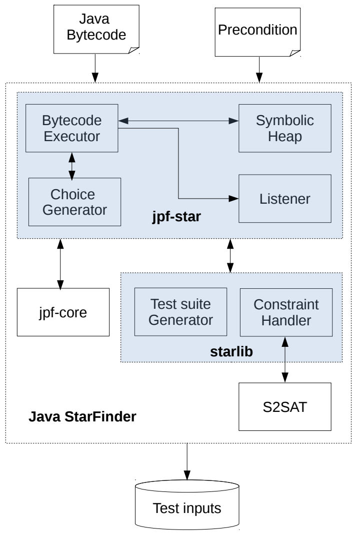

Figure 1: Architecture of Java StarFinder

jpf-star is our JPF extension for "classical" symbolic execution. It replaces the concrete execution semantics of `jpf-core` to manipulate the symbolic heap configuration. For example, when loading a reference-type variable `x` (by executing the bytecode `ALOAD`), it first checks if `x` is pointing to a heap node, or if it is constrained by an inductive definition. In the latter case, it needs to instantiate the variable to all possible *valid* cases (non-deterministic choices).

starlib is a common library to handle separation logic constraint, including parsing the precondition, unfolding the formula, and so on. It is independent of `jpf-star`, and its dependence on JPF is also minimal. So that it can be re-used in other systems, e.g., concolic execution, or re-used outside JPF.

S2SAT is a satisfiability solver for separation logic [9, 10]. It is used to check if a (symbolic) heap configuration is satisfiable. Unsatisfiable configuration means the current path is infeasible. At the end of a feasible path, the model of `S2SAT` is used to generate test input, e.g., an AVL tree, that exercises that path.

3 RESULTS

To evaluate our tool, we compare it against JBSE, a state-of-the-art symbolic execution engine for heap-based programs. We use the same benchmarks that were used to evaluate JBSE. Due to space limit, we only present the results for doubly linked list, AVL tree and red-black tree. A more throughout comparison can be found in [11].

For each generated test input, we check its validity by passing it as argument to the corresponding *repOK* method in the data structure. If *repOK* method returns true, the test input is deemed valid. We then use JaCoCo [1] to measure the branch coverage of test inputs generated by the tools.

The experimental results are shown in Figure 2. `LOC` means the number of lines of code. Columns `#Tests` show the results in form of the number of valid test inputs over the number of generated test inputs. Columns `Cov.(%)` show the coverage of valid test inputs. The results show that all test inputs generated by JSF are valid, and they achieve much higher coverage than JBSE's test inputs.

DS/Project	LOC	Method	JSF		JBSE	
			#Tests	Cov.(%)	#Tests	Cov.(%)
Doubly linked list	354	addFirst	1/1	100	2/4	100
		addIndex	6/6	100	0/18	0
		addLast	1/1	100	0/4	0
		add	1/1	100	0/4	0
		clear	2/2	100	0/8	0
		clone	2/2	100	0/2665	0
		contains	8/8	100	0/1744	0
		getFirst	3/3	100	0/5	0
		getLast	3/3	100	0/5	0
		get	4/4	100	0/18	0
		indexOf	8/8	100	0/1744	0
		lastIndexOf	8/8	100	0/1744	0
		inList	6/6	100	0/17	0
		removeFirst	2/2	100	0/9	0
		removeIndex	3/3	100	0/33	0
		removeLast	2/2	100	0/9	0
		remove	6/6	100	0/1776	0
		set	4/4	100	0/80	0
		size	1/1	100	0/1	0
		toArray	3/3	100	0/385	0
AVL tree	249	findMax	6/6	100	1/7	33.33
		findMin	5/5	100	1/7	33.33
		find	14/14	100	1/35	25
		insert	23/23	100	18/76	100
		isEmpty	4/4	100	2/2	100
		makeEmpty	1/1	100	1/1	100
		maxElement	6/6	100	7/14	100
		minElement	8/8	100	6/15	100
		printTree	2/2	100	5/18	100
Red-black tree	515	containsKey	8/8	100	10/16	100
		containsValue	25/25	100	2/2	16.67
		firstKey	4/4	100	3/8	100
		lastKey	4/4	100	3/8	100
		get	8/8	100	10/16	100
		put	142/142	100	11/26	83.67
		remove	123/123	100	13/23	35.62

Figure 2: The experimental results with JSF and JBSE

4 CONCLUSION AND FUTURE WORK

We present JSF, a tool for test input generation of heap-based programs using symbolic execution and separation logic. The experimental results show that our approach generates only valid test cases and obtain high coverage for methods of nontrivial data structures.

For future work, we might investigate machine learning and/or bi-abduction techniques to synthesize separation logic preconditions.

Acknowledgments. The first author is partially supported by the Google Summer of Code 2017 program.

REFERENCES

[1] JaCoCo Java Code Coverage Library. http://www.eclemma.org/jacoco/.
[2] Java PathFinder. http://babelfish.arc.nasa.gov/trac/jpf/.
[3] P. Braione, G. Denaro, and M. Pezzè. JBSE: A Symbolic Executor for Java Programs with Complex Heap Inputs. FSE 2016, pages 1018–1022. ACM, 2016.
[4] X. Deng, J. Lee, and Robby. Bogor/Kiasan: A K-bounded Symbolic Execution for Checking Strong Heap Properties of Open Systems. ASE '06, pages 157–166.
[5] B. Hillery, E. Mercer, N. Rungta, and S. Person. Exact Heap Summaries for Symbolic Execution. VMCAI 2016, pages 206–225, 2016.
[6] S. S. Ishtiaq and P. W. O'Hearn. BI as an assertion language for mutable data structures. POPL '01, pages 14–26. ACM, 2001.
[7] S. Khurshid, C. S. Păsăreanu, and W. Visser. Generalized symbolic execution for model checking and testing. TACAS'03, pages 553–568. Springer-Verlag, 2003.
[8] J. C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, July 1976.
[9] Q. L. Le, J. Sun, and W.-N. Chin. Satisfiability Modulo Heap-Based Programs. In *CAV*, pages 382–404. Springer International Publishing, 2016.
[10] Q. L. Le, M. Tatsuta, J. Sun, and W. Chin. A Decidable Fragment in Separation Logic with Inductive Predicates and Arithmetic. In *CAV 2017*, pages 495–517.
[11] L. H. Pham, Q. L. Le, Q. Phan, J. Sun, and S. Qin. Enhancing Symbolic Execution of Heap-based Programs with Separation Logic for Test Input Generation. *CoRR*, abs/1712.06025, 2017.
[12] J. Reynolds. Separation Logic: A Logic for Shared Mutable Data Structures. In *LICS*, pages 55–74, 2002.