The ERIGONE Model Checker

Quick Start Guide

Mordechai (Moti) Ben-Ari
Department of Science Teaching
Weizmann Institute of Science
Rehovot 76100 Israel
http://stwww.weizmann.ac.il/g-cs/benari/

December 16, 2012

1 Introduction

ERIGONE is a *model checker* that can simulate and verify concurrent and distributed programs. It is a partial reimplementation of the SPIN model checker and supports a large subset of PROMELA, the modeling language used by SPIN. The ERIGONE distribution contains a comprehensive User's Guide and software documentation; this document is intended to be a quick introduction to working with ERIGONE.

ERIGONE is run from the command line. EUI is a development environment for ERIGONE with a graphical user interface; it can be downloaded from the JSPIN project in Google Code `http://code.google.com/p/jspin/`. A separate Quick Start Guide for running ERIGONE using EUI is available at that site.

The ERIGONE software is copyrighted under the GNU General Public License. The copyright statement and the text of the license are included in the distribution.

Introductory textbooks on concurrency and model checking are:

- M. Ben-Ari. *Principles of Concurrent and Distributed Programming (Second Edition)*. Addison-Wesley, 2006.
- M. Ben-Ari. *Principles of the Spin Model Checker*. Springer, 2008.

2 Installation and execution

This section describes installation and execution of ERIGONE under the Windows operating system. For other systems, see the User's Guide.

Download the archive `erigone-N.zip` (where `N` is the latest version number) from the `Erigone` project at Google Code `http://code.google.com/p/erigone/` and open the archive to a clean directory. The executable file is `erigone.exe`, and there are subdirectories containing the documentation and example programs. To run ERIGONE, execute the command:

```
erigone [arguments] filename
```

where `filename` is the name of the file containing the PROMELA source code; the default extension is `pml`. The command-line arguments can be displayed by `erigone -h`.

3 Simulating a PROMELA program

This section show how to simulate two PROMELA programs for the mutual exclusion problem; the first has an error that causes mutual exclusion to be violated, whereas mutual exclusion always holds in the second program.

The "Second Attempt"

The first program is the "Second Attempt" to solve the mutual exclusion problem (Ben-Ari, 2006, Section 3.6) and is contained in the file `second.pml` in the `examples` directory:

```
bool wantp = false, wantq = false;
byte critical = 0;

active proctype p() {
    do
    :: !wantq;
       wantp = true;
       critical++;
       assert (critical == 1);
       critical--;
       wantp = false;
    od
}

active proctype q() { /* Symmetrical */ }
```

Run:

```
erigone examples\second
```

The output will be similar to:

```
simulation terminated=assert statement is false,
  line=22,statement={assert (critical == 1)},
```

because it is highly likely that a run will cause mutual exclusion to be violated.

There are a large number of options for tracing the execution of ERIGONE. For example:

```
erigone -dm examples\second
```

displays the states of the simulation:

```
initial state=0,p=1,q=1,wantp=0,wantq=0,critical=0,
next state=1,p=1,q=2,wantp=0,wantq=0,critical=0,
  ...
next state=16,p=3,q=3,wantp=1,wantq=1,critical=0,
next state=17,p=3,q=4,wantp=1,wantq=1,critical=1,
next state=18,p=4,q=4,wantp=1,wantq=1,critical=2,
simulation terminated=assert statement is false,
  line=22,statement={assert (critical == 1)},
```

Since the variable `critical` has the value 2 when the assert statement in line 22 of the program is evaluated in state 18, an error has occurred and the simulation terminates.

All the output of ERIGONE is in a uniform format that is easy to read and easy to process by other tools. Each line consists of a sequence of comma-terminated named associations of the form `"name=value,"`.

Mutual exclusion with a semaphore

The second program uses a semaphore to achieve mutual exclusion and is contained in the file `sem.pml` in the `examples` directory:

```
byte sem = 1;
byte critical = 0;

active proctype p() {
  do ::
    atomic {
      sem > 0;
      sem--
    }
    critical++;
    assert(critical == 1);
    critical--;
    sem++
  od
}

active proctype q() { /* The same */ }

active proctype r() { /* The same */ }
```

Run:

```
erigone examples\sem
```

to simulate the program. The simulation will run until the step limit is reached:

```
simulation terminated=too many steps,
  line=11,statement={assert(critical == 1)},
```

You can run with the `-dm` argument to display the (very large number of) states. Enter ctrl-c to stop the simulation prematurely.

These simulations were *random simulations*, where the (nondeterministic) selection of the next statement to execute is chosen randomly. See the User's Guide for information on how to run an *interactive simulation* where you control the section of statements. *Guided simulation* is explained in the next section.

4 Verifying a PROMELA program with assert statements

Let us verify these two programs. Run:

```
erigone -s examples\second
```

where the argument `-s` means to run a verification in *Safety mode*. The output will be:

```
verification terminated=assert statement is false,
  line=22,statement={assert (critical == 1)},
```

Now run:

```
erigone -g examples\second
```

to run a *guided simulation* using the trail which describes the *counterexample*, the sequence of states found during the verification that leads to a violation of mutual exclusion. Add the argument `-dm` to display the states in the counterexample.

For the semaphore program, running:

```
erigone -s examples\sem
```

gives:

```
verification terminated=successfully,
```

There are numerous options for tracing the verification; for example, the argument `-dh` will print out which states were generated during the search and stored in the hash table (or not stored because they were already there):

```
inserted=true,p=1,q=1,wantp=0,wantq=0,critical=0,
inserted=true,p=2,q=1,wantp=0,wantq=0,critical=0,
inserted=true,p=3,q=1,wantp=1,wantq=0,critical=0,
inserted=true,p=4,q=1,wantp=1,wantq=0,critical=1,
inserted=true,p=5,q=1,wantp=1,wantq=0,critical=1,
inserted=true,p=6,q=1,wantp=1,wantq=0,critical=0,
inserted=false,p=1,q=1,wantp=0,wantq=0,critical=0,
  ...
```

5 Verifying a safety property using LTL

ERIGONE can verify a correctness property written as a formula in *linear temporal logic (LTL)*. There are two methods of specifying an LTL property: internally within the program or externally in a separate file. It is easier to use the first method; see the User's Guide for information on using LTL properties in files.

The program in the file `second-ltl.pml` is the same as that in `second.pml` except that the assert statements have been removed and replace by an LTL formula for the correctness property:

```
ltl { [](critical<=1) }
```

Read this as: the value of `critical` is *always* (`[]`) less than or equal to 1.

Run:

```
  erigone -s -t examples\second-ltl
```

where the argument `-t` tells ERIGONE to use the LTL formula. The result is:

```
verification terminated=never claim terminated,
  line=9,statement={critical--},
```

The phrase *never claim terminated* is a technical term used by the model checker to report that an error has been found. The counterexample can be displayed by running a guided simulation.

Running a verification for the program with the semaphore gives the result that the verification was terminated successfully.

6 Verifying a liveness property using LTL

Consider the following program, which is a very simple example used to demonstrate the concept of fairness:

```
byte n = 0;
bool flag = false;

active proctype p() {
  do
  ::  flag -> break;
  ::  else -> n = 1 - n;
  od
}

active proctype q() {
  flag = true
}
```

If process `q` ever executes, it will set `flag` to `true` and process `p` will then terminate, but in an unfair computation (where `q` never executes) this may never happen. Let us now try to prove the property `<>flag` meaning that *eventually* `flag` becomes true. The PROMELA program can be found file `fair1.pml` and it contains the LTL formula:

```
ltl { <>flag }
```

Run:

```
erigone -a -t examples\fair1
```

where -a means perform a verification in *Acceptance mode*. In this mode, the model checker searches for an *acceptance cycle*, which is the technical term for an error in the verification of an LTL formula containing *eventually* (<>). The result is:

```
verification terminated=acceptance cycle,
  line=7,statement={else -> n = 1 - n},
```

Run a guided simulation with the -dm argument:

```
initial state=0,p=1,q=1,n=0,flag=0,
start of acceptance cycle=1,
next state=1,p=3,q=1,n=0,flag=0,
next state=2,p=1,q=1,n=1,flag=0,
next state=3,p=3,q=1,n=1,flag=0,
next state=4,p=1,q=1,n=0,flag=0,
next state=5,p=3,q=1,n=0,flag=0,
simulation terminated=end of acceptance cycle,
line=7,statement={else -> n = 1 - n},
```

You can see that state 5 is the same as state 1 so the computation can continue indefinitely by cyclically repeating these five states, while `flag` is false (= 0) in all the states. This happened because the path is constructed by taking transitions only from process `p`.

Perform a verification in *Acceptance mode with weak fairness* by using the argument `-f`.

```
erigone -f -t examples\fair1
```

In this mode, only weakly fair computations are candidates for acceptance cycles. In a weakly fair computation, continually enabled transitions must eventually be taken. Since the assignment statement `flag=true` is obviously continually enabled, it must be taken. When the verification is run, the result is: The result is:

```
verification terminated=successfully,
```