1. INTRODUCTION

This work aims to help developers writing unit tests for Node.js via automatic test generation. Our idea is shown in figure 1.

- Instrument functions of the SUT to spy the values of the external inputs and the expected outputs.
- Users interact with the instrumented source code using different test scenarios.
- Synthesize the unit tests based on the collected input values and output values.

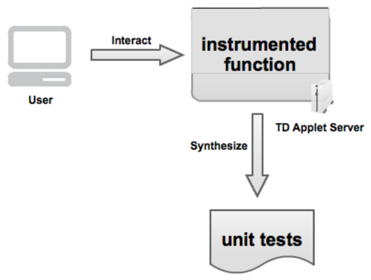

Figure 1: Unit test generation process

To prove our concept, we are implementing a prototype. In our implementation, we leverage existing libraries, including Esprima[1], Sinon.js[2], and Dust.js[3]. The libraries reduce our implementation effort while posing some restrictions. The main restriction is that we only guarantee correct tests for functions of referential transparency and side-effect free. We would discuss further in section 2.

The rest of the paper is organized as follows. Section 2 presents the implementation of our prototype. Section **??** presents preliminary result of this work. Finally, we summarize this work in section **??**.

2. IMPLEMENTATION

There are mainly three parts in our prototype including instrumentation, runtime library, and test synthesis. In figure 2, we show the flow of our prototype. First, the source is instrumented with runtime library. Next, the instrumented code is executed to collect test data. Finally, the test data is fed into template to synthesize unit tests. In the following, we present these parts with more details.

2.1 Instrumentation

We use Esprima to parsing source code of Node.js modules. To simplify processing afterward, we made the following assumptions on source code.

- Curly braces are added for blocks.
- Expressions are simplified so we can insert code and preserve the correct behavior.

The source code is translated into the canonical form if it does not conform to the assumptions.

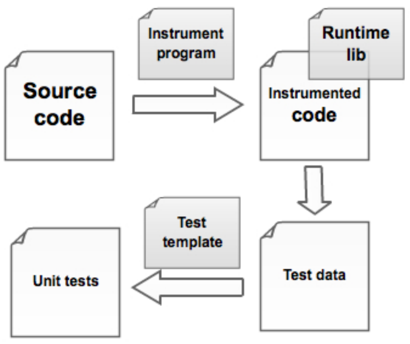

Figure 2: Flow

We target to generate unit tests for exported functions in Node.js modules. Instead of generating unit tests for all exported functions, we expect developers to specify functions under test by writing specific leading comments, /* tag */. In figure 3, the function `f()` is under test since there is the leading comments. Note that we expect exported function name is the same as the function under test for simplicity. For example, in figure 3, `module.exports.f` and `f()` are of the same name. In this case, we can simply call `module.exports.f()` in unit tests. This restriction could be lifted by alias analysis.

```
/* atg */
function f(p) {
  return fs.readFileSync(p);
};
var obj = {
    f: f;
};
module.exports = obj;
```

Figure 3: Source code

In figure 4, we shown the instrumented code for the source code in figure 3. In line 1, we use our runtime library to wrap the Sinon.js so that runtime library will be executed before and after a function invocation. In line 5 and 14, we create spies on the function under test `f()` and it's dependency `fs.readFileSync()`.

A function can return it's results by callbacks. A callback is an function argument which is then invoked inside the function. We spy the results passed into a callback so we can handle function with callbacks.

A function can depends on global variables or it can modify global variables. That is, it is not referential transparency or is not side-effect free. However, Sinon.js doesn't store the values of global variables so we only guarantee correct tests for functions of referential transparency and side-effect free. This restriction can be lifted by storing the values of global variables in runtime library. Moreover, a function can depends on closure variables. For this case, we can store the values of closure variables but there is not general rule to setup the value of closure variables for unit tests.

```
1
2  atg.wrap(sinon);
3  /* atg */
4  function _f(p) {
5    sinon.spy(fs, 'readFileSync',
6          {'filePath': FS_FILEPATH});
7
8    var res = fs.readFileSync(p);
9
10   f.dependencies = [fs.readFileSync];
11   fs.readFileSync.restore();
12   return res;
13 };
14 var f = sinon.spy(_f,
15         {'filePath': FILEPATH, 'sut': true});
16 var obj = {
17     f: f;
18 };
19 module.exports = obj;
```

Figure 4: Instrumented code

2.2 Runtime Library

We use runtime library to simplify the instrumentation, both program to instrumentation and instrumented code, Runtime library mainly provides functions to collect test data. We leverage Sinon.js in our runtime library.

`sinon.spy()` creates a proxy for each spied function. A proxy records arguments, invokes spied function, and finally records return value. We further creates a proxy for Sinon.js's proxy so that we can customize actions before and after invoking spied function. Before invoking spied function, we record `filePath` for requiring the function in unit tests, e.g., line 6 and line 15 of figure 4. After invoking spied function, we synthesize a unit test for that invocation. Test synthesis is presented in section 2.3.

As we mention, runtime library mainly provides functions to collect test data. We define an intermediate representation (IR) of test data. The IR abstracts detail of SUT execution while retains essential information for test synthesis. The IR is base on the Sinon.js spy API since we use Sinon.js to collect test data. We give the IR in figure 5.

```
{
    filePath : "",
    funName: "",
    thisValue: {}
    args: [],
    returnValue: {},
    dependencies: []
}
```

Figure 5: Intermediate representation

2.3 Test Synthesis

To synthesize a unit test with collected test data, we use Dust.js template engine. The benefit of using template engine is to separate test data from what shall be tested in a unit test. As a result, it is easy to customize unit tests and add new unit tests by reusing test templates. In appendix A, we give a template example, which generates the unit test in appendix B.

In the line 21, 26, 34 of the template, we use '`se`' filter to serialize values. We need to serialize the values of variables to write them down in unit tests. JSON.stringify() and JSON.parse() are the standard methods for serialization but they cannot be applied to functions. For function serialization, we use toString() to serialize them and use eval() to unserialize them back. Note that, native functions, e.g., process.abort, cannot be serialized with toString(). Since not every value is serializable, there will be information loss in serialization. When matching the actual result and expected result, we also apply serialization on actual result to take the information loss into account. Currently, our prototype handles primitive value, non-native functions, and basic objects. Further investigation on prototype property is needed.

3. REFERENCES

[1] Esprima. In *http://esprima.org/*.
[2] Sinon.js. In *http://sinonjs.org/*.
[3] Dust.js. In *http://akdubya.github.io/dustjs/*.

APPENDIX

A. TEST TEMPLATE

```
'use strict';
{#dependencies}
    var {.moduleName} = require('{.filePath}');
{/dependencies}
var sinon = require('sinon'),
    sut = require('{filePath}');

describe('{testFilePath}', function(){
    before(function () {
        {#dependencies}
            sinon.stub({.moduleName}, '{.funName}');
        {/dependencies}
    });
    after(function() {
        {#dependencies}
            {.moduleName}.{.funName}.restore();
        {/dependencies}
    });
    it('{funName}', function() {
        {#dependencies}
            {.moduleName}.{.funName}.returns({.returnValues[0]|se|s});
        {/dependencies}

        var args = [];
        {#args}
          args.push({.|se|s});
        {/args}

        var spy = sinon.spy(sut, '{funName}');
        sut.{funName}.apply(sut, args);

        var spyCall = spy.getCall(0);
        {#returnValue}
            sinon.assert.match(spyCall.returnValue, {returnValue|se|s});
        {/returnValue}
    });
})
```

B. UNIT TEST

```
'use strict';
var fs = require('fs');
var sinon = require('sinon'),
    sut = require('../../../demo/demo.js');
describe('tests/unit/demo/demo.js-1407772455462.js', function() {
    beforeEach(function() {
        sinon.stub(fs, 'readFileSync');
    });
    afterEach(function() {
        fs.readFileSync.restore();
    });
    it('f', function() {
        fs.readFileSync.returns([123, 10, 32, 32,...,2, 32, 10, 125]);
        var args = [];
        args.push("/Users/gdhuang/Documents/atg/demo/../config/default.json");
        var spy = sinon.spy(sut, 'f');
        sut.f.apply(sut, args);
        var spyCall = spy.getCall(0);
        sinon.assert.match(spyCall.returnValue, [123, 10, 32, 32,...,2, 32, 10, 125]);
    });
})
```