Concurrent and Distributed Systems

Talk Four

Introduction to Model Checking
and
Promela Language

Maxim Mozgovoy

On Formal Verification (1)

Concurrent programs are hard to develop, debug, and test.

If some program works now, we cannot be sure it works always.

Race conditions may happen (in some cases our program works, but sometimes it fails – the result depends on the events beyond our control, i.e., *luck*).

How else can we improve their reliability?

One of the possible instruments is formal verification.

On Formal Verification (2)

Formal verification is the process of proving the correctness of your program (in contrast to more popular testing).

E.g., it is possible to provide a strict proof that quicksort indeed sorts arrays, and that Dijkstra's algorithm indeed finds the shortest path in the graph between the two given nodes (similar to proving mathematical theorems).

Numerous tools exist for computer-assisted formal verification.

E.g., a verification system can check various user assumptions about the code.

On Formal Verification (3)

Formal verification is a complicated and slow process.

In practice it is used when potential bugs are truly expensive (i.e., in medical systems, aerospace software, hardware chips).

Ex: in 1994 a bug was discovered in Intel's Pentium processor. Some floating-point arithmetical operations provided inaccurate results.

Intel decided to replace defective units, losing around $475 mln

(around $728 mln in 2013 prices)

On Formal Verification (4)

Formal verification is a complicated and slow process.

In practice it is used when potential bugs are truly expensive (i.e., in medical systems, aerospace software, hardware chips).

Ex: in 1996 a European rocket Ariane 5 has crashed due to a software bug. Old fragments of code for Ariane 4 were reused in Ariane 5, but some new 64 bit sensor data couldn't fit old 16 bit variables, causing overflow.

The losses were over $370 mln

(around $551 mln in 2013 prices)

Model Checking

Model checking is a type of formal verification, useful for concurrent programs.

Model checking software can analyze different scenarios of code execution to find potential deadlocks and race conditions.

Many checkers can test some user-specified properties of a system (a specialized language is used to describe properties).

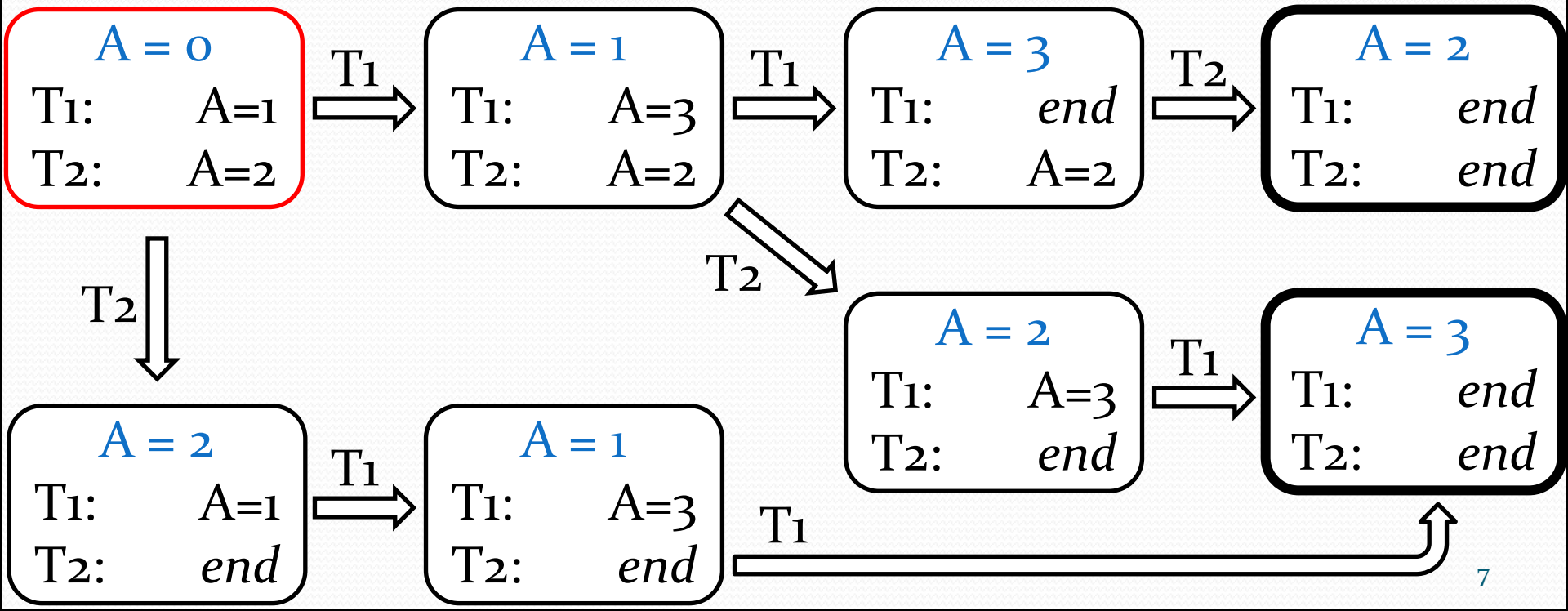
One of the most popular general-use model checkers is SPIN. It requires that the programs to be checked are written in a special modeling language called Promela.

There are tools and techniques for Promela <=> C conversion, assisting model checking of real C code.

Model Checking – How?

Model checking is done by building and analyzing the program's state space diagram (see **Talk 2**).

The complete diagram is impossible to build (it is huge), but various optimizations are used in practical model checkers.



Installing and Running SPIN (1)

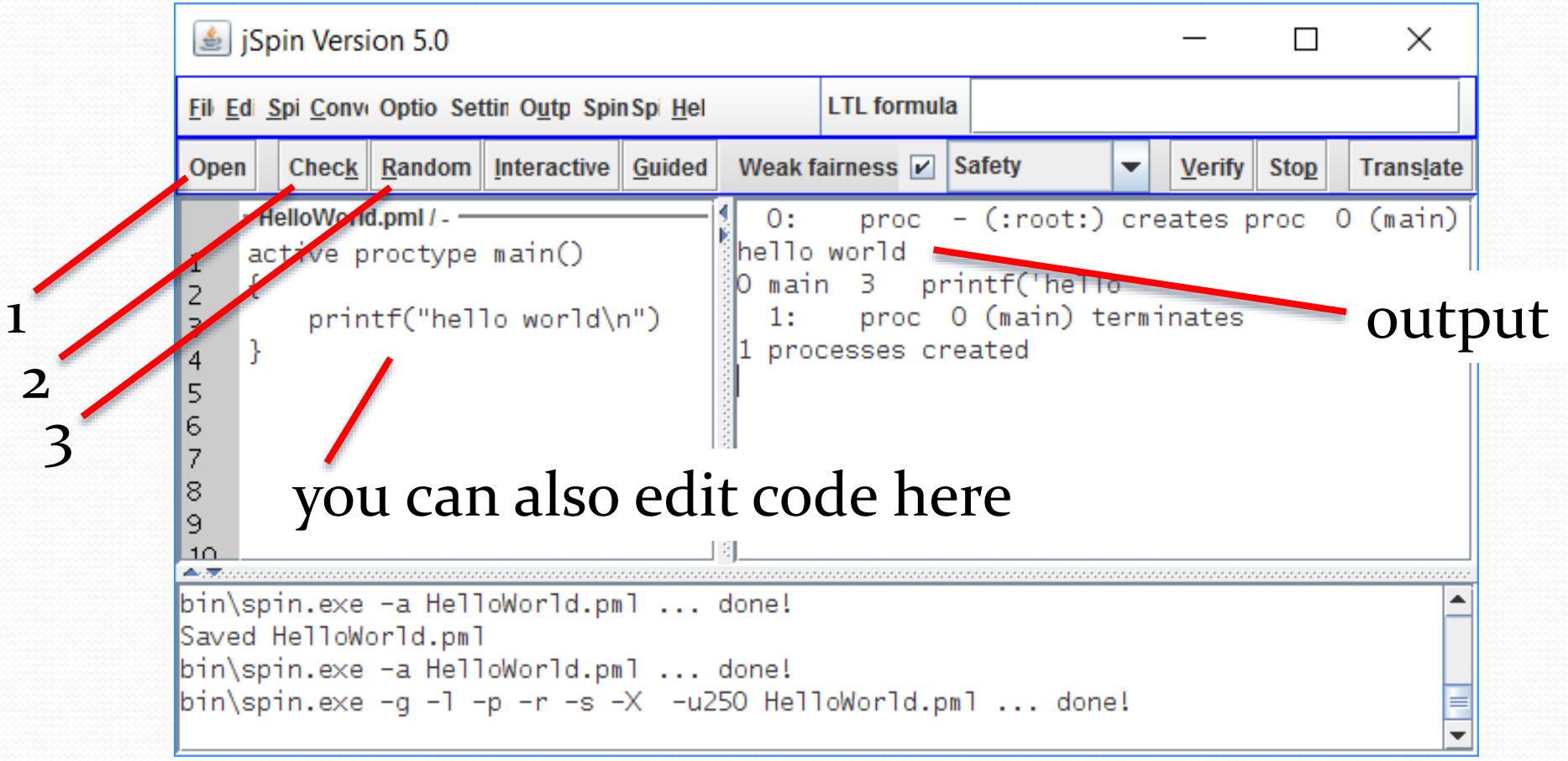
SPIN is a complex command-line software, but we will use a convenient graphical frontend jSpin. Please try to install jSpin using the links from our Moodle page and create your first Promela program:

```
active proctype main()
{
    printf("hello world\n")
}
```

Save it into the file HelloWorld.pml

Installing and Running SPIN (2)

Open HelloWorld.pml with **Open**[1] button in jSpin, and check it for syntax errors **Check**[2] button. Then click **Random**[3] to run the code. You should see the output with some debug info.



Writing Code in Promela

We will use only small subset of Promela to keep things simple.

In general:

- A program consists of several processes running in parallel.
- Processes can use local and global variables.
- Each statement is treated as atomic.
- SPIN runs parallel Promela code by interleaving statements (i.e. as if it is executed on a single-core CPU).
- If there is a choice which statement to run next, SPIN selects it randomly (hence, **Random** button in jSpin).
- Some statements can block the process until certain condition is changed (like a semaphore).

Declaring Processes

Use **active proctype** *Name*() construction:

```
active proctype P1() {
    printf("I am P1\n")   // printf() is used like
}                         // in C language

active proctype P2() {
    printf("I am P2\n")   // note: use newline \n
}                         // otherwise you might
                          // not see the output
```

Result:

```
I am P1
I am P2
```

or

```
I am P2
I am P1
```

Declaring Copies of the Same Process

Use **active [*N*] proctype** *Name*() construction:

```
active [3] proctype P() {
    printf("I am P%d\n", _pid)
}

// _pid is a unique integer process ID
// _pid values are assigned starting from zero
// processes are created in order of declaration
```

Result:

```
I am P1
I am P0
I am P3
```

(or in any other order)

Using Local and Global Variables

Use **bool** and **int** keywords and arithmetic operations:

```
bool flag = false                  // global variable
int z[2] = { 1, 2 }                // initialized array

active [2] proctype P() {
     int x = 5, k = 7 // declare and assign local
     int y[3]                      // variables and arrays
     flag = true                   // assign global variable
     x++                           // perform increment
     y[0] = x + 7                  // assign
     printf("x = %d, y[0] = %d\n", x, y[0])
} // prints "x = 6, y[0] = 13"
```

Expression Statements

Logical expressions can be used as statements!
They will block the process until the specified condition is satisfied (much like semaphores).

```
bool flag = false
active proctype P1(){ flag = true }
active proctype P2(){ printf("P2: %d\n", flag) }
active proctype P3(){ flag == true // expression
                      printf("P3: %d\n", flag) }
```

Result:

P2 will print "P2: 0" or "P2: 1"

P3 will always print "P3: 1"

Careful! Don't block your program forever!

Atomic Blocks (1)

You can combine several instructions into a single atomic block using **atomic** keyword:

```
bool flag = false
active proctype P1(){
    flag = true
    flag = false
}
active proctype P2(){ printf("%d\n", flag) }
```

Result:

P2 will print `0` or `1`

Atomic Blocks (2)

You can combine several instructions into a single atomic block using **atomic** keyword:

```
bool flag = false
active proctype P1(){
    atomic { flag = true
             flag = false }
}
active proctype P2(){ printf("%d\n", flag) }
```

Result:

P2 will always print 0

Statement Separators

Several **statements** (assignments, expressions, etc.) can be placed on the same line if they are separated with a semicolon:

```
active proctype P1(){
     bool flag = true; flag = false
}
```

The symbol -> is used for the same purpose, so you can write:

```
active proctype P1(){
     bool flag = true -> flag = false
}
```

Branching Statement (if…fi) (1)

Promela **if** statements are a bit tricky:

```
if
:: option1 -> statement1; statement2; ...
:: option2 -> statement1; statement2; ...
...
fi
```

```
int x = 5  // example: print "five"
if
:: x == 5 -> printf("five\n")
:: x != 5 -> printf("not five\n")
fi
```

Branching Statement (if…fi) (2)

If several choices match, SPIN will take a random choice:

```
// example: print "five", "positive", or "odd"
int x = 5
if
:: x == 5      -> printf("five\n")
:: x > 0       -> printf("positive\n")
:: x % 2 == 1  -> printf("odd\n")
:: else        -> printf("something else\n")
fi
```

If no choices match, the process will be blocked!

Use **else** option to execute something if all other choices fail.

Looping Statement (do…od)

Promela **do** statement works like a repeated **if**.

Use **break** statement to exit the loop:

```
int i = 0
do
:: i >= 5 -> break
:: else   -> printf("%d\n", i)
             i++
od
```

Result: the loop executes five times.

Note: Since -> is the same as semicolon, lines like

:: $expr_1$ -> $expr_2$ …

contain two or more statements and thus are not atomic!

Additional Capabilities

Symbolic names:

```
mtype = { Red, Green, Blue }  // 3 int constants
…
mtype cr = Red // assign value to a variable cr
```

Preprocessor constants (like in C):

```
#define N 5
…
int a[N]
```

Jumps:

```
goto exit
…
exit: printf("exited!\n")
```

Examples: Array Sorting

```
int Semaphore = 2

active proctype SortFirstHalf(){
    printf("sorting the first half\n")
    Semaphore--
}

active proctype SortSecondHalf(){
    printf("sorting the second half\n")
    Semaphore--
}

active proctype Merging(){
    Semaphore == 0; // runs AFTER sorting
    printf("merging array halves\n")
}
```

Examples: Bank Account (Buggy!)

```
int Semaphore = 2, Account = 0

active [2] proctype Spouse(){
    int t = 0; int i = 0
    do
    :: i >= 10 -> break
    :: else -> t = Account; t++; Account = t; i++
    od
    Semaphore--
}

active proctype main(){
    Semaphore == 0
    printf("Account = %d\n", Account)
}
```

Examples: Bank Account (Fixed, v.1)

```
int Semaphore = 2, Account = 0

active [2] proctype Spouse(){
    int t = 0; int i = 0
    do
    :: i >= 10 -> break
    :: else -> atomic { t = Account; t++
                        Account = t; i++ }
    od
    Semaphore--
}

active proctype main(){
    Semaphore == 0
    printf("Account = %d\n", Account)
}
```

Examples: Bank Account (Fixed, v.2)

```
int Semaphore = 2, Account = 0

active [2] proctype Spouse(){
    int t = 0; int i = 0
    do
    :: i >= 10 -> break
    :: else -> Account++; i++
    od
    Semaphore--
}

active proctype main(){
    Semaphore == 0
    printf("Account = %d\n", Account)
}
```

Conclusions

- When a program is difficult to debug and test, and potential bugs are expensive, formal verification methods are used.
- One of the instruments of formal verification, useful for concurrent programs, is model checking.
- Model checking software can automatically analyze a program's state space diagram and find whether some conditions theoretically can happen.
- In particular, model checkers can test the presence of deadlocks, race conditions, and find out whether some user-defined logical statements hold.
- One of the most popular model checkers is SPIN. It uses Promela language to model algorithms.