JBSE: A Symbolic Executor for Java Programs with Complex Heap Inputs

Pietro Braione[§], Giovanni Denaro[§], Mauro Pezzè[§♯]

[§] Università di Milano-Bicocca
Viale Sarca, 336
Milano, Italy
{braione, denaro}@disco.unimib.it

[♯] Università della Svizzera italiana (USI)
Via Giuseppe Buffi, 13
Lugano, Switzerland
mauro.pezze@usi.ch

ABSTRACT

We present the Java Bytecode Symbolic Executor (JBSE), a symbolic executor for Java programs that operates on complex heap inputs. JBSE implements both the novel Heap EXploration Logic (HEX), a symbolic execution approach to deal with heap inputs, and the main state-of-the-art approaches that handle data structure constraints expressed as either executable programs (`repOk` methods) or declarative specifications. JBSE is the first symbolic executor specifically designed to deal with programs that operate on complex heap inputs, to experiment with the main state-of-the-art approaches, and to combine different decision procedures to explore possible synergies among approaches for handling symbolic data structures.

CCS Concepts

•Software and its engineering → Software testing and debugging; Formal software verification;

Keywords

Symbolic Execution, Heap data structures, RepOk, Heap Exploration Logic, Pointer Assertion Logic, Alloy

1. INTRODUCTION

Symbolic execution was introduced almost forty years ago [22, 8] for testing software systems, and is now a mature technique that finds relevant applications. Industrial symbolic executors like SAGE [17], Pex [31], JPF-SE [1], Apollo [2] and Klover [18] analyse x86 binaries, .Net, Java, Php and C++ programs, respectively.

Advances in constraint satisfiability and dynamic symbolic execution have improved the applicability and scalability of symbolic execution, and current symbolic execution approaches address well many of the problems that arise when symbolically executing software systems. Dynamic symbolic execution mixes symbolic with concrete execution to deal both with formulas that cannot be (efficiently) solved by a constraint solver and with the imprecision caused by the interaction with external code. Combinations of symbolic execution with control flow analysis [7], random testing [25] and evolutionary search [3] guide the exploration of the execution space to mitigate the path explosion problem. Compositional techniques compute and reuse function summaries to reduce the amount of proofs required during symbolic execution [16]. Klee and jCute incrementally solve constraints that allow for similar solutions to improve the speed of constraint solving [7, 30]. Klee [7] and SAGE [17] model pointers using the theory of arrays with selections and updates implemented by solvers to accurately model memory, and deal with pointers.

Following the impressive advances of symbolic execution and the maturity of many research and industrial symbolic executors, recent research approaches focus on the problem of efficiently executing programs that operate on complex heap inputs. To efficiently deal with heap inputs, symbolic execution needs to take into account the assumptions that characterize the dynamic data structures allocated in the heap to avoid exploring infeasible traces.

Lazy initialization approaches cope with heap inputs by enriching the path conditions with assumptions that identify the heap states that determine the execution of the different paths [21, 10, 5, 29]. Lazy initialization systematically enumerates all the possible objects that can bind to the references in the input heap, and identifies alternative path conditions for each binding. Plain lazy initialization is a brute force enumeration of the possible input heap states, and may produce many path conditions that violate assumptions of the data structures in the heap. For example, when analyzing a program that takes as input a doubly-linked list, any assumption that binds the references `next` and `previous` between consecutive list nodes in a way that breaks the mutual reachability of the nodes invalidates the path conditions.

Many approaches enrich lazy initialization with constraints that capture the properties of the input data structures, and use such properties to limit the enumeration of infeasible input states. The main approaches proposed so far either enumerate the valid (non-partial) symbolic data structures before starting symbolic execution [10, 29] or interleave symbolic execution with the evaluation of the structural properties. The evaluation approaches include both executing checking programs that encode the structural properties operationally [32] and querying some established satisfiability prover that evaluates property specifications in some

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

FSE '16 November 13–18, 2016, Seattle, WA, USA
© 2016 ACM. ISBN .
DOI:

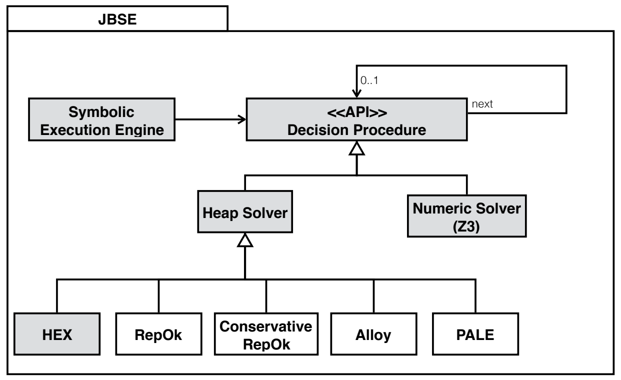

Figure 1: Logical architecture of JBSE

logic [19, 27, 28, 26, 13].

In a recent paper we introduced the Heap EXploration Logic (HEX) to efficiently address the problem of symbolically executing programs that manipulate complex input data structures in the heap. Differently from previous approaches, HEX enables to evaluate the properties of the data structures incrementally, that is, reasoning separately on any new assumption taken in the lazy initialization algorithm without having to reconsider the previous assumptions every time, with major performance improvements [6].

This demo paper presents the Java Bytecode Symbolic Executor (JBSE), a symbolic executor for Java augmented with a decision procedure for HEX, which drives the symbolic evaluation of dynamic heap data structures. JBSE currently implements the main state-of-the-art approaches that handle data structure constraints, expressed as either executable programs (`repOk` methods [23]) that check the heap state against the invariants of the data structures [10, 32], or declarative specifications in different languages, including HEX [6], Alloy [19] and the Pointer Assertion Logic (PAL) [27].

This paper presents (i) the first symbolic executor specifically designed to deal with Java programs that depend on complex heap inputs, (ii) an implementation of the HEX decision procedure that evaluates HEX properties to drive the symbolic execution, (iii) a parametric decision procedure that allows both comparative and integrated experiments with different approaches.

The remainder of this paper is organized as follows. Section 2 presents JBSE and recalls the essential elements of HEX. Section 3 introduces the demonstration artefact that provides empirical evidence of the maturity and scalability of JBSE and HEX on a set of experiments. Section 4 summarizes the contribution of the paper.

2. JBSE

JBSE is a symbolic executor specifically designed to analyze and generate test cases for software systems that take as input both numeric values and heap data structures.

Figure 1 illustrates the logical architecture of JBSE and highlights the set of decision procedures implemented within the tool. The available decision procedures, abstracted with the API `Decision Procedure`, include a classic solver for numeric constraints that JBSE instantiates with Z3 [9] and a set of *heap solvers*[1] that solve the constraints on the input heap expressed in different languages and with different checking engines:

`HEX` The decision procedure for our novel approach HEX [6]. We recall the essential elements of HEX at the end of this section.

`RepOk` The classic approach that rely on evaluating `repOk` methods to enumerate the valid instances of a data structure upon the first access to a symbolic reference that points to the data structure [10].

`ConservativeRepOk` The approach of Visser et al., who extend the `repOk` methods as executable properties that can be conservatively evaluated against partially initialised data structures [32].

`Alloy` The use of the Alloy bounded satisfiability prover [19], to verify the satisfiability of properties of the the data structure specified in Alloy, provided an upper bound to the number of objects that can be part of the initial heap.

`PALE` The use of the Pointer Assertion Logic Engine (PALE) decision procedure for unbounded problems that builds on monadic second order logic [27].

While the concepts embodied in the decision procedures based on `repOk` methods have been proposed in the context of the symbolic execution of heap data structures, the use of HEX, Alloy and PALE in the context of symbolic execution is an original contribution of JBSE.

JBSE can be configured with any subset of decision procedures, and can thus provide experimental data to comparatively evaluate the decision procedures used alone, as well as any combination of them.

When used with multiple decision procedures, JBSE links `Decision Procedure` in a chain (relation `next` in Figure 1). When refining the current path condition with a new assumption, JBSE queries the first-choice decision procedure, and accepts or discards the current symbolic state based on the satisfiability of the path condition as given by the decision procedure. If the decision procedure is inconclusive, JBSE moves to the `next` decision procedure (if any) until either obtaining a conclusive verdict or exhausting the available decision procedures.

The core novelties of JBSE are the implementation of different decision procedures according to the logic architecture illustrated in Figure 1, and the decision procedure that implements the HEX approach [6] and that we summarize below.

Figure 1 highlights the components that comprise the decision logic of JBSE when configured to work with HEX with a grey background. In this configuration, JBSE sends all satisfiability queries to the HEX heap solver, which delegates Z3 to determine the validity of the assumptions on primitive variables, and works out the path conditions when the novel assumptions predicate on references to heap objects.

While the state-of-the-art checking engines re-evaluate the consistency of the whole symbolic heap for each new assumption that emerges during symbolic execution, the HEX specifications of the structural properties of data structures

[1]The demonstration package described in Section 3 includes the documentation of the API to extend JBSE with additional heap solvers.

$$\begin{aligned}&\mathit{any}.\texttt{root}(.\texttt{left}|.\texttt{right})* \ \mathit{instanceof}\ \texttt{BTNode}\\ &\quad \mathit{aliases\ nothing} && (1)\\ &\mathit{any}.\texttt{root}(.\texttt{left}|.\texttt{right})+.\texttt{parent}\ \mathit{instanceof}\ \texttt{BTNode}\\ &\quad \mathit{aliases\ ref.up.up} && (2)\\ &\mathit{any}.\texttt{root}(.\texttt{left}|.\texttt{right})*.\texttt{parent}\ \mathit{instanceof}\ \texttt{BTNode}\\ &\quad \mathit{expands\ to\ nothing} && (3)\\ &\mathit{any}.\texttt{root}(.\texttt{left}|.\texttt{right})+.\texttt{parent}\ \mathit{instanceof}\ \texttt{BTNode}\\ &\quad \mathit{not\ null} && (4)\\ &\mathit{any}.\texttt{root}.\texttt{parent}\ \mathit{instanceof}\ \texttt{BTNode}\\ &\quad \mathit{aliases\ nothing} && (5)\end{aligned}$$

Figure 2: HEX properties of a binary tree

can be checked incrementally against each newly taken assumption. Thus, the HEX checking engine works separately on each assumption, with significant performance gains.

Figure 2 illustrates the core aspects of HEX through the example specification of the properties of a binary tree data structure. Each property specification consists of (i) a regular expression pattern over field names to identify references to heap objects, (ii) an object type that indicates the objects which the property applies to, and (iii) a constraint on the possible initializations of the references that match with the pattern and the object type.

In Figure 2, the pattern *any*.`root`.(`left`|.`right`)∗ in property (1) identifies any reference that is reachable from the root node of the tree (field `root`) by traversing some sequence of children (a chain of accesses through fields `left` and `right`), and that refers to an object of type `BTNode`. The pattern states that these references cannot be alias of other references (*alias nothing*); thus excluding data structures that are not trees.

The other properties encode the assumptions that characterize the parent relation in a tree: The `parent` reference must always be an alias of the immediate predecessor of its owner object (*aliases ref.up.up* in property (2)), can never be a newly assumed object (*expands to nothing* in property (3)), and cannot be `null` (*not null* in property (4)) with the exception of the `parent` of the root node that cannot accept any alias (*aliases nothing* in property (5)) and thus must be `null`.

The `HEX` decision procedure checks for the validity of assumptions on the references in the input heap by checking whether the references involved in the assumptions match both the pattern and the object type of any property, and discards the matching references that contradict the initialization constraints.

3. DEMONSTRATION ARTEFACT

The `JBSE` demonstration artefact presents empirical data about the maturity and scalability of `JBSE` to analyze nontrivial Java programs that operate on complex heap data structures. The artefact provides evidence of the effectiveness and efficiency of the `JBSE` symbolic executor and the `HEX` heap solver to symbolically execute programs that operate on various types of heap data structures, including classic recursive data structures commonly used in many application domains and application specific data structures. It also shows the relative strengths and weaknesses of the different heap solvers integrated in `JBSE`. Both `JBSE`[2] and the demonstration artefact[3] are available as open source.

The demonstration artefact includes 274 experiments that challenge `JBSE` to analyze classes that manipulate classic recursive data structures provided as input in the heap, and 3 experiments that illustrate the ability of `JBSE` to analyze components of open source projects. The classic recursive data structures considered in the experiments are Java classes that implement doubly-linked lists and balanced (both red-black and AVL) trees [12, 14]. The open source components considered in the experiments are TSAFE and two components of Google Closure-compiler project.

TSAFE is the Tactical Separation Assisted Flight Environment, an air traffic control application [11]. Its inputs include data structures that record the position and the movements of the aircraft in the controlled space. TSAFE has been a popular benchmark in the community for many years now. The demonstration artefact provides empirical evidence of the effectiveness of `JBSE` to analyse a set of correctness properties of the TSAFE system.

The Google Closure-compiler project is a popular tool that operates on parse tree inputs for optimizing JavaScript code.[4] The demonstration artefact provides empirical evidence of the effectiveness of `JBSE` to generate test cases that reveal known faults for two different versions of the Google Closure-compiler project, which we refer to as *Closure37* and *Closure72*, respectively [20].

The experiments with the classic recursive data structures illustrate the ability of `JBSE` to deal with complex constraints of different nature, the experiments with the open source components provide data about the behavior of `JBSE` in the presence of software of growing size. The classes that implement the classic data structure cumulatively amount to 1,048 lines of code, while the classes analyzed in TSAFE, Closure37 and Closure72 amount to 607, 7,972 and 5,972 lines of code, respectively. The artefact includes the specification of the properties of all data structures referred in the experiments in all the languages accepted by the heap solvers implemented in `JBSE`.

The artefact demonstrates the contribution of `JBSE` for the analysis of programs that operates on complex heap data structures with three sets of experiment pools: the *classic symbolic execution*, the *HEX symbolic execution* and the *comparative symbolic execution pool*.

The *classic pool* illustrates the problems and limitations of classic symbolic execution when dealing with complex heap data structures. It considers the $\texttt{JBSE}_{Z3}$ configuration, where the decision procedure is instantiated with the Z3 numeric solver only, without any heap solver. The $\texttt{JBSE}_{Z3}$ configuration is a classic symbolic executor that implements (unconstrained) lazy initialization to enumerate the heap configurations that determine the execution of the different program paths. The experiments show that $\texttt{JBSE}_{Z3}$ analyzes massive amounts of infeasible executions, largely hampering performance and scalability.

The *HEX pool* illustrates the effectiveness and efficiency of `JBSE` and HEX. It considers the $\texttt{JBSE}_{HEX}$ configuration, where the decision procedure is instantiated with the `HEX` heap solver paired with Z3, and indicates that the `HEX` heap

[2]http://pietrobraione.github.io/jbse
[3]https://sites.google.com/site/pietrobraione/fse2016
[4]https://code.google.com/p/closure-compiler

Table 1: Feasible/Infeasible execution traces computed with JBSE combined with the different heap solvers

	`HEX`		`RepOk`		`ConsRepOk`		`Alloy`		`PALE`	
Subject	**F**	**I**	**F**	**I**	**F**	**I**	**F**	**I**	**F**	**I**
TSAFE	290	**0**	560	1,690	290	840	290	**0**	n.a.*	
Closure37	2,856	**0**	6,016	90,240	2,856	3,570	2,856	**0**	n.a.*	
Closure72	925	**0**	1,110	25,850	925	24,120	925	**0**	n.a.*	
Data structures	42,966	**0**	329,185	**0**	41,227	447,350	41,227	**0**	1,768	616

*PALE does not support cyclic graphs of objects and thus cannot model the invariants for this data type

Table 2: Absolute/relative execution time in minutes of JBSE combined with the different heap solvers

	`HEX`	`RepOk`		`ConsRepOk`		`Alloy`		`PALE`	
Subject	**Time**	**Time**	`/HEX`	**Time**	`/HEX`	**Time**	`/HEX`	**Time**	`/HEX`
TSAFE	**2**	9	(4×)	52	(26×)	56	(28×)	n.a.*	
Closure37	**7**	154	(22×)	41	(6×)	306	(44×)	n.a.*	
Closure72	**2**	12	(6×)	251	(125×)	17	(8×)	n.a.*	
Data structures	**26**	43	(2×)	1322	(51×)	1881	(72×)	1412	(54×)

*PALE does not support cyclic graphs of objects and thus cannot model the invariants for this data type

solver effectively selects only the feasible execution traces, thus avoiding wasting the analysis budget on infeasible executions. The experiments confirm that $\texttt{JBSE}_{HEX}$ analyzes only the feasible executions with massive performance improvements with respect to $\texttt{JBSE}_{Z3}$.

The *comparative pool* compares the heap solvers implemented in `JBSE`, namely, `HEX`, `RepOk`, `ConservativeRepOk`, `Alloy` and `PALE`. Table 1 and Table 2 summarize the results of the *comparative pool.*[5] Table 1 reports the amount of both feasible and infeasible execution traces (columns *F* and *I*, respectively) that `JBSE` computes in each experiment when configured with any of the available heap solvers, and Table 2 reports the time that `JBSE` takes to complete the execution of each experiment as both the absolute number of minutes (columns *Time*) and the relative time with respect to HEX (columns */HEX*).

In both tables, rows *TSAFE*, *Closure37* and *Closure72* report the data computed during the analysis of the 3 open source components when `JBSE` is configured with the different heap solvers, with the exception of PALE, which does not support cyclic graphs of objects. Rows *Data structures* report the data cumulatively achieved across the 274 experiments with the recursive data structures.

The data in Table 1 confirm that HEX and `Alloy` analyze only feasible executions, while the other approaches (`RepOk`, `ConservativeRepOk` and `PALE`) analyze many infeasible executions. The data also indicate that the `RepOk` approach, which naïvely evaluates the invariants of the data structures at the beginning of the symbolic execution of the target programs, results in analysing significantly larger amounts of feasible executions than the other approaches. The reason is that the `RepOk` approach suffers from over-constraining the symbolic inputs, that is, it fosters the analysis of multiple distinct assumptions on objects and fields that are not accessed in the programs. HEX suffers from limited over-constraining problems only in the red-black tree experiment, where it augments the valid symbolic states with assumptions on the red/black status of the internal data nodes, to assist the incremental evaluation of the invariants.

Table 2 shows the execution time of `JBSE` when combined with the different heap solvers. These data confirm the relevant improvement of HEX over all alternative approaches.

[5]The detailed results of all the experiments are included in the demonstration package.

In summary, our experiments indicate that only `HEX` and `Alloy` are fully precise, that is, they do not select infeasible executions. `RepOk` and `ConservativeRepOk` fail to cope with complex invariants involving more than one input data structure, while `PALE` cannot specify circular data structures and cannot reason on relations between object structures and numeric fields. `HEX` outperforms all other approaches in efficiency, confirming the benefits of incrementally checking constraints as in HEX rather than re-evaluting the validity of the whole heap at each step. In particular `HEX` is from 8 to 72 times quicker than `Alloy`, the only other precise approach. The readers must also notice that the performance of `Alloy` depends on scope bounds of the Alloy analyzer. Determining the optimal scope bounds, that is, the smallest scope bound that preserves precision, is a very expensive human activity that the data reported in the figure do not consider. The demonstration artefact uses optimal bounds for all programs, bounds that we found after many explorative runs of the symbolic executor, but such a fine tuning is hardly sustainable in practice.

4. CONCLUSIONS

This demo paper presents `JBSE`, the first symbolic executor specifically designed to deal with programs that operate on heap data structures. `JBSE` provides the first implementation of the HEX symbolic execution approach, and provides the support for experimenting with different approaches to identify symbolic data structures, as well as for combining different decision procedures and explore possible synergies.

The demo package provides a framework for evaluating `JBSE` and the HEX approach both in isolation and in comparison with `RepOk`, `ConservativeRepOk`, `Alloy` and `PALE`, and includes data that confirm our research hypothesis about the efficiency of incrementally checking the constraints that characterize complex heap inputs.

5. REFERENCES

[1] S. Anand, C. S. Păsăreanu, and W. Visser. JPF-SE: A symbolic execution extension to Java PathFinder. In *Proceedings of the International Conference on Tools and Algorithms for Construction and Analysis of Systems*, TACAS '07, pages 134–138. Springer, 2007.

[2] S. Artzi, A. Kiezun, J. Dolby, F. Tip, D. Dig, A. Paradkar, and M. D. Ernst. Finding bugs in dynamic web applications. In *Proceedings of the International Symposium on Software Testing and Analysis*, ISSTA '08, pages 261–272. ACM, 2008.

[3] A. Baars, M. Harman, Y. Hassoun, K. Lakhotia, P. McMinn, P. Tonella, and T. Vos. Symbolic search-based testing. In *Proceedings of the International Conference on Automated Software Engineering*, ASE '11, pages 53–62. IEEE Computer Society, 2011.

[4] A. Bouajjani, C. Drăgoi, C. Enea, and M. Sighireanu. Accurate invariant checking for programs manipulating lists and arrays with infinite data. In S. Chakraborty and M. Mukund, editors, *Automated Technology for Verification and Analysis*, LNCS, pages 167–182. Springer, 2012.

[5] P. Braione, G. Denaro, and M. Pezzè. Enhancing symbolic execution with built-in term rewriting and constrained lazy initialization. In *Proceedings of the 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, pages 411–421. ACM, 2013.

[6] P. Braione, G. Denaro, and M. Pezzè. Symbolic execution of programs with heap inputs. In *Proceedings of the European Software Engineering Conference held jointly with the ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ESEC/FSE '15, pages 602–613, 2015.

[7] C. Cadar, D. Dunbar, and D. Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the Symposium on Operating Systems Design and Implementation*, OSDI '08, pages 209–224. USENIX Association, 2008.

[8] L. A. Clarke. A program testing system. In *Proceedings of the 1976 Annual Conference*, ACM '76, pages 488–491. ACM, 1976.

[9] L. De Moura and N. Bjørner. Z3: An efficient SMT solver. In *Proceedings of the International Conference on Tools and Algorithms for Construction and Analysis of Systems*, TACAS/ETAPS '08, pages 337–340. Springer, 2008.

[10] X. Deng, J. Lee, and Robby. Bogor/Kiasan: A k-bounded symbolic execution for checking strong heap properties of open systems. In *Proceedings of the International Conference on Automated Software Engineering*, ASE '06, pages 157–166. ACM, 2006.

[11] G. D. Dennis. TSAFE: Building a trusted computing base for air traffic control software. Master's thesis, Massachusetts Institute of Technology, 2003.

[12] 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.

[13] C. Enea, V. Saveluc, and M. Sighireanu. Compositional invariant checking for overlaid and nested linked lists. In *Proceedings of the European Conference on Programming Languages and Systems*, ESOP'13, pages 129–148. Springer-Verlag, 2013.

[14] J. P. Galeotti, N. Rosner, C. G. López Pombo, and M. F. Frias. Analysis of invariants for efficient bounded verification. In *Proceedings of the International Symposium on Software Testing and Analysis*, ISSTA '10, pages 25–36. ACM, 2010.

[15] J. Geldenhuys, N. Aguirre, M. Frias, and W. Visser. Bounded lazy initialization. In G. Brat, N. Rungta, and A. Venet, editors, *NASA Formal Methods*, LNCS 7871. Springer Berlin Heidelberg, 2013.

[16] P. Godefroid. Compositional dynamic test generation. In *Proceedings of the Symposium on Principles of Programming Languages*, POPL '07, pages 256–267. ACM, 2007.

[17] P. Godefroid, M. Y. Levin, and D. Molnar. Sage: Whitebox fuzzing for security testing. *ACM Queue*, 10(1):20–27, 2012.

[18] I. G. Guodong Li and S. Rajan. Klover: a symbolic execution and automatic test generation tool for C++ programs. In *Proceedings of the International Conference on Computer Aided Verification*, CAV '11, pages 53–68. Springer, 2011.

[19] D. Jackson. *Software Abstractions: Logic, Language, and Analysis.* The MIT Press, 2006.

[20] R. Just, D. Jalali, and M. D. Ernst. Defects4J: A database of existing faults to enable controlled testing studies for Java programs. In *Proceedings of the International Symposium on Software Testing and Analysis*, ISSTA '14, pages 437–440. ACM, 2014.

[21] S. Khurshid, C. S. Păsăreanu, and W. Visser. Generalized symbolic execution for model checking and testing. In *Tools and Algorithms for Construction and Analysis of Systems*, LNCS 2619. Springer, 2003.

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

[23] B. Liskov and J. Guttag. *Program Development in Java: Abstraction, Specification, and Object-Oriented Design.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 2000.

[24] P. Madhusudan and X. Qiu. Efficient decision procedures for heaps using STRAND. In *Proceedings of the 18th International Conference on Static Analysis*, SAS'11, pages 43–59. Springer-Verlag, 2011.

[25] R. Majumdar and K. Sen. Hybrid concolic testing. In *Proceedings of the International Conference on Software Engineering*, ICSE '07, pages 416–426. IEEE Computer Society, 2007.

[26] S. McPeak and G. C. Necula. Data structure specifications via local equality axioms. In *Proceedings of the 17th International Conference on Computer Aided Verification*, CAV'05. Springer-Verlag, 2005.

[27] A. Møller and M. I. Schwartzbach. The Pointer Assertion Logic Engine. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*, PLDI '01, pages 221–231, 2001.

[28] Z. Rakamarić, R. Bruttomesso, A. J. Hu, and A. Cimatti. Verifying heap-manipulating programs in an SMT framework. In *Proceedings of the 5th International Conference on Automated Technology for Verification and Analysis*, ATVA'07, pages 237–252. Springer-Verlag, 2007.

[29] N. Rosner, J. Geldenhuys, N. Aguirre, W. Visser, and M. F. Frias. BLISS: improved symbolic execution by bounded lazy initialization with SAT support. *IEEE Transactions on Software Engineering*, 41(7):639–660,

2015.

[30] K. Sen and G. Agha. CUTE and jCUTE: Concolic unit testing and explicit path model-checking tools. In *Proceedings of the International Conference on Computer Aided Verification*, CAV '06, pages 419–423. Springer, 2006.

[31] N. Tillmann and J. de Halleux. Pex: White box test generation for .NET. In *Proceedings of the International Conference on Tests and Proofs*, TAP '08, pages 134–153. Springer, 2008.

[32] W. Visser, C. S. Păsăreanu, and S. Khurshid. Test input generation with java pathfinder. In *Proceedings of the International Symposium on Software Testing and Analysis*, ISSTA '04, pages 97–107. ACM, 2004.

[33] G. Yorsh, A. Rabinovich, M. Sagiv, A. Meyer, and A. Bouajjani. A logic of reachable patterns in linked data-structures. In *Proceedings of the 9th Conference on Foundations of Software Science and Computation Structures*, FOSSACS'06. Springer-Verlag, 2006.