The ERIGONE Model Checker

User's Guide

Version 3.2.5

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 partial reimplementation of the SPIN Model Checker. The goal is to facilitate learning concurrency and model checking.

- ERIGONE is single, self-contained, executable file so that installation and use are trivial.
- ERIGONE produces a detailed trace of the model checking algorithms. The contents of the trace are customizable and a uniform keyword-based format is used that can be directly read or used by other tools.
- Extensive modularization is used in the design of the ERIGONE program to facilitate understanding the source code. This will also enable researchers to easily modify and extend the program.

ERIGONE implements a large subset of PROMELA that is sufficient for demonstrating the basic concepts of model checking for the verification of concurrent programs. No language constructs are added so that programs for ERIGONE can be used with SPIN when more expressiveness and better performance are desired.

ERIGONE is written in ADA 2005 for reliability, maintainability and portability. No non-standard constructs are used.

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 archive.

Acknowledgements

I would like to thank Gerard J. Holzmann for his generous assistance throughout the development of this project. The original compiler was developed by Trishank Karthik Kuppusamy under the supervision of Edmond Schonberg.

2 References

- Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking*. MIT Press, 2008.
- M. Ben-Ari. *Ada for Software Engineers (Second Edition with Ada 2005)*. Springer, 2009.
- M. Ben-Ari. *Principles of the Spin Model Checker*. Springer, 2008.
  The sample programs from the book have been adapted for ERIGONE and can downloaded as `psmc-erigone.zip`.
- M. Ben-Ari. *Principles of Concurrent and Distributed Programming (Second Edition)*. Addison-Wesley, 2006.

- Gerard J. Holzmann. *The Spin Model Checker: Primer and Reference Manual*. Addison-Wesley, 2004. The abbreviation *SMC* is used to refer to this book.

- Fred Kröger and Stephan Merz. *Temporal Logic and State Systems*. Springer, 2008.

ERIGONE	`http://stwww.weizmann.ac.il/g-cs/benari/erigone/`
	`http://code.google.com/p/erigone/`
GNAT	`https://libre.adacore.com/`
SPIN	`http://spinroot.com/`

3 Installation, building and execution

- **Installation**
  Download the archive `erigone-N.zip` from Google Code. Open the archive to a clean directory. If you wish to examine or modify the source code, download the the archive `erigone-source-N.zip` and open it to the same directory. The following subdirectories are created:

  - `docs`: documentation;
  - `examples`: PROMELA source code of example programs;
  - `src`: the source code of ERIGONE;
  - `trace`: the source code of the TRACE program;
  - `list`: the source code of the LIST program;
  - `vmc`: the source code of the VMC program;
  - `vmc-examples`: examples for the VMC program.

  The file `readme.txt` in `examples` describes the commands and expected output for simulating and verifying the PROMELA programs in the directory.

- **Execution**

  ```
  erigone [arguments] filename
  ```

  In addition to the ERIGONE model checker, the archive contains additional programs described in separate sections:

  - COMPILER: Run the compiler by itself (Section 7);
  - LIST: Format the output of the compiler (Section 8);
  - TRACE: Format the output of the model checker (Section 9);
  - VMC: Generate graphs of the model checking algorithm (Section 10).

- **File names**
  If an extension is not given in `filename`, the default PROMELA source file extension is `pml`. Other file names are obtained by using the file root and predefined extensions. The output of a compilation is the *automata file* with extension `aut` (AUTomata) that contains the symbols and transitions resulting from the compilation. The extension of the trail file that contains a counterexample for an unsuccessful verification is `trl` (TRaiL). If the LTL correctness property `filename` is not given, the default is the root with extension `prp` (correctness PRoPerty).

- **Building**
  ERIGONE was built with the GNAT ADA 2005 compiler. The source code is divided into several subdirectories and the GNAT Project Manager is used to build the program. The file `erigone.gpr` describes the project configuration. Object files are placed in a subdirectory of `src` called `obj` that must exist before executing the project manager.

  To build, execute:

  ```
  gnatmake -Perigone [-O1] [-gnatn] [-g]
  ```

  The `-g` argument is necessary if you want to use the `gdb` debugger. For a production build, optimization level `-O1` significantly improves the performance, while `-gnatn` enables inlining of subprograms.

  The program COMPILER is built within the same directory using the project configuration file `compiler.gpr`. The program LIST, TRACE and VMC are built within their own directories using the appropriate `gpr` files.

4 Command-line arguments

Execution mode

`-r`	[R]andom simulation (default)
`-i`	[I]nteractive simulation
`-g`	[G]uided simulation
`-gN`	[G]uided simulation using the N'th trail
`-s`	Verification of [s]afety
`-a`	Verification of [a]cceptance
`-f`	Verification with [f]airness
`-c`	[C]ompilation only

Execution limits (with defaults)

`-lhN`	[H]ash slots	22
`-llN`	[L]ocation stack	3
`-lpN`	[P]rogress steps	1
`-lsN`	[S]tate stack	2
`-ltN`	[T]otal steps	10

The parameter `N` is in thousands (except for hash slots). For the stacks, the value is the number of **stack frames** and not the number of *bytes*. *Total steps* is the number of steps of a simulation or verification before the execution is terminated. For a verification, a progress report can be printed after every *progress steps* have been executed if `g` is included in the display arguments. The hash table is allocated 2^N slots, where $16 \leq N \leq 32$.

Additional execution arguments

`-h`	Display the [h]elp screen
`-mN`	Stop after the [m]'th error
`-nN`	Ra[n]dom seed
`-t[-][f]`	Read L[T]L formula

If `-m0` is used, *all* errors are reported and numbered trail files are written for each one. Random numbers are used for random simulation and for search diversity (Section 5.2). The default seed for random simulation is obtained from the clock.

The `-t` parameter is used to specify that an LTL formula is to be used in a verification. The formula may be embedded in the PROMELA source code or read from a file:

```
-t            use "ltl { ... }" if it exists
-t            otherwise, use default file "filename.prp"
-t-name       use "ltl name { ... }"
-tname.prp    use named file "name.prp"
```

Display arguments

The argument `-d` displays all the data. Selective display is possible with the argument `-dX`, where `X` is a string of one or more character taken from the following tables.

Compile and runtime messages:

`g`	Pro[g]ress messages
`p`	[P]rogram (transitions and symbols)
`r`	[R]un-time statistics
`v`	[V]ersion and copyright notice

Transitions:

`a`	[A]ll transitions from a state
`c`	[C]hosen transition (simulation)
`e`	[E]xecutable transitions from a state
`l`	[L]ocation stack
`t`	[T]rail
`y`	B[y]te code

States:

`h`	State in the [h]ash table
`m`	States in a si[m]ulation
`o`	Channel c[o]ntents
`s`	[S]tate stack

LTL translation:

`b`	[B]üchi automaton
`n`	[N]odes in the LTL tableau

Debug arguments

The argument `-u` displays debugging data that will be of interest when modifying the program. Selective display is possible with the argument `-uX`, where `X` is a string of one or more character taken from the following tables.

`c`	[C]ompiler tokens
`i`	[I]nterpreter stack
`r`	P[r]reprocessor

5 The subset of SPIN that is implemented

5.1 PROMELA subset

All constructs of the PROMELA language are implemented except for:

- `unsigned` and `typedef`.
- Macros except for `#define`.[1]
- Progress labels and never claims except those generated by LTL formulas.
- The specifiers `hidden`, `local`, `xr`, `xs` are ignored but don't cause compilation errors.
- `priority`, `unless`.

[1]One line only; continuation lines are not supported.

- C code insertion.

There are limitations on certain other constructs:

- Bitwise and shift operators are supported only for one-byte types.
- `d_step` is implemented like `atomic`.
- Only the first form of the `for` statement (iteration over bounds given by expressions) is implemented.
- You cannot initialize local variables declared after a statement. (This is likely to happen if an `inline` declares a local variable.)

Limits on the size of the PROMELA models are compiled into ERIGONE (Appendix A). If you need to verify larger models, you will need to rebuild the software.

5.2 Search diversity

By default, the state space of a model is searched in a fixed order. ERIGONE supports *search diversity* where randomness is introduced into the search by specifying a seed `-nN` for a verification.[2] For example, consider the Tseitin clauses associated with the graph $K_{4,4}$ given in the file `sat4-unsat` in the directory `examples`; this set of clauses is unsatisfiable and a safety verification will prove this in 1835006 steps. The file `sat4-sat` contains the same set of clauses with the parity of one literal changed so that the set is satisfiable. With the seed `-n91`, the counterexample (satisfying interpretation) is found in 127389 steps, while with the seed `-n58` only 361 steps are needed.

The Java program `GenerateSatERI` can be used to generate Tseitin clauses corresponding to $K_{n,n}$ together with random satisfiable instances and batch files with random seeds. However, ERIGONE is limited to sets of clauses for $n \leq 4$.

[2]For an introduction to search diversity and an explanation of the examples, see: M. Ben-Ari and F. Kaloti-Hallak, Demonstrating random and parallel algorithms with Spin, *ACM Inroads*, in press.

6 Display format

Data is displayed in a uniform format: lines of named associations, each of which is terminated with a comma. This format is verbose, but easy to read and easy to parse by postprocessors. [3]

6.1 Data structure display

The display begins with a title line (v) and a line with the parameters of the execution (r):

```
Erigone v3.1.0, Copyright 2008-11 by Moti Ben-Ari, GNU GPL.
execution mode=simulation,simulation mode=random,seed=-1,trail number=0,total steps=10,
```

The symbol table (p) includes for each variable its type, size, length (for arrays), flags (scope is global or local, parameter or not), the offset into its frame in the state vector and the byte code (y) for initialization:

```
variables=6,
name=n,type=byte_type,offset=0,length=1,size=1,scope=0,
  parameter=0,byte code={load_const 0 0,byte_store 0 0,},
name=finished,type=byte_type,offset=1,length=1,size=1,scope=0,
  parameter=0,byte code={load_const 0 0,byte_store 1 0,},
name=P.i,type=byte_type,offset=0,length=1,size=1,scope=1,
  parameter=0,byte code=,
name=P.temp,type=byte_type,offset=1,length=1,size=1,scope=1,
  parameter=0,byte code=,
name=Q.i,type=byte_type,offset=0,length=1,size=1,scope=1,
  parameter=0,byte code=,
name=Q.temp,type=byte_type,offset=1,length=1,size=1,scope=1,
  parameter=0,byte code=,
symbol table end=,
```

Tables of numeric and string constants (if any) are also displayed when p is given.
The frame table gives the offset of the frame for global and local variables:

```
frame table start=,
global=,offset=0,
pid=0,offset=2,
pid=1,offset=4,
pid=2,offset=6,
frame table end=,
```

For each process, a table of its transitions is display containing: the source and target state, various flags, the source code and the byte code:

[3]In the following description, the display option needed to obtain each item is given in parentheses and lines have been elided and reformatted to fit the page. Furthermore, the display of later versions might be somewhat different from what appears here.

```
transitions start=,
processes=3,
process=P,initial=1,transitions=7,
number=0,source=1,target=3,atomic=0,end=0,accept=0,line=7,
  statement={(i> 10)},byte code={byte_load 2 0,iconst 10 0,icmpgt 0 0,},
number=1,source=1,target=4,atomic=0,end=0,accept=0,line=8,
  statement={else},byte code={logic_else 0 0,},
number=2,source=3,target=9,atomic=0,end=0,accept=0,line=13,
  statement={finished++},byte code={iinc 1 1,},
number=3,source=4,target=5,atomic=0,end=0,accept=0,line=9,
  statement={temp=n},byte code={byte_load 0 0,byte_store 3 0,},
  ...
transitions end=,
```

6.2 LTL translation display

The display of the translation of the LTL formula to a Büchi automaton (BA) appears before that of the transitions of the processes, because the transitions of the BA are added to the transition table as a never claim. The display starts with the LTL formula together with the formula obtained by pushing negation inward (b). This is followed by the nodes of the tableau (n):

```
ltl formula=![]<>csp,
push negation=<>[]!csp,
nodes start=,
expanding=,node=1,incoming={0,},new={<>[]!csp,},old=,next=,with set=,
expanding=,node=2,incoming={0,},new={[]!csp,},old={<>[]!csp,},next=,with set=,
expanding=,node=4,incoming={0,},new={!csp,},old={[]!csp,<>[]!csp,},next={[]!csp,},with set=,
expanding=,node=4,incoming={0,},new=,old={!csp,[]!csp,<>[]!csp,},next={[]!csp,},with set=,
expanding=,node=5,incoming={4,},new={[]!csp,},old=,next=,with set={4,},
expanding=,node=6,incoming={4,},new={!csp,},old={[]!csp,},next={[]!csp,},with set={4,},
expanding=,node=6,incoming={4,},new=,old={!csp,[]!csp,},next={[]!csp,},with set={4,},
  ...
exists=,node=6,new incoming={4,6,},
  ...
nodes end=,
```

A BA is extracted from the tableau (b) and optimized:

```
optimized buchi automaton start=,
source=0,target=0,atomic=0,end=0,accept=0,line=0,statement={1},
  byte code={iconst 1 0,},
source=0,target=4,atomic=0,end=0,accept=0,line=0,statement={!csp},
  byte code={bit_load 3 0,logic_not 0 0,},
source=4,target=4,atomic=0,end=0,accept=1,line=0,statement={!csp},
  byte code={bit_load 3 0,logic_not 0 0,},
optimized buchi automaton end=,
```

The BA is used to create a never claim process in the transition table (p). In addition, the set of accept states is identified to make them easier to find.

6.3 Display of a simulation

A simulation begins from an initial state (`m`):

```
initial state=,P=1,Q=1,Finish=1,n=0,finished=0,P.i=1,P.temp=0,Q.i=1,Q.temp=0,
```

For each step of the simulation, the display can contain the set of all transitions from the state (`a`), the set of executable transitions (`e`), the chosen transition (`c`) and the next state in the simulation (`m`):

```
all transitions=5,
process=P,source=1,target=3,atomic=0,end=0,accept=0,line=7,
  statement={(i> 10)},byte code={byte_load 2 0,iconst 10 0,icmpgt 0 0,},
process=P,source=1,target=4,atomic=0,end=0,accept=0,line=8,
  statement={else}, byte code={logic_else 0 0,},
process=Q,source=1,target=3,atomic=0,end=0,accept=0,line=19,
  statement={(i> 10)},byte code={byte_load 4 0,iconst 10 0,icmpgt 0 0,},
process=Q,source=1,target=4,atomic=0,end=0,accept=0,line=20,
  statement={else},byte code={logic_else 0 0,},
process=Finish,source=1,target=2,atomic=0,end=0,accept=0,line=29,
  statement={finished== 2},byte code={byte_load 1 0,iconst 2 0,icmpeq 0 0,},

executable transitions=2,
process=P,source=1,target=4,atomic=0,end=0,accept=0,line=8,
  statement={else},byte code={logic_else 0 0,},
process=Q,source=1,target=4,atomic=0,end=0,accept=0,line=20,
  statement={else},byte code={logic_else 0 0,},

chosen transition=,
process=Q,source=1,target=4,atomic=0,end=0,accept=0,line=20,
  statement={else},byte code={logic_else 0 0,},

next state=,P=1,Q=4,Finish=1,n=0,finished=0,P.i=1,P.temp=0,Q.i=1,Q.temp=0,
```

To display the data in buffered channels use `-u`. The first field is the current number of messages in the channel; this followed by the messages themselves in brackets:

```
next state=,Pinit=3,ch1=1,a=0,b=0,c=0,channel1={2,[1,2,3,],[4,5,6,],},
```

The simulation can terminated by an assertion violation, or a valid or invalid end state, or by exceeding the total steps allowed:

```
simulation terminated=valid end state,
```

The runtime statistics are then displayed (`r`):

```
steps=87,
times=,compilation=0.33,simulation=0.18,
```

6.4 Display of a verification

During a verification, the sets of all transitions (a) and executable transitions (e) can be displayed as for a simulation. Operations on the stacks can be displayed, both for the state stack (s):

```
push state=10,p=6,q=6,wantp=1,wantq=1,critical=0,
  ...
top state=10,p=6,q=6,wantp=1,wantq=1,critical=0,
  ...
pop state=10,reason=no_more_transitions,
```

and for the location stack (l):

```
push transition=16,process=0,transition=5,never=0,visited=false,last=false,
  ...
top transition=16,process=0,transition=5,never=0,visited=true,last=false,
  ...
pop transition=16,
```

An attempt is made to insert each new state into the hash table (h); it may succeed or fail:

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

For verification with acceptance, the "inner" flag will be displayed indicating if the state is part of the outer or inner search; the "seed" state is also displayed:

```
seed=,p=4,q=13,:never:=4,wantp=1,wantq=1,turn=2,csp=0,inner=1,fair=3,
  ...
inserted=true,p=14,q=11,:never:=4,wantp=1,wantq=1,turn=2,csp=1,inner=1,fair=1,
```

For verification with fairness, the counter of the copies of the states is displayed
The outcome and the runtime statistics (r) are displayed at the end of a verification:

```
verification terminated=never claim terminated,

steps=47,
state stack elements=9,element size=22,memory=198,
transition stack elements=14,element size=5,memory=70,
states stored=19,matched=9,total=28,element size=22,memory=418,
times=,compilation=0.03,verification=0.09,
```

Progress messages can be displayed during a long verification (g).

7 The COMPILER program

COMPILER is a main program that runs the compiler as a separate program producing the automata file:

```
  compiler promela-filename [automata-filename]
```

8 The LIST program

The LIST program reads the automata file and formats it for display. Run it after running the compiler:

```
compiler filename
list filename
```

or

```
erigone -c filename
list filename
```

9 The TRACE program

The TRACE program performs string processing on the output data written by ERIGONE. It prints the output of a simulation in tabular form. First execute:

```
erigone -dcmop filename > filename.trc
```

to write a file with the symbol table, states and transitions taken. Then execute:

```
trace [arguments] filename
```

The arguments are:

`-tn`	Number of lines between column [t]itles
`-ln`	Column width for [l]ine numbers
`-pn`	Column width for [p]rocesses
`-sn`	Column width for [s]tatements
`-vn`	Column width for [v]ariables
`-xs`	E[x]clude variables with `s`
`-ms`	Exclude state[m]ents with `s`

Here is an example of the output of this program:

```
Trace v1.00, Copyright 2008-9 by Moti Ben-Ari, GNU GPL.
Proc Line Statement        wantp  wantq  critic
                           0      0      0
p    7    {!wantq}         0      0      0
q    19   {!wantp}         0      0      0
q    20   {wantq=true}     0      1      0
p    8    {wantp=true}     1      1      0
q    21   {critical++}     1      1      1
p    9    {critical++}     1      1      2
p    10   {assert(critical 1      1      2
simulation terminated=assert statement is false,
```

The `-x` and `-m` arguments are used for filtering the output. The strings `s` are lists of strings terminated by `#`:

```
-xwant# -massert#!#
```

The meaning is that a column for a *variable* (`-x`) or a row for a *statement* (`-m`) will not be printed if one of those strings appears within the variable or statement, respectively. With the above arguments, the columns for `wantp` and `wantq` will not be printed, nor will the rows with the `assert` statements or the negation operator.

10 The VMC program

VMC (*Visualization of Model Checking*) is a postprocessor of the trace output of ERIGONE that generates a sequence of graphs showing the incremental building and traversing of the state space during a simulation or verification. First, run ERIGONE using the display options shown in the examples below and redirect the output to a file with extension `trc`; for the Third Attempt, the commands for a simulation and a safety verification are:

```
erigone    -dehlmprs third > third.trc
erigone -s -dehlmprs third > third.trc
```

Now run VMC:[4]

```
vmc third
```

This will create a sequence of files in the DOT format of GRAPHVIZ. The file names will have a sequence number appended. Now, run the DOT program on each of these files. For Windows, the following batch command will do this for each file, where the file name is substituted for the parameter `%1` and we assume that DOT is in the path:

```
for %%F in (%1-*.dot) do dot -Tpng %%F > %%~nF.png
```

[4]VMC knows whether a simulation or verification was run from the information in the `trc` file.

You can now display the graphs using software such as `IrfanView`.

A Windows batch file for the entire procedure is included in the archive.

The "prologue" of the DOT file (specifying the size of the nodes, etc.) can be different for each program and is contained in the file `filename.prg`. If this file does not exist, it is created with default values.

Reading the graph

Each node is rectangular and is labeled with:

- For each process, the line number and statement at the location counter in this state. For a simulation, transitions that are executable are decorated with the at-sign `@`.
- For each variable, its value. The order of the variables is the same as that in which they are declared. (The order can be checked in the `aut` file if there is any doubt.)
- For a verification, an *end state* is decorated with a sharp `#` and an accept state is decorated with an asterisk `*`.
- For a fairness verification, the count of the copy of the state is displayed within square brackets.

Visual effects are applied to states as follows:

- The node for the current state has an elliptical border.
- A node for a state that is in error is colored red.
- For a verification:
  - Nodes for states on the stack have a bold border.
  - Nodes for matched states in the hash table have a double border.
  - Nodes for states in an inner search are filled in.
- For a simulation:
  - The number of borders of a node indicates the number of times that the node has been visited.

The archive `vmc-examples.zip` contains PROMELA for three programs, together with the DOT and PNG files that result from running VMC and DOT. The program `second` causes an assert statement to evaluate to false, `third` has an invalid end state, and `fair` contains an accept cycle if run with `-a` rather than `-f`.

A Size limits

Here is a list of the limits compiled into ERIGONE and the package specifications where they appear. The declaration of the size of the compressed state vector is in a separate package `Config_State` to enable a future implementation of per-model verifiers with minimal recompilation. To restrict the coupling between the compiler and the model checker, the limits marked `*` are declared independently in `Compiler_Global`.

Declarations in package `Global`		
Identifier	Type	Meaning
`Byte`	`mod 256`	Values and indices
`Name*`	`String(1..32)`	Identifiers
`Line*`	`String(1..128)`	Source statements, strings

Declarations in package `Config`		
Identifier	Type	Meaning
`Process_Index`	`Byte range 0..7`	Processes
`Symbol_Index`	`Byte range 0..31`	Symbols
`Message_Index`	`Byte range 0..3`	Elements in a channel message
`Channel_Index`	`Byte range 1..8`	Channels
`Data_Index`	`Byte range 0..63`	Bytes of data in a state
`Transition_Index*`	`Byte range 0..254`	Transitions in a process
`Location_Index`	`Byte range 0..15`	Transitions from a single state
`Byte_Code_Index*`	`Byte range 0..255`	Byte codes per statement
`Interpret_Index`	`Byte range 0..63`	Interpretation stack
`Node_Index`	`Integer range 0..511`	Nodes in the state space diagram
`Max_Futures`	`constant := 4`	Number of future formulas

Declarations in package `Config_State`		
Identifier	Type	Meaning
`Process_Size_Index`	`Byte range 0..7`	Size of processes in a state vector
`Variable_Size_Index`	`Byte range 0..31`	Size of variables in a state vector

Declarations in package `Compiler_Global`		
Identifier	Type	Meaning
`Table_Index`	`Byte range 0..63`	Size of the symbol table
`Token_Index`	`Byte range 0..4095`	Number of tokens in the source