xWIDL: Modular and Deep JavaScript API Misuses Checking Based on eXtended WebIDL

Zhen Zhang
University of Science and Technology of China, China
zgzhen@mail.ustc.edu.cn

Abstract

JavaScript is the de facto language of the Web, but is notoriously error-prone to use. 65% of common bugs like *undefined/null variable usage* are DOM-related. Besides DOM, JS APIs are also expected to manipulate graphic hardware and asynchronous I/O, which makes the condition even worse. Although WebIDL provides a formal contract between JS developers and platform implementation, its expressivity is too limited to support *deep* checking of API misuses. We propose the eXtended WebIDL (**xWIDL**) language and a *modular* API misuses checking framework based on xWIDL. We discuss how to handle the data exchange between JS analyzer and SMT-based verifier. Finally, we test our implementation in a case study manner and report our findings on its efficiency and modularity.

Categories and Subject Descriptors D.2.4 [*Software Engineering*]: Software/Program Verification; F.3.1 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs

Keywords JavaScript, WebIDL, static analysis, interface definition language, program verification, data-flow analysis

1. Motivation and Related Work

JavaScript (abbr. JS) is a very popular language used in Web, but its users suffer greatly from unexpected bugs[1], mainly due to its *lack* of a static, strong and powerful type system. Previous work on JavaScript static analysis include WALA, TAJS[3], SAFE[5], JSAI[8], Flow etc., but they didn't address checking of the *platforms APIs*, like browser DOM APIs, Node.js APIs or third-party libraries. Handling them properly is **vital**, since JavaScript, as a scripting language, is meant to use APIs intensively.

SPLASH Companion'16, October 30 – November 4, 2016, Amsterdam, Netherlands
ACM. 978-1-4503-4437-1/16/10...$15.00
http://dx.doi.org/10.1145/2984043.2998545

To solve this, new TAJS[4] hardwired the DOM APIs inside analyzer. However, this approach is *unmaintainable* and *unscalable*: when an API changes, we have to update the analyzer manually; plus that DOM is just one of many platforms[9], it is impractical to keep track of everything.

To improve scalability, we advocate WebIDL[7]: an interface definition language used in modern Web standards. It is coarsely a collection of API names and their types. $\mathsf{SAFE}_{\mathsf{WAPI}}$[2] first took advantage of *vanilla* WebIDL to check API misuses such as wrong number/type of arguments. But this is *not* enough to tackle the complex beast like WebGL applications: For example, when you call `gl.createBuffer()`, the returned `WebGLBuffer` object is *implicitly* registered in the *context object* `gl`, and some APIs like `gl.bindBuffer()` takes this fact as pre-condition. Unfortunately, all these are written *informally* in standard since *vanilla* WebIDL can't express this.

Motivated by these problems, we did the following contributions: 1) Propose the xWIDL language to extend WebIDL with semantic-level specification of platform APIs. 2) Propose a modular API checking framework and design a protocol for inter-module communication. 3) Implement `xwidl-engine` to enforce the protocol and check the xWIDL specification using SMT-based verifier.

2. Approach

We first define our extension to WebIDL, next present the protocol design, then discuss how to transform data of different representations between analyzer and verifier.

2.1 xWIDL Language

We extend WebIDL by adding five kinds of *annotations*, which are inserted as comments in existing definitions:

- `ghost`: Modeling of hidden states
- `requires`: Pre-condition for an operation
- `ensures`: Post-condition for an operation
- `effects`: Imperative specification of operation effect
- `callbacks`: Specification of callback behavior

Here is an example snippet of xWIDL:

```
callback Handler = void (DOMString);
dictionary Request { long length; };
interface Reader {
  ///- ghost attribute long reqs;
  short add(Request? req, Handler cb);
  ///- requires req != null
  /*/- ensures if req.length > 0
              then ret == 1
              else ret == 0 */
  ///- callbacks cb when (req.length > 0)
                  with "hello world"
  ///- effects { this.reqs++; }
}
```

First, *ghost* state `reqs` detects ordering violation bugs between calls, and we use *effects* clause to update it. Second, *requires* detects null parameter bug. Finally, *ensures* and *callbacks* improve overall analyzing precision.

2.2 Client-Server Architecture and Protocol Design

Unlike the monolithic architecture of TAJS and $\mathsf{SAFE}_{\mathsf{WAPI}}$, we **decouple** the API checking logic (i.e. *server*) from analyzer (i.e. *client*) by defining a *protocol*. This architecture enjoys high degree of **modularity**: the changes on client side will never effect the server side and vice versa; Also, it is highly *reusable* since existing analyzers can simply piggyback on our checker to start detecting API misuse.

The *protocol* is a standard query interface between the client and server. It first defines a minimal set of *common language definitions*, including primitive value and assertion expression. It then describes the process of establishing connection, bootstrapping session, exchanging queries, and finishing. Format of query/reply is designed to capture the essence of API calling/returning, while satisfying the basic need of verifier and restriction of general analyzer.

2.3 Transformation of Data between Different Representations

The key problem to protocol enforcement is how to exchange data (value) between two ends. In data-flow analyzer, symbol's value is an element of *abstract semantic domain* lattice, and symbol `x` can be both `bool` and `int` at the same time. Information is highly summarized this way, thus improving the analyzer's efficiency. However, in the SMT-based verifier, a symbol is strongly-typed and its possible value is defined in terms of logical constraints on it.

For example, calling the `add` method in snippet (2.1) returns an integer to client, which must ultimately be transformed into one of { Positive, Negative, Zero, $\bot$, $\top$ }. How to reflect the value constraints in verifier space into such an abstract number? We introduce the idea of *domain probing*: Apply a set of *domain assertions* $\{x > 0, x = 0, x < 0\}$ on variable to *probe* its possible ranges, then collect verification results in a specific *assertion context* as an array of boolean flags. If analyzer receives, for example, $[true, false, false]$, then it knows that $x > 0 \wedge x \neq 0 \wedge x \geq 0$, which means that $\bar{x} =$ Pos would be a good enough return value.

3. Preliminary Results

We implemented the design in Haskell, and used Dafny[6] as verification backend. In `xwidl-engine` package, we developed protocol interpretation strategies and *verification-unit* generation algorithms. While benefiting from many Dafny language features, such as datatype and class abstraction, we found it non-trivial to encode effectful statements in the presence the framing restriction. The successful application of Dafny confirms the correctness of our checking logic.

We conducted a study of 12 use cases by a simulated client. The support for ordinary WebIDL features is comparable to $\mathsf{SAFE}_{\mathsf{WAPI}}$. Besides, our *requires-ensures-effects* specification triple and ghost state modeling are shown to be effective. However, our simple implementation is slow: a complex query could easily take up to seconds, and most time is spent on running Dafny verification.

4. Conclusion and Future Work

In conclusion, we show that it is possible to check JavaScript API misuses in a *deep* and *modular* way: Through annotation and verification, we can reason about more misuses than $\mathsf{SAFE}_{\mathsf{WAPI}}$; We also show the possibility of decoupling API checker from analyzer, thus enabling a more modular and general framework than existing ones.

In the future, we plan to improve the support of *infinite* lattice representation, instantiate xWIDL with a production-level analyzer, and improve the efficiency of checker.

Acknowledgments

Special thanks to Prof. Yu Zhang for her help and guidance.

References

[1] F. Ocariza, K. Bajaj, K. Pattabiraman, and A. Mesbah. *An empirical study of client-side JavaScript bugs* IEEE 2013

[2] S. Bae, H. Cho, I. Lim, and S. Ryu. *SAFEWAPI: Web API misuse detector for web applications* FSE 2014

[3] S. H. Jensen, A. Møller, and P. Thiemann. *Type Analysis for JavaScript* SAS 2009

[4] S. H. Jensen, M. Madsen, and A. Møller. *Modeling the HTML DOM and browser API in static analysis of JavaScript web applications* ESEC/FSE 2011

[5] H. Lee, S. Won, J. Jin, J. Cho, and S. Ryu. *SAFE: Formal specification and implementation of a scalable analysis framework for ECMAScript* FOOL 2012

[6] K. R. M. Leino. *Dafny: An Automatic Program Verifier for Functional Correctness* LPAR 2010

[7] C. McCormack. *Web IDL W3C* 2012

[8] V. Kashyap, K. Dewey, E. A. Kuefner, J. Wagner, K. Gibbons, J. Sarracino, B. Wiedermann, and B. Hardekopf. *JSAI: a static analysis platform for JavaScript* FSE 2014

[9] A. Guha, B. Lerner, J. G. Politz, and S. Krishnamurthi. *Web API verification: Results and challenges* In *Analysis of Security APIs*, 2012