Kex at the SBFT 2023 Java Tool Competition

1st Azat Abdullin
JetBrains Research
Paphos, Cyprus
azat.abdullin@jetbrains.com

2nd Marat Akhin
JetBrains Research
Amsterdam, The Netherlands
marat.akhin@jetbrains.com

***Abstract*—Kex is a platform for analysis of JVM programs which mainly focuses on automatic test generation with the aim to maximize branch coverage criterion. Kex takes a set of compiled classes as an input and uses symbolic execution to analyze control flow graphs of the program under test (PUT). Symbolic engine produces a set of interesting inputs which are converted into a JUnit 4 test suite using Java reflection library. Kex can generate tests in fully static mode without running any actual code (Kex-symbolic) and in concolic mode (Kex-concolic) which combines symbolic and concrete executions. This paper summarizes the results and experiences of Kex-symbolic and Kex-concolic participation in the eleventh edition of the Java unit testing tool competition at the International Workshop on Search-Based and Fuzz Testing (SBFT) 2023.**

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

I. Introduction

This paper discusses the results obtained by two versions of Kex [1], Kex-symbolic and Kex-concolic, on the benchmarks of the eleventh edition of the Java unit testing tool competition at the International Workshop on Search-Based and Fuzz Testing (SBFT) 2023. Kex-symbolic received a final score of 4.89 and ranked fifth, while Kex-concolic ranked fourth with a score of 3.92. The full report on the competition can be found in [2].

II. Tool description

Kex[1] is a platform for analysis of Java bytecode, which supports loading Java bytecode, building an intermediate model and analyzing it via SMT solvers [3]. It mainly focuses on automatic test generation, but can be used for other things, such as crash reproduction, program analysis and repair. Kex is mainly used as a standalone command-line tool. Table I summarizes the features of Kex in the standard format of SBFT tool competition. As an input, Kex accepts the program under test (PUT) in the form of compiled JVM bytecode files and generates a test suite aiming to reach full branch coverage; these tests are emitted as JUnit 4 test classes. Users can also provide additional information to guide the test generation process of Kex through the intrinsics library[2].

Processing the PUT in the form of compiled bytecode files is performed using Kfg[3] library, which helps with construction and analysis of PUT control flow graphs (CFG). Kfg is also used to perform code instrumentation for the concolic mode.

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 experience
Input and Output of the tool	
Input	Bytecode of the PUT
Output	JUnit 4 test cases
Operation	
Interaction	Through the command line
User guidance	Through the intrinsics library
Source of information	github.com/vorpal-research/kex
Maturity	Research prototype
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	
Platform description	see [1]

Kex works as a symbolic execution engine and uses SMT solvers to perform the constraint solving. For each path of the control flow graph, 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. This year we configured Kex to use Z3 [4] SMT solver via KSMT [5] wrapper library.

Another important part of Kex symbolic engine is Kex-rt [4]. Kex-rt is a standard library approximation designed specifically for Kex. It uses Kex intrinsics to create a simplified representation for the more complex and important parts of Java standard library (e.g. collections, primitive wrappers, etc.). This allows Kex to better and easier analyze code that heavily uses Java standard library classes.

A. *Kex-symbolic*

Kex-symbolic is an implementation of the more-or-less traditional automatic test generation based on symbolic execution [6]. Kex-symbolic works at the basic block level of control flow graphs. At each step it selects one of the traversed

[1]https://github.com/vorpal-research/kex
[2]https://github.com/vorpal-research/kex-intrinsics
[3]https://github.com/vorpal-research/kfg

[4]https://github.com/AbdullinAM/kex-rt

paths of the PUT and tries to explore it one basic block further. Each instruction of a basic block is also checked for possible errors (e.g., null pointer exceptions, index out of bound exceptions). When Kex-symbolic encounters a method call, it resolves all possible candidate methods which can be called at that program point (using the type information collected on the current symbolic execution path).

A big part of every symbolic execution engine is their path selection module, i.e., a module that decides what paths are considered at each step. There are a lot of various path selection algorithms for symbolic execution. Kex-symbolic was developed as a proof-of-concept prototype and currently only supports breadth-first search.

When Kex-symbolic reaches an error-triggering path or the end of a program, it invokes an SMT solver and generates input data which satisfies the explored path. That input data is then converted into a JUnit 4 test case. The specific peculiarities of test cases generated by Kex are discussed in section II-C.

B. Kex-concolic

Kex-concolic is an extension of Kex-symbolic that combines symbolic execution capabilities with concrete execution [7]. At each step of analysis Kex-concolic chooses one uncovered branch from the execution tree and tries to generate a test case that covers it. Generated test case is then executed to ensure that the branch has been covered, if this is true, the new execution path is added to the program execution tree. During the concrete step Kex-concolic also collects the symbolic execution trace of the program, which is then used to explore new paths along that execution. Initial seeds for concolic testing are generated using *easy-random-4.2.0*[5] library.

Kex-concolic uses Kfg library for instrumenting the bytecode of the PUT with additional instructions to collect the symbolic traces. The instrumentation process can be time consuming, especially for PUT with large classpath, and the startup time of Kex-concolic is longer than of Kex-symbolic. However, concrete execution helps Kex-concolic to catch up during the analysis phase, because it is able to explore the execution tree of the PUT more efficiently. Kex-concolic uses context-guided search [8] for path exploration.

C. JUnit test case generation

Both symbolic and concolic tools produce test inputs as a set of symbolic data. Test case generation module transforms that symbolic data into executable JUnit test cases. One of the main problems of test case generation for object-oriented languages is the *encapsulation problem*: one cannot simply access private properties of an object to set them to the desired values. There are several approaches that attempt to overcome this problem (e.g., [9], [10], [11]), however, all of them are complicated, time consuming and unfortunately do not guarantee any results. Thus, in the context of the competition, Kex-symbolic and Kex-concolic were configured to use Java reflection library for object creation. This reduces the quality of the generated test cases, but maximizes the coverage. An example of the generated test case is given in listing 1.

[5]https://github.com/j-easy/easy-random

TABLE II
FINAL RANKINGS

Tool	CoverageR	UnderstandabilityR	OverallR
EvoSuite	1.79	2.23	1.83
Utbot-concolic	2.61	2.13	2.56
Utbot-fuzzer	3.76	3.00	3.68
Kex-symbolic	4.995	3.95	4.89
Kex-concolic	3.95	3.69	3.92

Each set of symbolic inputs is converted into its own separate test case. A test case consists of a set of utility methods that use Java reflection, a `setup` method that performs data initialization and a `test` method that invokes method of a PUT.

Main limitations of Kex come from the underlying techniques it uses. Symbolic execution is known to be limited in its ability to analyze large-scale programs, as it encounters the state explosion problem. Another weakness of Kex is the quality of the generated test cases. As mentioned earlier, they are aimed at maximizing the coverage rather than their understandability and maintainability.

III. Benchmark results

Table II reports the final rankings of the competition. We can see that Kex-concolic was ranked fourth with a score of 3.92, while Kex-symbolic received a score of 4.89 and was ranked fifth.

Table II shows, that each tool was evaluated using two metrics: coverage of the PUT and the understandability of the generated test cases. As mentioned in section II-C, Kex-symbolic and Kex-concolic prioritized coverage over the test case understandability, so the fact that both tools were ranked lowest in terms of the test case understandability is expected.

Coverage rating of every tool is considered based on the achieved line coverage, branch coverage, mutation coverage and time budget. The exact formula for score computation as well as the more detailed commentary on the overall results can be found in the full competition report [2]. Let's take a look at the coverage results achieved by Kex-symbolic and Kex-concolic in more detail.

First of all, competition organizers reported that both Kex tools failed to generate tests for the *ta4j* project. After analyzing the logs from the competition runs, we found the Kfg library had failed with `NullPointerException` on one of the JAR files in *ta4j* classpath, thus both tools were not able to even start the test generation.

Figures 2 and 3 report the distribution of line and branch coverage ratios for each tool participated in the competition for 30 and 120 second time budgets. Based on the results we can conclude Kex has significantly improved compared to the previous editions of the Java unit testing tool competition

```
public class ArrayListValuedHashMap-init-90775276022 {
    Object term22200;
    // number of utility methods here
    // ...
    @Before
    public void setup() throws Throwable {
        try {;
        Object term22072 = newInstance(Class.forName(
            "org.apache.commons.collections4.multimap.ArrayListValuedHashMap"
        ));
        HashMap term22248 = new HashMap();
        term22200 = newInstance(Class.forName(
            "org.apache.commons.collections4.multimap.HashSetValuedHashMap"
        ));
        setField(term22200, term22200.getClass(), "map", term22248);
        } catch (Throwable e) {};
    }
    @Test
    public void test() throws Throwable {
        try {;
        Class<?> klass = Class.forName(
            "org.apache.commons.collections4.multimap.ArrayListValuedHashMap"
        );
        Class<?>[] argTypes = new Class<?>[1];
        argTypes[0] = Class.forName(
            "org.apache.commons.collections4.MultiValuedMap"
        );
        Object[] args = new Object[1];
        args[0] = term22200;
        callConstructor(klass, argTypes, args);
        } catch (Throwable e) {};
    }
};
```

Fig. 1. Example of a test case generated by Kex-concolic

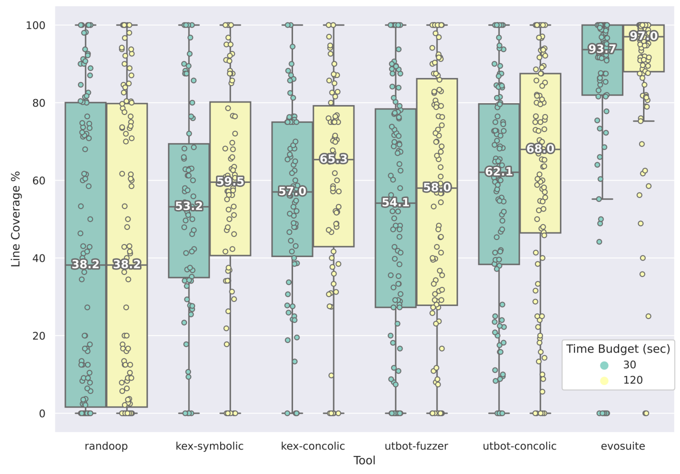

Fig. 2. Line coverage ratio

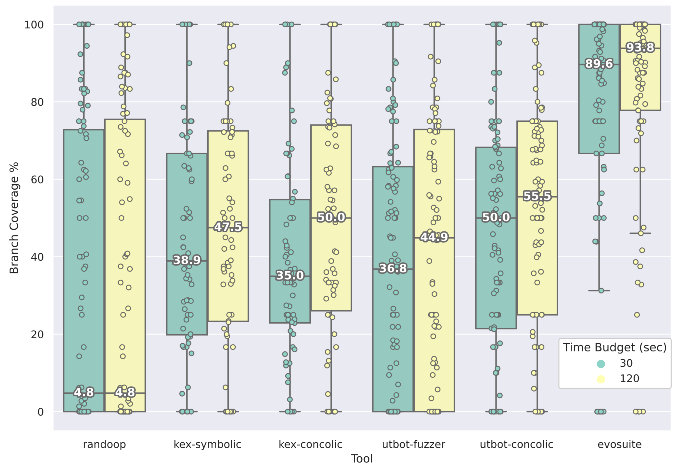

Fig. 3. Branch coverage ratio

in terms of the achieved coverage [12]. Kex performed on-par in terms of line and branch coverage with the only other symbolic execution based tool *utbot-concolic* [13]. We consider the results successful considering the time budget limitations imposed by the contest, which somewhat disfavor symbolic execution based tools.

Another metric used in the coverage ranking is the mutation score. It is computed as the number of mutants killed relative to the total number of mutants (a mutant is considered killed if at least one of the generated test cases fails on it). The contest organizers have reported that both Kex-symbolic and Kex-concolic received mutation score of 0. If we consider the test case generated by Kex (listing 1) more thoroughly, we can see that:

- the generated test cases do not include any test oracles;
- the generated test cases intercept and suppress any ex-

ceptions that are thrown during the test case execution.

These peculiarities exist because Kex tools have not been optimized for mutation score. In the future we plan to add test oracle generation to improve the quality of the generated test cases.

Overall we can conclude that Kex-symbolic and Kex-concolic have performed much better than Kex tools in the previous competitions [12]. We were able to significantly improve the coverage metrics achieved by our tools. However, the contest has revealed several important weaknesses of the Kex tools:

- Kex has several implementation issues that affect its reliability on some PUTs;
- Kex needs to improve the quality of its generated test cases;
- Kex still needs to improve its coverage metrics to be able to compete with search-based tools like *evosuite* [14].

IV. Conclusion

This paper reports on the participation of Kex-symbolic and Kex-concolic automatic test generation tools in the eleventh edition of the SBFT Java unit testing tool competition. Kex-concolic was ranked fourth with an overall score of 3.92, while Kex-symbolic received an overall score of 4.89 and was ranked fifth. Despite low rankings, we can conclude that both tools have performed well. The results of Kex show a significant improvement in coverage metrics compared to the previous year contest results. The contest has shown that the main weakness of both tools is the quality of generated tests. We are planning to improve our tools using gathered insights and to continue participating in the future contests.

References

[1] A. Abdullin and V. Itsykson, "Kex: A platform for analysis of JVM programs," *Information and Control Systems*, no. 1, pp. 30–43, 2022. [Online]. Available: http://www.i-us.ru/index.php/ius/article/view/15201

[2] J. Gunel and T. Valerio, "SBFT tool competition 2023 - java test case generation track," in *16th IEEE/ACM International Workshop on Search-Based and Fuzz Testing, SBFT 2023, Melbourne, Australia, May 14, 2023*, 2023.

[3] C. Barrett and C. Tinelli, "Satisfiability modulo theories," in *Handbook of Model Checking*. Springer, 2018, pp. 305–343.

[4] 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.

[5] "Ksmt," 2023. [Online]. Available: https://github.com/UnitTestBot/ksmt

[6] R. Baldoni, E. Coppa, D. C. D'elia, C. Demetrescu, and I. Finocchi, "A survey of symbolic execution techniques," *ACM Computing Surveys (CSUR)*, vol. 51, no. 3, pp. 1–39, 2018.

[7] K. Sen, "Concolic testing," in *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, 2007, pp. 571–572.

[8] H. Seo and S. Kim, "How we get there: A context-guided search strategy in concolic testing," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2014, pp. 413–424.

[9] Y. Zhang, R. Zhu, Y. Xiong, and T. Xie, "Efficient synthesis of method call sequences for test generation and bounded verification," in *37th IEEE/ACM International Conference on Automated Software Engineering*, 2022, pp. 1–12.

[10] N. Chen and S. Kim, "Star: Stack trace based automatic crash reproduction via symbolic execution," *IEEE transactions on software engineering*, vol. 41, no. 2, pp. 198–220, 2014.

[11] P. Braione, G. Denaro, A. Mattavelli, and M. Pezzè, "Combining symbolic execution and search-based testing for programs with complex heap inputs," in *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2017, pp. 90–101.

[12] A. Abdullin, M. Akhin, and M. Belyaev, "Kex at the 2022 sbst tool competition," in *2022 IEEE/ACM 15th International Workshop on Search-Based Software Testing (SBST)*. IEEE, 2022, pp. 35–36.

[13] "Utbotjava," 2023. [Online]. Available: https://github.com/UnitTestBot/UTBotJava

[14] G. Fraser and A. Arcuri, "Evosuite: automatic test suite generation for object-oriented software," in *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, 2011, pp. 416–419.