A Tutorial Introduction to Shared Memory Concurrency

K. V. S. Prasad
(for the course TDA383/DIT390)
Department of Computer Science
Chalmers University

August 28, 2016

We begin concretely and directly with a standard concurrent programming language. That is, we begin by merely presenting some standard techniques of the day, as if they came directly by applying reason to the world. But of course they didn't. It is instructive and enriching to learn when and why a technique or view arose, and to see how it has changed or can change our world. We do a bit of this later in the course.

Here, we learn to program in Promela[1, 2], a language that shares the approach to concurrency taken by languages such as Java, Erlang and Ada. But it is much easier to directly get into concurrent programming using Promela than any of the other languages. Later, we see what language aspects are missing in Promela, and why it is a modelling or specification language rather than a full-fledged programming language.

Promela sources

Section 4.7 of the textbook[3] is an introduction to Promela. Ben-Ari also gives many example programs linked to the textbook, in Promela, Java, C and Ada.

There are many good lecture slides for Promela available on the web, for example [4, 5, 6, 7], but these are often from somewhat more advanced courses, so you might want to look at them after you have learned a little more. Or skim them without worrying about the bits you don't understand.

Acknowledgement: Much of the description of Promela constructs below has been adapted from the sources [1, 2, 3, 5, 6, 7]. I thank my colleagues Ann Lilieström, Raùl Jiménez and Nicholas Smallbone for their help in improving this document.

1 The Spin interface

The first thing to do is to run our very own Spin interface.

Near the top left is a *cheat sheet*, for later use. To close the cheat sheet, click on the *close window* link at the top left of the sheet. Turn off all the radio buttons in the top right hand pane (labelled "variables", "events", etc.). You are now ready to start.

1.1 Hello World

```
/* Just to show you how to put comments in.
*/
init {printf("Hello World!\n")}
```

Figure 1: `hello.pml`

Click on *examples* at the top left. From the drop-down menu choose `hello.pml`, the obligatory first program. This looks reassuringly like languages you have seen before[8], except that the keyword `main` has been replaced by `init`. The `printf` command is similar to its namesake in C. Its first argument is an arbitrary string in double quotes. The sequence "backspace n" inserts a newline.

```
$ spin  hello.pml
      Hello World!
1 process created
```

Figure 2: Output from `hello.pml`

Run the program by clicking on the *Run* button, and you will see the output above in the bottom right hand pane. The top line tells you what the interface did — it ran the Spin interpreter— and the second line is the printout we asked for. The third line tells us that `init` was run as a *process*.

2 Concurrent Processes

Ordinary sequential programs consist of a sequence of commands to be executed by a nameless active entity, which we perhaps think of as the CPU, or just "the computer" or even "it"—as in "why did it do that?", the despairing cry familiar from debugging. This active entity manipulates data, which is passive.

In a concurrent program, all activity is carried out by *processes* (you can think of them as *actors* or *agents*), several of which may be active at the same time. What "active at the same time" means we will see shortly. Each process is like a sub-program, manipulating data which is either *local* to itself (i.e., private) or *global*, (i.e., shared with other processes).

2.1 The `proctype` declaration and the `run` command

In a Promela program, `init` is a special process, declared and run by the `init` keyword. It runs once, and is used to initialize variables and run other processes.

```
/* A process that prints out its name */

proctype Q() {
    printf("Q\n")
}

init { run Q()}
```

Figure 3: `one-Q.pml`

Every other process, like Q in `one-Q.pml`, must have its code defined by a `proctype` declaration. When Q is `run` (or *spawn*ed) by another process[1], it becomes *active*, and can execute its code[2]. An active process *terminates* when it reaches the end of its code, and all the processes that it spawned have terminated. So `init` in `one-Q.pml` terminates when Q does.

When you run `one-Q.pml`, it prints out "Q" and tells you that 2 processes were created—Q and `init`. It's true that `hello.pml` did as much with less machinery, but the point of a proctype is that more than one instance can be run. If the code in Q is big, it saves us rewriting, just like a procedure or function.

2.2 Multiple instances of the same process

```
/* Three instances of a process that prints out its name */

proctype Q() {
    printf("Q\n")
}

init { run Q(); run Q(); run Q()}
```

Figure 4: `three-Qs.pml`

`Three-Qs.pml` might seem silly too. It always prints out Q on three successive lines. But the output from Spin includes monitoring information—the amount of indentation says which process is printing, and when you run `threeQs.pml` repeatedly, you see differing patterns. If we use differing concurrent processes, we see why.

[1] We call the spawning process the *parent* and the spawned process the *child*.

[2] Note that `run` does *not* mean "run to completion". That would just be a procedure call. Parent and child would not then be able to run at the same time.

2.3 Processes with parameters

```
/* processes printing ids, to see who went first.*/

proctype P(int i) {
    printf("P(%d)\n", i)
}

init { run P(3); run P(7)}
```

Figure 5: `id-print.pml`

In the program `id-print.pml`, the proctype declaration for P says that it takes a parameter of type `int`. Very like a procedure or function taking parameters, this allows, in the last line of the program, `init` to run two instances of P, one with the parameter 3 and one with the parameter 7. So we get P(3) and P(7) active concurrently.

In the `printf` command, the "%d" inside the string shows where the value of the variable `i`, listed after the string, is to appear. The "d" says the integer `i` is to be printed out in decimal format. The syntax comes from C.

Try running this program repeatedly. It will sometimes print "P(3)" followed by "P(7)", and sometimes "P(7)" followed by "P(3)". Can you guess why?

2.4 Interleaving and Non-determinism

In `id-print.pml` (5), the concurrent processes P(3) and P(7) had only one command each to execute, so we had two possible outputs. What if each had two commands?

```
/* Shared printing of a sentence, active processes.*/

active proctype P() {
    printf("John\n");
    printf("read\n")
}

active proctype Q() {
    printf("the\n");
    printf("book\n")
}
```

Figure 6: `shared-sentence.pml`

The program `shared-sentence.pml` (6) uses a new keyword, `active`, which prefixed to a proctype definition means that an instance of that proctype will be active in the initial system state. Using `active` can often make `init` superfluous. (But processes that are instantiated through an active prefix cannot be passed arguments. This last is not quite the truth, but will do for now).

Run the program. You will see that the output sequences consist of all possible permutations of the words in "John read the book," with the restriction that "John" precedes "read", and "the" precedes "book." (Precedes, but not necessarily immediately). Do you understand why?

2.4.1 Scheduling: Active does not mean running

`init` in `id-print.pml` (5) makes first P(3) and then P(7) active. In `shared-sentence.pml` (6), both P and Q are active at the start. But *active* in concurrency parlance does *not* mean running; it only means *ready to run*. When an active process actually runs is up to a *scheduler*, a run-time support entity that allocates CPU time to processes.

A fundamental assumption in concurrent programming is that there is no one-to-one mapping between CPUs and processes. There might be exactly as many CPUs as processes, or fewer or more. The CPUs may be shared with other users and other programs. We do not know when an active process is actually running, and there is no way to find out from within the program.

2.4.2 Non-determinism

Because of scheduling (and bus arbitrators, and delays for I/O, for accessing busy resources, etc.), concurrent programs are necessarily *non-deterministic*. A program can behave differently on different runs, even with the same input.

A little reflection reveals one consequence: debugging will not work as it does for sequential programs. We might spot the symptoms of a bug, but we cannot catch the bug by break points and re-running because there is no guarantee that a bug will reappear when we want it to. We will see later how to debug concurrent programs.

Notice that the non-determinism appeared even though there were no shared variables, which implies nothing very interesting can happen, in one sense. Each process simply does its own work, and its results are not affected by the others. But the processes interact via *shared resources* like CPUs, buses, I/O devices, web resources, etc., and this affects how fast the processes actually run.

2.4.3 Assume nothing about how fast a process runs

Suppose you and several other students are reading a notice board (but not writing on it or otherwise modifying it). Your work remains unaffected by the number of other students, but how long it may take you to do your work cannot be predicted—if there is a huge crowd around the notice board, it will likely take you longer to get done with the step "check the notice board".

So the first rule in building a concurrent program is to assume nothing about the speed of any process: how long it will take to do anything. If you have an appointment at 11 o'clock, and you don't know how long it will take, you don't arrange to meet a friend at 12 o'clock. Instead, you say, "I'll call you when I'm done." We return to such programmed *synchronisation* after first studying a higher level construct.

3 Atomic actions

```
/* Shared printing of a sentence, active atomic processes.*/

active proctype P() {
    atomic{
    printf("John\n");
    printf("read\n")
    }
}

active proctype Q() {
    atomic{
    printf("the\n");
    printf("book\n")
    }
}
```

Figure 7: `shared-sentence-atomic.pml`

Suppose we say that "John read the book" (... that I gave him) and "the book John read" (... is very famous) are both acceptable phrases, but not interleavings such as "John the book read". How can we modify the program `shared-sentence.pml` (6) to produce only the two phrases we like, but not the ungrammatical ones?

A partial solution is to observe that "the" and "book" must not be separated, and that similarly "John read" should always turn up as one unit. There is a construct to ensure just such packagings[3]. A sequence of commands enclosed in curly braces and preceded by the keyword `atomic` is executed as one indivisible unit: either the entire sequence is executed or none of it. The commands in the sequence cannot be interleaved with those of another processes.

The program `shared-sentence-atomic.pml` (7) uses atomic actions to ensure it will only produce one of the two acceptable phrases.

[3]Of course we could package such complete phrases into one `printf` command, and rely on the atomicity of single commands, but that simply avoids the question of how to package together what might be independent commands in other contexts.

3.1 How big should atomic actions be?

Atomic actions remove certain unwanted interleavings, taming some unwanted non-determinism. What if they are made as big as possible? If the code for process P is just one atomic action, then P cannot be used in a program that needs interplay between processes.

This raises two issues. One is that while non-determinism (here manifesting as interleaving) can cause problems, it is not always bad! It arises from interaction, which is the whole point of concurrent programming. We will soon see programs that actually use interaction instead of just suffering from it. Non-interacting concurrent programs are simply independent.

The other is that concurrent processes interacting via say a database need to lock the database up for as short time as possible, so atomic actions using the database should be kept short. How short can they be? This is an important question both theoretically and historically, as we shall see in the course.

4 Shared variables, guards and blocking

```
/* Increment a variable in two processes, to encode who went first. */
int n = 0;  int finished = 0;

proctype P(int i) {
    n = 10*n + i;
    finished++  /* Process terminates */
}

proctype Finish() {
    finished == 2;   /* Wait for termination */
    printf("n = %d\n", n)
}

init { run P(3); run P(7); run Finish()}
```

Figure 8: `id-count.pml`

The program `id-count.pml` (8) is like `id-print.pml` (5). It too runs two processes P(3) and P(7), and tells us which ran first, but gives out this information encoded: the output can be either 37 (if P(3) began first) or 73 (if P(7) began first).

The program `id-count.pml` is our first with *variables*! The *variable declarations* say that `n` and `finished` are both of type `int` and initialised to 0. Note that these variables are *global*. The encoded output is in `n` and `finished` *synchronises* the printing of `n`.

The command `finished == 0` looks like a boolean expression; in Promela, a boolean that constitutes the entire command is called a *guard*. Executing the guard B means, if the boolean B is true, then moving on to the next command, and if B is false, then *blocking* until B becomes true (which can only happen because of an action by another, unblocked, process). Blocking (also called *waiting* or *sleeping*) means the process will execute no commands.

So if Finish executes `finished == 0` when both P(3) and P(7) have terminated, it executes the `printf` command next. If either of P(3) or P(7) is still active, Finish blocks till they both terminate.

4.1 Synchronisation tames non-determinism

Running `id-count.pml` (8) produces the expected output, but if you comment out the first line of Finish (move the `/*` to the start of the line) and the output can now be not only 37 or 73, but also 0, 3 or 7. This is because Finish can now run at any time, after just one of P(3) or P(7) have run, or even before either. The non-determinism is greater.

Putting the guard in (uncomment the first line again) cuts out these unwanted runs. The synchronisation keeps the non-determinism to what we wanted.

4.2 More on blocked processes

In concurrent programming systems, the scheduler keeps track of process states. It only allocates CPU time to active processes, never to blocked or terminated ones. How the process state is stored and how the scheduler works are part of the run-time support of the implementation, and no concern of the concurrent programmer.

A guard B can also be though of as `while not B do skip`, called a *busy wait*[4]. This may indeed be how guards are sometimes implemented (for example, if each process has a dedicated CPU). But the scheduler will see a busy wait as running code, and so will keep allocating CPU time to it. If there is no other CPU where another process can make B `true`, any time allocated to a busy waiting process is just wasted. Worse, if the scheduler runs runnable processes till they block or terminate[5], the busy waiting process will lock the whole system up.

5 Assertions

So far, we have used Spin to show that runs produce differing outputs for non-deterministic programs, and quite small amounts of non-determinism can produce a very large number of outputs, certainly too large to check them all manually. For example, take program `demo-assert.pml` (9), a slight extension of `id-count.pml` (8). It's easy to see that this program will produce $133579 \leq \texttt{n} \leq 975331$, but it's hard to get either 133579 or 975331

[4] If you are in a queue and you don't have a book or a phone or thoughts to keep you occupied, and are not just calmly switched off either, you end up busy-waiting, an enervating business.

[5] A *time-sharing* scheduler would, after a *time-slice*, give the CPU to another process.

by simply saying "Run" to Spin. It chooses the interleaving sequence, and there are 360 possible outputs (do you see why?). (It is possible to get Spin to make choose a particular path through the non-determinism using "Interactive"—try it with a small program like `id-print.pml` (5)).

Clearly, exhaustive testing of a concurrent program is not possible by literally watching every possible run of it. But Spin can do exhaustive testing (called *model checking*), by checking if a given property (called an *assertion*) holds for every run of a program. In `demo-assert.pml` (9), there are three assertions, one of them incorrect and commented out. Run the program as usual first. Next, ensure "Mode" is set to "Verify" and click

```
/* Assertion demo */
int n = 0;  int finished = 0;

proctype P(int i) {
    n = 10*n + i;
     finished++;  /* Process terminates */
}

proctype Finish() {
finished == 6;  /* Wait for termination */
printf("n = %d\n", n);
    /* assert(n > 133579) */
    assert(n\%2==1);
    assert(n >= 133579 && n <= 975331)
}

init {run P(3); run P(7); run P(5); run P(1); run P(3); run P(9);
       run Finish()}
```

Figure 9: `demo-assert.pml`

on "Verify". A few lines down in the input is a line like `State-vector 76 byte, depth reached 31, errors: 0`, which confirms that your two assertions hold for every run of the program. By the way, `&&` means logical "and", and `%` means modulo division.

Next, uncomment the incorrect assertion, turn on the button "Run generated trail" and "Statements", and click on "Verify" again. This time, you are told right at the top that an assertion has been violated, and which one. Further down, you are shown an execution sequence that leads to `n=133579`, violating the assertion.

Oddly enough, you learn more from an assertion that fails, since you get an example run that violates it. Assertions that succeed give you the satisfaction of confirming that they hold, but you don't know why.

6 Examples

6.1 Control flow

That we have come this far using only sequencing (within a single process) is a demonstration that concurrency is a powerful control flow primitive. But we will need `if` and loops for more meaningful examples. Both constructs in Promela use guards.

In `max.pml` (11), the `proctype c` uses an `if` construct. This has, between the keywords `if` and `fi`, any number of execution sequences, each preceded by a double colon. Immediately after the `::` comes a guard, followed by the `->` arrow. If one of the guards in an `if` statement is true, the sequence following the `->` is executed. If more than one guard is true, one of the corresponding sequences is selected non-deterministically. If all guards are false, the process will block until one of them becomes true. The `else` is only chosen if all the other guards are false.

In `count.pml` (10), the `proctype P` uses a loop. This looks very like the `if` construct, except that it is bounded by `do` and `od`. Only one guard can be selected at a time. Afterits execution sequence completes, the `do` loops. To terminate the repetition, use a `break` statement, which transfers control to the instruction immediately following the `od`.

```
/* Increment a variable in two processes. Final value can be two!! */
#define TIMES 3
byte n = 0;  byte finished = 0;

active [2] proctype P() {
     byte i = 1;  byte temp;
     do
     :: ( i > TIMES ) -> break
     :: else ->
           temp = n;
           n = temp + 1;
           i++
     od;
     finished++;  /* Process terminates */
}

active proctype Finish() {
     finished == 2;  /* Wait for termination */
     printf("n = %d\n", n);
     assert (n > 3);  /* Assert can't be 2 */
}
```

Figure 10: `count.pml`

6.2 Gate count

There are two entries to a park, and as a person comes in through either gate, we increment the global variable `n` in the program `count.pml`(10). Unfortunately, this is not done by a single (atomic) instruction `n = n+1`, but by the sequence `temp = n; n = temp+1` (which reflects what often happens at machine level). If three people enter through each gate, the final value of `n` can lie between 2 and 6. As exercises, figure out how each value can arise. As further exercises, figure out assert statements to confirm various properties, and to produce not only counterexamples for final values 1 and 0, but to show you how each possible value arises. What happens if you use `n++` instead of `temp = n; n = temp+1`? What if you use `atomic{temp = n; n = temp+1}`?

6.3 The largest of 5 given positive integers

```
int p = 35; int q = 57; int r = 23; int s = 17; int t = 87;
int max = 0;  int finished = 0;

proctype c(int i) {
    atomic {if
            :: (i > max) -> max = i; printf("%d\n", i)
            :: else -> skip;
            fi};
    finished++;  /* Process terminates */}

proctype Finish() {
    finished == 5;  /* Wait for termination */
    assert(max >= p && max >= q && max >= r && max >= s && max >= t);
    printf("max = %d\n", max);}

init {atomic{run c(p); run c(q); run c(r); run c(s); run c(t)};
      run Finish()}
```

Figure 11: `max.pml`

We have so far not actually used the power of concurrency, and we have treated non-determinism as if it were only a bad thing to be tamed. The example `max.pml` (11) sets this right. It is a program to print out the largest of 5 given positive integers.

A sequential version of max would have to specify the order in which the comparisons are made (say p, q, r, s, t); also, it would always update `max` thrice (to 35, 57 and 87). Our `max.pml` might update `max` 5 times or just once. And we don't care about the order of comparisons. Indeed, the need to specify things we don't care about is one of the major reasons why sequential programming is so tedious and error prone.

Experiment with running and verifying the program. What happens if you comment out the `atomic` in `c`? Or move it so that only `max = i` and `printf` are within the `atomic` but not the comparison `i > max`? (If you comment out the `atomic` in `init`, the program will still be verifiable, but because `c` is so small, you will find that the processes spawned earlier tend to finish before the later ones get started, so it's not as much fun).

6.4 Primes

The program `hunt.pml` (12) prints out the primes in ascending order. It does so by letting every integer from 2 to MAX mark all its multiples as such. When they are all done, the unmarked integers are the primes. The program retains trace prints to show you the non-determinism in the work; as in `max.pml` (11), we do not specify in what order the marking is to proceed.

7 Ways to study concurrent programming

7.1 Radical

We could note how concurrent processes behave in real life, and then design programming constructs to produce these behaviours. So we can now simulate or model the real-life situation we began with. This is a modern approach made possible by 60 years of experience with concurrent programming, and indeed we will use this approach sometimes, as we go along. It will give us radical new ways to program, taking concurrency as basic, not as an add-on to sequential programming. Sadly, such matters are outside the main scope of the course, so we can only give you a taste.

7.2 Historical

Alternatively, we can begin with the concrete situation that faced 1950's programmers—how to allow the card reader (CDR) and line printer (LPT) to work largely at the same time as the CPU, instead of having only one running at a time, which is the obvious way to program. This leads us, via interrupts and machine level programming, up through interrupts and semaphores to higher level abstractions.

7.3 Conventional

References

[1] Promela - Wikipedia page. https://en.wikipedia.org/wiki/Promela. Accessed: 2016-08-24.

[2] Rob Gerth. Concise Promela reference. http://spinroot.com/spin/Man/Quick.html, 1997. Accessed: 2016-08-24.

[3] Mordechai Ben-Ari. Principles of concurrent and distributed programming, 2006.

```
#define MAX 30
int p [MAX+1];  /* index goes from 0 to MAX.  Will use only 2..MAX  */
int finished = 0;

proctype c(int d) {
    int n = 2*d;
    do
    :: n>MAX | p[d]!=1 -> break;
    :: else -> if
               :: p[n]!=1 -> skip;
               :: else -> p[n]=d; printf("%d divides %d\n", d, n);
               fi;
               n=n+d;
    od;
    finished++
}

proctype Finish() {
    int n = 2;
    finished == MAX-1;
    do
    :: n>MAX -> break;
    :: else -> if
               :: p[n]==1 -> printf("%d\n",n);
               :: else -> skip
               fi;
               n++
    od
}

init {
    int n=2;
    do
    :: n>MAX -> break;
    :: else -> p[n] = 1; n++;
    od;
    atomic{
    n=2;
    do
    :: n>MAX -> break;
    :: else -> run c(n); n++
    od
    };
    run Finish()
}
```


Figure 12: `hunt.pml`

[4] Ayu Yusoff. Lecture Slides: Tutorial for Promela, Heriot-Watt University. http://www.macs.hw.ac.uk/~air/dsp-spin/PROMELA.swf, 2014. Accessed: 2016-08-24.

[5] Wolfgang Ahrendt. Lecture Slides: Introduction to Promela - in the course Software Engineering using Formal Methods, Chalmers University. www.cse.chalmers.se/edu/year/2014/course/TDA293/Lectures/Files/PROMELAIntroductionPS.pdf, 2014. Accessed: 2016-08-24.

[6] Andrew Ireland. Lecture Slides: Promela I - in the course Distributed Systems Programming, Heriot-Watt University. www.macs.hw.ac.uk/~air/dsp-spin/lectures/lec-3-promela-1.pdf, 2014. Accessed: 2016-08-24.

[7] Andrew Ireland. Lecture Slides: Promela II - in the course Distributed Systems Programming, Heriot-Watt University. www.macs.hw.ac.uk/~air/dsp-spin/lectures/lec-3-promela-2.pdf, 2014. Accessed: 2016-08-24.

[8] Creators admit Unix, C hoax. https://www-users.cs.york.ac.uk/susan/joke/c.htm. Accessed: 2016-08-24.