Running Examples

To clarify the use of RV-Monitor on the included examples, we will walk through the setup and running of several examples using the tool.

Preparation

There are several steps required to prepare to run the RV-Monitor examples.

First, the JDK and JRE must both be installed and on the path. javac and java must be able to run without errors. Then, RV-Monitor (http://runtimeverification.com/monitor) must be downloaded, installed, and added to the system PATH, with the rv-monitor command able to run without error.

Lastly, rv-monitor-rt.jar must both be added to your Java System classpath. See an article on modifying the system CLASSPATH (not the PATH) here (http://docs.oracle.com/javase/tutorial/essential/environment/paths.html). rv-monitor-rt.jar can be found in the lib directory of the RV-Monitor install folder. On bash, the change would look something like appending to your ~/.bashrc:

```
CLASSPATH=$CLASSPATH:[aspectjrt.jar path]:[rv-monitor-rt.jar path]:.
```

Make sure you add a "." to the classpath as well, making sure that Java also looks for sources in whatever your current working directory is. This is required to run the examples.

> **Note**
>
> On Windows, both the PATH and CLASSPATH are delimeted by a ";" rather than a ":". Using the wrong character will cause the examples to not work correctly.

If you do not add the two Jars we mentioned to the classpath, your commands will look something like:

```
$ javac HasNext_1/HasNext_1.java rvm/HasNextRuntimeMonitor.java -cp ~/RV-Monitor/lib/rv-monitor-rt.jar
```

instead of

```
$ javac HasNext_1/HasNext_1.java rvm/HasNextRuntimeMonitor.java
```

and

```
$ java HasNext_1.HasNext_1 -cp ~/RV-Monitor/lib/rv-monitor-rt.jar:.
```

instead of

```
$ java HasNext_1.HasNext_1
```

RV-Monitor + Java

Welcome! (../)

Quickstart (../quickstart/)

Running Examples ()

- Preparation
- RV-Monitor + Java
  - HasNext property
  - SafeFileWriter property
  - As an agent
- RV-Monitor + C
- propertydb
- JavaMOP
- Analyzing logs

HasNext property

The first example we will explore uses the FSM formalism to express the Java API property that the next method of an iterator must not be called without the hasNext method being previously called, to exclude the possibility of a null or exception for safety. The property is as follows:

```
full-binding HasNext(Iterator i) {
    event hasnext(Iterator i) {} // after
    event next(Iterator i) {} // before

    fsm :
        start [
            next -> unsafe
            hasnext -> safe
        ]
        safe [
            next -> start
            hasnext -> safe
        ]
        unsafe [
            next -> unsafe
            hasnext -> safe
        ]

    alias match = unsafe

    @match {
        System.out.println("next called without hasNext!");
    }
}
```

To see this property prevent unsafe accesses in real code (the source of which you can and should explore), use the following commands:

```
$ cd examples/FSM/HasNext
$ rv-monitor rvm/HasNext.rvm
$ javac rvm/HasNextRuntimeMonitor.java HasNext_1/HasNext_1.java
$ java HasNext_1.HasNext_1
```

Note that RV-Monitor reports that next has been called without hasNext four times. Is this consistent with the expected behavior of HasNext_1.java?

To run the second example, there is no need to resynthesize the monitoring library. Simply

```
$ javac HasNext_2/HasNext_2.java
$ java HasNext_2.HasNext_2
```

This time, RV-Monitor reports no errors, implying that our program is correct with regards to calls to hasNext on an iterator, the property we defined.

> **Note**
>
> All provided commands will work in Windows, though in Windows the canonicalized commands would use “\” rather than “/” as a path separator.

SafeFileWriter property

We now consider a new property, SafeFileWriter.rvm. This property is written in the ERE formalism, using standard syntax for extended regular expressions. The property is designed to ensure that there are no writes to a file after it is closed, again a property of the Java API and potential source for program bugs or errors. The property is as follows:

```
SafeFileWriter(FileWriter f) {
    static int counter = 0;
    int writes = 0;

    event open(FileWriter f) { //after
        this.writes = 0;
    }
    event write(FileWriter f) { //  before
        this.writes ++;
    }
    event close(FileWriter f) {} //  after

    ere : (open write write* close)*

    @fail {
        System.out.println("write after close");
        __RESET;
    }
    @match {
        System.out.println((++(counter))
             + ":" + writes);
    }
}
```

Unlike in the previous property, where the FSM defined an invalid execution of the program and the matching of a state indicates an error, this regular expression instead defines the *correct* execution of a FileWriter in Java. A file must first be opened, written to some number of times, then closed. The correct execution trace for a given File object is this sequence of events happening some number of times. We thus arrive at the property above, and test it with two small pieces of example code.

To run SafeFileWriter from the RV-Monitor root:

```
$ cd examples/ERE/SafeFileWriter
$ rv-monitor rvm/SafeFileWriter.rvm
$ javac rvm/SafeFileWriterRuntimeMonitor.java
SafeFileWriter_1/SafeFileWriter_1.java
$ java SafeFileWriter_1.SafeFileWriter_1
```

Note the error messages generated, indicating unsafe use of the FileWriter object.

To run the second example:

```
$ javac SafeFileWriter_2/SafeFileWriter_2.java
$ java SafeFileWriter_2.SafeFileWriter_2
```

This time, no errors are reporting, again indicating correct execution.

To run the same example (SafeFileWriter) with a different formalism (LTL) simply run the same commands as above from the RV-Monitor root, replacing the first command with:

```
$ cd examples/LTL/SafeFileWriter
```

Note that the specification is identical, other than the property, which is now:

```
ltl : [](write => (not close S open))
```

This linear temporal logic means that at a write, there should not have been in the past a close, and that open must have occured after the start. This represents the same property as the regular expression, and demonstrates the ability to use multiple formalisms depending on the knowledge and desires of the property developer.

As an agent

As part of the RV-Monitor distribution, we also provide an agent that is pre-compiled with over 200 formal properties of four packages in the Java API (see the propertydb section below). It is possible to use this agent with any Java application.

All of the above Java examples can also be run with the provided RV-Monitor Java agent. The `examples/agent` directory contains several Java files similar to those in the above examples but with no instrumentation or RV-Monitor related code whatsoever. It is possible to automatically instrument and monitor all of these programs through the following commands:

```
$ cd examples/agent
$ javac [program].java
$ java -javaagent:rv-monitor-all.jar [program]
```

You can also replace Java with the `rv-monitor-all` command in any Java program execution as follows:

```
$ cd examples/agent
$ javac [program].java
$ rv-monitor-all [program]
```

to automatically add and execute the agent.

No changes to the CLASSPATH or PATH are required with this method, which bundles all required dependencies in the agent. Any of these programs can also be run unmonitored without the agent by simply omitting the `javaagent` flag above. To generate such agents with configurable sets of properties, we use the JavaMOP tool (see below).

RV-Monitor + C

RV-Monitor is available for C as well as Java, allowing for the generation of C monitoring libraries. We briefly mention this project through a single example, seatbelt.rvm. This is meant to simulate the case in which the state of a vehicle is being monitored as part of a C program, for example during the monitoring of bus traffic between control systems on a vehicle.

The seatbelt property is as follows:

```
SeatBelt {
    event seatBeltRemoved(){fprintf(stderr, "Seat belt removed.\n");}

    event seatBeltAttached() {fprintf(stderr, "Seat belt attached.\n");}

    fsm : unsafe [
         seatBeltAttached -> safe
        ]
        safe [
            seatBeltRemoved -> unsafe
        ]

    @safe {
        fprintf(stderr, "set max speed to user input.\n");
    }

    @unsafe {
        fprintf(stderr, "set max speed to 10 mph.\n");
    }
}
```

To run the C example basic_car, use following commands would be used:

> **Note**
>
> This example may not run on Windows without gcc and make installed.

```
$ make
$ ./test1
$ ./test2
```

Note that we do not currently bundle C capabilities as part of the free RV-Monitor. If you are interested in RV-Monitor for C, contact us (http://runtimeverification.com/support).

propertydb

Another important component of the RV-Monitor ecosystem is propertydb, which is a database of over 200 real production-quality properties related to the Java and Android API's. Information and documentation on this project is available on its Github (http://github.com/runtimeverification/property-db), however, we will present one property for the sake of discussion.

This property, Closeable_MultipleClose.rvm represents a property of the Java API:

```
Closeable_MultipleClose(Closeable c) {
    event close(Closeable c) {
    }

    ere: close close+

    @match {
        MOPLogging.out.println(Level.CRITICAL, __DEFAULT_MESSAGE);
        MOPLogging.out.println(Level.CRITICAL, "close() was invoked
multiple times.");
    }
}
```

Note the ease with which we can define a generic property and apply it to multiple applications regardless of their implementation details. RV-Monitor properties are inherently reuseable due to their abstraction of the concrete implementation details of a program into events, which provide the basis for the logic formulae we use.

JavaMOP

The same examples which we ran above with RV-Monitor are also compatible with JavaMOP. JavaMOP allows for the use of a single file to both generate monitoring libraries for your code and insert them in your codebase automatically through AspectJ. Please peruse the .mop files we compile here with javamop to see the syntax that provides this instrumentation, and consider the code of the examples (eg - HasNext_1/HasNext_1.java). In our RV-Monitor examples, these examples were required to import the runtime monitors and call their event methods directly. In JavaMOP, there is no mention of any monitoring code in the codebase being monitored at all, providing complete separation of monitoring and instrumentation.

```
$ cd JavaMOP/examples/FSM/HasNext
$ javamop HasNext.mop
$ ajc HasNextMonitorAspect.aj HasNext_1/HasNext_1.java -1.6 -d HasNext_1/
$ cd HasNext_1
$ java HasNext_1

$ cd ..
$ ajc HasNextMonitorAspect.aj HasNext_2/HasNext_2.java -1.6 -d HasNext_2/
$ cd HasNext_2
$ java HasNext_2

$ cd JavaMOP/examples/ERE/SafeFileWriter
$ javamop SafeFileWriter.mop
$ ajc SafeFileWriterMonitorAspect.aj
SafeFileWriter_1/SafeFileWriter_1.java -1.6 -d SafeFileWriter_1/
$ cd SafeFileWriter_1/
$ java SafeFileWriter_1

$ cd ..
$ ajc SafeFileWriterMonitorAspect.aj
SafeFileWriter_2/SafeFileWriter_2.java -1.6 -d SafeFileWriter_2/
$ cd SafeFileWriter_2/
$ java SafeFileWriter_2
```

Refer to the JavaMOP README for more information on using JavaMOP.

Analyzing logs

In addition to monitoring software execution, RV-Monitor is able to check logical properties over text-based log files. These properties can be anything that is Turing computable, and do not require storing the entire log files. This makes RV-Monitor ideal for in-depth analysis of large logfiles which may be impractical to analyze with traditional techniques like grep.

An example of RV-Monitor being used to analyze log files is bundled with the distribution. To compile and run the example, use the following commands:

```
$ cd examples/FSM/PostfixLog
$ rv-monitor rvm/MultipleConnectionCheck.rvm
$ rv-monitor rvm/UserSessionLog.rvm
$ javac PostfixLogAdapter.java rvm/*.java
$ java PostfixLogAdapter
```

PostfixLogAdapter contains the logic for turning log lines into RV-Monitor events, capturing the useful information therein with a regular expression and calling the appropriate event with the correct parameters as follows:

```
// Define a pattern for each RV-Monitor event
Pattern connectPattern = Pattern.compile("(.*?)( " + username +
    ")(.*)(\\[\\d+\\]: connect from )(.*)");
Pattern disconnectPattern = Pattern.compile("(.*?)( " + username +
    ")(.*)(\\[\\d+\\]: disconnect from )(.*)");
Pattern submitMessagePattern = Pattern.compile("(.*?)( " + username +
    ")(.*)(\\[\\d+\\]: )(.*)(: client=)(.*)");
Pattern failToSendPattern = Pattern.compile("(.*?)( " + username +
    ")(.*)(\\[\\d+\\]: )(.*)(reject: RCPT from )(.*?)(: )(.*)");
// If any pattern matches the line being processed, fire the RV-Monitor
event with appropriate parameters
Matcher connectLineMatcher = connectPattern.matcher(line);
while (connectLineMatcher.find()) {

UserSessionLogRuntimeMonitor.connectEvent(getUser(connectLineMatcher.group(5)),

        connectLineMatcher.group(1));

MultipleConnectionCheckRuntimeMonitor.connectEvent(getUser(connectLineMatcher.group(5)),

        connectLineMatcher.group(1));
}
...
```

If needed, such files can also be generated (though not with our provided tools) from simple configuration files defining a regular expression and the location of the parameters of each RV-Monitor event therein.