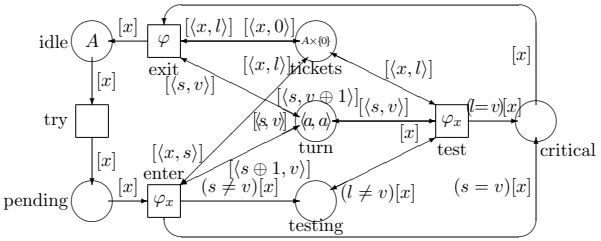
Let us examine the following algorithm:

> **TicketME algorithm (informal):** A shared variable holds a pair $\langle next, granted \rangle$ of values in $\{1, \ldots, n\}$, initially $\langle 1, 1 \rangle$. The *next* component represents the next "ticket" into the critical section, while the *granted* component represents the last "ticket" that has been granted permission to enter the critical section. When a process enters the trying section, it "takes a ticket," that is, it copies and increments the *next* component modulo n. When the ticket of a process is equal to the *granted* component, it goes to the critical region. When process exits the critical section, it increments the *granted* component modulo n.

Nancy A. Lynch: Distributed Algorithms, 1996, ISBN 1-55860-348-4



The Maria description of the net is:

```
// Owner of a lock (0=free, 1..n=some process 1..n)
typedef unsigned (0..2) owner_t;
// The number of a process (1..>owner_t)
typedef owner_t (1..) client_t;
typedef struct {
  client_t next;
  client_t granted;
} turn_t;
typedef struct {
  client_t client;
  owner_t owner;
} ticket_t;
place idle (0..#client_t) client_t: client_t client: client;
place pending (0..#client_t) client_t;
place testing (0..#client_t) client_t;
place critical (0..1) client_t;
place turn (1) turn_t: <turn_t;
place tickets (#client_t) ticket_t: client_t client: { client, 0 };

trans try
in { place idle: client; }
out { place pending: client; };
trans enter
in {
  place pending: client;
  place turn: { next, granted };
  place tickets: { client, t };
}
out {
  place testing: (next != granted)#client;
  place critical: (next == granted)#client;
  place turn: { +next, granted };
  place tickets: { client, next };
};
trans test
in {
  place testing: client;
  place turn: { next, granted };
  place tickets: { client, t };
}
out {
  place testing: (t == 0 || is client_t t != granted)#client;
  place critical: (t > 0 && is client_t t == granted)#client;
  place turn: { next, granted };
  place tickets: { client, t };
};
trans exit
in {
  place critical: client;
  place turn: { next, granted };
  place tickets: { client, t };
}
out {
  place idle: client;
  place turn: { next, +granted };
  place tickets: { client, 0 };
};

reject (1 subset place critical) && (2 subset place critical);
reject (1 subset place critical);

#ifdef FAIR
// A fairness assumption for transition exit
weakly_fair trans exit;
// A fairness assumption for transitions enter and test
weakly_fair client_t c:
(trans enter: client == c) || (trans test: client == c);
#endif // FAIR
```

We model check the following properties with Maria:

1. Two processes will never be in the critical section at the same time
$$\Box\neg(\text{critical}(1) \wedge \text{critical}(2))$$
2. Process 1 never gets to enter the critical section
$$\Box\neg\text{critical}(1)$$
3. Process 1 is able enter the critical section
$$\Box(\text{pending}(1) \rightarrow \Diamond\text{critical}(1))$$
4. Process 2 can exit the critical section
$$\Box(\text{critical}(2) \rightarrow \Diamond\text{idle}(2))$$

The properties will be checked by using a technique called *model checking*. We describe the properties in temporal logic, here LTL, and check the properties with Maria. The model checker will provide us with a counterexample if the property does not hold. The technique used in Maria is based on Büchi automata. Büchi automata are automata on infinite words. Both the negation of the property and the reachability graph of the system are interpreted as Büchi automata, and then their product is checked. If the language accepted by the product automaton is not empty, we know that the property does not hold. In practice, the checking is done by searching for an accepting run of the automaton, and if one is found, it is then displayed as a counterexample.

We load the net description to Maria with command `maria -m ticket.pn`
The command will not perform reachability analysis to the net. It will only load the net description. The net description can also be loaded in Maria with command `@0$model "ticket"`

Maria uses a separate program to translate LTL formulae to Büchi automata. Before the formula can be checked, Maria must be told the used translator with command
`@0$translator "lbt"`
Lbt is here the name of the used translator. The argument of the command `translator` is the name of the translator executable. The translator can also be loaded with the command line option `-p "lbt"`.

When the net description and the LTL-Büchi translator have been loaded, the LTL formulae can be checked by writing the formula at the Maria prompt, for example:
`@0$[]<>(2 subset place P)`
Maria will then perform on-the-fly model checking, generating only the part of the reachability graph needed for generating the counterexample. If no counterexample is found, the whole reachability graph will be generated. By adding a command `visual` in front of the formula, the possible counterexample will be displayed graphically.

The properties 1 and 2 can be checked by using Maria `reject` construct (see the attached net description). The `reject`ϕ is equivalent to LTL formula $\Box\neg\phi$, but the property is checked without the additional overhead of constructing the Büchi automaton and synchronizing it with the reachability graph. For properties 3 and 4, we must however use the LTL model checker.

When examining the counterexamples 3 and 4, we notice that they are not really sensible. That is, in a real system the sort of behavior shown in the counterexamples would probably not exist. The executions of the system are not *fair*. Maria has built-in fairness assumptions for model checking. The necessary assumptions have already been included in the model, and we can take them in use with switch `-DFAIR` and check the properties 3 and 4 again. This time, only fair executions are taken into account when searching for counterexamples.

The command `weakly_fair` corresponds to the fairness assumption $\Diamond\Box e \rightarrow \Box\Diamond f$ and `strongly_fair` the assumption $\Box\Diamond e \rightarrow \Box\Diamond f$. If a fairness assumption is defined for a group of transitions, it requires that if one transition of the group is enabled infinitely or infinitely often, some of the transitions of the group must be fired.