

Introduction to formal methods

August 26, 2008

The system

To formalize verification, we need to formalize the system, the specification language in which the properties are expressed, and the verification. Formalization entails syntax, i.e., notation, and semantics, i.e., meaning.

We start with the system. For a system's syntax there are many options, like a program, a number of parallel processes or even (a description of) a piece of hardware. For each of these forms of syntax then a semantics is defined that yields for a specific system what the behavior.

For our first investigations of verification, it is the behavior that is most important. So for now, we do not provide a concrete syntax, but leave this intuitive and implicit, but provide only a semantic model.

0.1 Semantic model

In the model we represent the various states that a system can be in during its processing. Also for now, we take a very simplistic view of what can be expressed in a state, namely just that some simple properties like being or not being at some location, using or not using some resource, are or are not the case. I.e., in a state propositional variables have a value *tt* (true) or *ff* (false). We also represent how the system can traverse the state-space, i.e., from which states it can go to other states. This leads to the following, now formal, model.

Definition 0.1.1 A *model* is a triple $M = (W, R, V)$.

1. W is a non-empty set of states (worlds).
2. $R \subseteq W \times W$ is a binary (accessibility) relation on W .
3. Var a set of variables.
4. V a valuation function $V : W \times Var \longrightarrow Values$.

Which states is (are) starting state(s) can either be incorporated as an extension of the model, or be coded in the variables.

From the informal syntax of the system, we obtain such formal model, again in an informal manner (!) by, roughly, imagining all states and transitions the system can go through. This can lead to finite but also to infinite models.

Note, that concurrency is not represented. This is intentional: the logics we will consider represent concurrent transitions by interleaving, i.e., non-deterministic sequential execution. A more detailed notion of program than the logics can handle would not be useful.

0.2 Example

As a running example we use a simplification of Peterson's algorithm [1983]. To aid the intuition, the algorithm is first presented informally as a picture of two parallel processes P_1 and P_2 , in Figure 1.

Location names, e.g., n_1 for the *non-critical section* in P_1 , are used as propositions that are true if and only if control in a process resides at such location. On the edges guards, such as $n_2 \vee (t_2 \wedge s_2)$, now interpreted as a proposition, and assignments appear, the latter being indicated by ":=". A guard indicates the enabledness of a transition, an assignment its effect on the program variables. The starting state of the algorithm with control residing in n_1 and n_2, s_1 equal to *true* and s_2 equal to *false* is denoted by $\{n_1 \wedge n_2 \wedge s_1\}$.

Note, that this representation indeed uses an only intuitive, informal, implicit syntax: things like guards or assignments on edges are not part of our formal model at all.

The main purpose of the algorithm is to ensure *mutual exclusion*, i.e., that the processes are never in their *critical sections* c_1 or c_2 simultaneously. A secondary aim is to ensure *eventual access*, i.e., that each process, if it desires to, eventually enters its critical section. A process expresses such desire by being in its *trying section*, t_1 or t_2 .

To see how tricky design of such a mutual exclusion algorithm is, consider as a first approximation an algorithm like the one presented in Figure 1 with, however, nothing written on the transitions except for guards $n_2 \vee t_2$ and $n_1 \vee t_1$ on the respective lower edges. A process can then, from its non-critical section, always freely enter its trying section. From there, it can progress to its critical section if and only if the other process is not in its critical section. It can then go back to its non-critical section. Obviously, this ensures mutual exclusion. A

little less obviously, eventual access is not guaranteed. There are two scenarios that always deny, say, process P_2 , access to its critical section.

1. P_1 never leaves its critical section.
2. P_1 leaves its critical section, but P_2 , although already in t_2 , is never quick enough in entering c_2 : P_1 passes through n_1 and t_1 and enters c_1 again before P_2 does so.

The model for this first approximation is as follows.

1. $W = \{s_1, s_2, \dots, s_8\}$.
2. $R = \{(s_1, s_2), (s_1, s_3), \dots, (s_7, s_3), (s_8, s_2)\}$.
3. $Var = \{n_1, t_1, c_1, n_2, t_2, c_2\}$
4. $V = ((s_1, n_1), tt), ((s_1, n_2), tt), \dots, ((s_8, t_1), tt), ((s_8, c_2), tt)$.

NB Such a model can much easier be understood representing it as a picture of nodes with propositions in them, and arrows. The nodes represent the states, the propositions represent the valuation (they contain the true propositions), the arrows represent the transitions.

Note, that this picture is quite different from the informal representation of the syntax of the algorithm: firstly, it is formal, as the representation in words shows, second, there is no more concurrency and there are no guards on the transitions anymore.

There is a simple and natural way to remedy the first possibility: just disallow a process to remain indefinitely in its critical section. The second possibility is a scheduling problem, and less easy to solve. Of course, one could postulate a scheduler taking care of this situation. However, this would require a very clever scheduler that would at least have to remember some details about each process' access history. This is not very realistic. Therefore, in the present exposition, the scheduler is kept simple and the burden is shifted to the algorithm.

The algorithm presented in Figure 1 uses shared signature propositions. Process P_1 has signature s_1 and P_2 has signature s_2 . When a process enters the trying section, the corresponding signature is set to *true*, the other one to *false*. If both processes are in their trying section simultaneously, the signature corresponding to the process that arrived there later has value *true*. As the guards on the outgoing edges show, in that case, quite naturally, the other process has priority of access. The next time round, the roles are reversed.

Initially, both processes are in their non-critical sections, n_1 and n_2 , and the value of s_1 is arbitrarily set to true; it is updated as soon as a process makes a move.

The only assumption necessary to ensure eventual access is that if some transitions are always enabled, than at least one of these is taken.

This is an important observation, because it leads to a simple interpretation of which sequences of states our formal model should be representing: all sequences of states that can be taking all possible the transitions.

Note, that if the example is slightly changed, namely having self-loops around the non-critical sections, this assumption is not enough any more to ensure eventual access. It is then also required that both processes will make progress if there is an always enabled transition.

If also self-loops around the critical sections are allowed, a further assumption is required, namely that no process stays indefinitely in t_1, c_1, t_2 or c_2 while the outgoing edge is enabled.

We will be investigating the following properties, the first two of which already have been introduced.

1. Mutual Exclusion.
2. Eventual Access.
3. Non-blocking - intuitively: a process can always enter its trying section. This is an interesting property, because a process does not need to request such. In terms of transitions it means that along every path through the model, from every state where n holds, there is a branch that leads to a state where t holds.

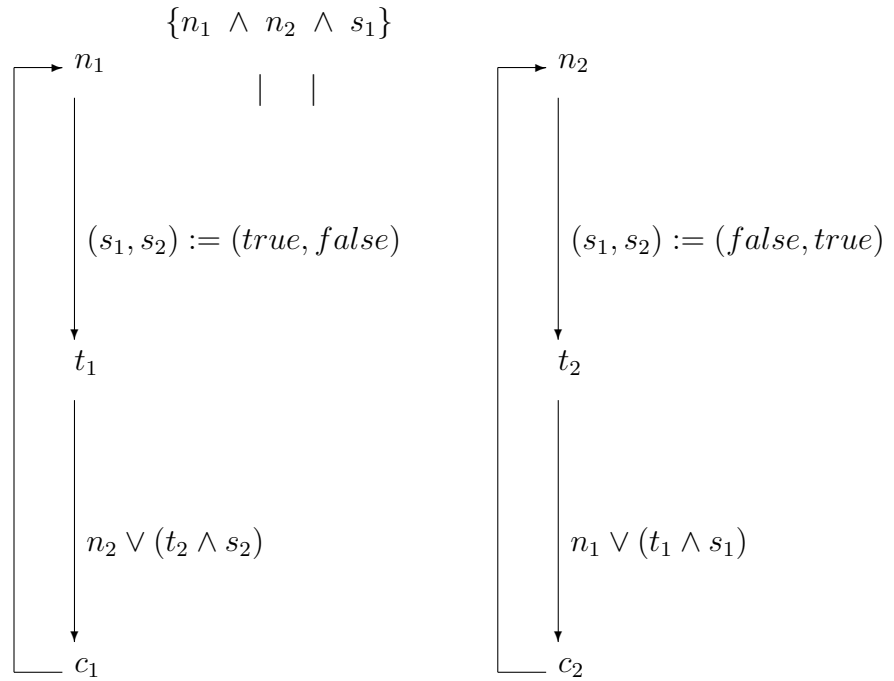


Figure 1: The algorithm for MUTEX