Kex at the 2021 SBST Tool Competition

Azat Abdullin*, Marat Akhin† and Mikhail Belyaev‡
Peter the Great St. Petersburg Polytechnic University
JetBrains Research
Saint Petersburg, Russia
Email: {*abdullin, †akhin, ‡belyaev}@kspt.icc.spbstu.ru

***Abstract*—Kex is an automatic white-box test generation tool for Java programs, which is able to generate executable test suites (as JUnit test suites) aiming to satisfy the branch coverage criterion. It uses symbolic execution to analyze control flow graphs of the program under test (PUT) and produces interesting symbolic inputs for each basic block of PUT. Kex then feeds these symbolic inputs to an original backward-search based algorithm called Reanimator, which generates executable JUnit test cases satisfying the symbolic inputs. This paper summarizes the results and experiences of Kex participation in the ninth edition of the Java unit testing tool competition at the International Workshop on Search-Based Software Testing (SBST) 2021.**

***Index Terms*—automatic test generation, symbolic execution, software testing**

I. Introduction

This paper discusses the results obtained by the Kex [1] tool on the benchmarks of ninth edition of Java unit testing tool competition at the International Workshop on Search-Based Software Testing (SBST) 2021, where Kex received a score of 44.21 and ranked fifth. The full report on the competition can be found in [2].

II. Kex

Kex is an automatic white-box [3] unit test generation tool for Java [1], which is mainly usable as a command-line tool. Table I summarizes the features of Kex in the standard format of SBST tool competition. As an input, Kex accepts a program under test (PUT) in a form of compiled JVM bytecode files and generates a test suite aiming to reach full branch coverage; tests are emitted as JUnit 4 test classes.

Processing the PUT in the form of compiled bytecode files is done using Kfg[1] library, which helps with analysis and construction of PUT control flow graphs (CFG). Kfg is also used to do various CFG transformations, e.g., loop unrolling or inlining, to help with the subsequent analyses.

Kex works as a symbolic execution engine and uses SMT solvers to perform the constraint solving. For each CFG of the target program Kex builds its own intermediate representation called *predicate state*. Predicate state serves as an inter-layer between Kfg and SMT formulae, allowing it to easily support multiple SMT solvers (such as Boolector [4], Z3 [5] or STP [6]). It is also used to perform additional, SMT-specific transformations. In the SBST 21 Java unit testing tool competition Kex was configured to use the Z3 solver.

[1]https://github.com/vorpal-research/kfg

TABLE I
Classification of the Kex unit test generation tool

Prerequisites	
Static or dynamic	Combined
Software type	JVM bytecode
Lifecycle phase	Unit testing for Java programs
Environment	Java 8
Knowledge required	JUnit 4
Experience required	Basic unit testing knowledge
Input and Output of the tool	
Input	Bytecode of the PUT and dependencies
Output	JUnit 4 test cases
Operation	
Interaction	Through the command line
User guidance	–
Source of information	https://github.com/vorpal-research/kex
Maturity	Research prototype, under development
Technology behind the tool	Symbolic execution
Obtaining the tool and information	
License	Apache 2.0
Cost	Open source
Support	None
Empirical evidence about the tool	
–	

Kex works at a basic block level of CFG[2]. At each step it selects a previously uncovered basic block from the CFG and tries to generate a symbolic input which will cover it. If the generation succeeds, Kex feeds the generated symbolic input into its test generation component called Reanimator. It is based on a original backward search algorithm that tries to find a sequence of actions to generate the target object, while using symbolic execution to infer the effects these actions have on the object state. Given an object representation (in any form), Reanimator attempts to generate a valid code snippet which constructs this object *using its publicly available API*.- These code snippets are then used as test cases for PUT.

Main practical limitations of Kex come from the underlying techniques it uses for analysis and test case generation. Symbolic execution is known to be limited in its ability to analyze large-scale programs, as it encounters the state explosion problem. One may attempt to overcome this problem by underapproximating the program behaviour (e.g., through loop unrolling), but this trade-offs performance for precision. Another limitation of Kex is the test generation process. During the analysis phase Kex can produce symbolic inputs

[2]Branch coverage is achieved by generating placeholder basic blocks for empty if/else branches.

TABLE II
RESULTS OF KEX ON THE CONTEST BENCHMARKS (30 SECONDS TIME BUDGET)

Benchmark	Line cov.	Cond. cov.	Mut. cov.	Mut. kill
GUAVA-61	2,25	0,00	1,60	0,00
GUAVA-200	13,06	13,27	18,91	18,91
GUAVA-118	0,00	0,00	0,00	0,00
GUAVA-226	1,59	0,00	1,35	1,35
GUAVA-128	56,69	1,35	1,58	0,00
GUAVA-11	46,81	41,18	58,54	58,54
GUAVA-181	70,69	64,71	92,16	92,16
GUAVA-134	2,30	0,00	1,30	1,30
GUAVA-231	6,00	0,00	0,00	0,00
GUAVA-232	15,71	11,83	0,00	0,00
GUAVA-108	9,59	4,80	2,31	1,98
GUAVA-71	14,29	0,00	0,00	0,00
GUAVA-273	5,47	1,85	4,76	0,00
GUAVA-96	3,21	0,00	2,70	2,70
GUAVA-82	64,08	66,18	73,68	73,68
GUAVA-46	46,23	27,55	53,13	37,19
GUAVA-192	0,37	0,00	0,00	0,00
GUAVA-199	6,67	0,00	0,00	0,00
GUAVA-213	7,14	0,00	0,00	0,00
GUAVA-148	23,08	4,17	12,77	12,77
GUAVA-156	2,06	0,00	0,89	0,89
GUAVA-227	6,25	0,00	0,00	0,00
GUAVA-237	34,63	37,33	39,25	3,88
GUAVA-254	48,84	42,86	51,52	51,52
GUAVA-267	24,23	28,21	25,97	25,97
SEATA-11	0,00	0,00	0,00	0,00
SEATA-27	0,00	0,00	0,00	0,00
SEATA-6	0,00	0,00	0,00	0,00
SEATA-5	0,00	0,00	0,00	0,00
SEATA-25	0,00	0,00	0,00	0,00
SEATA-28	0,00	0,00	0,00	0,00

which are not constructible using the public API of PUT, and Reanimator will not be able to convert such inputs to tests.

III. BENCHMARK RESULTS

Table II reports the results of Kex on two of the six benchmark projects (*guava* and *seata*) for the 30 seconds time budget. Competition organizers reported that Kex failed to produce any test cases for the other four projects. For each benchmark program, the table reports the score in terms of the line coverage, condition coverage, mutant coverage and mutant kill ratio, averaged over 6 experiment runs.

Data in table II shows Kex has not performed really well in the competition. It has failed to analyze four out of six projects and was able to achieve any coverage only on *guava*. This problem can be explained by its relatively low degree of maturity, as this was the first time Kex was used as a standalone off-the-shelf tool, and not as a research prototype. Thus, its implementation issues has limited its ability to perform on most of the competition benchmark. At the current moment, we cannot give a more detailed explanation of the Kex downsides on the benchmark projects.

For the *guava* project, however, we can see Kex results average with about 20% line coverage and 14% branch coverage. These numbers are more competitive with the results of other participating tools, but still require further analysis. As currently we do not have access to full experimental data, we decided to manually analyze the source code of *guava* project on those benchmarks where Kex has performed worse than the average (i.e., achieving less than 20% line coverage). We found out that these benchmarks contain features not currently supported by Reanimator, such as abstract classes or non-static inner classes. Some of the benchmark classes also use Java reflection types, which Kex currently cannot generate.

We can summarize our insights from the contest as follows.

- Kex has some implementation issues which affect its reliability and robustness on complex PUT;
- Both Kex and Reanimator currently do not support some of the more complex language features (e.g., abstract classes, inner classes, reflection);
- When not constrained by its limitations, Kex can generate test suites with an average of 20% code coverage under the 30 seconds time budget.

The competition has revealed a number of Kex weaknesses both in terms of its reliability and test generation ability. We are going to continue working on these weaknesses to get a more mature and stable automatic test generation tool.

IV. CONCLUSION

This paper reports on the participation of the Kex test generation tool in the ninth SBST Java Unit Testing Tool Contest. With an overall score of 44.21, Kex was ranked fifth. Benchmark used in the contest pointed out several issues and limitations of Kex, which would help us find ways to improve its performance. The benchmark infrastructure will allow us to continue testing the tool and prepare it for the future contests.

REFERENCES

[1] Kex. [Accessed 05.03.2021]. [Online]. Available: https://github.com/vorpal-research/kex/tree/sbst-21

[2] S. Panichella, A. Gambi, F. Zampetti, and V. Riccio, "Sbst tool competition 2021," in *International Conference on Software Engineering, Workshops, Madrid, Spain, 2021*. ACM, 2021.

[3] P. Godefroid, M. Y. Levin, D. A. Molnar *et al.*, "Automated whitebox fuzz testing." in *NDSS*, vol. 8, 2008, pp. 151–166.

[4] R. Brummayer and A. Biere, "Boolector: An efficient SMT solver for bit-vectors and arrays," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2009, pp. 174–177.

[5] L. De Moura and N. Bjørner, "Z3: An efficient SMT solver," in *Tools and Algorithms for the Construction and Analysis of Systems*. Springer-Verlag, 2008, pp. 337–340.

[6] V. Ganesh and T. Hansen. STP constraint solver: Simple theorem prover SMT solver. [Accessed 25/02/2021]. [Online]. Available: https://stp.github.io/