Complete Program

```
proctype P() {
    int x = 15, y = 20;
    int a = x, b = y;

    do
    :: a > b   ->   a = a - b
    :: b > a   ->   b = b - a
    :: a == b  ->   break
    od
    printf("GCD(%d, %d) = %d\n", x, y, a);
}
```

Program Declaration

- "`proctype P()`" declares no-argument program `P`

- Can include arguments:

```
proctype P(int x, int y) {
    int a = x, b = y;

    etc.
```

Getting Started, I

- Can start processes using "`run`" operator:
`run P(15, 20)`

- Also, can declare process with
"`active proctype`"

- Adding "`active`" means "declare AND RUN this program"

- To start two processes executing same code, use:
`active [2] proctype P(int x, int y)`

Getting Started, II

- Can create an initial process that runs before any of the "`proctype`" processes

- This process must be named `init`

Example, I

One way to do Part 1 of the assignment:

```
proctype increment1toN(byte N) {
    ... logic for this process ...
}


init {
    ... other statements ...
    run increment1toN(5);
    run increment1toN(5);
    ... other statements ...
}
```


Example, II

Another way to do Part 1 of the assignment:

```
active [2] proctype increment1toN() {
    N = ... initial value ...

    ... logic for this process ...
}
```

Predefined Variables

- "_pid" is process ID
- "_nr_pr" is number of active processes
- Examples:

```
printf("process %d: n goes from %d to %d\n", _pid, temp, n);

if
:: _nr_pr == 1 -> printf("at end n = %d\n", n);
fi
```


Blocking Statements, I

- Concurrent programs must often wait for some event

- Possible to guard ANY statement

- This:

```
_nr_pr == 1 -> printf("at end n = %d\n", n);
```

is the same as:

```
if
:: _nr_pr == 1 -> printf("at end n = %d\n", n);
fi
```

Blocking Statements, II

- "->" arrow is just syntactic sugar

- Can write expression by itself; if it doesn't evaluate to non-zero then program will block

- This:

```
_nr_pr == 1;
printf("at end n = %d\n", n);
```

is the same as:

```
_nr_pr == 1 -> printf("at end n = %d\n", n);
```

Atomicity, I

- Individual Promela statements are atomic

- WARNING!!! In Promela, expressions are statements too (hence expressions are atomic)

- Example – here, division by zero IS possible:

```
if
::  a != 0 -> c = b / a
::  else    -> c = b
fi
```

- Reason: expression "`a != 0`" is atomic as is statement "`c = b / a`"

Atomicity, II

- It is tempting to regard the entirety of "`a != 0 -> c = b / a`" as atomic

- But it consists of TWO atomic parts, "`a != 0`" and "`c = b / a`

- Remember that this:

```
a != 0 -> c = b / a
```

could be written as:

```
a != 0;     /* may block */
c = b / a
```

- The latter more obviously contains two atomic parts

Atomicity, III

- To group statements together atomically use `atomic`

- Example:

```
atomic {
    a != 0;     /* may block */
    c = b / a
}
```

- As in this example, first statement of an `atomic` block may be a (potentially) blocking expression

Atomic & Run

- `run` only STARTS a concurrent process

- `atomic` prevents execution of any other actions besides those in its body

- Therefore, to start a group of processes that should run concurrently:

```
atomic {
    run P1(...);
    run P2(...);
        ...
    run PN(...);
}
```

- At conclusion of `atomic` block: all processes have been started but none is yet running

Assert

- Syntax: "`assert(expression)`"

- If `expression` evaluates to 0, simulation stops

- If assertion fails then model doesn't satisfy property denoted by "`expression`"

Example

- Part 1 of the assignment says "Show that the final value of `n` may be outside the range 5-10. Write a Promela model that includes an assertion that is violated if `n` is out of range when the two processes are finished"

- Therefore, write:

```
byte n = 0;   /* make n global */

... model logic ...

... wait for model processes to finish ...
assert(5 <= n && n <= 10);
```

Variable Size

- Use smallest integer variable that will fit the need

- E.g., for integers known to be small use "`byte`" (8 bits) instead of "`int`" (32 bits)

- Reason: in "verification" mode Erigone simulates all possible values of variable

I/O

- `printf` output appears only during debugging, not during verification

- Why no scanf?

Holzmann writes:

"there is no matching `scanf` statement to read information from the input. The reason is that we want verification models to be *closed* to their environment. A model must always contain *all* the information that could possibly be needed to verify its properties. It would be rather clumsy, for example, if the model checker would have to be stopped dead in its tracks each time it needed to read information from the user's keyboard."

Running Erigone

- "`.pml`" is Promela file extension
- Run as "`erigone [arguments] file`"
- Erigone has 4 execution modes:
  1. Random simulation – at any choice point, Erigone makes random selection
  2. Interactive – at any choice point, user makes selection
  3. Verification – Erigone checks all possible paths
  4. Guided – after failed verification, re-run particular path that failed

Simulation Mode

- The default

- Run as "`erigone file`" (no arguments; Erigone will add "`.pml`" extension if necessary)

- Only one random path tested

- Useful mostly for model debugging

Interactive Mode

- Run as "`erigone -i file`"

- Useful only for deep investigation of specific case

Verification Mode

- Run as "`erigone -s file`"

- If run long enough, will check all reachable states

- Runs until first of:

  - Successful finish
  - Unsuccessful finish (e.g., divide by zero)
  - Failed assertion
  - Step limit reached (default is 10K)

- To run with higher step limit:
"`erigone -s -ltN file`"
where `N` is number of thousands

- E.g., "`-lt1000`" means run up to 1,000,000 steps

Guided Mode

- Verification that ends with assertion failure leaves behind a "trail file" ("`.trl`" extension)

- Trail file documents (in machine readable form) path that led to failure

- Now that failing case is known, use trail file to execute *guided* simulation, typically displaying all states:
"`erigone -g -dm file`"

- "`g`" argument means guided mode;
"`dm`" argument means display all states

- Erigone will be guided by "`file.trl`"

Erigone Compiler Messages

- Model checking is leading-edge technology

- Therefore: users are assumed to be savvy

- Therefore: error messages are often not too helpful

- Parser just reports its state at moment of failure – it is assumed that you can figure out what went wrong

Example

- Example error message:

```
exception=compiler_declarations.compilation_error,
message=variable or process declaration expected:8:2:},
```

- Bad declaration was on line 8 – the first number of the error message is the line number

- Go to that line, inspect it & lines immediately before it