JaVerT 2.0: Compositional Symbolic Execution for JavaScript

JOSÉ FRAGOSO SANTOS, Imperial College London, UK
PETAR MAKSIMOVIĆ, Imperial College London, UK and Mathematical Institute SASA, Serbia
GABRIELA SAMPAIO, Imperial College London, UK
PHILIPPA GARDNER, Imperial College London, UK

We propose a novel, unified approach to the development of compositional symbolic execution tools, bridging the gap between classical symbolic execution and compositional program reasoning based on separation logic. Using this approach, we build JaVerT 2.0, a symbolic analysis tool for JavaScript that follows the language semantics without simplifications. JaVerT 2.0 supports whole-program symbolic testing, verification, and, for the first time, automatic compositional testing based on bi-abduction. The meta-theory underpinning JaVerT 2.0 is developed modularly, streamlining the proofs and informing the implementation. Our explicit treatment of symbolic execution errors allows us to give meaningful feedback to the developer during whole-program symbolic testing and guides the inference of resource of the bi-abductive execution. We evaluate the performance of JaVerT 2.0 on a number of JavaScript data-structure libraries, demonstrating: the scalability of our whole-program symbolic testing; an improvement over the state-of-the-art in JavaScript verification; and the feasibility of automatic compositional testing for JavaScript.

CCS Concepts: • **Theory of computation → Separation logic**; **Program specifications**; **Program verification**; Abstraction; Pre- and post-conditions; • **Software and its engineering → Correctness**; *Formal methods*; *Compilers*; Automated static analysis; Formal language definitions;

Additional Key Words and Phrases: symbolic execution, compositionality, bi-abduction, dynamic languages

ACM Reference Format:
José Fragoso Santos, Petar Maksimović, Gabriela Sampaio, and Philippa Gardner. 2019. JaVerT 2.0: Compositional Symbolic Execution for JavaScript. *Proc. ACM Program. Lang.* 3, POPL, Article 66 (January 2019), 31 pages. https://doi.org/10.1145/3290379

1 INTRODUCTION

Symbolic execution is a program analysis technique that systematically explores multiple program paths by running a given program using symbolic values. It has successfully been used for both program testing (e.g. [Cadar et al. 2008a,b; Godefroid 2007; Godefroid et al. 2005, 2008, 2010]) and bounded verification (e.g. [Anand et al. 2007, 2009]). Still, it remains, to this day, mainly focused on whole-program analysis. This is particularly evident in the context of dynamic languages: for example, all major symbolic execution tools for JavaScript (JS) [Li et al. 2014; Saxena et al. 2010; Sen et al. 2015; Wittern et al. 2017] require the entire program in order to function correctly. This approach, however, is not aligned with the main use cases of JS: client-side Web applications often execute multiple scripts coming from different origins in the context of the same Web page; and server-side Node.js applications commonly import external modules.

Authors' addresses: José Fragoso Santos, Imperial College London, UK, jose.fragoso.santos@imperial.ac.uk; Petar Maksimović, Imperial College London, UK, pmaksimo@ic.ac.uk, Mathematical Institute SASA, Serbia; Gabriela Sampaio, Imperial College London, UK, g.sampaio17@imperial.ac.uk; Philippa Gardner, Imperial College London, UK, pg@doc.ic.ac.uk.

2475-1421/2019/1-ART66
https://doi.org/10.1145/3290379

We believe that an analysis that can reason about JS programs as they are written in practice needs to be *compositional*, in that it needs to be *local*, working on part of a program operating on part of its state, and *incremental*, generating function summaries that can later be used in the analysis of other functions. Program analysis based on separation logic (SL) [Ishtiaq and O'Hearn 2001; Reynolds 2002] is compositional [Harman and O'Hearn 2018; O'Hearn 2018] and has been successfully applied to large real-world codebases. A prime example of this is Infer [Calcagno et al. 2015], a fully automatic compositional tool aimed at lightweight bug-finding for static languages (C, C++, Java, Objective C), which is part of the code review pipeline at Facebook.

We propose a novel, unified approach to the development of compositional symbolic execution tools, bridging the gap between symbolic execution and compositional program reasoning based on SL. In doing so, we bring benefits to both worlds: symbolic execution tools gain access to succinct summaries in the form of SL specifications, whereas SL proof systems become tightly linked to efficient implementations based on symbolic execution. Using this approach, we build JaVerT 2.0, a new verification and testing framework for JS [ECMA TC39 2011] that follows the language semantics without simplifications. JaVerT 2.0 is underpinned by a compositional symbolic execution tool for JSIL, our intermediate representation for JS [Fragoso Santos et al. 2018a,b]. JSIL comes with a trusted compiler and, importantly, has the same memory model as JS by design, meaning that we can easily lift the results of analyses on compiled JSIL code back to the original JS code.

JaVerT 2.0 supports: *whole-program symbolic testing*,[1] which is two orders of magnitude faster than our previous symbolic execution tool, Cosette [Fragoso Santos et al. 2018a]; *verification*, which significantly improves on our previous semi-automatic verification tool, JaVerT [Fragoso Santos et al. 2018b]; and, for the first time, *automatic compositional testing* based on bi-abduction. We evaluate these three types of analysis, focussing on a number of simple data-structure libraries. Our results demonstrate the scalability of our whole-program symbolic testing, an improvement over the state-of-the-art in JavaScript verification, and the feasibility of automatic compositional testing for JavaScript, minimising the annotation burden of the developer. Furthermore, we apply whole-program symbolic testing to the real-world Buckets.js [Santos 2016] data structure library, which has over 65K downloads on npm [npm, Inc. 2018]. We reproduce previously known bugs [Fragoso Santos et al. 2018a], but also discover a new one. The times that we obtain are competitive, which indicates that our analysis can scale to much larger codebases.

1.1 Overview

Syntax-directed program analyses tend to follow the semantics of the targeted languages, often resulting in overly verbose, repetitive formalisms and implementations with substantial code duplication. We propose a new approach for designing compositional symbolic execution tools that factors out the overlap between the language semantics and the analysis, leading to streamlined formalisms that are strongly connected to modular implementations with little code duplication. We believe that this approach is language-independent; in this paper, we apply it to JSIL.

Our key insight consists of splitting the JSIL semantics into two components: a *Semantics Module* and a *state instantiation*. Similar decompositions have arisen in the context of abstract interpretation [Blazy et al. 2016; Darais et al. 2015; Horn and Might 2010], but we believe we are the first to use it in the design of a symbolic execution tool.

■ **The Semantics Module.** The Semantics Module, described in detail in §3, is the bedrock for both the formal development and the implementation of JaVerT 2.0. It describes the behaviour of JSIL

[1]We use the term *symbolic testing* to mean 'static symbolic execution with boundedness guarantees', in the style of Khurshid et al. [2003] and Torlak and Bodík [2014]. The analysis explores all paths and unrolls loops up to a given bound. We discuss this further and compare with related work in detail in §7.

commands in terms of a *state signature*: that is, a set of general state functions, reminiscent of *local actions* in SL [Calcagno et al. 2007; Dinsdale-Young et al. 2013], which capture the fundamental ways in which JSIL programs interact with JSIL states (for example: evaluating an expression; allocating a new object; or retrieving the value of an object property). This general state signature can then be instantiated to obtain a specific JSIL semantics. In JaVerT 2.0, we provide three state instantiations for the Semantics Module: *concrete*, *instrumented*, and *symbolic*, respectively obtaining the concrete, instrumented, and symbolic semantics of JSIL.

One of the main novelties of our proposed architecture is its unique emphasis on *error reporting*. The Semantics Module includes a general error reporting mechanism that accurately describes the causes of failure whenever an error occurs. Among the supported error types, *specification errors* play a crucial role in the design of the system. They occur when the instantiated JSIL semantics does not have enough information to execute the command at hand: for example, during a property lookup, the inspected object property might be missing from the state, in which case we do not know whether or not it exists. Specification errors carry important information that describes how to trigger the error, and also information on how to correct it. This information is essential for our analysis, as it allows us to provide meaningful error messages during whole-program symbolic testing, but also guides the inference of resource of the bi-abductive execution.

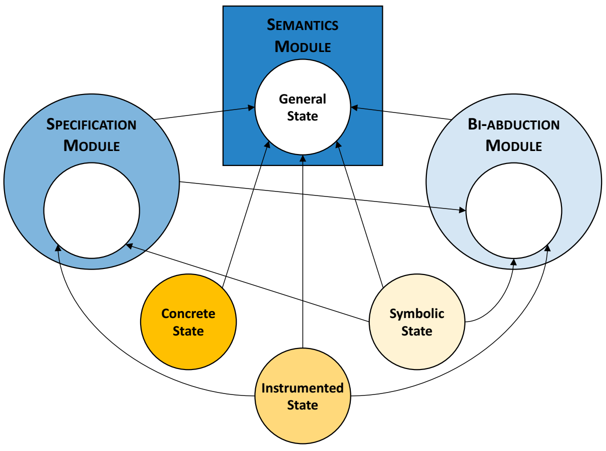


Fig. 1. Unified Symbolic Analysis for JSIL

● **The Concrete Semantics.** The concrete semantics allows us to run JSIL programs concretely. This is essential for ensuring that the Semantics Module captures the intended behaviour of the language. It also allows us to test our infrastructure against the ECMAScript official test suite, Test262 [ECMA TC39 2017], by first compiling it to JSIL and then executing it concretely. In this way, we establish trust in the compilation.

● **The Instrumented Semantics.** The frame property [Ishtiaq and O'Hearn 2001; Reynolds 2002] is essential for local reasoning about programs that alter the heap state. Intuitively, the frame property means that the output of a program does not change when the state in which it is run successfully is extended. The JSIL semantics, however, just like JavaScript semantics, does not observe the frame property: one can introduce bugs into a JSIL/JavaScript program by extending the state in which the program was run. Our solution is to design an instrumented state instantiation, and corresponding instrumented semantics [Fragoso Santos et al. 2018a], that exhibits the frame property by explicitly keeping track of object properties that we know are not present. By having the instrumented semantics as a proper interim stage between the concrete semantics and the symbolic semantics, we obtain more modular reasoning and substantially simpler proofs than previous approaches based on weak locality [Gardner et al. 2012, 2008].

○ **The Symbolic Semantics.** The symbolic semantics represents the core of our compositional symbolic execution tool. It is obtained by lifting the instrumented state instantiation to the symbolic level, following a standard approach [Torlak and Bodík 2013, 2014], and plugging the lifted state instantiation into the Semantics Module. Unlike the majority of the existing bug-finding symbolic

Table 1. Use cases: combining modules and state instantiations. We label the implemented combinations with † and those that do not yield useful results with –.

	Module			
State Implementation	**Semantics**	**Specification**	**Bi-Abduction**	**Specification + Bi-Abduction**
● **Concrete**	Concrete Interpreter†	–	–	–
● **Instrumented**	Instrumented Interpreter	Executable Specifications	–	–
○ **Symbolic**	Whole-Program Symbolic Testing†	Verification†	Automatic Local Testing	Automatic Compositional Testing†

execution tools for JavaScript [Li et al. 2014; Saxena et al. 2010; Wittern et al. 2017] which target specific bug patterns and are not rigorously formalised, the symbolic semantics underpinning JaVerT 2.0 is general, fully formalised, and proven sound.

● **The Specification Module.** The Specification Module, described in detail in §4, establishes the connection between the JSIL SL assertion language and JSIL states, allowing us to use SL specifications as procedure summaries during instrumented and symbolic execution. The Specification Module extends a given state signature with a built-in mechanism for executing procedure calls abstractly, using SL specifications. By plugging the instrumented state into the Specification Module and the resulting state into the Semantics Module, we obtain an instrumented semantics that supports executable specifications, meaning that SL specifications can be used to jump over procedure calls. If, instead, we plug the symbolic state into the Specification Module and the resulting state into the Semantics Module, we arrive at *verification* for JSIL, where one can write inductive predicates and SL specifications to describe the behaviour of their programs and obtain full functional correctness guarantees.

○ **The Bi-Abduction Module.** To support automatic compositional testing, we extend the JSIL symbolic semantics with a bi-abductive mechanism [Calcagno et al. 2011] for automatically inferring the missing resource of specification errors. Instead of creating the bi-abductive semantics from scratch, we design the Bi-Abduction Module, described in detail in §5, which extends a given state signature with a built-in mechanism for on-the-fly correction of specification errors during execution. By plugging the symbolic state into the Bi-Abduction Module and the resulting state into the Semantics Module, we obtain *automatic local testing* for JSIL. On the other hand, by plugging the symbolic state into the Specification Module, the resulting state into the Bi-Abduction Module, and then that state into the Semantics Module, we obtain *automatic compositional testing* in the style of Infer [Calcagno et al. 2015], where one can get, *without providing any annotations*, specifications that describe the behaviour of their functions up to a bound, and where specifications of previously analysed functions can be re-used for the analysis of functions that call them.

Summary. In Table 1, we summarise the different ways in which we can combine the proposed modules and state instantiations to obtain different types of analysis for JavaScript/JSIL. For a static language, such as Java, the corresponding table would only contain the concrete and the symbolic state instantiations, and the analyses at the instrumented level would lift to the concrete level.

2 USING JAVERT 2.0

JaVerT 2.0 is a tool for JavaScript developers, designed to assist them in the testing and verification of their programs. It is not meant for analysing JavaScript code in the wild. We demonstrate how JaVerT 2.0 can be used by developers for: **(1)** scalable whole-program symbolic testing; **(2)** semi-automatic verification; and **(3)** fully automatic compositional testing.

```
1  function evalExpr (store, e) {
2    if (typeof e !== "object") throw new Error ("E:Type");
3    switch (e.type) {
4      case "lit"   : return e.val
5      case "var"   : return store.get(e.name)
6      case "unop"  :
7        var arg_v = evalExpr(store, e.arg);
8        return evalUnop (e.op, arg_v)
9      case "binop" :
10       var left_v  = evalExpr(store, e.left);
11       var right_v = evalExpr(store, e.right);
12       return evalBinop (e.op, left_v, right_v)
13     default : throw new Error("Expr") }
14 }
15
16 function evalUnop (op, v) {
17   switch (op) {
18     case "-"   : return -v
19     case "not" : return !v
20     case "abs" : return v < 0 ? -v : v
21     default    : throw new Error ("UnOp") }
22 }
23
24 function evalBinop (op, v1, v2) {
25   switch (op) {
26     case "+"   : return v1 + v2
27     case "-"   : return v1 - v2
28     case "or"  : return v1 || v2
29     case "and" : return v1 && v2
30     default    : throw new Error("BinOp") }
31 }
```

Symbolic test 1:

```
1 var x = symb();
2 assume(typeof x !== "object");
3 try { evalExpr(store, x) } catch (e) {
4   assert(e.message === "E:Type")
5 }
```

Symbolic test 2:

```
1 var x = symb();
2 assume(typeof x === "object");
3 try { evalExpr(store, x) } catch (e) {
4   var msg = e.message;
5   assert (msg === "Expr" || msg === "UnOp" ||
6           msg === "BinOp");
7 }
```

Symbolic test 3:

```
1  var n = symb_number(), op = symb_string();
2  var lit = { type: "lit", val: n };
3  var e   = { type: "unop", op: op; arg: lit };
4  assume (op !== "not");
5  try {
6    var ret = evalExpr(store, e);
7    var abs_ret = n < 0 ? -n : n;
8    assert (((op === "-")   && (ret === -n)) ||
9            ((op === "abs") && (ret === abs_ret)));
10 } catch (e) {
11   assert(e.message === "UnOp")
12 }
```

Fig. 2. Expression Evaluator implementation (left); symbolic tests (right)

Our running example is a JS implementation of an *expression evaluator*, given in Figure 2 (left). It contains three functions: `evalExpr`, for evaluating a given expression under a given store; `evalUnop`, for applying a given unary operator to a given value; and `evalBinop`, for applying a given binary operator to two given values. Expressions are evaluated with respect to a store, which is assumed to be a key-value map exposing a method `get` for recovering the value associated with a given key (we re-use the key-value map implementation of Fragoso Santos et al. [2018b]). For space reasons, we omit the store initialisation in the symbolic tests of Fig. 2 (right). Expressions are represented in memory as AST objects. For instance, the expression `x + 1` corresponds to the object: { `type`: `"binop"`, `op`: `"+"`, `left`: { `type`: `"var"`, `name`: `"x"`}, `right`: { `type`: `"lit"`, `val`: `1`} }.

Whole-Program Symbolic Testing. Developers commonly write unit tests to check that, given some concrete inputs, their code produces the desired outputs. With JaVerT 2.0, developers can write unit tests with *symbolic* inputs/outputs and use simple assertions to describe the properties that the outputs must satisfy. JaVerT 2.0 allows users to symbolically execute such tests, providing *concrete counter models* in case of failure which the developer can potentially use to correct the error.

Symbolic tests are more effective than concrete ones, as one single symbolic test often covers a range of program executions that corresponds to a number of concrete unit tests. For instance, consider Symbolic Test 1 in Figure 2 (right), which tests the behaviour of `evalExpr` on all non-object inputs. We would need five concrete tests to perform the same check (corresponding to the cases when the type of the input is equal to `"undefined"`, `"boolean"`, `"number"`, `"string"`, or `"function"`).

Despite its simplicity, the code of the expression evaluator exposes a common JavaScript bug. To understand the bug, consider Symbolic Test 2, which states that, when given an input of type `"object"`, `evaExpr` does not throw a JS native error. This is tested in lines 7-9, by asserting that if an error is thrown, it must correspond to one of the errors thrown by the code itself. However, running JaVerT 2.0 on this test returns a concrete counter-example for the assertion: `e` = `null`. Indeed, if we run `evalExpr` with `e` set to `null`, the JS semantics throws a *type error*. This happens because, in

```
  predicate ExprVal(e:Obj, bnds:Set, v) :=
   e |-> { type: "lit", val: v },
   e |-> { type: "var", name: #x  } * ((#x, v) in bnds),
   e |-> { type: "unop", op: #op, arg: #arg } * ExprVal(#arg, bnds, #v) * UnOp(#op, #v, v),
   e |-> { type: "binop", op: #op, left: #left, right: #right }  * ExprVal(#left, bnds, #l)
                 * ExpVal(#right, bnds, #r) * BinOp(#op, #l, #r, v)

predicate UnOp(op, v1, v2) :=                     predicate BinOp(op, vl, vr, v) :=
 (op = "-") * types(v1:Num) * (v2 = -v1),          (op = "+") * types(vl,vr:Num) * (v = vl+vr),
 (op = "!") * types(v1:Bool) * (v2 = not v1), ...  (op = "-") * types(vl,vr:Num) * (v = vl-vr), ...
```

Fig. 3. Specification of the Expression Evaluator - Predicate Definitions

JavaScript, `typeof null` evaluates to `"object"`, meaning that the execution reaches line 4 of `evalExpr`, causing the program to throw an error when trying to access the property `"type"` of `null`.

Finally, Symbolic Test 3 covers three different behaviours of the `evalExpr` and `evalUnop` functions at the same time, effectively testing all possible behaviours of `evalUnop` on numeric inputs.

Writing unit tests is a mundane task that programmers do not generally do well. For instance, the real-world *Buckets.js* library [Santos 2016], on which we evaluate JaVerT 2.0 in §6, comes with a comprehensive, yet incomplete test suite that failed to detect two bugs. Symbolic tests reduce the burden on the programmer, bringing immediate value to the software development process.

Verification. We illustrate how JaVerT 2.0 can be used to specify and verify the expression evaluator. First, we design a predicate `ExprVal`(`e`, `bnds`, `v`), where `e` points to the AST of the expression we are evaluating, and `v` is the value to which the expression evaluates under the store with bindings given by `bnds` (a set of pairs of the form (`x`, `v`) where `x` is a variable name and `v` is its value). The predicate definition is given in Fig. 3. It uses two auxiliary predicates: **(1)** `UnOp`(`op`, `v1`, `v2`), stating that `v2` is the result of applying the unary operator `op` to `v1`, and **(2)** `BinOp`(`op`, `vl`, `vr`, `v`), stating that `v` is the result of applying the binary operator `op` to `vl` and `vr`. Predicate definitions are separated by a comma and all logical variables are prefixed with a # and are implicitly existentially quantified. We describe one base case and one recursive case of `ExprVal`; the remaining cases are similar.

- [Var] In the variable case, `e` points to an object with a property `type` with value `"var"` and a property `name` with value `#x` (note that `#x` denotes an existentially quantified logical variable). Finally, the definition requires that (`#x`, `v`) is contained in the set of bindings, `bnds`.
- [Binary Operator] In the binary operator case, `e` points to an object with properties `type`, `op`, `left`, and `right`, with values `"binop"`, `#op`, `#left`, and `#right`. The definition further requires that both `ExprVal`(`#left`, `bnds`, `#l`) and `ExprVal`(`#right`, `bnds`, `#r`) hold, meaning that the result of evaluating the AST pointed to by, respectively, `#left` and `#right`, in a store with contents `bnds` is equal to, respectively, `#l` and `#r`. Finally, the definition requires that `BinOp`(`#op`, `#l`, `#r`, `v`) holds, meaning that `v` corresponds to the application of the binary operator `#op` to `#l` and `#r`.

Having designed these predicates, we can now specify (and automatically verify) the functions `evalExpr`, `evalUnop`, and `evalBinop`. JaVerT 2.0 supports two types of specifications (specs): *normal specs*, for cases in which the function terminates normally; and *error specs*, for cases in which a it terminates by throwing an error. In Figure 4 (left), we give all of the normal specs of `evalExpr`, `evalUnop`, and `evalBinop`, as well as one error spec of `evalExpr`. We note that logical variables that appear in the pre-condition maintain the same value in the post-condition. Also, normal specs refer to the return value in the post-condition via the special variable `ret`, whereas error specs use the special variable `err`. Despite the complexity of the JS semantics, these specs are fairly simple. Below, we explain the two given specs of `evalExpr`.

```
{ ExprVal(e, #bnds, #v) *
  KVMap(store, #bnds) *
  ScopeEvalExpr() }
evalExpr(store, e)
{ PRE * (ret = #v) }

{ typeof(e) != Obj *
  ScopeEvalExpr() }
evalExpr(store, e)
{ PRE *
  ErrorObject(err, "E:Type") }

{ UnOp(op, l, #v) }
evalUnop (op, l)
{ UnOp(op, l, #v) *
  (ret = #v) }

{ BinOp(op, l, r, #v) }
evalBinop (op, l, r)
{ BinOp(op, l, r, #v) *
  (ret = #v) }
```

```
{ PRE }
2.  if (typeof e !== "object")
{ PRE }
{ e |-> { type:"unop", op:#op, arg:#arg } *
  ExprVal(#arg, #bnds, #va) * UnOp(#op, #va, #v) *
  KVMap(store, #bnds) * ScopeEvalExpr() }
3.  switch (e.type)
{ e |-> { type:"unop", op:#op, arg:#arg } *
  ExprVal(#arg, #bnds, #va) * UnOp(#op, #va, #v) *
  KVMap(store, #bnds) * ScopeEvalExpr() }
7.  var arg_v = evalExpr(store, e.arg)
{ e |-> { type:"unop", op:#op, arg:#arg } *
  ExprVal(#arg, #bnds, #va) * UnOp(#op, #va, #v) *
  KVMap(store, #bnds) * ScopeEvalExpr() * Scope(arg_v: #va) }
8.  return evalUnop (e.op, arg_v)
{ e |-> { type:"unop", op:#op, arg:#arg } * UnOp(#op, #va, #v) *
  ExprVal(#arg, #bnds, #va) * KVMap(store, #bnds) *
  ScopeEvalExpr() * Scope(arg_v: #va) * ret = #v }
{ PRE * ret = #v }
```

Fig. 4. Specification of the Expression Evaluator - Function Specs

- [Normal Return] The pre-condition states that e points to the AST of an expression that evaluates to #v in a store with contents #bnds and that store points to a key-value map with contents #bnds. The post-condition states that the function maintains the pre-condition and returns #v.
- [Error Return] The pre-condition states that e is not an object, whereas the post-condition states that the functions maintains the pre-condition and throws an error with message "Expr".

Both specs include in the pre-condition a predicate assertion ScopeEvalExpr() which describes all the resources that function evalExpr can access via its scope chain and which it may need during its execution. More concretely, those are the function objects corresponding to the three interpreter functions and the JS Error constructor. This predicate may be different for every function literal existing in the code. Unlike the original JaVerT tool [Fragoso Santos et al. 2018b], JaVerT 2.0 can generate these scope predicates automatically, relieving users of the need to describe the contents of the scope chains of the functions that they want to specify/verify.

Another important improvement of JaVerT 2.0 over JaVerT is that JaVerT 2.0 supports automatic *unfold/fold* reasoning [Nguyen et al. 2007] over user-defined inductive predicates. Hence, while JaVerT 2.0 requires no additional annotations to verify the specs given in Figure 4, JaVerT would need an unfold annotation before line 4 of the example and a fold annotation before every return/throw statement. This greatly reduces the annotation burden on the developer. To better understand the unfold/fold mechanism, let us consider the (partial) proof trace given in Figure 4 (right), which starts from the pre-condition of the first spec of evalExpr, shown in the left column. For clarity, we highlight in red the resources required for executing every JS statement.

- To symbolically execute line 2, JaVerT 2.0 does not need to unfold ExprVal(e, #bnds, #v), as the type of e is exposed in the predicate definition (cf. Figure 3).
- In contrast, to symbolically execute line 3, JaVerT 2.0 needs to access the property type of the object pointed to by e, folded inside the ExprVal(e, #bnds, #v) predicate. JaVerT 2.0 detects that there is a predicate with e as an argument and unfolds it, obtaining four possible symbolic states—one per definition of ExprVal. Here, we describe the proof trace corresponding to the *unop* case, where the symbolic execution jumps to line 7 after evaluating the guard of the switch.

- In line 7, instead of stepping inside the code of `evalExpr`, JaVerT 2.0 uses its specification to symbolically execute the recursive call. The return value is then assigned to the JS variable `arg_v`, which we capture via the assertion `Scope(arg_v: #va)`. Note that, in JS, the scope is emulated in the heap, so this assertion is not equivalent to `arg_v = #va` (cf. [Fragoso Santos et al. 2018b]).
- Finally, in line 8, JaVerT 2.0 uses the spec of `evalUnop` to symbolically execute the function call, concluding that it returns the value denoted by `#v`, which, incidentally, is also the return value of `evalExpr`. When `evalExpr` returns, JaVerT 2.0 needs to check that the current symbolic state entails its post-condition. As part of solving this entailment, JaVerT 2.0 folds back the original `ExprVal(e, #bnds, #v)` predicate assertion, obtaining the exact post-condition.

Automatic Compositional Testing. JaVerT 2.0 is the first tool that supports automatic compositional testing for JS code. It receives as input a JS program and generates two sets of specs for each function in the program: **(1)** *success specs*, describing behaviours that do not result in a JS native error (e.g., `TypeError` or `ReferenceError`), and **(2)** *bug specs*, describing the behaviours for which a JS native error is thrown. As this approach is compositional, some of the reported bug specs do not happen in practice, as their pre-conditions are never realised by design. It is the job of the programmer to identify which bug specs can actually happen and correct the program accordingly. This is feasible because JaVerT 2.0 specs use abstractions that JavaScript developers can understand, and hide almost all of the internals of the language. Below, we show two automatically generated specs of `evalExpr`:

```
  { e = null * . . . }
  evalExpr(store, e)
{ PRE * TypeError(err) }
```

```
{ e |-> { type: "unop", op: "-", arg: #a } *
     #a |-> { type: "lit", val: #n } *
          types(#n:Num) * . . .              }
          evalExpr(store, e)
        { PRE * (ret = -#n) }
```

The bug spec on the left captures the expression evaluator bug, stating that when `e` equals `null`, the program throws a `TypeError`. The right spec is a success spec, stating that when `e` points to the AST of the expression −`#n`, the result of running `evalExpr` on `e` is, in fact, the value −`#n`. In both cases, the behaviour of `evalExpr` is independent of the store, so it need not be featured in the spec.

As JaVerT 2.0 does not synthesise loop invariants or infer abstractions for recursive functions, the automatically generated specs describe program behaviour *up to a given bound*. These specs can be viewed as symbolic unit tests whose purpose is to find JS native errors. Just as for symbolic tests, the pre-condition of a bug spec can be concretised, revealing a concrete execution triggering the corresponding JS native error. On the other hand, success specs can be turned into actual symbolic tests [Fragoso Santos et al. 2018a], meaning that, for example, after fixing the bug, we could collect all the success specs to produce a comprehensive symbolic test suite for our expression evaluator.

The automatic compositional testing of JaVerT 2.0 has its limitations. The JS semantics has significant internal branching (for example, due to type coercions), which can cause a search space explosion during spec generation. To help guide the tool, users can explicitly give hints about the intended use-cases of the functions to the bi-abduction engine. These hints are extremely simple to write, are much less verbose than full-blown specs or handwritten symbolic tests, and can efficiently guide the bi-abduction, as illustrated in §6, as well as serve as documentation for the code. For example, if the parameter `op` of the procedure `evalUnop(op, v)` should always be a string by design, we could give the hint `assume(typeof op === "string")` and cut the search space accordingly.

3 THE SEMANTICS MODULE

We present the Semantics Module of our unified analysis framework for JSIL. We give the syntax of JSIL and comment on its expressivity (§3.1). We introduce the interface of the Semantics Module (§3.2) and the general JSIL semantics (§3.3). Following our previous approaches [Fragoso Santos et al.

2018a,b], we instantiate the Semantics Module to obtain the concrete and symbolic JSIL semantics (§3.4–§3.5). We highlight our novel treatment of errors, essential for meaningful error reporting for whole-program symbolic testing, as well as for the inference of resource of the bi-abductive semantics, shown in §3.3.1 and §3.5.2.

3.1 JSIL Syntax

JSIL is a simple goto language with top-level procedures and commands that operate on object heaps.

The Syntax of JSIL

$$
\begin{aligned}
&\lambda \in \mathcal{L}it ::= n \in \mathcal{N} \mid b \in \mathcal{B} \mid s \in \mathcal{S} \mid \mathsf{undefined} \mid \mathsf{null} \mid \qquad e \in \mathcal{E} ::= \lambda \mid x \in \mathcal{X} \mid \hat{x} \in \hat{\mathcal{X}} \mid \ominus e \mid e \oplus e \\
&\qquad\qquad \mathsf{empty} \mid l \in \mathcal{L} \mid \tau \in \mathcal{T} \mid f \in \mathcal{F}id \mid \overline{\lambda} \mid \{\overline{\lambda}\} \\
&bc \in \mathcal{B}cmd ::= \mathsf{skip} \mid x := e \mid x := \mathsf{new}\,(e) \mid x := [e, e] \mid [e, e] := e \mid \mathsf{delete}\,(e, e) \mid x := \mathsf{hasProp}\,(e, e) \mid \\
&\qquad\qquad x := \mathsf{getProps}\,(e) \mid x := \mathsf{metaData}\,(e) \\
&c \in Cmd ::= bc \mid \mathsf{goto}\ i \mid \mathsf{goto}\ [e]\ i,\ j \mid \mathsf{assume}\,(e) \mid \mathsf{assert}\,(e) \mid x := \mathsf{arguments} \mid \mathsf{return} \mid \mathsf{throw} \mid \\
&\qquad\qquad x := e(\overline{e})\ \mathsf{with}\ j \mid x := \mathsf{extern}\ e(\overline{e})\ \mathsf{with}\ j \mid x := \mathsf{apply}\,(e, e)\ \mathsf{with}\ j \\
&\qquad\qquad proc \in \mathsf{Proc} ::= \mathsf{proc}\ f(\overline{x})\{\overline{c}\} \qquad p \in \mathcal{P} ::= \{\overline{proc}\}
\end{aligned}
$$

JSIL *literals* include numbers, booleans, strings, the special values undefined, null, and empty, object locations, types, procedure identifiers, and lists and sets of values. JSIL *expressions*, $e \in \mathcal{E}xp$, include literals, program variables x, symbolic variables $\hat{x}$, and various unary and binary operators.

JSIL *basic commands* are used for the manipulation of extensible objects and have no impact on the control flow of the program. They include: the skip command; variable assignment; object creation; property access, assignment, deletion, membership, and collection; and object metadata collection, introduced in this paper and described shortly.

JSIL *commands* include basic commands, conditional and unconditional gotos, procedure calls, commands for assuming and asserting facts about the execution of the program, and new commands for argument collection, external procedure calls, procedure application, and procedure termination. The goto commands are straightforward. The procedure call $x := e(\overline{e})$ with j is dynamic: the procedure identifer is obtained by evaluating the JSIL expression e. If the procedure terminates normally, control proceeds to the next command, and to the j-th command otherwise. External procedure calls can step outside the execution of a JSIL program, whereas procedure application receives the procedure identifier and a *JSIL list* containing the procedure parameters. The argument collection command returns the values of the arguments with which the current procedure was called.

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 can return either normally or in error mode, using the return and throw commands, respectively. In both cases, the value that is returned is the value of the dedicated variable `ret`. A JSIL program $p \in \mathcal{P}$ is a set of top-level procedures, and its entry point is always the special procedure main.

JSIL as an IR for JavaScript Analysis. JSIL has all of the constructs needed to precisely capture and reason about the entire ES5 standard. In particular: **(1)** the dynamic features of JS (extensible objects, dynamic property access, and dynamic procedure calls) are native in JSIL; **(2)** the intricate control flow patterns of JavaScript statements (e.g. `switch` and `try–catch`) can be expressed in the JSIL goto-based control flow; **(3)** the `eval` statement and the `Function` constructor, which require parsing of JS code, are modelled via external procedure calls; **(4)** functions that require a variable number of arguments are modelled using procedure application; **(5)** the `arguments` object is modelled with the help of the argument collection command; and **(6)** the internal properties of JS objects are modelled using object metadata, streamlining the performance of JaVerT 2.0 on compiled JS code.

3.2 The Semantics Module Interface

The Semantics Module operates on general JSIL states $\Sigma \in \mathbb{S}$ and general JSIL values $\mathsf{v}, \mathsf{p}, \mathsf{l} \in \mathbb{V}$. For clarity, we use p and l to refer to general values denoting properties and locations, respectively. Also, 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 $\underline{\mathsf{v}}$. We design the functions of the Semantics Module Interface (SMI) so that the analysis, the proofs, and the implementations are modular and streamlined, minimising redundancy wherever possible. We introduce the key SMI functions in several groups, according to their use.

First, we require general states to contain a variable store $\mathsf{P} : \mathcal{X} \rightharpoonup \mathbb{V}$, mapping program variables $x \in \mathcal{X}$ to general values. Stores have three functions associated with them:

- the **store getter** (GetStore), $\mathcal{GS}(\Sigma)$, which returns the store associated with the state Σ;
- the **store setter** (SetStore), $\mathcal{SS}(\Sigma, x, \mathsf{v})$, which returns the state obtained from Σ by updating the value of x to v in the store of Σ; and
- the **store replacer** (ReplaceStore), $\mathcal{RS}(\Sigma, \mathsf{P})$, which returns the state obtained from Σ by replacing its variable store by P.

Next, we have the SMI functions that operate only on JSIL expressions and values. These are:

- the **expression evaluator**, $\mathcal{E}v(\mathsf{P}, e)$, which returns the value of e under store P;
- **assumption**, $\mathcal{A}sm(\Sigma, e)$, which extends Σ by assuming that e evaluates to true; and
- the **satisfiability checker** (SATCheck), $\mathcal{S}at(\Sigma, e)$, which returns true if e is satisfiable in the state Σ, and false otherwise.

Finally, there are the SMI functions that describe the *local actions* of JSIL, that is, the fundamental ways through which JSIL interacts with its memory model. We have:

- the **object allocator**, $\mathcal{A}lloc(\Sigma)$, which returns a value denoting a fresh location as well as the new state that keeps track of that allocation;
- the **object property collector** (GetProperties), $\mathcal{GP}(\Sigma, e)$, which returns the set of property names associated with the object denoted by e;
- the **cell getter** (GetCell), $\mathcal{GC}(\Sigma, e_1, e_2) \rightsquigarrow \Sigma', (\mathsf{l}, \mathsf{p}, \underline{\mathsf{v}})$, which retrieves, potentially non-deterministically, the value associated with a given property of a given object: if $\mathcal{GC}(\Sigma, e_1, e_2) \rightsquigarrow \Sigma', (\mathsf{l}, \mathsf{p}, \underline{\mathsf{v}})$ holds, then: $\mathsf{l} = \mathcal{E}v(\mathcal{GS}(\Sigma), e_1)$, $\mathsf{p} = \mathcal{E}v(\mathcal{GS}(\Sigma), e_2)$, $\underline{\mathsf{v}}$ denotes the value of the property p of the object at location l, and Σ' denotes a potential re-arrangement of Σ after property inspection (discussed in more detail in §3.5);
- the **cell setter** (SetCell), $\mathcal{SC}(\Sigma, \mathsf{l}, \mathsf{p}, \underline{\mathsf{v}})$, which returns the state obtained from Σ by creating/updating the property p of the object at location l to have value $\underline{\mathsf{v}}$;
- the **metadata getter** (GetMetadata), $\mathcal{GM}(\Sigma, e)$, which returns the metadata of the object e; and
- the **metadata setter** (SetMetadata), $\mathcal{SM}(\Sigma, e, \mathsf{v})$, which sets the metadata of the object denoted by e to the value v.

3.3 General JSIL Semantics

Transitions for basic commands have the form $\langle \Sigma, bc \rangle \rightsquigarrow \mathcal{O}$, meaning that the execution of the basic command bc in the state Σ evaluates to the outcome $\mathcal{O}$. The outcome $\mathcal{O}$ can either be a general state Σ' or a JSIL error ξ. We show the non-failing transitions of the semantics in Fig. 5, and delay the presentation of errors to §3.3.1. These rules illustrate how the behaviour of JSIL basic commands can be broken down using the functions of the SMI—in particular, the local actions. For example, GetCell is used pervasively to factor out the common behaviour of almost all basic commands that interact with objects: property access, assignment, deletion, and membership.

$$\textsc{Skip}\ \ \langle \Sigma, \mathsf{skip}\ \rangle \rightsquigarrow \Sigma \qquad \textsc{Assignment}\ \ \frac{\mathsf{v} = \mathcal{E}v(\mathcal{GS}(\Sigma), e)}{\langle \Sigma, x := e \rangle \rightsquigarrow \mathcal{SS}(\Sigma, x, \mathsf{v})} \qquad \textsc{Object Creation}\ \ \frac{\mathsf{v} = \mathcal{E}v(\mathcal{GS}(\Sigma), e) \quad (\Sigma', 1) = \mathcal{A}lloc(\Sigma) \quad \Sigma'' = \mathcal{SM}(\Sigma', 1, \mathsf{v})}{\langle \Sigma, x := \mathsf{new}\,(e) \rangle \rightsquigarrow \mathcal{SS}(\Sigma'', x, 1)}$$

$$\textsc{Property Access}\ \ \frac{\mathcal{GC}(\Sigma, e_1, e_2) \rightsquigarrow \Sigma', (-, -, \mathsf{v})}{\langle \Sigma, x := [e_1, e_2] \rangle \rightsquigarrow \mathcal{SS}(\Sigma', x, \mathsf{v})} \qquad \textsc{Property Assignment}\ \ \frac{\mathsf{v} = \mathcal{E}v(\mathcal{GS}(\Sigma), e_3) \quad \mathcal{GC}(\Sigma, e_1, e_2) \rightsquigarrow \Sigma', (1, \mathsf{p}, -)}{\langle \Sigma, [e_1, e_2] := e_3 \rangle \rightsquigarrow \mathcal{SC}(\Sigma', 1, \mathsf{p}, \mathsf{v})} \qquad \textsc{Property Deletion}\ \ \frac{\mathcal{GC}(\Sigma, e_1, e_2) \rightsquigarrow \Sigma', (1, \mathsf{p}, \mathsf{v}) \quad \Sigma'' = \mathcal{SC}(\Sigma', 1, \mathsf{p}, \varnothing)}{\langle \Sigma, \mathsf{delete}\,(e_1, e_2) \rangle \rightsquigarrow \Sigma''}$$

$$\textsc{Property Membership}\ \ \frac{\mathcal{GC}(\Sigma, e_1, e_2) \rightsquigarrow \Sigma', (-, -, \underline{\mathsf{v}}) \quad \Sigma'' = \mathcal{SS}(\Sigma', x, \underline{\mathsf{v}} \neq \varnothing)}{\langle \Sigma, x := \mathsf{hasProp}\,(e_1, e_2) \rangle \rightsquigarrow \Sigma''} \qquad \textsc{Property Collection}\ \ \frac{\mathsf{v} = \mathcal{GP}(\Sigma, e) \quad \Sigma' = \mathcal{SS}(\Sigma, x, \mathsf{v})}{\langle \Sigma, x := \mathsf{getProps}\,(e) \rangle \rightsquigarrow \Sigma'} \qquad \textsc{MetaData Collection}\ \ \frac{\mathcal{GM}(\Sigma, e) = \mathsf{v}}{\langle \Sigma, x := \mathsf{metaData}\,(x, e) \rangle \rightsquigarrow \mathcal{SS}(\Sigma, x, \mathsf{v})}$$

Fig. 5. General semantics of basic commands, non-failing transitions : $\langle \Sigma, bc \rangle \rightsquigarrow \Sigma'$

$$\textsc{Basic Command}\ \ \frac{\mathsf{cmd}(i) = bc \quad \langle \Sigma, bc \rangle \rightsquigarrow \Sigma'}{\langle \Sigma, \mathsf{cs}, i \rangle^{\mathsf{C}} \rightsquigarrow \langle \Sigma', \mathsf{cs}, i{+}1 \rangle^{\mathsf{C}}} \qquad \textsc{Assume}\ \ \frac{\mathsf{cmd}(i) = \mathsf{assume}\,(e) \quad \Sigma' = \mathcal{A}sm(\Sigma, e)}{\langle \Sigma, \mathsf{cs}, i \rangle^{\mathsf{C}} \rightsquigarrow \langle \Sigma', \mathsf{cs}, i{+}1 \rangle^{\mathsf{C}}} \qquad \textsc{Assert - True}\ \ \frac{\mathsf{cmd}(i) = \mathsf{assert}\,(e) \quad \mathcal{S}at(\Sigma, \neg e) = \mathsf{false}}{\langle \Sigma, \mathsf{cs}, i \rangle^{\mathsf{C}} \rightsquigarrow \langle \Sigma, \mathsf{cs}, i{+}1 \rangle^{\mathsf{C}}} \qquad \textsc{Goto}\ \ \frac{\mathsf{cmd}(i) = \mathsf{goto}\ j}{\langle \Sigma, \mathsf{cs}, i \rangle^{\mathsf{C}} \rightsquigarrow \langle \Sigma, \mathsf{cs}, j \rangle^{\mathsf{C}}}$$

$$\textsc{Cond. Goto - True}\ \ \frac{\mathsf{cmd}(i) = \mathsf{goto}\ [e]\, j,\ k \quad \Sigma' = \mathcal{A}sm(\Sigma, e)}{\langle \Sigma, \mathsf{cs}, i \rangle^{\mathsf{C}} \rightsquigarrow \langle \Sigma', \mathsf{cs}, j \rangle^{\mathsf{C}}} \qquad \textsc{Arguments}\ \ \frac{\mathsf{cmd}(i) = x := \mathsf{arguments} \quad \mathsf{cs} = (-, \mathsf{v}, -, -, -, -) \quad \Sigma' = \mathcal{SS}(\Sigma, x, \mathsf{v})}{\langle \Sigma, \mathsf{cs}, i \rangle^{\mathsf{C}} \rightsquigarrow \langle \Sigma', \mathsf{cs}, i{+}1 \rangle^{\mathsf{C}}} \qquad \textsc{Return/Throw}\ \ \frac{\mathsf{cmd}(i) = \mathsf{return}\,/\mathsf{throw} \quad \mathsf{cs} = (-, -, \mathsf{P}', x, j, k) :: \mathsf{cs}' \quad \mathsf{P} = \mathcal{GS}(\Sigma) \quad \Sigma' = \mathcal{RS}(\Sigma, \mathsf{P}') \quad \Sigma'' = \mathcal{SS}(\Sigma', x, \mathsf{P}(\mathsf{ret}))}{\langle \Sigma, \mathsf{cs}, i \rangle^{\top} \rightsquigarrow \langle \Sigma'', \mathsf{cs}', j/k \rangle^{\top}}$$

$$\textsc{Procedure Call}\ \ \frac{\mathsf{cmd}(i) = x := e(e_i|_{i=0}^{n})\ \mathsf{with}\ j \quad \mathsf{P} = \mathcal{GS}(\Sigma) \quad \mathcal{E}v(\mathsf{P}, e) = f \quad \mathsf{args}(f) = [x_i|_{i=0}^{m}] \quad \mathsf{v}_i = \mathcal{E}v(\mathsf{P}, e_i)|_{i=0}^{n} \quad \mathsf{v}_i = \mathsf{undefined}|_{i=n+1}^{m} \quad \mathsf{P}' = [x_i \mapsto \mathsf{v}_i|_{i=0}^{m}] \quad \Sigma' = \mathcal{RS}(\Sigma, \mathsf{P}')}{\langle \Sigma, \mathsf{cs}, i \rangle^{\mathsf{C}} \rightsquigarrow \langle \Sigma', (f, \mathsf{v}_i|_{i=0}^{m}, \mathsf{P}, x, i{+}1, j) :: \mathsf{cs}, 0 \rangle^{\mathsf{C}}}$$

$$\textsc{External Procedure Call}\ \ \frac{\mathsf{cmd}(i) = x := \mathsf{extern}\ e(e_i|_{i=0}^{n})\ \mathsf{with}\ j \quad \mathsf{P} = \mathcal{GS}(\Sigma) \quad \mathcal{E}v(\mathsf{P}, e) = f \quad \mathsf{v}_i = \mathcal{E}v(\mathsf{P}, e_i)|_{i=0}^{n} \quad \Psi(f) = \phi \quad \phi(p, \mathsf{v}_i|_{i=0}^{n}) = p', \mathsf{v}, b \quad \Sigma'' = \mathcal{SS}(\Sigma', x, \mathsf{v}) \quad k = \text{if } b \text{ then } i + 1 \text{ else } j}{\Psi \vdash \langle p, \Sigma, \mathsf{cs}, i \rangle^{\mathsf{C}} \rightsquigarrow \langle p', \Sigma'', \mathsf{cs}, k \rangle^{\mathsf{C}}}$$

Fig. 6. General semantics of commands, non-failing transitions (excerpt): $\Psi \vdash \langle p, \Sigma, \mathsf{cs}, i \rangle^{\mathsf{C}} \rightsquigarrow \langle p', \Sigma', \mathsf{cs}', j \rangle^{\mu}$

For transitions of commands, we introduce *execution modes*, $\mu \in \mathcal{M}$, and *call stacks*, $\mathsf{cs} \in \mathcal{CS}$. JSIL has four execution modes: C, denoting that the execution should proceed; $\mathsf{N}(\mathsf{v})$ and $\mathsf{E}(\mathsf{v})$, denoting, respectively, that the execution terminated normally or in error mode, with return value v; and $\mathsf{F}(\xi)$, denoting that the execution has failed with an error ξ (discussed shortly). Call stacks are *non-empty* lists of tuples of the form $(f, args, \mathsf{P}, x, i, j)$, where: f is the identifier of the procedure being executed; *args* is a list containing the values of the parameters with which this procedure was called; P is the store of the caller of f; x is the variable to which the return value of f will be assigned; and i and j are, respectively, the indexes to which the control jumps when f returns normally or with an error. We define the *initial call stack*, cs_{main}, as $[(\mathsf{main}, [], \emptyset, \mathtt{ret}, 0, 0)]$, where the variable `ret` holds the output of the entire program. We model external procedure calls as semantic functions $\phi \in \Phi : \mathcal{P} \times \mathbb{V}^n \rightharpoonup \mathcal{P} \times \mathbb{V} \times \mathcal{B}$, which take as parameters the current program and a list of values, and (possibly non-deterministically) produce a new program, a return value, and a boolean indicating if the execution should continue normally or in error mode. These functions are available via an external procedure table, $\Psi : \mathcal{F}id \rightharpoonup \Phi$. Transitions for commands are of the form $\Psi \vdash \langle p, \Sigma, \mathsf{cs}, i \rangle^{\mu} \rightsquigarrow \langle p', \Sigma', \mathsf{cs}', j \rangle^{\mu'}$, meaning that, given an external procedure table Ψ and starting from a program p, state Σ and execution mode μ, the evaluation of the i-th command of the leading procedure of cs generates the program p', state Σ', call stack cs', and the next command

$$\textsc{Property Access - Error Propagation} \quad \frac{\mathcal{GC}(\Sigma, e_1, e_2) \rightsquigarrow \xi}{\langle \Sigma, x := [e_1, e_2] \rangle \rightsquigarrow \xi}$$

$$\textsc{Property Access - Absent Property} \quad \frac{\mathcal{GC}(\Sigma, e_1, e_2) \rightsquigarrow \Sigma', (\mathsf{l}, \mathsf{p}, \varnothing)}{\langle \Sigma, x := [e_1, e_2] \rangle \rightsquigarrow \mathsf{RE}((\mathsf{l}, \mathsf{p}) \mapsto \varnothing)}$$

$$\textsc{Property Membership - Spec Error} \quad \frac{\mathcal{GC}(\Sigma, e_1, e_2) \rightsquigarrow \mathsf{SE}(p_f, \overline{s_c}, p_c)}{\langle \Sigma, x := \mathsf{hasProp}\,(e_1, e_2) \rangle \rightsquigarrow \mathsf{SE}^{\dagger}(p_f, \overline{s_c})p_c}$$

$$\textsc{Basic Command - Error Propagation} \quad \frac{\mathsf{cmd}(p, \mathsf{cs}, i) = bc \quad \langle \Sigma, bc \rangle \rightsquigarrow \xi}{\langle \Sigma, \mathsf{cs}, i \rangle^{\top} \rightsquigarrow \langle \Sigma, \mathsf{cs}, i \rangle^{\mathsf{F}(\xi)}}$$

$$\textsc{Procedure Call - Id Type Error} \quad \frac{\mathsf{cmd}(i) = x := e(-)\,\mathsf{with}\, - \quad \tau = \mathsf{typeOf}\,(\mathcal{Ev}(\mathcal{GS}(\Sigma), e)) \quad \tau \neq \mathtt{Fid}}{\langle \Sigma, \mathsf{cs}, i \rangle^{\top} \rightsquigarrow \langle \Sigma, \mathsf{cs}, i \rangle^{\mathsf{F}(\mathsf{TE}(e, \mathtt{Fid}, \tau))}}$$

Fig. 8. General Semantics for Basic Commands and Commands, failing transitions (excerpt)

to be evaluated is the j-th command of the leading procedure of cs', in execution mode μ'. The non-failing transitions for commands (except [Cond. Goto - False], for space reasons) are given in Fig. 6. For clarity, we keep the program and the external procedure table implicit in the rules that do not use them. Also, we write $\mathsf{cmd}(i)$ to denote the i-th command of the leading procedure of cs.

3.3.1 Treatment of Errors. The general JSIL semantics explicitly handles errors that occur during the evaluation of a program. Their syntax is shown in Fig. 7, and a selection of failing transitions for JSIL basic commands and commands is shown in Fig. 8. We describe errors using standard separation logic assertions. Pure assertions are quantifier-free first-order formulae over general JSIL values, whereas spatial assertions describe the JSIL heap: the emp assertion describes the empty heap; the cell assertion, $(\mathsf{v}_l, \mathsf{v}_p) \mapsto \underline{\mathsf{v}}$, states that the property denoted by v_p of the object denoted by v_l has the value denoted by $\underline{\mathsf{v}}$; the no-properties assertion, $\mathtt{noProps}(\mathsf{v}_l, \mathsf{v}_{ps})$, states that the object denoted by v_l has no properties other than possibly those in the set denoted by v_{ps}; and the metadata assertion, $\mathtt{MetaData}(\mathsf{v}_l, \underline{\mathsf{v}})$, states that the metadata of the object denoted by v_l is equal to the value denoted by $\underline{\mathsf{v}}$.

Pure Assertions:
$p ::= \mathsf{true} \mid \mathsf{false} \mid \mathsf{v} = \mathsf{v} \mid \mathsf{v} \leq \mathsf{v} \mid \neg p \mid p \vee p$

Spatial Assertions:
$s ::= \mathsf{emp} \mid (\mathsf{v}, \mathsf{v}) \mapsto \underline{\mathsf{v}} \mid \mathtt{noProps}(\mathsf{v}, \underline{\mathsf{v}}) \mid \mathtt{MetaData}(\mathsf{v}, \underline{\mathsf{v}})$

Errors:
$\xi \in \mathsf{Err} ::= \mathsf{TE}(\mathsf{v}, \tau_1, \tau_2) \mid \mathsf{RE}(\overline{s}) \mid \mathsf{AE}(p) \mid \mathsf{SE}(p_f, \overline{s_c}, p_c) \mid \mathsf{SE}^{\dagger}(\overline{s_c}, p_c)$

Fig. 7. JSIL Errors

We first focus on **type errors**, $\mathsf{TE}(\mathsf{v}, \tau_1, \tau_2)$, and **resource errors**, $\mathsf{RE}(\overline{s})$, which correspond to well-known errors that can occur during program execution. These errors can only be reported, but not corrected, by our analysis. A type error, $\mathsf{TE}(\mathsf{v}, \tau_1, \tau_2)$, occurs when a value v is expected to have type τ_1, but instead has type τ_2. For instance, if we call a procedure with a number instead of a procedure identifier, we will get a type error (Fig. 8, [Procedure Call - Id Type]). A resource error, $\mathsf{RE}(\overline{s})$, occurs when a command requires a spatial resource, described by the list of assertions $\overline{s}$, but we know with certainty that this resource is absent. For example, the rule [Property Access - Absent Property] in Fig. 8 states that if we try to look up a property that we know does not exist (indicated by the GetCell returning $\varnothing$), then we get the appropriate resource error.

An **assertion error**, $\mathsf{AE}(p)$, is an analysis-related error that occurs when the pure constraint p that we are attempting to assert does not hold in the current state.

As our symbolic execution is compositional, we can reason about partial programs running in partial states. This gives rise to another type of error, specific to our analysis, which is the **specification error**. Specification errors (spec errors, $\mathsf{SE}(p_f, \overline{s_c}, p_c)$), occur when we do not have information about a required spatial part of the state. They carry three pieces of information: the failing constraint, p_f, which tells us how to trigger the error; and two correctives, $\overline{s_c}$ (spatial) and p_c (pure), which together tell us how to correct the error. This information is essential for our analysis: it allows us to provide meaningful error messages during whole-program symbolic testing; and it guides the inference of resource of the bi-abductive execution. We revisit spec errors in §3.5.2.

SetCell - Positive Update
$$\frac{\sigma = (\rho, h, m) \qquad h' = h[(l, p) \mapsto v]}{\mathcal{SC}_{\text{c}}(\sigma, l, p, v) \triangleq (\rho, h', m)}$$

SetCell - Negative Update
$$\frac{\sigma = (\rho, h, m) \qquad h = h' \uplus (l, p) \mapsto -}{\mathcal{SC}_{\text{c}}(\sigma, l, p, \varnothing) \triangleq (\rho, h', m)}$$

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

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

GetProperties
$$\frac{\sigma = (\rho, h, m) \quad l = \mathcal{E}v_c(\rho, e) \qquad l \in \mathsf{dom}(h) \cup \mathsf{dom}(m) \qquad h \triangleleft l = \{ p_1, \ldots, p_m \}}{\mathcal{GP}_{\text{c}}(\sigma, e) \triangleq \{ p_1, \ldots, p_m \}}$$

SAT Check
$$\frac{b = \mathcal{E}v_{\text{c}}(\mathcal{GS}(\sigma), e)}{\mathcal{S}at_{\text{c}}(\sigma, e) \triangleq b}$$

Assume - Success
$$\frac{\mathcal{E}v_{\text{c}}(\mathcal{GS}(\sigma), e) = \mathsf{true}}{\mathcal{A}sm_{\text{c}}(\sigma, e) \triangleq \sigma}$$

Assume - Type Error
$$\frac{\mathcal{E}v_{\text{c}}(\mathcal{GS}(\sigma), e) = v \quad \tau = \mathsf{typeOf}(v) \quad \tau \neq \mathsf{Bool}}{\mathcal{A}sm_{\text{c}}(\sigma, e) \triangleq \mathsf{TE}(v, \mathsf{Bool}, \tau))}$$

Fig. 9. Concrete JSIL semantics, selected transitions

In the general semantics, spec errors are either propagated (e.g. Fig. 8, [Property Access - Error Propagation]) or are silenced, denoted by $\mathsf{SE}^{\dagger}(\overline{s_c}, p_c)$, in the case of the property assignment, membership check, and property collection (e.g. Fig. 8, [Property Assignment - Missing Error]). The need for **silent specification errors** arises due to a dissonance between our analysis, which is compositional and based on separation logic, and the semantics of JSIL. We discuss this in detail in §3.5.2, after introducing the concrete and the symbolic JSIL semantics.

We note that our modular approach to the general JSIL semantics greatly reduces the number of failing transitions for basic commands and commands. For example, if we weren't using GetCell, we would have to repeat all of its possible failing cases for every rule that uses it.

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 $f_1 \uplus f_2$.

3.4 Concrete JSIL Semantics

The instantiation of the Semantics Module to the concrete case is straightforward. Concrete JSIL values, $v \in \mathcal{V}$, correspond to JSIL literals. A concrete JSIL state, $\sigma = (\rho, h, m)$, consists of: a store ρ; partially mapping program variables to JSIL values; a heap h, partially mapping pairs of object locations and property names (strings) to JSIL values; and a metadata table m, partially mapping object locations to JSIL values. We write $\langle \sigma, bc \rangle \rightsquigarrow_{\text{c}} \sigma'$ for the concrete semantic judgement for JSIL basic commands and $\langle p, \sigma, cs, i \rangle^{\mu} \rightsquigarrow_{\text{c}} \langle p', \sigma', cs', j \rangle^{\mu'}$ for JSIL commands.

We provide a selection of definitions for the SMI functions in Fig. 9. Expression evaluation is standard (symbolic variables cannot be evaluated). Allocation returns the unchanged state, together with a fresh location obtained from an oracle. The SetCell positive update is standard; the SetCell negative update removes the given property of a given object from the heap. These two rules are never applicable at the same time, as $v \neq \varnothing$. GetCell, given an object denoted by e_1 and a property name denoted by e_2, returns the associated value if the object has the property, and $\varnothing$ otherwise. GetProperties returns the set of properties of a given object. SatCheck amounts to returning the value of the evaluated (boolean) expression. Assumption succeeds if the assumed (boolean) expression evaluates to true and does not change the state. Finally, we show an error case, in which Assumption is called with a non-boolean expression, resulting in a type error.

Instrumented JSIL Semantics. JSIL, like JavaScript, does not observe the frame property, meaning that it is possible to cause a JSIL program to behave incorrectly by *extending* the state in which it was run successfully. On the other hand, our analysis is compositional, meaning that we must reason about programs given partial state information, which is challenging for languages without

$$\textsc{Allocation}\quad \frac{\begin{array}{c}\hat{\sigma} = (\hat{\rho}, \hat{h}, \hat{d}, \hat{m}, \pi) \\ \hat{l} = \mathsf{GenSym}(\hat{\mathcal{L}}) \\ \hat{d}' = \hat{d}[\hat{l} \mapsto \{\}]\end{array}}{\mathcal{A}lloc_{\mathsf{S}}(\hat{\sigma}) \triangleq ((\hat{\rho}, \hat{h}, \hat{d}', \hat{m}, \pi), \hat{l})}$$

$$\textsc{Assumption}\quad \frac{\begin{array}{c}\hat{\sigma} = (\hat{\rho}, \hat{h}, \hat{d}, \hat{m}, \pi) \\ \hat{b} = \mathcal{E}v_{\mathsf{S}}(\hat{\rho}, e) \\ Sat_{\mathsf{S}}(\hat{\sigma}, \pi \wedge \hat{b})\end{array}}{\mathcal{A}sm_{\mathsf{S}}(\hat{\sigma}, e) = (\hat{\rho}, \hat{h}, \hat{d}, \hat{m}, \pi \wedge \hat{b})}$$

$$\textsc{SetCell}\quad \frac{\begin{array}{c}\hat{\sigma} = (\hat{\rho}, \hat{h}, \hat{d}, \hat{m}, \pi) \\ \hat{h}' = \hat{h}[(\hat{l}, \hat{p}) \mapsto \underline{\hat{v}}]\end{array}}{\mathcal{SC}_{\mathsf{S}}(\hat{\sigma}, (\hat{l}, \hat{p}), \underline{\hat{v}}) \triangleq (\hat{\rho}, \hat{h}', \hat{d}, \hat{m}, \pi)}$$

$$\textsc{GetCell - Found}\quad \frac{\begin{array}{c}\hat{\sigma} = (\hat{\rho}, \hat{h}, \hat{d}, \hat{m}, \pi) \\ \hat{l}, \hat{p} = \mathcal{E}v_{\mathsf{S}}(\hat{\rho}, e_1), \mathcal{E}v_{\mathsf{S}}(\hat{\rho}, e_2) \\ \pi \vdash \hat{p} = \hat{p}' \quad \hat{h}(\hat{l}, \hat{p}') = \underline{\hat{v}}\end{array}}{\mathcal{GC}(\hat{\sigma}, e_1, e_2) \rightsquigarrow_s (\hat{\rho}, \hat{h}, \hat{d}, \hat{m}, \pi), (\hat{l}, \hat{p}', \underline{\hat{v}})}$$

$$\textsc{GetCell - Not Found}\quad \frac{\begin{array}{c}\hat{\sigma} = (\hat{\rho}, \hat{h}, \hat{d}, \hat{m}, \pi) \\ \hat{l}, \hat{p} = \mathcal{E}v_{\mathsf{S}}(\hat{\rho}, e_1), \mathcal{E}v_{\mathsf{S}}(\hat{\rho}, e_2) \\ \pi \vdash \hat{p} \notin \hat{d}(\hat{l}) \\ \hat{h}' = \hat{h} \uplus (\hat{l}, \hat{p}) \mapsto \varnothing \\ \hat{d}' = \hat{d}[\hat{l} \mapsto \hat{d}(\hat{l}) \uplus \{\hat{p}\}]\end{array}}{\mathcal{GC}(\hat{\sigma}, e_1, e_2) \rightsquigarrow_s (\hat{\rho}, \hat{h}, \hat{d}, \hat{m}, \pi), (\hat{l}, \hat{p}, \varnothing)}$$

$$\textsc{GetCell - Branch - Found}\quad \frac{\begin{array}{c}\hat{\sigma} = (\hat{\rho}, \hat{h}, \hat{d}, \hat{m}, \pi) \\ \hat{l}, \hat{p} = \mathcal{E}v_{\mathsf{S}}(\hat{\rho}, e_1), \mathcal{E}v_{\mathsf{S}}(\hat{\rho}, e_2) \\ \hat{h}(\hat{l}, \hat{p}_i) = \underline{\hat{v}} \quad \hat{h}\triangleleft\hat{l} = \{\hat{p}_1, \ldots, \hat{p}_n\} \\ \pi \vdash \hat{d}(\hat{l}) = \{\hat{p}_1, \ldots, \hat{p}_n\} \\ \pi' = \pi \wedge (\hat{p} = \hat{p}_i) \quad Sat_{\mathsf{S}}(\hat{\sigma}, \pi')\end{array}}{\mathcal{GC}(\hat{\sigma}, e_1, e_2) \rightsquigarrow_s (\hat{\rho}, \hat{h}, \hat{d}, \hat{m}, \pi'), (\hat{l}, \hat{p}_i, \underline{\hat{v}})}$$

$$\textsc{GetProperties}\quad \frac{\begin{array}{c}\hat{\sigma} = (\hat{h}, \hat{d}, \hat{\rho}, -, \pi) \quad \hat{l} = \mathcal{E}v_{\mathsf{S}}(\hat{\rho}, e) \\ \hat{h}\triangleleft\hat{l} = \{\hat{p}_0, \ldots, \hat{p}_n\} \\ \hat{h} = - \uplus ((\hat{l}, \hat{p}_i) \mapsto \hat{v}_i)\,|_{i=0}^{m} \\ \pi \vdash \hat{d}(\hat{l}) = \{\hat{p}_0, \ldots, \hat{p}_m\} \\ \forall_{0 \le i \le n}\, \hat{v}_i \neq \varnothing \quad \forall_{n < i \le m}\, \hat{v}_i = \varnothing\end{array}}{\mathcal{GP}_{\mathsf{S}}(\hat{\sigma}, e) \triangleq \{\hat{p}_0, \ldots, \hat{p}_n\}}$$

Fig. 10. Symbolic JSIL semantics, selected non-failing transitions

the frame property. We deal with this issue as is done in [Fragoso Santos et al. 2018a]: we instantiate the Semantics Module with an instrumented state, obtaining a JSIL semantics that observes the frame property by explicitly keeping track of *absent* object properties. Due to lack of space, we do not present this semantics in detail, but do discuss its relevant design decisions in the next section.

3.5 Symbolic JSIL Semantics

We instantiate the Semantics Module to the symbolic JSIL semantics, which enables whole-program symbolic testing of JSIL programs. We instantiate general values to *symbolic values*, $\hat{v} \in \hat{\mathcal{V}} \triangleq v \mid \hat{x} \mid \ominus \hat{v} \mid \hat{v} \oplus \hat{v}$, and write $\hat{n}$, $\hat{b}$, $\hat{s}$, $\hat{l}$, and $\hat{p}$, to denote, respectively, symbolic numbers, booleans, strings, locations, and property names. A symbolic JSIL state, $\hat{\sigma} = (\hat{\rho}, \hat{h}, \hat{d}, \hat{m}, \pi)$, consists of a symbolic store $\hat{\rho}$, symbolic heap $\hat{h}$, a symbolic domain table $\hat{d}$, a symbolic metadata table $\hat{m}$, and a path condition π. Symbolic stores and metadata tables are obtained from their concrete counterparts by allowing symbolic values in place of concrete ones. Symbolic heaps, $\hat{h} \in \hat{\mathcal{H}} : ((\mathcal{L} \uplus \hat{\mathcal{L}}) \times \hat{\mathcal{V}}) \rightharpoonup \hat{\mathcal{V}}_{\varnothing}$, map pairs of object locations (both symbolic and concrete) and symbolic values to symbolic values extended with $\varnothing$. This means that symbolic heaps can explicitly track *absent* object properties. We denote the set of properties an object at location $\hat{l}$ has in the heap $\hat{h}$ by $\hat{h}\triangleleft\hat{l}$. Symbolic domain tables partially map object locations to sets of properties that the objects may have. Heaps and domain tables are connected via the *heap-domain invariant*, maintained by the semantics: for a given location $\hat{l}$, if $\hat{d}(\hat{l})$ is defined, then $\hat{h}\triangleleft\hat{l} \subseteq \hat{d}(\hat{l})$. This means that the object at location $\hat{l}$ *can have at most the properties that are in $\hat{d}(\hat{l})$, and all other properties are known to be absent.* Path conditions [Baldoni et al. 2018] bookkeep the constraints on the symbolic variables that led the execution to the current symbolic state. We write $\langle \hat{\sigma}, bc \rangle \rightsquigarrow_{\mathsf{S}} \hat{\sigma}'$ for the symbolic semantic judgement for JSIL basic commands and $\langle p, \hat{\sigma}, \hat{cs}, i \rangle^{\mu} \rightsquigarrow_{\mathsf{S}} \langle p', \hat{\sigma}', \hat{cs}', j \rangle^{\mu'}$ for JSIL commands. For clarity, we conflate JSIL logical values with logical values, and JSIL logical operators with boolean logical operators.

We give a selection of definitions for the SMI functions in Fig. 10. Expression evaluation is standard (symbolic variables evaluate to themselves). Allocation creates a fresh location and updates the domain table accordingly. Assumption evaluates the (boolean) expression passed as parameter and adds it to the current path condition. SetCell simply updates the heap.

The essence of the symbolic semantics lies in the interaction between GetCell, the heap, and the domain table. In the four GetCell rules, we are looking up the value of the property $\hat{p}$ of the object at location $\hat{l}$. In the [GetCell - Found] rule, we can prove that $\hat{p}$ is equal to one of the properties already in the heap, and we return the corresponding value. In this case, we do not require knowledge about

Symbolic Values
$$\mathcal{I}_\varepsilon(v) \triangleq v \qquad \mathcal{I}_\varepsilon(\hat{x}) \triangleq \varepsilon(\hat{x})$$

Symbolic Heaps
$$\mathcal{I}_\varepsilon((-,-) \mapsto \varnothing) \triangleq \emptyset \qquad \mathcal{I}_\varepsilon((\hat{l},\hat{p}) \mapsto \hat{v}) \triangleq (\mathcal{I}_\varepsilon(\hat{l}), \mathcal{I}_\varepsilon(\hat{p})) \mapsto \mathcal{I}_\varepsilon(\hat{v})$$

Symbolic States
$$\frac{\mathcal{I}_\varepsilon(\hat{h}) = h \quad \mathcal{I}_\varepsilon(\hat{\rho}) = \rho \quad \mathcal{I}_\varepsilon(\hat{m}) = m \quad \mathcal{I}_\varepsilon(\pi) = \mathsf{true}}{\mathcal{I}_\varepsilon(\hat{h}, \hat{d}, \hat{\rho}, \hat{m}, \pi) \triangleq (h, \rho, m)}$$

Type Error
$$\mathcal{I}_\varepsilon(\mathsf{TE}(\hat{v}, \tau_1, \tau_2)) \triangleq \mathsf{TE}(\mathcal{I}_\varepsilon(\hat{v}), \tau_1, \tau_2)$$

Resource Error
$$\mathcal{I}_\varepsilon(\mathsf{RE}(\overline{s})) \triangleq \mathsf{RE}(\mathcal{I}_\varepsilon(\overline{s}))$$

Assertion Error
$$\frac{\mathcal{I}_\varepsilon(p) = \mathsf{false}}{\mathcal{I}_\varepsilon(\mathsf{AE}(p)) \triangleq \mathsf{AE}(\mathsf{false})}$$

Specification Error
$$\frac{\mathcal{I}_\varepsilon(p) = \mathsf{true}}{\mathcal{I}_\varepsilon(\mathsf{SE}(-, \overline{s}, p)) \triangleq \mathsf{RE}(\mathcal{I}_\varepsilon(\overline{s}))}$$

Fig. 11. Interpretation: Symbolic to Concrete (relevant cases)

the domain of $\hat{l}$—it may, but need not be defined. In the [GetCell - Not Found] rule, the domain of $\hat{l}$ is defined and we can prove that $\hat{p}$ is not in the domain. Then, we know with certainty that the property does not exist, and return $\varnothing$. This rule reveals an important property of GetCell: *after executing GetCell, the inspected cell is guaranteed to exist in the heap.* Here, the inspected none-cell is migrated from the domain to the heap, and the domain is extended to maintain the heap-domain invariant. This choice localises reasoning about resource *to the GetCell only.* As a consequence, the SetCell is trivial; otherwise, we would have to repeat the reasoning of the GetCell in SetCell as well.

In the case that neither of the two above-mentioned entailments can be proven, the symbolic semantics can still try to branch on $\hat{p}$ being equal to any of the properties of the object in the heap ([GetCell - Branch - Found]) and also on it not being equal to any of them. This branching can occur only if we have *full knowledge* about the object at $\hat{l}$, meaning that its properties that are in the heap must be equal to those that are in its domain.

Another state function that requires full knowledge about the object is GetProperties, which returns the set of the properties of a given object. To do this correctly, we need to filter out the none-cells that might be in the heap.

3.5.1 Interpretation: from the Symbolic to the Concrete. Before we present error reporting and the theoretical results, we connect the symbolic and the concrete JSIL semantics. There are two main sources of tension: **(1)** the symbolic semantics can deal with symbolic variables, the concrete one cannot; and **(2)** symbolic states can be partial (i.e. may have *no information* about object properties), whereas concrete states are not (i.e. object properties are always either present or absent).

We resolve **(1)** by introducing concretisation functions, $\varepsilon : \hat{\mathcal{X}} \rightharpoonup \mathcal{V}$, which map symbolic variables to concrete values. We extend their domain to JSIL expressions, basic commands, commands, and programs in the standard way. To deal with **(2)**, we define interpretation functions, $\mathcal{I}_\varepsilon : \hat{\mathcal{V}} \rightharpoonup \mathcal{V}$, which map symbolic values to concrete values. An interpretation functions is parameterised by a concretisation function. We extend the domain of $\mathcal{I}_\varepsilon$ to include symbolic stores, symbolic heaps, symbolic metadata tables, assertions, path conditions, symbolic states, errors, call stacks, and execution modes, most of which is straightforward. We highlight the important cases in Fig. 11. When it comes to symbolic values, $\mathcal{I}_\varepsilon$ leaves concrete values unchanged and evaluates symbolic variables using ε. When interpreting symbolic heaps, none-cells are forgotten, since the concrete state has no information about absent properties, whereas cells with positive information are interpreted component-wise. Symbolic states are also interpreted component-wise, noting that domain tables are forgotten, as they have no counterpart in the concrete semantics, and also that the interpretation of the path condition must equal true, effectively meaning that our symbolic state needs to be satisfiable. We delay the interpretation of errors to the next subsection, together with a formal correspondence result between the symbolic and the concrete semantics.

3.5.2 Errors in the Symbolic JSIL Semantics. We focus on specification errors, which are crucial for error management and reporting in our analysis. Given a spec error, $\mathsf{SE}(p_f, \overline{s_c}, p_c)$, the *failing*

GetCell - Missing Property (New)

$$\frac{\begin{array}{c}\hat\sigma = (\hat\rho, \hat h, \hat d, \hat m, \pi) \\ \mathcal{E}v_s(\hat\rho, e_1) = \hat l \quad \mathcal{E}v_s(\hat\rho, e_2) = \hat p \\ \hat h \triangleleft \hat l = \{\hat p_1, \ldots, \hat p_n\} \quad p_f = \hat p \in \hat d(\hat l) \wedge \hat p \notin \{\hat p_1, \ldots, \hat p_n\} \\ \mathcal{S}at(\hat\sigma, p_f) \quad \underline{\hat v} = \mathsf{GenSym}(\mathcal{V}_\varnothing) \quad \overline{s} = [(\hat l, \hat p) \mapsto \underline{\hat v}]\end{array}}{\mathcal{GC}(\hat\sigma, (e_1, e_2)) \rightsquigarrow_s \mathsf{SE}(p_f, \overline{s}, p_f)}$$

GetCell - Missing Property (Existing)

$$\frac{\begin{array}{c}\hat\sigma = (\hat\rho, \hat h, \hat d, \hat m, \pi) \\ \mathcal{E}v_s(\hat\rho, e_1) = \hat l \quad \mathcal{E}v_s(\hat\rho, e_2) = \hat p \\ \hat h \triangleleft \hat l = \{\hat p_1, \ldots, \hat p_n\} \quad p_f = \hat p \in \hat d(\hat l) \wedge \hat p \notin \{\hat p_1, \ldots, \hat p_n\} \\ \mathcal{S}at(\hat\sigma, p_f) \quad \mathcal{S}at(\hat\sigma, \hat p = \hat p_i)\end{array}}{\mathcal{GC}(\hat\sigma, (e_1, e_2)) \rightsquigarrow_s \mathsf{SE}(p_f, \mathsf{emp}, \hat p = \hat p_i)}$$

Fig. 12. Symbolic JSIL semantics, selected failing transitions

constraint, p_f, is the constraint under which the error is triggered, whereas the *spatial corrective*, $\overline{s_c}$ and the *pure corrective*, p_c, describe how to correct the error. In particular, the spatial corrective holds the missing resource and the pure corrective connects that resource to the rest of the state. The way to conceptualise the correctives is: if we are in the state $\hat\sigma$ and we got a spec error $\mathsf{SE}(p_f, \overline{s_c}, p_c)$, then if we were to extend $\hat\sigma$ with $\overline{s_c}$ and p_c, the spec error would no longer occur. This design is tightly coupled with the inference of resource of the bi-abductive semantics, as described in §5.

To understand spec errors better, consider the two failing GetCell transitions shown in Fig. 12. In both cases, we are looking up the property $\hat p$ of the object at location $\hat l$, the location exists in the heap, we have full knowledge about the object, and it is possible to have no information about the property ($\mathcal{S}at(\hat\sigma, p_f)$). Then, we can correct the error in two ways. First, the property $\hat p$ could indeed be a new property ([GetCell - Missing Property (New)]), in which case we add it to the heap with an arbitrary value (possibly equal to $\varnothing$), together with the information that it is new. This case reveals an insight: *if the spatial corrective is not empty, the failing constraint is the pure corrective.* Second, the property $\hat p$ could be equal to one of the existing properties, but we might not be able to prove it ([GetCell - Missing Property (Existing)]). Still, if that is possible ($\mathcal{S}at(\hat\sigma, \hat p = \hat p_i)$), we could extend the state with $\hat p = \hat p_i$ and the error would not occur. In this case, the spatial corrective is empty, as there is no missing resource—it has been resolved by the pure corrective.

Interpretation of Errors. Type errors, resource errors, and assertion errors are preserved by interpretation (cf. Fig. 11). This is intuitive: for instance, if a symbolic value is of a certain type, then we must be able to choose concrete values for its symbolic variables and still maintain that type.

Specification errors in the symbolic semantics, in contrast, correspond to resource errors in the concrete semantics. This is because information about resource is treated differently between the two semantics: in the symbolic world, if we have no information, we actually know nothing about the resource; in the concrete world, this means that the resource is absent.

```
proc f(o) {
  ...
  [o, "p"] := 0
  ...
}
```

Finally, silent spec errors do not have an interpretation, as they do not transfer from the symbolic to the concrete semantics, due to a mismatch between separation logic and languages with extensible objects. Consider the JSIL procedure f on the right, which receives an object o and assigns to its property "p". As we are in separation logic, the pre-condition of f must have information about (o, "p"); otherwise, the symbolic analysis will signal a specification error. Now, when we try to produce a corresponding concrete error, the obtained concrete state will also have no information about (o, "p"). However, the concrete property assignment will succeed, adding the new property to the object, as it does not require the resource to be present (cf. Fig. 5, Fig. 9, [GetCell - Not Found] rule). This would not be the case with languages such as Java, where properties cannot be added to objects after creation.

Bounded Soundness and Verification. In order to state the soundness and verification theorems, we give the requirements that external procedures must meet.

Definition 3.1 (Well-Behaved External Procedures). An external procedure table Ψ is well-behaved, written $\mathcal{WB}_s(\Psi)$, *if and only if*, for every procedure $\phi \in \mathsf{dom}(\Psi)$, it holds that:

$$\phi(p, \hat v_i|_{i=0}^n) = p', \hat v, \hat b \implies \forall \varepsilon.\ \phi(\varepsilon(p), \mathcal{I}_\varepsilon(\hat v_i)|_{i=0}^n) = \varepsilon(p'), \mathcal{I}_\varepsilon(\hat v), \mathcal{I}_\varepsilon(\hat b)$$

We can now formally connect the symbolic and the concrete JSIL semantics. Theorem 3.2 states that if external procedures are well-behaved, then all executions in the symbolic JSIL semantics that do not conclude with a silent specification error can be interpreted concretely. In the statement of the theorems, all variables that are not existentially quantified are implicitly universally quantified. Also, for a given symbolic state $\hat{\sigma}$, we denote its path condition by $\hat{\sigma}.\pi$.

Theorem 3.2 (Bounded Soundness).

$$\mathcal{WB}_s(\Psi) \wedge \Psi \vdash \langle p, \hat{\sigma}, \hat{cs}, i\rangle^C \rightsquigarrow_s^* \langle p', \hat{\sigma}', \hat{cs}', j\rangle^\mu \wedge \mu \neq \mathsf{F}(\mathsf{SE}^\dagger(-,-)) \implies$$
$$\forall \varepsilon.\ \mathcal{I}_\varepsilon(\hat{\sigma}'.\pi) = \mathsf{true} \implies \Psi \vdash \langle \varepsilon(p), \mathcal{I}_\varepsilon(\hat{\sigma}), \mathcal{I}_\varepsilon(\hat{cs}), i\rangle^C \rightsquigarrow_c^* \langle \varepsilon(p'), \mathcal{I}_\varepsilon(\hat{\sigma}'), \mathcal{I}_\varepsilon(\hat{cs}'), j\rangle^{\mathcal{I}_\varepsilon(\mu)}$$

Onward, we write $\Psi \vdash \langle p, \hat{\sigma}, \hat{cs}, i\rangle^C \twoheadrightarrow_s \{\langle p_k, \hat{\sigma}_k, \hat{cs}_k, j_k\rangle^{\mu_k}|_{k=0}^n\}$ to mean that the following statements hold: **(1)** $\forall\ 0 \leq k \leq n.\ \Psi \vdash \langle p, \hat{\sigma}, \hat{cs}, i\rangle^C \rightsquigarrow_s^* \langle p_k, \hat{\sigma}_k, \hat{cs}_k, j_k\rangle^{\mu_k} \wedge \mu_k \neq F(\mathsf{SE}^\dagger(-,-))$; and **(2)** $\bigvee_{k=0}^n \hat{\sigma}_k.\pi \implies \hat{\sigma}.\pi$. Informally, $\Psi \vdash \langle p, \hat{\sigma}, \hat{cs}, i\rangle^C \twoheadrightarrow_s \{\langle p_k, \hat{\sigma}_k, \hat{cs}_k, j_k\rangle^{\mu_k}|_{k=0}^n\}$ means that the set $\{\langle p_k, \hat{\sigma}_k, \hat{cs}_k, j_k\rangle^{\mu_k}\ |_{k=0}^n\}$ contains all possible configurations reachable from $\langle p, \hat{\sigma}, \hat{cs}, i\rangle$.

Theorem 3.3 (Verification).

$$\mathcal{WB}_s(\Psi) \wedge \Psi \vdash \langle p, \hat{\sigma}, \hat{cs}, i\rangle^C \twoheadrightarrow_s \{\langle p_k, \hat{\sigma}_k, \hat{cs}_k, j_k\rangle^{\mu_k}|_{k=0}^n\} \implies$$
$$\forall \varepsilon.\ \mathcal{I}_\varepsilon(\hat{\sigma}'.\pi) = \mathsf{true} \implies \exists k.\ \Psi \vdash \langle \varepsilon(p), \mathcal{I}_\varepsilon(\hat{\sigma}), \mathcal{I}_\varepsilon(\hat{cs}), i\rangle^C \rightsquigarrow_c^* \langle \varepsilon(p_k), \mathcal{I}_\varepsilon(\hat{\sigma}_k), \mathcal{I}_\varepsilon(\hat{cs}_k), j_k\rangle^{\mathcal{I}_\varepsilon(\mu_k)}$$

Finally, the verification theorem states that if we have explored all possible execution paths starting from a given configuration and there are no silent specification errors, then the execution of the program starting from any interpretation of the *initial* symbolic state will result in an interpretation of one of the final symbolic states. As the whole-program symbolic execution does not support loop invariants, JSIL programs that contain unbounded loops cannot be verified.

Meaningful Error Reporting. Our goal is to report meaningful errors to the user during whole-program symbolic testing. Let us assume that the analysis terminated with an error in the final configuration $\langle -, \hat{\sigma}, -, -\rangle^{\mathsf{F}(\xi)}$. Then, we give back to the user: **(1)** the description of the error ξ; and **(2)** a concretisation ε of the symbolic variables of $\hat{\sigma}$ that triggers the error.[2]

We believe that this information is useful for users doing whole-program symbolic testing: it allows them to potentially gain insight into the nature of the error in the more familiar, concrete setting. The errors that can occur here are: type errors, resource errors, and assertion failures. As the execution is whole-program, we cannot witness spec errors or silent spec errors. Given Theorem 3.2, this means that the error will always be reproducible at the concrete level.

Providing meaningful error messages during verification is a much more difficult task, as they would need to be lifted from JSIL to JavaScript properly in the context of specifications, predicate manipulation, and unification (cf. §4). This is part of our further work, as outlined in §8.

4 THE SPECIFICATION MODULE

We introduce a new treatment of separation logic (SL) assertions, by defining them in terms of the general local actions of the Semantics Module. This allows us to: use SL specifications during symbolic execution to jump over procedure calls instead of re-executing a procedure at each call site; and use symbolic execution to verify SL specifications efficiently instead of re-implementing an SL proof system from scratch. In this way, we bridge the gap between classical symbolic execution and SL proof systems, bringing benefits to both worlds. We accomplish this by designing the *Specification Module*, which receives a state signature as input and generates a new state signature with a built-in mechanism for executing procedure calls abstractly, using SL specifications.

[2]This concretisation can be obtained from, for example, an SMT solver. We use Z3 of De Moura and Bjørner [2008].

Interpretation of SL-Assertions. When designing a Separation Logic for a given programming language, one normally first chooses an assertion language and connects it to the concrete states of the programming language via a satisfiability relation [Reynolds 2002]; therefore, an assertion can be viewed as the set of *concrete states* that satisfy it. Here, we define the meaning of assertions abstractly using the signature of general JSIL states. In this way, we effectively establish the connection between our assertion language and the concrete, instrumented, and symbolic states of the language using the same definition.

JSIL Assertions are given by the grammar $P, Q ::= \texttt{emp} \mid p \mid s \mid P * Q$, where emp denotes the empty heap and p and s range over pure and spatial assertions, respectively, as defined in §3.3.1, but lifted to expressions instead of general values. We refer to assertions distinct from emp and $- * -$ as *simple assertions*. We describe the semantics of JSIL assertions w.r.t. general states using two functions: $\textsc{Produce}(\Sigma, P, \theta)$ and $\textsc{Consume}(\Sigma, P, \theta)$, both of which take as parameters a general state Σ, an assertion P, and a substitution θ, mapping the logical variables in P to general values. Produce returns a new general state Σ', obtained by adding the resource of P to Σ, whereas $\textsc{Consume}(\Sigma, P, \theta)$ returns a new general state Σ' obtained by consuming the resource of P from Σ, and a new substitution θ' extending θ with the bindings for the logical variables in P. Importantly, if $\textsc{Consume}(\Sigma, P, \theta) = (\Sigma', \theta')$, then $\textsc{Produce}(\Sigma', P, \theta') = \Sigma$. Using these two functions, we solve the standard frame inference problem (FIP) at the general level.

Unification Plans. When linking an assertion P to a state Σ, one has to find appropriate bindings for the logical variables in P. To do this, we introduce *unification plans* (UPs, Definition 4.1). Informally, a UP is an ordering of the simple assertions in P that guarantees that Consume need not backtrack at runtime. To construct UPs, we define *in-variables* (ins) and *out-variables* (outs) for each assertion and JSIL expression, resembling predicate parameter modes of Nguyen et al. [2008]. Intuitively, the out-variables of an assertion are those that can be computed using the in-variables and the current state. For instance, given the assertion $(e_1, e_2) \mapsto e_3$, if we know the bindings for e_1 and e_2, we can compute the bindings for e_3. For logical expressions, the out-variables are those that can be computed given the value of the whole expression, whereas the in-variables are those that we need to know to compute the value of the whole expression. For instance, $out([\hat{x}_1, \hat{x}_2, \hat{x}_3]) = \{\hat{x}_1, \hat{x}_2, \hat{x}_3\}$, as we can compute the values of $\hat{x}_1$, $\hat{x}_2$, and $\hat{x}_3$ if given the entire list. In contrast, for example, $out(\hat{x}_1 + \hat{x}_2) = \{\}$, because we cannot compute the values of either $\hat{x}_1$ or $\hat{x}_2$ solely from the value of $\hat{x}_1 + \hat{x}_2$. We note that it is not always possible to generate a UP for an SL-assertion, and onward consider only assertions that admit a UP. In the implementation, we cater for assertions that cannot be turned into UPs by requiring the user to manually provide a substitution for their unification. This is a design choice that allows tractable verification in the presence of assertions that represent JavaScript heaps.

Argument	**IN**	**OUT**
$\hat{x}$	$\{\hat{x}\}$	$\{\hat{x}\}$
$[e_1, ..., e_n]$	$\cup_{i=1}^{n}(in(e_i))$	$\cup_{i=1}^{n}(out(e_i)))$
$e_1 + e_2$	$in(e_1) \cup in(e_2)$	$\emptyset$
...	...	...
$(e_1, e_2) \mapsto e_3$	$in(e_1) \cup in(e_2)$	$out(e_3)$
$\texttt{noProps}(e_1, e_2)$	$in(e_1) \cup in(e_2)$	$\emptyset$
$\texttt{MetaData}(e_1, e_2)$	$in(e_1)$	$out(e_2)$
$e_1 = e_2$	$in(e_1)/in(e_2)$	$out(e_2)/out(e_1)$

Definition 4.1 (Unification Plan). A *unification plan up* is a sequence of simple assertions $q_i \mid_{i=0}^{n}$ such that for all $0 \leq i \leq n$, it holds that: $in(q_i) \subseteq \left(\cup_{j=0}^{i-1} out(q_j) \right)$

Produce and Consume. In Figure 13, we show the Produce algorithm on the left and a fragment of the Consume algorithm on the right. For clarity, we define these algorithms in terms of UPs rather than assertions. The Consume algorithm uses the function $\mathcal{U}_{\mathsf{E}}(\Sigma, e, \mathsf{v}, \theta)$ for unifying a JSIL expression e against a general value v, either extending the input substitution θ with the appropriate bindings, in case of success, or generating a pure assertion describing the failure. Both

```
function PRODUCE($\Sigma$, $up$, $\theta$)
  match $up$ with
  | [ ] : return $\Sigma$
  | $(e_1, e_2) \mapsto e_3 :: up'$ :
     let $\Sigma' = \mathcal{SC}(\Sigma, \theta(e_1), \theta(e_2), \theta(e_3))$ in
        return PRODUCE($\Sigma'$, $up'$, $\theta$)
  | noProps$(e_1, e_2) :: up'$ :
     let $\Sigma' = \mathcal{SP}(\Sigma, \theta(e_1), \theta(e_2))$ in
        return PRODUCE($\Sigma'$, $up'$, $\theta$)
  | MetaData$(e_1, e_2) :: up'$ :
     let $\Sigma' = \mathcal{SM}(\Sigma, \theta(e_1), \theta(e_2))$ in
        return PRODUCE($\Sigma'$, $up'$, $\theta$)
  | $(x = e) :: up'$ :
     let $\Sigma' = \mathcal{SS}(\Sigma, x, \theta(e))$ in
        return PRODUCE($\Sigma'$, $up'$, $\theta$)
  | $p :: up'$ : let $\Sigma' = \mathcal{A}sm(\Sigma, \theta(p))$ in
        return PRODUCE($\Sigma'$, $up'$, $\theta$)
end function
```

```
function CONSUME($\Sigma$, $up$, $\theta$)
    match $up$ with
    | [ ] : return Succ $(\theta, \Sigma)$
    | $(e_1, e_2) \mapsto e_3 :: up'$ :
       match $\mathcal{GC}(\Sigma, \theta(e_1), \theta(e_2))$ with
          | $\Sigma', (\mathtt{l}, \mathtt{p}, \mathtt{v})$ :
             let $\Sigma'' = \mathcal{RC}(\Sigma, \mathtt{l}, \mathtt{p})$ in
             match $\mathcal{U}_{\mathsf{E}}\,(\Sigma'', e_3, \mathtt{v}, \theta)$ with
             | Succ $(\theta')$ : return CONSUME($\Sigma''$, $up'$, $\theta'$)
             | Fail $(\pi_f)$ : return SE$(\pi_f, [], \neg\pi_f)$
          | $\xi$ : return $\xi$
       | $(e_1 = e_2) :: up'$ :
             let $\hat{v}_2 = \mathcal{E}v(\mathsf{P}, \theta(e_2))$ in
             match $\mathcal{U}_{\mathsf{E}}\,(\Sigma, e_1, \hat{v}_2, \theta)$ with
             | Succ $(\theta')$ : return CONSUME($\hat{\sigma}$, $up'$, $\theta'$)
             | Fail $(\pi_f)$ : return SE$(\pi_f, [], \neg\pi_f)$
    | ...
end function
```

Fig. 13. Interpretation of Assertions via PRODUCE and CONSUME

algorithms are tail-recursive: if given an empty unification plan, they both return, if not, they produce/consume the first simple assertion in the given unification plan, and continue recursively.

The PRODUCE and CONSUME algorithms reveal the connections between assertions and the local actions of the state. For example, the cell assertion $(e_1, e_2) \mapsto e_3$ is connected to the GetCell, SetCell, and RemoveCell[3] local actions. One way to think about these three functions is as a getter (GetCell), a setter (SetCell), and a remover (RemoveCell) of the cell assertion. Other assertions can also be associated with local actions: e.g., pure assertions are managed using Assume and SatCheck.

While the PRODUCE algorithm is fairly straightforward—it uses the setters of the state (SetCell, SetProperties, SetMetadata, SetStore, and Assume) to generate the footprint of the given assertions—the CONSUME algorithm is more involved. For instance, when consuming a cell assertion, the algorithm proceeds as follows: **(1)** it applies the current substitution θ to e_1 and e_2 (the *ins* of the cell assertion), obtaining two expressions with no existentially quantified logical variables; **(2)** it applies GetCell to the resulting expressions, obtaining the values $\mathtt{l}$, $\mathtt{p}$, and $\mathtt{v}$, respectively denoting the location, property name, and value corresponding to the cell assertion in the current state Σ; **(3)** it removes the cell assertion from the state, using the RemoveCell function; and **(4)** it unifies the obtained value $\mathtt{v}$ against e_3 using the auxiliary function $\mathcal{U}_{\mathsf{E}}$; if the unification succeeds, the algorithm continues; if it does not, it produces a specification error describing the failure.

We now illustrate how the CONSUME algorithm works using the example of the assertion below, shown in the symbolic state depicted in Figure 14.

```
(e, "type") -> "unop" * (e, "arg") -> #x *
   (#x, "type") -> "lit" * (#x, "val") -> #y
```

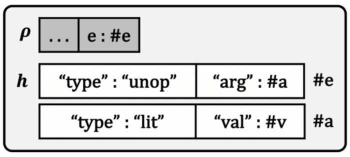


Fig. 14. Symbolic State

Notice that any unification plan for the assertion above needs to place the assertion `(e, "arg") -> #x` prior to the assertions `(#x, "type") -> "lit"` and `(#x, "val") -> #y`, since we need to find the binding for `#x` before consuming the cell assertions that have location `#x`. Therefore, one possible unification plan for this assertion would be as follows: **(1)** `(e, "type") -> "unop"`; **(2)** `(e, "arg") -> #x`, obtaining the binding for `#x`; **(3)** `(#x, "type") -> "lit"`;

[3]RemoveCell ($\mathcal{RC}$) originates from the Specification Module and removes a given cell from the state.

and **(4)** (#x, "val") −> #y, obtaining the binding for #y. In the end, the final substitution returned by the algorithm is $\theta = [\#x \mapsto \#a, \#y \mapsto \#v]$.

Applying Specifications. JSIL Logic specifications have the form $fl : \{P\}\, f(\overline{x})\, \{Q\}$, where P and Q are the pre- and post-conditions of the procedure with identifier f and parameters $\overline{x}$. Each spec is associated with a return flag $fl \in \{\mathsf{nm}, \mathsf{er}\}$, indicating if the procedure terminates via a return or a throw. We give the rule for the function ApplySpec(Σ, *spec*, $\overline{\mathsf{v}}$), for applying the spec *spec* to the state Σ, using the arguments given by $\overline{\mathsf{v}}$. ApplySpec returns a triple $(\Sigma', \mathsf{v}, fl)$, consisting of the final state Σ', the return value v, and the return flag fl. The rule is applied as follows: **(1)** we replace the calling state store P with a new store P', mapping the procedure parameters to the call arguments, obtaining a new state Σ_1; **(2)** we *consume* the precondition of the spec from the state Σ_1, obtaining both the bindings for the logical variables in the precondition, θ, and the frame state Σ_2 (corresponding to Σ_1 minus the resource of the precondition); **(3)** we *produce* the post-condition of the spec in the state Σ_2 using the bindings computed in step **(2)**, and, finally, **(4)** we restore the calling store P and return the value of ret in the final store of the procedure.

Apply Spec

$$\frac{\begin{array}{c} spec = fl : \{P\}\, f(x_0, \ldots, x_m)\, \{Q\} \quad \mathsf{P}' = [x_i \mapsto \mathsf{v}_i|_{i=0}^{m}] \\ \mathsf{P} = \mathcal{GS}(\Sigma) \quad \Sigma_1 = \mathcal{RS}(\Sigma, \mathsf{P}') \\ (\theta, \Sigma_2) = \textsc{Consume}(\Sigma_1, P) \quad \Sigma_3 = \textsc{Produce}(\Sigma_2, Q) \\ \mathsf{v} = \mathcal{E}v(\mathcal{GS}(\Sigma_3), \mathtt{ret}) \quad \Sigma' = \mathcal{RS}(\Sigma_3, \mathsf{P}) \end{array}}{\textsc{ApplySpec}(\Sigma, spec, \overline{\mathsf{v}}) = (\Sigma', \mathsf{v}, fl)}$$

We extend the Semantic Module so that, when parameterised with a state signature that supports abstract specification execution, it uses the ApplySpec function for executing procedure calls when specifications are provided. We show the new procedure call rule, applied as follows: **(1)** we evaluate the expression denoting the procedure name and obtain its spec; **(2)** we evaluate the parameters of the procedure; **(3)** we use the ApplySpec function to compute the final state, return value, and flag; and **(4)** we transfer control to the next command, when $fl = \mathsf{nm}$, or the j-th command, when $fl = \mathsf{er}$; in both cases, the value of x in the store is set to the returned value.

Procedure Call - Use Spec

$$\frac{\begin{array}{c} \mathsf{cmd}(i) = x := e(e_i|_{i=0}^{n})\ \mathsf{with}\ j \quad \mathsf{P} = \mathcal{GS}(\Sigma) \\ f = \mathcal{E}v(\mathsf{P}, e) \quad spec = \mathcal{S}pec(f) \quad \overline{\mathsf{v}} = \mathcal{E}v(\mathsf{P}, e_i|_{i=0}^{n}) \\ (\Sigma', \mathsf{v}, fl) = \textsc{ApplySpec}(\Sigma, spec, \overline{\mathsf{v}}) \\ k = \ \mathsf{if}\ fl = \mathsf{nm}\ \mathsf{then}\ i{+}1\ \mathsf{else}\ j \end{array}}{\langle \Sigma, \mathsf{cs}, i \rangle^{\mathsf{C}} \rightsquigarrow_{\mathsf{spec}} \langle \mathcal{SS}(\Sigma', x, \mathsf{v}), \mathsf{cs}, k \rangle^{\mathsf{C}}}$$

Verification. By plugging the symbolic state instantiation into the Specification Module and the resulting state instantiation into the Semantics Module, we obtain a symbolic semantics with support for abstract execution of procedure calls with SL-specifications. We denote this semantics relation by $\rightsquigarrow_{\mathsf{spec(s)}}$ and re-state the Verification Theorem (Theorem 3.3) in terms of it. In the following, we use $\langle p, \sigma, cs, i \rangle \Downarrow_{\mathsf{c}}^{\Psi}$ to mean that the concrete execution starting with configuration $\langle p, \sigma, cs, i \rangle$ with external procedure table Ψ terminates.

Theorem 4.2 (Verification).

$$\begin{array}{l} \mathcal{WB}_s(\Psi) \wedge \Psi \vdash \langle p, \hat{\sigma}, \hat{cs}, i \rangle^{\mathsf{C}} \twoheadrightarrow_{\mathsf{spec(s)}} \{\langle p_k, \hat{\sigma}_k, \hat{cs}_k, j_k \rangle^{\mu_k}|_{k=0}^{n}\} \implies \\ \quad \forall \varepsilon.\ \mathcal{I}_\varepsilon(\hat{\sigma}'.\pi) = \mathsf{true} \ \wedge\ \langle \varepsilon(p), \mathcal{I}_\varepsilon(\hat{\sigma}), \mathcal{I}_\varepsilon(\hat{cs}), i \rangle \Downarrow_{\mathsf{c}}^{\Psi} \implies \\ \quad\quad \exists k.\ \Psi \vdash \langle \varepsilon(p), \mathcal{I}_\varepsilon(\hat{\sigma}), \mathcal{I}_\varepsilon(\hat{cs}), i \rangle^{\mathsf{C}} \rightsquigarrow_{c}^{*} \langle \varepsilon(p_k), \mathcal{I}_\varepsilon(\hat{\sigma}_k), \mathcal{I}_\varepsilon(\hat{cs}_k), j_k \rangle^{\mathcal{I}_\varepsilon(\mu_k)} \end{array}$$

Observe that, in contrast to the symbolic semantics of §3, $\rightsquigarrow_{\mathsf{spec(s)}}$ has support for recursive procedures with SL-specifications. Hence, we can use $\rightsquigarrow_{\mathsf{spec(s)}}$ to verify JSIL programs that contain unbounded loops (using recursion). Given a specification $fl : \{P\}\, f(\overline{x})\, \{Q\}$, the verification of JaVerT 2.0 proceeds as follows: **(1)** It associates each parameter in $\overline{x}$ with a fresh logical variable, obtaining a substitution θ; **(2)** It converts the precondition P to a symbolic state using the Produce algorithm and θ; **(3)** It runs the abstract symbolic execution on the initial symbolic state, obtaining a set of possible final states; **(4)** It uses the Consume algorithm to unify the post-condition Q against every possible final state and fl.

5 THE BI-ABDUCTION MODULE

Automatic compositional testing mandates symbolic execution of procedures for which we have no specifications. To accommodate this, we extend the JSIL symbolic semantics with a bi-abductive mechanism [Calcagno et al. 2009b] for automatically inferring the missing resource whenever a specification error occurs. Instead of creating a new bi-abductive symbolic analysis from scratch, we design a new module that receives a state signature as input and generates a new state signature with a built-in mechanism for on-the-fly correction of specification errors during execution. Following this approach, we can reuse both the JSIL Semantics Module and our implementation of the symbolic state, obtaining a modular implementation of the bi-abductive analysis with clear meta-theoretical results. This is made possible by the design decisions underpinning our system.

Bi-abductive state. Given a general state Σ, the Bi-Abductive Module constructs a bi-abductive state Σ_{bi} of the form (Σ, Σ_{af}), where Σ (the *main state*) describes the state at a given execution point, and Σ_{af} (the *anti-frame*) describes the resources that need to be added to the original initial state so that the program can reach that execution point without faulting. The main state and the anti-frame have the same type: for our analysis, they are both symbolic states.

Bi-abductive Rules. The construction of the bi-abductive signature is guided by our general error treatment. Selected rules are presented in Fig. 15. Successful transitions of the state functions are lifted directly (cf. [Assume - Success], [GetCell - Found]), and so are type/resource errors (cf. [GetCell - Type Error]), as they cannot be corrected (e.g. an expression's type cannot be changed once it has been established). Similarly, if we know for sure that a given cell does not exist in a given state, we cannot extend that state with that cell.

Spec errors, $\mathsf{SE}(-, \overline{s}, p)$, are corrected using the spatial and pure correctives, $\overline{s}$ and p, respectively. For instance, the rule [GetCell - Spec Error] is applied as follows: **(1)** we execute the GetCell function with the supplied arguments on the main state, Σ, obtaining a spec error $\mathsf{SE}(-, \overline{s}, p)$; **(2)** we use the correctives to extend the main state and the anti-frame using the Produce function of §4; and **(3)** we recursively call GetCell on the extended state (Σ', Σ'_{af}). This call will succeed, given the design of our error reporting mechanism.

Assume - Success

$$\frac{\mathcal{A}sm(\Sigma, e) = \Sigma' \quad \mathcal{A}sm(\Sigma_{af}, e) = \Sigma'_{af}}{\mathcal{A}sm((\Sigma, \Sigma_{af}), e) = (\Sigma', \Sigma'_{af})}$$

GetCell - Found

$$\frac{\Sigma_{bi} = (\Sigma, \Sigma_{af}) \qquad \mathcal{G}C(\Sigma, e_1, e_2) \rightsquigarrow \Sigma', (\mathsf{l}, \mathsf{p}, \mathsf{v})}{\mathcal{G}C(\Sigma_{bi}, e_1, e_2) \rightsquigarrow_{\mathsf{bi}} (\Sigma', \Sigma_{af}), (\mathsf{l}, \mathsf{p}, \mathsf{v})}$$

GetCell - Type Error

$$\frac{\mathcal{G}C(\Sigma, e_1, e_2) \rightsquigarrow \mathsf{TE}(e, \tau_1, \tau_2)}{\mathcal{G}C((\Sigma, \Sigma_{af}), e_1, e_2) \rightsquigarrow_{\mathsf{bi}} \mathsf{TE}(e, \tau_1, \tau_2)}$$

GetCell - Spec Error

$$\frac{\begin{array}{l}\Sigma_{bi} = (\Sigma, \Sigma_{af}) \\ \mathcal{G}C(\Sigma, e_1, e_2) \rightsquigarrow \mathsf{SE}(-, \overline{s}, p) \\ \Sigma' = \textsc{Produce}(\Sigma, \overline{s} * p) \\ \Sigma'_{af} = \textsc{Produce}(\Sigma_{af}, \overline{s} * p) \\ \mathcal{G}C((\Sigma', \Sigma'_{af}), e_1, e_2) \rightsquigarrow_{\mathsf{bi}} \Sigma'_{bi}, (\mathsf{l}, \mathsf{p}, \mathsf{v})\end{array}}{\mathcal{G}C(\Sigma_{bi}, e_1, e_2) \rightsquigarrow_{\mathsf{bi}} \Sigma'_{bi}, (\mathsf{l}, \mathsf{p}, \mathsf{v})}$$

Fig. 15. Selected Bi-abductive Rules

Example. To illustrate the inner workings of the bi-abductive symbolic execution, we appeal to the expression evaluator example shown in §2. In Fig. 16, we give a simplified compilation of part of the code of `evalExpr`. It first checks whether the expression `e` denotes a literal (lines 1, 2), in which case it returns the respective value (lines 3 and 4); otherwise, it checks if `e` denotes a unary operator (line 5), in which case, it evaluates the respective argument (lines 6 and 7) and returns the result of `evalUnop` (lines 9 and 10).

```
1        t := [e, "type"];
2        goto [t = "lit"] lit r1;
3   lit: ret := [t, "val"];
4        return;
5    r1: goto [t = "unop"] uo r2;
6    uo: arg := [e, "arg"];
7        arg_v := "evalExpr"(store, arg);
8        op := [e, "op"];
9        ret := evalUnop(op, arg_v);
10       return;
11   r2: ...
```

Fig. 16. Expr. Evaluator (JSIL fragment)

Fig. 17 shows a bi-abductive symbolic execution trace corresponding to the execution of the code snippet in Fig. 16. The symbolic states are shown before and after each executed JSIL command. We highlight parts of the anti-frame in yellow when added for the first time, and onward in white. We note that, in general, the anti-frame can be modified by the execution—in this particular example, it is not. In the bottom half of the diagram, we elide parts of the store for space reasons.

The execution starts without resource, with the procedure parameters, store and e, assigned fresh symbolic variables, #s and #e, in the store. We execute x := [e, "type"] and get a spec error as the heap is empty: we correct it by adding the inspected cell (with a fresh value #t) to the main state/anti-frame, and continue. We reach the command goto [t = "lit"] lit r1 where the execution branches, as we have no knowledge of #t. We show the execution of the negative branch, where #t != "lit", adding the constraint to the path condition of the main state/anti-frame. Next, we reach another branching point, goto [t = "unop"] uo r2, and show the positive branch, registering that #t = "unop". We then obtain the argument of the unary operator, arg := [e, "arg"], bi-abducing the property "arg" of the object #e. Next, we run into a recursive call: arg_v := "evalExpr"(store, arg). We then suspend this bi-abduction until we have explored the other branches and collected the base cases of the function. Once re-activated, this bi-abduction will branch on all of them—we show the literal case, which has the spec:

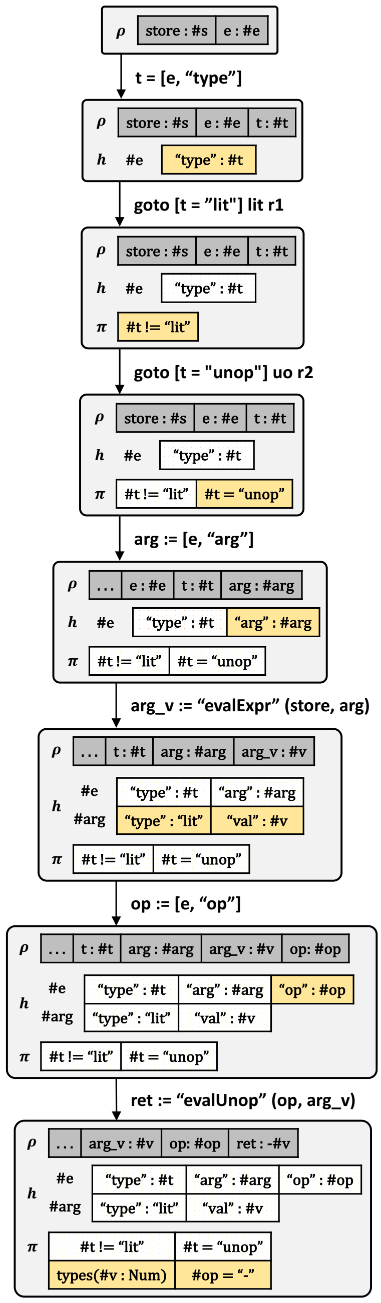
```
{ (e, "type") ↦ "lit" * (e, "val") ↦ #v }
                 evalExpr(store, e)
                 { PRE * ret = #v }
```

To satisfy this pre-condition, we add to the anti-frame the object at location #arg, with properties "type" (with value "lit") and "val" (with value #v). The post-condition extends the store with the pair (arg_v, #v). Next, we get the unary operator, bi-abducing the property "op" of #e, and evaluate it on the obtained value, branching on the specs of the procedure evalUnop. Here, we apply the following spec:

```
{ op = "-" * types(v:Num) }
       evalUnop(op, v)
      { PRE * ret = -v }
```

extending the anti-frame to meet the pre-condition and the store with the information from the post-condition, obtaining ret = −#v. Finally, we return, constructing a new spec of evalExpr, integrating the information from the original state and the bi-abduced anti-frame:

```
⎧ (e = #e) * (#e, "type") ↦ "unop" * (#e, "op") ↦ "-"  ⎫
⎨     * (#e, "arg") ↦ #arg * (#arg, "type") ↦ "lit"     ⎬
⎩         * (#arg, "val") ↦ #v * types(#v:Num)          ⎭
                    evalExpr(store, e)
                    { PRE * ret = -v }
```



ρ store : #s e : #e
t = [e, "type"]
ρ store : #s e : #e t : #t
h #e "type" : #t
goto [t = "lit"] lit r1
ρ store : #s e : #e t : #t
h #e "type" : #t
π #t != "lit"
goto [t = "unop"] uo r2
ρ store : #s e : #e t : #t
h #e "type" : #t
π #t != "lit" #t = "unop"
arg := [e, "arg"]
ρ ... e : #e t : #t arg : #arg
h #e "type" : #t "arg" : #arg
π #t != "lit" #t = "unop"
arg_v := "evalExpr" (store, arg)
ρ ... t : #t arg : #arg arg_v : #v
h #e "type" : #t "arg" : #arg
#arg "type" : "lit" "val" : #v
π #t != "lit" #t = "unop"
op := [e, "op"]
ρ ... t : #t arg : #arg arg_v : #v op: #op
h #e "type" : #t "arg" : #arg "op" : #op
#arg "type" : "lit" "val" : #v
π #t != "lit" #t = "unop"
ret := "evalUnop" (op, arg_v)
ρ ... arg_v : #v op: #op ret : -#v
h #e "type" : #t "arg" : #arg "op" : #op
#arg "type" : "lit" "val" : #v
π #t != "lit" #t = "unop"
types(#v : Num) #op = "-"

Fig. 17. Bi-abductive Symbolic Execution

The bi-abductive symbolic execution discussed above generates a success spec. The value of success specs lies in the fact that they can be converted into concrete/symbolic tests [Fragoso Santos et al. 2018a], effectively providing the user with a comprehensive concrete/symbolic test suite for the analysed program and bringing immediate value to the software development process.

Soundness Theorem. The *Bi-abductive Soundness Theorem* (Theorem 5.1) connects successful bi-abductive executions to successful parameter-state executions. Concretely, given a bi-abductive execution $\langle p, (\Sigma, \Sigma_{af}), \mathsf{cs}, i\rangle^{\mu} \leadsto^{*}_{\mathsf{bi}} \langle p', (\Sigma', \Sigma'_{af}), \mathsf{cs}', j\rangle^{\mu'}$, we construct a parameter-state execution by extending the initial state Σ with the missing resource computed during that execution, Σ''_{af} (we denote state extension by $\circ$). Informally, the state Σ''_{af} corresponds to the initial anti-frame Σ_{af} subtracted from the final one, Σ'_{af}.

Theorem 5.1 (Bi-abductive Soundness Theorem).

$$\langle p, (\Sigma, \Sigma_{af}), \mathsf{cs}, i\rangle^{\mu} \leadsto^{*}_{bi} \langle p', (\Sigma', \Sigma'_{af}), \mathsf{cs}', j\rangle^{\mu'} \wedge \mu' \neq \mathsf{F}(\xi) \implies$$
$$\exists \Sigma''_{af}.\ \Sigma'_{af} = \Sigma_{af} \circ \Sigma''_{af} \wedge \langle p, \Sigma \circ \Sigma''_{af}, \mathsf{cs}, i\rangle^{\mu} \leadsto^{*} \langle p', \Sigma', \mathsf{cs}', j\rangle^{\mu'}$$

By plugging the symbolic state instantiation into the Specification Module, the resulting state instantiation into the Bi-abduction Module, and the resulting state instantiation into the Semantics Module, we obtain a symbolic semantics with support for abstract execution of procedure calls and automatic inference of resource. We denote this semantics relation by $\leadsto_{\mathsf{bi}\,(\mathsf{spec(s)})}$. We can use this relation to bi-abduce SL-specifications for a procedure $f(\overline{x})$ up to a given bound as follows:

(1) We construct an initial bi-abductive configuration $\langle p, (\hat{\sigma}_0, \emptyset), \hat{cs}_0, i\rangle$, where $\hat{\sigma}_0 = (\hat{\rho}_0, \emptyset, \emptyset, \emptyset, \mathsf{true})$, $\hat{\rho}_0 = [x_i \mapsto \hat{x}_i|_{i=0}^{n}]$, and $\hat{cs}_0 = [(\mathsf{main}, [], \emptyset, \mathtt{ret}, 0, 0)]$. Observe that the initial store $\hat{\rho}_0$ maps each parameter of f, x_i, to a freshly generated logical variable $\hat{x}_i$.
(2) We execute the procedure f using $\leadsto_{\mathsf{bi}\,(\mathsf{spec(s)})}$.
(3) For each execution trace $\langle p, (\hat{\sigma}_0, \emptyset), cs_0, i\rangle^{\mathsf{C}} \leadsto^{*}_{\mathsf{bi}\,(\mathsf{spec(s)})} \langle p, (\hat{\sigma}', \hat{\sigma}_{af}), \hat{cs}_0, j\rangle^{\mathsf{N}(\hat{v})}$ terminating in normal mode, we construct a specification $\mathsf{nm} : \{\hat{\sigma}_0 \circ \hat{\sigma}_{af}\}\, f(\overline{x})\, \{\hat{\sigma}' \wedge \mathtt{ret} = \hat{v}\}$. Analogously, for each execution trace $\langle p, (\hat{\sigma}_0, \emptyset), cs_0, i\rangle^{\mathsf{C}} \leadsto^{*}_{\mathsf{bi}\,(\mathsf{spec(s)})} \langle p, (\hat{\sigma}', \hat{\sigma}_{af}), \hat{cs}_0, j\rangle^{\mathsf{E}(\hat{v})}$ terminating in error mode, we construct a specification $\mathsf{er} : \{\hat{\sigma}_0 \circ \hat{\sigma}_{af}\}\, f(\overline{x})\, \{\hat{\sigma}' \wedge \mathtt{ret} = \hat{v}\}$.

6 EVALUATION

We evaluate the three styles of analysis JaVerT 2.0 supports: whole-program symbolic testing, verification, and automatic compositional testing, focussing on a number of simple data-structure libraries. The results demonstrate scalability of whole-program symbolic testing, an improvement over our previous work on JS verification, and creation of useful specifications using bi-abduction, minimising the annotation burden of the developer. In addition, we perform whole-program symbolic testing of the real-world Buckets.js [Santos 2016] data structure library, which has over 65K downloads on npm [npm, Inc. 2018]. We reproduce our previously reported bugs [Fragoso Santos et al. 2018a], but also discover a new one, in times that are almost two orders of magnitude faster, suggesting scalability of our whole-program symbolic testing to much larger codebases. As in [Fragoso Santos et al. 2018a,b], we perform the entire evaluation on a machine with an Intel Core i7-4980HQ CPU 2.80 GHz, DDR3 RAM 16GB, and a 256GB solid-state hard-drive running OSX.

JS-2-JSIL Coverage and Correctness. Before we proceed to the evaluation, we briefly discuss the coverage and correctness of the JS-2-JSIL compiler used in JaVerT 2.0. JS-2-JSIL has full support for the entire core of ES5 Strict, as well as most of the associated built-in libraries. We do not implement the Date, RegExp, and JSON libraries, which are orthogonal to the core. As discussed in §3, JSIL has all of the constructs required to support full ES5; that extension is technical in nature. Additional constructs are likely to be needed to support ES6. We test JS-2-JSIL against the official ECMAScript test suite, Test262 [ECMA TC39 2017], using both the concrete and the symbolic JSIL interpreters.[4] Out of the 10469 tests for ES5 Strict, we identify 8797 tests appropriate for our

[4]We test the symbolic JSIL interpreter to ensure that it behaves the same as the concrete interpreter on Test262.

Table 2. Verification Results

Name	JS loc/ JSIL loc	Funcs	Specs	Spec Chars	Pred Chars	Executed JSIL cmds	LCmds (POPL'18)	**LCmds**	Time (POPL'18)	**Time**
`BST`	69/877	5	5	1,573	438	1,577	19	**0**	7.38s	**7.41s**
`DLL`	14/373	3	3	807	225	705	N/A	**0**	N/A	**3.27s**
`ExprEval`	25/557	3	7	1,196	1,814	1,540	N/A	**0**	N/A	**4.95s**
`IDGen`	16/289	4	4	642	276	364	0	**0**	0.73s	**1.28s**
`KV-Map`	23/427	4	9	899	1,812	1,583	8	**6**	3.37s	**5.54s**
`PriQ`	42/789	7	10	2,373	1,241	1,467	5	**0**	7.14s	**8.62s**
`SLL`	12/262	3	3	807	225	550	N/A	**0**	N/A	**2.48s**
`Sort`	22/319	2	2	845	279	556	6	**0**	1.78s	**2.16s**
`Test262`	113/1237	7	16	620	0	1,844	0	**0**	3.46s	**3.63s**

coverage, of which we pass 100%. We note that we execute `eval` statements concretely, but do not reason about them symbolically. Such analyses, as well as many similar ones (for example, those reasoning about regular expressions) could be incorporated on top of JaVerT 2.0 via the external procedure call mechanism of JSIL. This highlights the modularity of our framework.

6.1 Case Studies

We evaluate JaVerT 2.0 on simple data-structure libraries: singly- and doubly-linked lists, binary search trees, key-value maps, priority queues, and sorted lists. We also include the expression evaluator, an identifier generator, and several examples taken from the Test262 test suite.

Results: Verification. We fully specify and verify the case studies, with the results shown in Table 2. For each case study, we give the number of: JS/JSIL lines of the implementation; associated functions; specifications; characters of the specifications and supporting predicates; executed JSIL commands; and required logical commands for JaVerT [Fragoso Santos et al. 2018b] and for JaVerT 2.0. Finally, we contrast the JaVerT and JaVerT 2.0 times.

The results show two tangible improvements over JaVerT. First, all examples except KV-Map no longer require any logical commands, with the KV-Map needing fewer. Second, even though the fold/unfold mechanism is automatic in JaVerT 2.0, the obtained times are comparable. On the other hand, these results once again affirm that verification comes at a price. The required predicate bootstrap and the specifications are intricate and lengthy (cf. Table 2: spec chars, pred chars), requiring developers to be well-versed in both JavaScript semantics and program logic.

Results: Whole-Program Symbolic Testing. We write symbolic tests that achieve full line coverage for each of the case studies. The results are given in Table 3 (left). We report the obtained times, together with the number of: tests; characters of the tests; and executed JSIL commands. What is immediately evident is that, when compared to verification, the number of executed commands is substantially higher, whereas the times are shorter. The former is due to the fact that whole-program symbolic testing does not use summaries and every procedure is executed on each call. This also contributes to the latter, together with the fact that there is no predicate manipulation. We highlight that our entailment engine is able to discharge the majority of queries for whole-program symbolic testing, minimising solver time to the point where it is negligible. This is partly what contributes to its speed and scalability, further demonstrated in §6.2.

The symbolic tests of JaVerT 2.0 are useful for debugging JS code and are more intuitive for developers to write than specifications. In particular, Tables 2 and 3 (left) show that fewer characters are required for symbolic tests with full line coverage than for the verification bootstrap and specifications. In addition, it took us less time to write the symbolic tests than it did to write the specs. On the other hand, in the general case, symbolic testing does not offer full correctness guarantees.

Table 3. Results for whole-program symbolic testing (left) and bi-abduction (right)

Name	Tests	Test Chars	Executed JSIL cmds	**Time**
`BST`	6	1,663	86,405	**2.21s**
`DLL`	6	625	19,975	**0.54s**
`ExprEval`	10	1,568	46,664	**2.17s**
`IdGen`	3	242	6,070	**0.29s**
`KV-Map`	5	539	56,494	**1.65s**
`PriQ`	5	797	38,661	**0.99s**
`SLL`	6	559	17,759	**0.49s**
`Sort`	6	675	22,693	**0.73s**

Name	S/E/B specs	**Time**	Hints	S/E/B specs	**Time**
`BST`	27/0/5	**11.46s**	7	27/0/0	**9.48s**
`DLL`	10/0/3	**2.38s**	3	10/0/0	**2.25s**
`ExprEval`	269/142/3	**82.35s**	8	71/54/1	**38.35s**
`IDGen`	4/0/0	**0.37s**	1	3/0/0	**0.35s**
`KV-Map`	9/12/10	**3.75s**	6	4/2/0	**1.81s**
`PriQ`	15/1/12	**7.45s**	8	15/1/0	**6.15s**
`SLL`	6/0/2	**1.35s**	2	6/0/0	**1.43s**
`Sort`	8/0/2	**2.64s**	4	8/0/0	**1.91s**

Results: Automatic Compositional Testing. We automatically construct specifications for the case studies using bi-abduction. As in whole-program symbolic testing, these specifications capture correctness up to a bound. In the left-hand part of Table 3 (right), we give the number of success, error, and bug specs (S/E/B specs) created by fully automatic bi-abduction, as well as the obtained times. We remind the reader that these specifications describe a superset of the intended behaviours of the function, and may contain false positive bug reports that need to be examined by the developer or filtered via a separate automated phase.

The obtained times reflect the substantial internal branching of the JavaScript semantics; in particular, the numerous implicit coercions may cause a state search explosion, as observed in the ExprEval example. As stated in §2, to contain this, the developer can give hints to indicate how the function is to be used. These hints are extremely simple and amount to stating that a given variable or object property has a given type. In the right-hand part of Table 3 (right), we give the number of hints required to isolate the intended behaviours of our examples, the number of the obtained specs (which are now all applicable), and the obtained times, which are, in most cases, noticeably faster.

We believe that these results constitute a solid first attempt at bi-abduction for real-world dynamic languages. We see great room for improvement w.r.t. the speed, by minimising the footprint of the generated specs and containing the JS semantics in a principled way via executable specs.

6.2 Whole-Program Symbolic Testing of Real-World Libraries: Buckets.js

We analysed the Buckets.js data structure library of Santos [2016], widely used by JavaScript (JS) developers, with more than 65K downloads on npm [npm, Inc. 2018]. This library is ideal for our analysis, as it uses almost all essential JS features, has an associated unit test suite, and does not have any external dependencies.

We reuse the Buckets.js symbolic test suite of Fragoso Santos et al. [2018a], which has 100% line coverage. We successfully reproduce the bugs reported therein and discover an additional bug, discussed shortly. We present our results in Table 4. For each data structure in Buckets.js, we report: **(1)** the number of JS/JSIL lines of the implementation;

Table 4. Whole-program symbolic testing of the Buckets.js library

Name	JS loc/ JSIL loc	Test info (#/sloc/cloc)	Executed JSIL cmds	Time (PPDP'18)	Time
`array`	71/1942	**9/166**/329	214,219	181.36s	**5.02s**
`bag`	237/7194	**7/78**/265	831,017	520.72s	**10.50s**
`bst`	326/8052	**11/216**/759	2,436,519	3605.98s	**18.67s**
`dict`	84/2374	**7/116**/170	247,952	109.13s	**4.25s**
`heap`	128/4001	**4/92**/626	1,005,906	1318.37s	**5.70s**
`llist`	153/3138	**9/149**/370	373,247	216.21s	**7.56s**
`mdict`	184/5496	**6/118**/189	686,171	527.17s	**8.94s**
`queue`	183/4233	**6/111**/146	246,470	124.02s	**4.58s**
`pqueue`	154/5067	**5/70**/283	1,469,179	1749.15s	**8.60s**
`set`	124/3902	**6/86**/271	1,311,103	380.74s	**16.49s**
`stack`	176/4079	**4/91**/104	188,132	105.65s	**3.28s**
`Total`	N/A	74/1293/3512	9,010,615	8853.90s	**93.59s**

(2) the number of symbolic tests, their number of lines, and the number of lines of the corresponding concrete tests shipped with the library; **(3)** the total number of JSIL commands executed in the symbolic tests; **(4)** the times reported by Fragoso Santos et al. [2018a]; and **(5)** our obtained times.

The data highlights a tangible benefit of symbolic over concrete testing: developers can obtain full coverage with fewer lines of code. We focus on our obtained times, almost two orders of magnitude faster than those of Fragoso Santos et al. [2018a]. These times indicate the maturity of our symbolic testing framework and the scalability of the analysis—effectively, we were able to symbolically analyse a real-world JS library containing 1.3K lines of code with 100% line coverage, fully adhering to the semantics of the language, running millions of JSIL commands, in one and a half minutes.

Bug: Linked lists. We discovered a bug in the implementation of the Buckets.js linked list library. In particular, our symbolic tests for the `ElementAtIndex(n)` and `removeElementAtIndex(n)` functions, which, respectively, inspect and remove the *n*-th element of the list, revealed unexpected behaviour when `n` was a positive, non-integer number. In practice, the $\lceil n \rceil$-th element was returned, instead of an error message indicating improper indexation. This bug was left undetected by the concrete unit tests of Buckets.js because of their incomplete coverage, and by our symbolic testing done in Fragoso Santos et al. [2018a] because that analysis relied on integer, rather than real arithmetic.

7 RELATED WORK

The literature covers a wide range of verification tools based on whole-program symbolic analysis [Boyapati et al. 2002; Claessen et al. 2015; Claessen and Hughes 2000; Dolby et al. 2007; Milicevic et al. 2007; Runciman et al. 2008; Seidel et al. 2015], and separation logic (SL) tools [Berdine et al. 2005; Calcagno et al. 2015, 2011; Distefano and Parkinson 2008; Jacobs et al. 2011; Yang et al. 2008], mainly aimed at static languages. We first describe existing work on symbolic execution and logic-based verification for JavaScript. We then describe the analysis techniques most related to JaVerT 2.0: bi-abduction in the context of SL and incremental techniques for bounded model checking (BMC) and symbolic execution (SE).

This paper is strongly influenced by our previous work on JS verification (JaVerT) [Fragoso Santos et al. 2018b] and symbolic execution (Cosette) Fragoso Santos et al. [2018a]. JaVerT and Cosette share part of their infrastructure, but have different purposes. JaVerT is the first verification toolchain for dynamic languages based on SL. Cosette is a framework for whole-program symbolic testing of JS programs, also used for specification-driven bug-finding. JaVerT 2.0 unifies and significantly improves these analyses. It improves JaVerT by providing built-in support for automatic unfold/fold reasoning over user-defined inductive predicates. It improves Cosette by providing a native implementation of a symbolic execution for JSIL which is two orders of magnitude more performant than the original Cosette implementation. Finally, JaVerT 2.0 is the first tool to support fully automatic compositional testing based on bi-abduction for dynamic languages.

Symbolic Execution for JavaScript. The majority of the existing bug-finding symbolic execution tools for JavaScript are whole-program and target specific bug patterns, such as security vulnerabilities related to the misuse of strings [Saxena et al. 2010], malformed Web API requests [Wittern et al. 2017], and DOM API specific bugs [Li et al. 2014]. These tools are fully automatic and aim at code in the large, primarily focusing on scalability and coverage issues. The work closest to ours is *Jalangi* [Sen et al. 2015], a general-purpose symbolic execution tool for JavaScript, which, for scalability reasons, does not follow the semantics of the language precisely. In contrast, JaVerT 2.0 is *trustworthy*: it does follow the semantics of JavaScript and its theoretical underpinnings are formalised and proven sound, which allows us to use its symbolic execution engine both for testing and verification. Moreover, JaVerT 2.0 is not limited to targeting a specific category of bugs—its bug-finding aspect is general.

Logic-based Verification for JavaScript. KJS [Ştefănescu et al. 2016] is a symbolic verification tool for core ES5 obtained by instantiating the general $\mathbb{K}$ framework with the semantics of JavaScript [Park et al. 2015]. Like JaVerT 2.0, KJS can be used to verify functional correctness properties of small data structure libraries. However, KJS specifications are complex and error-prone, explicitly exposing all language internals. Furthermore, KJS is not compositional, as it does not allow partial descriptions of JavaScript objects. Finally, KJS does not implement any form of error reporting, making verification bugs extremely difficult to pinpoint and correct.

Swamy et al. [2013] use the Dijkstra monad and its type inference algorithm to prove the absence of runtime errors for a small fragment of JavaScript (ES3). Concretely, they translate that fragment to F* and generate verification conditions that check for the absence of runtime errors. JaVerT 2.0 aims at similar errors, but supports a large, rigorously defined subset of JavaScript (ES5).

Both [Ştefănescu et al. 2016] and [Swamy et al. 2013] provide strong correctness guarantees, but have scalability limitations and their tools have not been applied to real-world code. For instance, the Buckets.js library would be out of their reach, as they do not come with abstractions to accurately describe, e.g. JS arrays, for-in loop invariants, and higher-order functions. For fully automatic and whole-program symbolic testing, JaVerT 2.0 does not require such abstractions, and can, therefore, be used for analysing substantially larger, more complex codebases.

Bi-abduction. Introduced by Calcagno et al. [2009b], bi-abduction was initially used to compute lightweight specifications for heap-manipulating programs operating on standard data structures, such as lists. It was later implemented as part of Infer [Calcagno and Distefano 2011; Calcagno et al. 2015], an industrial-strength bug-finding tool targeting memory safety of C, Java, Objective C, and C++ programs. Infer attempts to build a compositional proof of a given program by joining together the proofs of its constituents. Failed proof attempts are turned into bug reports and presented to the user. Infer has some limitations with respect to proof generation: for example, it can only *Infer* [sic] loop invariants involving a restricted number of data structures, such as lists and list segments. This, however, is of little practical consequence as Infer is mainly used as a bug-finding tool.

Additionally, bi-abduction has been applied to different forms of program reasoning, including: automatic verification of concurrent systems [Calcagno et al. 2009a]; inference of pure information about weakly-specified data structures, such as size, sum or height [Trinh et al. 2013]; and automatic synthesis of higher-order predicates for polymorphic data structures [Le et al. 2014].

All the aforementioned works in the literature describe bi-abduction in the context of simple imperative languages. We are the first to study the use of this technique in the context of a real-world highly complex dynamic language, such as JavaScript. Furthermore, we are the first to define the bi-abductive mechanism in terms of the error reporting mechanism of the underlying analysis, streamlining its implementation and meta-theoretical results.

Bounded Model Checking. Bounded Model Checking (BMC) is a program analysis technique whose main purpose is bug-finding. Introduced by Biere et al. [1999] and widely applied successfully since by, for example [Cho et al. 2013; Clarke et al. 2004; Lal et al. 2012; Sinha et al. 2012], BMC normally works by first unrolling loops up to a given bound *syntactically*, then compiling the unrolled program to first-order constraints, and, finally, passing these constraints to an SMT solver, which checks their satisfiability. JaVerT 2.0 also unrolls loops up to a given bound for whole-program symbolic testing and automatic compositional testing, albeit not syntactically.

A BMC analysis is whole-program as a rule, in the style of our whole-program symbolic testing. To our knowledge, the only BMC work that considers incrementality is that of Cho et al. [2013], in which the authors introduced Blitz, a scalable bounded model checker for C code. Blitz scales because, unlike other BMC-based tools, it analyses the program bottom-up, starting from the functions in which there are assertions to be verified, rather than starting from the `main` function. For each such function f, Blitz first constructs violation pre-conditions: that is, first-order summaries under which the assertion is violated. Then, it analyses the callers of f and attempts to construct their pre-conditions for which the violation pre-condition of f would hold at the call site of f. This process is then repeated further up the call graph until the `main` function is reached. In this way, Blitz is able to construct BMC instances incrementally. This is similar to the way in which our bi-abduction specifications propagate up the call graph.

Symbolic Execution. Symbolic execution tools can be broadly divided into two main groups: *static* and *dynamic*. Static SE tools, such as [Anand et al. 2007, 2009; Khurshid et al. 2003; Torlak and Bodík 2014], explore the entire symbolic execution tree up to a pre-established bound, providing bounded verification guarantees in the style of BMC and our Theorem 3.2. Dynamic SE tools, such as [Cadar et al. 2008a,b; Godefroid 2007; Godefroid et al. 2005, 2008, 2010; Ramos and Engler 2015], pioneered by dart [Godefroid et al. 2005], normally work by pairing up a concrete execution with a symbolic execution in order to allow the symbolic execution to fall back to the concrete execution whenever it produces symbolic formulae that are not supported by the underlying constraint solver. Dynamic SE tools are mainly aimed at automatic test generation and generally do not provide any verification guarantees. There is a vast corpus of research on both static and dynamic SE tools, see [Cadar et al. 2011] and [Cadar and Sen 2013] for comprehensive surveys on the topic. Here, we describe the use of summaries and automatic inference of resource in both approaches.

- *Summaries:* smart [Godefroid 2007] was the first dynamic SE tool with support for summaries. It tests functions in isolation in a bottom-up manner, encodes test results as first-order constraints and re-uses these constraints as summaries in the testing of other functions. smash [Godefroid et al. 2010] is a unified SE framework for testing and verification of C programs. Like smart, smash is incremental: it analyses one function at a time, building first-order verification and testing summaries that can be later used in the analysis of other functions. Its key innovation is the tight integration of verification and testing summaries, allowing the SE engine to use both types of summary in a demand-driven manner. In both tools, summaries consist of first-order formulae relating inputs to outputs and do not describe the heap. Furthermore, they are mainly used as a mechanism for improving performance in the context of whole-program analysis, and are not used by either tool to report errors for functions in isolation.

- *Inference of resource:* [Khurshid et al. 2003] were the first to propose *lazy initialisation* as a means of supporting dynamically allocated data structures in Java programs during SE, without requiring an *a priori* bound on input sizes. Lazy initialisation works by initialising heap resource on an "as-needed" basis in a similar style to our bi-abductive module. Lazy initialisation has later been applied to static/dynamic SE of C and Java programs [Deng et al. 2012, 2007; Sen and Agha 2006] with more general data types including references and arrays.
  Further refinements of lazy initialisation include: *under-constrained symbolic execution* [Engler and Dunbar 2007] and *bounded lazy initialisation* [Geldenhuys et al. 2013; Rosner et al. 2015]. Both techniques aim at reducing the number of spurious errors reported to the user due to lazily initialised values. Under-constrained SE keeps track of all lazily initialised values during SE and only reports a bug involving these values if it can prove that the bug occurs for every possible concretisation of all lazily initialised values. Bounded lazy initialisation improves on lazy initialisation by taking advantage of precomputed relational bounds on the interpretation of object fields to reduce the number of explored spurious structures at SE time.
  Recently, [Ramos and Engler 2015] have created uc-klee, an extension of klee [Cadar et al. 2008a] with support for under-constrained SE, which contains an interface for users to manually silence spurious errors by lazily specifying input preconditions using simple C code. This technique is similar to our use of hints for the bi-abductive analysis in the form of simple *assume* statements.

8 CONCLUSIONS AND FURTHER WORK

We have introduced a unified approach for developing compositional symbolic execution tools for JavaScript, marrying classical symbolic execution and compositional program reasoning based on separation logic. The key insights of this approach are: **(1)** a modular design of the analysis framework, leading to a streamlined formalism with clear meta-theoretical results, strongly connected to the implementation; **(2)** an explicit treatment of execution errors, enabling meaningful error reporting and driving the bi-abductive inference of resource; and **(3)** a mechanism for re-using separation-logic specifications within symbolic execution. Using this approach, we have created JaVerT 2.0, a trustworthy JavaScript verification and testing framework that supports whole-program symbolic testing, verification, and automatic compositional testing. We have successfully evaluated the range of JaVerT 2.0 analyses on a number of simple data-structure libraries. We have also used whole-program symbolic testing to detect bugs in real-world JavaScript code.

There are numerous avenues for further work. Examples include: extending the coverage of JaVerT 2.0 to full ES5 and/or more recent versions of the standard; extending JSIL logic with support for higher-order reasoning; designing a heuristic mechanism for invariant synthesis; and improving our underlying first-order solver with more advanced reasoning capabilities for strings, sets, multisets, maps, and other mathematical structures. In the near future, we plan to:

- improve our error reporting in the context of verification by developing an interactive error diagnosis mechanism, possibly using ideas from Dillig et al. [2012];
- automatically convert the bi-abduced specifications to a concrete test suite for the given program, which the user could then run in their JS engine of choice to corroborate the results of JaVerT 2.0;
- improve the performance of the bi-abductive analysis by: using well-crafted heuristics in the inference of missing resource, simplifying the generated specifications, and containing the JS semantics in a principled way via executable specifications.

Finally, we will investigate the generalisation of the meta-theory of JaVerT 2.0 by making it additionally parametric on the memory model of the targeted language. In this way, we would be able to move beyond JavaScript/JSIL to languages such as C, Java, or WebAssembly.

ACKNOWLEDGMENTS

We would like to thank the reviewers, whose comments have improved the overall quality of the paper. We would also like to thank Emma Tye and Sacha-Élie Ayoun, whose work during their UROP and MSc projects improved the infrastructure of JaVerT 2.0.

Fragoso Santos, Gardner, and Maksimović were supported by the EPSRC Programme Grant 'REMS: Rigorous Engineering for Mainstream Systems' (EP/K008528/1). 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. Sampaio was supported by a CAPES Foundation Scholarship, process number 88881.129599/2016-01.

REFERENCES

S. Anand, C. S. Păsăreanu, and W. Visser. 2007. JPF–SE: A Symbolic Execution Extension to Java PathFinder. In *TACAS*. 134–138.

S. Anand, C. S. Păsăreanu, and W. Visser. 2009. Symbolic Execution with Abstraction. *International Journal on Software Tools for Technology Transfer* 11, 1 (2009), 53–67.

R. Baldoni, E. Coppa, D. Cono D'Elia, C. Demetrescu, and I. Finocchi. 2018. A Survey of Symbolic Execution Techniques. *ACM Computing Surveys* 51, 3 (2018), 50:1–50:39.

J. Berdine, C. Calcagno, and P. W. O'Hearn. 2005. Smallfoot: Modular Automatic Assertion Checking with Separation Logic. In *FMCO*. 115–137.

A. Biere, A. Cimatti, E. Clarke, and Y. Zhu. 1999. Symbolic Model Checking without BDDs. In *TACAS*. 193–207.

S. Blazy, V. Laporte, and D. Pichardie. 2016. An Abstract Memory Functor for Verified C Static Analyzers. In *ACM SIGPLAN Notices*, Vol. 51. 325–337.

C. Boyapati, S. Khurshid, and D. Marinov. 2002. Korat: Automated Testing based on Java Predicates. In *ISSTA*. 123–133.

C. Cadar, D. Dunbar, and D. R. Engler. 2008a. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs.. In *OSDI*, Vol. 8. 209–224.

C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler. 2008b. EXE: Automatically Generating Inputs of Death. *ACM Transactions on Information and System Security* 12, 2 (2008), 10:1–10:38.

C. Cadar, P. Godefroid, S. Khurshid, C. S. Păsăreanu, K. Sen, N. Tillmann, and W. Visser. 2011. Symbolic Execution for Software Testing in Practice: Preliminary Assessment. In *ICSE*. 1066–1071.

C. Cadar and K. Sen. 2013. Symbolic Execution for Software Testing: Three Decades Later. *Commun. ACM* 56 (2013), 82–90.

C. Calcagno and D. Distefano. 2011. Infer: An Automatic Program Verifier for Memory Safety of C Programs. In *NASA Formal Methods Symposium*. 459–465.

C. Calcagno, D. Distefano, J. Dubreil, D. Gabi, P. Hooimeijer, M. Luca, P. W. O'Hearn, I. Papakonstantinou, J. Purbrick, and D. Rodriguez. 2015. Moving Fast with Software Verification. In *NASA Formal Methods Symposium*. 3–11.

C. Calcagno, D. Distefano, P. W. O'Hearn, and H. Yang. 2009b. Compositional Shape Analysis by Means of Bi-Abduction. In *POPL*. 289–300.

C. Calcagno, D. Distefano, P. W. O'Hearn, and H. Yang. 2011. Compositional Shape Analysis by Means of Bi-Abduction. *JACM* 58 (2011), 26:1–26:66.

C. Calcagno, D. Distefano, and V. Vafeiadis. 2009a. Bi-abductive Resource Invariant Synthesis. In *APLAS*. 259–274.

C. Calcagno, P. W. O'Hearn, and H. Yang. 2007. Local Action and Abstract Separation Logic. In *LICS*. IEEE Computer Society, 366–378.

C. Y. Cho, V. D'Silva, and D. Song. 2013. BLITZ: Compositional Bounded Model Checking for Real-world Programs. In *ASE*. 136–146.

K. Claessen, J. Duregård, and M. H. Palka. 2015. Generating Constrained Eandom Data with Uniform Distribution. *J. Funct. Program.* 25 (2015).

K. Claessen and J. Hughes. 2000. QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs. In *ICFP*. 268–279.

E. Clarke, D. Kroening, and F. Lerda. 2004. A Tool for Checking ANSI-C Programs. In *TACAS*. 168–176.

A. Ştefănescu, D. Park, S. Yuwen, Y. Li, and G. Roşu. 2016. Semantics-Based Program Verifiers for All Languages. In *OOPSLA*. 74–91.

D. Darais, M. Might, and D. Van Horn. 2015. Galois Transformers and Modular Abstract Interpreters: Reusable Metatheory for Program Analysis. *ACM SIGPLAN Notices* 50 (2015), 552–571.

L. De Moura and N. Bjørner. 2008. Z3: An Efficient SMT Solver. In *TACAS*. 337–340.

X. Deng, J. Lee, and Robby. 2012. Efficient and Formal Generalized Symbolic Execution. *ASE* 19 (2012), 233–301.

X. Deng, Robby, and J. Hatcliff. 2007. Towards A Case-Optimal Symbolic Execution Algorithm for Analyzing Strong Properties of Object-Oriented Programs. In *SEFM*. 273–282.

I. Dillig, T. Dillig, and A. Aiken. 2012. Automated Error Diagnosis Using Abductive Inference. *PLDI* 47 (2012), 181–192.

T. Dinsdale-Young, L. Birkedal, P. Gardner, M. J. Parkinson, and H. Yang. 2013. Views: Compositional Reasoning for Concurrent Programs. In *POPL*. 287–300.

D. Distefano and M. Parkinson. 2008. jStar: Towards Practical Verification for Java. In *OOPSLA*, Vol. 43. 213–226.

J. Dolby, M. Vaziri, and F. Tip. 2007. Finding Bugs Efficiently with a SAT Solver. In *FSE*. 195–204.

ECMA TC39. 2011. *The 5th Edition of the ECMAScript Language Specification.* Technical Report. ECMA.

ECMA TC39. 2017. Test262 Test Suite. https://github.com/tc39/test262. (2017).

D. Engler and D. Dunbar. 2007. Under-constrained Execution: Making Automatic Code Destruction Easy and Scalable. In *ISSTA*. ACM, 1–4.

J. Fragoso Santos, P. Maksimović, T. Grohens, J. Dolby, and P. Gardner. 2018a. Symbolic Execution for JavaScript. In *PPDP*. 11:1–11:14.

J. Fragoso Santos, P. Maksimović, D. Naudžiūnienė, T. Wood, and P. Gardner. 2018b. JaVerT: JavaScript Verification Toolchain. *PACMPL* 2, POPL (2018), 50:1–50:33.

P. Gardner, S. Maffeis, and G. Smith. 2012. Towards a Program Logic for JavaScript. In *POPL*. 31–44.

P. Gardner, G. Smith, M. J. Wheelhouse, and U. Zarfaty. 2008. Local Hoare Reasoning about DOM. In *PODS*. ACM, 261–270.

J. Geldenhuys, N. Aguirre, M. F. Frias, and W. Visser. 2013. Bounded Lazy Initialization. In *NASA Formal Methods*. 229–243.

P. Godefroid. 2007. Compositional Dynamic Test Generation. In *POPL*, Vol. 42. 47–54.

P. Godefroid, N. Klarlund, and K. Sen. 2005. DART: Directed Automated Random Testing. In *ACM Sigplan Notices*, Vol. 40. 213–223.

P. Godefroid, M. Y. Levin, and D. A. Molnar. 2008. Automated Whitebox Fuzz Testing. In *NDSS*, Vol. 8. 151–166.

P. Godefroid, A. V. Nori, S. K. Rajamani, and S. Tetali. 2010. Compositional May-must Program Analysis: Unleashing the Power of Alternation. In *POPL*. 43–56.

M. Harman and P. W. O'Hearn. 2018. From Start-ups to Scale-ups: Opportunities and Open Problems for Static and Dynamic Program Analysis. In *SCAM*.

D. Van Horn and M. Might. 2010. Abstracting Abstract Machines. In *ICFP*. 51–62.

S. S. Ishtiaq and P. W O'Hearn. 2001. BI as an assertion language for mutable data structures. *POPL* 36 (2001), 14–26.

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*. 41–55.

S. Khurshid, C. S. Păsăreanu, and W. Visser. 2003. Generalized Symbolic Execution for Model Checking and Testing. In *TACAS*. 553–568.

A. Lal, S. Qadeer, and S. K. Lahiri. 2012. A Solver for Reachability Modulo Theories. In *CAV*. 427–443.

Q. L. Le, C. Gherghina, S. Qin, and W.-N. Chin. 2014. Shape Analysis via Second-order Bi-abduction. In *CAV*. 52–68.

G. Li, E. Andreasen, and I. Ghosh. 2014. SymJS: automatic symbolic testing of JavaScript web applications. In *FSE*. 449–459.

A. Milicevic, S. Misailovic, D. Marinov, and S. Khurshid. 2007. Korat: A Tool for Generating Structurally Complex Test Inputs. In *ICSE*. 771–774.

H. H. Nguyen, C. David, S. Qin, and W.-N. Chin. 2007. Automated Verification of Shape and Size Properties via Separation Logic. In *VMCAI*. 251–266.

H. H. Nguyen, V. Kuncak, and W.-N. Chin. 2008. Runtime Checking for Separation Logic. In *VMCAI*. 203–217.

npm, Inc. 2018. npm, a Package Manager for JavaScript. https://www.npmjs.com. (2018).

P. W. O'Hearn. 2018. Continuous Reasoning: Scaling the Impact of Formal Methods. In *LICS*. 13–25.

D. Park, A. Stefănescu, and G. Roşu. 2015. KJS: A Complete Formal Semantics of JavaScript. In *PLDI*. 346–356.

D. A. Ramos and D. R. Engler. 2015. Under-Constrained Symbolic Execution: Correctness Checking for Real Code. In *USENIX Security Symposium*. 49–64.

J. C. Reynolds. 2002. Separation Logic: A Logic for Shared Mutable Data Structures. In *LICS*. 55–74.

N. Rosner, J. Geldenhuys, N. M. Aguirre, W. Visser, and M. F. Frias. 2015. BLISS: Improved Symbolic Execution by Bounded Lazy Initialization with SAT Support. *IEEE Transactions on Software Engineering* 41 (2015), 639–660.

C. Runciman, M. Naylor, and F. Lindblad. 2008. Smallcheck and Lazy Smallcheck: Automatic Exhaustive Testing for Small Values. In *ICFP*. 37–48.

M. Santos. 2016. Buckets-JS: A JavaScript Data Structure Library. https://github.com/mauriciosantos/Buckets-JS. (2016).

P. Saxena, D. Akhawe, S. Hanna, F. Mao, S. McCamant, and D. Song. 2010. A Symbolic Execution Framework for JavaScript. In *IEEE S&P*. 513–528.

E. L. Seidel, N. Vazou, and R. Jhala. 2015. Type Targeted Testing. In *ESOP*. 812–836.

K. Sen and G. Agha. 2006. CUTE and jCUTE: Concolic Unit Testing and Explicit Path Model-Checking Tools. In *CAV*. 419–423.

K. Sen, G. C. Necula, L. Gong, and W. Choi. 2015. MultiSE: Multi-path Symbolic Execution using Value Summaries. In *FSE*. 842–853.

N. Sinha, N. Singhania, S. Chandra, and M. Sridharan. 2012. Alternate and Learn: Finding Witnesses without Looking All Over. In *CAV*. 599–615.

N. Swamy, J. Weinberger, C. Schlesinger, J. Chen, and B. Livshits. 2013. Verifying Higher-order Programs with the Dijkstra Monad. In *PLDI*. 387–398.

E. Torlak and R. Bodík. 2013. Growing Solver-aided Languages with Rosette. In *SPLASH*. 135–152.

E. Torlak and R. Bodík. 2014. A Lightweight Symbolic Virtual Machine for Solver-aided Host Languages. In *PLDI*. 530–541.

M.-T. Trinh, Q. L. Le, C. David, and W.-N. Chin. 2013. Bi-abduction with Pure Properties for Specification Inference. In *APLAS*. 107–123.

E. Wittern, A. T. T. Ying, Y. Zheng, J. Dolby, and J. A. Laredo. 2017. Statically Checking Web API requests in JavaScript. In *ICSE*. 244–254.

H. Yang, O. Lee, J. Berdine, C. Calcagno, B. Cook, D. Distefano, and P. W. O'Hearn. 2008. Scalable Shape Analysis for Systems Code. In *CAV*. 385–398.