JaVerT: JavaScript Verification and Testing Framework

Invited Talk

Philippa Gardner
Imperial College London, UK
pg@imperial.ac.uk

ABSTRACT

We present a novel, unified approach to the development of compositional symbolic execution tools, which bridges the gap between traditional symbolic execution and compositional program reasoning based on separation logic. We apply our approach to JavaScript, providing support for full verification, whole-program symbolic testing, and automatic compositional testing based on bi-abduction.

CCS CONCEPTS

• **Theory of computation → Separation logic**; **Program analysis**; *Logic and verification*; • **Software and its engineering →** *Automated static analysis*;

KEYWORDS

Symbolic execution, Separation logic, Formal semantics, JavaScript

ACM Reference Format:
Philippa Gardner. 2018. JaVerT: JavaScript Verification and Testing Framework: Invited Talk. In *The 20th International Symposium on Principles and Practice of Declarative Programming (PPDP '18), September 3–5, 2018, Frankfurt am Main, Germany.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3236950.3236974

ACKNOWLEDGMENTS

This invited talk is based on joint work with José Fragoso Santos, Petar Maksimović, and Gabriela Sampaio.

Compositional Symbolic Analysis

Traditional symbolic execution has been widely applied to the analysis of programs written in both static and dynamic languages. To this day, however, it remains mainly focussed on whole-program analysis. Such an analysis has two drawbacks. First, it is not aligned with the way in which programmers write their code. Developers tend to import various external modules, so the assumption that the entire codebase is always at our disposal for analysis is tenuous at best. Second, it does not scale well, as minor changes to one part of the code require the entire analysis to be repeated. We believe that an analysis that can be used to reason about programs as they are written and can scale to large codebases needs to be compositional.

Separation logic provides compositional analysis of programs. Recent tools based on separation logic have been shown to be tractable for real-world programs embedded within large software environments. The prime example of this is Infer [1], a fully automatic compositional tool that is part of the code review pipeline at Facebook and is aimed at lightweight bug-finding for programs written in static languages such as C, C++, Java, and Objective C. Infer owes its scalability precisely to the compositionality of the underlying analysis. In particular, it can generate function summaries that can then be re-used in the analysis of other functions.

We propose a novel approach to the development of compositional symbolic execution tools, which connects traditional symbolic execution and compositional program reasoning based on separation logic. Tool developers familiar with both forms of analysis are likely to have an intuition that such a connection is possible. We make this intuition precise. We develop a unified theory of compositional symbolic execution, designed in such a way that it can be directly implemented. By doing so, we bring tangible benefits to both worlds: symbolic execution tools gain access to succinct summaries in the form of separation logic specifications; and separation logic proof systems become tightly linked to efficient implementations based on symbolic execution.

We apply our approach to provide compositional symbolic execution for JavaScript. We stress that JavaScript is just an exemplar, and strongly believe that our overall approach is language independent. We introduce JaVerT 2.0, a verification and testing framework for JavaScript (ECMAScript 5 Strict). JaVerT 2.0 supports: full verification, which significantly improves on our previous semi-automatic verification tool, JaVerT [5]; whole-program symbolic testing, which is two orders of magnitude faster than our previous symbolic tool, Cosette [4]; and, for the first time, automatic compositional testing based on bi-abduction. The implementation of JaVerT 2.0 is directly guided by our unified theory.

Just as it was the case for JaVerT and Cosette, the analysis of JaVerT 2.0 is performed on JSIL, our intermediate representation for JavaScript. JSIL is a simple goto language that natively supports the main dynamic features of JavaScript, namely extensible objects, dynamic property access, and dynamic function calls. It comes with JS-2-JSIL, a trusted, thoroughly tested compiler from JavaScript to JSIL. We have purposefully designed the memory model of JSIL to match the memory model of JavaScript, allowing us to easily lift the results of our analyses done on compiled JSIL code back to the original JavaScript code.

We have successfully applied JaVerT 2.0 both to whole-program and compositional symbolic testing of real-world JavaScript libraries, finding previously unknown bugs, and to the verification of small data structure libraries. The results that we have obtained so far indicate that JaVerT 2.0 is scalable. In the future, we will aim at highly used JavaScript frameworks, such as jQuery or React.js.

PPDP '18, September 3–5, 2018, Frankfurt am Main, Germany
© 2018 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-6441-6/18/09.
https://doi.org/10.1145/3236950.3236974

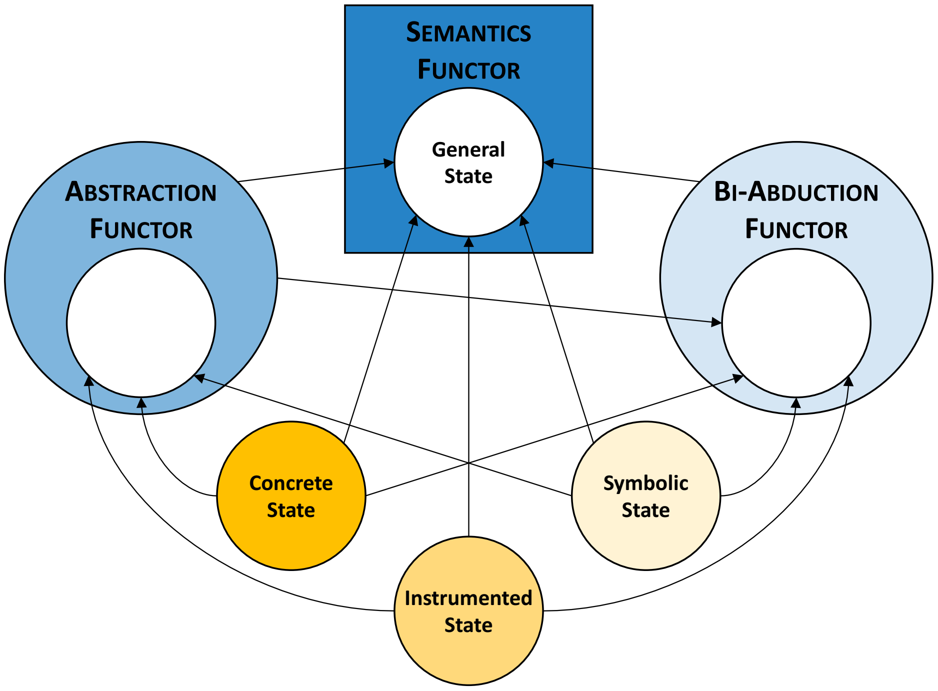

Figure 1: Unified Symbolic Analysis: Functors + State Instantiations

Methodology

Symbolic analyses tend to closely follow the semantics of their targeted languages. This results in overly verbose, repetitive formalisms and implementations with a lot of code duplication. We propose a new methodology for designing compositional symbolic analyses that factors out the overlap between the semantics and the analysis, leading to streamlined formalisms with minimal redundancy and more modular implementations. The key insight of our methodology consists of splitting the semantics of the targeted language, in our case JSIL, into two components: a *Semantics Functor* and a state instantiation.

■ **The Semantics Functor (SF).** The Semantics Functor is the bedrock for both the formal development and the implementation of symbolic analyses. It describes the behaviour of the language commands in terms of general state functions that capture the fundamental ways in which one can interact with the JSIL state: for example, evaluating an expression, allocating a new object, retrieving the value of a given property of a given object, checking whether or not the state satisfies a given constraint, etc. More concretely, the Semantics Functor is parameterised by a state signature that it uses to define the behaviour of JSIL commands, returning a JSIL semantics that is consistent with the state instantiation. In JaVerT 2.0, we parameterise the Semantics Functor with three state instantiations: *concrete*, *instrumented*, and *symbolic*, respectively obtaining the concrete, instrumented, and symbolic semantics of JSIL.

● **The Concrete Semantics.** The concrete semantics allows us to run JSIL programs concretely, which is essential for ensuring that the Semantics Functor captures the intended behaviour of the language. It also allows us to test our infrastructure against the ECMAScript official test suite, Test262 [3], by first compiling it to JSIL and then executing it concretely. In this way, we establish trust in the compilation.

● **The Instrumented Semantics.** We know from separation logic [10] that the frame property is essential for local compositional reasoning about programs that alter the heap state. Intuitively, the frame property means that the output of a program does not change when the state in which it is run successfully is extended. The JSIL semantics, however, just like JavaScript semantics, does not observe the frame property: it is possible to change the output of a JSIL/JavaScript program and even introduce bugs by extending the state in which the program was run. Our solution is to design an instrumented semantics [4] that exhibits the frame property by explicitly keeping track of object properties that we know are not present. By having the instrumented semantics as a proper interim stage between the concrete semantics and the symbolic semantics, we obtain more modular reasoning and substantially simpler proofs than previous approaches based on weak locality [6, 7].

○ **The Symbolic Semantics.** The symbolic semantics represents the core of our symbolic analysis. It is obtained by lifting the instrumented state instantiation to the symbolic level, following standard approaches [13, 14], and plugging it into the Semantics Functor. Unlike the majority of the existing bug-finding symbolic execution tools for JavaScript [8, 12, 15] which target specific bug patterns and are not rigorously formalised, the symbolic semantics underpinning JaVerT 2.0 is fully general and proven sound.

Error Reporting. One of the main novelties of our proposed architecture is its unique emphasis on *error reporting*. State instantiations

	State Functors			
State Implementation	**Semantics**	**Abstraction**	**Bi-Abduction**	**Abstraction + Bi-Abduction**
Concrete	Concrete Execution	Executable Specifications	-	-
Instrumented	Instrumented Execution	Executable Specifications	Automatic Concrete Test Synthesis	Automatic Concrete Test Synthesis with Specs
Symbolic	Whole-Program Symbolic Testing	Verification	Automatic Symbolic Testing	Automatic Compositional Testing

Table 1: Use cases: combining functors and state implementations

are required to include an error reporting mechanism that accurately describes the causes of failure whenever an error occurs. We distinguish two types of errors: *must*-errors and *may*-errors. Must-errors correspond to the well-known errors that can occur during the execution of a program; for instance: **(i)** type errors, when the type of a given value is wrong, and **(ii)** resource errors, when a command requires a given spatial resource and we know with certainty that this resource is absent. May-errors occur when the semantics does not have enough information to execute the command at hand; for instance, during a property lookup, the inspected object property might be missing, in which case we do not know whether or not it exists. We note that may-errors only occur at the instrumented and symbolic levels, where states are interpreted as partial. At the concrete level, all errors are must-errors, as we have full information about the state.

Must-errors can only be reported to the user; they cannot be corrected as we know that they must [sic!] occur. In contrast, may-errors can either be corrected or reported. In both cases, the missing information is added to the state. When reporting a may-error to the user, we add the information that turns the error into a must-error. When correcting a may-error, we add the information that is required by the semantics to successfully execute the command that originally triggered the error.

The Abstraction Functor (AF). Compositional program reasoning mandates that we are able to re-use the results of analysing a given procedure when analysing a procedure that calls it. In JaVerT 2.0, we use separation-logic specifications as procedure summaries. To this end, we design a new functor, the Abstraction Functor, which receives a state signature as input and generates a new state signature with a built-in mechanism for executing procedure calls abstractly using separation logic specifications. The main role of the Abstraction Functor is to establish the connection between the JSIL assertion language and JSIL states. The key innovation is that instead of defining the meaning of assertions in terms of concrete states using a standard *satisfiability relation* [10], the meaning of assertions is defined with respect to an arbitrary state signature, connecting our assertion language to concrete, instrumented, and symbolic JSIL states *in a single place*. Additionally, the Abstraction Functor allows the user to describe their data structures using *inductive predicates*, and also provides mechanisms for their *automatic* folding and unfolding during the analysis.

By plugging the symbolic state into the Abstraction Functor and the resulting state into the Semantics Functor, we are able to use our infrastructure for implementing a verification tool for JSIL. In a nutshell, we verify that a JSIL procedure satisfies its specification by: **(1)** converting the precondition of the procedure to a symbolic state; **(2)** symbolically executing the procedure on that symbolic state; and **(3)** checking that all the obtained final symbolic states entail the postcondition of the procedure. The abstraction functor is essential for steps **(1)** and **(3)**.

The Bi-Abduction Functor (BF). To support automatic testing, we extend the JSIL symbolic semantics with a bi-abductive mechanism [2] for automatically inferring the missing resource of may-errors. Instead of creating the bi-abductive analysis from scratch, we design the Bi-Abduction Functor. It receives a state signature and generates a new state signature with a built-in mechanism for on-the-fly correction of may-errors during execution.

By plugging the symbolic state into the Bi-Abduction Functor and the resulting state into the Semantics Functor, we obtain a modular implementation of a bi-abductive analysis for JSIL with clear meta-theoretical results. By combining the Abstraction and the Bi-Abduction Functors, we enable the resulting bi-abductive analysis to make use of previously inferred specifications when analysing new procedures.

Use Cases

In Table 1, we summarise the different ways in which one can combine the proposed functors and state implementations to obtain different types of analysis for dynamic languages. For static languages, the table would have only the concrete and the symbolic state implementations, and the analyses at the instrumented level would transfer to the concrete level. We walk through the columns of the table, each corresponding to a different combination of functors that can be applied to a state implementation, briefly describing the obtained analysis for each of the cases.

Semantics Functor. If we instantiate the Semantics Functor with the concrete/instrumented/symbolic state implementation, we obtain the concrete/instrumented/symbolic interpreter. The *concrete JSIL interpreter* is useful for establishing the correctness of both the Semantic Functor and our compiler from JavaScript to JSIL, JS-2-JSIL. The symbolic JSIL interpreter is used to enable a *whole-program symbolic testing* tool in the style of Cosette [4], where users can write symbolic tests using first-order assertions and obtain correctness guarantees up to a bound. We have no direct application for the instrumented interpreter.

Semantics + Abstraction Functor. If we instantiate the Abstraction Functor with the concrete/instrumented/symbolic state implementation, and use that state to instantiate the Semantics Functor, we obtain a concrete/instrumented/symbolic JSIL interpreter with support for executable specifications. More concretely, the obtained interpreters include a mechanism for using separation logic specifications to abstractly execute procedure calls. Furthermore, the abstract symbolic interpreter can be used to enable a *verification* tool in the style of JaVerT [5], where users can write inductive predicates and separation logic specifications to describe the behaviour of their programs and obtain full functional correctness guarantees.

Semantics + Bi-abduction Functor. If we instantiate the Bi-abduction Functor with the instrumented/symbolic state implementation, and use that state to instantiate the Semantics Functor, we obtain an instrumented/symbolic JSIL interpreter with support for bi-abductive inference of missing resource. More concretely, the obtained interpreters include a mechanism for automatically correcting may-errors during execution, allowing it to proceed as if no error had occurred. Note that at the concrete level there are no may-errors; therefore, the application of the Bi-abduction Functor to the concrete state implementation is pointless. Furthermore, the bi-abductive symbolic interpreter can be used for *automatic symbolic testing*, where users can test their code for the absence of native errors without having to write the tests themselves.

Semantics + Abstraction + Bi-abduction Functor. If we instantiate the Abstraction Functor with the instrumented/symbolic state implementation, then use that state to instantiate the Bi-Abductive Functor, and then use *that* state to instantiate the Semantic Functor, we obtain a symbolic semantics with support for bi-abductive inference of missing resource and executable specifications. At the symbolic level, this combination yields *automatic compositional testing* in the style of Infer [1], where users can obtain, *without providing any annotations*, specifications that describe the behaviour of their functions up to a bound, and where specifications of previously analysed functions can be re-used for the analysis of functions that call them. The obtained specifications serve two important purposes. First, those that describe failed executions reveal potentially undesired behaviours and bugs in the analysed code. Second, those that describe successful executions can be straightforwardly re-used to construct a systematic symbolic test suite for the analysed program.

Evaluation

JaVerT 2.0 currently supports: whole-program symbolic testing, verification, and automatic compositional testing. It unifies and significantly advances our previous work on whole-program symbolic testing (Cosette, [4]) and verification (JaVerT, [5]). It improves Cosette by providing a native implementation of a symbolic execution for JSIL which is two orders of magnitude more performant than the original Cosette implementation based on Rosette [13, 14], a symbolic virtual machine for the development of solver-aided languages. It improves JaVerT by providing built-in support for automatic unfold/fold reasoning over user-defined inductive predicates and meaningful error reporting. Finally, JaVerT 2.0 is the first tool to support fully automatic compositional testing based on bi-abduction for dynamic languages.

We evaluate the three styles of analysis that our unified framework currently supports focussing on a number of simple data structure libraries. Our results demonstrate an improvement over the state-of-the-art in verification, scalability of whole-program symbolic testing, and creation of useful specifications using bi-abduction, minimising the annotation burden of the developer. Furthermore, we symbolically test the real-world Buckets.js [11] data structure library, which has over 65K downloads on npm [9]. We reproduce previously known bugs [4], but also discover a new one. The times that we obtain are competitive, which indicates that our analysis can scale to much larger codebases.

REFERENCES

[1] Cristiano Calcagno, Dino Distefano, Jeremy Dubreil, Dominik Gabi, Pieter Hooimeijer, Martino Luca, Peter O’Hearn, Irene Papakonstantinou, Jim Purbrick, and Dulma Rodriguez. 2015. Moving Fast with Software Verification. In *NASA Formal Methods*, Klaus Havelund, Gerard Holzmann, and Rajeev Joshi (Eds.). Springer International Publishing, 3–11.

[2] Cristiano Calcagno, Dino Distefano, Peter W. O’Hearn, and Hongseok Yang. 2011. Compositional Shape Analysis by Means of Bi-Abduction. *J. ACM* 58, 6 (Dec. 2011), 26:1–26:66.

[3] ECMA TC39. 2017. Test262 test suite. https://github.com/tc39/test262.

[4] José Fragoso Santos, Petar Maksimović, Théotime Grohens, Julian Dolby, and Philippa Gardner. 2018. Symbolic Execution for JavaScript. In *Proceedings of the 45th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (PPDP’18) (accepted for publication).*

[5] José Fragoso Santos, Petar Maksimović, Daiva Naudžiūnienė, Thomas Wood, and Philippa Gardner. 2018. JaVerT: JavaScript Verification Toolchain. *PACMPL* 2, POPL (2018), 50:1–50:33. https://doi.org/10.1145/3158138

[6] Philippa Gardner, Sergio Maffeis, and Gareth Smith. 2012. Towards a program logic for JavaScript. In *Proceedings of the 40th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL’13).* ACM Press, 31–44.

[7] Philippa Gardner, Gareth Smith, Mark J. Wheelhouse, and Uri Zarfaty. 2008. Local Hoare reasoning about DOM. In *Proceedings of the Twenty-Seventh ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, PODS 2008.* 261–270.

[8] Guodong Li, Esben Andreasen, and Indradeep Ghosh. 2014. SymJS: automatic symbolic testing of JavaScript web applications. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, (FSE-22), Hong Kong, China, November 16 - 22, 2014.* 449–459.

[9] npm, Inc. 2018. npm, a package manager for javascript. https://www.npmjs.com.

[10] John C. Reynolds. 2002. Separation Logic: A Logic for Shared Mutable Data Structures. In *Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science (LICS ’02).* IEEE Computer Society, Washington, DC, USA, 55–74.

[11] Mauricio Santos. 2016. Buckets-JS: A JavaScript Data Structure Library. https://github.com/mauriciosantos/Buckets-JS.

[12] Prateek Saxena, Devdatta Akhawe, Steve Hanna, Feng Mao, Stephen McCamant, and Dawn Song. 2010. A Symbolic Execution Framework for JavaScript. In *31st IEEE Symposium on Security and Privacy, S&P 2010, 16-19 May 2010, Berleley/Oakland, California, USA.* 513–528.

[13] Emina Torlak and Rastislav Bodík. 2013. Growing solver-aided languages with rosette. In *ACM Symposium on New Ideas in Programming and Reflections on Software, Onward! 2013, part of SPLASH ’13, Indianapolis, IN, USA, October 26-31, 2013.* 135–152.

[14] Emina Torlak and Rastislav Bodík. 2014. A lightweight symbolic virtual machine for solver-aided host languages. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI ’14, Edinburgh, United Kingdom - June 09 - 11, 2014.* 54.

[15] Erik Wittern, Annie T. T. Ying, Yunhui Zheng, Julian Dolby, and Jim Alain Laredo. 2017. Statically checking web API requests in JavaScript. In *Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20-28, 2017.* 244–254.