Spekl Documentation

Release 0.0.6

John L. Singleton

August 31, 2015

Contents

Contents:

CHAPTER 1

Getting Started with Spekl

How can we know that our software does what it is supposed to do? Techniques like unit testing are good for increasing our confidence that a program does what it is supposed to do, but ultimately they are weak approximations. Often times it's impossible to encode all the possible edge cases into a unit test, and if it is possible, it may be extremely time consuming to do so.

What do we do? Enter Formal Methods.

Formal Methods is an area of Computer Science that aims to address the problem of verifying what software does by using mathematical models and techniques. It's also a term that strikes fear into the hearts of productive engineers everywhere.

Most Formal Methods techniques involve specifications, static checkers, runtime assertion checkers, and SMT-solvers. However, getting them to work together is often difficult and error-prone. Spekl is a platform for streamlining the process of authoring, installing, and using specifications and formal methods tools.

CHAPTER 2

Installation

Installing Spekl is easy. To install, simply download the installer for your platform from Spekl's releases releases page and run it.

At the moment, users on Linux and OSX are required to have the Git binaries installed on your path. This requirement will be lifted in future versions of Spekl. Windows users don't need to have Git installed.

CHAPTER 3

Your First Verification Project

In this section we're going to see just how easy it is to verify your programs using Spekl. Spekl supports lots of different tools like OpenJML, SAW, and FindBugs. In this section we are going to perform extended static checking on a small application to show you how easy Spekl makes the verification process.

To start, create a new directory you want to store this example in:

```
~ » mkdir my-project
~ » cd my-project
```

Next, initialize the project in that directory. You can do this with the `spm init` command. This command is interactive but for this example we are going to just accept the defaults `spm init` uses.

```
~ » spm init
[spm] INFO  - [command-init]
[spm] INFO  - [new-project] Creating new verification project...
Project Name? [default: my project]
Project Id? [default: my.project]
Project Version? [default: 0.0.1]
#
# Basic Project Information
#
name            : my project
project-id      : my.project
version         : 0.0.1

#
# Checks
#
# hint: Use spm add <your-tool> to add new checks here
#

##
## Example
##
# checks :
#   - name        : openjml-esc
#     description : "OpenJML All File ESC"
#     language    : java              # might not need this, because it is implied by the tool
#     paths       : [MaybeAdd.java]

#     tool:
#       name      : openjml-esc
#       version   : 0.0.3
```

```
#       pre_check :  # stuff to do before a check
#       post_check:  # stuff to do before a check

#     # specs:
#     #   - name: java-core
#     #     version: 1.1.1


Does this configuration look reasonable? [Y/n] y
[spm] INFO  - [new-project] Writing project file to spekl.yml
[spm] INFO  - [new-project] Done.
```

This command creates a file called `spekl.yml` in the directory you execute `spm init` in. Edit that file to look like the listing, below.

```
#
# Basic Project Information
#
name             : my project
project-id       : my.project
version          : 0.0.1

checks :
  - name         : openjml-esc
    description  : "OpenJML All File ESC"
    paths        : [MaybeAdd.java]

    tool:
      name       : openjml-esc
      pre_check :  # stuff to do before a check
      post_check:  # stuff to do before a check
```

What did we do in the listing, above? In the checks section we defined a check called `openjml-esc`. This is the extended static checker provided by OpenJML, a tool that is able to check programs written in the JML Specification Language. You don't need to know JML to follow this example, but JML is an excellent modeling language that is widely known (meaning, you should probably learn it).

Continuing with the example above, we defined just one check here. Note that we have specified that we want to use OpenJML declaratively — we haven't specified *how* to use OpenJML. Also note that OpenJML depends on things like SMT solvers which may be difficult for new users to configure. We haven't needed to specify anything about them, either.

Note that in the `paths` element we specified that we want to check the file `MaybeAdd.java`. We'll create this file next. Note that the `paths` element can contain a comma-separated list of paths that may contain wildcards. You use this to specify the files you want to run a given check on.

Next, put the following text into the file `MaybeAdd.java` in the current directory

```
public class MaybeAdd {

  //@ requires 0 < a && a < 1000;
  //@ requires 0 < b && b < 1000;
  //@ ensures  0 < \result;
  public static int add(int a, int b){
      return a-b;
  }


  public static void main(String args[]){
```

```
      System.out.println(MaybeAdd.add(1,2));

  }

}
```

In this minimal class you can see that we wrote a minimal example that (wrongly) adds two integers. Let's see what happens when we run this example with Spekl. To do that, first let's tell Spekl to install our tools:

```
~ » spm install
```

This command will kick off an installation process that will install `z3`, `openjml`, and `openjml-esc`. The output will look like the following:

```
[spm] INFO  - [command-install] Finding package openjml-esc in remote repository
[spm] INFO  - [command-install] Starting install of package openjml-esc (version: 1.7.3.20150406-5)
[spm] INFO  - [command-install] Examining dependencies...
[spm] INFO  - [command-install] Will install the following missing packages:
[spm] INFO  - [command-install] -  openjml (version: >= 1.7.3 && < 1.8)
[spm] INFO  - [command-install] -  z3 (version: >= 4.3.0 && < 4.3.1)
[spm] INFO  - [command-install] Finding package openjml in remote repository
[spm] INFO  - [command-install] Starting install of package openjml (version: 1.7.3.20150406-1)
[spm] INFO  - [command-install] Examining dependencies...
[spm] INFO  - [command-install] Installing package openjml (version: 1.7.3.20150406-1)
[spm] INFO  - [command-install] Downloading Required Assets...
openjml-dist         : [================================================] 100%
[spm] INFO  - [command-install] Running package-specific installation commands
[spm] INFO  - [command-install-scripts]  Unpacking the archive...
[spm] INFO  - [command-install] Performing cleanup tasks...
[spm] INFO  - [command-install] Cleaning up resources for asset  openjml-dist
[spm] INFO  - [command-install] Writing out package description...
[spm] INFO  - [command-install] Completed installation of package openjml (version: 1.7.3.20150406-1)
[spm] INFO  - [command-install] Finding package z3 in remote repository
[spm] INFO  - [command-install] Starting install of package z3 (version: 4.3.0-2)
[spm] INFO  - [command-install] Examining dependencies...
[spm] INFO  - [command-install] Installing package z3 (version: 4.3.0-2)
[spm] INFO  - [command-install] Downloading Required Assets...
Z3 Binaries for Windows : [================================================] 100%
[spm] INFO  - [command-install] Running package-specific installation commands
[spm] INFO  - [command-install-scripts]  Unpacking Z3...
[spm] INFO  - [command-install] Performing cleanup tasks...
[spm] INFO  - [command-install] Cleaning up resources for asset  Z3 Binaries for Windows
[spm] INFO  - [command-install] Writing out package description...
[spm] INFO  - [command-install] Completed installation of package z3 (version: 4.3.0-2)
[spm] INFO  - [command-install] Installing package openjml-esc (version: 1.7.3.20150406-5)
[spm] INFO  - [command-install] Downloading Required Assets...
[spm] INFO  - [command-install] Running package-specific installation commands
[spm] INFO  - [command-install] Performing cleanup tasks...
[spm] INFO  - [command-install] Writing out package description...
[spm] INFO  - [command-install] Completed installation of package openjml-esc (version: 1.7.3.201504
[spm] INFO  - [command-install] Installing specs....
[spm] INFO  - [command-install] Done. Use `spm check` to check your project.
```

After that completes, we can run a check with the following command:

```
~ » spm check
```

The output from the check will look like the following:

```
[spm] INFO  - [command-check] Running all checks for project...
[spm] INFO  - [command-check] Running check: OpenJML All File ESC
[spm] INFO  - Configuring solver for Z3...
[spm] INFO  - Running OpenJML in ESC Mode...

.\MaybeAdd.java:7: warning: The prover cannot establish an assertion (Postcondition: .\MaybeAdd.java
        return a-b;
        ^
.\MaybeAdd.java:5: warning: Associated declaration: .\MaybeAdd.java:7:
    //@ ensures  0 < \result;
        ^
.\MaybeAdd.java:13: warning: The prover cannot establish an assertion (Precondition: .\MaybeAdd.java
        System.out.println(MaybeAdd.add(1,2));
                                        ^
.\MaybeAdd.java:3: warning: Associated declaration: .\MaybeAdd.java:13:
    //@ requires 0 < a && a < 1000;
        ^
4 warnings
```

As you can see in the output above, the extended static checker has correctly detected that our implementation did not satisfy the specification. Let's fix that. To do that, replace the − operation in the `MaybeAdd` class with +. Your listing should look like the following:

```
public class MaybeAdd {

  //@ requires 0 < a && a < 1000;
  //@ requires 0 < b && b < 1000;
  //@ ensures  0 < \result;
  public static int add(int a, int b){
      return a+b;
  }


  public static void main(String args[]){

      System.out.println(MaybeAdd.add(1,2));

  }

}
```

Let's see if this works now:

```
~ » spm check
```

The output from the check will look like the following:

```
[spm] INFO  - [command-check] Running all checks for project...
[spm] INFO  - [command-check] Running check: OpenJML All File ESC
[spm] INFO  - Configuring solver for Z3...
[spm] INFO  - Running OpenJML in ESC Mode...
```

Since OpenJML didn't emit any errors, it means that the code we wrote satisfies the specifications.

CHAPTER 4

Next Steps

This is just a sample of the many things you can do with Spekl. As a user of Spekl most of your work will consist of adding and running checks. To browse some of the available checks, head over to the recipes section, here: *What are Spekl Recipes?*.

CHAPTER 5

What are Spekl Recipes?

Spekl makes it easy to drop in new verification checks into your projects. To that end we've created an easy to use reference of available checks in Spekl we call "Recipes." Browse the sections below to find out the kinds of checks you can add to your programs.

CHAPTER 6

OpenJML

OpenJML is a suite of tools for editing, parsing, type-checking, verifying (static checking), and run-time checking Java programs that are annotated with JML statements stating what the program's methods are supposed to do and the invariants the data structures should obey. JML annotations state preconditions, postconditions, invariants and the like about a method or class; OpenJML's tools will then check that the implementation and the specifications are consistent.

The Java Modeling Language (JML) is a behavioral interface specification language (BISL) that can be used to specify the behavior of Java modules. It combines the design by contract approach of Eiffel and the model-based specification approach of the Larch family of interface specification languages, with some elements of the refinement calculus.

More About this Tool:

- JML Specification Language
- OpenJML Project Homepage

6.1 Runtime Assertion Checking

6.2 Extended Static Checking

CHAPTER 7

FindBugs

FindBugs uses static analysis to inspect Java bytecode for occurrences of bug patterns. Static analysis means that FindBugs can find bugs by simply inspecting a program's code: executing the program is not necessary. This makes FindBugs very easy to use: in general, you should be able to use it to look for bugs in your code within a few minutes of downloading it. FindBugs works by analyzing Java bytecode (compiled class files), so you don't even need the program's source code to use it. Because its analysis is sometimes imprecise, FindBugs can report false warnings, which are warnings that do not indicate real errors. In practice, the rate of false warnings reported by FindBugs is less than 50%.

More About this Tool:

- FindBugs Project Homepage

7.1 Run FindBugs and Generate HTML Reports

```
checks :
  - name        : findbugs-html
    description : "FindBugs HTML Report"
    check       : html
    paths       : [A.class]  # your class files

    tool:
      name      : findbugs
```

7.2 Run FindBugs and Generate XML Reports

```
checks :
  - name        : findbugs-xml
    description : "FindBugs XML Report"
    check       : xml
    paths       : [A.class]  # your classfiles

    tool:
      name      : findbugs
```

CHAPTER 8

SAW

The Software Analysis Workbench (SAW) provides the ability to formally verify properties of code written in C, Java, and Cryptol. It leverages automated SAT and SMT solvers to make this process as automated as possible, and provides a scripting language, called SAW Script, to enable verification to scale up to more complex systems.

More About this Tool:

- Galois Homepage
- SAW Project Homepage

8.1 Verify that Two Implementations are Equivalent

```
checks :
  - name        : saw
    description : "SAW"
    check       : equiv-c
    paths       : [] #
    reference:
      file     : ffs_ref.c   # the reference file
      function : ffs_ref     # the reference function
    test:
      file     : ffs_test.c  # the file to check
      function : ffs_test    # the function to check

    tool:
      name      : saw
```

CHAPTER 9

Checker Framework

Are you tired of null pointer exceptions, unintended side effects, SQL injections, concurrency errors, mistaken equality tests, and other run-time errors that appear during testing or in the field?

The Checker Framework enhances Java's type system to make it more powerful and useful. This lets software developers detect and prevent errors in their Java programs. The Checker Framework includes compiler plug-ins ("checkers") that find bugs or verify their absence. It also permits you to write your own compiler plug-ins.

More About this Tool:

- Checker Framework Homepage

9.1 Nullness Checker

CHAPTER 10

Indices and tables

- genindex
- modindex
- search