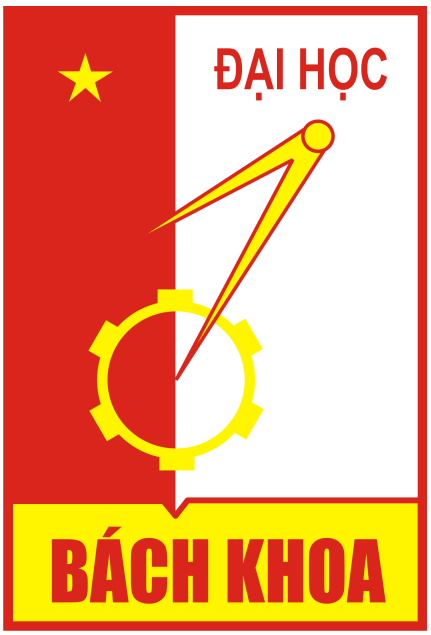
Final Term Project

Automata Application In System Verification

Instructor: Ph.D Trần Vĩnh Đức

Group 2:
Nguyễn Văn Lợi
Hồ Sỹ Thành
Cao Tấn Minh



School Of Information And Communication Technology
Hanoi University Of Science And Technology
Ha Noi
January 14, 2018

Contents

List of Figures

List of Tables

This page intentionally left blank

Part I
Introduction

This project is about using specific application of automata in verification. The purpose of this documentation is to give you an overview of automata and apply it to solve some practical problem with the help from the well-known software verification tool: Spin.

1 Automata

Automata theory is the study of abstract machines and automata, as well as the computational problems that can be solved using them.

The figure 1 shows us various classes of automata theory. Each one has some different

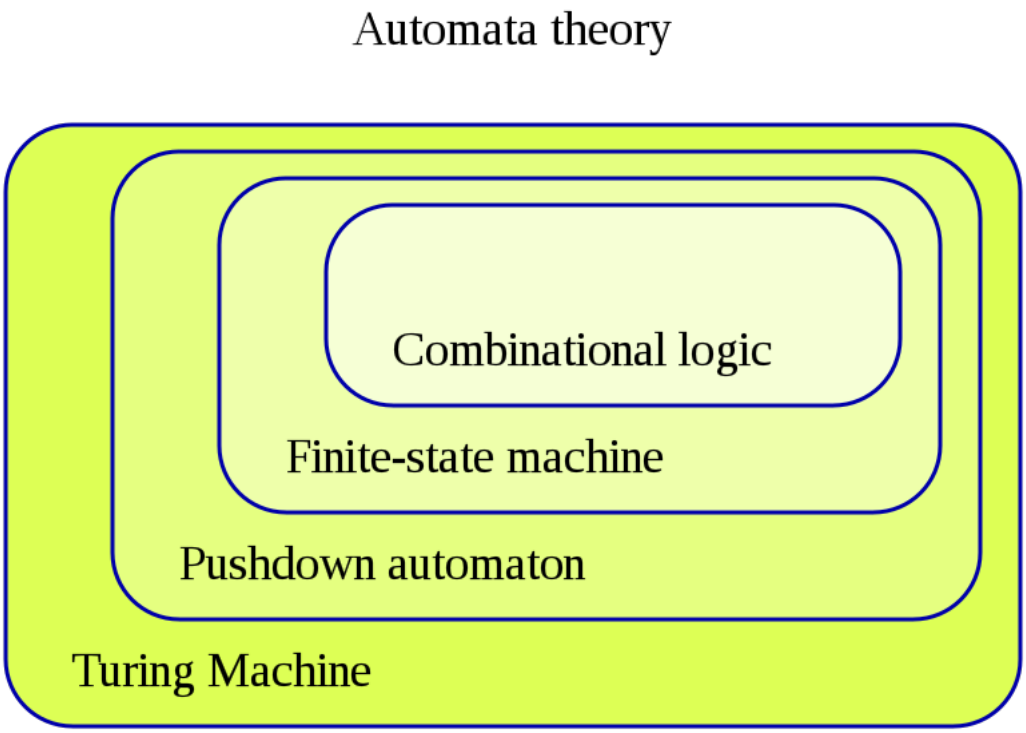


Figure 1: The Classes of Automata

properties and is used for different purposes. Automata is also a major part of some field like artificial intelligent, theory of computation,...

1.1 Definition

A deterministic finite automaton is represented formally by a 5-tuple $\langle Q, \Sigma, \delta, q_0, F \rangle$, where:

- Q is a finite set of states.
- Σ is a finite set of symbols, called the alphabet of the automaton.
- δ is the transition function.

- q_0 is the start state.
- F is accept states

Other key concepts you should keep in mind are Input Word, Run, Accepting Word, Recognized Language, Recognizable languages,...

Input Word An automaton reads a finite string of symbols $a_1, a_2, ..., a_n$, where $a_i \in \Sigma$, which is called an input word. The set of all words is denoted by Σ^*.

Run A sequence of states $q_0, q_1, q_2, ..., q_n$, where $q_i \in Q$ such that q_0 is the start state and $q_i = \delta(q_{i-1}, a_i)$ for $0 < i \leq n$, is a run of the automaton on an input word $w = a_1, a_2, ..., a_n \in \Sigma^*$. In other words, at first the automaton is at the start state q_0, and then the automaton reads symbols of the input word in sequence. When the automaton reads symbol a_i it jumps to state $q_i = \delta(q_{i-1}, a_i)$. q_n is said to be the final state of the run.

Accepting Word A word $w \in \Sigma^*$ is accepted by the automaton if $q_n \in F$

Recognized Language An automaton can recognize a formal language. The language $L \subseteq \Sigma^*$ recognized by an automaton is the set of all the words that are accepted by the automaton.

Recognizable languages The recognizable languages are the set of languages that are recognized by some automaton. For the above definition of automata the recognizable languages are regular languages. For different definitions of automata, the recognizable languages are different.

1.2 Lemma

Pumping lemma If A is a regular language, then there is a number p (the pumping length) where if s is any string in A of length at least p, then s may be divided into three pieces, $s = xyz$, satisfying the following conditions:

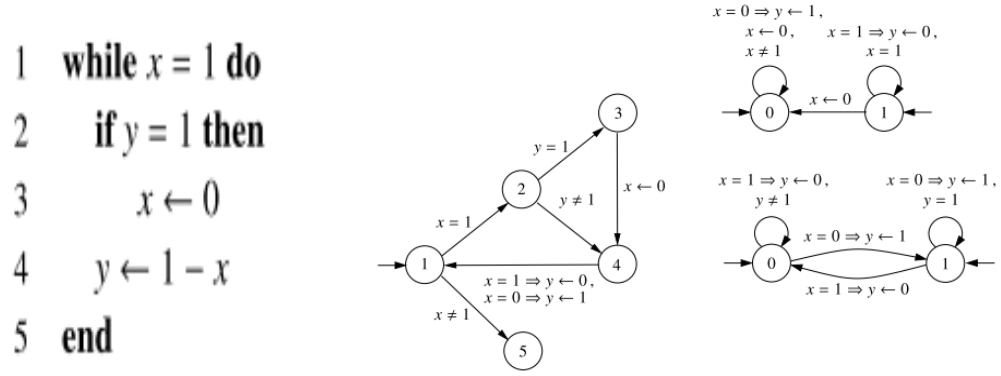
1. for each $i \geq 0$, $xy^iz \in A$,
2. $|y| > 0$, and
3. $|xy| \leq p$.

2 Automata application: system verification

Each model, class in automata theory plays an important role in several specific fields, like text processing, artificial intelligent, compiler construction. One of the main application of automata theory is the automata verification of a hardware of software systems. Given a system and some properties as input, we would use automata to determine whether or not this system satisfies that properties.

2.1 Programs As Networks Of Automata

Given a program, we will build a network of automata based on each variable and program counter. For example, with the program in figure 2



(a) Program (b) Network of automata

Figure 2: Program with two boolean variable x, y and all states are final

2.2 Concurrent Programs

With the help of asynchronous product and graph algorithms, we can verify concurrent program more efficiently. For example, with a concurrent program in figure 3, firstly we can build a network of automata in figure 4. Then using algorithm in figure 5, we can construct a automata result in figure 6. Finally, analyze the requirement and apply the breadth first search or depth first search to search for solution in asynchronous product automata.

```
        repeat                          repeat
nc0 :       b0 ← 1              nc1 :       b1 ← 1
t0  :       while b1 = 1 do skip t1 :       if b0 = 1 then
c0  :       b0 ← 0              q1  :           b1 ← 0
        forever                 q'1 :           while b0 = 1 do skip
                                                goto nc1
                                c1  :       b1 ← 0
                                        forever
```

Figure 3: Concurrent program

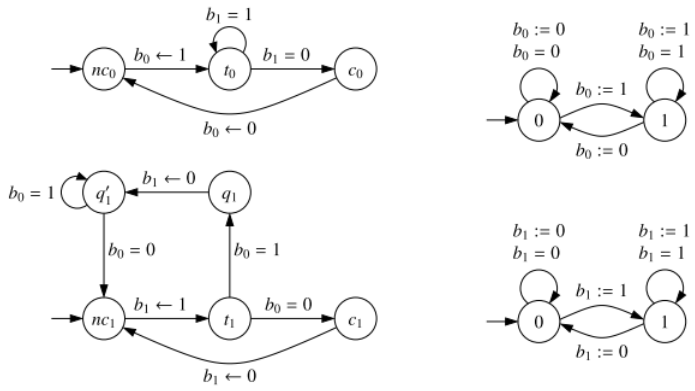


Figure 4: Network of Automata

2.3 Coping With The State-Explosion Problem

The key problem of this approach is that the number of states of E can be as high as the product of the number of states of the the components $A_1, ..., A_n$, which can easily exceed

AsyncProduct$(A_1, \ldots, A_n)$
Input: a network of automata $\mathcal{A} = \langle A_1, \ldots, A_n \rangle$, where
$A_i = (Q_i, \Sigma_i, \delta_i, Q_{0i}, F_i)$ for every $i = 1, \ldots n$.
Output: NFA $A_1 \otimes \cdots \otimes A_n = (Q, \Sigma, \delta, Q_0, F)$ recognizing $L(\mathcal{A})$.

```
1  Q, δ, F ← ∅
2  Q_0 ← Q_01 × ··· × Q_0n
3  W ← Q_0
4  while W ≠ ∅ do
5     pick [q_1, ..., q_n] from W
6     add [q_1, ..., q_n] to Q
7     if ⋀_{i=1}^n q_i ∈ F_i then add [q_1, ..., q_n] to F
8     for all a ∈ Σ_1 ∪ ... ∪ Σ_n do
9        for all i ∈ [1..n] do
10          if a ∈ Σ_i then Q'_i ← δ_i(q_i, a) else Q'_i = {q_i}
11       for all [q'_1, ..., q'_n] ∈ Q'_1 × ... × Q'_n do
12          if [q'_1, ..., q'_n] ∉ Q then add [q'_1, ..., q'_n] to W
13          add ([q_1, ..., q_n], a, [q'_1, ..., q'_n]) to δ
14 return (Q, Σ, δ, Q_0, F)
```

Figure 5: Asynchronous product algorithm

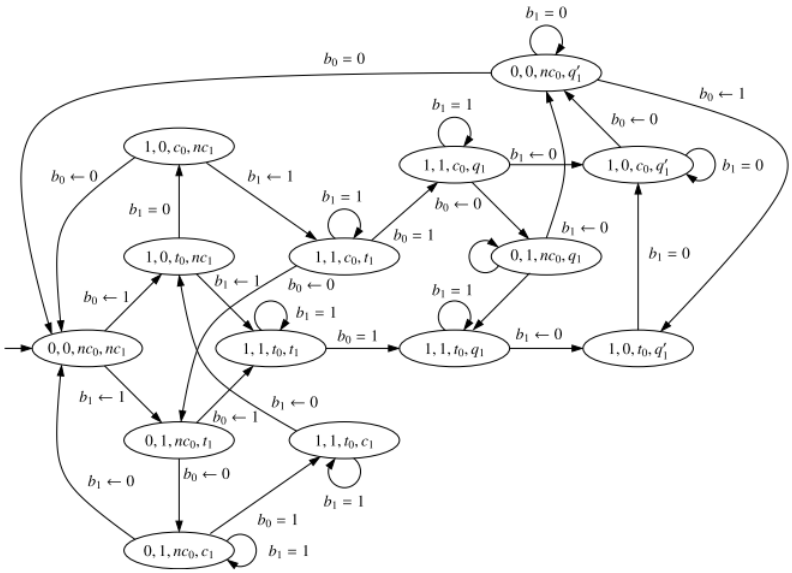


Figure 6: Asynchronous product

the available memory. This is called the state-explosion problem, and the literature contains a wealth of proposals to deal with it.

The automata-theoretic approach constructs an NFA V recognizing the potential executions of the system that violate the property one is interested in, and checks whether the automaton $E \cap V$ is empty, where E is an NFA recognizing the executions of the system. This is done by constructing the set of states of $E \cap V$, while simultaneously checking if any of them is final.

3 Spin project

Spin is a popular open-source software verification tool, used by thousands of people worldwide. The tool can be used for the formal verification of multi-threaded software applications. The tool was developed at Bell Labs in the Unix group of the Computing Sciences Research Center, starting in 1980. The software has been available freely since 1991, and continues to evolve to keep pace with new developments.

Spin provides direct support for the use of multi-core computers for model checking runs

- supporting the verification of both safety and liveness properties.
  Spin works on-the-fly which is really helpful to deal with state-explosion problem

Part II
Spin and Promela model language

4 Promela

Promela is a verification modeling language. It provides a vehicle for making abstractions of protocols (or distributed systems in general) that suppress details that are unrelated to process interaction. The intended use of Spin is to verify fractions of process behavior, that for one reason or another are considered suspect. The relevant behavior is modeled in Promela and verified. A complete verification is therefore typically performed in a series of steps, with the construction of increasingly detailed Promela models. Each model can be verified with Spin under different types of assumptions about the environment (e.g., message loss, message duplications etc). Once the correctness of a model has been established with Spin, that fact can be used in the construction and verification of all subsequent models.

Promela programs consist of processes, message channels, and variables. Processes are global objects. Message channels and variables can be declared either globally or locally within a process. Processes specify behavior, channels and global variables define the environment in which the processes run.

4.1 Data types

4.1.1 Primitive data types

Typename	C-equivalent
bit or bool	bit-field
byte	uchar
short	short
int	int

Table 1: Promela data types

4.1.2 Structure data types

They are very similar to C syntax. For instance,

```
short newArray[20];
```

declares an array called newArray of 20 short element

4.2 Executability

In Promela, there is no difference between conditions and statements, we will call both of them statements. A statement is either executable or blocked. when statement is executable, the process runs and when statement is blocked, the process stops at that point and wait until statement is executable again. Therefore, instead of writing a C while loop:

```
while(a != b)
        continue;
```

we just need to type:

```
(a == b) \\ wait for a == b and then statement is executable
```

4.3 Process types

One program may contain multiple processes running simultaneously. Thus we need a syntax and also a type to model those processes. In Promela, it provides process type which manage and inspect the shared variables and message channel between processes. Below is the definition of a process A with internal variable a:

```
proctype A()
{
        int a;
        a = 5;
}
```

In Promela, the semicolon and the arrow are both valid **statement separator**. But for coding convention we usually use arrow for if-else statement to indicate causal relation between two statement:

```
int count = 3;
proctype firstProcess()
{
        (count == 4) -> count = 3;
}
proctype secondProcess()
{
        count = count + 1;
}
```

In this example, we also see that two processes are manipulating the same data, the shared global variable count.

4.4 Process Instantiation

In every Promela program, there is at least one process that is always be executed: a process of type **init**. For example,

```
init
{
        short index = 0;
        index = index + 2;
}
```

seems like any other processes, it declares a variable named index and increases it with 2. However, the **init** process can initialize global variables and processes using run statement. So the common code structure would be,

```
int count;
proctype firstProcess()
{
        (count == 4) -> count = 3;
}
proctype secondProcess()
{
        count = count + 1;
}
init
{
        count = 3;
        run firstProcess();
        run secondProcess();
}
```

However, if we want multiple processes start at the same time, we can instead use **active** keyword preceding the process prototype,

```
int count;
active proctype firstProcess()
{
        (count == 4) -> count = 3;
}
active proctype secondProcess()
{
        count = count + 1;
}
```

we can of course put parameter to the processes like in C functions,

```
int count;
proctype firstProcess(int gap)
{
        (count == 4) -> count = gap;
}
init
{
        count = 4;
        run firstProcess(3);
}
```

4.5 Atomic Sequences

Use **atomic** keyword to create Atomic sequences. It indicates that the inside sequences will be executed as atomic sequences as one indivisible unit. Syntax:

```
atomic
{
        sequences of code;
}
```

4.6 Message Passing

Message channels are used to model the transfer of data from a process to another. They are declared either locally or globally, for instance as follows:

```
chan qname = [16] of { short, short, int, chan };
```

We can think of a channel as a queue data structure. Here, qname is a name of specific channel, chan is the channel type, this channel can store up to 16 messages (queueSize is equal to 16), each message contains two short variable, one int variable and one channel name.

The statement:

```
qname!expr
```

sends the value of expr to channel qname. This statement will be blocked if channel is full

The statement:

```
qname?msg
```

sends a massage from channel to msg. This statement will be blocked if channel is empty

Here is an example that uses some of the mechanisms introduced so far:

```
proctype A(chan q1)
{
      chan q2;
      q1?q2;
      q2!123
}

proctype B(chan qforb)
{
      int x;
      qforb?x;
      printf("x = %d\n", x)
}

init {
      chan qname = [1] of { chan };
      chan qforb = [1] of { int };
      run A(qname);
      run B(qforb);
      qname!qforb
}
```

here, q_1, qname refer to the same channel, q_2, qforb refer to the same channel

4.7 Controll Flow

Case selection Using relative values between two variables a and b to choose between two options:

```
if
:: (a == b) -> sequence of statements
:: (a != b) -> sequence of statements
fi
```

Repetition Instead of if keyword, we use do keyword

```
do
:: sequence of statements
:: (break condition statement) -> break
od
```

Unconditional jumps It's identical to C, which use goto label for unconditional jump:

```
if
:: sequence of statements
:: (a != b) -> goto labelA
fi
labelA: sequence of statements
```

4.8 Modeling Procedures and Recursion

Procedures can be modeled as processes, even recursive ones. The return value can be passed back to the calling process via a global variable, or via a message. The following program illustrates this:

```
proctype fact(int n; chan p)
{
```

```
        \\ n channel will be create in total
        chan child = [1] of { int };
        int result;

        if
        :: (n < 0) -> goto Error
        :: (n <= 1) -> p!1
        :: (n >= 2) ->
                \\ compute (n - 1)! and sends it to child channel
                run fact(n-1, child);
                \\ sends (n - 1)! from child channel to result
                child?result;
                \\ sends n! to p channel
                p!n*result;
        fi
        Error: printf("Invalid input value\n");
}
init
{
        chan child = [1] of { int };
        int result;

        run fact(7, child);
        child?result;
        printf("result: %d\n", result)
}
```

5 Spin

Spin is a tool for analyzing the logical consistency of concurrent systems, specifically of data communication protocols. The system is described in a modeling language called Promela (Process Meta Language). The language allows for the dynamic creation of concurrent processes. Communication via message channels can be defined to be synchronous (i.e., rendezvous), or asynchronous (i.e., buffered).

Given a model system specified in Promela, Spin can either perform random simulations of the system's execution or it can generate a C program that performs an efficient online verification of the system's correctness properties. During simulation and verification Spin checks for the absence of deadlocks, unspecified receptions, and unexecutable code. The verifier can also be used to verify the correctness of system invariants, it can find non-progress execution cycles, and it can verify correctness properties expressed in next-time free linear temporal logic formulae.

5.1 The Simulator

If Spin is invoked without any options it performs a random simulation. With option -n N the seed for the simulation is set explicitly to the integer value N .

A group of options pglrs can be used to set the desired level of information that the user wants about the simulation run. Every line of output normally contains a reference to the source line in the specification that caused it.

- -p Shows the state changes of the Promela processes at every time step.
- -g Shows the current value of global variables at every time step.
- -l Shows the current value of local variables, after the process that owns them has changed state. It is best used in combination with option -p.

- -r Shows all message receive events. It shows the process performing the receive, its name and number, the source line number, the message parameter number (there is one line for each parameter), the message type and the message channel number and name.
- -s Shows all message send events.

5.2 The Analyzer

Spin understands three other options (-a, -m, and -t):

- -a Generates a protocol specific analyzer.
- -t Write to .trail file if there was any error

Part III
Program

6 Requirement

Design a model, build it and verify it using Spin

Prerequisites

1. cygwin32
2. spin
3. C-compiler and C-preprocessor
4. iSpin GUI (recommended)
5. ActiveTcl
6. graphviz-2.38 (recommended)

7 Design

Design two threads with a deadlock between the two, using global variable as shared variable instead of message channel

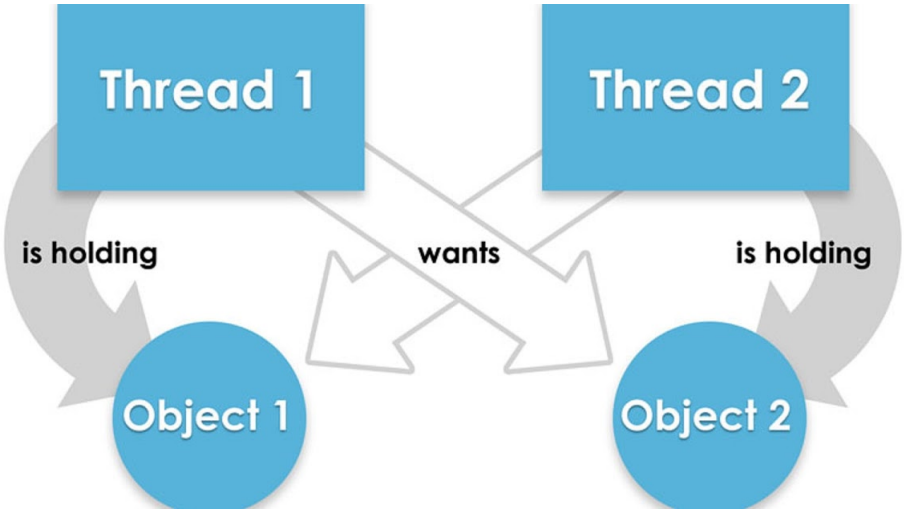


Figure 7: Deadlock of threads

8 Implementation

1. Install cygwin32
   - step 1: install cygwin32 using setup-x86.exe file
   - step 2: after 5 next steps, select first site and click next
   - step 3: after 6 next steps, "Select Packages" window opened, install all necessary packages for spin: bison(devel), yacc(devel), gcc-core: GNU Compiler Collection, gcc-g++: GNU Compiler Collection, libgcc1, gdb: The GNU Debugger, make: The GNU version of the 'make' utility, WindowMaker: GNUstep window manager

    - step 4: create new search path to bin folder of cygwin32
2. Install spin: in pc_ spin647, copy spin.exe to bin folder of cygwin32
3. Install C-compiler and C-preprocessor
    - not needed if you have followed all above steps
    - needed to run model checker generated by spin
    - (recommended): GNU gcc compiler
    - can be installed using cygwin32 setup above.
    - after step 3, it's done if you don't want to use graphical user interface
4. Install iSpin - graphical user interface for spin
    - step 1: in pc_ spin647, copy ispin to any where you plan to work
    - step 2: to start ispin, you need Tcl/Tk programming language
5. Install ActiveTcl: one distribution of Tcl PL
    - step 1: in Final, open ActiveTcl*.exe
    - step 2: following step to insatll ActiveTcl
6. Install graphviz-2.38 for automata view
7. Configure ispin
8. Create search path
9. Use Spin
10. write model using Promela
11. simulate, search and verify model

For details, please see README.md file

9 Testing, Validation and System Maintenance

Read README.md file for additional information

10 User documentation

User documentation was written in README.md file. Read README.md file for additional information