Comparing test and production behavior for dynamic languages

Internship at KTH from May 13, 2019 to July 15, 2019

Quentin Le Dilavrec[1], supervised by Benoit Baudry[2]

[1] Univ. Rennes Quentin.Le-dilavrec@ens-rennes.fr
[2] KTH baudry@kth.se

Abstract Wordpress is the most popular Content Manager System. It is Open-Source and power more than 34 percent of all websites. Any improvement of the quality of a system like WordPress can have a great impact, be it on developers, website creators or end-users. Like other web-applications, WordPress is increasing its usage of the client-side. All those web-apps rely on JavaScript, which dynamic nature allows great flexibility but challenges previous methods of instrumentation, analysis, and visualization.

1 Introduction

Wordpress is the most popular Content Manager System. It is Open-Source and power more than 34 percent of all websites [1]. The official repository contains approximately 47,000 plugins, cumulating 600 million downloads [3], and there is an entire economy based on these plugins. So any improvement of the quality of a system like WordPress can have a great impact, be it on developers, website creators or end-users. In web-applications, increased usage of the front-end can have many advantages, it can off-load servers, provide more dynamic features, such as markdown. But integrating new languages and new compilation pipelines to a project as big as WordPress is not trivial.

Since a few years, the WordPress project has seen its proportion of javascript quickly increase especially for the front-end part [3] under the name Gutenberg. JavaScript is the programming language of the web, it runs on almost all devices through web browsers, and took the web thanks to his flexibility and accessibility. It is multi-paradigm and its dynamic nature allows anyone to fiddle with the behavior of any websites. Moreover, it evolves swiftly and not with any frictions, with a new version of the ECMAScript specification published almost every years, followed by major companies or foundations that try to implement new features in their respective web browsers. It makes projects that revolves around transcompilation the most popular and widespread javascript software, they mainly focus on compatibility between versions of the ECMAScript specification and the different web browsers but they also allow developers to add their own features to the language and try to push the language forward.

But the dynamic nature of vanilla JavaScript make it hard to understand but also hard to analyze by only relying on source code. The most favored approach to this problem is to use typing systems and ask developers to annotate their code. There are therefore in addition to basic syntax checking and basic refactoring tools, static analyzers that implement many of the features available in more static languages, such as intelli-sense, which is developed by Microsoft. But although these tools are very powerful, they are not adapted to some complex cases. The information accessible at runtime are thus needed to completely understand the behavior of some piece of code.

The main contributions of this paper are:

- An efficient and flexible instrumentation technique of JavaScript.
- An algorithm that allows to compare traces and extract patterns.
- A dynamic visualization that uses traces to see the codebase from a global point to fine grained-interactions.
- An evaluation of the contributions on WordPress

2 Dynamic analysis of javascript

Dynamic software analysis is a way of analysing a software by executing its program while doing measurements. Accessing information only available at runtime, allow to complete a static analysis and deepen the understanding of a given piece of software.

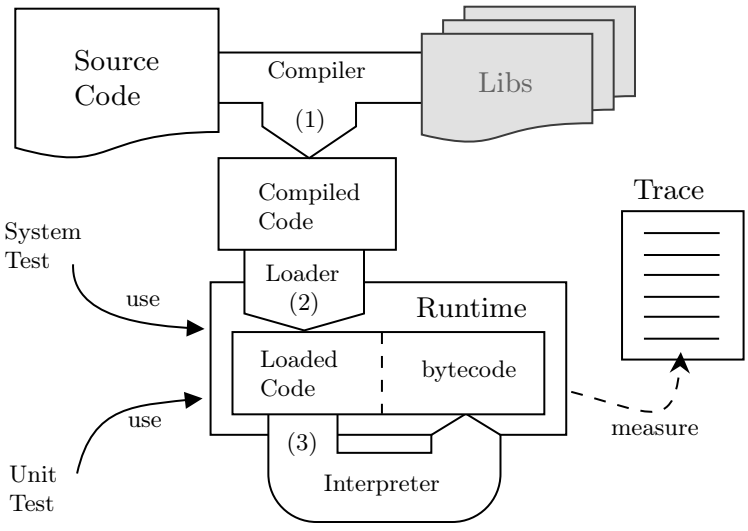


Figure 1: The instrumentation can be done at multiple layers

2.1 Workload

While the workload is the preliminary requirement for dynamic program analysis, acquiring such a workload remain a challenge. Today, research works that perform dynamic analysis rely on the following techniques to build workload:

- In [6] Wang et al. compare in-field executions to unit test executions using inputs from Massive Open Online Courses (MOOCs)

but also real world systems that use library software in a realistic way.

- In [5] Jiang et al. detect abnormal traces produced by requests done to server-side software.

The additional knowledge that can be acquired through dynamic analysis depends above all on the representativeness and diversity of the inputs. These inputs can come directly from a user of the application and even through other software, but also from robots during automated tests phases. In the domain of software testing, system tests are often semi-automated and very big, while unit tests are fully automated and simple. In most cases, the set of possible entries is infinite, as are the set of executions and therefore the set of tests. This inherent lack of exhaustiveness and the cost of running tests require us to limit the possible entries.

2.2 Instrumenting to log run-time information

To make dynamic analysis we need to gather dynamic information on our study subject. It is done by putting probes in the application, those probes will log information in the form of a trace that will later be analyzed. When these probes are only used to do measurements, to avoid deviating from reality, they should not produce any side effect, even temporally. Thus it is necessary to choose what needs to be measured by the probes. The process of inserting these probes is called instrumentation, it can be done at multiple layers and at different times. Choosing when and where to instrument is mainly a matter of what we want to measure, but also what we have access to. To ease later analysis relative to a particular layer, the instrumentation should take place at the closest layer possible. If the aim is to provide information on source code, the easiest way is to instrument source code. In the case of when, the instrumentation can be done at compile-time (1), during the loading (2) of the runtime or by the interpreter (3).

2.3 Analysis

The main ambition here is to make use of what has been done in software analysis on mature systems [6] and adapt them to the development pipeline of web applications. For a tool focussing on software quality improvement to be widely accepted by the developer community, it should not produce false

negative and warning should be ordered by priority. Moreover in our case due to its dynamic nature, one of the greatest difficulty when reading javascript code is to understand the context in which it will be executed. So on top of a warning, an optimization or a new test, if it requires the intervention of developers, some kind of contextualization should be provided.

There is many ways to analyze runtime behaviors as shown in [6], behavioral models based on the k-Tails algorithm can be used to compare mined models from the field to mined models from tests. But unfortunately, this algorithm needs refined traces, as computing the behavioral model of raw traces is a waste of resources if half the model end up thrown away. Considering the pace of changes in web development and the large and heterogeneous community there are many benefits to adapting behavioral models methods to the way those software are developed.

3 Contribution

We will now present our contributions, starting with our approach to instrument a JavaScript project such as Gutenberg, then how we analyze traces to help improve the quality of unit tests and the overall understanding of the system. We choose to instrument JavaScript programs by modifying the text of the scripts, as it only needs to have access to the code somewhere in the general pipeline, thus being more flexible.

3.1 Instrumenting

Whether inside a browser for system tests or in a headless interpreter during unit tests, the collection of execution data is carried out by adding a new processing step in the compiler, loader or interpreter, this new step is in charge of modifying the program before its execution, the modifications are mainly focused on intercepting and storing execution data for later use. To instrument web applications, we created an Abstract Syntax Tree transformation that can be easily integrated into most pipelines, it conforms to *TreeJS*, the official Abstract Syntaxe Definition of JavaScript and is implemented through the *babeljs* compiler. Technically this transformation is done using a simple visitor pattern, the object and nature of the modifications will be discussed next.

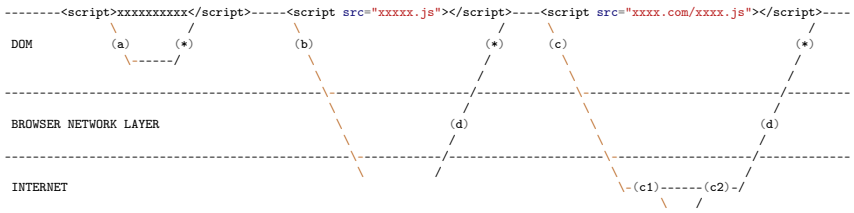
Tracing all data available at runtime is not practically possible, it is therefore essential to choose what should be measured. To compare executions in production to executions in tests for Java projects, Wang and al. [6] trace the methods calls, assuming that in Java, methods are the main reusable structures. In JavaScript, the main structure holding reusable pieces of code are functions. However there can be declared in many different ways, namely `lambda expression` ; `named function` ; `anonymous function` ; `object method` ; `Function/eval` ; `class methods`. The official Abstract Syntax Definition regroup them as a union under the name `Function`. As a common denominator they can all be called using the

standard call syntax. But they can 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 allows great flexibility for developers but makes the analysis more complicated. To make it the least intrusive possible, we decided to instrument all those kinds of functions by only adding an instruction at the start of the declaration body. Only the case of lambda expression without a block (only allowing an expression but no instructions), needed more modifications, so we choose to transform the body into a block and wrap the original expression in a return instruction, as doesn't change the call stack. This transformation is common as older versions of javascript don't have lambda expressions. Back to the instruction content, it adds contextual information to the current trace. The most important information is the identifier of the called function, but we also tried to get the values of parameters to see if it was possible to extract some useful knowledge. Gathering parameters is complicated as it requires to serialize big, complex and recursive structures, so we choose to limit the serialization to very small depth.

Dynamic instrumentation can be done in a browser, we found 2 ways to instrument JavaScript that allow changing the text of the scripts. One can be implemented using a browser extension to modify the way a page is loaded, the other method uses the *DevTools* API to intercept HTTP requests and responses. The first method is more accessible as it only needs to install a browser extension and have access to the DOM, the second one is more intrusive and needs to be launch as a new OS process, but it gives access to low-level API and by default make use of the cache, load scripts asynchronously and offload the instrumentation to a background process. The loading protocol of scripts in a web page[3] is very complex, to make it clearer the declaration of scripts can be split into 3 kinds, as shown in Listing 1. Those 3 kinds of script tags are used in different situations. The first one is an inline script so the content is part of the page, the second one loads a script file from the same domain as the page, and the third one loads a script file from a remote domain. For performances reasons non-inline scripts are asynchronously loaded, and the file they are referring to can not be modified. Moreover, there is no http request to load inline scripts, it is thus only modifiable at a network level by modifying the original page. To avoid misses, each instrumentation needs to be done before (*) the javascript interpreter parse and evaluate the content of the given scripts.

1. The DOM layer is were the global scope lies. Thanks to the Mutation Observer API[4], it is possible to listen to parsing events and modify scripts loaded on the page, as the DOM is constructed. The script in charge of the instrumentation needs to be directly included in the original page as the first script, or dynamically added during loading. Dynamically adding script

[3] https://html.spec.whatwg.org/multipage/scripting.html#script
[4] https://developer.mozilla.org/en-US/docs/Web/API/MutationObserver



Listing 1: The different approaches to interecpt different kind of script in the web browser

tags can be done with a web browser extension[5] or using an API[6] accessible from outside of the web browser. When a parsing event is triggered, we handle the 3 kinds of scripts in different ways. Inline scripts are the easiest to modify, in (a) we replace the content of the script with its instrumented version. For local scripts, in (b) the script file is programmatically fetched, then its content is instrumented and assigned as its new content, the script becomes inline. Remote scripts are more complex, in (c) we need to change the URL to redirect it to a local HTTP server. This local server will fetch the original script content (c1) then instrument it (c2) and finally send back the modified version.

This approach is functionally correct but it does not scale very well and end up with some noticeable impact on performances. On one hand it can't make use of the web browser cache, so it requires to instrument web pages at each load. On the other hand, the instrumentation logic is executed in the same environment as the web application, leading to perturbations during loading.

2. Instrumenting http responses is the second method that we tried. It is more efficient as it makes use of the web browser cache but it can only be implemented through the Fetch domain[7] of the Devtools API. It is much more simple as we only need to intercept and instrument incoming responses to requested scripts. The instrumentation is done in another process, relieving the web browser of some stress and isolating the instrumentation logic from the rest of the application. For now, it does not work on inline scripts but instrumenting HTML along with their inline scripts could be used. It uses Puppeteer[8] as a wrapper around the DevTools API to grant some more portability between OSes.

Compile-time instrumentation is needed to instrument unit tests in most javascript projects. In fact, unit tests are often run out of web browsers, in more lightweight interpreter like *nodejs*. Moreover all of the big javascript projects use

[5] https://developer.chrome.com/extensions
[6] https://chromedevtools.github.io/devtools-protocol/
[7] https://chromedevtools.github.io/devtools-protocol/tot/Fetch
[8] https://github.com/GoogleChrome/puppeteer

some kind of compilation pipeline, as it allows for greater development flexibility and automating retro-compatibility. The more used js compiler are *Browserify* and *Babeljs* used together.

3.2 Analyzing

Considering the size and complexity of today web applications, any analysis tool aiming for real-world software should consider scalability as its major constraint. But improving a software can be split in 3 steps, detecting a problem, then finding its position and finally solving the problem. Detecting and finding problems don't need to be very precise nor exhaustive so it can accommodate with a big quantity of data. And solving a problem should only require a partial view of the available data. Moreover here traces produced through instrumentation are structured. So we choose to use and extend a relational database to analyze and process traces, then an interactive visualization to display feedback at different stages.

The processing of traces can be done in multiple ways. We tried to apply the methodology of Wang et al. [6] on the analysis of runtime traces of calls to methods in Java, but due to the variety of symbols and the size of our traces, this method cannot meet our requirements for memory and efficiency. In our case realistic traces contain at least 1000 unique symbols and contain around 1 million entries. In fact, we are 2 orders of magnitude higher in terms of the number of symbols and the size of the traces compared to the maximum values evaluated in [2]. Moreover, for a project like WordPress, no developer needs a representation this precise of the whole system. That is why we moved towards a more NLP-oriented approach, using n-grams. Using n-grams to analyze source code is already relevant as shown by Hindle, Abram, et al. [4]. And n-grams of traces can be used to detect abnormal execution [5]. It allows processing traces halfway between purely statistical analysis and behavioral analysis. With this new information, on top of giving developers a representation of the context of the execution of their code, it might allow us to run algorithms like k-Tails more efficiently. But if done naively, enumerating and processing n-grams can also be wasteful, consequently, we choose a more recursive and incremental approach that enumerate n-grams starting from a set of given symbols. Thus it relies on the same assumption than k-Tails but only needs and gives a partial view. Anyway in the case of tests improvements, there is a cost at adding tests as they require maintenance and computing power, so by relying on statistics of already computed n-grams, we can on behaviors more relevant to test (see the evaluation section for examples).

Interactive visualization can be used to get a global view on a large quantity of data. To ease the reading of that information, it is essential to aggregate relevant data while allowing to zoom on particular parts. In a large codebase such as Gutenberg, representing all functions declared in the codebase can be

difficult, mostly because of the large number of functions –almost 10000 in the case of Gutenberg– but also because we want to show relations between functions. We produced two representation, that comes in complement to the code. The first representation uses a tree structure similar to those found in some popular IDE where the outline of source files are shown in the file explorer. Here in addition to the basic folding tree, the size and color of nodes are proportional to some metrics that we can compute from our traces.

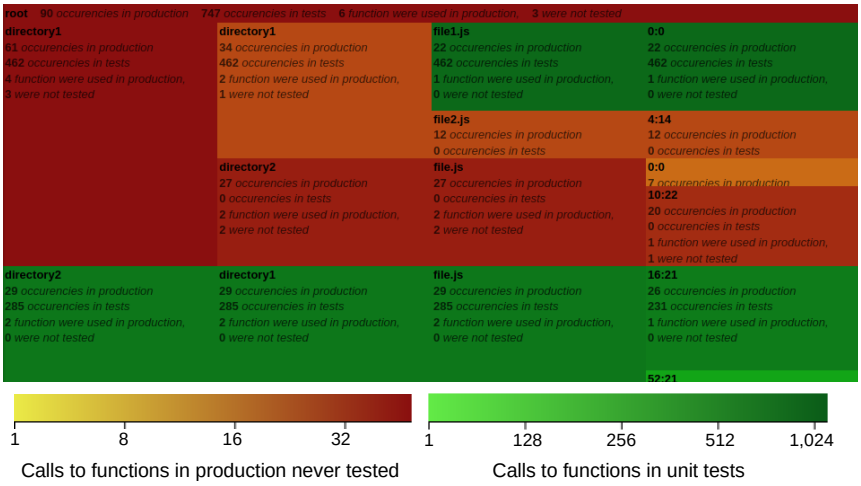


Figure 2: Example of our global view representation

Figure 2 is an example of this representation. With the size of nodes that are proportional to the usage of function in production and the colors are proportional to the usage of function in unit tests. It allows to easily see functions used many times but never tested. With this representation, it is even possible to represent the parameters passed to functions or other structures similar to trees. However there are some limitations, as it is necessary to keep the proportion right to be able to compare elements. Circles are not usable to represent nodes because of geometric limitations, but we successfully used rectangles in this particular layout that also allows seeing the name of nodes. In this layout, the size of a node is its height. The metrics usable to compute the size of nodes are also limited as they need to be aggregated with a sum. Where used in a system that allows interactivity, it is possible to zoom on nodes and using an IDE it is possible to jump to the code and to the next representation that we will be presented in the next paragraph.

The second representation shows the context of functions during execution. It can be used on a function to understand its context of use and allows grasping how a function is used by presenting which context should be reproduced to test that function efficiently. It dynamically mines the traces using the processing

method presented earlier. Here the produced n-grams are fused to construct a graph with the starting function at its center. Each node represents calls to a function, the width represents the number of calls made to this function, each transition width represents the number of times a sequence of calls was made. Nodes are given the same color when they represent calls to the same function. Here it is possible to dynamically search for bigger n-grams and using an IDE, it is also possible to jump to code. Moreover to compensate for the merge of nodes, while hovering a link, it highlights other links contained in the same n-grams. This representation is made to also accommodate a model mined with a k-tail algorithm. As an example, the Figure 3 shows the context of a function called in a loop that always start after a unique function and end with another unique function.

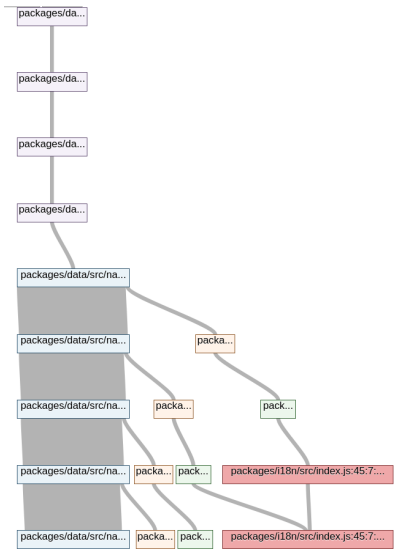


Figure 3: Context of a function called in a loop

4 Evaluation

We will now show the results obtained through the analysis of Gutenberg, trough a simple procedure and using limited inputs to show that even with a simple experimental procedure, it is possible to analyze a large codebase and propose improving modifications.

4.1 Experimental procedure

all the experiments were run on an *ArchLinux*, *intel i7* laptop with 8go ram. the source code of WordPress was first instrumented during compilation using our AST transformation. production traces were made on an instrumented chrome browser, the user inputs are reproducing the writing of a very short post only made of text. Tests traces were made by running the unit tests with *NodeJs*, each run produces many traces, one for each test. the following results will present 12 production traces and 3 sets of unit test traces each containing 767 traces. for a total of 6.3 million calls

4.2 Macro Analysis

We can get a fist look at our data in Figure 4 using a Venn diagram. At first glance, we can see that our system tests only used a small part of declared functions, nonetheless around one-third of used functions were not tested. with this figure, it can be concluded that at least 1122 functions need new tests. But we can't conclude anything on functions tested but not used as our set of input is restricted.

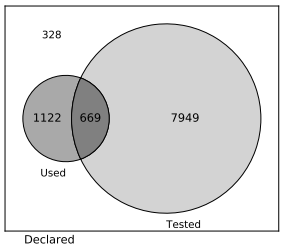


Figure 4: Venn Diagram on functions in Gutenberg

To see which functions need new tests, the Figure 5 present a static view of the first interactive viewer presented in the contribution. The precision of this representation allows to find the most used functions that are never tested. As explained in the contribution, the structure of the code base is preserved down to the functions, data available on functions are aggregated to form the nodes of the tree, it is thus possible to traverse the tree and find functions that need new tests by order of priority. On the one hand, the more a function is called, the larger its rectangle is. On the other hand, the more a function is used without being tested, the more the color tends towards red, and the more the function is tested, the more its color tends towards dark green.

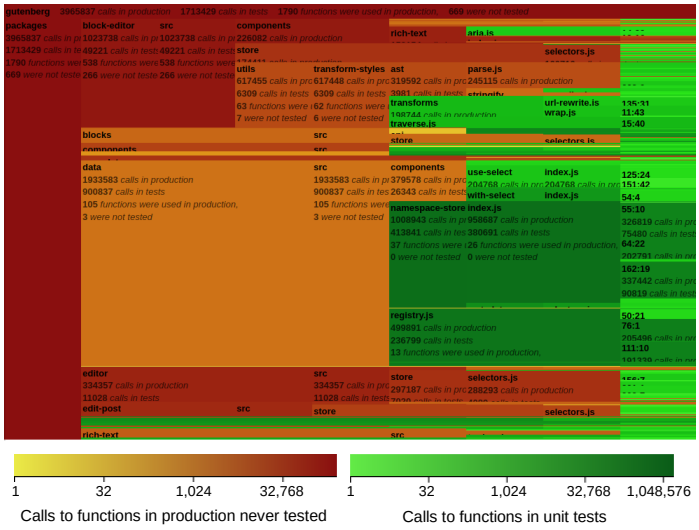


Figure 5: Test coverage of functions used in production

The Figure 5 shows that 62% of used functions are tested at least one time and the functions never unit tested makes it for 6.8% of calls. The interactive version was used to find a few functions needing new tests.

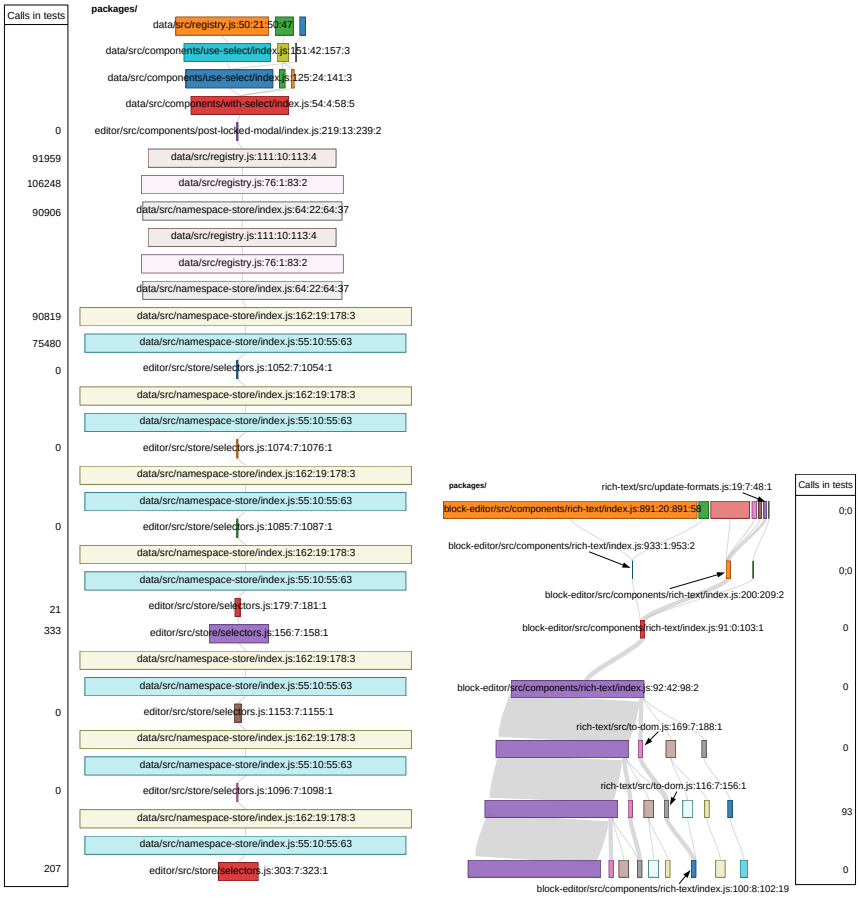
4.3 Preparing new tests

In the previous section, functions lacking tests were found, here the aim is to understand the context of these functions to produce the most relevant unit tests. The second representation presented in the contribution is designed to achieve that goal.

On the left, the figure 6a shows a long sequence of calls, containing multiple functions never tested by reproducing this context it would be possible to add a first test to 6 functions in one go thus reducing the overhead. looking in more detail on those non-tested functions, they are simple accessors and the pale functions are well tested, but looking at the code, they are part of a fairly complex program made of many closures (functions returning functions). so more tests won't do any harm here. the figure 6b shows a loop, it is similar to the example presented in the contribution but in a more complicated case. There are more branches at the start and the end of the context so the new tests skeleton should be made of these sequence of calls. Once the sequence is chosen, it is even possible to fetch the parameters in the traces, and fill up the calls parameters.

5 Conclusion

We have created an analysis tool for dynamic languages, capable of tracing information accessible at runtime, then process it and finally display useful



(a) Sequence of non-tested functions (b) Called in a loop

Figure 6: Context of some functions

feedback. The results obtained can help developers at understanding the behavior of a piece of software, but it can also be feed to more classical automated test generation tools. This tool has been evaluated on WordPress and could easily be adapted to other web applications or JavaScript packages. For now, it is integrated into Visual Studio Code to interact with source code. The repository containing the resources presented in this report can be found at https://github.com/quentinLeDilavrec/m1_internship. There are many improvements possible on this tool, some of them were mentioned earlier but the aim is to push developers on taking this tool further by adapting it to their needs.

References

1. W3Techs content management systems b. w3techs.com. Accessed: 2019-07-04.
2. Ivan Beschastnikh, Yuriy Brun, Jenny Abrahamson, Michael D Ernst, and Arvind Krishnamurthy. Unifying fsm-inference algorithms through declarative specification. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 252–261. IEEE Press, 2013.
3. Jordi Cabot. Wordpress: A content management system to democratize publishing. *IEEE Software*, 35(3):89–92, 2018.
4. Abram Hindle, Earl T Barr, Zhendong Su, Mark Gabel, and Premkumar Devanbu. On the naturalness of software. In *2012 34th International Conference on Software Engineering (ICSE)*, pages 837–847. IEEE, 2012.
5. Guofei Jiang, Haifeng Chen, Cristian Ungureanu, and Kenji Yoshihira. Multiresolution abnormal trace detection using varied-length n-grams and automata. *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)*, 37(1):86–97, 2006.
6. 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.