Generating Formal Models with the Java Symbolic Simulator

Galois, Inc. | 421 SW 6th Avenue, Suite 300 | Portland, OR 97204

Introduction

This document provides a step-by-step guide to using the command-line version of Galois' Java Symbolic Simulator, `jss`, to perform rigorous mathematical analysis of Java programs. To illustrate its use, we describe how to formally prove that a Java implementation of the MD5 message digest algorithm is functionally equivalent to a reference specification. The ability to perform this sort of proof brings benefits such as allowing programmers to experiment with efficient, customized implementations of an algorithm while retaining confidence that the changes do not affect the overall functionality.

We assume knowledge of Java, and a rough understanding of cryptography. However, we do not assume familiarity with symbolic simulation, formal modeling, or theorem proving. Additionally, following along with the entire tutorial will require the use of the Cryptol tool set [1] and the ABC logic synthesis system from UC Berkeley [2].

Installation and configuration of those tools is outside the scope of this tutorial. However, instructions on setting up the environment for `jss` (i.e., installing the JDK 1.6 and Bouncy Castle class files) can be found in the next section.

In the examples of interaction with the simulator and other tools, lines beginning with a hash mark (`#`) or short text followed by an angle bracket (such as `abc 01>`) indicate command-line prompts, and the following text is input provided by the user. All other uses of monospaced text indicate representative output of running a program, or the contents of a program source file.

Setting up the Environment

The `jss` tool requires the classes that form Java's runtime in order to execute most useful Java code; the current `jss` release has been tested with JDK 1.6. Furthermore, this tutorial relies on the Bouncy Castle implementation of MD5, and so some supporting class files are required for that as well. Fortunately, this is as simple as installing or locating two Java JAR files to provide as arguments to `jss`.

The JDK JAR File

The location of the JDK JAR file is platform-dependent.

If you're on Windows or Linux, download the appropriate Java SE 6 JDK via the web forms at:

http://www.oracle.com/technetwork/java/javase/downloads/index.html

The page lists the Java 7 JDK first, which hasn't been thoroughly tested with `jss`. For now, the Java 6 JDK is recommended.

Once the JDK is is installed, the main runtime library (called `rt.jar`) can be found at:

```
<JDK INSTALLATION ROOT>\jre\lib\rt.jar
```

[1]Galois, Inc. Cryptol. http://corp.galois.com/cryptol
[2]Berkeley Logic Synthesis and Verification Group. {ABC}: A System for Sequential Synthesis and Verification http://www.eecs.berkeley.edu/~alanmi/abc/

e.g.: `C:\jdk1.6.0_22\jre\lib\rt.jar`

On Linux, the path is as above, rooted at the location where the Java 6 JDK was extracted, and using forward slashes (/) to separate path components.

If you're on Mac OS X, the JDK should already be installed along with the platform developer tools. Executing the shell command `locate classes.jar` should reveal the location of a JDK 6 runtime jar suitable for use with `jss`. E.g.,

```
/System/Library/Java/JavaVirtualMachines/1.6.0.jdk/Contents/Classes/classes.jar
```

In either case, the identified file (either `rt.jar` or `classes.jar`) should be the file substituted for `JDK_JAR` in the command line invocations shown in subsequent sections.

The Bouncy Castle JAR File

To obtain code needed for the Bouncy Castle MD5 implementation employed in this tutorial, download the Bouncy Castle JAR file via

http://www.bouncycastle.org/download/bcprov-jdk16-145.jar

This should be the file substituted for `BC_JAR` in the command line invocations shown in subsequent sections.

Galois Symbolic API

The final component necessary for successful symbolic simulation of a Java program is the JAR file for Galois symbolic simulator API, which provides a collection of utility methods useful for controlling symbolic simulation.

The simulator control API is in the file `galois.jar` in the `bin` directory of the distribution archive for `jss`. This should be the file substituted for `SYM_JAR` in the command lines shown later.

Compiling the Example

The example code for this tutorial is located in the same directory as the tutorial document itself in the path where you unpacked the `jss` distribution. Each section of this tutorial will list the commands necessary to follow along with the verification being described.

Before performing symbolic simulation or equivalence checking, the example wrapper around the MD5 algorithm must be compiled from Java source code to `.class` file containing byte code for the Java Virtual Machine (JVM), using the following command (replacing `JDK_JAR` and `BC_JAR` with the appropriate JAR file names for your system, as described earlier):

```
# javac -g -cp SYM_JAR:BC_JAR:JDK_JAR JavaMD5.java
```


This will result in a new file, `JavaMD5.class` in the same directory as the source file. The `-cp` option tells the compiler where to find code for the standard Java libraries and the Bouncy Castle implementaiont, needed to type-check the example. The `-g` option tells the compiler to include debugging information in the output, so that `jss` will be able to determine the names of local variables and method paremeters, as well as to determine which JVM instructions correspond with which lines of source code. This latter option can be omitted, but the debugging information can make the simulator more convenient to use.

NB: The colon character (`:`) is used to delimit JAR file names on UNIX systems (including Mac OS X). On Windows systems, the semicolon character (`;`) is used instead. For example, assuming that `jss` is in a directory listed the `%PATH%` environment variable, an invocation on a Windows system may look like:

```
# javac -g -j galois.jar;bcprov-jdk16-145.jar;"C:\jdk1.6.0_22\jre\lib\rt.jar" JavaMD5.java
```

Symbolic Simulation

The Java Symbolic Simulator takes the place of a standard Java virtual machine but makes it possible to reason about the behavior of programs on a wide range of potential inputs (in general, all possible inputs) rather than a fixed set of test vectors. The standard Java virtual machine executes programs on concrete input values, producing concrete results. Symbolic simulation works somewhat similarly, but allows inputs to take the form of symbolic variables that represent arbitrary, unknown values. The result is then a mathematical formula that describes the output of the program in terms of the symbolic input variables.

Given a formula representing a program's output, we can then either evaluate that formula with specific values in place of the symbolic input variables to get concrete output values, or compare the formula to another, using standard mathematical transformations to prove that the two are equivalent.

One downside of symbolic simulation, however, is that it cannot easily handle interaction with the outside world. If a program simply takes an input, performs some computation, and produces some output, symbolic simulation can reliably construct a model of that computation. Cryptographic algorithms typically fall into this category. In cases where a program does some computation, produces some output, reads some more input, and continues the computation based on the new information, we either need a model of the outside world, or to assume that the input could be completely arbitrary, and reason about what the program would do for any possible interactive input.

The Java Symbolic Simulator can evaluate simple output methods, such as the ubiquitous `System.out.println`. If its argument is known to be a specific, concrete value at the time that the simulator encounters the call, it prints out a string equivalent to what a program running under the normal JVM would print. When the argument is a symbolic expression involving some unknown inputs, however, the output is a representation of this symbolic expression.

However, when dealing with symbolic values, printing results is not necessarily the most useful mode of operation. Instead, the Java Symbolic Simulator provides a set of special methods for performing operations on symbolic values, and for emitting the formal model representing symbolic values of interest. The rest of this tutorial will demonstrate how to use these methods to generate a formal model of the MD5 digest algorithm, and then compare this model to a reference specification.

Supplying Symbolic Input to MD5

The directory containing this tutorial contains an example source file, `JavaMD5.java`, which creates a simple wrapper around the MD5 digest function from the The Legion of the Bouncy Castle. We will use this source file as a running example, and step through what each line means in the context of symbolic simulation.

The `JavaMD5` class first imports from `org.bouncycastle.crypto.digests` to get access to the MD5 digest code, and from `com.galois.symbolic` for access to the Java Symbolic Simulator's special methods for using symbolic values.

The first variable declaration in the `main` function is the one with the most relevance to symbolic simulation.

```
byte[] msg = Symbolic.freshByteArray(16);
```

This declaration creates a new array of bytes, but one with completely symbolic contents. Its size is fixed (16 elements), but each of those elements is a symbolic term representing an arbitrary byte value. In its current form, `jss` is not capable of reasoning about objects of unknown or unbounded size, so the user must choose a specific size for the input message.

The next two declarations are completely standard Java, and create a new, concrete array of 16 bytes along with a `MD5Digest` object.

```
byte[] out = new byte[16];
MD5Digest digest = new MD5Digest();
```

These variables have no direct connection to symbolic values upon declaration, but the values stored in them will be symbolic if they depend on the values in the `msg` array.

Next, calculation of the message digest happens in the usual way, by calling two methods in the Bouncy Castle API, just as they would occur in a typical application.

```
digest.update(msg, 0, msg.length);
digest.doFinal(out, 0);
```

The next and final statement does the work of creating a formal model from the MD5 digest code. This method instructs the symbolic simulator to generate a formula that describes how the value of the variable `out` depends on the values of the elements of `msg` and then write that formula to a file called `JavaMD5.aig`.

```
Symbolic.writeAiger("JavaMD5.aig", out);
```

Ultimately, we want to generate a formal model that describes the output of the digest function, in terms of whatever symbolic inputs it happens to depend on. In the case of MD5, this includes every byte of the input message. However, for some algorithms, it could include only a subset of the symbolic variables in the program.

The formal model that the simulator generates takes the form of an And-Inverter Graph (AIG), which is a way of representing a boolean function purely in terms of the logical operations `and` and `not`. The simplicity of this representation makes the models easy to reason about, and to compare to models from other sources. However, the same simplicity means that the model files can be very large in comparison to the input source code.

Running the Simulator

To generate a formal model from the example described in the previous section, we can use the `jss` command, which forms the command-line front end of the Java Symbolic Simulator. It needs to know where to find the Java class files for the standard library, the Bouncy Castle encryption libraries, and the Galois Symbolic libraries. The latter should be found automatically by the tool as long as the `jss` is being invoked from the distribution directory hierarchy. It also requires the compiled version of the program to simulate, created with `javac` as described earlier.

Given a compiled `JavaMD5.class` files, the following command will run `jss` to create a formal model; substitute `JDK_JAR` and `BC_JAR` with the appropriate JAR file names for your system, as described earlier.

```
# jss -c . -j SYM_JAR:BC_JAR:JDK_JAR JavaMD5
```

NB: As mentioned earlier, the path separator is different on Windows than on UNIX-based systems. On Windows, the command line will look something like the following:

```
# jss -c . -j galois.jar;bcprov-jdk16-145.jar;"C:\jdk1.6.0_22\jre\lib\rt.jar" JavaMD5
```

This will result in a file called `JavaMD5.aig` that can be furthier analyzed using a variety of tools, including the Galois Cryptol tool set, and the ABC logic synthesis system from UC Berkeley.

Verifying the Formal Model Using Cryptol

One easy way to verify a Java implementation against a reference specification is through the Cryptol tool set. Cryptol is a domain-specific language created by Galois for the purpose of writing high-level but precise specifications of cryptographic algorithms. The Cryptol tool set has built-in support for checking the equivalence of different Cryptol implementations, as well as comparing Cryptol implementations to external formal models, including models in AIG form.

This tutorial comes with two Cryptol files, `MD5.cry` and `compare-md5.cry`. The former is a Cryptol specification of the MD5 algorithm. In particular, it contains the function `md5_ref`, which is specialized to operate on 16-byte messages, and should have equivalent functionality to the Bouncy Castle implementation.

To compare the functionality of the two implementations, we have several options. As mentioned earlier, formal models can be evaluated on concrete inputs, or compared to other formal models using proof techniques to show equivalence for all possible inputs. The contents of `compare-md5.cry` show how to compare the formal model of the Java implementation against the Cryptol reference specification.

```
include "MD5.cry";
extern AIG md5_java("JavaMD5.aig") : [16][8] -> [128];
theorem MatchesRef : {m}. md5_java m == md5_ref m;
```

The first line imports the contents of `MD5.cry`, for access to the `md5_ref` function. Then, the `extern AIG` line makes the contents of `JavaMD5.aig` available as a function called `md5_java` that takes 16 8-bit values as input and produces one 128-bit value as output. Finally, the third line states a theorem: that the functions `md5_java` and `md5_ref` should produce the same output for all possible inputs.

We can load `compare-md5.cry` into the Cryptol tool set, yielding the following output:

```
# cryptol compare-md5.cry
Cryptol version 1.8.23, Copyright (C) 2004-2012 Galois, Inc.
                                          www.cryptol.net
Type :? for help
Loading "compare-md5.cry"..
  Including "MD5.cry".. Checking types..
  Loading extern aig from "JavaMD5.aig".. Processing.. Done!
*** Auto quickchecking 1 theorem.
*** Checking "MatchesRef" ["compare-md5.cry", line 4, col 1]
Checking case 100 of 100 (100.00%)
100 tests passed OK
[Coverage: 0.00%. (100/340282366920938463463374607431768211456)]
compare-md5>
```

By default, the Cryptol interpreter processes every `theorem` declaration by automatically evaluating the associated expression on a series of random values, and ensuring that it always yields `true`. In this case, it tried 100 random values, and the two functions yielded the same output in each case. However, the number of possible 16-byte inputs is immense, so 100 test cases barely scratches the surface.

To gain a higher degree of confidence that the functions do have the same functionality for all possible inputs, we can attempt to prove their equivalence deductively. From Cryptol's command line:

```
compare-md5> :set symbolic
compare-md5> :prove MatchesRef
Q.E.D.
compare-md5> :fm md5_ref "MD5-ref.aig"
```

This tells the Cryptol interpreter to switch to symbolic simulation mode (which is one way it can generate formal models from Cryptol functions) and then attempt to prove the theorem named `MatchesRef`. On a reasonably modern machine (as of March 2013), the proof should complete in several minutes. The output `Q.E.D.` means that the proof was successful.

Finally, the `:fm` command tells the interpreter to generate a formal model of the `md5_ref` function and store it in the file `MD5-ref.aig`. We can then use this formal model to perform the same proof using an external tool such as ABC, as described next.

Verifying the Formal Model Using ABC

ABC is a tool for logic synthesis and verification developed by researchers at UC Berkeley. It can perform a wide variety of transformations and queries on logic circuits, including those in the AIG form discussed earlier.

As an alternative approach to the equivalence check from the previous section, we can use the `cec` command in ABC to attempt to prove the model generated by the symbolic simulator equivalent to the model generated from the Cryptol specification.

```
# abc
UC Berkeley, ABC 1.01 (compiled Nov 21 2012 09:44:18)
abc 01> cec ./JavaMD5.aig ./MD5-ref.aig
Networks are equivalent.
abc 01>
```

Evaluating Formal Models on Concrete Inputs

So far, we have demonstrated how to use the API of the symbolic simulator to generate formal models, in the form of And-Inverter Graphs, that describe the symbolic values of particular program variables, and then use external tools to analyze those formal models. The API also provides the ability to evaluate a formal model on specific concrete inputs from within the simulator.

In the example from `JavaMD5.java`, the variable `out` depends on symbolic inputs and is therefore represented by a symbolic model. However, given concrete values for symbolic inputs that the output depends on, the model can be reduced to a concrete final value. Evaluation of a symbolic model to a concrete value uses one of the `Symbolic.evalAig` methods, depending on the type of the output variable of interest.

Because the symbolic model describing an output variable may have inputs of various different types, the symbolic simulator API provides a class hierarchy to represent possible input values. The `CValue` class represents concrete input values, and has inner subclasses `CBool`, `CByte`, `CInt`, and `CLong`, to represent inputs of a variety of sizes.

Given a model output variable and an array of concrete inputs, the `evalAig` function produces a new value of the same type as the given output variable, but one that is guaranteed to contain a concrete value from the perspective of the symbolic simulator. The caller of `evalAig` is responsible for providing the correct number of inputs. Otherwise, the simulator may throw a runtime error.

For example, if we replace the call to `Symbolic.writeAiger` in the example above with the following code, the simulator will print out the result of evaluating the symbolic model on a concrete message.

```
byte[] result = Symbolic.evalAig(out,
  new CValue[] {
    new CValue.CByte((byte) 0x68), // h
    new CValue.CByte((byte) 0x65), // e
    new CValue.CByte((byte) 0x6c), // l
    new CValue.CByte((byte) 0x6c), // l
    new CValue.CByte((byte) 0x6f), // o
    new CValue.CByte((byte) 0x20), //
    new CValue.CByte((byte) 0x77), // w
    new CValue.CByte((byte) 0x6f), // o
    new CValue.CByte((byte) 0x72), // r
    new CValue.CByte((byte) 0x6c), // l
    new CValue.CByte((byte) 0x64), // d
    new CValue.CByte((byte) 0x21), // !
    new CValue.CByte((byte) 0x21), // !
    new CValue.CByte((byte) 0x21), // !
    new CValue.CByte((byte) 0x21), // !
    new CValue.CByte((byte) 0x21)  // !
  });
for(int i = 0; i < result.length; i++) {
  System.out.println(result[i]);
}
```

The directory containing this document contains a second file, `JavaMD5Eval.java` which contains this concrete evaluation code. It can be compiled and simulated using the same process as for the original example.

```
# javac -g -j BC_JAR:JDK_JAR JavaMD5Eval
# jss -c . -j BC_JAR:JDK_JAR JavaMD5Eval
149
205
6
99
121
234
137
128
62
136
212
85
72
```



```
225
82
213
```

Checking Boolean Properties

The tutorial so far has focused on models describing the output of an algorithm in terms of symbolic inputs. Sometimes, however, it can be valuable to reason about the truth of some higher-level property about the program (potentially about the output it generates, or potentially about some intermediate value). In the context of Java, this can amount to reasoning about expressions of type `boolean`, and which can therefore be represented in a model using a single output bit.

For properties of this sort, it is possible to generate AIG models using the approach described earlier. Alternatively, models with a single output bit can also be translated into DIMACS CNF format, the standard input format for boolean satisfiability (SAT) solvers. As a simple example, consider the problem of showing that, in the Java semantics of integer math, multiplying a byte by two is the same as adding it to itself. The example file `ExampleCNF.java` contains the following lines:

```
byte x = Symbolic.freshByte((byte)0);
Symbolic.writeCnf("ExampleCNF.cnf", x + x == 2 * x);
```

The formulas emitted in CNF format for boolean expressions are negated, with the rationale that they tend to be used to prove the validity of expressions, rather than satisfiability. Therefore, a result of "unsatisfiable" from a SAT solver indicates that the formula passed to `writeCnf` is valid (always true).