The EUI Development Environment for 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 14, 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. EUI is a development environment for ERIGONE that includes a graphical user interface and algorithms to format and display the result of a simulation and verification. EUI is an adaptation of the JSPIN development environment for SPIN.

There are comprehensive User's Guides and software documentation for both ERIGONE and EUI. This document is intended to be a quick introduction to working with ERIGONE using the EUI environment.

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.

Acknowledgment:
I would like to thank for H. Peter Gumm for his comments on this guide.

2 Installation and execution

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

EUI needs the Java JRE, version 1.5 at least.

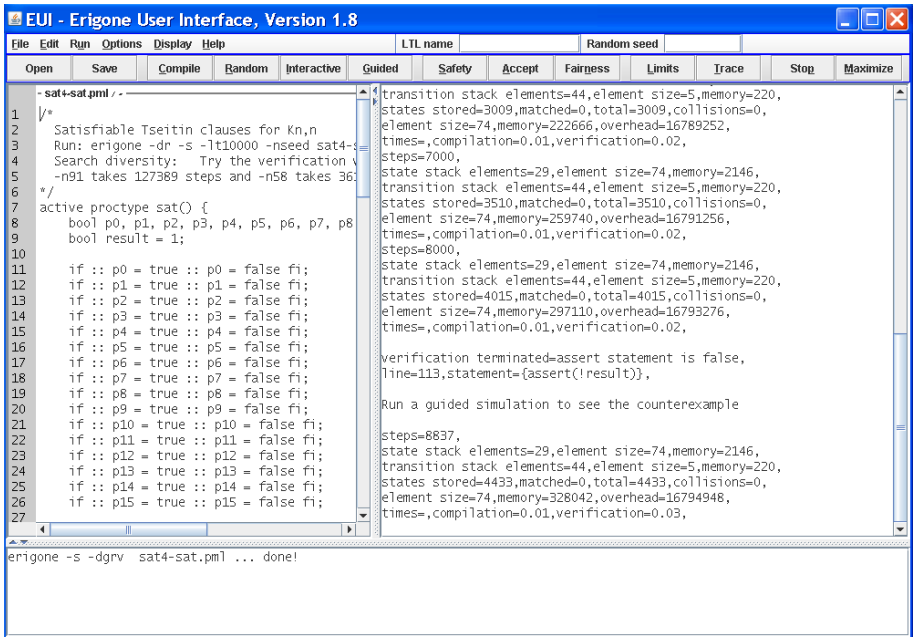
Download the EUI installation file `eui-N.exe` (where `N` is the latest version number) from the JSPIN project at Google Code `http://code.google.com/p/jspin/` and execute the installation file. The installation contains the executable file for ERIGONE so you don't need to install it separately.

By default, an icon to run EUI is installed on the Desktop. If Java is associated with `jar` files, double-clicking on the icon will run EUI. Click on Open to open a PROMELA source file in the edit window on the left.

The user interface of EUI is shown on the following page.

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 */ }
```

Open the file `second.pml` in the `examples` directory. The PROMELA source code will be displayed in the left-hand pane. Click on Random to run a simulation. The result will be something like:

```
Erigone v2.1.3, Copyright 2008-9 by Moti Ben-Ari, GNU GPL.
execution mode=simulation,
simulation mode=random,seed=-1,trail number=0,total steps=10,

Process Statement            critical      wantp      wantq
q       19 !wantp                   0          0          0
p        7 !wantq                   0          0          0
q       20 wantq = true             0          0          0
p        8 wantp = true             0          0          1
p        9 critical++               0          1          1
q       21 critical++               1          1          1
q       22 assert (critica          2          1          1

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

steps=7,
times=,compilation=0.01,simulation=0.01,
```

The states of the simulation are displayed in tabular form similar to that used in Ben-Ari (2006). The first column is the process from which a statement was executed; the next column contains the line number and source code of the statement; the rest of the columns give the values of the variables in the state. Following the table, the result of the simulation is given. Since `critical` has the value 2 when the assert statement in line 22 of the program is evaluated, an error has occurred and the simulation terminates.

The tabular display of the scenario is created by the EUI software from the output of ERIGONE. The other information is displayed in the raw format used by ERIGONE: 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 */ }
```

Open the file `sem.pml` and click Random. Since no error is encountered the states of the simulation will be displayed for a very long time (until a step count limit is reached). Click Stop 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

Open the file `second.pml` and click on Safety to perform a verification in *Safety mode*. The result will be displayed in the right-hand pane:

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

Run a guided simulation to see the counterexample
```

Click Guided 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.

Open `sem.pml` and click on Safety. The result is as expected:

```
verification terminated=successfully,
```

5 Verifying a safety property using LTL

ERIGONE can verify a correctness property written as a formula in *linear temporal logic (LTL)*. 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 an LTL formula has been added:

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

ltl { [](critical<=1) }

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

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

Read the LTL formula as: the value of `critical` is *always* (`[]`) less than or equal to 1. Click Safety to perform a verification in Safety mode. The output will be:

```
Erigone v2.1.0, Copyright 2008-9 by Moti Ben-Ari, GNU GPL.
execution mode=safety,error number=1,hash slots=22,state stack=2,
  location stack=3,progress steps=1,total steps=10,

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

Run a guided simulation to see the counterexample

steps=47,
  ...
times=,compilation=0.04,verification=0.06,
```

The first lines give information about the configuration used to run the verification. This is followed by the result of the verification. The phrase *never claim terminated* is a technical term used by the model checker to report that an error has been found. Finally, information on the performance of the model checker is displayed. 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 (file `fair1.pml`) which is a very simple example used to demonstrate the concept of fairness:

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

ltl { <>flag }

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.

Click on Accept to 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 a 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 to see the counterexample
```

Click on Guided to conduct a guided simulation that demonstrates the computation with the acceptance cycle that is a counterexample to the correctness property.

Click on Fairness to perform a verification in *Acceptance mode with weak fairness*. 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:

```
verification terminated=successfully,
```