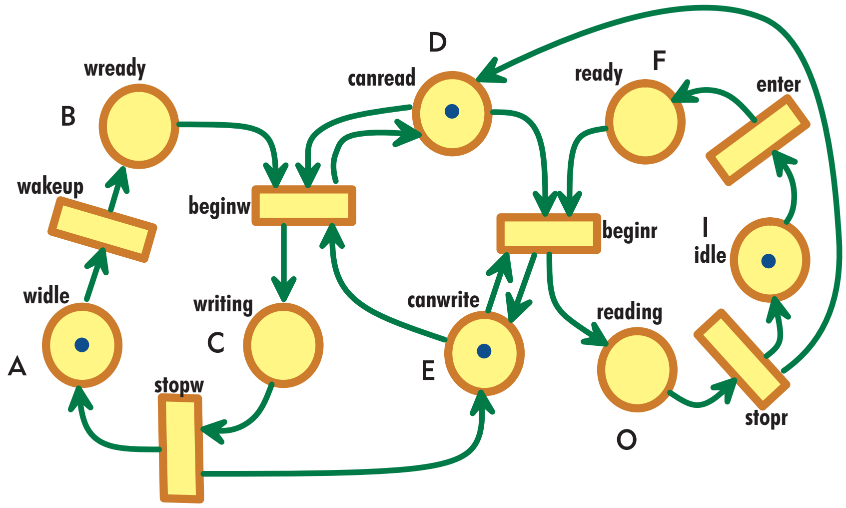
How to convert a Petri net to a program

As some of you may have noticed, Petri nets are at their own level of abstraction and implementing a given Petri net specification is an art of its own. A lot depends on what the intended model of concurrency is. Synchronous communication is closer to Petri nets in some respects; asynchronous communication may require some further work.

Below, we will outline an implementation of the state testing mutex algorithm using the operations `send` and `recv`, familiar from TCP/IP socket based communication.



We make a distinction between testing, getting, and putting a token from a place. There are two processes, the `writer` and the `reader`, and there can only be one instance of each running at any time. (There are other kinds of solutions to the readers/writers problem which make it possible to have several instances of each process.)

The `writer` process has

- 3 places which are local: `widle`, `wready`, and `writing`. They are modelled as integer variables.
- Two places shared with the `reader` process: `canread` and `canwrite`. They are modelled as integer variables but putting/getting a token to/from each of them requires communication with the `reader`.

The `reader` process has

- 3 places which are local: `idle`, `ready`, and `reading`. They are modelled as integer variables.
- Two places shared with the `writer` process: `canread` and `canwrite`. They are modelled as integer variables but putting/getting a token to/from each of them requires communication with the `writer`.

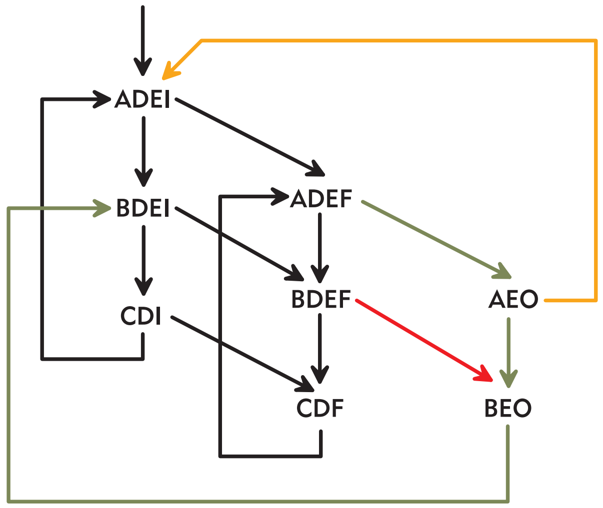
It seems the `writer` process only tests the token in `canread` and the `reader` process only tests the token in `canwrite`. The operations `send` and `recv` are based on asynchronous communication but will work with synchronous communication, too. (In the latter case, acknowledgements are unnecessary.)

```
process writer
      var widle: integer := 1;
          wready: integer := 0;
          writing: integer := 0;
          canread: integer := 1;
          canwrite: integer := 1;
          enter_crit: boolean := FALSE;

   procedure check_tokens (canwrite_op: canwrite_id)
      var place: place_id;
          oper:  oper_id;     // recv operation

      if canwrite_op = Q_CANWRITE_GET
         send (Q_CANWRITE, Q_GET)
      elseif canwrite_op = Q_CANWRITE_PUT
         send (Q_CANWRITE, Q_PUT)
      endif
      while recv (place, oper) or (not canwrite_op = Q_CANWRITE_NONE)
         if place = Q_CANREAD and oper = Q_GET
            canread := canread - 1
            send (Q_CANREAD, Q_ACK)
            log (widle, wready, writing, canread, canwrite)
         else if place = Q_CANREAD and oper = Q_PUT
            canread := canread + 1
            send (Q_CANREAD, Q_ACK)
            log (widle, wready, writing, canread, canwrite)
         else if place = Q_CANWRITE and oper = Q_ACK
            canwrite_op := Q_CANWRITE_NONE
         endif
      endwhile
```

```
   begin // writer process
   ...
   while true
      log (widle, wready, writing, canread, canwrite)
      enter_crit := calculate_a_little ()
      check_tokens (CANWRITE_NONE)
      if widle and enter_crit
         widle := 0
         wready := 1
         log (widle, wready, writing, canread, canwrite)
      endif
      if wready and canread and canwrite
         check_tokens (CANWRITE_GET)
         if canread
            canwrite := 0
            wready := 0
            writing := 1
            log (widle, wready, writing, canread, canwrite)
            //
            // now in critical section (writing)
            //
            critical_section (Q_WRITE);
            //
            // now exiting critical section (writing)
            //
            canwrite := 1
            writing := 0
            idle := 1
            enter_crit := FALSE
         endif
         check_tokens (CANWRITE_PUT)
         log (widle, wready, writing, canread, canwrite)
      endif
   endwhile
endprocess
```

The reader process is implemented in a similar manner. Each of the processes will produce a log which can then be compared with the reachability graph of the Petri net. The observant reader may already have noticed that the `writer` process only produces partial markings as it only 'sees' five of the places. Therefore we will be satisfied if a sequence of matching, reachable markings is found. Otherwise, the implementation has to be deemed faulty.

Sometimes your task is only to implement one part of the specification. In that case you might only want to focus on 'your' (visible) transitions (instead of markings) and regard the other transitions as invisible. Checking if a trace of fired visible transitions can be found in the reachability graph is fairly simple.



In the above reachability graph, the writer process has to find marking sequences such as ADE*-BDE*-CD*-ADE*, ADE*-BDE*-BE*-BDE*, ADE*-AE*-ADE*, etc.

Sometimes both the implementation and Petri net have deadlocks. The only acceptable case is when their deadlocks coincide.

The state testing mutex algorithm is best envisioned as a means to guarantee mutual exclusion in the presence of threads (or, in general, processes sharing RAM memory). Then a TCP/IP style `send`/`recv` would by no means be a good idea. Semaphores or shared variables might be a more efficient way of communication. The critical resource could be e.g. a file. (Mutual exclusion could also be achieved by means of file locking but it is very likely it is less efficient than a shared file when mutex is guaranteed with our Petri net approach.)

Can you find any mistakes in the implementation? Do you think the implementation makes sense?