A Primer on Model Checking

Mordechai (Moti) Ben-Ari

Model checking is a widely used formal method for the verification of concurrent programs. This article starts with an introduction to the concepts of model checking, followed by a description of Spin, one of the foremost model checkers. Software tools for teaching concurrency and nondeterminism using model checking are described: Erigone, a model checker for teaching; jSpin, a development environment; VN, a visualization of nondeterminism.

1 INTRODUCTION

The June 2009 issue of *inroads* contained a special section on Formal Methods in Education and Training. Several of the papers described the central place of *model checking* for verification of programs, primarily concurrent and distributed programs [14,17]. Model checking verifies a program by using software to analyze its state space, as opposed to the mathematical deductive methods proposed by the pioneers of verification like C.A.R. Hoare. Edmund Clarke, Allen Emerson and Joseph Sifakis received the 2008 Turing Award for their invention of model checking. Software systems for model checking have become a cornerstone of both hardware and software development, especially in high integrity systems.

I believe that model checking is appropriate as a vehicle to teach formal methods to introductory students: real model checkers are easy to use and, initially, little mathematical maturity is required. This article starts with an introduction to the concepts of model checking, followed by a description of Spin, one of the foremost model checkers. Then, I present software tools that I have developed for teaching concurrency and nondeterminism using model checking.

2 A BIT OF LOGIC

Suppose that you want to know if `~p->(p->q)` is a valid formula of the propositional calculus. You could do what every mathematician does for a living: *prove* it using a deductive system consisting of axioms assumed to be valid and rules of inference that preserve validity. The same can be done for computer programs: Deductive systems pioneered by C.A.R. Hoare [8] can be used to prove that a program is correct, meaning that the execution of the program gives an output that satisfies a correctness property expressed as a formula in logic. Deductive systems were extended to concurrent computation by Amir Pnueli and Zohar Manna [12,13]. To express time-dependent correctness properties like "the program never deadlocks" or "an interrupt is eventually handled," their deductive system uses *temporal logic*, which is able to elegantly express concepts like "never" and "eventually." Amir Pnueli received the 1996 Turing Award for his work on *temporal logic*. Sadly, he passed away in November 2009.

Constructing a proof requires mathematical insight and tenacity, but not everyone is blessed with these talents, nor with the time needed to construct such proofs. Is there another way? Con-

sider again the formula `A = ~p->(p->q)` and the claim that it is *valid*, meaning: the formula is true whatever values are given to the atomic propositions p and q. But any atomic proposition is either true or false, so there are only four *interpretations* to check: `p=true` and `q=true`, `p=true` and `q=false`, `p=false` and `q=true`, `p=false` and `q=false`. In fact, `A` is true in each of these interpretations; therefore, the formula `A` is valid.

Mathematicians tend to reject proofs by exhaustive checking of all cases as being less satisfying than deductive proofs, and with good reason. First, they are not applicable for proving theorems about integers and real numbers, which are infinite domains so that the number of interpretations is infinite and they cannot be exhaustively checked. Second, they offer no insight into why a theorem is true. But we computer scientists have more practical concerns. If we can check all computations of a program and show that they all satisfy a correctness property, we will be willing to forego elegance and be more than satisfied that our program has been proved correct.

3 MODEL CHECKING

The problem with concurrent programs is that the number of possible computations is astronomical, so it seems that exhaustive checking is impractical as a method of gaining confidence in the correctness of the program. In the 1980s, Clarke, Emerson and Sifakis showed that it can be feasible to check all possible computations of a concurrent program. Their key insight was to note that both a concurrent program and its correctness property can be transformed into *nondeterministic finite automata (NDFA)* and "run" simultaneously. Given the NDFA corresponding to the program and the NDFA corresponding to the *negation* of the correctness property (expressed in temporal logic), a *model checker* searches for an "input string" accepted by *both* automata. If it finds one, the input represents a computation of the program that falsifies the correctness claim; therefore, the program is not correct and the computation can be reported as a counterexample to the correctness claim.

Let me demonstrate these concepts with a trivial concurrent program that models changing the value of a memory cell, where the arithmetic is done in a register. The two processes could be an applications program together with an interrupt routine. When an interrupt occurs, the contents of the registers are saved and then restored when the interrupt routine is exited; the effect is that the application program and the interrupt routine have separate sets of registers.

```
integer n = 0;

process P
     integer regP = 0;
p1:  load n into regP
p2:  increment regP
p3:  store regP into n
p4: end

process Q
     integer regQ = 0;
q1:  load n into regQ
q2:  increment regQ
q3:  store regQ into n
q4:  end
```

A *state* of this program has five components: the *instruction pointers (IP)* of the two processes and the values of the three variables (the two local registers and the global variable `n`). One possible state is (IP(`P`)=p3, IP(`Q`)=q1, `regP`=1, `regQ`=0, `n`=0); since all states have the same components, it will be convenient to abbreviate them using positional instead of named notation: (p3, q1, 1, 0, 0). The initial state is (p1, q1, 0, 0, 0) and there are two transitions from this state, one obtained by executing the instruction at p1 and one obtained by executing the instruction at q1. The number of possible states is finite: there are four values for each of the two IPs, and at most three values (0,1,2) for each of the three variables, in total, 4x4x3x3x3=432 states. Furthermore, there are at most two outgoing transitions from each state. It is tedious, but easy, to construct the NDFA that corresponds to the possible computations. One possible computation is:

(p1,q1,0,0,0) → (p2,q1,0,0,0) →
(p3,q1,1,0,0) → (p4,q1,1,0,1) →
(p4,q2,1,1,1) → (p4,q3,1,2,1) →
(p4,q4,1,2,2).

The last state is a final state of the automaton, since there are no transitions from either p4 or q4.

Let us now try to prove the correctness claim that "when the program terminates, the value of n is 2" or, more formally:

```
terminates -> (n = 2).
```

Its negation is

```
terminates && !(n = 2),
```

which can also be expressed as

```
terminates && (n != 2).
```

This can be translated into a trivial automaton with one state containing this formula and one transition that loops back to the state. By checking the entire state space, it is easy to see that there is a computation that is simultaneously accepted by the NDFA for the program (that is, the program terminates) and by the NDFA for the negation of the correctness claim:

(p1,q1,0,0,0) → (p2,q1,0,0,0) →
(p2,q2,1,0,0) → (p3,q2,1,0,0) →
(p3,q3,1,1,0) → (p4,q3,1,1,1) →
(p4,q4,1,1,1).

Not only have we proved that the correctness claim can be falsified, but we have also found a *counterexample*, that is, a specific computation where the claim is false. We can use this counterexample to find the bug, which can be either in the program or in the correctness claim.

If a correctness claim is true, checking the state space will *not* find a computation that is both a legal execution of the program and also satisfies the negation of the correctness claim. Note the double negation: a program is correct if there is *no* computation *negating* correctness. This can be somewhat confusing at first, but you quickly get used to it.

A Primer on Model Checking

continued

4 MODEL CHECKING ALGORITHMS

The theoretical insight of Clarke, Emerson and Sifakis was followed up by research on algorithms and implementation techniques for dealing with the astronomical number of states that can be part of a computation. Fortunately, real computations cannot have an infinite number of (distinct) states, for the simple reason that the number of bits in a computer's memory is finite. A variable of type `int` can have at most 2^{32} values unlike a mathematical integer which can have an infinite number of values. Still, in a program that uses 1 megabyte of memory where each byte has 2^8 possible values, memory alone accounts for $2^8 \times 2^{20}$ different possible states of the computation, and this has to be multiplied by the number of transitions in the NDFA for each process and by the number of transitions in the NDFA for the correctness property.

We are saved by the fact that not all states are *reachable*. Consider, again, the above example, where we calculated that there are 432 possible states. But, all three variables *must* have the value zero whenever process P is at p1 and process Q is at q1. In other words, the only reachable state of the form (p1,q1,...,...,...) is (p1,q1,0,0,0); the other 26 states of the form (p1,q1,...,...,...) are not reachable and need not be consider when constructing the state space.

We can reduce the number of reachable states that need be explored and the resources required to check them by generating them *on-the-fly*. The reachable states can be constructed by starting with the initial state and generating the states that are reached by following each of the outgoing transitions. Synchronously, new states of the NDFA representing the correctness claim are generated. Whenever these new states are generated, the correctness property is checked: if the property is false, the model checker reports an error; if not, generation of new states continues.

The state space is structured as a directed graph, so constructing and exploring the state space is done by a breadth-first or depth-first search of the directed graph. (The state space for the example is shown in Figure 1.) In practice, depth-first search is preferred because requires relatively little memory: a stack that records the states visited and the number of the last transition taken in each state. The use of a stack means that reporting a counterexample is trivial: it is simply the list of states and transitions that appear on the stack. Fortunately (well, actually, unfortunately), we tend to write more programs with bugs than we do correct programs, in which case the entire space of reachable states need not be generated. We need only generate states until an error is encountered and this often happens sooner rather than later.

Consider the example again. Suppose that we wish to verify the (correct) property:

```
terminated -> (n <= 2)
```

and suppose that our depth-first search has checked all computations starting with:

(p1,q1,0,0,0) → (p2,q1,0,0,0) → (p3,q1,1,0,0).

Since no error is found, the search will backtrack, generate the state (p2,q2,0,0,0) and continue on; again, it will not find an error. Eventually, the search will start to explore computations that start with a transition of process Q, leading to:

(p1,q1,0,0,0) → (p1,q2,0,0,0) → (p2,q2,0,0,0).

But wait. This state has already been generated and found not to lead to an error, so there is no reason to explore it again. In a large program, there may be a large subgraph of reachable states starting from any given state, so model checking will be much more efficient if we don't check the same subgraph more than once.

To prevent duplicated effort, a model checker must maintain a data structure with the set of all states that have been encountered; whenever a new state is generated, this data structure is checked and if the state appears in it the search continues with the next reachable state. The data structure commonly used is a hash table, since we only insert elements and check if they are already there; removing elements is never done. This hash table for storing states determines the memory requirements and (along with CPU time) limits the size of models that

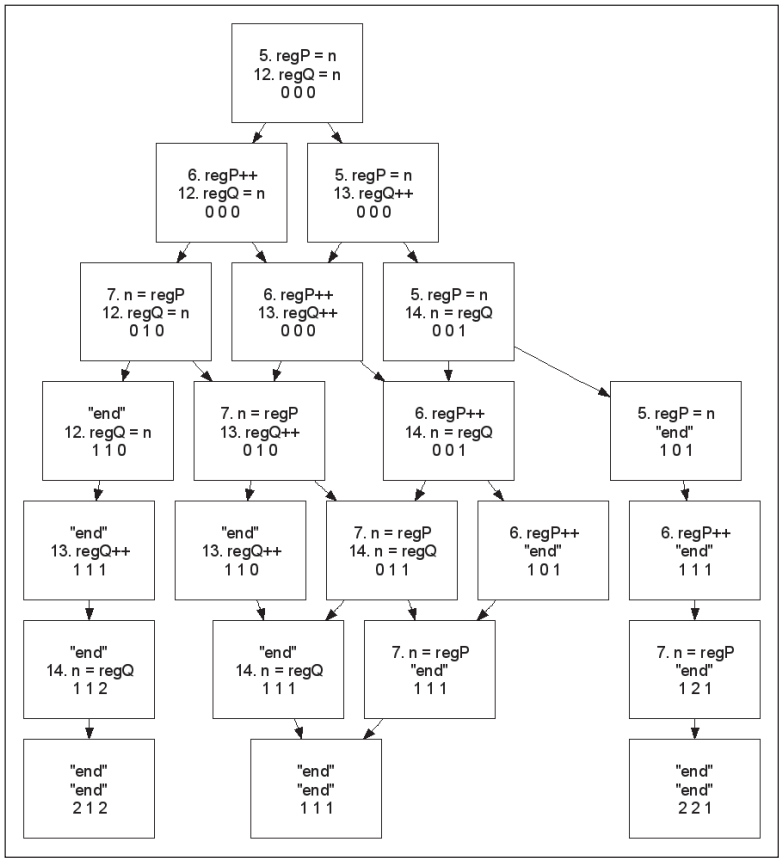

Figure 1: The state space of the example

can be checked. Much effort has been devoted to improving the hash algorithms and compression algorithms for storing the states. Fortunately, large memories are relatively cheap and this has made it feasible to increase the size of models that can be checked.

5 MODELING SYSTEMS

Model checking is primarily used in the verification of properties of concurrent and distributed systems where the actual *content* of the data is immaterial. We are interested in verifying that a message that is sent over a network is received, but the actual content of the message is not important. Similarly, we may need to show that a real-time system doesn't deadlock, or that a resource is used atomically, or that a server responds to a client, but the actual computation that potentially deadlocks, or the content of the resource, or the nature of the service do not affect the correctness of the synchronization mechanism or the communications protocol. Therefore, the first step in model checking is to write a *model* of the system to be verified.

A model of a system is a description of the system at a level that is usually higher than what can be efficiently implemented. Abstract models are important because they enable the software engineer to write specifications that are understandable and that do not prematurely constrain an implementation. Models written in a formal specification language have the additional advantage that their correctness properties can be verified. While it is true that implementing an abstract model can lead to errors, *implementation* errors are usually easier to find and correct than errors in a specification.

Suppose that we wish to model a temperature controller in a power station. The possible values returned by the thermometer may range over several hundred degrees, but—as we all know from bitter experience—bugs invariably occur at the limits of the value of a variable. Rather than model the entire range of the thermometer, it will generally be sufficient to model a handful of representative values: the lowest value, the highest value and one or two more within the range. The challenge for a software engineer is to model a system in sufficient detail so that it is faithful to reality, but at a sufficiently high level of abstraction so that it is easy to understand and tractable to verify.

6 SPIN

Before we turn to a description of Spin, one of the foremost model checkers, let us note some other important model checkers:

- SMV/NuSMV [7] uses a different temporal logic for specifying correctness properties and a different way of storing and searching the state space. This approach is widely used for verifying synchronous systems such as hardware, where many components change their state simultaneously upon receiving a clock pulse.
- Java PathFinder [18] is an attempt to verify programs written in an actual programming language, as opposed to a simple modeling language as in Spin. The verification is, therefore, more realistic, but the size of the models can be verified is limited because a Java program is much more detailed with many more states.
- Concurrent programs abstract away from absolute time and are based upon the interleaving of atomic instructions from the processes. The UPPAAL model checker [11] models time so that it can be used to check the correctness of real-time programs.

Spin (http://spinroot.com) is a model checker that has been developed over many years by Gerard J. Holzmann, currently at NASA/JPL [9]. In 2001, Holzmann received the ACM Software Systems Award for Spin. Originally designed for verifying communications protocols, it has since become one of the most widely used verification tools. Spin is particularly suited for modeling concurrent and distributed systems that are based upon interleaving of atomic instructions.

The Spin model checker is one of the leading verification tools used by professional software engineers, but to my surprise I found that it is eminently suitable as a teaching tool.

After I learned about Spin and model checking several years ago, I rewrote my concurrency textbook to include material on these topics [2], and later wrote an introductory textbook on Spin itself [3]. What attracted me to Spin was a unique intersection of two seemingly conflicting universes: professional software and pedagogical software. We are all familiar with the conflicting pressures: should students be taught using pedagogical software that facilitates learning, or should they be trained to use the most up-to-date professional tools? Although we can often demonstrate improved learning outcomes using the former, students and employers frequently expect us to teach the latter. The Spin model checker is one of the leading verification tools used by professional software engineers, but to my surprise I found that it is eminently suitable as a teaching tool.

The suitability of Spin for teaching results from the same dilemma that is at the base of model checking: constrained and efficient or expressive and inefficient. Holzmann chose the former: models in Spin are written in *Promela*, a language with a very limited set of features. Expressions and assignment statements in Promela have a syntax and semantics that are almost the same as those in C and Java. Dijkstra's guarded commands are used for the control structures to facilitate writing nondeterministic programs. While

guarded commands may be initially unfamiliar, they are not difficult to understand. The data types are integers and bytes, together with one-dimensional arrays. The rest of the features in Spin are those needed to build models of concurrent systems: processes, a construct for specifying that a sequence of statements is atomic, and channels that are generalizations of CSP channels. Correctness properties are expressed using either local `assert` statements or globally using *linear temporal logic*. That's it. No pointers, functions, parameters, classes, generics, constructors. Since the syntax and semantics of the sequential part of Promela are relatively familiar, the instructor can concentrate on teaching concurrency and verification, while students can write models with very little learning "overhead."

7 TOOLS FOR TEACHING WITH SPIN

Spin is written in C and is distributed as a single executable file for Windows and Linux. One way that Spin achieves efficiency is that it does not perform the model checking itself; instead, it generates a highly optimized model checking program in C for each model and each correctness claim to be verified. The user need not look at this C code, although to use Spin you need to have a C compiler installed. A verification in Spin can be run with a three-line script (generate the verifier, compile it and run it); alternatively, a development environment can be used. A TCL/TK-based environment XSpin was developed by Holzmann. For pedagogical use, I developed the JSpin environment. Aside from the standard features of an environment (file handling, editing and invoking the tools), jSpin includes a filtering capability that can present the output in a more understandable form. It is worth emphasizing that when using jSpin, the model checking itself is being performed by the professional Spin software, so that the experience gained is directly transferable to graduate research and industrial practice.

An Example

Let us return to the example of updating a global variable and let us execute it in a loop, ten times in each process:

```
integer n = 0;

process P
integer regP = 0;
do 10 times
     load n into regP
     increment regP
     store regP into n
end

process Q
integer regQ = 0;
do 10 times
     load n into regQ
     increment regQ
     store regQ into n
end
```

Before reading further, try to analyze the program yourself and answer the question: What are the possible values of n when the program terminates?

Here is the Promela source code for this program:

```
byte n = 0, finish = 0;

active [2] proctype P() {
  byte register, counter = 0;
  do :: counter = 10 -> break
     :: else ->
            register = n;
            register++;
            n = register;
            counter++
  od;
  finish++
}
active proctype WaitForFinish() {
  finish == 2;
  printf("n = %d\n", n)
}
```

The declaration `active [2] proctype P()` creates two processes so we don't have to replicate source code in order to model multiple processes. The guarded command `do-od` has two alternatives; the guards are evaluated and a nondeterministic choice is made between them. The alternative `else` is taken only if all other alternatives are false. Therefore, the command in this program implements a familiar for-loop. The only truly unfamiliar statement is the *expression* `finish == 2`. In Promela, an expression can be used as a statement and its meaning is: if the expression is true go to the next statement; otherwise, the process is *blocked*. Of course, such a statement is meaningful only in the context of a concurrent program where another process can change the values of variables so that the expression becomes true and thus executable. The intention here is that the value of n be printed only when both processes `P` have terminated. When `finish` is equal to 2, the only executable process is `WaitForFinish` and it will print the value of `n`.

Spin can be run in four modes:

- *Random simulation* mode uses a random number generator to resolve the nondeterminism inherent in a concurrent program (from which process should an instruction be executed?) as well as the possible nondeterminism in the guarded commands of a single process. In this mode, Spin can replace the classical concurrency simulator [6] as a tool for studying concurrent programs.
- *Interactive simulation mode* enables the user to choose the next instruction to be executed. Interactive simulation is also supported in concurrency simulators and is essential for demonstrating scenarios (such as those for starvation or fairness) that are very unlikely to occur randomly.
- In *verification mode*, Spin systematically searches the entire state space looking for a counterexample, a computation that violates a correctness specification.

- If a counterexample is found, a *trail* of the incorrect computation can be used in *guided simulation* mode to recreate the computation to the user to examine.

If you run the above example in random simulation mode, the interleaving of the two processes is chosen by a random number generator. From experience, the value printed will usually be in the range 14–18. It is easy to see that if the processes are executed sequentially, the final value of n is 20, while if executed in "perfect" interleaving (choosing to execute one statement alternately from each process), the result is 10. Spin can also be run in interactive mode, where the user resolves the nondeterminism inherent in the interleaving of the processes. You can easily get the program to print 20 or 10.

Incredibly, there is a scenario in which the final value of n is 2 [4]! This is highly unlikely to occur in a random simulation, and in fact, it is quite difficult to find the scenario. For years, I taught that the possible answers are between 10 and 20; when a student received the answer 9 on a random simulation, it took quite some time before I was convinced that it was a genuine result and not a bug in the concurrency simulator we were using. Once I figured out how a scenario can give 9 for the final result of n, it was not too difficult to construct the scenario for 2.

Spin can also be run in interactive mode, where the user resolves the nondeterminism inherent in the interleaving of the processes.

Suppose that we claim—as I once believed—that the algorithm always gives results greater than or equal to 10. Let us now ask Spin to prove that correctness claim by adding the assertion `assert (n >= 10)` as the last statement of the process `WaitForFinish`. Run Spin in verification mode and within a few seconds, we are told that the claim is in error. Spin writes a trail file, which can be used by a guided simulation to reconstruct the scenario that is the counterexample. We can also try to verify the program with the assertion `assert (n>2)`. Again, Spin finds an error and can reconstruct the scenario for a counterexample, namely, one for which the final value of n is 2.

Our simple example has used just assertions, but model checkers like Spin are able to find counterexamples for correctness properties expressed in temporal logic such as:

```
[]!deadlock,
```

meaning "it is *always* true that deadlock is false," that is, "the program never deadlocks." This *safety* property is relatively easy to check, since it is sufficient that a single state exist where deadlock is true. Another example is:

```
[](request-resource ->
      <>granted-resource),
```

meaning "*always* (if a resource is requested, it is later granted)" or "whenever a resource is requested, *eventually* it is granted." This *liveness* property is difficult to check, since a counterexample is one in which a computation containing a request for the resource continues *indefinitely* without granting the resource. In terms of the graph of the state space, there must be a *strongly-connected component (SCC)* where `granted-resource` is false in all of its states and the SCC is reachable from a state where `request-resource is true`.

8 NONDETERMINISM

Nondeterminism is a central concept in computer science. It was first defined by Rabin and Scott for their nondeterministic finite automata [15] and the most intractable theoretical problem in CS (P=NP?) asks whether nondeterminism can make algorithms more efficient. Nondeterminism appears frequently in applications: grammars of programming languages, algorithms, and the interleaving model of concurrency. Nevertheless, it is a difficult subject to teach. For a historical survey of nondeterminism and an analysis of the pedagogical problems, see [1].

At first glance, models checkers like Spin would seem to be inappropriate for teaching nondeterminism, because the semantics of concurrency is *universal* (all computations must be correct), whereas for an NDFA the semantics is existential (a string is accepted if there exists a computation that terminates in an accepting state after reading the entire string). However, a simple technique enables Spin to implement NDFAs and other nondeterministic algorithms such as the 8-queens problem (see Chapter 8 of [3] for several examples).

An NDFA is easily programmed using the guarded if-command for nondeterministic transitions. Consider, for example, an NDFA whose set of transitions from state q5 is:

{(q5, a, q7), (q5, a, q3), (q5, b, q5)}.

This can be easily modeled by the labeled guarded command:

```
q5:
if
:: input == 'a' ->
     input = next-symbol; goto q7
:: input == 'a' ->
     input = next-symbol; goto q3
:: input == 'b' ->
     input = next-symbol; goto q5
fi
```

Next, add the alternative:

```
:: end-of-input -> assert(false)
```

to the guarded command for every accepting state. Accepting computations of the NDFA now correspond to computations that ex-

A Primer on Model Checking

continued

ecute assert(false). Since Spin only reports computations that are errors, we artificially make "good" computations (one that are accepting for the NDFA) into errors to be found.

Each of Spin's modes of execution corresponds to a different way of understanding NDFAs:

- Random simulation is the execution of the NDFA with arbitrary resolution of nondeterministic transitions.
- Interactive simulation is the execution of an NDFA with an oracle (you) ensuring that an accepting computation is found.
- Verification represents the metalevel determination if there exists an accepting computation or not.

The VN software tool that I developed demonstrates these concepts. It automatically generates the Promela program from a graphical representation of an NDFA. Accepting and rejecting computations are also displayed graphically (Figure 2). By slight modifications of the generated Promela programs, it was easy to add other features: searching for all accepting computations of a given string or of strings of a given length, and partitioning the inputs of a deterministic automaton into equivalence classes.

9 A SIMPLIFIED MODEL CHECKER

Although Spin is relatively easy to use, it is not entirely trivial since it requires that a C compiler be installed. Furthermore, the output of Spin is not uniform or well defined, so the output filtering in jSpin was difficult to develop and is somewhat fragile. Finally, Spin itself is written in C and has little documentation, so it is quite difficult to study the source or to modify it. For these reasons, I developed the Erigone model checker [5] that is compatible with Spin. It solves the above problems: it is a single executable file; the output uses a uniform named association format that is easy read and easy to parse by postprocessors; it is written in a high-level language (Ada 2005) and much effort was invested to ensure that the program is well structured and well documented. There is a version of jSpin that uses Erigone instead of Spin, and VN now uses Erigone instead of Spin so that it is easier to install and use.

Erigone is intended as a pedagogical tool for learning concurrency with model checking. Since it supports a large subset of Promela, when a student needs the performance of a professional tool to verify a model, the transition to using Spin is immediate.

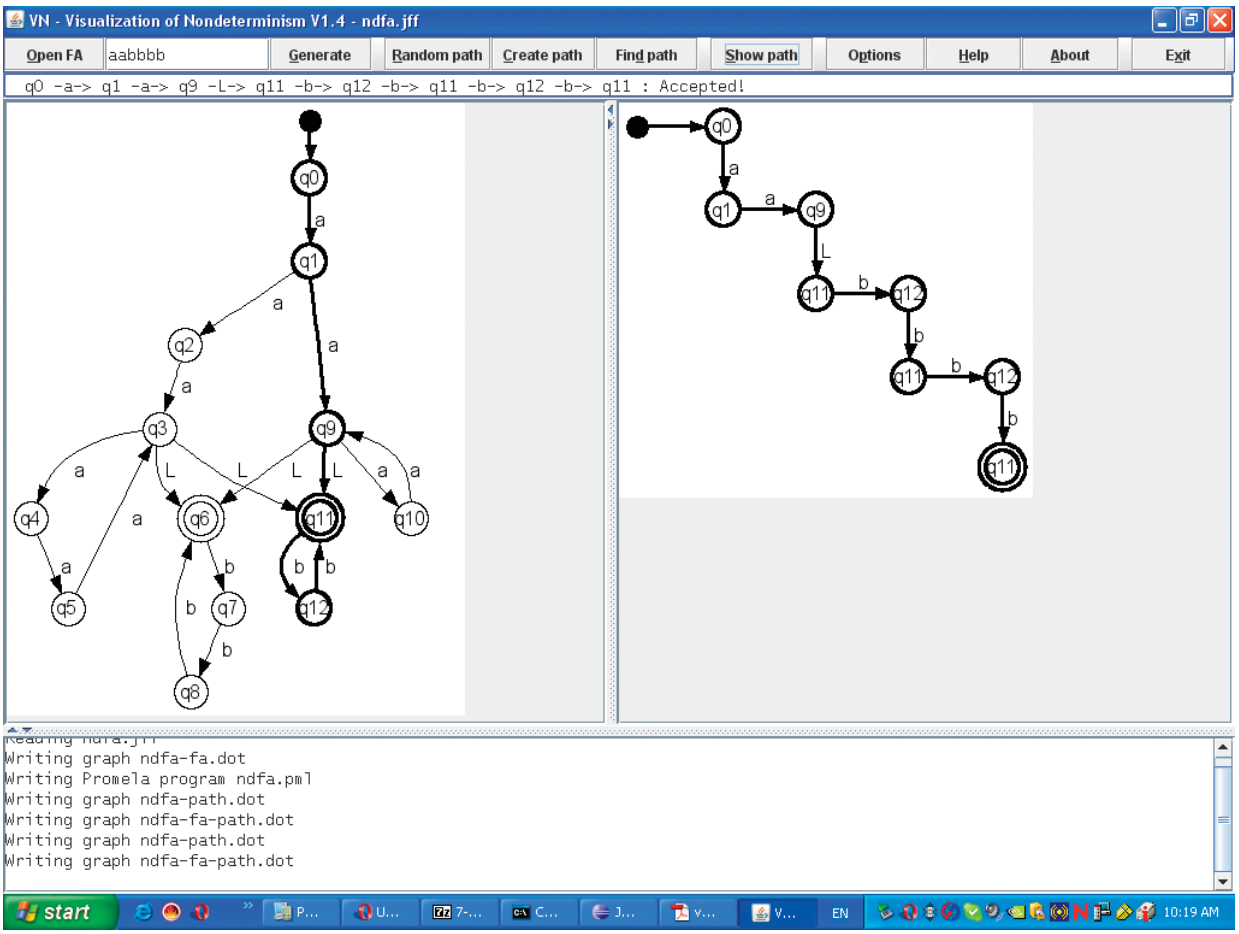

Figure 2: VN visualizing an accepting computation of an NDFA

Erigone can also be used to learn model checking itself. The basic algorithms are relatively simple, but they are based on important concepts like backtracking and data compression, and use central algorithms and data structures like depth-first search of directed graphs, stacks and hash tables.

10 SUMMARY

Formal methods are no longer academic curiosities. They have been used to give customers a guarantee of the correctness of a program [16] (unlike our packaged applications with their infamous "end-use license agreements"), and recently the correctness of the kernel of an operating system has been verified [7].

Model checking is a formal method that can facilitate learning important CS concepts like concurrency, verification, and nondeterminism. Just as importantly, model checking can motivate the study of discrete mathematics and theoretical computer science by showing how they are used in a real-world application: automata theory (programs as NDFAs), logic (the propositional and temporal logics of correctness claims), graph theory (depth-first search and SCCs), hashing functions, data compression.

Although Spin is a professional software tool widely used in industry, the simple and clean syntax and semantics of Promela, and the ease of running simulations and verifications make it ideal as a teaching tool, and I have developed tools such as jSpin and Erigone to simplify the use of model checking even further. I believe that formal methods should be taught as early as possible in the computer science curriculum and model checking is an excellent way to do so.

ACKNOWLEDGEMENTS

I would like to thank Peter Henderson for suggesting that I write this article and for his helpful comments on early drafts. Michal Armoni assisted in the design of VN. Gerard Holzmann's help was invaluable during the development of the software tools.

References

[1] Michal Armoni and Mordechai Ben-Ari. The concept of nondeterminism: Its development and implications for education. *Science & Education*, 18(8):1005–1030, 2009. Reprinted in: *SIGCSE Bull.* 41(2), 2009, 141--160.
[2] Mordechai Ben-Ari. *Principles of Concurrent and Distributed Programming (Second Edition).* Addison-Wesley, Harlow, UK, 2006.
[3] Mordechai Ben-Ari. *Principles of the Spin Model Checker*. Springer, London, 2008.
[4] Mordechai Ben-Ari. Tool presentation: Teaching concurrency and model checking. In *Proceedings of the 16th International SPIN Workshop on Model Checking Software*, pages 6–11, Berlin, Heidelberg, 2009. Springer-Verlag.
[5] Mordechai Ben-Ari and Alan Burns. Extreme interleavings. *IEEE Concurrency*, 6(3):90–91, 1998.
[6] Bill Bynum and Tracy Camp. After you, Alfonse: A mutual exclusion toolkit. *SIGCSE Bull.*, 28(1):170–174, 1996.
[7] Alessandro Cimatti, Edmund M. Clarke, Enrico Giunchiglia, Fausto Giunchiglia, Marco Pistore, Marco Roveri, Roberto Sebastiani, and Armando Tacchella. NuSMV 2: An opensource tool for symbolic model checking. In *CAV '02: Proceedings of the 14th International Conference on Computer Aided Verification*, pages 359–364, London, UK, 2002. Springer-Verlag.
[8] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, 1969.
[9] Gerard J. Holzmann. *The Spin Model Checker: Primer and Reference Manual.* Addison-Wesley, Boston MA, 2004.
[10] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Michael Norrish, Rafal Kolanski, Thomas Sewell, Harvey Tuch, and Simon Winwood. seL4: Formal verification of an OS kernel. In Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles, Big Sky, MT, 2009, 207–220..
[11] Kim G. Larsen, Paul Pettersson, and Wang Yi. UPPAAL in a nutshell. *International Journal on Software Tools for Technology Transfer*, 1(1–2):134–152, 1998.
[12] Zohar Manna and Amir Pnueli. *The Temporal Logic of Reactive and Concurrent Systems. Volume I: Specification.* Springer-Verlag, New York, 1992.
[13] Zohar Manna and Amir Pnueli. *The Temporal Verification of Reactive Systems. Volume II: Safety*. Springer-Verlag, New York, 1995.
[14] Hideaki Nishihara, Koichi Shinozaki, Koji Hayamizu, Toshiaki Aoki, Kenji Taguchi, and Fumihiro Kumeno. Model checking education for software engineers in Japan. *SIGCSE Bull.*, 41(2):45–50, 2009.
[15] Michael O. Rabin and Dana Scott. Finite automata and their decision problems. *IBM Journal of Research and Development*, 3(2):636–644, 1959.
[16] Philip E. Ross. The exterminators. *IEEE Spectrum*, pages 36–41, September 2005.
[17] Yasuyuki Tahara, Nobukazu Yoshioka, Kenji Taguchi, Toshiaki Aoki, and Shinichi Honiden. Evolution of a course on model checking for practical applications. *SIGCSE Bull.*, 41(2):38–44, 2009.
[18] Willem Visser, Klaus Havelund, Guillaume Brat, and SeungJoon Park. Model checking programs. In *ASE '00: Proceedings of the 15th IEEE International Conference on Automated Software Engineering*, page 1–36, Washington, DC, USA, 2000. IEEE Computer Society.

MORDECHAI (MOTI) BEN-ARI
Department of Science Teaching
Weizmann Institute of Science
Rehovot 76284 Israel
benari@acm.org
http://stwww.weizmann.ac.il/g-cs/benari/

Categories and Subject Descriptors: F.3.1 [Logics And Meanings Of Programs] Specifying and Verifying and Reasoning about Programs; K.3.2 [Computers and Education] Computer and Information Science Education
General Terms: Verification
Keywords: Model checking, Verification, Concurrent programming, Spin, Erigone

DOI: 10.1145/1721933.1721950
©2010 2153-2184/10/0300 $10.00☆

Explore Computing History

The Charles Babbage Institute

www.cbi.umn.edu