Spoon
Getting started ▼
Querying source code elements ▼
Code Transformation ▼
Code Transformation
Transformation Examples
Transformation with annotations
Transformation with Templates
Semantic patching
Code Generation
Testing transformations
Usage ▼
Spoon Meta model ▼

Semantic patching

The spoon-smpl submodule provides a prototype implementation of a subset of SmPL (Semantic Patch Language) for a subset of Java. SmPL patches can be thought of as traditional plain text patches with enhanced expressiveness thanks to support for the syntax and semantics of specific programming languages. For example, a Java SmPL patch can be written to specify a generic transformation that reorders of a series of method call arguments without having to worry about matching any specific literal variable names or other literal argument expressions.

Installation

On a Unix-like system, the following set of commands should be sufficient for getting spoon-smpl up and running from scratch.

```
$ git clone -b smpl https://github.com/mkforsb/spoon.git
$ cd spoon/spoon-smpl
$ mvn package
$ ./tools/smplcli.sh
usage:
smplcli ACTION [ARG [ARG ..]]

    ACTIONs:
        patch        apply SmPL patch
                     requires --smpl-file and --java-file

        check        run model checker
                     requires --smpl-file and --java-file

        checksub     run model checker on every subformula
                     requires --smpl-file and --java-file

        rewrite      rewrite SmPL input
                     requires --smpl-file

        compile      compile SmPL input
                     requires --smpl-file

        ctl          compile and print CTL formula
                     requires --smpl-file

    ARGs:
        --smpl-file FILENAME
        --java-file FILENAME
```

Alternatively, the command line application can be invoked directly as:

```
$ java -cp <classpath> spoon.smpl.CommandlineApplication
```

Basic usage

The basic use case of spoon-smpl involves at minimum two files: one .java source file and one semantic patch. For this tutorial, we will use the following two files:

File 1: example semantic patch (patch.smpl)

```
@@
type T;
identifier ret;
constant C;
@@
- T ret = C;
... when != ret
- return ret;
+ return C;
```

This example patch removes local variables only used to return a constant.

File 2: example Java source (Program.java)

```
public class Program {
    public int fn1() {
        int x = 1;
        return x;
    }

    public int fn2(boolean print) {
        int x = 2;

        if (print) {
            System.out.println("hello from fn2");
        }
```

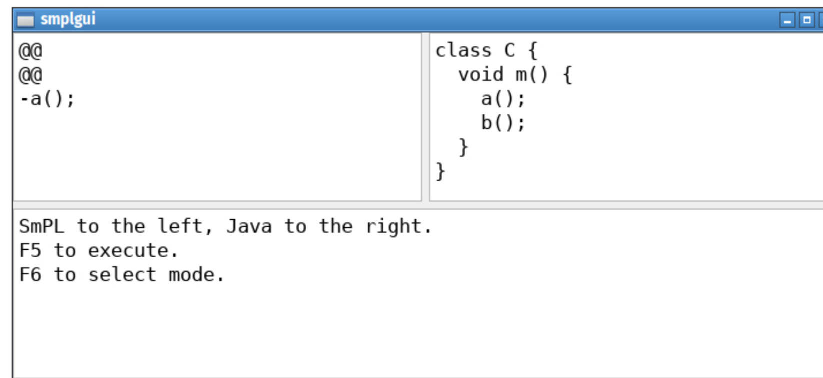
```
        return x;
    }

    public int fn3(boolean print) {
        int x = 3;

        if (print) {
            System.out.println(x);
        }

        return x;
    }
}
```

We then apply the semantic patch to the Java source code as follows (output also shown):

```
$ ./tools/smplcli.sh patch --smpl-file patch.smpl --java-file Program.java

public class Program {
    public int fn1() {
        return 1;
    }

    public int fn2(boolean print) {
        if (print) {
            java.lang.System.out.println("hello from fn2");
        }
        return 2;
    }

    public int fn3(boolean print) {
        int x = 3;
        if (print) {
            java.lang.System.out.println(x);
        }
        return x;
    }
}
```

Graphical interface

There is a very simple graphical interface available in `tools/smplgui.py`. This tool requires Python 3 and a suitable `python3-pyqt5` package providing Qt5 bindings, in particular the module `PyQt5.QtGui`. Furthermore, the tool currently assumes it is executing on a Unix-like system from a working directory in which the file `./tools/smplcli.sh` is available to run spoon-smpl. As such, it is recommended to start the tool from the spoon-smpl root folder using the command `$ ./tools/smplgui.py`.



The tool provides two panes for editing the semantic patch and some Java source code, respectively. The upper left pane contains the semantic patch, while the upper right pane contains the Java source code. Finally, the tool provides a number of modes for invoking spoon-smpl using the inputs shown in the two panes. To change mode one presses the F6 key followed by the key corresponding to the desired mode, as shown in the image below. To execute the currently selected mode, one presses the F5 key.

```
Choose mode by pressing the corresponding key:
F1: check
F2: checksub
F3: rewrite
F4: compile
F5: patch
F6: ctl
F7: gentest
```

These modes correspond to the `ACTION` alternatives present in `spoon.smpl.CommandLineApplication`, with the addition of the `gentest` mode which generates a test case in a special format for the inputs present in the two upper panes.

Batch processing

Spoon-smpl provides a batch processing mode in which a single semantic patch is applied to a full source tree recursively. This mode is implemented in the form of a Spoon `Processor` that also features a `main` method. The following example command is intended

to be executed in the spoon-smpl root directory, where a call to `mvn package` has placed a full suite of `.jar` files in the `./target` sub-directory.

```
$ java -cp $(for f in target/*.jar; do echo -n $f:; done) spoon.smpl.SmPLProcessor \
       --with-diff-command "bash -c \"diff -U5 -u {a} {b}\""                        \
       --with-smpl-file "path/to/patch.smpl"                                        \

       ## The following options are passed to spoon.Launcher, more may be added
       -i "path/to/target/source"                                                   \
       -o "path/to/write/output"                                                    \
       -p spoon.smpl.SmPLProcessor
```

The expression `-cp $(for f in target/*.jar; do echo -n $f:; done)` collects and places on the classpatch all `.jar` files found in the `target` sub-directory.

The `--with-diff-command` option expects a shell-executable command string containing the placeholder expressions `{a}` and `{b}`. The placeholders are substituted for the full paths to the pretty-printed input and the pretty-printed output respectively, for each modified file in the source tree. For example, in the event that spoon-smpl during batch processing has modified a file `Program.java`, the option used in the example command would result in a command akin to the following being executed:

```
bash -c "diff -U5 -u /tmp/9qKMH/Program.java /tmp/CYd40/Program.java"
```

Developing

The following code shows the core workflow of spoon-smpl, and is intended to guide developers towards finding the code for the component(s) of interest:

```
boolean tryApplyPatch(String plainTextSmPLCode, CtExecutable patchTargetExe) {
    // Parse a plain text SmPL patch
    SmPLRule rule = SmPLParser.parse(plainTextSmPLCode);

    // Create the CFG from the executable block
    SmPLMethodCFG cfg = new SmPLMethodCFG(patchTargetExe);

    // Create the CTL model from the CFG
    CFGModel model = new CFGModel(cfg);

    // Create the model checker
    ModelChecker checker = new ModelChecker(model);

    // Run the model checker on the formula that encodes the SmPL patch
    // This uses the visitor pattern
    //   We ask the formula tree to accept the model checker visitor
    rule.getFormula().accept(checker);

    // Fetch the results
    ModelChecker.ResultSet results = checker.getResult();

    // If we get an empty result, there were no matches
    // If we get no witnesses, there were no transformations to apply
    if (results.isEmpty() || results.getAllWitnesses().isEmpty()) {
        // Restore metamodel changes applied by SmPLMethodCFG
        model.getCfg().restoreUnsupportedElements();
        return false;
    }

    // Apply transformations
    Transformer.transform(model, results.getAllWitnesses());

    // Copy any new methods added by the patch
    if (rule.getMethodsAdded().size() > 0) {
        Transformer.copyAddedMethods(model, rule);
    }

    // Restore metamodel changes applied by SmPLMethodCFG
    model.getCfg().restoreUnsupportedElements();
    return true;
}
```