# Contract-Driven Testing of JavaScript Code

Phillip Heidegger and Peter Thiemann

Albert-Ludwigs-Universität Freiburg, Germany
{heidegger,thiemann}@informatik.uni-freiburg.de

**Abstract.** JSContest is a tool that enhances JavaScript with simple, type-like contracts and provides a framework for monitoring and guided random testing of programs against these contracts at the same time. Function contracts in JSContest serve a dual role as specifications of the input/output behavior and as test case generators. Generation of test data for a contract is principally random, but can be guided by annotations on the contract to achieve higher coverage. Annotations may indicate dependencies among parameters or between parameters and the result or they may select lightweight program analyses, the results of which influence the choice of test data. A case study substantiates that JSContest finds type-related errors with high probability.

## 1 Introduction

A quality software system should be correct, reliable, and maintainable. Developers try to achieve these qualities for their product by applying well-known software development processes, like the V-model, RAD, RUP, AUP, XP, Scrum, just to name a few. A commonality of all these processes is that they prescribe a substantial amount of testing [26]. On top of that, the processes that emphasize correctness usually include some variant of Meyer's Design by Contract [25].

Nowadays, an increasing number of systems is being developed using scripting languages like Perl, PHP, Ruby, and JavaScript. These languages greatly speed up the development process because of the flexibility gained by their dynamic language features (dynamic typing, weak typing, meta programming, etc). However, this flexibility makes it harder to specify the intended behavior of a system and to demonstrate that the system adheres to a specification. Thus, it is much harder to establish correctness, reliability, and maintainability of such a system.

A promising avenue of related measures to increase the assurance in code in dynamic languages is the introduction of gradual typing [30], of contracts that are monitored at run time [17,31], and of languages that enable the interoperation and the gradual migration of functionality between statically typed and untyped parts of a program [32,24,4]. A common theme of these works is that the statically typed part of the code is shielded from the untyped (or dynamically typed) parts by type or contract annotations. The run-time monitoring of these annotations guarantees that badly composed data from the untyped parts cannot corrupt the statically typed parts of the program.

Our work is inspired by these ideas to make scripting languages safer but takes a different path influenced by work on testing. We would like to help programmers improve the quality of *existing* programs now, so that designing a new language is not an option. We believe that quality software should document and specify its definitions with contracts, hence we developed a typing-inspired contract facility that can be gradually introduced into existing programs. As a software verification is not feasible for an incompletely specified system, we aim for contract monitoring and for automatic contract validation via guided random testing. Furthermore, a verification system rarely produces a counterexample, whereas testing provides one immediately. Programmers find this outcome of a validation more appealing than abstract error messages.

Our target language is JavaScript [12], a language that is widely used for web-based applications. Originally, JavaScript was conceived for writing small scripts that animate web pages (Dynamic HTML) or transfer data to and from Java applets that do the heavy lifting for the application. It is an interpreted language with a weak, dynamic type system, so that all programming errors have to be detected by testing. Nevertheless, Web 2.0 applications contain a substantial part of JavaScript code and there are substantial libraries[1] that support the creation of such applications.

### Contributions

We have developed the tool JSConTest that provides a contract language for JavaScript. The contract language allows to attach software contracts to arbitrary top-level definitions. Contracts can be as simple as type signatures and as complex as the programmer desires. JSConTest creates a test suite from the thus annotated program that addresses the validation of the contracts in two ways: It implements run-time contract monitoring and it performs random testing with input data derived from the contracts. JSConTest minimizes test cases that exhibit program errors using the ideas of delta debugging [33]. The programmer can choose the desired degree of assurance by selectively specifying contracts and by adjusting the precision of the contracts used.

To improve the coverage of random testing, JSConTest can perform guided random testing, which includes results from program analyses in the selection of test data. In many cases, guided random testing achieves coverage similar to concolic testing, but without requiring symbolic evaluation or solving of equations. In addition, JSConTest allows the inclusion of user-specified tests as well as user-specified contracts.

JSConTest does not impose any run-time penalty on program runs outside of the test suite.

The current version of the tool can be downloaded from the web[2].

[1] Wikipedia lists 23 JavaScript libraries.

[2] `http://proglang.informatik.uni-freiburg.de/jscontest/`

```
1  /** int → int */
2  function f(x) { return 2 * x; };
3
4  /** (int,int) → bool */
5  function p(x,y) {
6    if (x != y) {
7      if (f(x) == x + 10) return "true"; // error
8    };
9    return false;
10 };
```

**Fig. 1.** JavaScript code with contracts.

## 2 A Tour of JSConTest

Suppose a programmer has written a function `p` in JavaScript to serve as a predicate on two integer values. Hence, the type signature of the function is (int,int) → bool. JSConTest allows the programmer to specify a signature for a function by writing it in a special comment above the function as shown in Fig. 1. Unfortunately, the function may violate its contract because it contains a **return** statement that returns a string instead of a boolean value. A static type system would reject the definition of `p` with this contract, but JSConTest needs to produce a concrete counterexample, first.

To do so, JSConTest generates a test suite from the program that contains code for the contracts of functions `f` and `p`. The test suite contains a registry that keeps track of which contract belongs to which function. Running the test suite amounts to visiting each contract in the registry, generating and executing random test cases for it, and collecting test cases for which the contract failed.

For the particular example code, it is highly unlikely that random testing actually detects the problem with the code. Spotting this defect requires the test case generator to guess a value for `x` such that $x * 2 == x + 10$ holds, but a random integer solves this equations with probability $2^{-32}$. (The value of `y` is uncritical with very high probability.) For example, a failing test case would set `x` to 10 and `y` to 1208, so that both conditions in the code are true and `p` returns the string value "true". For this return value, the check against the contract bool fails and JSConTest reports a counterexample.

However, it is to be expected that a large number of random tests can be applied to `p` without finding a counterexample.[3] So, the question is: How can we increase the probability that JSConTest finds a counterexample that exposes the defect in the code?

One possibility is to write down an additional, more specific contract for the function as in /** (int,int) → bool | (10,int) → bool */. This contract is an example for a combined contract, where the function has to obey each of the contracts

[3] To execute these tests in a browser near you use `http://proglang.informatik.uni-freiburg.de/jscontest/ex1.html`. To view the static results of such a test run see: `http://proglang.informatik.uni-freiburg.de/jscontest/s_ex1.html`.

separated by |. The second contract uses the singleton contract 10 to force the examination of x=10. With this contract, JSConTest finds a counterexample almost certainly (with probability $1-2^{-32}$), but proceeding in this way leads to unnatural and unreadable contracts, which get close to manually specified test cases. In fact, using combinations of singleton contracts exclusively corresponds to specifying single test cases. For instance, the test case contract /** (10,1208) → true */ also spots the problem.

### 2.1 Guided Random Testing

To address the problem realistically, JSConTest can perform *guided* random tests that rely on the results of program analysis. Annotations on the contracts tell the system which analysis to perform and how to use the result of the analysis. For example, applying the annotation @numbers to the integer contract instructs the system to collect all constant numbers from the code and to build its random arguments by evaluating expressions constructed from these numbers and the primitive arithmetic operations.

In the example, the contract (int@numbers,int) → bool would be suitable for p. Thus instructed, the analysis collects the list of constants [0,1,10] and passes it to the generator of the contract int together with the four default operations [+,−,∗,/]. Instead of generating a completely random integer value, the generator of the contract generates a random expression tree, where the leaves are values from the list of constants (with probability 0.5) or random integers (with probability 0.5), and uses the value of that expression tree as the input value. The nodes of the tree are picked from the operation set to yield trees like $(0 - 10 * 1 + 1289)$. The probabilities for creating a leaf or an internal node are chosen such that the probability of generating small trees is high. Thus, the probability to generate a tree with value 10 for x is high[4].

The webpage `http://proglang.informatik.uni-freiburg.de/jscontest/ex1a.html` illustrates that usually less that ten test cases need to be generated to find a counterexample for the contract.

Another example (see Fig. 2) demonstrates that the success of the expression tree approach is not accidental, even though finding a counterexample amounts to finding a solution to a Diophantine equation. The code in Fig. 2 has a similar defect as p but hides it under more complicated conditions. JSConTest usually finds the counterexample x=18, y=11, z=4 in a few seconds.

The example code in Fig. 3 demonstrates another use of collected information. The function h is a predicate for an object and should return only boolean values. But the return statement in line 4 returns the string value "true". To find this defect, JSConTest has to randomly generate an object with at least two properties, p and quest, the values of which must convert to true. As in the previous examples, the probability of randomly generating an object with these two labels is virtually zero, but again there is an analysis @labels that collects

[4] The probability is greater than $12^{-1}$, but we skip the details as they are not important.

```
1 /** (int@numbers,int@numbers,int@numbers) → bool */
2 function fut_1(x,y,z) {
3   if ((x*3+5 == y*5+4) && (x*2−1 == z*9 − 1))
4     return "true";
5   return false;
6 };
```

**Fig. 2.** Complicated conditions.

```
1 /** (object) → bool */
2 function h(x) {
3   if ((x) && (x.p) && (x.quest))
4     return "true";
5   return false;
6 };
```

**Fig. 3.** Object access.

the labels used inside the function. Changing the object contract to object@labels leads to the quick discovery of a counterexample (usually in five tests or less).

### 2.2 Monitoring

Another aspect of JSConTest is run-time monitoring of contracts. To this end, JSConTest inserts assertions into the bodies of functions that have a contract (as specified by Findler and Felleisen [15]). Every time a function with a combination of contracts is called, the test library checks which contracts have argument parts that accept the passed parameters. If no such contract exists, then the arguments are not valid and the function call is rejected (because the caller does not respect the contract). This rejection is signaled as an event, which can be picked up by a registered logger. A logger may ignore the event, print a warning, stop program execution, or it may invoke a debugger at that point. If the arguments are valid, then the result value of the function is checked against the result parts of the contracts that accepted the arguments. Failure of any of these checks is again signaled as an event.

As an example, consider the function g in Fig. 4, which calls function f from Fig. 1. The code contains an error in a literal: The programmer mixed up zero and the letter O, writing "3O" instead of "30". This error causes NaN to be passed to f, where it is caught by the contract (int) → int. Such an error would be very hard to spot without contracts and monitoring because of JavaScript's liberal type conversions.

```
12 /** (int,int) → bool */
13 function g(x,y) {
14   return (f(x * "3O") == 60); // error
15 }
```

**Fig. 4.** Catching an error in a string literal.

$i \in \texttt{int}, f \in \texttt{float}, s \in \texttt{string}, b \in \texttt{bool}$

Primitive contracts

| | | |
|---|---|---|
| $p ::=$ | $\texttt{undf} \mid \texttt{void} \mid \top$ | undefined, any value |
| $\mid$ | $\texttt{bool} \mid b$ | boolean values |
| $\mid$ | $\texttt{string} \mid s$ | string values |
| $\mid$ | $\texttt{int} \mid i \mid \texttt{[}i;i\texttt{]}$ | integers, integer intervals |
| $\mid$ | $\texttt{float} \mid f \mid \texttt{[}f;f\texttt{]}$ | floats, float intervals |
| $\mid$ | $\texttt{object}$ | object |
| $\mid$ | $\texttt{js:ident}$ | defined in JavaScript |

Composite contracts

| | | |
|---|---|---|
| $c ::=$ | $p$ | |
| $\mid$ | $c\texttt{@numbers} \mid c\texttt{@strings} \mid c\texttt{@labels}$ | analysis information |
| $\mid$ | $\texttt{(}d,\ldots,d\texttt{) -> }d$ | functions |
| $\mid$ | $\{p_1 : c_1, p_2 : c_2, \ldots, p_n : c_n(,\ldots)^?\}$ | objects |
| $\mid$ | $\texttt{[}c\texttt{]}$ | arrays |

Annotations

$an ::= \texttt{~noAsserts} \mid \texttt{~noTests} \mid \texttt{\#Tests:}i$

Dependent contracts

$d ::= c \mid c\texttt{(\$}i,\ldots,\texttt{\$}i\texttt{)}$

Top-level contracts (embedded in JavaScript comments)

$t ::= \text{'/**'}\ c\ an^*\ (\ \text{'|'}\ c\ an^*)^*\ \text{'*/'}$

**Fig. 5.** Syntax of contracts.

## 3 Contracts

Fig. 5 summarizes the syntax of JSConTest's contract language. Its main design goal is to let programmers specify interfaces easily. Compared to the design by contract methodology, these contracts play dual role as checkable assertions that can be attached to JavaScript values (contract monitoring) and as test case generators for such values and functions.

The primitive contracts are analogous to types used in languages with a static type system (e.g., Java, C, C++). They correspond one to one to the types available in JavaScript. Their interpretation is strict, that is, it does *not* include the application of JavaScript's type conversions. For example, the contract `bool` accepts only the values `true` and `false` and the contract `int` accepts signed 32-bit integer values. For the primitive datatypes there are also singleton contracts, which only accept a single value. Singleton contracts for floats, integers, strings, and booleans are specified by simply writing the corresponding literal value.

```
1 /** (int,int) → bool */
2 function comp(x,y) {
3   if ((x == y) && (x < 10) && (x > 1)) return "true";
4   return false;
5 }
```

**Fig. 6.** A comparison operator.

There is also support for intervals of integers and floats with the syntax [i;i] or [f;f]. For instance, [0;1.0] is the float interval from zero to one.

The primitive contract form js:ident allows programmers to roll their own contracts by writing an appropriate definition directly in JavaScript. This contract form enables having concise specifications of function interfaces which encapsulate complex test code. JSConTest supports such handwritten contracts by providing various utility functions.

Composite contracts are built from primitive contracts by combining them in different ways. An enriched contract adds specific guidance (in the form of a static analysis) to the test case generation for a contract, it does not change its interpretation as an assertion. Our current implementation supports three analyses. The analysis @numbers collects all numeric constants from the function body, the analysis @strings collects string constants, and the @labels analysis static property names. A function contract specifies contracts for the arguments (and their number) and the result. An object contract specifies contracts for the listed properties of an object. An acceptable object may have further properties, which are not checked. The existence of ... at the end of the object contract forces the generator to randomly add more properties to the object. Otherwise only the properties specified for the object are generated. Arrays are considered homogeneous, so an array contract specifies a single contract for all elements of the array.

Function contracts may contain various kinds of dependencies. For example, the parameters of a function may depend on each other as illustrated by the example code in Fig. 6. The contract /** (int@numbers,int@numbers) → bool */ can find a counterexample, but with probability on the order of $10^{-3}$ although @numbers = [0,1,10] has just three elements. The problem is that a counterexample needs to set x and y to the same value between one and ten.

Adding a second contract (int@numbers,id($1)) → bool to comp significantly raises the probability to find a counterexample. This contract expresses that passing a copy of the first argument as the second argument is a special case, which should get special treatment. If p is supposed to be a comparison, then the additional contract (top,id($1)) → true expresses this kind of information.

The dependencies must by acyclic. The compiler detects and rejects any cyclic definitions of dependent contracts.

As JavaScript is a higher-order language, JSConTest must be able to deal with function-typed arguments and results to functions (recursively). Hence, there is a need no only for checkers at function type, but also for generators

at function type. At this point, the dual role of our contracts actually pays off because both are needed, in general:

$$\boxed{\begin{array}{ll}(A \to B).\text{check} & \text{requires } A.\text{generate and } B.\text{check} \\ (A \to B).\text{generate} & \text{requires } A.\text{check and } B.\text{generate}\end{array}}$$

In particular, the generator for $(A \to B)$ produces a function that first invokes $A$.check on its argument $x$ and signals failure if the check fails. Then it uses $B$.generate (potentially exploiting dependencies as already explained) to create an output value $y$ for the function. Depending on the user's choice between pure or impure functions, this value can be memoized.

At the top-level, each function can have a list of contracts attached to it. A function must fulfill all of them, thus JSConTest generates tests as well as assertions for all contracts. A number of annotations can be attached to each contract to modify the behavior of the contract compiler. If `~noTests` is present in the annotations, the contract is not added to the test suite. The annotation `~noAsserts` avoids the generation of assertions for the contract. The annotation `#Tests:`$i$ changes the number of tests that the generated test suite executes for the contract. These annotations are only needed in special situations. Our case study in Sec. 5 runs into some of them and provides a detailed explanation.

## 4 Implementation

The contract compiler is implemented in roughly 6000 lines of OCaml (including about 2000 lines devoted to parsing JavaScript). It parses the JavaScript file, creates an abstract syntax tree of the source code, and parses the contracts from the comments of the JavaScript code. Next, the compiler analyzes the dependencies between the contracts to ensure that they are acyclic. Finally, it generates code for the contracts as well as code to connect them to the test framework. Depending on the chosen options and annotations, it additionally transforms the functions under contract by adding monitoring assertions.

The test library consists of about 1900 lines of browser-independent JavaScript. It comprises three parts. The first part manages test cases and assertions. The second part provides utility function for user-written contracts. The third part deals with loggers, which provide a user-configurable way of reacting on contract violations and assertions.

### 4.1 Test Cases and Assertions

Besides using the syntax introduced in Fig. 5, the library may also be used to construct contracts manually. For that reason (and also to simplify the compilation), the library contains many predefined contracts, for example Top, Null, Undefined, Boolean, True, False, String, Number, Function, Object, PObject, AInteger. Most of them are self explanatory, but PObject and AInteger deserve further explanation. PObject is a contract that takes an array of property names as argument. It implements object contracts that rely on information generated by the label

```
1  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
2          "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
3  <html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
4    <head>
5      <title>Test Framework</title>
6      <meta http-equiv="Content-Script-Type" content="text/javascript"/>
7      <link href="style.css" rel="stylesheet" type="text/css" />
8      <script type="text/javascript" src="testsrandom.js"></script>
9      <script type="text/javascript" src="ex1.test.js"></script>
10     <script type="text/javascript">
11       var dl = TESTS.createEnumLogger('logger',"ul");
12       TESTS.registerLogger(dl);
13     </script>
14   </head>
15   <body onload="TESTS.run()">
16     <h1>Test Framework</h1>
17     <div id="logger"></div>
18   </body>
19 </html>
```

**Fig. 7.** HTML page generated to drive a test suite.

analysis of the contract compiler. The behavior of AInteger is similar. It takes two parameters, the list of constants found in the function body and a list of binary functions for combining two integer values. By default, the second parameter is initialized to the basic arithmetic operations.

As a concession to the dual role of contracts, any object implementing a primitive contract provides two methods, one to check if a value adheres to the contract and another to randomly generate a value that is guaranteed to adhere to the contract.

Fig. 7 shows the HTML page generated as a driver for the test suite for the program `ex1.test.js`. It loads the library (line 8) and the compiled contracts (line 9). The script from line 10 to line 13 registers a logger. In line 15 the run method is registered as the onload handler to execute the tests after the page has been loaded. The content of the page is empty except for the title and a div element, in which the logger inserts the results of the random tests.

### 4.2 Custom Contracts

The library contains a number of contract combinators, for example, a union and an intersection operation. There are also a number of contracts for objects, which allow the specification of a set of properties, each with its own contract, or that allow property names composed of only characters and digits, but no special characters, etc. It is also possible to specify recursive contracts for objects by using names as follows:

```
1 Let ("o", EObject[{name: "m",
2                    contract: Function ([Name ("o")], Boolean)}])
```

```
1 function check(x,y) {
2   return x === y;
3 };
4 function generate(v) {
5   return v;
6 };
7 Id = new newSContract(check,generate,"Id");
```

**Fig. 8.** Implementation of the dependent contract Id.

Let binds a contract to a name and this contract can be retrieved with the Name function. EObject constructs an object contract from an association list of property names to contracts. The resulting contract describes an object with an m method the argument of which must be an object of the same kind.

Currently, this technique is not reflected in the surface contract language, because the main purpose of the contract language is to easily specify the interface of a function and we have not found an example where it was needed.

The library contains further functions that aid in writing contracts from scratch. For example, the constructor newSContract: (check, generate, cdes) → Contract creates a fully general contract. The function check: Top → bool is called by the test library to check if a value fulfills a contract. The procedure generate: void → value is supposed to create a valid value for the contract, viz., every value returned by the generate function must pass the check function. The third parameter is a string that identifies the contract if a logger asks for a string representation of it. To support the implementation of custom contracts a set of check functions, viz. isNull, isTrue, isInt, isIInt, isArray, isObject, and a set of generators are available, viz. genNull, genBoolean, genInt, genIInt, genNumber, genLength, genString, genObject. Some of the generators take parameters, which you can use to modify their default behavior, but they all work without parameters, too. In such a case everything is chosen randomly.

### 4.3 Dependent Contracts

For dependent contracts, the methods check and generate have additional arguments. For example, consider the dependent function contract (bool,int) → Id($2). In this case, the check method of the Id contract is eventually invoked with two parameters (see 8 for its definition). The first is the value that is to be checked and the second is the value that is referred to by the $2 part of the contract, viz. the integer value of the second argument of the checked function.

The test for such a contract first generates argument values according to bool and int. Then it computes the result of the function call on these arguments and invokes the check method for Id with the result and the generated int value. In this particular case, Id.check tests its arguments for equality so that the contract describes a function that ignores its first argument (which must be boolean) and returns its second, integer argument.

In general, the check method for a dependent contract has the signature check: (Top, ...) → bool, where Top stands for the value against which the contract is to be checked and the (arbitrary many) remaining arguments are the ($i,. . . ,$j) additional parameters passed to the contract.

In principle, there could even be a contract with a variable number of arguments such as: (int,Dep($1),Dep($1,$2)) → bool. A test for this contract calls the generate methods for int, then for Dep with one extra parameter, and finally for Dep with two extra parameters. The order of these calls is determined by the preceding dependency analysis, that is, it would be the same for the contract (Dep($3),Dep($3,$1),int) → bool.

Analogous to the case of the check method, the signature of the generate method changes for a dependent contract. The new signature is generate: (. . . ) → value, which means that the generate method can use the information from the values on which the contract depends to generate its result.

### 4.4 Logging

The library communicates its results through a logging framework. Generated tests and assertions always invoke the logging framework to publish their results as events.

Using the function registerLogger, any number of logger objects can be registered with the library as shown for the createEnumLogger in Fig. 7. Thus, a logger can be provided to create a nice browser output, another one to create a log file, publish the results to a web service (createWebLogger) and so on.

A logger object can also provide an interface to a debugger. Such a logger may stop execution and invoke the debugger at each failing assertion. Unfortunately, the JavaScript specification does not prescribe a standard interface for such tools, so the library cannot supply a generic solution for this purpose.

### 4.5 Transformation

We illustrate the transformation of contracts into JavaScript code with an example. Fig. 9 shows the result of compiling the contract for function f from Fig. 1. The compiled code consists of three parts. The first part is the function enriched with assertions (line 2-6). The second part overrides the toString method of the function object (line 7-13). The override ensures that the toString method returns the original source code rather than the mangled compiled code. The goal is to facilitate interaction with a debugger and also to have the test framework produce readable output. The last part is code that creates the contracts and adds test cases to the test framework (line 16-21).

The code makes frequent use of the JavaScript idiom (**function**() {...}) () to create a new nested scope in which to execute the body .... Nested scopes are used to avoid the pollution of the (single) JavaScript namespace with auxiliary definitions.

Similarly, the compiled code employs a single, configurable global object (default: TESTS) to interact with the framework. No further global variables are

```
1  var f =
2    (function () {
3      function f_own (x) {
4        var p_1 = TESTS.assertParams(["p_0"], arguments, f_own);
5        return p_1.assertReturn((2. * x));
6      };
7      (function () {
8        function f (x) {
9          return (2. * x);
10       };
11       f_own.toString = function() {return f.toString();};
12     })();
13     return f_own;
14   })();
15 /* compiled code for p and g is omitted */
16 (function () {
17   var c_0 = TESTS.Function([TESTS.Integer], TESTS.Integer);
18   TESTS.add("f", f, c_0, 1000. );
19   TESTS.setVar("p_0", c_0);
20  })();
21 /* test cases for p and g are omitted */
```

**Fig. 9.** Compiled code for function f.

used. This strategy is required because JavaScript has no namespace management and also because static scoping is not guaranteed[5]. To avoid global variables, the library provides an attribution mechanism to read and write values (getVar, setVar). The real implementation chooses names that are unlikely to clash with user defined ones.

## 5 Case Study: Huffman Decoding

In this case study we implement a Huffman decoder in JavaScript and specify its interface with the contract system. Writing this decoder is not difficult, but nevertheless JSConTest uncovered errors during its development. Usually, the test framework found counterexamples after just one or two test runs. In each case, the counterexample simplified the debugging of the code significantly.

After some iterations of fixing small errors in the functions under test, we found an error in the specification. The contract js:ht is a custom contract for Huffman trees. A Huffman tree is either a node or a leaf. A node then contains two Huffman trees as children. Leaves and nodes both contain additional information, which we ignore for now. The problem is that a valid Huffman tree has to have a depth greater than 0. If the tree consists of only one leaf, no bits are

[5] For example, source code may be loaded dynamically by an HTTP-Request and then executed by an eval expression, which may place bindings in an arbitrary scope.

```
1  function generate() {
2    function genRandomLeaf() {
3      return makeHuffmanLeaf(TESTS.genStringL(1), TESTS.genNInt(0,1));
4    };
5    function genRandomNode(l,r) {
6      return makeHuffmanNode([],TESTS.genNInt(0,1),l,r);
7    };
8    function cdes() { return "makeHuffmanNode"; };
9    var gN = { getcdes: cdes, arity: 2, f: genRandomNode};
10   return TESTS.genTree(isHuffmanTree,[genRandomLeaf],[gN],0.5,true);
11 };
12 var gen = TESTS.restrictTo(isHuffmanNode,generate);
13 var ht = new TESTS.newSContract(isHuffmanTree,gen,"HuffmanTree");
14
15 /** Top → bool */
16 function isHuffmanLeaf(v) { /* function body here */ };
17
18 /** Top → bool */
19 function isHuffmanNode(v) { /* function body here */ };
20
21 /** Top → bool | js:ht → true ~noAsserts */
22 function isHuffmanTree(v) { /* function body here */ };
```

**Fig. 10.** Custom contract ht — check for a Huffman tree.

consumed by the decoder to generate an output string and the decoder enters an infinite loop. We discovered this bug when our framework generated a Huffman tree of the form HuffmanLeaf {s: '', w: ...}. This input results in a stack overflow which is reported by our framework as a failing test.

The fix for this problem was to adjust the generator for Huffman trees. We use the function restrictTo from our library to filter the result of the generator, such that trees containing only one leaf are rejected.

The contract constructor newSContract suffices to create ht. The check function is presented in Fig. 10. The generate function makes use of the library function genTree for generating random trees (also presented in Fig. 10). One contract of genTree is: (Top → bool, [void → value], [op], [0;1.0], true) → value. The first parameter is a predicate, the second one a list of functions that generate the leaves of the tree. The second array [op] contains a list of objects that contain functions that generate a new value from a list of values. The arity specifies how many subtrees the individual function takes. The fourth parameter is a probability to choose an operation during the generation. The last parameter specifies if the first array is a list of functions that generate values. If this parameter is set to false, then the method assumes that the array itself contains values. As the functions isHuffmanLeaf, isHuffmanNode, and isHuffmannTree already exist in the original code, the custom contract ht requires just 13 lines of code and tree lines of contract specification (line 15,18,21).

Using the custom contract ht, it is easy to write the contracts for the other operations on Huffman trees. Function huffmandecode has contract (js:ht, [[0;1]]) → string, a function returning a string, where the first parameter is a Huffman tree and the second one an array of bits, which are integers from the interval [0;1].

For this contract, the dual role of isHuffmannTree as the type test to be used by the programmer as well as for defining the ht contract leads to a circularity. Its contract specifies that isHuffmannTree is applicable to any value and returns a boolean. The second half says that if the argument is a Huffman tree, then it should return true. The circularity arises when doing contract monitoring for this function. In that case, an invocation of isHuffmannTree asserts the contract ht on its argument, which in turn uses isHuffmannTree for checking it, which gives rise to an infinite loop. Hence, the `~noAsserts` annotation is needed to avoid generating assertions for this contract.

To obtain some intuition of the quality of guided random testing, we applied mutation testing [2,1] to the Huffman decoder. The mutator swaps makeHuffmanLeaf and makeHuffmanNode, [left,right], [true,false], [0,1], and [===,==,!=,!==], [-+]. The mutator randomly determines the number of modifications to apply and repeatedly applies a modification at a random place in the program.

Our test run generated 716 mutations of the original huffman.js, which were all submitted to the contract compiler and then the resulting test runs were analyzed. There were 88 files (about 12% of all files) for which all contracts passed successfully. All other files were rejected outright by our test suite. The average run time of each file in the browser is 5 seconds.

It is not surprising that there are modified versions for which all tests pass, because some of the randomly chosen modifications (e.g., swapping left and right) leads to files, in which nothing is wrong from a type perspective. For these 88 files, we verified manually that the modifications of these file are not observable on the type level, e.g. if a contract states Top → bool and inside the function body **return** true; is changed to **return** false;, the contract is fulfilled by the original and the modified version. Finding such bugs is either the job of a hand-written test suite or of a more detailed specification. There were also modifications that affected the contracts themselves. For example, a modification that swaps 0 and 1 increased the number of tests from 50 to 51 because it took place in a ... #Tests:50 annotation. Such a modification does not affect the semantics of the program.

As the generated tests rejected 88% of the mutations, we argue that our testing framework detects a type error with very high probability.

## 6 Related Work

Purely manual testing per se does not guarantee any kind of coverage criterion and its effectiveness depends highly on the experience of the tester and on the system of the chosen approach to testing. Hence, manual testing should be backed up by further kinds of testing. Random testing [3,20] is one promising candidate, which is surprisingly effective [18], but which does not give guarantees with

respect to coverage [27,10]. However, there are a number of approaches and tools that support random testing and that employ various means for improving coverage.

JCrasher [9] is a black-box random testing tool for the Java programming language. It analyzes a set of classes with the goal to find a crashing program fragment involving methods of these classes. It constructs fragments by applying methods with random parameters to randomly constructed objects and then using these objects as a basis for randomly generating further method calls. There is no further specification of contract needed for JCrasher as the failure criterion is a program crash.

In contrast, JSConTest can test against user-specified contracts and can also do run-time monitoring. Moreover, JSConTest improves coverage by performing a limited amount of glass-box testing by collecting constants from the code to performed guided testing.

The QuickCheck library [8] for Haskell, a purely functional programming language, enables the statement of properties of program constructs, which are then automatically tested. Test cases are randomly generated from the types of the variables in properties. Additionally, programmers can specify their own generators. In contrast, JSConTest derives its test cases from contracts, which can be more expressive than types, and it is only geared to test contracts (although it could be extended to test properties as well). JSConTest handles test case generation for imperative JavaScript objects, which go beyond functions and primitive data. Another difference is that QuickCheck performs pure black-box testing whereas JSConTest's inclusion of program analysis information places it on the brink to glass-box testing.

DoubleCheck [11] is an adaptation of QuickCheck to the ACL2 language implemented in the PLT programming environment[16][6]. It is used as a verification aid to generate counterexamples for properties of programs that ACL2 cannot prove right away. The idea is to restate these properties guided by the counterexamples. PLT-redex also comes with a random testing facility that has detected errors in semantics specifications [23].

RUTE-J [2] is a framework that enables writing unit tests for Java that make use of some portion of randomness. It can randomize a list of method calls as well as input data and it performs minimization of failing test cases.

Randoop [28] is a tool for directed random testing of Java classes. It generates test cases in a similar way as JCrasher, but additionally uses the test outcomes as feedback to avoid creating useless or outright erroneous tests.

Similarly, the ARTOO system [7] performs adaptive random testing for Eiffel. It adapts previous ideas from the ART approach [6] to an object-oriented setting. Its underlying idea is that tests are more effective if they evenly cover the parameter space of the method under test. Its execution requires a distance metric on the input values.

[6] The DrScheme teaching languages also provide QuickCheck-style testing of contracts.

A highly effective approach to randomized testing is the DART system [19]. It performs what has been coined concolic testing: it combines running concrete test cases with symbolic execution of the underlying code. Guided by the outcome of concrete test cases it generates symbolic predicates for the branches taken in the computation. It employs theorem proving to systematically falsify these predicates and thus attempts to cover all branch alternatives, which is often successful. JSConTest is inspired by this system, but relies on a much more lightweight approach (collecting constants), which requires a larger number of test cases, for increasing the coverage.

A different approach to generating test cases is bounded exhaustive testing, which systematically enumerates all inputs below a certain size threshold. This approach is implemented, for example, in the Smallcheck system for Haskell [29] and also in the Korat system for Java [5]. The idea here is that counterexamples are usually small and that the exhaustive tests give some guarantees, at least for finite structures like functions over finite domains. This approach is complementary to the random testing approach presently chosen by JSConTest.

There are also JavaScript testing frameworks, for example, JSUnit[7], JsTester[8], FireUnit[9], JSCoverage[10], JSMock[11], and rhinounit[12]. However, these frameworks are in the tradition of unit testing frameworks like JUnit[13]. Their focus is on automating the execution of unit tests, but not on the creation of these tests, which is left to the human developer. In contrast, JSConTest only requires the manual construction of interface specifications, which is a good idea anyway. Also, JSConTest is currently restricted to functional testing of the JavaScript code, it does not test the interactive behavior (GUI testing), nor the interface to web services via XmlHTTPRequest. These extensions are left to future work.

JSConTest is inspired by, but complementary to work on type analysis for JavaScript [22,21]. The focus of these works is to determine the type safety of JavaScript programs by static analysis (abstract interpretation and constraint-based analysis, respectively). Neither work supports type specifications, nor test case generation.

## 7 Conclusion

JSConTest is the first testing tool for JavaScript that supports the specification of type contracts. These type contracts are validated in two ways, by contract monitoring and by guided random testing. The latter is a new approach to improve the coverage of random testing, which is very effective and easy to imple-

[7] `http://www.jsunit.net/`
[8] `http://jstester.sourceforge.net/`
[9] `http://fireunit.org/`
[10] `http://siliconforks.com/jscoverage/`
[11] `http://jsmock.sourceforge.net/`
[12] `http://code.google.com/p/rhinounit/`
[13] `http://junit.org/`

ment. Contract violations are usually discovered within a few test runs. Mutation testing shows that JSConTest verifies type contracts with a high probability.

In future work, we plan to extend the generators in the style of JCrasher and Randoop. We also plan to incorporate specifications for side effects. As the current generation procedure for objects constructs only trees, we plan to lift this restriction by incorporating a generation algorithm as in Shekoosh [13].

## References


1. J. H. Andrews, L. C. Briand, and Y. Labiche. Is mutation an appropriate tool for testing experiments? In *ICSE '05: Proc. 27th International Conference on Software Engineering*, pages 402–411, New York, NY, USA, 2005. ACM.
2. J. H. Andrews, S. Haldar, Y. Lei, and F. C. H. Li. Tool support for randomized unit testing. In *RT '06: Proceedings of the 1st International Workshop on Random Testing*, pages 36–45, Portland, Maine, 2006. ACM.
3. D. L. Bird and C. U. Munoz. Automatic generation of random self-checking test cases. *IBM Syst. J.*, 22(3):229–245, 1983.
4. B. Bloom, J. Field, N. Nystrom, J. Östlund, G. Richards, R. Strniša, J. Vitek, and T. Wrigstad. Thorn: Robust, concurrent, extensible scripting on the JVM. In S. Arora and G. T. Leavens, editors, *Proc. 24th ACM Conf. OOPSLA*, pages 117–136, Orlando, Florida, USA, 2009. ACM.
5. C. Boyapati, S. Khurshid, and D. Marinov. Korat: Automated testing based on Java predicates. In *ISSTA '02: Proc. 2002 ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 123–133, Roma, Italy, 2002. ACM.
6. T. Y. Chen, F.-C. Kuo, R. G. Merkel, and T. H. Tse. Adaptive random testing: The ART of test case diversity. *J. Systems and Software*, 83(1):60–66, 2010.
7. I. Ciupa, A. Leitner, M. Oriol, and B. Meyer. ARTOO: Adaptive random testing for object-oriented software. In W. Schäfer, M. B. Dwyer, and V. Gruhn, editors, *ICSE 2008*, pages 71–80. ACM, 2008.
8. K. Claessen and J. Hughes. QuickCheck: A lightweight tool for random testing of Haskell programs. In P. Wadler, editor, *Proc. ICFP 2000*, pages 268–279, Montreal, Canada, Sept. 2000. ACM Press, New York.
9. C. Csallner and Y. Smaragdakis. JCrasher: An automatic robustness tester for Java. *Software—Practice & Experience*, 34(11):1025–1050, Sept. 2004.
10. J. Duran and S. Ntafos. An evaluation of random testing. *Transactions on Software Engineering*, 10(4):438–444, 1984.
11. C. Eastlund. DoubleCheck your theorems. In *ACL2 2009*, Boston, MA, 2009.
12. ECMAScript Language Specification. `http://www.ecma-international.org/publications/files/ECMA-ST/Ecma-262.pdf`, Dec. 2009. ECMA International, ECMA-262, 5th edition.
13. B. Elkarablieh, Y. Zayour, and S. Khurshid. Efficiently generating structurally complex inputs with thousands of objects. In Ernst [14], pages 248–272.
14. E. Ernst, editor. *21st ECOOP*, volume 4609 of *LNCS*, Berlin, Germany, July 2007. Springer.
15. R. B. Findler and M. Felleisen. Contracts for higher-order functions. In S. Peyton-Jones, editor, *Proc. ICFP 2002*, pages 48–59, Pittsburgh, PA, USA, Oct. 2002. ACM Press, New York.

16. R. B. Findler, C. Flanagan, M. Flatt, S. Krishnamurthi, and M. Felleisen. DrScheme: A pedagogic programming environment for Scheme. In H. Glaser and H. Kuchen, editors, *Intl. Symp. Programming Languages, Implementations, Logics and Programs (PLILP '97)*, volume 1292 of *LNCS*, Southampton, England, Sept. 1997. Springer.
17. R. B. Findler, M. Flatt, and M. Felleisen. Semantic casts: Contracts and structural subtyping in a nominal world. In M. Odersky, editor, *18th ECOOP*, volume 3086 of *LNCS*, pages 364–388, Oslo, Norway, June 2004. Springer.
18. J. E. Forrester and B. P. Miller. An empirical study of the robustness of Windows NT applications using random testing. In *4th USENIX Windows System Symposium*, Seattle, Aug. 2000.
19. P. Godefroid, N. Klarlund, and K. Sen. Dart: Directed automated random testing. In *Proc. 2005 ACM Conf. PLDI*, pages 213–223, Chicago, IL, USA, June 2005. ACM Press.
20. R. G. Hamlet. Random testing. In J. Marciniak, editor, *Encyclopedia of Software Engineering*, pages 970–978. Wiley, 1994.
21. P. Heidegger and P. Thiemann. Recency types for dynamically-typed object-based languages. In *2009 International Workshop on Foundations of Object-Oriented Languages (FOOL'09)*, Savannah, Georgia, USA, Jan. 2009. `http://www.cs.cmu.edu/~aldrich/FOOL09/heidegger.pdf`.
22. S. H. Jensen, A. Møller, and P. Thiemann. Type analysis for JavaScript. In *Proc. 16th International Static Analysis Symposium, SAS '09*, volume 5673 of *LNCS*. Springer-Verlag, Aug. 2009.
23. C. Klein and R. B. Findler. Randomized testing in PLT Redex. In *Workshop on Scheme and Functional Programming 2009*, Boston, MA, USA, 2009.
24. J. Matthews and R. B. Findler. Operational semantics for multi-language programs. *ACM TOPLAS*, 31:12:1–12:44, Apr. 2009.
25. B. Meyer. *Object-Oriented Software Construction*. Prentice-Hall, 2nd edition, 1997.
26. G. J. Myers and C. Sandler. *The Art of Software Testing*. John Wiley & Sons, 2004.
27. A. J. Offutt and J. H. Hayes. A semantic model of program faults. In *1996 ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 195–200, San Diego, CA, USA, Jan. 1996.
28. C. Pacheco. *Directed Random Testing*. Ph.D., MIT Department of Electrical Engineering and Computer Science, Cambridge, MA, USA, June 2009.
29. C. Runciman, M. Naylor, and F. Lindblad. Smallcheck and Lazy Smallcheck: Automatic exhaustive testing for small values. In *Haskell '08: Proc. of the first ACM SIGPLAN Symposium on Haskell*, pages 37–48, Victoria, BC, Canada, 2008. ACM.
30. J. Siek and W. Taha. Gradual typing for objects. In Ernst [14], pages 2–27.
31. S. Tobin-Hochstadt and M. Felleisen. The design and implementation of typed scheme. In P. Wadler, editor, *Proc. 35th ACM Symp. POPL*, pages 395–406, San Francisco, CA, USA, Jan. 2008. ACM Press.
32. P. Wadler and R. B. Findler. Well-typed programs can't be blamed. In *ESOP '09: Proceedings of the 18th European Symposium on Programming Languages and Systems*, pages 1–16, York, UK, 2009. Springer-Verlag.
33. A. Zeller and R. Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE Trans. Softw. Eng.*, 28(2):183–200, 2002.