Tool Presentation: Teaching Concurrency and Model Checking

Mordechai (Moti) Ben-Ari

Department of Science Teaching
Weizmann Institute of Science
Rehovot 76100 Israel
moti.ben-ari@weizmann.ac.il,
http://stwww.weizmann.ac.il/g-cs/benari/

Abstract. This paper describes a set of software tools developed for teaching concurrency and model checking. jSpin is an elementary development environment for Spin that formats and filters the output of a simulation according to the user's specification. SpinSpider uses debugging output from Spin to generate a diagram of the state space of a Promela model; the diagram can be incrementally displayed using iDot. VN supports teaching nondeterministic finite automata. The Erigone model checker is a partial reimplementation of Spin designed to be easy to use, well structured and well documented. It produces a full trace of the execution of the model checker in a format that is both readable and amenable to postprocessing.

1 Introduction

Concurrency is a notoriously difficult subject to learn because the indeterminate behavior of programs poses challenges for students used to debugging programs by trial and error. They must learn new concepts of specification and correctness, as well as formal methods such as state transition diagrams and temporal logic. Several years ago, I became convinced that the Spin model checker is appropriate for teaching concurrency. This led me to write a new edition of my concurrency textbook [1], followed by an introductory textbook on Spin [2].

During this period, I developed software tools for teaching with Spin: jSpin, a development environment; SpinSpider, which automatically generates the state space diagram of a Promela program that can be viewed incrementally with iDot; VN, a tool to support learning nondeterminism of finite automata (NDFA). Following a short description of these tools, the paper describes the Erigone model checker, a partial reimplementation of Spin designed to simplify the modeling checking of Promela programs used for teaching concurrency and to facilitate the teaching of model checking itself.[1]

[1] Screenshots of the tools can be found on the website given in Section 8.

2 jSpin: A development environment for Spin

I believe that concurrency can and should be taught much earlier than it is traditionally, even in high schools [3]. In spite of its origins in academic research and industrial applications, SPIN is an appropriate vehicle for teaching. PROMELA's syntax and low-level semantics are close to C and thus familiar to students who typically learn Java, C++ or C# as their first language. The downside of using SPIN for teaching is that it requires a C compiler that can be difficult for beginning students to install, and it is a command-line tool with a daunting list of arguments, while XSPIN interface is oriented at professional users.

JSPIN was developed as a development with a simple GUI that allows the student to edit PROMELA programs and to execute SPIN in its various modes with a single mouse click or keypress. JSPIN formats the output of a simulation (random, interactive or guided) in a tabular representation of a scenario that shows the values of the variables after executing each step. You can specify variables and statements to include or exclude, so that less important information like incrementing indices does not clutter the display.

An example of the output of JSPIN is shown below. It is for a program that repeatedly increments a global variable concurrently in two processes by loading its value into a local variable and then storing the new value. Surprisingly, there is a scenario in which the final value of the global variable `n` can be two. The final part of a trail is shown below: the barrier variable `finished` and the loop indices `i` have been excluded, as have the statements that increment the indices.

```
Process Statement          P(0):temp  Q(1):temp  n
0 P     8   else           8          1          9
0 P     9   temp = n       8          1          9
0 P     10  n = (temp+1)   9          1          9
1 Q     22  n = (temp+1)   9          1          10
1 Q     25  finished = (fi 9          1          2
0 P     13  finished = (fi 9          1          2
2 Finis 29  finished==2    9          1          2
spin: text of failed assertion: assert((n>2))
```

JSPIN is implemented in JAVA, as are SPINSPIDER, IDOT and VN.

3 SpinSpider: Visualizing the State Space

SPINSPIDER uses data available from SPIN to generate the complete state space of a PROMELA program, which is written out in the *dot* language and laid out by the DOT tool of GRAPHVIZ [4]. SPINSPIDER is integrated into the JSPIN environment, although it can be run as a standalone application. IDOT displays the graphics files generated by SPINSPIDER interactively and incrementally.

4 VN: Visualization of Nondeterminism

Nondeterminism is a related, but important, concept that is difficult for students to understand. In particular, it is difficult to understand the definition of *acceptance* by an NDFA. VN visually demonstrates nondeterminism by leveraging simulation and verification in SPIN together with the graph layout capabilities of DOT. Its input is an XML representation of an NDFA generated interactively by JFLAP [5]. For any NDFA and input string, a PROMELA program is generated with embedded `printf` statements that create a file describing the path in the NDFA taken by an execution. VN runs the program in random simulation mode to show that arbitrary scenarios will be generated for each execution. In interactive simulation mode, the user resolves the nondeterminism like an oracle. Verification is used to show the existence of an accepting computation.

5 The Erigone Model Checker

My experience—both teaching concurrency and developing the tools described above—led me to develop the ERIGONE model checker, a simplified re-implementation of SPIN. The rationale and design principles were as follows.

Installation and execution The installation of a C compiler is a simple task for an experienced person, but a potential source of error for inexperienced students. Therefore, ERIGONE is a single executable file. This implies that the size of the state vector is determined at compile-time, but this is not a problem for the small programs taught in a concurrency course. The constants defining the size of the state vector are declared with limited scope so that a future version could generate a program-specific verifier with minimize recompilation.

Tracing The implementation of the tools described was difficult because there is no uniform and well documented output from SPIN. ERIGONE uses a single format—named association—that is both easy to read (albeit verbose) and easy to postprocess. Here are a few lines of the output of a step of a simulation of the "Second Attempt" to solve the critical section problem [1, Section 3.6]:

```
next state=,p=3,q=8,wantp=1,wantq=0,critical=0,
all transitions=2,
process=p,source=3,target=4,...,statement={critical++},...,
process=q,source=8,target=2,...,statement={!wantp},...,
executable transitions=1,
process=p,source=3,target=4,...,statement={critical++},...,
chosen transition=,
process=p,source=3,target=4,...,statement={critical++},...,
next state=,p=4,q=8,wantp=1,wantq=0,critical=1,
```

Four data options are displayed here: *all the transitions* from a state, the *executable transitions* in that state, the *chosen transition* and the *states of the simulation*. Fifteen arguments independently specify which data will be output: (a) the symbol table and transitions that result from the compilation; (b) for an

LTL to BA translation, the nodes of the tableau and the transitions of the BA; (c) for a simulation, the options shown above; (d) for a verification, the sets of all and executable transitions, and the operations on the stacks and on the hash table; (e) runtime statistics and progress.

Postprocessing A postprocessor TRACE was written that implements a filtering algorithm like the one in JSPIN that formats a scenario as a table, including and excluding variables and statements as specified by the user. Because of the uniform output of ERIGONE, it could be implemented in a few hours.

Well structured and well documented Learning model checking is difficult because there is no intermediate description between the high-level pseudocode in research papers and books, and the low-level C code of SPIN. This also has implications for research into model checking, because graduate students who would like to modify SPIN's algorithms have to learn the C code. During the development of ERIGONE, continuous effort was invested in refactoring and documentation to ensure the readability and maintainability of the software. The program is constructed from 45 modules and the documentation of the software structure runs to 11 pages (not including that for the compiler).

6 The current status of Erigone

Currently, ERIGONE implements enough of PROMELA to work with the basic concepts and algorithms of concurrency found in textbooks like [1], in particular, the safety and liveness of Dijkstra's "four attempts" leading to Dekker's algorithm. Weak fairness is implemented and weak semaphores can be defined using `atomic`. Correctness specifications can be given using `assert` statements and as LTL formulas.

A fuller implementation awaits progress on the compiler, which is being implemented by a student. In particular, I plan to support channels for teaching distributed algorithms, and arrays for implementing concurrent algorithms like the dining philosphers, solving nondeterministic algorithms [6], [2, Chapter 11], and for simulating NDFAs in PROMELA.

7 The implementation of Erigone

ERIGONE is a single program consisting of several subsystems: (1) a top-down compiler that translates PROMELA into transitions with a JVM-like byte code for the statements and expressions; (2) a model checker that implements the algorithms as described in [7], except that an explicit stack for pending transitions from the states is used instead of recursion; (3) a translator of LTL to BA using the algorithm in [8].

The compiler and the LTL-to-BA translator can be run independently of the model checker, and the postprocessor TRACE is a separate program.

ERIGONE is implemented in ADA 2005. This language was chosen because of its superb facilities for structuring programs and its support for reliable software. ADA also has excellent concurrency constructs that can be used in future

investigations into parallel model checking. The container library introduced in Ada 2005 is used in the compiler and the LTL to BA translator; in the model checker, arrays are preferred for efficiency, except that a container is used for the hash table.

8 Availability of the tools

All these tools are freely available under the GNU General Public License and can be downloaded from links on the following page:

`http://stwww.weizmann.ac.il/g-cs/benari/home/software.html`

as well as from Google Code, which also contains Subversion repositories.

The GNAT compiler from AdaCore was used; it is freely available under the GNU GPL for Windows and Linux, the primary platforms used by students.

jSpin is used at several universities for teaching concurrency with Spin. Erigone is a new tool and I am looking for educators who would be willing to use it in their teaching and to help develop it further.

9 Acknowledgements

Mikko Vinni, a student at the University of Joensuu, developed iDot, and Trishank Karthik Kuppusamy, a student at New York University, wrote the Promela compiler under the supervision of Edmond Schonberg. Michal Armoni helped design VN.

I am deeply indebted to Gerard Holzmann for his unflagging assistance throughout the development of these tools. I would also like to thank the many victims of my emails asking for help with the model-checking algorithms.

References

1. Ben-Ari, M.: Principles of Concurrent and Distributed Programming (Second Edition). Addison-Wesley, Harlow, UK (2006)
2. Ben-Ari, M.: Principles of the Spin Model Checker. Springer, London (2008)
3. Ben-David Kolikant, Y.: Understanding Concurrency: The Process and the Product. PhD thesis, Weizmann Institute of Science (2003)
4. Gansner, E.R., North, S.C.: An open graph visualization system and its applications to software engineering. Software Practice & Experience **30**(11) (2000) 1203–1233
5. Rodger, S.H., Finley, T.W.: JFLAP: An Interactive Formal Languages and Automata Package. Jones & Bartlett, Sudbury, MA (2006)
6. Floyd, R.W.: Nondeterministic algorithms. J. ACM **14**(4) (1967) 636–644
7. Holzmann, G.J.: The Spin Model Checker: Primer and Reference Manual. Addison-Wesley, Boston MA (2004)
8. Gerth, R., Peled, D., Vardi, M.Y., Wolper, P.: Simple on-the-fly automatic verification of linear temporal logic. In: Proceedings of the Fifteenth IFIP WG6.1 International Symposium on Protocol Specification, Testing and Verification XV. (1996) 3–18

Plan for Oral Presentation

All of the tools described in the paper will be demonstrated:

- jSpin. The Promela program that increments a global variable:

```
byte n = 0, finished = 0;
active proctype P() {
  byte  i = 1, temp;
  do :: (i > 10) -> break
     :: else     -> temp = n;
                    n = temp + 1;
                    i++
  od;
  finished++;
}
active proctype Q() { ... }
active proctype Finish() {
  finished == 2;
  assert (n > 2);
}
```

  will be used to show how to run it in all simulation modes and in verification mode. The output filtering options will be shown on the trail of a counterexample.
- SpinSpider. The automatic generation of the state space will be shown for two examples: the Third Attempt, where the state space includes a deadlock state, and the simple example for fairness:

```
byte n = 0;
bool flag = false;
active proctype p() {
  do
  ::  flag -> break;
  ::  else -> n = 1 - n;
  od
}
active proctype q() {
  flag = true
}
```

  where both fair and unfair paths can be generated and displayed.
- VN. An NDFA from the JFLAP examples will be used for demonstrating the various concepts of acceptance by an NDFA.
- Erigone. I will show in more detail the output of the model checker, especially when used for verification. If time permits, I will describe the software structure of the program.