Symbolic Execution for JavaScript

José Fragoso Santos
Imperial College London, UK
jose.fragoso.santos@imperial.ac.uk

Petar Maksimović
Imperial College London, UK
Mathematical Institute SASA, Serbia
p.maksimovic@imperial.ac.uk

Théotime Grohens
ENS Paris, France
theotime.grohens@ens.fr

Julian Dolby
IBM Research, New York, USA
dolby@us.ibm.com

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

ABSTRACT

We present a framework for trustworthy symbolic execution of JavaScripts programs, whose aim is to assist developers in the testing of their code: the developer writes symbolic tests for which the framework provides concrete counter-models. We create the framework following a new, general methodology for designing compositional program analyses for dynamic languages. We prove that the underlying symbolic execution is sound and does not generate false positives. We establish additional trust by using the theory to precisely guide the implementation and by thorough testing. We apply our framework to whole-program symbolic testing of real-world JavaScript libraries and compositional debugging of separation logic specifications of JavaScript programs.

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

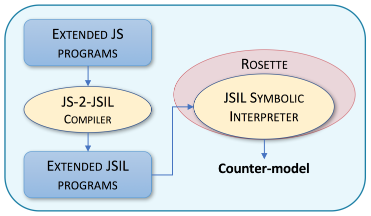
ACM Reference Format:
José Fragoso Santos, Petar Maksimović, Théotime Grohens, Julian Dolby, and Philippa Gardner. 2018. Symbolic Execution for JavaScript. In *The 20th International Symposium on Principles and Practice of Declarative Programming (PPDP '18), September 3–5, 2018, Frankfurt am Main, Germany*. ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3236950.3236956

1 INTRODUCTION

JavaScript (JS) is the most widespread dynamic language, used by 95.1% of websites [51]. Due to its dynamic, complex nature, it is a difficult target for symbolic analysis. Recent symbolic execution tools for JS [30, 42, 52] have made significant progress: they are fully automatic, aim at code in the large, and primarily focus on scalability and coverage issues. However, they do have some limitations. They only target specific bug patterns rather than general-purpose symbolic execution, and depend on whole-program analysis.

Whole-program analysis is not sufficient for JavaScript applications, which are commonly run in highly dynamic execution environments: for example, client-side programs dynamically load and execute third-party code as a matter of course. One also needs to be able to analyse incomplete code in a compositional manner. Designing compositional analyses for dynamic languages such as JavaScript, however, is non-trivial. This is because, unlike static languages such as C, C++ and Java, dynamic languages do not observe the *frame property* [39], essential for compositionality. Intuitively, the frame property means that the output of a program cannot change if the state in which it is run is extended. In JavaScript, it is possible to introduce bugs by extending the state in which a program is run. To our knowledge, there has been no previous work on compositional symbolic execution for JavaScript.

We introduce Cosette, a framework for trusted symbolic execution of JavaScript (ECMAScript 5 Strict [18]). Its aim is to provide *general-purpose* symbolic analysis and *assist* developers in the testing of their code. We present a new, general methodology for developing *compositional* analyses for languages that do not satisfy the frame property, and apply it to the design of the symbolic execution of Cosette. In contrast to existing tools, the symbolic analysis of Cosette is fully formalised in a way that guides the implementation and is also *trusted*, in that it follows the JavaScript semantics and does not produce false positive bug reports. We apply Cosette to whole-program symbolic testing of real-world JavaScript libraries and compositional debugging of separation logic specifications of JavaScript programs. We illustrate these use cases in §2.



We show the architecture of Cosette on the right. We extend JS with constructs for creating and reasoning about symbolic values. Using JS-2-JSIL [22], a trusted compiler from JS to the JSIL intermediate language, we compile the extended JS program to an extended JSIL program. The core of Cosette is a new symbolic interpreter for JSIL, written in Rosette [48, 49], a framework for creating symbolic interpreters. The JSIL symbolic interpreter either outputs a concrete counter-model for the given assertions, or guarantees correctness up to a given bound for unfolding loops and recursive predicates.

Our new, general methodology underpinning the compositional JSIL symbolic interpreter consists of three stages: **(1)** we design a JSIL instrumented semantics that exhibits the frame property by explicitly keeping track of object properties that we know are *not* *present*; **(2)** we define the JSIL symbolic semantics by lifting the JSIL instrumented semantics; and **(3)** we link the JSIL symbolic semantics to the JSIL concrete semantics by describing the frames that can be safely added to the initial state. The key innovation is to have the instrumented semantics as a proper interim stage in the design of the symbolic execution, obtaining more modular reasoning and substantially simpler proofs. We present this methodology in §3.

An essential goal for us was to establish trust in Cosette. In §3.5, we give a *bounded soundness result* for the JSIL symbolic semantics and prove that Cosette never produces false positive bug reports. These results, combined with the correctness of JS-2-JSIL and the fact that the memory models of JavaScript and JSIL are the same by design, enable us to lift the results of analyses done on compiled JSIL code back to JS (cf. §3.7, §4.4). We implement the JSIL instrumented interpreter following the instrumented semantics of §3.3 to the letter. We test the combination of JS-2-JSIL and the JSIL instrumented interpreter using Test262, the official ECMAScript test suite [19]. Out of the 10469 tests for ES5 Strict, we identify 8330 tests appropriate for our coverage, of which we pass 100%. Finally, we ensure that the JSIL symbolic interpreter obtained by the Rosette lifting of the instrumented interpreter is consistent with the symbolic semantics of §3.4 by constructing symbolic unit tests for each JSIL command, assuming the premises and asserting the conclusion of the appropriate rule of the symbolic semantics.

We apply Cosette to whole-program symbolic testing. A general developer can use Cosette for symbolic testing of their code by having symbolic inputs instead of concrete inputs and stating the constraints that the output needs to satisfy as simple, intuitive first-order assertions over these inputs. If a test fails, Cosette provides the concrete input that causes it to fail, exposing bugs in the tested code. In §5, we evaluate Cosette on two real-world JavaScript data structure libraries, where, using fewer tests, we achieve better coverage (100%) than the concrete unit test suites shipped with the libraries, and discover unexpected bugs in both libraries.

We also apply Cosette to specification-driven bug-finding. Functional correctness specifications of JS programs are highly intricate, with only a few tools (JaVerT [22] and KJS [16, 36]) supporting such expressivity. When these tools cannot prove that a given function satisfies a specification, the developer has to understand a complicated proof trace to discover the error (JaVerT), or act with essentially no feedback (KJS). In §2.2, we show how Cosette can be used for debugging separation logic (SL) specifications of JS programs in JaVerT, made possible by its compositionality. Our approach, described in §4, consists of translating the SL specifications into symbolic tests and running these tests using Cosette. If one such symbolic test fails, we can be sure that the code does not satisfy its specification. More importantly, Cosette then generates a concrete witness that invalidates the specification. This information greatly simplifies the debugging of both specifications and code.

2 USING COSETTE

We illustrate how Cosette can be used both for whole-program and compositional symbolic testing of JavaScript programs. We demonstrate how Cosette explicitly exposes the resilience of JavaScript programs to the environment, not considered by standard symbolic execution tools, but essential for compositional analysis.

Our running example is a *key-value map* implementation, given in Figure 1a. It contains four functions: `Map`, for constructing an empty map; `get`, for retrieving the value associated with a given key; `put`, for inserting/updating key-value pairs; and `validKey`, for deciding whether or not a key is valid. The library implements a *key-value map* as an object with property `_contents`, denoting the object storing the map contents. The named properties of `_contents` and their value attributes correspond to the map keys and values. The functions `get`, `put`, and `validKey` are shared between all map objects, as they are defined in `Map.prototype`, the prototype[1] of objects created using `Map` as a constructor. The `get` function returns the value associated with a given key, or `null` if the key is not in the map. To check that the given key is in the map, `get` uses the built-in function `hasOwnProperty`, which lives in `Object.prototype`, the default prototype of all objects. The `put` function updates the map if the supplied key is valid, and otherwise throws an error.

In Figure 1b, we show a general heap of key-value maps created by the library. The `map` object, whose prototype is `Map.prototype`, has the property `_contents`, pointing to the `contents` object. The `contents` object holds the key-value pairs, and its prototype is `Object.prototype`. The `Map.prototype` object holds the `get`, `put`, and `validKey` functions,[2] and its prototype is also `Object.prototype`. Finally, the `Object.prototype` holds the `hasOwnProperty` function that is called by `Map.prototype.get`.

2.1 Whole-program Symbolic Testing

Developers are used to writing unit tests—verifying that, given some concrete inputs, their code produces the expected outputs. Using Cosette, they can write unit tests with *symbolic* inputs and outputs, testing a broad range of behaviours with a single symbolic test. For example, one unit test for the `put` function consists of inserting a valid key-value pair (`k`, `v`) into a map and then verifying that it has been inserted correctly. In Cosette, this test can be written as in Figure 1c. First, we declare `k` to be a symbolic string and `v` to be an symbolic number, using Cosette's constructs for creating symbolic variables. Next, we assume that `k` is a valid key. Next, we create a new map, put the (symbolic) key-value pair (`k`, `v`) into the map and retrieve the value corresponding to `k`. Finally, we assert that the retrieved value is equal to the one we had previously put.

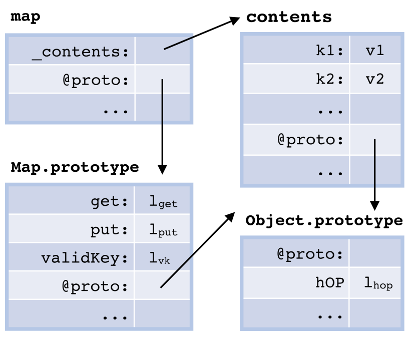
When running Cosette on this test, if the `validKey(k)` function was implemented incorrectly,[3] we will obtain the counter-model `k = "hasOwnProperty"`. To understand this error, recall the heap and the implementation of `get` from Figure 1. We can see that, if we were to put the key `"hasOwnProperty"` into the contents object of a map, then the lookup of `c.hasOwnProperty` done by `get` will not reach `Object.prototype` as intended, resolving instead to the `hasOwnProperty` property of the `contents` object (Figure 1c, below).

This example highlights how Cosette does not require specialist knowledge and can be used as a testing tool by a general JS developer. The annotations amount to the creation of symbolic variables and the writing of minimal intuitive assumptions and assertions, in contrast with the annotation burden of verification tools.

[1]In JavaScript, inheritance is modelled through *prototype chains*. On property lookup, `o.p`, we first check if the property `p` is present in the object `o`, in which case its value is returned. Otherwise, we check if `p` is present in the prototype of `o`, and so forth.
[2]In JavaScript, functions are also modelled as objects in the heap.
[3]For instance, `validKey(k)` may only require that `k` is a string, which is a reasonable implementation in the sense that it disallows JavaScript's implicit coercions.

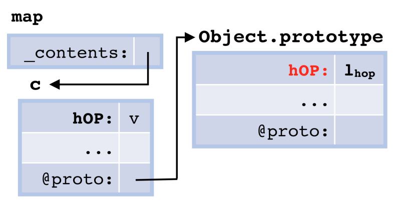
```
function Map () { this._contents = {} }

Map.prototype.get = function (k) {
  var c = this._contents;
  if (this.validKey(k)) {
    return (c.hasOwnProperty(k) ? c[k] : null)
  } else throw new Error("Invalid Key");
}

Map.prototype.put = function (k, v) {
  if (this.validKey(k)) {
    this._contents[k] = v
  } else throw new Error("Invalid Key");
}

Map.prototype.validKey = function (k) { ... }
```

(a) Library implementation



(b) General library heap

```
var k = symb_string();
var v = symb_number();
assume(validKey(k));
var m = new Map(); m.put(k, v);
var result = m.get(k);
assert(result = v);
```



(c) Simple symbolic test (above); the "hasOwnProperty" bug (below)

Figure 1: Running example: JavaScript key-value map library

2.2 Specification-driven Bug-finding

Cosette can also be used for compositional symbolic analysis of JS functions in isolation, where the user specifies the functions using pre- and post-conditions. These specifications may account for only the parts of the heap required for running the function and can involve recursive and non-recursive predicates. We always unfold recursive predicates to a bound specified by the user.

Just as symbolic tests generalise concrete tests, so specifications generalise symbolic tests. Given a JavaScript function, its specification, and the unfold depth for predicates, Cosette generates symbolic tests to verify that the function conforms to the specification up to that given depth. If this is not the case, Cosette will return a concrete counter-model that invalidates the specification. Unlike whole-program analysis tools, Cosette also tests if the given specification is compositional: that is, if it is resilient against all possible contexts in which the function can be run. It reports back to the user any found sources of non-compositionality.

Cosette supports the specification of symbolic states via simple separation logic assertions in the style of JaVerT [22]. Developers have at their disposal a number of built-in predicates that capture the fundamental concepts of JavaScript (discussed throughout the text), and can define their own predicates as well. For instance, learning from the previous symbolic test, we could define the following predicate for describing valid keys:

```
  ValidKey(k) := types(k : Str) * (k <> "hasOwnProperty"),
```

meaning that k is a valid key if it is a string that is not equal to "hasOwnProperty". From there, if we wanted to do a full functional correctness specification of the Map library, we could, guided by the heap in Figure 1b, define the following two predicates:

```
Map (m, mp, kvs) := JSObjWithProto(m, mp) *
 DProp(m, "_contents", c) * JSObject(c) * KVPairs(c, kvs) *
 NoProp(m, "get") * NoProp(m, "put") * NoProp(m, "validKey")
 * NoProps(c, FProj(kvs))
KVPairs (c, kvs) := (kvs = { }),
                    (kvs = {(k, v)} U kvs') * ValidKey(k) *
                      DProp(c, k, v) * KVPairs(c, kvs')
```

The Map predicate states that a map object is a standard JavaScript object with a given prototype mp, and that it has the property _contents, which points to a JavaScript object c. Using the KVPairs predicate, it also states that c holds the key-value pairs kvs. The KVPairs(c, kvs) predicate is recursive: kvs is either empty or contains a key-value pair (k, v), such that the key k is valid, the object c has property k with value v, and KVPairs(c, kvs') holds for the remaining pairs in kvs'. The highlighted parts of Map describe the compositionality requirements: in red, we state that map objects must not have the properties "get", "put", and "validKey"; in blue, we state that the object c has no other properties except for the map keys.[4] We also require a MapProto(mp) predicate, describing that mp is a valid map prototype: i.e., that it defines the put, get, and validKey methods. To avoid clutter, we keep its definition opaque.

We show one specification of put(k, v). We assume a map object m, with key-value pairs kvs and prototype Map.Prototype. We also assume that k is valid and not in the map. For compositionality, highlighted in blue, we state that k is not non-writable in Object.prototype.[5] In the end, we have that the key-value pair has been inserted into the map.

```
{ Map(m, mp, kvs) * MapProto(mp) *
  ValidKey(k) * (k ∉ FProj(kvs)) *
   Writable(Object.prototype, k) }
              m.put(k, v)
{     Map(m, kvs -u- (k, v)) *
          MapProto(mp) *
   Writable(Object.prototype, k) }
```

When symbolically testing the specifications of the Map library functions, the "hasOwnProperty" bug will not be triggered again, as ValidKey(k) has been corrected. However, if we were to forget the highlighted parts in the Map definition and put specification, we would encounter other issues, exposing the tension between compositionality and dynamic languages.

First, if we omitted the part of the Map predicate highlighted in red, Cosette would complain, when testing the put function, that it has no information about the property "put" of the map object m and cannot perform the lookup m.put (Fig. 2, above). This means that the library code is not resilient to frames in which a map object m has the property put. Analogous issues would arise for the "get" and "validKey" properties.

[4] NoProp(o, p) states that the object o does not have property p; NoProps(o, props) states that the object o has no properties outside of those from the set props; the FProj operator extracts the set of keys from the set of key-value pairs.
[5] In JavaScript, object properties can be non-writable, meaning that their value cannot be changed. In this specification, we assume that the property exists and is writable. There is an analogous specification for put, in which the property does not exist.

Second, if we omitted the parts highlighted in blue, when executing `this._contents[k] = v` (Fig. 1a, line 12), Cosette would complain that it has no information about the property `k` in the prototype chain of the contents object (Fig. 2). This is required as the semantics of JavaScript has to look up the value of the property `k` of the contents object before performing the assignment, to check if it is allowed (i.e., that the property can be written to). As `k` is symbolic, what this means is that the library is not resilient to frames in which there are non-writable properties in `Object.prototype`. A similar issue arises for the `Map` constructor (Fig. 1a, line 1), which requires the property `"_contents"` not to be non-writable in the prototype chains of map objects.

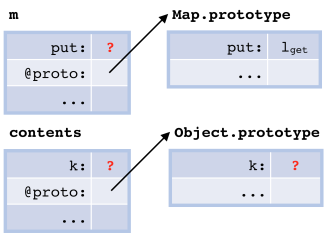


Figure 2. Compositionality

What this example illustrates is that, in order to be compositional, specifications of programs written in dynamic languages have to explicitly state which parts of the heap must not be present. Cosette is able to detect and report compositionality-related issues, such as those presented above, which are highly likely to remain unnoticed in whole-program analysis, as there we always have complete information about the entire contents of the heap.

3 SYMBOLIC EXECUTION FOR JSIL

We present a new methodology for designing compositional program analyses for dynamic languages, illustrating our approach by defining a new compositional symbolic execution for JSIL, an intermediate representation for JS analysis [22]. We develop a new, abstract semantics for JSIL, in the spirit of [50], which we instantiate to obtain the concrete, instrumented, and symbolic semantics. This abstract semantics is the bedrock for both the formal development *and* the implementation of the analysis. This approach streamlines the formalism, avoiding redundancy, makes the choices in the design of the instrumented and symbolic semantics explicit, and leads to modular implementations, avoiding code duplication.

3.1 JSIL Syntax and Abstract Semantics

JSIL is a simple goto language with top-level procedures and commands operating on object heaps. It natively supports the main dynamic features of JavaScript: extensible objects; dynamic property access; and dynamic procedure calls. Its syntax is shown below.

Syntax of the JSIL Language

$$v \in \mathcal{V} ::= n \mid b \mid s \mid \mathsf{undefined} \mid \mathsf{null} \mid \mathsf{empty} \mid l \mid \tau \mid f \mid [v_i|_{i=0}^{n}] \mid \{ v_i|_{i=0}^{n} \}$$
$$e \in \mathcal{E}xp ::= v \mid x \mid \hat{x} \mid \ominus e \mid e \oplus e$$
$$bc \in \mathcal{B}cmd ::= \mathsf{skip} \mid x := e \mid x := \mathsf{new}() \mid x := [e, e] \mid [e, e] := e \mid \mathsf{delete}(e, e) \mid x := \mathsf{hasProp}(e, e) \mid x := \mathsf{getProps}(e)$$
$$c \in \mathcal{C}md ::= bc \mid \mathsf{goto}\ i \mid \mathsf{goto}\ [e]\ i,\ j \mid x := e(e_i|_{i=0}^{n})\ \mathsf{with}\ j \mid \mathsf{assume}(e) \mid \mathsf{assert}(e)$$
$$proc \in \mathsf{Proc} ::= \mathsf{proc}\ f(\overline{x})\{\overline{c}\} \qquad i, j \in \mathcal{I}nd \triangleq \mathbb{N} \cup \{i_{\mathrm{nm}}, i_{\mathrm{er}}\}$$

JSIL *values*, $v \in \mathcal{V}$, include numbers, booleans, strings, the special values undefined, null, and empty, object locations l, types τ, procedure identifiers f, and lists and sets of values. JSIL *expressions*, $e \in \mathcal{E}xp$, include JSIL values, JSIL program variables x, various unary and binary operators, and symbolic variables $\hat{x}$, introduced in this paper. Symbolic variables range over symbolic strings, numbers, booleans, and locations, denoted respectively by $\hat{s}$, $\hat{n}$, $\hat{b}$, and $\hat{l}$.

JSIL *basic commands* enable the manipulation of extensible objects and do not affect control flow. They include skip, variable assignment, object creation, property access, property assignment, property deletion, membership check, and property collection. JSIL *commands* include JSIL basic commands and control flow commands: conditional and unconditional gotos, dynamic procedure calls, and two new commands for symbolic reasoning, assume and assert. The goto commands are intuitive: goto i jumps to the i-th command of the active procedure, and goto $[e]$ i, j jumps to the i-th command if e evaluates to true, and to the j-th otherwise. The dynamic procedure call $x := e(\overline{e})$ with j evaluates e and $\overline{e}$ to obtain the procedure identifier and arguments, executes the appropriate procedure with these arguments, and, in the end, assigns its return value to x. If the procedure raises an error, control is transferred to the j-th command; otherwise, it follows to the next command.

A JSIL procedure is of the form proc $f(\overline{x})\{\overline{c}\}$, where f is its identifier, $\overline{x}$ are its formal parameters, and its body $\overline{c}$ is a sequence of JSIL commands. Procedures return via two dedicated indexes, i_{nm} and i_{er}, using two dedicated variables, `ret` and `err`. If a procedure reaches the i_{nm} index, it returns normally with the return value denoted by `ret`; if it reaches i_{er}, it returns an error, with the error value denoted by `err`. A JSIL program $p \in \mathcal{P}$ is a set of top-level procedures, and its entry point is the special procedure main.

Abstract semantics. We design the abstract semantics to make the analysis, the proofs, and the implementations as modular as possible. At its core are the GetCell and GetDomain functions, which precisely pinpoint the way in which JSIL interacts with the heap, factoring out the common behaviour of JSIL basic commands. It also provides standard constructs for reasoning about symbolic values, whose concrete and instrumented semantics are trivial.

The abstract semantics is parametric on abstract states $\Sigma \in \mathbb{S}$ and abstract values $\mathsf{v}, \mathsf{p}, \mathsf{l} \in \mathbb{V}$. Abstract states are assumed to have a store $\mathsf{P} : \mathcal{X} \rightharpoonup \mathbb{V}$, mapping program variables $x \in \mathcal{X}$ to abstract values. Stores are accessible via a store selector, $\Sigma.\mathsf{sto}$. We use the special symbol $\varnothing$ (read: none) to denote the absence of a property in an object, write $\mathbb{V}_{\varnothing}$ for the set $\mathbb{V} \cup \{\varnothing\}$ and range over it using $\underline{\mathsf{v}}$. Abstract states expose the following functions and relations:

JSIL Expression Evaluation, $\mathcal{E}v(\mathsf{P}, e)$, which evaluates to the value of the JSIL expression e under store P.

Store Update, $\mathcal{SU}(\Sigma, x, \mathsf{v})$, which denotes the state obtained from Σ by updating the value of x to v in $\Sigma.\mathsf{sto}$.

Heap Allocation, $\mathcal{A}lloc(\Sigma)$, which evaluates to a pair consisting of a value denoting a fresh location and the new state that keeps track of that allocation.

Heap Update, $\mathcal{HU}(\Sigma, \mathsf{l}, \mathsf{p}, \underline{\mathsf{v}})$, which denotes the state obtained from Σ by updating the value of property p of the object at location l to $\underline{\mathsf{v}}$ or creating that property, if it does not exist.

GetCell, $\mathcal{GC}(\Sigma, e_1, e_2) \rightsquigarrow \Sigma', (\mathsf{l}, \mathsf{p}, \underline{\mathsf{v}})$, which retrieves the value associated with a given property of a given object. Formally, if $\mathcal{GC}(\Sigma, e_1, e_2) \rightsquigarrow \Sigma', (\mathsf{l}, \mathsf{p}, \underline{\mathsf{v}})$ holds, then: l denotes the location resulting from the evaluation of e_1, p denotes the property name

Skip
$$\langle \Sigma, \mathsf{skip}\,\rangle \rightsquigarrow \Sigma$$

Property Collection
$$\frac{\mathsf{v} = \mathcal{GD}(\Sigma, e) \quad \Sigma' = \mathcal{SU}(\Sigma, x, \mathsf{v})}{\langle \Sigma, x := \mathsf{getProps}\,(e)\rangle \rightsquigarrow \Sigma'}$$

Assignment
$$\frac{\mathsf{P} = \Sigma.\mathsf{sto} \quad \mathsf{v} = \mathcal{E}v(\mathsf{P}, e)}{\langle \Sigma, x := e\rangle \rightsquigarrow \mathcal{SU}(\Sigma, x, \mathsf{v})}$$

Property Access
$$\frac{\mathcal{GC}(\Sigma, e_1, e_2) \rightsquigarrow \Sigma', (-, -, \mathsf{v})}{\langle \Sigma, x := [e_1, e_2]\rangle \rightsquigarrow \mathcal{SU}(\Sigma', x, \mathsf{v})}$$

Property Assignment
$$\frac{\mathcal{GC}(\Sigma, e_1, e_2) \rightsquigarrow \Sigma', (\mathtt{l}, \mathsf{p}, -) \quad \mathsf{P} = \Sigma.\mathsf{sto} \quad \mathsf{v} = \mathcal{E}v(\mathsf{P}, e_3)}{\langle \Sigma, [e_1, e_2] := e_3\rangle \rightsquigarrow \mathcal{HU}(\Sigma', \mathtt{l}, \mathsf{p}, \mathsf{v})}$$

Object Creation
$$\frac{(l, \Sigma') = \mathcal{A}lloc(\Sigma) \quad \mathcal{GC}(\Sigma', l, @proto) \rightsquigarrow \Sigma'', - \quad \Sigma''' = \mathcal{HU}(\Sigma'', l, @proto, \mathsf{null})}{\langle \Sigma, x := \mathsf{new}\,()\rangle \rightsquigarrow \mathcal{SU}(\Sigma''', x, l)}$$

Property Deletion
$$\frac{\mathcal{GC}(\Sigma, e_1, e_2) \rightsquigarrow \Sigma', (\mathtt{l}, \mathsf{p}, \mathsf{v})}{\langle \Sigma, \mathsf{delete}\,(e_1, e_2)\rangle \rightsquigarrow \mathcal{HU}(\Sigma', \mathtt{l}, \mathsf{p}, \varnothing)}$$

Member Check
$$\frac{\mathcal{GC}(\Sigma, e_1, e_2) \rightsquigarrow \Sigma', (-, -, \mathsf{v}) \quad \Sigma'' = \mathcal{SU}(\Sigma', x, \underline{\mathsf{v}} \neq \varnothing)}{\langle \Sigma, x := \mathsf{hasProp}\,(e_1, e_2)\rangle \rightsquigarrow \Sigma''}$$

Figure 3. Abs. sem. of basic commands: $\langle \Sigma, bc\rangle \rightsquigarrow \Sigma'$

resulting from the evaluation of e_2, $\underline{\mathsf{v}}$ denotes the value of the property p of the object at location $\mathtt{l}$, and Σ' denotes a potential re-arrangement of Σ after property inspection (discussed in §3.3).

GetDomain, $\mathcal{GD}(\Sigma, e)$, which denotes the set of property names associated with the object at location resulting from the evaluation of e in Σ. It is used in the Property Collection rule.

Assumption, $\mathcal{A}sm(\Sigma, e)$, which denotes the state obtained from Σ by stating that e is assumed to evaluate to true.

SAT Check, $\mathcal{S}at(\Sigma, e)$, which evaluates to true if the JSIL expression e is satisfiable in the state Σ, and to false otherwise.

Transitions for basic commands have the form $\langle \Sigma, bc\rangle \rightsquigarrow \Sigma'$, meaning that the execution of the basic command bc in the state Σ results in the state Σ' (Fig. 3). To describe transitions of commands, we introduce *execution modes*, μ, and *call stacks*, cs. JSIL has two execution modes: $\top$, meaning the execution can proceed; and $\bot$, meaning that the execution must stop due to an *assert failure.* Call stacks are lists of tuples of the form (f, P, x, i, j), where: f is the identifier of the executing procedure; P is the store of the procedure that called f; x is the variable that will hold the return value of f; and i (j) is the index to which the control jumps when f returns normally (with an error). The *initial call stack*, cs_{main}, is defined as $[(\mathsf{main}, \emptyset, out, 0, 0)]$, where *out* holds the output of the entire program. Transitions for commands have the form $p : \langle \Sigma, \mathsf{cs}, i\rangle^{\mu} \rightsquigarrow \langle \Sigma', \mathsf{cs}', j\rangle^{\mu'}$, meaning that, given a program p, state Σ and execution mode μ, the evaluation of the i-th command of the first procedure of cs generates the state Σ', call stack cs', and the next command to be evaluated is the j-th command of the first procedure of cs', in execution mode μ' (Fig. 4). For clarity, we keep the program implicit and write $\mathsf{cmd}(\mathsf{cs}, i)$ to denote the i-th command of the first procedure of cs.

Notation. In the following, we denote a function with an empty domain by $\emptyset$, and for a function $f : A \rightharpoonup B$, we denote its domain extension/update by $f[a \mapsto b]$ and the union of two functions with disjoint compatible domains by $f_1 \uplus f_2$.

3.2 JSIL Concrete Semantics

A JSIL concrete state σ is a pair (h, ρ), consisting of a heap and a store. A heap, $h \in \mathcal{H}$, is a partial function mapping pairs of object

Basic Command
$$\frac{\mathsf{cmd}(\mathsf{cs}, i) = bc \quad \langle \Sigma, bc\rangle \rightsquigarrow \Sigma'}{\langle \Sigma, \mathsf{cs}, i\rangle^{\top} \rightsquigarrow \langle \Sigma', \mathsf{cs}, i{+}1\rangle^{\top}}$$

Goto
$$\frac{\mathsf{cmd}(\mathsf{cs}, i) = \mathsf{goto}\ j}{\langle \Sigma, \mathsf{cs}, i\rangle^{\top} \rightsquigarrow \langle \Sigma, \mathsf{cs}, j\rangle^{\top}}$$

Cond. Goto - True
$$\frac{\mathsf{cmd}(\mathsf{cs}, i) = \mathsf{goto}\ [e]\ j,\ k \quad \Sigma' = \mathcal{A}sm(\Sigma, e)}{\langle \Sigma, \mathsf{cs}, i\rangle^{\top} \rightsquigarrow \langle \Sigma', \mathsf{cs}, j\rangle^{\top}}$$

Cond. Goto - False
$$\frac{\mathsf{cmd}(\mathsf{cs}, i) = \mathsf{goto}\ [e]\ j,\ k \quad \Sigma' = \mathcal{A}sm(\Sigma, \neg e)}{\langle \Sigma, \mathsf{cs}, i\rangle^{\top} \rightsquigarrow \langle \Sigma', \mathsf{cs}, k\rangle^{\top}}$$

Normal Return
$$\frac{\mathsf{cs} = (-, \mathsf{P}', x, i, -) :: \mathsf{cs}' \quad \mathsf{v} = (\Sigma.\mathsf{sto})(\mathsf{ret}) \quad \Sigma' = \mathcal{SU}(\Sigma[\mathsf{sto} \mapsto \mathsf{P}'], x, \mathsf{v})}{\langle \Sigma, \mathsf{cs}, i_{\mathsf{nm}}\rangle^{\top} \rightsquigarrow \langle \Sigma', \mathsf{cs}', i\rangle^{\top}}$$

Error Return
$$\frac{\mathsf{cs} = (-, \mathsf{P}', x, -, j) :: \mathsf{cs}' \quad \mathsf{v} = (\Sigma.\mathsf{sto})(\mathsf{err}) \quad \Sigma' = \mathcal{SU}(\Sigma[\mathsf{sto} \mapsto \mathsf{P}'], x, \mathsf{v})}{\langle \Sigma, \mathsf{cs}, i_{\mathsf{er}}\rangle^{\top} \rightsquigarrow \langle \Sigma', \mathsf{cs}', j\rangle^{\top}}$$

Procedure Call
$$\frac{\begin{array}{c}\mathsf{cmd}(\mathsf{cs}, i) = x := e(e_i\ |_{i=0}^{n})\ \mathsf{with}\ j \quad \mathsf{P} = \Sigma.\mathsf{sto} \quad \mathcal{E}v(\mathsf{P}, e) = f' \\ \mathsf{args}(f') = [x_1, \dots, x_m] \quad \mathsf{v}_i = \mathcal{E}v(\mathsf{P}, e_i)\ |_{i=0}^{n} \\ \mathsf{v}_i = \mathsf{undefined}\ |_{i=n+1}^{m} \quad \mathsf{P}' = [x_i \mapsto \mathsf{v}_i\ |_{i=0}^{m}]\end{array}}{\langle \Sigma, \mathsf{cs}, i\rangle^{\top} \rightsquigarrow \langle \Sigma[\mathsf{sto} \mapsto \mathsf{P}'], (f', \mathsf{P}, x, i{+}1, j) :: \mathsf{cs}, 0\rangle^{\top}}$$

Assume
$$\frac{\mathsf{cmd}(i) = \mathsf{assume}\,(e) \quad \Sigma' = \mathcal{A}sm(\Sigma, e)}{\langle \Sigma, \mathsf{cs}, i\rangle^{\top} \rightsquigarrow \langle \Sigma', \mathsf{cs}, i{+}1\rangle^{\top}}$$

Assert
$$\frac{\mathsf{cmd}(\mathsf{cs}, i) = \mathsf{assert}\,(e) \quad \mu = \text{if } \mathcal{S}at(\Sigma, \neg e) \text{ then } \bot \text{ else } \top}{\langle \Sigma, \mathsf{cs}, i\rangle^{\top} \rightsquigarrow \langle \Sigma, \mathsf{cs}, i{+}1\rangle^{\top}}$$

Figure 4. Abs. sem. of commands: $\langle \Sigma, \mathsf{cs}, i\rangle^{\mu} \rightsquigarrow \langle \Sigma', \mathsf{cs}', j\rangle^{\mu'}$

locations and property names (strings, $\mathcal{S}$) to JSIL values. A store, $\rho \in \mathcal{S}to$, maps program variables to JSIL values. In the following, we denote a heap cell by $(l, s) \mapsto v$, meaning that $h(l, s) = v$.

We instantiate the abstract semantics for the concrete case by providing appropriate definitions for the required abstractions. We write $\langle \sigma, bc\rangle \rightsquigarrow_{\mathsf{c}} \sigma'$ for the concrete semantic judgement for JSIL basic commands and $\langle \sigma, cs, i\rangle^{\mu} \rightsquigarrow_{\mathsf{c}} \langle \sigma', cs', j\rangle^{\mu'}$ for JSIL commands.

The instantiation of the JSIL abstract semantics to the concrete case is straightforward: if an object $\mathtt{l}$ has property p, GetCell returns the associated value, and $\varnothing$ otherwise; GetDomain returns the set of properties of a given object; heap allocation returns a fresh object location without changing the state; the rule for positive heap update is standard; and the negative heap update removes the given property of a given object from the heap. Note that the positive and negative heap update rules are not applicable at the same time, as v cannot be equal to $\varnothing$.

Selected Concrete Semantics Rules

GetCell - Found
$$\frac{l = \mathcal{E}v_{\mathsf{c}}(\rho, e_1) \quad p = \mathcal{E}v_{\mathsf{c}}(\rho, e_2) \quad h = - \uplus (l, p) \mapsto v \quad r = (l, p, v)}{\mathcal{GC}((h, \rho), e_1, e_2) \rightsquigarrow_{\mathsf{c}} (h, \rho), r}$$

GetCell - Not Found
$$\frac{l = \mathcal{E}v_{\mathsf{c}}(\rho, e_1) \quad p = \mathcal{E}v_{\mathsf{c}}(\rho, e_2) \quad (l, p) \notin \mathsf{dom}(h) \quad r = (l, p, \varnothing)}{\mathcal{GC}((h, \rho), e_1, e_2) \rightsquigarrow_{\mathsf{c}} (h, \rho), r}$$

GetDomain
$$\frac{l = \mathcal{E}v_{\mathsf{c}}(\rho, e) \quad (l, -) \notin \mathsf{dom}(h') \quad h = h' \uplus \left((l, p_i) \mapsto v_i\right)|_{i=0}^{m}}{\mathcal{GD}_{\mathsf{c}}((h, \rho), e) \triangleq \{p_1, \dots, p_m\}}$$

Heap Alloc.
$$\frac{\sigma = (h, -) \quad (l, -) \notin \mathsf{dom}(h)}{\mathcal{A}lloc_{c}(\sigma) \triangleq (l, \sigma)}$$

Positive Heap Update
$$\frac{h' = h[(l, p) \mapsto v]}{\mathcal{HU}_{\mathsf{c}}((h, \rho), l, p, v) \triangleq (h', \rho)}$$

Negative Heap Update
$$\frac{h = h' \uplus (l, p) \mapsto -}{\mathcal{HU}_{\mathsf{c}}((h, \rho), l, p, \varnothing) \triangleq (h', \rho)}$$

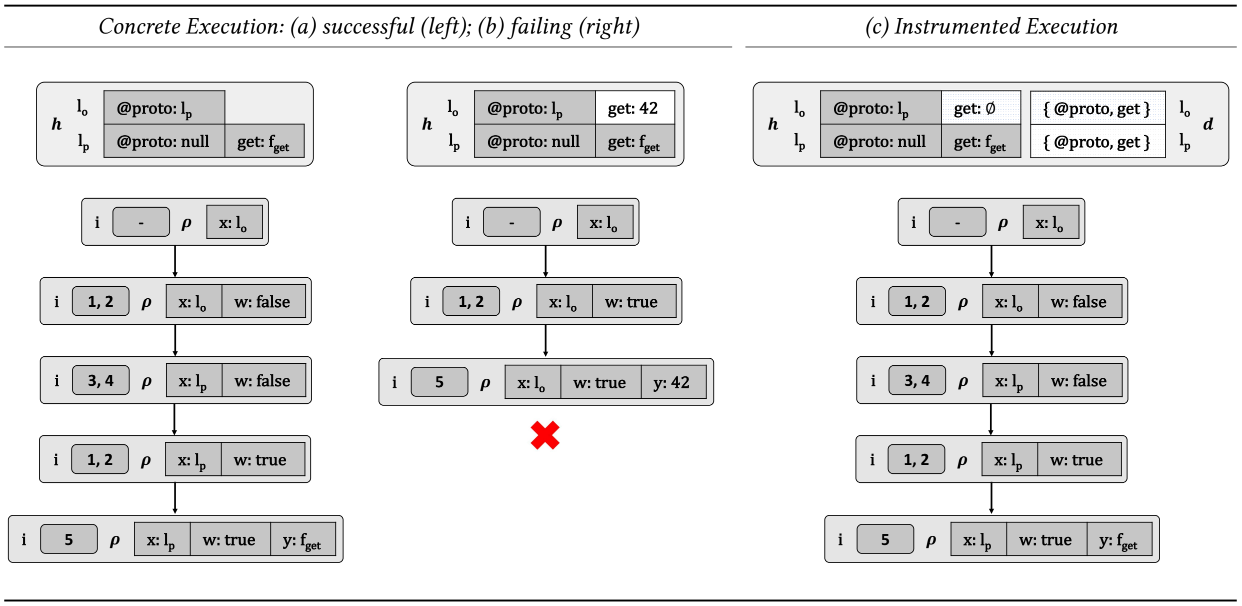


Table 1: Concrete vs. instrumented execution

Example: Frame. The JSIL concrete semantics does not satisfy the frame property. We illustrate this with the program on the right, which looks for the property *"get"* in the prototype chain of the object bound to x, reads the value of that property, and calls the procedure whose identifier is bound to that value. The program terminates successfully when run from the state described in Table 1 (a). However, if we extend the initial state with the frame $(l_o, \text{"get"}) \mapsto 42$, as shown in Table 1 (b), the procedure call fails, as 42 is not a procedure identifier. Note that, in Table 1, we factor the heap out as it is not modified by the code.

```
1   w := hasProp (x, "get")
2   goto [w] 5, 3
3   x := [x, "@proto"]
4   goto [x = null] i_er, 1
5   y := [x, "get"]
6   z := y() with i_er
```

3.3 JSIL Instrumented Semantics

Compositional analyses must reason about programs given partial state information. This is particularly challenging for languages that do not observe the frame property. We approach this problem by first designing an instrumented version of the language semantics that *does* observe the frame property. To achieve this, the JSIL instrumented semantics keeps track of both the present *and the absent* properties of a given object, using ideas from [22, 23].

An instrumented state $\underline{\sigma}$ is a triple $(\underline{h}, d, \rho)$ consisting of an instrumented heap, a domain table, and a store. An instrumented heap, $\underline{h} \in \mathcal{H}_{\emptyset} : \mathcal{L} \rightharpoonup \mathcal{S} \rightharpoonup \mathcal{V}_{\varnothing}$, can map object properties to $\varnothing$, explicitly declaring their absence. We refer to those cells as $\varnothing$-*cells* (read: none-cells). A domain table, $d : \mathcal{L} \rightharpoonup \mathcal{V}$, maps object locations to sets of properties that the corresponding objects may have, whereas all other properties are *known to be* absent. If $d(l)$ is defined and if $p \notin d(l)$, then the object at l *does not have* the property p, and that that property *cannot* be framed on. In contrast, any properties in $d(l)$ that are not in the heap can be safely framed on. Also, if we have that $(l, p) \in \mathsf{dom}(\underline{h})$, then it holds that $p \in d(l)$. On the other hand, if $d(l)$ is not defined, then all properties of the object at l that are not in the heap *can be* safely framed on.

We instantiate the abstract semantics to the instrumented case. We write $\langle \underline{\sigma}, bc \rangle \rightsquigarrow_{\mathsf{i}} \underline{\sigma}'$ for the instrumented semantic judgement for basic commands, and $\langle \underline{\sigma}, cs, i \rangle^{\mu} \rightsquigarrow_{\mathsf{i}} \langle \underline{\sigma}', cs', j \rangle^{\mu'}$ for commands. The omitted rules coincide with the concrete case.

Selected Instrumented Semantics Rules

Heap Update
$$\frac{\underline{\sigma} = (\underline{h}, d, \rho) \qquad \underline{h}' = \underline{h}[(l, p) \mapsto \underline{v}]}{\mathcal{HU}_{\mathsf{i}}(\underline{\sigma}, (l, p), \underline{v}) \triangleq (\underline{h}', d, \rho)}$$

GetCell - Found
$$\frac{\underline{\sigma} = (\underline{h}, -, \rho) \quad l = \mathcal{E}v_{\mathsf{c}}(\rho, e_1) \qquad p = \mathcal{E}v_{\mathsf{c}}(\rho, e_2) \quad \underline{h} = - \uplus (l, p) \mapsto \underline{v}}{\mathcal{GC}(\underline{\sigma}, e_1, e_2) \rightsquigarrow_{\mathsf{i}} \underline{\sigma}, (l, p, \underline{v})}$$

GetCell - Not Found
$$\frac{l = \mathcal{E}v_{\mathsf{c}}(\rho, e_1) \quad p = \mathcal{E}v_{\mathsf{c}}(\rho, e_2) \quad p \notin d(l) \qquad \underline{h}' = \underline{h} \uplus (l, p) \mapsto \varnothing \quad d' = d[l \mapsto d(l) \cup \{ p \}]}{\mathcal{GC}((\underline{h}, d, \rho), e_1, e_2) \rightsquigarrow_{\mathsf{i}} (\underline{h}', d', \rho), (l, p, \varnothing)}$$

GetDomain
$$\frac{l = \mathcal{E}v_{\mathsf{c}}(\rho, e) \quad \underline{h} = \underline{h}' \uplus \big((l, p_i) \mapsto v_i\big)\big|_{i=0}^{m} \quad (l, -) \notin \mathsf{dom}(\underline{h}') \qquad \{ p_1, \ldots, p_m \} = d(l) \quad \forall_{0 \le i \le n}\, v_i \neq \varnothing \quad \forall_{n < i \le m}\, v_i = \varnothing}{\mathcal{GD}_{\mathsf{i}}((\underline{h}, d, \rho), e) \triangleq \{ p_1, \ldots, p_n \}}$$

As $\underline{v}$ ranges over $\mathcal{V}_{\varnothing}$, the Heap Update rule may update the value of an object property to $\varnothing$. In the concrete case, the corresponding rule would simply remove that property from the heap. As an instrumented heap can contain $\varnothing$-cells, the GetDomain rule has to filter out properties mapped to $\varnothing$. Furthermore, in order to apply this rule, one has to have full information about the object: all properties from $d(l)$ must be present in the heap. Another difference between the two semantics is apparent in the rule GetCell - Not Found: while $\mathcal{GC}(\underline{\sigma}, e_1, e_2) \rightsquigarrow_{\mathsf{i}} -, (-, -, \varnothing)$ means that one is certain that the object denoted by e_1 does not have the property denoted by e_2, $\mathcal{GC}(\sigma, e_1, e_2) \rightsquigarrow_{\mathsf{c}} -, (-, -, \varnothing)$ only means that this property does not exist in the current heap.

Example: Frame revisited. In the rightmost column of Table 1, we give the instrumented execution of the frame example when the object at location l_o is guaranteed not to have the property *"get"*.

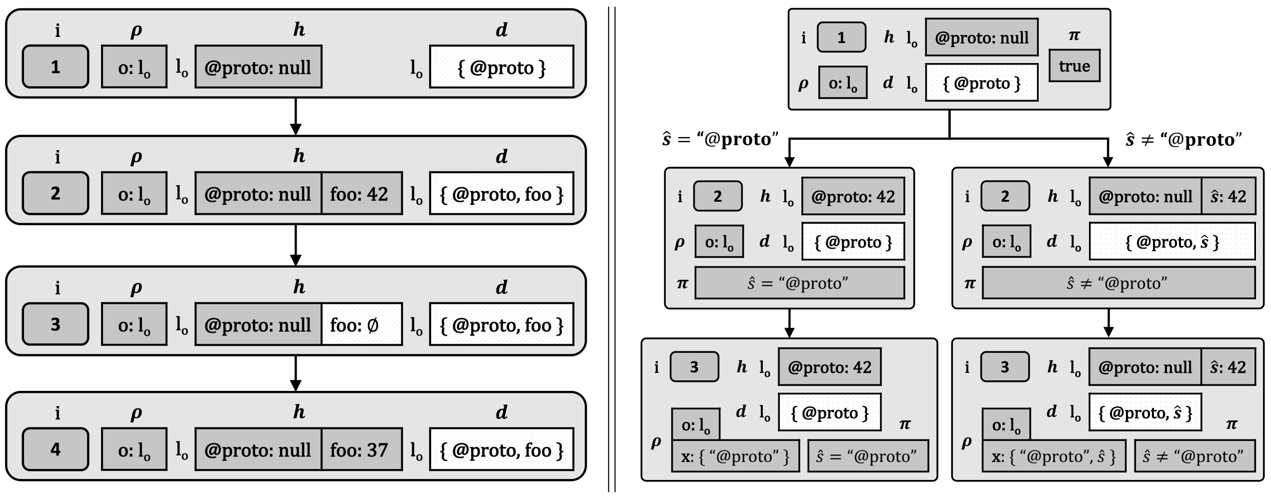


Figure 5: Execution traces: Instrumented GetCell (left); Symbolic branching (right)

Then, the instrumented heap cannot be extended with the frame $(l_o, \text{"get"}) \mapsto 5$ as it overlaps with $(l_o, \text{"get"}) \mapsto \varnothing$, meaning that the illustrated frame bug cannot be replicated in the instrumented setting. Also, if we remove $(l_o, \text{"get"}) \mapsto \varnothing$ from the initial heap, the instrumented execution will get stuck in the execution of line 0.

```
o := new ()
[o, "foo"] := 42
delete (o, "foo")
[o, "foo"] := 37
```

Example: Instrumented GetCell. To better understand the interaction between GetCell, the heap, and the domain table, consider the program on the right and its execution trace in Figure 5 (left). When executing the first property assignment, following the abstract Property Assignment rule, we must call $\mathcal{GC}(\text{o}, \text{"foo"})$ to obtain the current value of the property. As the property is not in the heap, we are in the Not Found case of GetCell. The domain is extended with *"foo"*, and the heap is extended with the none-cell $(\text{l}_\text{o}, \text{"foo"}) \mapsto \varnothing$, whose value is then updated to 42 by the heap update. After the deletion, its value is reset to $\varnothing$. The following property assignment will again call $\mathcal{GC}(\text{o}, \text{"foo"})$, but this time, we will be in the Found case of GetCell, as the property is in the heap. In a nutshell, GetCell ensures that all inspected properties, both present and absent, are represented in the heap.

Formal Guarantees. Theorem 3.1 states that the JSIL instrumented semantics observes the frame property. Lemma 3.2 relates the instrumented semantics with the concrete semantics via an erasure function $\mathcal{I}^{\text{i}\to\text{c}}$. Informally, $\mathcal{I}^{\text{i}\to\text{c}}(\underline{\sigma})$ denotes the concrete state obtained by erasing negative information in $\underline{\sigma}$ (i.e., $\varnothing$-cells and domain information). Lemma 3.2 states that if a given execution goes through in the instrumented semantics, then its concretisation (given by $\mathcal{I}^{\text{i}\to\text{c}}$) goes through in the concrete semantics. For $\underline{\sigma} = (\underline{h}, d, \rho)$, we write $\underline{\sigma} \uplus \underline{h}'$ to denote $(\underline{h} \uplus \underline{h}', d, \rho)$ defined only in the case when when $\underline{h}'$ does not redefine the properties stated not to exist by the domain table d.

Theorem 3.1 (Frame Property - Instrumented Semantics).

$$\langle \underline{\sigma}, cs, i\rangle^{\mu} \rightsquigarrow_i^* \langle \underline{\sigma}', cs', j\rangle^{\mu'} \Rightarrow \langle \underline{\sigma} \uplus \underline{h}_f, cs, i\rangle^{\mu} \rightsquigarrow_i^* \langle \underline{\sigma}' \uplus \underline{h}_f, cs', j\rangle^{\mu'}$$

Lemma 3.2 (Transparency for Instrumentation).

$$\langle \underline{\sigma}, cs, i\rangle^{\mu} \rightsquigarrow_i^* \langle \underline{\sigma}', cs', j\rangle^{\mu'} \implies \langle \mathcal{I}^{i\to c}(\underline{\sigma}), cs, i\rangle^{\mu} \rightsquigarrow_c^* \langle \mathcal{I}^{i\to c}(\underline{\sigma}'), cs', j\rangle^{\mu'}$$

3.4 JSIL Symbolic Semantics

We obtain the JSIL symbolic semantics by lifting the JSIL instrumented semantics to the symbolic level, following standard approaches [49]. This lifting, however, is easier for us to achieve because we only need to instantiate the abstract semantics, rather than re-examine every command of the language.

We instantiate the abstract values to *symbolic expressions*, $\hat{e} \in \widehat{\mathcal{E}}$, defined as follows: $\hat{e} \triangleq v \mid \hat{x} \mid \ominus\, \hat{e} \mid \hat{e} \oplus \hat{e}$. For clarity, we use $\hat{p}$ for symbolic expressions denoting property names, and $\hat{v}$ for symbolic expressions denoting arbitrary values.

A *symbolic state*, $\hat{\sigma} = (\hat{h}, \hat{d}, \hat{\rho}, \pi)$, consists of a symbolic heap $\hat{h}$, a symbolic domain $\hat{d}$, a symbolic store $\hat{\rho}$, and a path condition π. The symbolic heap, symbolic domain, and symbolic store are obtained from their instrumented counterparts of §3.3, by replacing concrete values with symbolic expressions: for example, a symbolic heap, $\hat{h} \in \hat{\mathcal{H}} : \hat{\mathcal{L}} \rightharpoonup \widehat{\mathcal{E}} \rightharpoonup \widehat{\mathcal{E}}_{\varnothing}$, maps pairs of object locations (both symbolic and concrete) and symbolic expressions to symbolic expressions extended with $\varnothing$. A *path condition* [5] is a first-order quantifier-free formula which accumulates constraints on the symbolic variables that direct the execution to the current symbolic state. For clarity of presentation, we conflate JSIL logical values and logical values, as well as the JSIL logical operators and boolean logical operators.

Selected Symbolic Semantics Rules

GetCell - Found

$$\frac{\mathcal{E}v(\hat{\rho}, e_1) = \hat{l} \quad \mathcal{E}v(\hat{\rho}, e_2) = \hat{p} \qquad \hat{h} = - \uplus (\hat{l}, \hat{p}') \mapsto \hat{v} \quad \hat{\sigma}' = (\hat{h}, \hat{d}, \hat{\rho}, \pi \wedge (\hat{p} = \hat{p}'))}{\mathcal{GC}((\hat{h}, \hat{d}, \hat{\rho}, \pi), e_1, e_2) \rightsquigarrow_{\text{s}} \hat{\sigma}', (\hat{l}, \hat{p}, \hat{v})}$$

GetCell - Not Found

$$\frac{\mathcal{E}v_{\text{s}}(\hat{\rho}, e_1) = \hat{l} \quad \mathcal{E}v_{\text{s}}(\hat{\rho}, e_2) = \hat{p} \qquad \hat{h}' = \hat{h} \uplus ((\hat{l}, \hat{p}) \mapsto \varnothing) \quad \hat{d}' = \hat{d}[\hat{l} \mapsto \hat{d}(\hat{l}) \cup \{\hat{p}\}]}{\mathcal{GC}((\hat{h}, \hat{d}, \hat{\rho}, \pi), e_1, e_2) \rightsquigarrow_{\text{s}} (\hat{h}', \hat{d}', \hat{\rho}, \pi \wedge \hat{p} \notin \hat{d}(\hat{l})), (\hat{l}, \hat{p}, \varnothing)}$$

Assume

$$\frac{\hat{b} = \mathcal{E}v_{\text{s}}(\hat{\sigma}.\text{sto}, e)}{\mathcal{A}sm_{\text{s}}(\hat{\sigma}, e) \triangleq \hat{\sigma} \wedge \hat{b}}$$

Check Sat - True

$$\frac{\hat{b} = \mathcal{E}v_{\text{s}}(\hat{\sigma}.\text{sto}, e) \qquad (\hat{\sigma}.\text{pc} \wedge \hat{b})\ \text{SAT}}{\mathcal{S}at_{\text{s}}(\hat{\sigma}, e) \triangleq \text{true}}$$

Check Sat - False

$$\frac{\hat{b} = \mathcal{E}v_{\text{s}}(\hat{\sigma}.\text{sto}, e) \qquad (\hat{\sigma}.\text{pc} \wedge \hat{b})\ \text{UNSAT}}{\mathcal{S}at_{\text{s}}(\hat{\sigma}, e) \triangleq \text{false}}$$

We instantiate the abstract semantics for the symbolic case. We write $\langle \hat{\sigma}, bc \rangle \rightsquigarrow_{\mathsf{s}} \hat{\sigma}'$ for the symbolic semantic judgement for basic commands and $\langle \hat{\sigma}, \hat{cs}, i \rangle^{\mu} \rightsquigarrow_{\mathsf{s}} \langle \hat{\sigma}', \hat{cs}', j \rangle^{\mu'}$ for commands. Symbolic call stacks, $\hat{cs}$, differ from concrete stacks in that they contain symbolic stores. For $\hat{\sigma} = (\hat{h}, \hat{d}, \hat{\rho}, \pi)$, we write $\hat{\sigma} \wedge \hat{b}$ to denote $(\hat{h}, \hat{d}, \hat{\rho}, \pi \wedge \hat{b})$, $\hat{\sigma}.\mathsf{hp}$ to denote $\hat{h}$, and $\hat{\sigma}.\mathsf{pc}$ to denote π.

Symbolic variables evaluate to themselves: $\mathcal{E}v_S(\hat{x}) = \hat{x}$. As they have meaning only in the symbolic semantics, they cannot be evaluated in the concrete/instrumented semantics. The GetCell of the symbolic semantics non-deterministically branches on the inspected property $\hat{p}$ being equal to any of the properties of the object at location $\hat{l}$ [GetCell - Found], and also on it being different from all of its properties [GetCell - Not Found]. In all cases, the relevant information (highlighted in blue) is recorded in the path condition. This non-determinism contrasts with the concrete/instrumented semantics, where it occurs only in object allocation. The Assume rule strengthens the path condition with the symbolic expression assumed to hold. The two Check SAT rules check satisfiability of a given expression under the current path condition.

Example: Symbolic Branching. To better understand the symbolic execution, consider the code on the right and its symbolic execution in Fig. 5 (right). This code:

```
1  o := new ()
2  o[ŝ] := 42
3  x := getProps (o)
4  assert (card x = 2)
```

(1) creates a new object o; **(2)** assigns 42 to the symbolic property ŝ of o; **(3)** collects the properties of o; and **(4)** asserts that o has two properties in the end. The last assertion will cause the symbolic execution to fail. The key insight is that the symbolic execution branches when executing the property assignment (line 2). There, it can either use the rule [GetCell - Found], where the inspected property coincides with one of the existing properties ($\hat{s}$ = *"@proto"*), or the rule [GetCell - Not Found], where the property is different from all of the existing properties ($\hat{s} \neq$ *"@proto"*). In the left branch, o has only a single property *"@proto"* (that gets updated to 42), whereas in the right branch, it has two properties, *"@proto"* and $\hat{s}$. Therefore, the assert command fails in the left branch and the symbolic execution produces the concrete counter-model $\hat{s}$ = *"@proto"*.

Formal Guarantees. We relate the symbolic to the instrumented semantics using *symbolic environments*, $\varepsilon : \hat{\mathcal{X}} \rightharpoonup \mathcal{V}$, mapping symbolic variables to concrete values. We assume that symbolic environments preserve types (e.g,. symbolic strings are mapped to strings). Given a symbolic environment ε, we use $\mathcal{I}_{\varepsilon}(\hat{e})$ to denote the interpretation of $\hat{e}$ under ε, with the key case being that of symbolic variables: $\mathcal{I}_{\varepsilon}(\hat{x}) = \varepsilon(\hat{x})$. In the standard way, we extend $\mathcal{I}_{\varepsilon}$ to symbolic heaps/domains/stores/call stacks, as well as programs. We use $\mathcal{I}_{\varepsilon}^{\mathsf{s} \rightarrow \mathsf{i}}(\hat{\sigma})$ to denote the instrumented state obtained by interpreting $\hat{\sigma}$ under ε: $\mathcal{I}_{\varepsilon}^{\mathsf{s} \rightarrow \mathsf{i}}((\hat{h}, \hat{d}, \hat{\rho}, \pi)) = (\mathcal{I}_{\varepsilon}(\hat{h}), \mathcal{I}_{\varepsilon}(\hat{d}), \mathcal{I}_{\varepsilon}(\hat{\rho}))$ if $\mathcal{I}_{\varepsilon}(\pi) = \mathsf{true}$; and is undefined otherwise. In all theorems, all non-quantified variables are implicitly universally quantified.

Theorem 3.3 states that, given a symbolic trace, $p : \langle \hat{\sigma}, \hat{cs}, i \rangle^{\mu} \rightsquigarrow_{\mathsf{s}}^{*} \langle \hat{\sigma}', \hat{cs}', j \rangle^{\mu'}$, and an instrumented state in the interpretation of $\hat{\sigma}$ *filtered by the final path condition*, there is an instrumented trace that will produce a final instrumented state in the interpretation of the final symbolic state. We use the final path condition $\hat{\sigma}'.\mathsf{pc}$ when picking the initial instrumented state because we only care about the initial instrumented states for which the instrumented execution follows the same path as the symbolic execution.

Theorem 3.3 (Bounded Soundness).

$$\begin{aligned} &p : \langle \hat{\sigma}, \hat{cs}, i \rangle^{\mu} \rightsquigarrow_{S}^{*} \langle \hat{\sigma}', \hat{cs}', j \rangle^{\mu'} \;\wedge\; \underline{\sigma} = \mathcal{I}_{\varepsilon}^{s \rightarrow i}(\hat{\sigma} \;\wedge\; \hat{\sigma}'.\mathsf{pc}\,) \\ &\quad \wedge\, cs = \mathcal{I}_{\varepsilon}(\hat{cs}) \;\Rightarrow\; \exists\, \underline{\sigma}', cs' \,.\, \mathcal{I}_{\varepsilon}(p) : \langle \underline{\sigma}, cs, i \rangle^{\mu} \rightsquigarrow_{i}^{*} \langle \underline{\sigma}', cs', j \rangle^{\mu'} \\ &\qquad\qquad \wedge\, \underline{\sigma}' = \mathcal{I}_{\varepsilon}^{s \rightarrow i}(\hat{\sigma}') \;\wedge\; cs' = \mathcal{I}_{\varepsilon}(\hat{cs}') \end{aligned}$$

3.5 Linking the Semantics

The last step of our methodology consists of linking the symbolic semantics to the concrete semantics in a way that describes the frames that can be safely added to the initial state. In other words, the interpretation of symbolic states needs to describe both the concretisation of that state and the safe frames for that concretisation.

Below, we give the formal interpretation of symbolic states, $\mathcal{I}_{\varepsilon}^{\mathsf{s} \rightarrow \mathsf{c}}(\hat{\sigma})$, which denotes a set of pairs of the form (σ, h_f), where σ is the concrete state obtained from $\hat{\sigma}$ using ε, and h_f is a concrete heap frame that does not overlap with the resource of σ. The positive resource is collected by the first component of $\mathcal{I}_{\varepsilon}^{\mathsf{s} \rightarrow \mathsf{c}}(\hat{h})$, whereas the negative resource is collected by its second component and by $\mathcal{I}_{\varepsilon}^{\mathsf{s} \rightarrow \mathsf{c}}(\hat{d})$. Here, we represent the negative resource as a set δ of pairs of the form (l, p) capturing the object properties that must not exist in the heap.

Symbolic State Interpretation

Empty Heap

$$\mathcal{I}_{\varepsilon}^{\mathsf{s} \rightarrow \mathsf{c}}(\emptyset) \triangleq \emptyset, \emptyset$$

Non-none Cell

$$\frac{h = (\mathcal{I}_{\varepsilon}(\hat{l}), \mathcal{I}_{\varepsilon}(\hat{p})) \mapsto \mathcal{I}_{\varepsilon}(\hat{v})}{\mathcal{I}_{\varepsilon}^{\mathsf{s} \rightarrow \mathsf{c}}((\hat{l}, \hat{p}) \mapsto \hat{v}) \triangleq (h, \emptyset)}$$

None Cell

$$\frac{\hat{h} = (\hat{l}, \hat{p}) \mapsto \varnothing \qquad \delta = \{\, (\mathcal{I}_{\varepsilon}(\hat{l}), \mathcal{I}_{\varepsilon}(\hat{p})) \,\}}{\mathcal{I}_{\varepsilon}^{\mathsf{s} \rightarrow \mathsf{c}}(\hat{h}) \triangleq (\emptyset, \delta)}$$

Heap Composition

$$\frac{\mathcal{I}_{\varepsilon}^{\mathsf{s} \rightarrow \mathsf{c}}(\hat{h}_i) = h_i, \delta_i \;|_{i=1,2} \qquad h = h_1 \uplus h_2 \quad \delta = \delta_1 \uplus \delta_2}{\mathcal{I}^{\mathsf{s} \rightarrow \mathsf{c}}(\hat{h}_1 \uplus \hat{h}_2) \triangleq h, \delta}$$

Symbolic Domains

$$\frac{\delta = \{(l, p) \mid \hat{l} \in \mathsf{dom}(\hat{d}) \wedge l = \mathcal{I}_{\varepsilon}(\hat{l}) \wedge p \notin \mathcal{I}_{\varepsilon}(\hat{d}(\hat{l}))\}}{\mathcal{I}_{\varepsilon}^{\mathsf{s} \rightarrow \mathsf{c}}(\hat{d}) \triangleq \delta}$$

Symbolic States

$$\frac{\hat{\sigma} = (\hat{h}, \hat{d}, \hat{\rho}, \pi) \quad \mathcal{I}_{\varepsilon}^{\mathsf{s} \rightarrow \mathsf{c}}(\hat{h}) = h, \delta_1 \qquad \mathcal{I}_{\varepsilon}^{\mathsf{s} \rightarrow \mathsf{c}}(\hat{d}) = \delta_2 \quad \mathcal{I}_{\varepsilon}(\hat{\rho}) = \rho \quad \mathcal{I}_{\varepsilon}(\pi) = \mathsf{true}}{\mathcal{I}_{\varepsilon}^{\mathsf{s} \rightarrow \mathsf{c}}(\hat{\sigma}) \triangleq \{((h, \rho), h_f) \mid \mathsf{dom}(h_f) \cap (\mathsf{dom}(h) \cup \delta_1 \cup \delta_2) = \emptyset\}}$$

Theorem 3.4 states the bounded soundness of the JSIL symbolic execution with respect to the concrete semantics. Moreover, this theorem quantifies over all possible frames, which is not the case for standard results in symbolic execution [30, 42, 44, 52], which do not establish frame resilience.

Theorem 3.4 (Bounded Soundness + Frame Resilience).

$$\begin{aligned} &p : \langle \hat{\sigma}, \hat{cs}, i \rangle^{\mu} \rightsquigarrow_{S}^{*} \langle \hat{\sigma}', \hat{cs}', j \rangle^{\mu'} \\ &\quad \wedge\, (\sigma, h_f) \in \mathcal{I}_{\varepsilon}^{S \rightarrow C}(\hat{\sigma} \;\wedge\; \hat{\sigma}'.\mathsf{pc}\,) \;\wedge\; cs = \mathcal{I}_{\varepsilon}(\hat{cs}) \\ &\qquad \Longrightarrow\; \exists\, \sigma', cs' \,.\, \mathcal{I}_{\varepsilon}(p) : \langle \sigma \uplus h_f, cs, i \rangle^{\mu} \rightsquigarrow_{C}^{*} \langle \sigma' \uplus h_f, cs', j \rangle^{\mu'} \\ &\qquad\qquad \wedge\, (\sigma', h_f) \in \mathcal{I}_{\varepsilon}^{S \rightarrow C}(\hat{\sigma}') \;\wedge\; cs' = \mathcal{I}_{\varepsilon}(\hat{cs}') \end{aligned}$$

The *bug-finding* corollary (Corollary 3.5) states that if we find a symbolic trace that results in a failed assertion, then there also exists a concrete execution for which that assertion fails. The analysis is designed in such a way that there are no false positives: if we find a failing symbolic trace, we can always instantiate its symbolic values to obtain a concrete counter-model for the failing assertion.

Corollary 3.5 (Bug-finding).

$$\begin{aligned} &p : \langle \hat{\sigma}, \hat{cs}, i \rangle^{\mu} \rightsquigarrow_{S}^{*} \langle -, -, j \rangle^{\perp} \\ &\quad \Longrightarrow\; \exists \varepsilon \,.\, (\sigma, -) \in \mathcal{I}_{\varepsilon}^{S \rightarrow C} \;\wedge\; \mathcal{I}_{\varepsilon}(p) : \langle \sigma, \mathcal{I}_{\varepsilon}(\hat{cs}), i \rangle^{\mu} \rightsquigarrow_{C}^{*} \langle -, -, j \rangle^{\perp} \end{aligned}$$

3.6 Implementation

Our three semantics for JSIL (§3.2–§3.4) are specifically designed with implementation in mind: one only needs to follow the rules, which are written operationally, modularly, and are syntax-directed.

However, implementing the entailment engine required by the symbolic semantics and practically handling the state explosion problem [15] is a challenging task. For this reason, we develop the first, proof-of-concept version of Cosette leveraging on Rosette [48, 49], a symbolic virtual machine that enables the development of solver-aided languages. Rosette extends Racket [37] with symbolic values and first-order assertions over these values, and languages implemented in Rosette can make use of its solver-aided facilities.

We implement the *instrumented* JSIL interpreter of §3.3 in Rosette, obtaining a JSIL symbolic interpreter for free. We show that this symbolic interpreter is consistent with the symbolic semantics given in §3.4, as described in §1. As Rosette does not finitise the symbolic execution of loops branching on symbolic values [1], the user must specify the maximum allowed number of such branchings. Once that bound is reached, the symbolic execution stops.

3.7 Lifting the results to JavaScript

All the results presented in this section lift to JavaScript in a straightforward way. One simply has to: **(1)** extend JavaScript with constructs for creating symbolic values and asserting and assuming first-order formulae; and **(2)** compile JavaScript to JSIL using a correct compiler. Here, we make use of JS-2-JSIL [22], a thoroughly tested and formally validated compiler from JavaScript to JSIL.

To illustrate the lifting, we restate Corollary 3.5 for JavaScript. Given a JavaScript symbolic state, $\hat{\sigma}_{\text{JS}}$, and program s, we translate both the state and program to JSIL, obtaining a JSIL symbolic state and context, $(\hat{\sigma}, \hat{cs})$, and a JSIL program p. We then run the obtained JSIL program in the symbolic semantics. If there is an assertion failure at the JSIL level, then, by the correctness of JS-2-JSIL, that failure is guaranteed to exist at the JavaScript level as well. Formally:

$$\begin{array}{l} C(\hat{\sigma}_{\text{JS}}) = (\hat{\sigma}, \hat{cs}) \ \wedge\ C(s) = p \ \wedge\ p : \langle \hat{\sigma}, \hat{cs}, i \rangle^{\mu} \rightsquigarrow_{\text{s}}^{*} \langle -, -, j \rangle^{\bot} \\ \quad \implies \ \exists\, \sigma_{\text{JS}} \,.\, \langle \sigma_{\text{JS}}, s \rangle \rightsquigarrow_{\text{JS}}^{*} \bot \end{array}$$

4 SPECIFICATION-DRIVEN BUG-FINDING

We show how to use Cosette for debugging JSIL code annotated with separation logic (SL) specifications. To achieve this, we extend the JSIL symbolic interpreter with a mechanism for asserting SL-assertions (§4.1) and show how to implement this mechanism by giving a sound decision procedure for solving the frame inference problem (FIP) [7] in the context of symbolic execution (§4.2). In §4.3, we present an algorithm for generating symbolic tests from SL-specifications, which guarantees that when a symbolic test fails, Cosette produces a concrete counter-model that invalidates the corresponding specification. We conclude with a discussion on how to lift the presented results to JavaScript (§4.4).

4.1 Symbolic Execution with SL-Assertions

JSIL Logic assertions [22] provide a way of describing *partial* symbolic states. They include boolean and comparison operations on JSIL expressions; the separating conjunction; and assertions for describing heaps. The emp assertion describes an empty heap. The *cell assertion*, $(e_1, e_2) \mapsto e_3$, states that the object at the location denoted by e_1 has the property denoted by e_2 with value denoted by e_3. The *object domain* assertion $\texttt{noProps}(e_1, e_2)$ states that the object at the location denoted by e_1 has at most the properties included in the set denoted by e_2. For instance, the assertion $\texttt{noProps}(\hat{l}, \{\, p_1, p_2 \,\})$ states that the object at location $\hat{l}$ has *at most* the properties p_1 and p_2; it might only have one of them, or none at all, but it cannot have others. We refer to assertions that are distinct from emp and $- * -$ as *simple assertions*, and use p, q to range over them.

JSIL Logic Assertions

$R, S \triangleq \text{true} \mid \text{false} \mid \neg R \mid R \wedge S \mid R \vee S \mid e = e \mid e \leq e$	Pure Asrts.
$P, Q \triangleq R \mid \texttt{emp} \mid (e, e) \mapsto e \mid P * Q \mid \texttt{noProps}(e, e)$	Asrts.

Without loss of generality, we implicitly assume that different symbolic locations denote different concrete locations. Furthermore, given a cell assertion $(e, -) \mapsto -$, we always assume e to be either a concrete location l or a symbolic location $\hat{l}$. Note that a symbolic state $\hat{\sigma} = (\hat{h}, \hat{d}, \hat{\rho}, \pi)$ corresponds to the assertion:

$$\Big(\circledast_{(\hat{l},\hat{p}) \in \text{dom}(\hat{h})}\ (\hat{l}, \hat{p}) \mapsto \hat{h}(\hat{l}, \hat{p}) \Big) * \Big(\circledast_{\hat{l} \in \text{dom}(\hat{d})}\ \texttt{noProps}(\hat{l}, d(\hat{l})) \Big) \\ * \Big(\textstyle\bigwedge_{x \in \text{dom}(\hat{\rho})}\ x = \hat{\rho}(x) \Big) * \pi$$

where $\circledast$ denotes the iterated separating conjunction [39].

Analogously, an assertion can be normalised into a symbolic state, by: mapping program variables in P to freshly generated logical variables of the appropriate type, obtaining a symbolic store $\hat{\rho}$; replacing all program variables in P with their bindings given by $\hat{\rho}$, obtaining an assertion P' with no program variables; and collecting all cell assertions in P' to form the symbolic heap, all object domain assertions to form the domain table, and all pure assertions to form the path condition. We refer to the normalised symbolic state corresponding to P by $\mathcal{N}(P)$. We also use $\mathcal{I}(\hat{\sigma})$ for denoting the set $\{(\sigma, h_f) \mid \exists \varepsilon \,.\, (\sigma, h_f) \in \mathcal{I}_{\varepsilon}^{\text{s} \to \text{c}}(\hat{\sigma})\}$ and $\mathcal{I}(P)$ for $\mathcal{I}(\mathcal{N}(P))$.

Inductive Predicates. Cosette does not support symbolic execution over inductive predicates, which are commonplace in SL-style specifications [6, 7]. As in [10], we deal with user-defined inductive predicates by *unfolding* those predicates up to a fixed, user-defined bound. We omit this unfolding mechanism as it is routine.

Asserting SL-Assertions. We extend JSIL commands with a special construct, $\text{assert}_{*}(P)$, for stating that the SL-assertion P must hold whenever that command is evaluated. The corresponding symbolic semantics rules are given below.

$$\textsc{SL-Assert - True}\quad \frac{\text{cmd}(\hat{cs}, i) = \text{assert}_{*}(P) \qquad \textsc{Unify}(\hat{\sigma}, P) = \text{Succ}(\hat{\sigma}_f)}{\langle \hat{\sigma}, \hat{cs}, i \rangle^{\top} \rightsquigarrow_{\text{s}} \langle \hat{\sigma}, \hat{cs}, i{+}1 \rangle^{\top}}$$

$$\textsc{SL-Assert - False}\quad \frac{\text{cmd}(\hat{cs}, i) = \text{assert}_{*}(P) \qquad \textsc{Unify}(\hat{\sigma}, P) = \text{Fail}(\pi_f) \qquad (\hat{\sigma}.\text{pc} \wedge \pi_f)\ \text{SAT}}{\langle \hat{\sigma}, \hat{cs}, i \rangle^{\top} \rightsquigarrow_{\text{s}} \langle \hat{\sigma}, \hat{cs}, i \rangle^{\bot}}$$

The rules use a partial decision procedure (PDP), $\textsc{Unify}(\hat{\sigma}, P)$, for determining if a given symbolic state $\hat{\sigma}$ satisfies an assertion P. There are two important criteria that $\textsc{Unify}(\hat{\sigma}, P)$ must satisfy:

(1) **Soundness**: If $\textsc{Unify}(\hat{\sigma}, P) = \text{Succ}(\hat{\sigma}_f)$, then it must hold that $\mathcal{I}(\hat{\sigma}) \subseteq \mathcal{I}(\hat{\sigma}_f \circ \mathcal{N}(P))$;[6]

(2) **Bug-Finding**: If $\textsc{Unify}(\hat{\sigma}, P) = \text{Fail}(\pi_f)$, then it must hold that $\mathcal{I}(\hat{\sigma} \wedge \pi_f) \cap \mathcal{I}(P) = \emptyset$. Observe that every concrete state and heap frame in $\mathcal{I}(\hat{\sigma} \wedge \pi_f)$ are counter-models for P. Also note

[6]We use $\circ$ for the composition of two symbolic states, defined component-wise as disjoint union for the heaps/domains/stores, and conjunction for the path conditions.

that, in the SL-Assert - False rule, the semantics only triggers an assertion failure when it finds a concrete witness for the failure, as it explicitly requires that $(\hat{\sigma}.\text{pc} \wedge \pi_f)$ be satisfiable.

4.2 The Frame Inference Problem

We describe the $\textsc{Unify}(\hat{\sigma}, P)$ PDP that we use as part of the JSIL symbolic interpreter, for proving entailments between symbolic states and finding counter models in case of failure. As is customary [9, 22, 24], this PDP first uses *pattern-matching* on the spatial part of the symbolic state, and then discharges the pure part of the entailment to an external constraint solver (in our case, Rosette).

When solving $\textsc{Unify}(\hat{\sigma}, P)$, the symbolic variables of P not in $\hat{\sigma}$ are assumed to be existentially quantified. As in [33], we topologically sort the simple assertions in P to find the appropriate bindings for such variables. For lack of space, we describe the frame inference algorithm for the setting without existentially quantified variables.

Given a symbolic state $\hat{\sigma}$ and an assertion P, $\textsc{Unify}(\hat{\sigma}, P)$ replaces the program variables in P with their bindings from $\hat{\sigma}.\text{sto}$ and then calls the Frame Inference Algorithm (Algorithm 1) on the list of all simple assertions in P. This algorithm uses two functions:

FIP GetCell. In case of success, $\mathcal{GC}_{\text{FIP}}(\hat{\sigma}, \hat{l}, \hat{p})$ returns the symbolic expression $\hat{v}$ associated with $(\hat{l}, \hat{p})$ in the heap component of $\hat{\sigma}$ and the state obtained by removing that cell from $\hat{\sigma}.\text{hp}$.

FIP GetDomain. In case of success, $\mathcal{GD}_{\text{FIP}}(\hat{\sigma}, \hat{l})$ returns the symbolic expression $\hat{v}_d$, denoting the domain of the object at location $\hat{l}$ in $\hat{\sigma}$, and the state obtained by removing all the negative resource associated with $\hat{l}$ from $\hat{\sigma}$.

We give selected rules for these functions, analogous to those in §3.4, except that: **(1)** both functions return a new symbolic state from which the matched resource is removed and **(2)** their corresponding constraints are lifted to the premise (highlighted in blue). In case of failure, both functions return a constraint π_f, under which the inspected resource is guaranteed not to exist (highlighted in red).

Selected FIP Rules

$\textsc{GetDomain}$
$$\frac{\begin{array}{c}\hat{h} = \hat{h}' \uplus \left((\hat{l}, \hat{p}_i) \mapsto \hat{v}_i\right) |_{i=0}^{m} \quad (\hat{l}, -) \notin \text{dom}(\hat{h}') \\ \forall_{0 \leq i \leq n}\, \hat{v}_i \neq \varnothing \quad \forall_{n < i \leq m}\, \hat{v}_i = \varnothing \quad \hat{v} = \{\hat{p}_i |_{i=n+1}^{m}\} \\ \hat{h}'' = (\hat{l}, \hat{p}_i) \mapsto \hat{v}_i |_{i=0}^{n} \quad \hat{d} = \hat{d}' \uplus (\hat{l} \mapsto \hat{v}')\end{array}}{\mathcal{GD}_{\text{FIP}}((\hat{h}, \hat{d}, \hat{\rho}, \pi), \hat{l}) \triangleq \text{Succ}((\hat{v}' \backslash \hat{v}), (\hat{h}' \uplus \hat{h}'', \hat{d}', \hat{\rho}, \pi))}$$

$\textsc{GetCell - Found}$
$$\frac{\begin{array}{c}(\hat{h}, \hat{d}, \hat{\rho}, \pi) = \hat{\sigma} \\ \hat{h} = \hat{h}' \uplus (\hat{l}, \hat{p}') \mapsto \hat{v} \\ \pi \vdash (\hat{p} = \hat{p}') \quad \hat{\sigma}' = (\hat{h}', \hat{d}, \hat{\rho}, \pi)\end{array}}{\mathcal{GC}_{\text{FIP}}(\hat{\sigma}, \hat{l}, \hat{p}) \triangleq \text{Succ}(\hat{v}, \hat{\sigma}')}$$

$\textsc{GetCell - Not Found}$
$$\frac{\begin{array}{c}(\hat{h}, \hat{d}, \hat{\rho}, \pi) = \hat{\sigma} \quad \pi \vdash \hat{p} \notin \hat{d}(\hat{l}) \\ \hat{\sigma}' = (\hat{h}, \hat{d}[\hat{l} \mapsto \hat{d}(\hat{l}) \cup \{\hat{p}\}], \hat{\rho}, \pi)\end{array}}{\mathcal{GC}_{\text{FIP}}(\hat{\sigma}, \hat{l}, \hat{p}) \triangleq \text{Succ}(\varnothing, \hat{\sigma}')}$$

$\textsc{GetCell - Fail with Domain Info}$
$$\frac{\begin{array}{c}\hat{h} = \hat{h}'' \uplus \left((\hat{l}, \hat{p}_i) \mapsto \hat{v}_i\right) |_{i=0}^{m} \\ (\hat{l}, -) \notin \text{dom}(\hat{h}'') \quad \pi \nvdash \hat{p} \notin \hat{d}(\hat{l}) \quad \pi \nvdash \hat{p} = \hat{p}_i |_{i=0}^{m}\end{array}}{\mathcal{GC}_{\text{FIP}}((\hat{h}, \hat{d}, \hat{\rho}, \pi), \hat{l}, \hat{p}) \triangleq \text{Fail}((\hat{p} \in \hat{d}(\hat{l})) \wedge (\wedge_{i=0}^{m}(\hat{p}_i \neq \hat{p})))}$$

$\textsc{GetCell - Fail without Domain Info}$
$$\frac{\begin{array}{c}\hat{h} = \hat{h}'' \uplus \left((\hat{l}, \hat{p}_i) \mapsto \hat{v}_i\right) |_{i=0}^{m} \\ (\hat{l}, -) \notin \text{dom}(\hat{h}'') \quad \hat{l} \notin \text{dom}(\hat{d}) \quad \pi \nvdash \hat{p} = \hat{p}_i |_{i=0}^{m}\end{array}}{\mathcal{GC}_{\text{FIP}}((\hat{h}, \hat{d}, \hat{\rho}, \pi), \hat{l}, \hat{p}) \triangleq \text{Fail}(\wedge_{i=0}^{m}(\hat{p}_i \neq \hat{p}))}$$

Algorithm 1 Frame Inference for Symbolic States

```
function Unification($\hat{\sigma}, \overline{p}$)
    match $\overline{p}$ with
    | [ ] : return Succ ($\hat{\sigma}$)
    | $(\hat{l}, \hat{p}) \mapsto \hat{v} :: \overline{q}$ :
        match $\mathcal{GC}_{\text{FIP}}(\hat{\sigma}, \hat{l}, \hat{p})$ with
        | Succ ($\hat{v}', \hat{\sigma}'$) : if ($\hat{\sigma}.\text{pc} \vdash \hat{v} = \hat{v}'$)
            then return Unification($\hat{\sigma}', \overline{q}$)
            else return Fail ($\hat{v} \neq \hat{v}'$)
        | Fail ($\pi_f$) : return Fail ($\pi_f$)
    | noProps($\hat{l}, \hat{v}$) :: $\overline{q}$ :
        match $\mathcal{GD}_{\text{FIP}}(\hat{\sigma}, \hat{l})$ with
        | Succ ($\hat{v}', \hat{\sigma}'$) : if ($\hat{\sigma}.\text{pc} \vdash \hat{v} \backslash \hat{v}' = \{\hat{p}_1, \ldots, \hat{p}_n\}$)
            then return Unification($\hat{\sigma}' \uplus (\hat{l}, \hat{p}_i) \mapsto \varnothing |_{i=1}^{n}, \overline{q}$)
            else return Fail ($\hat{v}' \nsubseteq \hat{v}$)
        | Fail ($\pi_f$) : return Fail ($\pi_f$)
    | $S :: \overline{q}$ : if ($\hat{\sigma}.\text{pc} \vdash S$)
        then return Unification($\hat{\sigma}, \overline{q}$)
        else return Fail ($\neg S$)
end function
```

The Frame Inference Algorithm has four possible cases:

- $\overline{p} = [\,]$: There is nothing left to unify; we return the current symbolic state, which constitutes the frame.
- $\overline{p} = (\hat{l}, \hat{p}) \mapsto \hat{v} :: -$: Using $\mathcal{GC}_{\text{FIP}}$, we obtain the symbolic expression $\hat{v}'$ associated with $(\hat{l}, \hat{p})$ in the current symbolic state and check that $\hat{v}' = \hat{v}$ under the current path condition. If the entailment holds, we proceed. Otherwise, we generate the *failing constraint* $\hat{v}' \neq \hat{v}$.
- $\overline{p} = \text{noProps}(\hat{l}, \hat{v}) :: -$: Using $\mathcal{GD}_{\text{FIP}}$, we obtain the domain $\hat{v}'$ of $\hat{l}$ in the current symbolic state and check that $\hat{v}' \subseteq \hat{v}$ under the current path condition. If the entailment holds, we extend the current symbolic state with the negative resource associated with $\hat{l}$ not captured by $\text{noProps}(\hat{l}, \hat{v})$ and proceed. Otherwise, we generate the *failing constraint* $\hat{v}' \nsubseteq \hat{v}$.
- $\overline{p} = S :: -$: If S is entailed by the current path condition, we proceed. Otherwise, we generate the *failing constraint* $\neg S$.

Example: Unification Algorithm. Let $\hat{\sigma} = (\emptyset, \hat{h}, \hat{d}, \text{true})$, where $\hat{h} = [(\hat{l}, \hat{p}_1) \mapsto \hat{v}]$ and $\hat{d} = [\hat{l} \mapsto \{\hat{p}_1\}]$. If the next assertion to be unified is $\text{noProps}(l, \{\hat{p}_1\})$, the unification will succeed due to the [GetDomain] rule. The heap of the resulting symbolic state will be equal to $\hat{h}$ and the domain table will be empty. On the other hand, if the next assertion is $\text{noProps}(l, \{\hat{p}_2\})$, the unification will fail. The [GetDomain] rule will return $\hat{v}' = \{\hat{p}_1\}$, but since we have no information on $\hat{p}_2$, we will not be able to determine the set difference $\hat{v}' \backslash \hat{v}$ and will fail with the failing constraint $\{\hat{p}_2\} \subsetneq \{\hat{p}_1\}$.

Formal Guarantees. $\textsc{Unify}(\hat{\sigma}, P)$ meets the criteria outlined in §4.1.

Theorem 4.1 (Soundness of FIP). *The following hold:*

(1) $\textsc{Unify}(\hat{\sigma}, P) = \text{Succ}(\hat{\sigma}_f) \implies \mathcal{I}(\hat{\sigma}) \subseteq \mathcal{I}(\hat{\sigma}_f \circ \mathcal{N}(P))$

(2) $\textsc{Unify}(\hat{\sigma}, P) = \text{Fail}(\pi_f) \implies \mathcal{I}(\hat{\sigma} \wedge \pi_f) \cap \mathcal{I}(P) = \emptyset$

4.3 From Specifications to Symbolic Tests

JSIL Logic specifications have the form $fl : \{P\}\, f(\overline{x})\, \{Q\}$, where P and Q are the pre- and postconditions of the procedure with identifier f and formal parameters $\overline{x}$. Each specification is associated with

$$\begin{array}{ll} \mathcal{T}(\mathit{fl}:\{P\}\,f(x_0,\ldots,x_n)\,\{Q\}) \triangleq & \mathcal{T}_{\mathsf{nm}}(f,\hat{x}_i|_{i=0}^{n},Q) \triangleq \textsf{proc main () \{} \\ \quad \textbf{let } \hat{\rho}=[x_i\mapsto \hat{x}_i|_{i=0}^{n}] \textbf{ in} & \quad 0 \,:\, x := f(\hat{x}_0,\ldots,\hat{x}_n) \textsf{ with } i_{\mathsf{er}} \\ \quad \textbf{let } \hat{\sigma},\,Q'=\mathcal{N}(\hat{\rho}(P)),\,\hat{\rho}(Q) \textbf{ in} & \quad i_{\mathsf{nm}} \,:\, \textsf{assert}_*\,(Q[x/\texttt{ret}]) \\ \quad \textbf{let } \mathit{proc}=\mathcal{T}_{\mathit{fl}}(f,\hat{x}_i|_{i=0}^{n},Q') \textbf{ in} & \quad i_{\mathsf{er}} \,:\, \textsf{assert}\,(\textsf{false}) \\ \quad\quad (\mathit{proc},\hat{\sigma}) & \} \end{array}$$

Figure 6: Symbolic Test Generation Algorithm

a return mode $\mathit{fl} \in \{\mathsf{nm}, \mathsf{er}\}$, indicating if the function returns normally or with an error. Intuitively, a specification $\mathit{fl} : \{P\}\,f(\overline{x})\,\{Q\}$ is valid for a given JSIL program p, if p contains a procedure with identifier f and "whenever f is executed in a state satisfying P, then, if it terminates, it does so in a state satisfying Q, with return mode fl". The formal definition is given below.

Definition 4.2 (Validity of JSIL Logic Specifications). A JSIL logic specification $\mathit{fl} : \{P\}\,f(\overline{x})\,\{Q\}$ is valid with respect to a program p, written $p \models \mathit{fl} : \{P\}\,f(\overline{x})\,\{Q\}$, if and only if it holds that:

$$\begin{array}{l}(\sigma, h_f) \in \mathcal{I}(P) \;\wedge\; \langle \sigma \uplus h_f, cs, 0\rangle^{\top} \leadsto^{*}_{\mathsf{c}} \langle \sigma', cs, i_{\mathit{fl}'}\rangle^{\top} \\ \quad\quad \Longrightarrow\; \mathit{fl}' = \mathit{fl} \;\wedge\; \exists \sigma''\,.\, \sigma' = \sigma'' \uplus h_f \;\wedge\; (\sigma'', h_f) \in \mathcal{I}(Q)\end{array}$$

for any call stack cs of the form $(f, -, -, -, -) :: -$.

Given a JSIL program p containing a procedure f with specification $\mathit{fl} : \{P\}\,f(\overline{x})\,\{Q\}$, our goal is to construct a symbolic test for checking if f behaves as its specification mandates. A symbolic test is a pair $(\mathit{proc}, \hat{h})$ consisting of a JSIL procedure with the code of the test and the initial symbolic heap on which to execute the test. Figure 6 presents the test generation procedure. The test generation function $\mathcal{T}$ is defined in terms of two auxiliary functions, $\mathcal{T}_{\mathsf{nm}}$ and $\mathcal{T}_{\mathsf{er}}$, for generating tests for nm-mode and er-mode specifications, respectively. For space reasons, we only present $\mathcal{T}_{\mathsf{nm}}$ ($\mathcal{T}_{\mathsf{er}}$ is equivalent). The test program p', denoted by $p[\mathsf{main} \mapsto \mathit{proc}]$, is obtained from the original program p and the test procedure proc by replacing the main of p with the new test procedure, proc.

Formal Guarantees. If the symbolic execution of a test generated for $\mathit{fl} : \{P\}\,f(\overline{x})\,\{Q\}$ finds a bug, the specification is not valid.

Theorem 4.3 (Bug-finding for JSIL SL Specifications).

$$\begin{array}{l}\mathcal{T}(\mathit{fl} : \{P\}\,f(\overline{x})\,\{Q\}) = (\mathit{proc}, \hat{\sigma}) \;\wedge \\ \quad p[\mathit{main} \mapsto \mathit{proc}] : \langle \hat{\sigma}, cs_{\mathsf{main}}, 0\rangle^{\top} \leadsto^{*}_{\mathcal{S}} \langle -, -, -\rangle^{\perp} \\ \quad\quad \Longrightarrow\; p \not\models \mathit{fl} : \{P\}\,f(\overline{x})\,\{Q\}\end{array}$$

4.4 Lifting the results to JavaScript

JaVerT specifications of JS functions are analogous to JSIL specifications of JSIL procedures. A JaVerT specification $\mathit{fl} : \{P_{\mathrm{JS}}\}\,f(\overline{x})\,\{Q_{\mathrm{JS}}\}$ is valid for a JS program s, written $s \models \mathit{fl} : \{P_{\mathrm{JS}}\}\,f(\overline{x})\,\{Q_{\mathrm{JS}}\}$ if and only if s contains a function literal with identifier f and "whenever f is executed in a state satisfying P_{JS}, then, if it terminates, it does so in a state satisfying Q_{JS}, with return mode fl". JS-2-JSIL is proven to correctly compile JaVerT specifications to JSIL specifications [22].

Testing JaVerT Specifications. We generate symbolic tests from a JS program and its JaVerT specifications by: converting them to a JSIL program with JSIL specifications, using JS-2-JSIL; generating a set of symbolic tests from the obtained JSIL program and JSIL specifications, as described in §4.3; and running the generated JSIL symbolic tests using the JSIL symbolic semantics described in §3.4. If Cosette finds a bug while running the tests, we will obtain a concrete counter-example triggering a specification violation.

Formal Guarantees. The correctness of JS-2-JSIL ensures that a JaVerT specification $\mathit{fl} : \{P_{\mathrm{JS}}\}\,f(\overline{x})\,\{Q_{\mathrm{JS}}\}$ is valid for a given JS program s *iff* the translated specification $C(\mathit{fl} : \{P_{\mathrm{JS}}\}\,f(\overline{x})\,\{Q_{\mathrm{JS}}\})$ is valid for the compilation of s. Hence, we can lift Theorem 4.3 to the JavaScript level in a straightforward way, as shown below.

Corollary 4.4 (Bug-finding for JaVerT Specifications).

$$\begin{array}{l}\mathcal{T}(C(\mathit{fl} : \{P_{\mathrm{JS}}\}\,f(\overline{x})\,\{Q_{\mathrm{JS}}\})) = (\mathit{proc}, \hat{\sigma}) \;\wedge \\ \quad C(s)[\mathit{main} \mapsto \mathit{proc}] : \langle \hat{\sigma}, cs_{\mathsf{main}}, 0\rangle^{\top} \leadsto^{*}_{\mathcal{S}} \langle -, -, -\rangle^{\perp} \\ \quad\quad \Longrightarrow\; s \not\models \mathit{fl} : \{P_{\mathrm{JS}}\}\,f(\overline{x})\,\{Q_{\mathrm{JS}}\}.\end{array}$$

5 EVALUATION

We discuss the coverage of Cosette and demonstrate that our prototype implementation is already useful for the debugging of real-world JavaScript code.

Cosette Coverage. The coverage of Cosette is dependent on the coverage of JS-2-JSIL. Currently, JS-2-JSIL covers the entire core of ES5 Strict and the majority of the built-in libraries. Several libraries orthogonal to the core, such as the Date, RegExp, and JSON libraries, are not supported. Extending JS-2-JSIL to the non-strict mode of the language and implementing the remaining built-in libraries is a technical exercise. A more substantial engineering effort would be required for moving to the later extensions and revisions of the standard, such as ES6, but this effort would ultimately amount to extending JSIL and JS-2-JSIL rather than re-designing them, as these versions all have ES5 as a common core.

Apart from the coverage of JS-2-JSIL, there are not many limitations to the reasoning capabilities of Cosette. Due to the limitations of Rosette, Cosette does not support the `eval` statement. Also, it supports the `for-in` statement with a fixed property enumeration. This is sound, because if a bug is found for a particular enumeration order, it still exists for an arbitrary enumeration order.

Whole-program Symbolic Testing. We first created a number of simple symbolic tests to demonstrate that Cosette can reason about essential JavaScript features, such as prototype inheritance, function closures, arrays, strings, as well as the substantially more challenging for-in statement and dynamic dispatch.

We have also analysed two real-world data structure libraries: Buckets.js [41], and queue-pri [27]. We chose these two libraries because they exercise many essential JavaScript features, because they come equipped with unit test suites, and because they do not have external dependencies. In addition, Buckets.js is widely used by developers, having over 65k downloads on npm [34].

For both libraries, we wrote comprehensive symbolic unit tests for all functions and compared them with the concrete unit tests that ship with the libraries. The results are presented in Table 2. For each file in the libraries, we report the number of JS executable lines in the code itself and including dependencies (slash-separated), the corresponding numbers of JSIL lines, the number of symbolic unit tests created by us and the number of concrete unit tests created by the library developers, the number of JS lines in the symbolic and concrete tests, their coverage measured as percentage of lines, and the average Cosette run time for the symbolic tests.

The results we obtained clearly demonstrate the benefits of symbolic testing using Cosette. In both cases, we were able to *obtain 100% line coverage*, improving the coverage of the Buckets.js library

Name	JS lines	JSIL lines	# Tests	Test lines	Line Cov. (%)	Avg. time
`arrays`	44/71	1251/1942	9/24	166/329	100/100	20s
`bag`	69/237	2041/7194	7/18	78/265	100/76.8	74s
`bstree`	143/326	3819/8052	11/31	216/759	100/98.6	5m27s
`dict`	57/84	1683/2374	7/14	116/170	100/80.7	15s
`heap`	57/128	2059/4001	4/15	92/626	100/96.5	5m29s
`llist`	126/153	2447/3138	9/21	149/370	100/94.4	24s
`multidict`	56/184	1871/5496	6/16	118/189	100/74.1	1m15s
`pqueue`	26/154	1066/5067	5/12	70/283	100/96.2	5m49s
`queue`	30/183	1095/4233	6/9	111/146	100/96.7	20s
`set`	40/124	1528/3902	6/12	86/271	100/70.0	1m01s
`stack`	23/176	941/4079	4/7	91/104	100/87.0	26s
`queue-pri`	19/164	872/5086	2/9	26/80	100/100	1m18s

Table 2: Tests for the Buckets.js and queue-pri libraries

by 12%. We achieved this using a total of *60% fewer tests*, and each test had on average *10% fewer lines of code.* We also discovered one bug in the Buckets.js library, as well as one in the queue-pri library.

For the testing, we used a machine with an Intel Core i7-4980HQ CPU 2.80 GHz and DDR3 RAM 16GB. We measured the execution time of each symbolic test and averaged the times across tests for each library file. The times we obtain reflect the fact that Rosette code is interpreted, rather than run natively. We aim at implementing our own symbolic execution tool from scratch in the future, which, given our experience with JaVerT, should reduce execution times by at least an order of magnitude.

Bug: Multi-dictionary in Buckets.js. We discovered a bug in the implementation of the Buckets.js multi-dictionary library. A multi-dictionary is a key-value map in which a single key holds an array of *distinct* values. Our symbolic tests for the `remove(key, value)` function, which removes a given key-value pair from the multi-dictionary, have revealed that the library wrongly treats the case in which we try to remove a key-value pair for a key with no associated values. In practice, a runtime error is thrown instead of `remove` returning `false`. This bug was not detected by the concrete unit tests associated with the library due to their incomplete coverage; we have fixed it and submitted an appropriate pull request.

A simplified version of the failing code is shown below. The `parent` variable is an internal variable of the `multidict` object that holds a Buckets.js dictionary from keys to arrays of values. The `arrays.remove` function is part of the Buckets.js array manipulation library, and removes a given value from a JavaScript array.

```
1 multidict.remove = function (key, value) {
2     if (value === undefined) { ... }
3     var array = parent.get(key);
4     if (arrays.remove(array, value)) { ... }
5     return false;
6 }
```

We were able to expose the bug with the following symbolic test:

```
1 var dict = new buckets.multidict();
2 var s = symb_string(), x = symb_number(), y = symb_number();
3
4 dict.set(s, x); dict.set(s, y);
5 var xRemoved = dict.remove(s, x);
6 var yRemoved = dict.remove(s, y);
7 assert (((not (x = y)) and      yRemoved) or
8              ((x = y)  and (not yRemoved)));
```

In the test, we create a new multi-dictionary `dict`, and insert two symbolic numbers `x` and `y` at key `s`, then remove them in order. If `x = y`, the value will be stored in the array only once. This means that the key in the second call to `remove` on line 6 is not present in the multi-dictionary any more, triggering the bug. When running Cosette on this test, we obtain the counter-model `x = y = 0` (any value of `s` works), and running the concrete instantiation of the test in Node.js raises an error. We fixed the bug by adding a check for `undefined` after line 3 in the code of `remove`, after which Cosette successfully discharged the assertion.

Bug: queue-pri. This library implements a priority queue that stores data with an optional priority value, which is either a number (the lower the value, the higher the priority) or `null` if no priority is provided, in which case the associated element is put at the end of the queue. Our symbolic tests of the `enqueue(data, pri)` method of the library show that elements enqueued with priority `0` were being wrongly enqueued at the end of the queue. We traced the bug to the way in which priority was calculated inside `enqueue`: `priority = pri || null`, which evaluates to `null` if the priority is not supplied, but also, due to JS semantics, if it is equal to 0. This bug was not caught by the library unit tests, even though their line coverage was 100%, because the developer had not considered inserting nodes with priority 0. This shows that Cosette is a useful tool for symbolic testing, because it fully follows the semantics of JavaScript and will expose corner cases that a developer may not be aware of.

Specification-driven Bug-finding. For this part of the evaluation, we revisited the JaVerT full functional correctness specifications of simple data structure libraries: key-value maps (cf. §2.2); priority queues; binary search trees (BSTs), and sorted lists [22]. These libraries are substantially smaller than real-world JS libraries.

For these libraries, we first investigated the relationship between the unfolding depth for recursive predicates and the code coverage of the resulting symbolic tests. We have observed that very few unfoldings are sufficient for achieving full coverage. In particular, the tests generated for the key-value map, priority queue, and the sorted list have full coverage already for unfold depth set to 1, while those for the BST reach full coverage for unfold depth set to 2.

To test the compositionality of Cosette, we crippled the specifications of the key-value map and the priority queue, exposing frame-related bugs, such as the ones shown in §2.2. Cosette was already able to detect all bugs with the unfold depth set to 1, which shows that it can successfully reason given partial state information, in contrast to standard symbolic execution tools for JavaScript.

6 RELATED WORK

The existing literature covers a wide range of JavaScript analysis techniques, including: type systems [2, 8, 20, 26, 31, 38, 47], control flow analysis [21], pointer analysis [25, 45] and abstract interpretation [4, 26, 28, 35], among others. We focus on the existing work on symbolic execution and logic-based verification for JavaScript, and discuss general techniques for specification-driven test generation.

Symbolic Execution for JS. The majority of the existing bug-finding symbolic execution tools for JavaScript target specific bug patterns, such as security vulnerabilities related to the misuse of strings [42] (for example, absence of sanitisation before security critical operations), malformed Web API requests [52], and DOM

API specific bugs [30]. These tools are fully automatic and aim at code in the large, primarily focusing on scalability and coverage issues. Cosette has a different purpose: it is not designed to be fully automatic, but to *assist* developers in testing their code. The work closest to ours is *Jalangi* [44], a general-purpose symbolic execution tool for JavaScript that implements a sophisticated state merging algorithm to deal with the problem of symbolic state explosion. However, Jalangi, as all existing symbolic execution tools for JavaScript, does not follow the semantics of the language precisely. In contrast, Cosette is *trustworthy*: it does follow the semantics of JavaScript and its theoretical underpinnings are formalised and proven sound. Therefore, it can be used both as a basis for building other more specific analyses (e.g., combining symbolic execution with a type-based analysis to increase the precision of the latter) and as a testing oracle for other symbolic execution tools for JavaScript that purposely ignore some corner cases of the JavaScript semantics. Furthermore, Cosette is the first symbolic execution tool for JavaScript that provides frame resilience guarantees, which are essential for any specification-related reasoning.

Specification-driven Testing. There is a long line of work on specification-driven test synthesis, dating back to *Quickcheck* [14]. *Quickcheck* and its followers [13, 40] translate Haskell type declarations into comprehensive test-suites. Recently, Seidel et al. [43] proposed *Target*, a test generation tool that advances the agenda of *Quickcheck* by supporting precise refinement types. The key insight of *Target* is to use an SMT solver for finding models for the supported type refinements. Specification-driven test generation has also been successfully applied to Java. *Korat* [11, 32] generates test cases for Java classes annotated with JML specifications [29], but it requires the class code to include a special Java method for checking if the class invariants hold. More recently, Dolby et al. [17] proposed a SAT-based approach for testing Java code annotated with relational logic specifications. To the best of our knowledge, there are no tools for specification-driven testing of JavaScript, and no tools for test generation based on separation logic specifications.

Logic-based Verification for JavaScript. Our work has been strongly influenced by JaVerT [22], a verification toolchain for JavaScript based on separation logic. JaVerT comes with a trusted compiler, JS-2-JSIL, from JavaScript to JSIL. We use JS-2-JSIL in Cosette, extending it with constructs for creating and reasoning about symbolic values. Like Cosette, JaVerT also performs its analysis on compiled JSIL code. However, the separation logic proof rules of JaVerT are not syntax-directed and offer little implementation insight. In contrast, our novel symbolic semantics for JSIL is syntax-directed and specifically designed to guide implementations. The purpose of JaVerT is to verify functional correctness specifications of, for example, data structure libraries. We reuse the specifications of these libraries to evaluate our specification-driven bug finding.

Alternatively, we might have used the matching logic specifications of KJS [16], a verification tool for core ES5 obtained by instantiating the general $\mathbb{K}$ framework with the semantics of JS [36]. Similarly to JaVerT, KJS has been used to verify functional correctness properties of small data structure libraries. However, KJS specifications are difficult to write and error-prone, as the developer has to explicitly address all language internals. They are also not compositional, as they do not allow partial descriptions of JS objects.

There is also the work of Swamy et al. [46], who prove absence of runtime errors for higher-order JavaScript (ES3) programs by: compiling JS programs to the logic of F*; generating verification conditions for the absence of runtime errors; and automatically discharging these conditions using Z3. However, as this analysis does not consider specifications, it was not possible for us to reuse its results for Cosette.

All of the above-mentioned verification tools provide strong correctness guarantees, but have severe scalability limitations, as they require loop invariants and abstractions for the recursive structures that the programs use. For instance, the Buckets.js library would be out of their reach, as they do not come with abstractions to accurately describe, for example, JavaScript arrays, for-in loop invariants, and higher-order functions. As a bug-finding tool, Cosette does not require any such abstractions, and can therefore be used for analysing substantially larger, more complex codebases.

7 CONCLUSIONS

We have presented Cosette, a trustworthy, compositional symbolic execution framework for JavaScript, combining the JS-2-JSIL compiler and our JSIL symbolic interpreter written in Rosette. We have applied Cosette to whole-program symbolic testing of real-world JavaScript libraries and compositional debugging of separation logic specifications of JavaScript programs.

We have developed a methodology for designing compositional program analyses for dynamic languages in general, and symbolic execution for JSIL in particular. We achieved this by introducing a new, abstract semantics for JSIL, which we instantiated to obtain the concrete, instrumented, and symbolic semantics. This abstract semantics is the bedrock for both the theoretical results and the implementation of the analysis. We prove that the JSIL symbolic execution of Cosette is sound, frame-resilient and does not generate false positives. We establish additional trust by using the theory to precisely guide the implementation and by thorough testing.

Cosette brings ideas from current separation logic research to the well-established setting of classical symbolic execution [3]. We believe that it is a stepping stone towards a *fully automatic* compositional symbolic execution tool for JavaScript in the style of Infer [12]. In future, our goal is to implement such a tool, drawing inspiration the JSIL abstract semantics presented here.

ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers, whose comments have improved the overall quality of the paper. Fragoso Santos, Gardner, and Maksimović were supported by the EPSRC Programme Grant 'REMS: Rigorous Engineering for Mainstream Systems' (EP/K008528/1), and the Department of Computing at Imperial College London. Maksimović was partially supported by the Serbian Ministry of Education and Science through the Mathematical Institute of the Serbian Academy of Sciences and Arts, projects ON174026 and III44006.

REFERENCES

[1] S. Anand, C. S. Păsăreanu, and W. Visser. 2009. Symbolic execution with abstraction. *International Journal on Software Tools for Technology Transfer* 11, 1 (01 Feb 2009), 53–67.

[2] C. Anderson, P. Giannini, and S. Drossopoulou. 2005. Towards Type Inference for JavaScript. In *Proceedings of the 19th European Conference on Object-Oriented Programming (ECOOP'05) (Lecture Notes in Computer Science)*. Springer, 428–452.

[3] E. Andreasen, L. Gong, A. Møller, M. Pradel, M. Selakovic, K. Sen, and C.-A. Staicu. 2017. A Survey of Dynamic Analysis and Test Generation for JavaScript. *Comput. Surveys* 50, 5 (2017).

[4] E. Andreasen and A. Møller. 2014. Determinacy in Static Analysis for jQuery. In *Proceedings of the 29th International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA'14)*. 17–31.

[5] R. Baldoni, E. Coppa, D. Cono D'Elia, C. Demetrescu, and I. Finocchi. 2016. A Survey of Symbolic Execution Techniques. *CoRR* abs/1610.00502 (2016). http://arxiv.org/abs/1610.00502

[6] J. Berdine, C. Calcagno, and P. O'Hearn. 2005. Smallfoot: Modular Automatic Assertion Checking with Separation Logic. In *FMCO*.

[7] J. Berdine, C. Calcagno, and P. O'Hearn. 2005. Symbolic Execution with Separation Logic. In *APLAS*.

[8] T. M. Bierman, M. Abadi, and M. Torgersen. 2014. Understanding TypeScript. In *Proceedings of the 28th European Conference on Object-Oriented Programming (ECOOP'14) (Lecture Notes in Computer Science)*. Springer, 257–281.

[9] M. Botinčan, M. Parkinson, and W. Schulte. 2009. Separation Logic Verification of C Programs with an SMT Solver. *Electron. Notes Theor. Comput. Sci.* 254 (Oct. 2009), 5–23.

[10] C. Boyapati, S. Khurshid, and D. Marinov. 2002. Korat: Automated Testing Based on Java Predicates. *SIGSOFT Softw. Eng. Notes* 27, 4 (July 2002).

[11] C. Boyapati, S. Khurshid, and D. Marinov. 2002. Korat: automated testing based on Java predicates. In *Proceedings of the International Symposium on Software Testing and Analysis, ISSTA 2002, Roma, Italy, July 22-24, 2002*. 123–133.

[12] C. Calcagno and D. Distefano. 2011. Infer: An Automatic Program Verifier for Memory Safety of C Programs. In *NASA Formal Methods - Third International Symposium, NFM 2011, Pasadena, CA, USA, April 18-20, 2011. Proceedings*. 459–465.

[13] K. Claessen, J. Duregård, and M. H. Palka. 2015. Generating constrained random data with uniform distribution. *J. Funct. Program.* 25 (2015).

[14] K. Claessen and J. Hughes. 2000. QuickCheck: a lightweight tool for random testing of Haskell programs. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP '00), Montreal, Canada, September 18-21, 2000*. 268–279.

[15] E. M. Clarke. 2008. Model Checking – My 27-Year Quest to Overcome the State Explosion Problem. In *Logic for Programming, Artificial Intelligence, and Reasoning*, I. Cervesato, H. Veith, and A. Voronkov (Eds.).

[16] A. Ştefănescu, D. Park, S. Yuwen, Y. Li, and G. Roşu. 2016. Semantics-Based Program Verifiers for All Languages. In *Proceedings of the 31th Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'16)*. ACM, 74–91. https://doi.org/10.1145/2983990.2984027

[17] J. Dolby, M. Vaziri, and F. Tip. 2007. Finding bugs efficiently with a SAT solver. In *Proceedings of the 6th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2007, Dubrovnik, Croatia, September 3-7, 2007*. 195–204.

[18] ECMA TC39. 2011. *The 5th edition of the ECMAScript Language Specification*. Technical Report. ECMA.

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

[20] A. Feldthaus and A. Møller. 2014. Checking Correctness of TypeScript Interfaces for JavaScript Libraries. In *Proceedings of the 29th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA*.

[21] A. Feldthaus, M. Schäfer, M. Sridharan, J. Dolby, and F. Tip. 2013. Efficient construction of approximate call graphs for JavaScript IDE services. In *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 752–761.

[22] J. Fragoso Santos, P. Maksimović, D. Naudžiūnienė, T. Wood, and P. Gardner. 2018. JaVerT: JavaScript Verification Toolchain. *PACMPL* 2, POPL (2018), 50:1–50:33. https://doi.org/10.1145/3158138

[23] P. Gardner, S. Maffeis, and G. 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.

[24] B. Jacobs, J. Smans, P. Philippaerts, F. Vogels, W. Penninckx, and F. Piessens. 2011. VeriFast: A powerful, sound, predictable, fast verifier for C and Java. In *NASA Formal Methods*. Springer, 41–55.

[25] D. Jang and K.-M. Choe. 2009. Points-to analysis for JavaScript. In *Proceedings of the 2009 ACM symposium on Applied Computing*. ACM, 1930–1937.

[26] S. Holm Jensen, A. Møller, and P. Thiemann. 2009. Type Analysis for JavaScript. In *Proceedings of the 16th International Static Analysis Symposium (SAS) (Lecture Notes in Computer Science)*, Vol. 5673. Springer, 238–255.

[27] J. Jones. 2016. Priority Queue Data Structure. https://github.com/jasonsjones/queue-pri.

[28] V. Kashyap, K. Dewey, E. A. Kuefner, J. Wagner, K. Gibbons, J. Sarracino, B. Wiedermann, and B. Hardekopf. 2014. JSAI: a static analysis platform for JavaScript. In *FSE*. 121–132.

[29] G. T. Leavens, A. L. Baker, and C. Ruby. 2006. Preliminary Design of JML: A Behavioural Interface Specification Language for Java. *SIGSOFT Softw. Eng. Notes* 31, 3 (2006).

[30] G. Li, E. Andreasen, and I. 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.

[31] Microsoft. 2014. *TypeScript language specification*. Technical Report. Microsoft.

[32] A. Milicevic, S. Misailovic, D. Marinov, and S. Khurshid. 2007. Korat: A Tool for Generating Structurally Complex Test Inputs. In *29th International Conference on Software Engineering (ICSE 2007), Minneapolis, MN, USA, May 20-26, 2007*. 771–774.

[33] H. H. Nguyen, V. Kuncak, and W.-N. Chin. 2008. Runtime Checking for Separation Logic. In *Proceedings of the 9th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI'08)*.

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

[35] Changhee Park and Sukyoung Ryu. 2015. Scalable and Precise Static Analysis of JavaScript Applications via Loop-Sensitivity. In *ECOOP*. 735–756.

[36] D. Park, A. Stefănescu, and G. Roşu. 2015. KJS: A Complete Formal Semantics of JavaScript. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2015)*. ACM, New York, NY, USA, 346–356. https://doi.org/10.1145/2737924.2737991

[37] Racket. 2017. The Racket Programming Language. racket-lang.org.

[38] A. Rastogi, N. Swamy, C. Fournet, G. Bierman, and P. Vekris. 2015. Safe & Efficient Gradual Typing for TypeScript. In *Proceedings of the 42nd ACM Symposium on Principles of Programming Languages*. ACM Press.

[39] John C. Reynolds. 2002. Separation Logic: A Logic for Shared Mutable Data Structures. In *LICS*.

[40] C. Runciman, M. Naylor, and F. Lindblad. 2008. Smallcheck and lazy smallcheck: automatic exhaustive testing for small values. In *Proceedings of the 1st ACM SIGPLAN Symposium on Haskell, Haskell 2008, Victoria, BC, Canada, 25 September 2008*. 37–48.

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

[42] P. Saxena, D. Akhawe, S. Hanna, F. Mao, S. McCamant, and D. 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.

[43] E. L. Seidel, N. Vazou, and R. Jhala. 2015. Type Targeted Testing. In *Programming Languages and Systems - 24th European Symposium on Programming, ESOP 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings*. 812–836.

[44] K. Sen, G. C. Necula, L. Gong, and W. Choi. 2015. MultiSE: multi-path symbolic execution using value summaries. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015, Bergamo, Italy, August 30 - September 4, 2015*. 842–853.

[45] M. Sridharan, J. Dolby, S. Chandra, M. Schäfer, and F. Tip. 2012. Correlation Tracking for Points-To Analysis of JavaScript. In *ECOOP*. 435–458.

[46] N. Swamy, J. Weinberger, C. Schlesinger, J. Chen, and B. Livshits. 2013. Verifying Higher-order Programs with the Dijkstra Monad. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '13)*. ACM, New York, NY, USA, 387–398. https://doi.org/10.1145/2491956.2491978

[47] P. Thiemann. 2005. Towards a Type System for Analysing JavaScript Programs. In *Proceedings of the 14th European Symposium on Programming Languages and Systems (Lecture Notes in Computer Science)*. Springer, 408–422.

[48] E. Torlak and R. 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.

[49] E. Torlak and R. 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.

[50] D. van Horn and M. Might. 2010. Abstracting abstract machines. In *Proceeding of the 15th ACM SIGPLAN international conference on Functional programming, ICFP 2010, Baltimore, Maryland, USA, September 27-29, 2010*. 51–62.

[51] W3Techs: Web Technology Surveys. 2017. Usage of JavaScript for websites. https://w3techs.com/technologies/details/cp-javascript/all/all.

[52] E. Wittern, A. T. T. Ying, Y. Zheng, J. Dolby, and J. 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.