Chalmers | GÖTEBORGS UNIVERSITET
K. V. S. Prasad, Dept. of Computer Science and Engineering

Notation for Concurrent Programming TDA383/DIT390

October 19, 2016

This is a copy of the Appendix to the examination question paper. As in the exam, the text of the Appendix starts on a new page. It summarises the notation used in the question paper.

You can use this notation in your answers, but you can also use Java, Erlang or Promela **provided you give your constructs the same semantics as the question requires.** The exact syntax of the programming notations you use is not so important as long as the graders can understand the intended meaning. If you are unsure, explain your notation.

A Appendix: Pseudo-code, LTL and Linda notations

A.1 SUMMARY OF BEN-ARI'S PSEUDO-CODE NOTATION

1. Global variables are declared centred at the top of the program.

   Data declarations are of the form `integer i := 1` or `boolean b := true`, giving type, variable name, and initial value, if any. Assignment is written `:=` also in executable statements. Arrays are declared giving the element type, the index range, the name of the array and the initial values. E.g., `integer array [1..n] counts := [0, ..., 0]`.

2. The statements of the processes are often in columns headed by the names of the process. If several processes `p(i)` have the same code, parameterised by `i`, they are given in one column. Indentation indicates sub-statements of compound statements.

3. All commands are numbered, but not control flow directions such as `loop forever` and `repeat`. If a continuation line is needed, it is left un-numbered or numbered by an underscore `p_`. Numbered statements are atomic. Assignments and expression evaluations are atomic.

4. The statement `await b` is equivalent to either `block until C` or to `while not b do nothing`, a *busy wait*. Which interpretation is meant will be pointed out in any question using `await`. Under the first interpretation, the system may *deadlock* (everyone is blocked); under the second, the system may *livelock* (everyone busy-waits). The only difference is in CPU-cycles. Both states show mutual impediment to progress, or circular waiting.

5. For channels, `ch => x` means the value of the message received from the channel `ch` is assigned to the variable `x`. and `ch <= x` means that the value of the variable `x` is sent on the channel `ch`.

6. A scenario is a list of the labels of the statements in the order of execution. With synchronous channels, sender and receiver act together, so show both statements as a pair being a single move in the scenario.

EXTENSION OF BEN-ARI'S PSEUDO-CODE NOTATION

1. You can explicitly declare processes by a line of the kind `proctype p(integer i)` giving the name of the process and its parameters. Explicit commands like `run p(5); run p(6)` are used to run processes, in this case to start process `p` with parameter 5, and then start another instance of `p` with parameter 6. An explicit `init` process starts the program.

   These extensions give new expressive power. The `run` command means the number of processes in a program can change during execution. Processes can pass channels as parameters. This allows the network of channels between processes to change dynamically.

2. We extend Ben-Ari's notation for channels, allowing `channel capacity(n) of boolean forks[5]`. This declares an array of channels, `fork[0]` through `fork[4]`, each a channel of buffer capacity `n`, carrying boolean values. So `n=0` specifies a synchronous channel, and `n=5` specifies an asynchronous channel with buffering capacity 5. For theoretical discussion, we can also permit n to be `infinite`. The capacity declaration `capacity(0)` can be dropped, (i.e. in that case, assume `n=0` and therefore synchrony).

3. Input commands are allowed to attach a `timeout` clause, a sequence of statements that must end with `goto` *label*. If the channel is empty when an input command runs, the process executes the code between `timeout` and `goto`, and then jumps to the statement *label*.

A.2 LOGIC

1. The symbols used here for the operators of propositional logic are: $\neg$ for "not", $\vee$ for "or", $\wedge$ for "and", and $\rightarrow$ for "implies", while p iff q (i.e., p if and only if q) is a convenient abbreviation for $(p \rightarrow q) \wedge (q \rightarrow p)$. These have the obvious meanings, but two differ from what might be your interpretation of the name. Note that $p \vee q$ ("p or q") is false iff both p and q are false. This is an "inclusive or", so $p \vee q$ is true if both p and q are true. Also, note that $p \rightarrow q$ ("p implies q") is false iff p is true and q is false. In particular, this means $p \rightarrow q$ is true if p is false.

2. A proposition such as q_2 (process q is at label q_2) is true of a state s iff process q is at q_2 in s.

3. We use Linear Temporal Logic (LTL), which is propositional logic with two added operators, $\Box$ and $\Diamond$. A formula ϕ of LTL holds for state s (or, s *satisfies* ϕ, written $s \models \phi$) if every path from s satisfies ϕ.

   A *path* is a possible future of the system, a possibly infinite sequence of states, each reachable from the previous state in the path.

   A path π satisfies $\Box\phi$ (written $\pi \models \Box\phi$) if ϕ is true of the first state of π, and for all subsequent states in π. The path π satisfies $\Diamond\phi$ (written $\pi \models \Diamond\phi$) if ϕ is true of some state in π.

   Note that $\Box$ and $\Diamond$ are duals:

$$\Box\phi \equiv \neg\Diamond\neg\phi \qquad \text{and} \qquad \Diamond\phi \equiv \neg\Box\neg\phi.$$

A.3 LINDA

In Linda programs, processes communicate via *tuples* posted in a *space*. The first element of a tuple is often a constant string, saying what kind of tuple it is. Processes interact with the space through three kinds of atomic actions.

`post(t)` Here `t` is a tuple $\langle x_1, x_2, ..\rangle$, where the x_i are constants or values of variables. `post(t)` posts `t` in the space, and unblocks an arbitrary process among those waiting for a tuple of this pattern.

`remove`$(x_1, x_2, ..)$ Here the parameters must be variables or constants. The command `remove`$(x_1, x_2, ..)$ removes a tuple $\langle x_1, x_2, ..\rangle$ that matches the pattern of the parameters, and assigns the tuple values to the variable parameters. If no matching tuple exists, the process is blocked. If there are several matching tuples, an arbitrary one is removed.

`read`$(x_1, x_2, ..)$ Like `remove`$(x_1, x_2, ..)$, but leaves the tuple in the space.

We allow two extensions of the input constructs `remove` and `read`:

1. `remove` and `read` actions can use `eval(n)` to mean the value of variable `n`. Suppose n=13. Then the pattern `read('start', n)` will match the tuple `('start', 14)`, resetting the value of `n` to 14, whereas `read('start', eval(n))` will only match the tuple `('start', 13)`.

2. To `remove` and `read` actions can be attached a `timeout` clause, a sequence of statements that must end with `goto` *label*. If there are no tuples that match the given pattern, the process executes the code between `timeout` and `goto`, and then jumps to the statement *label*.

——-END of APPENDIX——