Test-Driven Usability Improvements for Runtime Verification Tools

1. EMPIRICAL STUDIES

So far, the main objectives in our experiments have been to (i) see how well RV tools can work in practice and (ii) discover any potential hindrances to the adoption of these tools. This section provides some details about these experiments, including the tools we used, the properties that were verified, the preliminary results and and further experiments which we plan to conduct in the future.

1.1 Tools Used

The two RV tools that we have investigated so far are JavaMOP [?] for runtime monitoring and RV-Predict [?] for data-race prediction. We also used Ekstazi [?], a recently-developed RTS tool for evaluating RV in the context of software evolution. These three tools were developed in our labs and were selected for our initial experiments for pragmatic reasons. In the future, we plan to evaluate other RV tools like Prm4J [?] which also does runtime monitoring in the spirit of JavaMOP but claims to be more efficient.

JavaMOP is a runtime monitoring framework for Java applications. The inputs to JavaMOP are (i) a set of properties and (ii) the target program to be monitored dynamically. The output is the target program, after it has been integrated with monitors that are automatically synthesized from the input properties. The synthesized monitors use an event-based model to check the dynamic behaviors of the target program, and take user-defined actions when the properties are violated. We use the JavaMOP default of generating a warning each time a property is violated. Each warning contains the property name, the line number of the target program where violation occurred, a URL of the formal property definition and one sentence explaining the property. [We control for I/O in the results presented. Show examples of a property and a violation message. Say something about RV-Predict and Ekstazi. Show an example of a race in message in RV-Predict and the code corresponding to the race.]

1.2 Properties

One of the inputs to JavaMOP is a set of properties. We used a set of [181] properties which were manually formalized from 4 packages of the Java API [?]. The work presented in this paper is the first time these properties are evaluated on subject programs outside of the Dacapo benchmark [?]. It is therefore no surprise that we found opportunities to improve on these properties in the course of our experiments. Our approach is not limited to these manually formalized properties. In principle, the properties can come from other sources such as specification mining using program analysis [?] or data mining [?]. As long as such properties are expressed in some formalism that JavaMOP can parse, they may be used. We plan to extend our experiments to include such properties in the future.

1.3 Experimental Subjects

The target programs that we have used so far are shown in the *Project Name* column of Figure 1. All the target programs were either randomly selected from github, or chosen from the Apache Software Foundation repository. This provides a good spectrum of open-source projects and is reflective of common development practices across various project maturity levels. The requirements used for selecting target programs was that they should have JUnit tests, compile out of the box and use Maven [] as their build system. We focused initially on Maven-based projects but we expect the same results to hold for other build systems (e.g., Ant [?]), and we plan to evaluate this in the future.

1.4 Experimental Setup

Our experiments for evaluating JavaMOP and RV-Predict experiments consisted of two runs. In the first run, we executed the the tests in each project in order to collect baseline information such as the test execution time, the number of tests and the lines of code in the project. In the second run we integrated the RV tools with the target program and perfromed runtime verification while running the tests. Note that the only modification we made to these projects was to edit the Maven build script (i.e., `pom.xml`) to incorporate java agents for the RV tools. [explain the pom.xml modification?] After this, we still ran the tests with the same simple command that developers would normally use – `'mvn test'`.

For our experiments to investigate RV in the context of software evolution, we had three runs to (i) run JavaMOP while running all the tests in 20 versions of each project and collect a baseline number of warnings generated as well as the runtime overhead incurred; (ii) run Ekstazi to perform RTS over the 20 versions of each project and collect the tests

No.	Project	LOC	Tests	Violations	BaseTime(s)	MOPTime(s)	Overhead(%)
1	FXForm2	5355	34	1008	5.2	20.9	304.1
2	JAQ-InABox	3570	1	1	5.1	5.5	7.4
3	JSqlParser	7786	232	27895	8.9	342.2	3733.2
4	ObjectLayout	1305	19	0	9.0	22.4	148.8
5	androlog	2532	8	19	4.0	5.6	41.1
6	apache.commons-lang	63425	2497	0	23.8	29.9	25.4
7	asterisk-java	35659	215	145	10.5	17.5	65.9
8	bcel	35827	73	27	7.2	12.2	70.9
9	commons-beans	33007	1269	0	43.4	962.6	2120.1
10	commons-cli	6292	364	0	4.7	8.2	73.8
11	commons-codec	16160	616	0	9.7	11.7	20.7
12	commons-collections4	52040	13702	0	25.1	32.5	29.7
13	commons-discovery	2588	14	0	4.6	7.5	62.5
14	commons-fileupload	4316	71	0	5.7	9.9	73.8
15	commons-imaging	36657	93	0	30.0	38.9	29.6
16	commons-lang3	63425	2497	0	23.1	30.8	33.2
17	commons-validator	11982	416	0	7.3	11.0	51.0
18	compile-testing	1813	22	149	3.9	19.0	385.2
19	compress	28931	466	0	9.4	16.1	70.2
20	connector4java	1928	36	470	4.1	22.8	458.1
21	dbcp	18759	480	28	67.8	73.2	8.0
22	dropwizard-todo	796	13	5674	8.9	79.9	797.7
23	functor	21688	1134	0	10.7	18.9	76.4
24	hivemall	5360	24	224	4.7	9.4	100.1
25	htrace	1531	9	68	16.0	23.4	45.8
26	invokebinder	2818	97	12	3.9	6.1	57.2
27	jblas	12570	120	0	5.3	10.3	94.2
28	jline	3419	22	5707	3.5	62.9	1674.5
29	joda-time	82998	4057	3504	13.9	53.2	282.7
30	jpatterns	2604	33	4	3.8	7.4	94.7
31	jsoup	13556	413	48385	6.1	366.3	5891.9
32	junit	25916	867	0	12.9	29.1	125.3
33	laforge49	7245	56	185357	4.5	2133.0	47800.9
34	logback-encoder	637	18	1812	4.2	20.1	371.9
35	math	186796	5943	0	120.9	123.6	2.2
36	ogrisel.pignlproc	2296	19	4200	64.7	92.1	42.3
37	paper2ebook	142	2	9329	2.9	136.7	4623.2
38	scribe-java	5344	99	24	13.0	16.5	27.2
39	zookeeper-utils	455	4	1297	8.5	18.3	116.1
40	zxing	42493	373	457094	33.8	4399.0	12904.0
	Total	852021	36428	752433	654.8	9306.6	82941.3
	Arith. Mean	**21300.5**	**910.7**	**18810.8**	**16.4**	**232.7**	**2073.5**

Figure 1: Time overhead of monitoring each project

selected in each version and (iii) run JavaMOPwhile executing *only* the tests selected using Ekstazi in the 20 versions, and collect the number tests as well as the runtime overhead for this subset of the tests. Finally, the numbers from run (i) and run (iii) are compared and plotted. Figure **??** is shows an aggregated plot for all the projects. One interesting idea for future work would be to investigate the projects which currently have no violations, go back in their history to see the trend in the number of violations. We may also conduct experiments with warning suppression such that we measure, using a projects prior history, how long it will take to achieve zero-violations, with and without RTS.

1.5 Results

We highlight here some important take-aways from both sets of experiments.

- Mature projects had no violations, even over several versions
- The runtime overhead for the RV tool evaluation seem not to bad but better for JavaMOP than for RV-Predict.
- What is the effect of combining with RTS right now?
- Are there any trends that are already emerging from the combination with RTS?
- Answer the RQs from Section **??**.
- (Potential) Bugs that we found in code, in test, in property and with the tools.

1.6 Threats to Validity

- The properties we used for JavaMOP are limited to 4 packages
- We are assuming good test coverage. (maybe measure this?)
- We evaluated only our own tools