Testing Servo, the Parallel Browser Engine Project

Johannes Linke, David Schumann
Hasso Plattner Institute, University of Potsdam, Germany

Lecture “Software Testing, Verification and Analysis”
Winter Term 2015/2016
Supervision: Dr. Leen Lambers

24th January 2016

Contents

1 Introduction

The current generation of web browsers (Internet Explorer, Chrome, Firefox, and Safari) are mostly built on old and large code bases. Important design decisions whose results are visible still today have been made in a time where the IT landscape looked vastly different in terms of hardware, end-user device form factors, and security.

The Servo project aims at building a new browser engine for today's requirements. It is designed to be highly parallel, leveraging multi-core hardware and saving power on mobile devices, and exceptionally secure, by using a new programming language that eliminates certain classes of security-relevant programming errors.

The Servo project is developed and maintained by Mozilla Research in cooperation with Samsung. It is one of the largest projects being written in the Rust programming language, which is developed by Mozilla Research as well.

The authors of this report are learning Rust in their free time and wanted to test one of the major projects using it.

2 Current State of the Servo Project

In this section, we'll shortly discuss the choice of programming language for Servo, the contribution workflow, the testing and build infrastructure, and Servo's roadmap.

2.1 Main Programming Language: Rust

While some dependencies and build infrastructure code are written in other languages, the main programming language remains Rust. This works well with the main focus of Servo: a highly parallel, safe and fast alternative to current engines. Rust was designed to be just that: For instance, it is memory safe by design and requires the programmer to be very explicit with typing and error handling. This helps catching most errors at compilation time using static analysis. More details about such features of the Rust language are explained in section 5.1.

2.2 The Contribution Workflow

Servo is an open source project hosted on GitHub[1]. As it is common among open source projects, the core developers have established a well documented GitHub workflow[2]. It consists of the standard steps that go with any GitHub project:

1. Fork the original repository
2. Write and commit changes and new code into a new branch in the fork
3. Open a pull request from the new branch to the original repository
4. Discuss and fix any questions and suggestions by the reviewer
5. The reviewer states his approval, triggers the build bot to run automated tests and merges the pull request

Every pull request needs to pass the contribution checklist[3]. It covers technicalities such as: Every change needs to be accompanied by "tests relevant to the fixed bug or new feature"[4], which is to be evaluated by one of the reviewers from the core team. It also includes the Rust Code of Conduct[5] to ensure a friendly and welcoming environment for all contributors.

Defect testing is done through an extensive testing system. It is maintained by every developer as every code change needs to be accompanied by tests. The test suite is executed on every reviewed pull request by a build bot (bors-servo[6]) on a dedicated server farm[7]. This ensures that existing functionality is not broken by new code.

[1] `https://github.com/servo/servo`
[2] `https://github.com/servo/servo/wiki/Github-workflow`
[3] `https://github.com/servo/servo/blob/master/CONTRIBUTING.md`
[4] See footnote 3.
[5] `http://www.rust-lang.org/conduct.html`
[6] `https://github.com/bors-servo`
[7] `https://github.com/servo/servo/wiki/Buildbot-administration`

Further documentation can be found in the Servo wiki[8] hosted on GitHub. In addition to the above points it includes protocol notes on the biweekly meetings, long and short term goals, and a vast amount of documents aimed at helping with development and testing.

Tracking and Fixing of Issues As with most open source projects hosted on GitHub, the issue tracker[9] is used to track bugs, issues and features. It employs several handy features that make it easy to organize the frequent discussions and comments surrounding issues and pull request. Both get assigned numbers, which when mentioned in any comment automatically link to the corresponding issue or pull request. Issues automatically record when they were "mentioned" in another discussion. This builds a coherent map of issues and their corresponding fixes.

2.3 Current Testing Status

As of January 2016 the code base of Servo core (without dependencies) contains about 110,000 lines of code. These are tested with over 11,000 tests that are run using different commands:

- **./mach test-unit**: Runs 233 low level test binaries, that test important core code. Especially used in dependencies.
- **./mach test-wpt**: Runs 3764 Web Platform Tests (WPT).
- **./mach test-css**: Runs 7097 CSS Working Group tests.
- **./mach test-ref**: Runs 5 reference tests (including the famous acid2 test[10]). These consist of rendering two pages and comparing them.
- **./mach test-dromaeo**: Runs the Dromaeo test suite. This suite is geared towards JavaScript performance testing. It did not execute successfully when we called it.
- **./mach test-jquery**: Runs the jQuery test suite. It did not execute successfully when we called it.
- **./mach test-tidy**: Runs the source code tidiness check.

Some other commands are still documented in the wiki and help pages but are now deprecated (like ./mach test-content).

Currently servo suffers from a relatively large number of intermittently failing tests, currently about 70-80. This often slows down the process of contributing code, since pull requests with perfectly valid code might produce failing tests, requiring reviewers to issue new test runs until the intermittent failures disappear. We encountered this while sending a pull request that fixed about 130 code lints found by rust-clippy (see chapter 5.2.2).

[8] `https://github.com/servo/servo/wiki`
[9] `https://github.com/features`
[10] `http://www.webstandards.org/files/acid2/test.html`

Mozilla's "mach" System Mozilla's build system "mach" (from the German word "machen": to do) is at the core of the Servo project. Besides providing commands for running the various test suites as introduced above, it also makes it easy to compile and run the Servo binary or other related tools like rust-clippy (see chapter 5.2.2). While the Rust compiler brings its own package manager "cargo", which can also be used for compiling projects and running tests, it is not powerful enough (yet) to fulfill all the requirements a large project like Servo has, and thus its commands were wrapped inside mach.

2.4 Servo's Roadmap

Short-term goals of the Servo project are Windows support, finishing WebRender, a GPU-based rendering engine, and adding a graphical user interface (GUI) to Servo, whose window currently consists solely of the rendered output without GUI controls.

Being a research project, Servo's long-term outcomes are rather open-ended. Preliminary performance testing already shows major advantages over Gecko, the current engine of Mozilla Firefox, besides the security improvements that come naturally with choosing Rust. Furthermore large architectural improvements have been made possible by the clean-slate approach. Replacing Gecko with Servo in Mozilla's Firefox products appears like a reasonable goal, albeit the project is years away from implementing all of Gecko's functionality. As a compromise, Mozilla plans to use smaller, isolated modules written for Servo in Gecko, and will evaluate the feasibility of a standalone Servo browser, most likely starting with the Android platform[11].

[11] `https://github.com/servo/servo/wiki/Roadmap`

3 Verification and Validation (V&V)

Tests must be written for all changes made to the Servo code base that change functionality. While this is enforced quite rigorously, other aspects of V&V are not as ingrained into the Servo work flow.

3.1 The five V&V questions

When do verification and validation start? When are they complete? For Servo, V&V started with the very beginning of the project. In fact, the sixth commit to the project added a "make test" command and a first test.

As with many open source project that are developed by the community, the stakeholder or "customer" is the community itself, which means that developer and customer are often the same person. In the case of Servo the software is validated in bi-weekly meetings[12] where both short-term next steps and long term milestones are discussed.

V&V will likely never end for Servo. Even after the release there will always be bug fixes or new features that need to be tested as well, especially since the web is an evolving platform. One could argue that V&V ends when the project is abandoned but one can hardly call that a complete state.

What particular techniques should be applied during development? Servo is already a well tested system. But it lacks a lot of even basic V&V features like a report on any kind of test coverage. Until now every pull Request is supposedly tested fully by a test accompanying it. But an automatic check in the form of statement coverage would help to find edge cases that might have slipped through the review. Furthermore tests about parallel execution would be an important addition to the Servo test suite. These are hard to write and execute but a project having concurrency as one of its main goals should test it.

How can we assess the readiness of a product? As a research project it is hard to assess its completeness. The overarching long term goal of Servo is to replace the current web engine Gecko of the Mozilla Firefox Browser. But this is estimated to take at least several more years. The progress of Servo is currently assessed in biweekly meetings of the Servo core team. The mile stones and road map that results from these meetings are a good way to keep track of short- and long-term goals. It would help to have an assessment by an outside, non-developer party to prevent building unnecessary features and to keep a fresh new perspective regarding V&V on the project.

How can we control the quality of successive releases? The Servo project already has the basic automated testing feature covered: Regression tests with the merge of every pull request. This assures that no change makes the code base worse according to the test suite, which in turn places the whole quality assurance on software tests and makes

[12] `https://github.com/servo/servo/wiki/Meetings`

it even more important that these are as complete as possible. In addition to testing parallel execution, performance tests could also be part of Servos test suite. Benchmarks of Servo vs Gecko were computed at some point[13], but including them in the test suite would help to find performance degrading changes that slow down execution from one release to another.

How can the development process itself be improved? The development process used by Servo is very common among many professional open source projects with no exorbitant budgets. Without the resources to hire a separate testing team or a V&V specialist, the approach to have every developer test his own code and do full automated regression testing at every step is a successful model. It is hard to improve on this without raising the budget.

3.2 Test Classifications

Validation and Defect Testing The biggest part of the test suite used in the Servo project is focused on validation testing. The whole web platform test (WPT) suite and CSS WG suite are geared to validate that all aspects and features of HTML, CSS and JavaScript are correctly implemented. The few reference tests contain defect tests such as the famous acid2 test. Here a lot of different HTML and CSS rules are used to draw a smiley. If any of those rules are not implemented correctly, the smiley will not be drawn correctly.

Figure 1: Acid2. Correct rendering on the left. All others show rendering flaws

Development, Release and User Testing These three forms of testing are all present but hardly differentiable in the Servo project. This is due to the fact that the core development team combines developers, release testers and, in a certain way, users in the same team. The tests are mostly used during development for regression testing, but there is currently no separate test suite or separate test team intended for release testing. It is done by the community but mostly the core members as well. Finally the core developers could also be seen as the users, as Servo is no standalone software intended

[13] https://www.phoronix.com/scan.php?page=news_item&px=MTgzNDA

for end users, but instead more a library meant to be integrated into a GUI application like Mozilla's Firefox. Thus the core team develops, releases and decides when Servo is ready for the user: themselves. Being a library intended for internal development teams makes testing easier, since most requirements in terms of standards conformance and the exposed API can be tested rather easily. On the other hand, the team might be missing an outside perspective on the readiness and validity of the product.

Unit-, Integration- and System-testing As can be seen in chapter 2.3, the unit tests only make up a fraction of the overall test suite. Integration tests are sometimes used in the dependencies, but do not get executed when testing the main Servo binary. The biggest part are the system tests: All WPT and CSS WG tests are system level tests as they execute the Servo binary and use the whole rendering engine by rendering a complete web page, albeit testing only for small sets of features each. While the overall testing strategy is in no way complete, it forms a practical approach to V&V that is very efficient and complete enough for the given requirements, and does not slow down development with more elaborate processes that would be unnecessary in this context.

4 Test Automation

This section introduces several tools that run tests on software written in Rust and collect coverage.

4.1 cargo test

Rust's package manager cargo already has tools for running tests. Testing is as easy as annotating any method with `#[test]`, and then running `cargo test`. Cargo will then build a testing binary, run all annotated methods and print out a report that looks like this:

```
    Compiling gfx_tests v0.0.1 (file:///home/johannes/servo/components/servo)
    Running /home/johannes/servo/target/debug/deps/gfx_tests-9666be7e60be2090

running 6 tests
test text_util::test_transform_compress_none ... ok
test text_util::test_transform_compress_whitespace ... ok
test text_util::test_transform_compress_whitespace_newline_no_incoming ... ok
test text_util::test_transform_compress_whitespace_newline ... ok
test text_util::test_transform_discard_newline ... ok
test font_cache_thread::test_local_web_font ... ok

test result: ok. 6 passed; 0 failed; 0 ignored; 0 measured
```

Any crashes inside the tests (e.g. by failing assertions) are interpreted as test failures, methods that exit normally are successes.

Cargo's builtin testing features are not meant for large-scale applications and test suites. There are some smaller libraries like stainless[14] that are extending on cargo's features, e.g. by adding setup and teardown methods, but such libraries haven't seen much use yet.

Servo uses `cargo test` for running the unit tests. For running the more complex test suites, it builds the binaries with cargo but uses python scripts to execute the individual tests. See chapter 2.3 for more details.

4.2 Coverage Tools gcov and kcov

No type of coverage has ever been calculated for the whole Servo project, not even basic statement coverage. As this is one of the most common ways to indicate the completeness of a test suite we looked into generating a coverage report for Servo. We also hoped to find parts in the code base that are not yet covered by tests, which we could then examine further.

gcov Gcov[15] is a widely used source code coverage analysis tool that comes with the GNU Compiler Collection (GCC). It works with any LLVM-generated files. As the Rust compiler rustc uses LLVM as its backend we tried gcov first to generate a useable test coverage report. Unfortunately, rustc does not yet support all of LLVM's flags that gcov needs to produce coverage

[14] `https://github.com/reem/stainless`
[15] `https://gcc.gnu.org/onlinedocs/gcc/Gcov.html`

data[16]. This issue has already been reported[17] but was postponed a year ago due to the 1.0 release of rustc[18]. We briefly considered implementing gcov support for rustc, which would enable all rust projects to easily generate test coverage reports, but quickly realized that it would be out of scope.

kcov The community suggested to try the tool kcov[19], which has been used successfully for smaller Rust projects. It works by inserting breakpoints into every line of code and recording when they are triggered. This increases execution time significantly but proves to be a robust way to generate code coverage, as it works on arbitrary binaries with debug symbols. Therefore, generating correct debug symbols is enough to let kcov produce a test coverage report. Unfortunately, it does not compile on Mac OS or Windows, so we had to run our tests in a Linux (Ubuntu) virtual machine, which slows down test execution even more. Another issue we faced was that the version provided by the package manager (apt-get) was out of date. Compiling kcov from source failed at first, but was quickly fixed by adding another dependency (libbfd-dev) to the build flags.

Since Servo relies on mach to prepare the running environment, we couldn't start kcov on the Servo binary directly. Instead, our first coverage report was created by moving the Servo binary to servo_orig and placing the following script in its place:

```
#!/bin/sh
kcov --verify --exclude-pattern=/.cargo,../src /home/johannes/servo/cov
 /home/johannes/servo/servo-sync/target/debug/servo_orig "$@"
```

Now executing **./mach test** would run our script instead of the Servo binary, which passes all parameters to the original Servo binary but also executes kcov. In theory this should have yielded a complete test coverage report as kcov should also properly merge several executions of the same binary. This would be a necessary feature since mach restarts Servo for each of its tests. Unfortunately, each test took about 30-60 seconds, which implied a whole week of running time to generate a complete coverage report. Additionally, the merging of reports was not working correctly for us, but as the long execution time already disqualified kcov as a feasible coverage tool, we did not continue working on fixing the merging problem.

Additionally, the partial coverage reports we got differed from test run to test run. Lines of code covered by one exemplary test ranged from 26976 to 27104 throughout several iterations. These fluctuations are probably caused by non-deterministic behavior of Servo but could also be caused by inaccurate coverage reports on kcov's side. We can not be completely sure as it seemed infeasible to check the roughly 30,000 lines that were executed in one test run and trace the execution path.

[16] Section 4 in `http://ltp.sourceforge.net/documentation/technical_papers/gcov-ols2003.pdf`
[17] `https://github.com/rust-lang/rfcs/issues/646`
[18] `https://github.com/rust-lang/rust/issues/690`
[19] `https://simonkagstrom.github.io/kcov/`

4.3 Fuzzy Testing Using afl.rs

afl.rs[20], based on American fuzzy lop[21], is a package that allows finding bugs in Rust code through fuzzy testing. It works by taking exemplary input over stdin and mutating it. As a consequence, it requires that the binary to test reads its input over stdin. As Servo does not do this, we looked at a smaller component of Servo, the html-parser html5ever[22] instead. Here we built a test binary that would read html code over stdin and try to parse it. During the setup, we realized that afl.rs currently requires an old version of Rust (1.2) due to a bug in newer Rust versions[23]. Html5ever, on the other hand, required at least Rust 1.4 to work. This problem deemed fuzzy testing the html5ever component or Servo as a whole impossible without extensive debugging of the Rust compiler.

4.4 Doctests

As many other languages, Rust defines a syntax to document source code, and as in many other languages, the documentation can contain example code. It might look like this:

```
/// Constructs a new `Rc<T>`.
/// # Examples
/// ```
/// use std::rc::Rc;
/// let five = Rc::new(5);
/// ```
pub fn new(value: T) -> Rc<T> {
    // implementation goes here
}
```

In Rust however, these usage examples are executed and tested as part of `cargo test`. To simplify this process, the example code is automatically completed e.g. with a main function to make it a valid Rust program. More complex examples can be built, e.g. spanning multiple functions, and individual lines can be hidden to focus the user-visible documentation on the relevant parts. The full set of testing facilities, e.g. assertions, is available, making it possible to avoid outdated code examples altogether.

[20] https://github.com/frewsxcv/afl.rs
[21] http://lcamtuf.coredump.cx/afl/
[22] https://github.com/servo/html5ever
[23] https://github.com/frewsxcv/afl.rs/issues/11

5 Automatic Static Analysis

This section discusses various automatic static analysis (ASA) features and tools available for projects written in the Rust language.

5.1 ASA Features of the Rust Language

Rust has an extensive set of static analysis techniques built-in. They became necessary to achieve the first design goal of the language, safety, while not impeding the second and third, speed and concurrency.

For Servo, these ASA techniques are a crucial part of its security concept. For instance, a large fraction of the security vulnerabilities in Gecko, the engine currently used in Firefox, are related to use-after-free bugs, which are completely eliminated by the Rust compiler.

In the following we describe some of Rust's language concepts that let the Rust compiler prevent certain classes of programming errors that frequently occur in other languages. Finding such errors through static analysis is not always possible in other languages since they are not expressive enough and the programmer can't provide enough information in the code for an ASA tool to perform such an analysis.

Ownership In Rust, variables and objects are always "owned" by a scope (usually, a block of code delimited by curly braces) or another object. This ownership can be moved, for example by calling a method and handing over the object as a parameter, but the language provides no way to copy the ownership to an object. This way, the owner of an object can be statically determined. More importantly, each object can be deleted if its owning object is deleted or the owning scope ends. This way, it can be determined at compile-time when each object has to be deleted. This completely eliminates use-after-free bugs, a common source for crashes and security vulnerabilities. It also prevents most classes of memory leaks. Being a compile-time technique, this does not imply any run-time cost like garbage collectors do, which are used by other, less performance-oriented languages.

Borrowing To allow for more flexible movement of data, objects can be temporarily borrowed into other scopes. The compiler statically ensures that there exists either exactly one mutable borrow, through which the object can be mutated, or any number of immutable borrows. By preventing shared mutable state, a lot of programming errors are made impossible, for example data races. A data race occurs if two threads access the same data in an unsynchronized fashion and at least one of them writes. Since there can be only one thread writing at one specific data point at the same time, data races cannot happen. Iterator invalidation is another problem detected at compile time: Since iterating over a container borrows the container, modifying the container, which would need to happen through a mutable borrow, is not possible.

Safe memory accesses Likewise it is not possible to access uninitialized memory and, since pointers are not part of the language, null pointers do not exist as well. Not an ASA technique but related to memory safety, are the run-time bounds checks that are mandatory for all Rust data containers.

Code-generating macros Rust features a powerful macro syntax. It has several advantages over C and C++, where macros simply perform text replacements. Macros in Rust are hygienic, can generate different code depending on the type of input and take a variable number of parameters. In fact, a library has been written that expands regular expressions to native Rust

code, enabling a lot of optimization potential as well as compile-time syntax checking on the expression[24]. Another library enables specifying parsers through a macro-based API[25]. The resulting parsers are extremely fast and guaranteed to be safe.

Other features There are several smaller features, restrictions and well-chosen defaults of the Rust language that improve the robustness of programs, some of them being ASA techniques: For example, variables are immutable by default, encouraging a less error-prone programming style. Matches (the Rust-equivalent of switch-case statements) must be exhaustive, that is each possible value of the variable that is matched must be handled by a branch, making it impossible for the programmer to forget a case. Integer overflows, being another source of programming errors and security vulnerabilities, make Rust programs crash when compiled in debug mode. And finally, expected runtime errors such as IO-errors must be explicitly handled by the programmer or will lead to an immediate crash of the program, as opposed to undefined behavior in e.g. C and C++.

5.2 Lints

Lints are warnings about suspicious or non-optimal code. The Rust compiler itself or additional compiler plugins can report such lints.

5.2.1 Lints Reported by the Rust Compiler

Besides the language features outlined above, the rust compiler has some smaller ASA capabilities in the form of lints, which are reported as warnings during compilation. They include unused imports and variables, unnecessarily mutable variables, dead code, variable and function names not following the naming conventions, and unconditional recursions. There are some more lints that are disabled by default like missing documentation of public interfaces and the usage of the `unsafe` keyword.

5.2.2 rust-clippy

Clippy[26] is a compiler plugin that expands on the lints shipped with the Rust compiler and checks for common patterns that indicate inefficient, needlessly complex or unidiomatic Rust code. It is exceptionally easy to use since the complete procedure of setting it up consists of adding one line to the cargo configuration and main code file. Having done that, clippy will be run whenever any code in the project is compiled.

[24] `https://github.com/rust-lang-nursery/regex`
[25] `https://github.com/Geal/nom`
[26] `https://github.com/Manishearth/rust-clippy`

Example of clippy output:

```
servo/components/script/cors.rs:95:9: 98:10 warning: you seem to be trying to use match for
    destructuring a single pattern. Consider using ‘if let‘, #[warn(single_match)] on by default

servo/components/script/cors.rs:95   match referer.scheme_data {
servo/components/script/cors.rs:96       SchemeData::Relative(ref mut data) => data.path = vec![],
servo/components/script/cors.rs:97       _ => {}
servo/components/script/cors.rs:98   };

/servo/components/script/cors.rs:95:9: 98:10 help: try
if let SchemeData::Relative(ref mut data) = referer.scheme_data { data.path = vec![] }

for further information visit https://github.com/Manishearth/rust-clippy/wiki#single_match
```

The lints reported by clippy vary in type. Most of them are related to programming style or detecting patterns that were common in old Rust code but can be written simpler in newer versions of Rust. The `single_match` lint in the example above is such a lint; the `if let` construct wasn't available until late 2014. Other lints detect type casts that may lead to loss of precision or truncation, boolean expressions that are tautologies, or obvious bugs like out of bounds accesses with constants. Clippy also includes two lints that warn on complex (as in deeply nested) types and methods with a high cyclomatic complexity.

Servo already has clippy integrated, it can be run through `./mach clippy`. In practice we discovered that Servo used an old version of clippy, which was incompatible with Servo's version of Rust. The newest version of clippy didn't work either due to another incompatibility. Getting the current clippy to work for servo thus involved manually picking commits into a custom branch of clippy. Running it, we made the following findings:

- Clippy reported 417 warnings.
- Of these, 122 were in autogenerated code.
- Another 11 were false positives. Several more might be false positives as well, which we haven't investigated further.
- The vast majority of lints pointed at code that could be simplified.
- Some of these simplifications also led to potentially faster code.
- There were eight warnings on complex methods and two warnings on complex types.
- Not a single reported lint indicated a possible bug.

While working through the warnings, we fixed about 130 of them and made a pull request to Servo[27] and reported several false positives on clippy([28],[29],[30]).

[27] `https://github.com/servo/servo/pull/9123`
[28] `https://github.com/Manishearth/rust-clippy/issues/528`
[29] `https://github.com/Manishearth/rust-clippy/issues/529`
[30] `https://github.com/Manishearth/rust-clippy/issues/532`

5.2.3 Servo's Custom Lints

Additional lints are relatively easy to add to the compilation process. While clippy makes use of this feature for generally applicable lints, Servo itself adds a few lints that are specialized to Servo's use case. For example, some objects in the Servo codebase rely on being placed on the stack instead of the heap to provide certain safety guarantees. A compile-time lint prevents encapsulating these objects in containers that would place them on the heap. Furthermore, objects that have representations in JavaScript-controlled memory are statically checked for being either a root registered with the JavaScript garbage collector or contained within such a root.

5.3 rust-fmt

Although not strictly being an ASA tool, rust-fmt[31] should not go unmentioned. Rust-fmt formats rust code according to the Rust style guide[32] and an extensive set of customization parameters. With an ecosystem this young, rust-fmt and the style guide itself are still in a very early stage of development.

[31] `https://github.com/rust-lang-nursery/rustfmt`
[32] `https://github.com/rust-lang/rust/tree/master/src/doc/style`

6 Conclusion

Rust as a language is an excellent choice in areas where previously C++ was the only viable option. With its focus on concurrency and safety, it has major advantages over C++ whenever a project introduces concurrency or becomes complex enough to exhibit hard-to-debug problems related to invalid memory accesses. Eliminating these and other problems at compile time, Rust removes large sources of security vulnerabilities and programmer frustration.

Servo was in the lucky position to enter a field where an extensive set of tests (e.g. Web Platform Tests, CSS WG tests, reference tests) where already available. Combined with the policy to accompany each pull request with relevant test cases, Servo can be considered a well-tested application. However, no automated tools are used to verify that testing is complete by any metric, and thus this is likely not the case.

This is partly due to the fact that such tools are virtually nonexistent. Besides kcov, which doesn't seem to be suitable for projects of Servo's size, there are no tools that determine any coverage metrics for Rust projects, and tools in related fields like testing techniques (e.g. afl.rs) and static analysis (e.g. clippy) are scarce and in early stages of development. With Rust as an excellent basis for security- and stability-focused software, it will be exciting to see how the ecosystem develops.