Comparing test and production behavior for dynamic languages

Quentin Le Dilavrec

No Institute Given

Contents

1 Introduction

Testing a software system is a fundamental process in software development and considerable amount of the total development cost is reserved for software testing.

And extensive research citations has been done to improve software quality, through methods revolving around test and verification. But there is a trade-off between the flexibility granted by a languages and the cost of its analysis. In the case of dynamic languages like javascript, whose success come from the low entry cost and flexibility for developers, it combine the imperative and functional paradigm and allow to evaluate code at runtime, all those degree of freedom make it very hard to gather the necessary information that will allow in depth analysis. In this conditions even if tests are particularly adapted as they have access to runtime information, all the methods revolving around automated tests generation are lagging behind like analyzers because of the lack of runtime in formations. To close the gap we can try to harvest information at runtime but it come at a cost, first like tests it consume computing power, second it might change the behavior of the system, and third due to the vast number of run variations present in complex systems, the harvested information needs to represent the real usage of the system. The apparent trade-of here is between the precision of the instruments and the impact of those instruments on the studied systems. On one hand less precise traces representing runtime information will lessen the precision of the analysis, on the other hand instrumenting complex systems can impact performances and validity and can't be neglected as it will impede the accuracy of subsequent analysis.

We aim at useful analysis of widely used software, that allow us to give useful feedback on those software code quality, through mainly the coverage by tests of behavior observable in field.

The results obtained can help developers understanding the behavior of a piece of software, but can also be feed to more classical automated test generation tools.

2 Context/Related Work (mainly for m1 report, we were asked to be really clear and self contained)

- Automatic Testing
- input

- oracle
- coverage

Research in software engineering aim at improving our understanding of software, by applying state of the art knowledge coming from other domains of computer science, like big data, natural language, machine learning to the analysis of software but also formalizing practices coming from empirical experience in the industry. The main point of most tools in the field is to improve software quality, there are many approaches. Static analysis is particularly effective to make exhaustive verification but can't apply to all software and does not scale very well. Tests on the other hand can check the result computed by a piece of software, given some input, it can be applied to most software but lake exhaustivity. Due to the limitation of tests on exhaustivity one major point is to generate tests automatically to lessen human labor but also evaluating the usefulness of tests.

3 Logging javascript calls in browser and nodejs

high-level information on the behaviour of commonly used software, we looked at web applications running in browsers and also extensively tested with, for example, unit tests. Whether it is inside a browser or during compilation, the approach to gather data at runtime is done, by first adding a new step in the compilation/loading pipeline, this we allow to proceed with the modification of the program, that will intercept runtime data on the instrumented program. At runtime it is also necessary to expose a collection point to the instrumenting instruction that will handle the collection of data.

In the next subsections we will explain how we instrument actual javascript code to log function calls

3.1 logging

In [2] Wang and al. log the calls at method level because prior work has argued that method call sequences represent the best cost-benefit tradeoff for reproducing field failures [1]. However in javascript there are many ways to create reusable pieces of code,

- named function
- anonymous function
- object method
- lambda expression
- Function/eval
- class methods

The official Abstract Syntaxe Definition of *TreeJS* regroup them as a type union under the name Function. As a common denominator they all be called using the standard call syntax Moreover they can also be used and exposed in different ways,

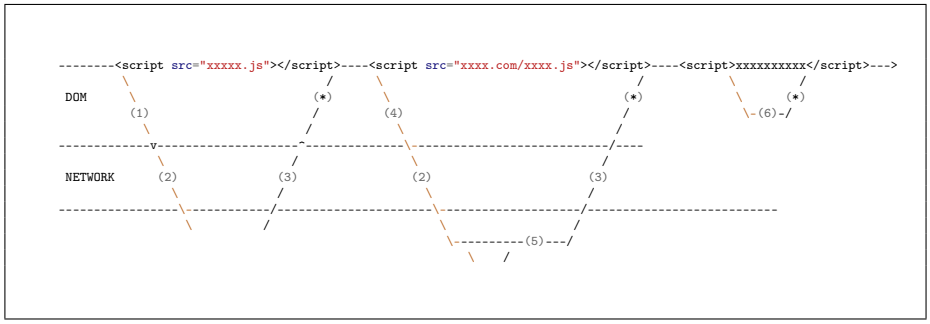
– as an attribute of an object
– returned by a function
– passed as a parameter
– in a variable

This variability allow a great flexibility for the developer but makes analysis more complicated. Thus we decided to instrument all those kinds of functions by only adding an instruction at the start of the declaration body, this instruction append to the trace a new element made of the identifier of the function and parameters We tried to make it the least intrusive possible by only adding this one line and serializing the parameters, in the case of lambda expression without a block (only allowing an expression but no instructions) we choose to

– transform it into a block, doesn't change call stack and wrap the expression in a return instruction
– use the coma operator, not very known
– wrap it with a call, problem with scope

3.2 dynamic instrumentation

In a browser, we found 2 way to instrument javascript that allowed us to change the text of the script. One can be implemented using an extension which allow us to modify the the loading of the page at parsing, the other make use of the *DevTools* API that allow us to intercept http requests and responses. The first method is most accessible one as it only need to install a new browser extension, the second one is more instrusive and need to be launch as a new process, but it allow us to access more low level API and make use of the cache, loading parallelization and offload the instrumentation to a background process.



Listing 1: Schema representing the different approaches to dynamic js instrumentation in browser

1. interception at DOM parsing, get the script tag as parameter, request script source, then instrument text of tag and populate innerhtml of tag

2. interception of outgoing query, to redirect to other url or directly respond to query
3. interception of response packet, intrument text in packet body then forward
4. interception at DOM parsing, get the script tag, ~~get script source code, then instrument text of tag~~, CORS domain policy forbid programmatic requests to remote domains, so need to modify url to go through intermediary server
5. use an intermediary server to instrument script, need to change the original url
6. interception at DOM parsing, get the script tag, then instrument inner text of tag

*) parsing and evaluation of javascript tab by web browser

(Intercepting) DOM parser events Our most accessible method to instrument javascript code rely on observing parse events [1], it allow us to modify scripts added on the page during DOM construction This observer can be directly included in the original page as the first script tag, on dynamically added to pages with a web browser extension or using the DevTools API But in some cases this approach can change the behavior of script tags. actually local scripts need to be transformed into to inline scripts. The main problem of this method is the impact on performance because it can't make use of the web browser cache, thus it need to instrument the page at every load. transformations examples? pseudo code?

(Intercepting) http requests The second method is more efficient as it make use of the cache but it can only be implemented though the Fetch domain of Devtools API. talk about pupeteer? here? or in the dynamic instrumentation section? to intercept and modify incoming responses to scripts requested. js parsing to instrument code is done in the puppeteer (nodejs) process, relieving the web browser of some stress and isolating the instrumentation process from the rest of the application. Don't work on inline scripts at this point (maybe intercepting html request and parsing the DOM) More than just an extension, but Docker container available (in this case it needs X).

3.3 compile time instrumentation

All of the big web applications uses some kind of compilation in there pipeline. It allow for greater development flexibility and automating retro-compatibility. We used created an AST transformation that integrate to most compilation pipelines, it conform to the official Abstract Syntaxe Definition of *TreeJS* and is implemented through the *babeljs* compiler. talk about problem of instrumentation passes?

[1] `https://developer.mozilla.org/en-US/docs/Web/API/MutationObserver`

4 Making use of traces

One of our main concern was to adapt state of the art software analysis methods to the development of web applications. In this condition some requirements are necessary:

- low latency responses
- no false negatives
- incremental responses from most important to less important
- no grand8 reports (no appearing and disappearing of error), only increase importance level, never decrease without user implication

Behavior of the system Extensive research [...] has been done to represent the behavior of software systems.

4.1 Processing of traces

logs or traces? Assuming that

4.2 Representing mined information

Another challenge is to relate the information mined from logs in a meaning and useful manner.

5 Application to Wordpress

Why Wordpress?

5.1 Experimental procedure

Production Multiple people with their personal computer do things with an instance of *Wordpress* shared over local network

Tests Run on a intel I7, gtx960M, 8Go RAM laptop.

5.2 Results

distribution of calls on dataset

```
production/test calls sorted by number of occurences of production.
y axis show # of calls
x axis show calls, only some interesting calls should be visible.
```

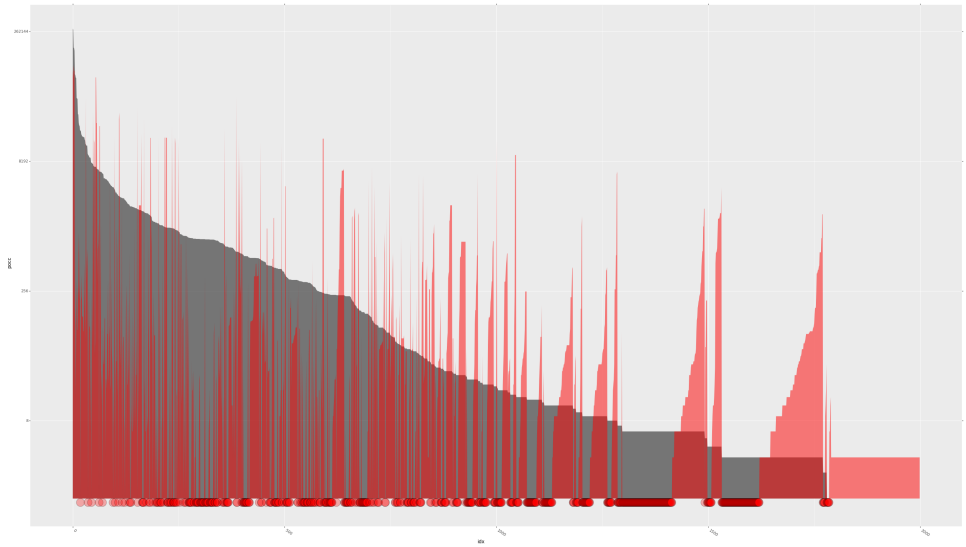


Figure 1. Distribution of number of calls per JS functions for production (grey) for tests (red) in traces of Wordpress

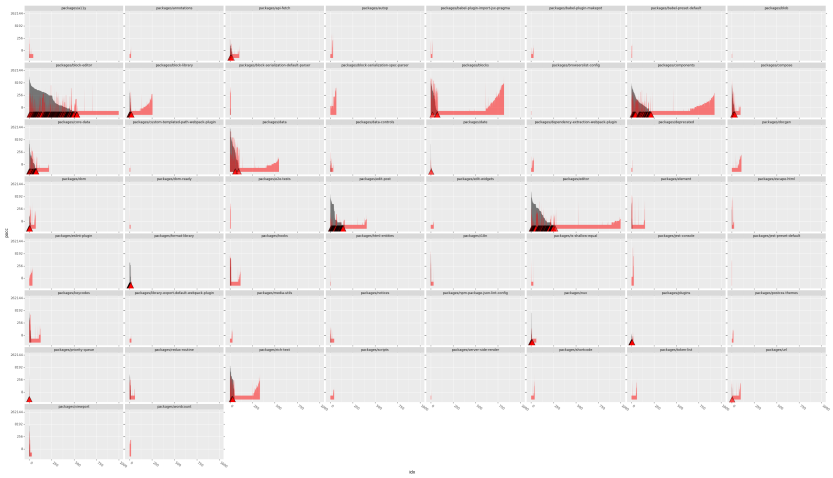


Figure 2. Distribution of number of calls per JS functions for production (grey) for tests (red) in traces of Wordpress split by packages

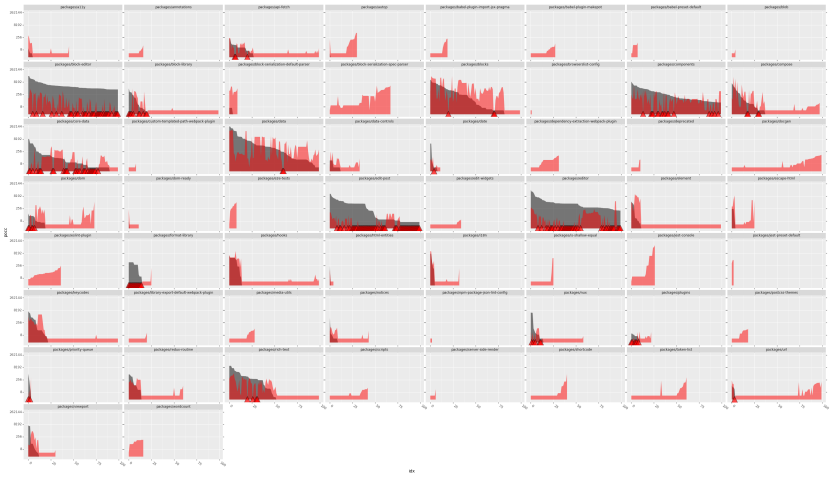
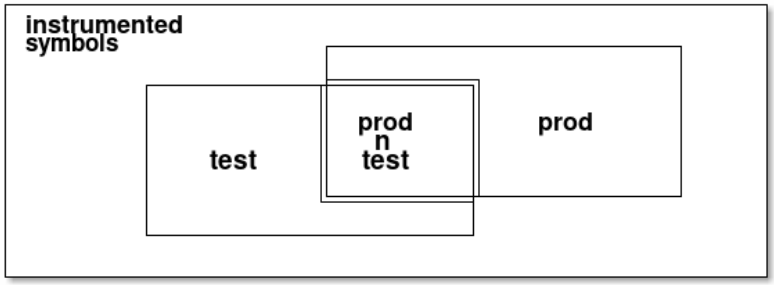


Figure 3. Distribution of number of calls per JS functions for production (grey) for tests (red) in traces of Wordpress split by packages with values truncated to 1000 calls, to make it bigger.

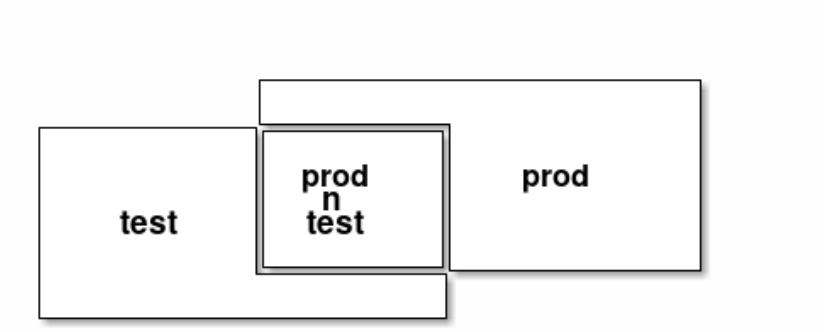
distribution of calls on dataset per gutenberg package

```
production/test calls sorted by number of occurences of production.
y axis show # of calls, (log2 scale)
x axis show calls, only some interesting calls should be visible.
data faceted by package
```

Venn diagram with symbols of functions



Venn diagram with function's symbols and parameters



Precise analysis of methods/functions usage context from field compared to tests

6 Conclusion

The results obtained can help developers understanding the behavior of a piece of software, but can also be feed to more classical automated test generation tools.

7 References IGNORE

References

1. Wei Jin and Alessandro Orso. Bugredux: Reproducing field failures for in-house debugging. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 474–484, Piscataway, NJ, USA, 2012. IEEE Press.
2. Qianqian Wang, Yuriy Brun, and Alessandro Orso. Behavioral execution comparison: Are tests representative of field behavior? In *2017 IEEE International Conference on Software Testing, Verification and Validation (ICST)*, page nil, 3 2017.

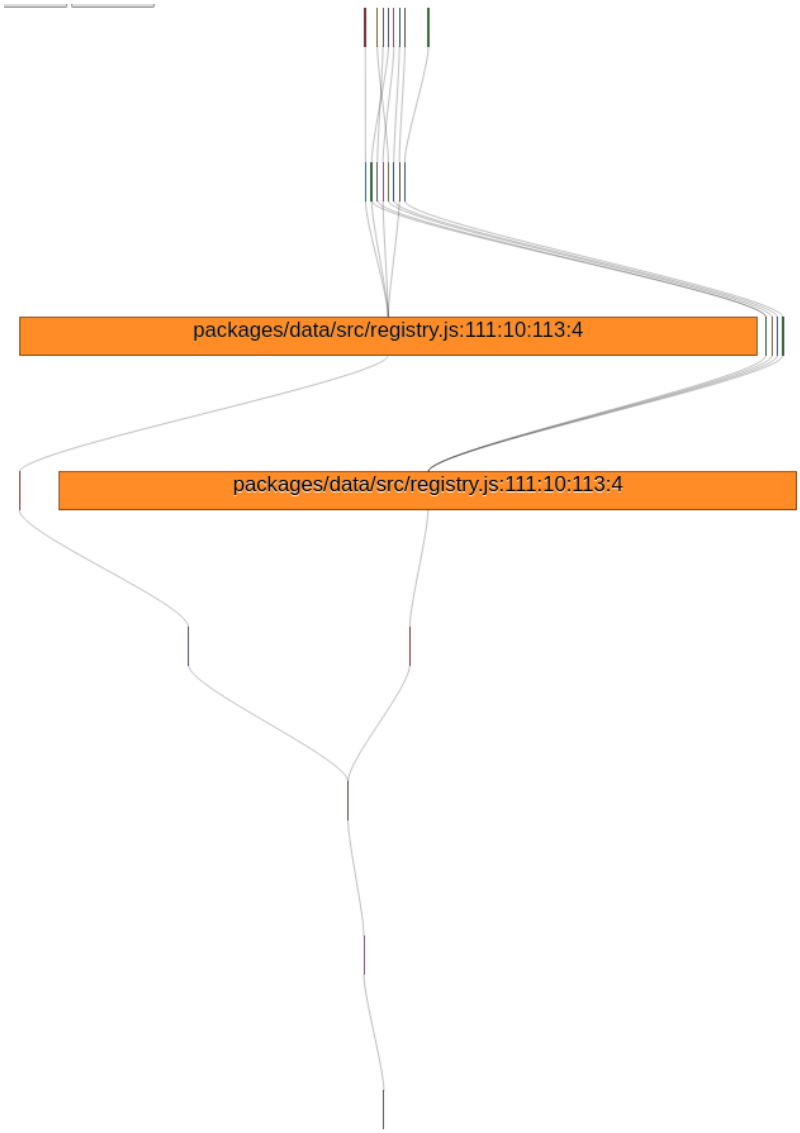


Figure 4. createNamespace usage context from field compared to tests, la largeur d'un noeud est proportionnnel à son nombre d'appels en production

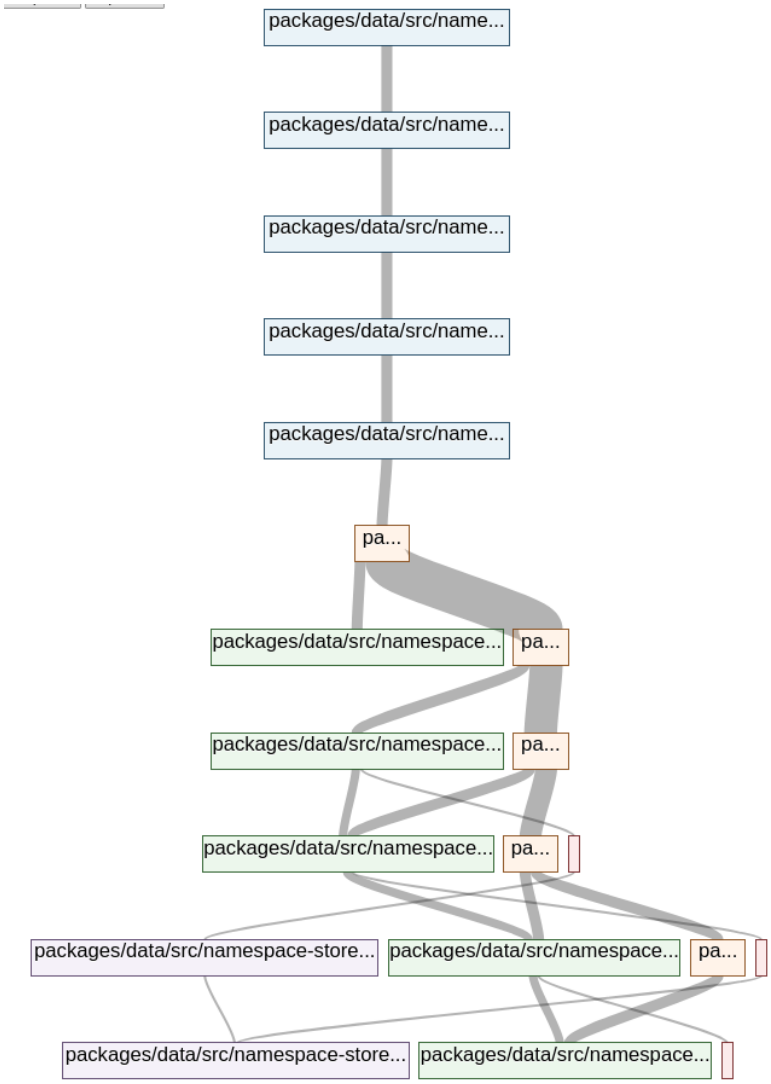


Figure 5. callback function wrapping a retruned selector in createNamespace, usage context from field compared to tests, la couleur des noeuds reprentente le rapport entre le nombre d'utilisations en production par rapport aux tests