Formal Methods in Software Development
Resume of the 07/10/2020 lesson

Igor Melatti and Ivano Salvo

- Murphi or Murφ, the simplest among "model checkers"
  - as all model checkers we will see in this course, Murphi may be freely downloaded with the source code, thus it may also be modified
  - links for download of all model checkers we will see are on the course web-page: `http://twiki.di.uniroma1.it/twiki/view/MFS/FormalMethodsInSoftwareDevelopment20202021`
- Formally, as all model checkers, Murphi needs the following input:
  1. a description of the system $\mathcal{S}$ you want to verify (i.e., the "model" you want to "check")
  2. this is essentially a Kriepke structure
  3. a property φ you want the system $\mathcal{S}$ to satisfy
- The output will be either OK or FAIL
  - if FAIL, it is possible to tell Murphi to print a *counterexample*
- In Murphi, both the description of $\mathcal{S}$ and of φ must be written in a single text file, following a precise syntax
  - in other model checkers we will see (e.g., SPIN), this syntax has a name; but this is not the case for Murphi
  - thus, we will refer to it simply as *Murphi input language*
  - as we will see, in many points Murphi input language is similar to some imperative programming language, especially Pascal (for statements) and C (for expressions)
- A description for $\mathcal{S}$ and φ written in the Murphi input language must be organized as follows:
  1. definitions of:
     - *constants*, also named *parameters*

- *data types*, divided in *simple* and *composed*
    - simple types are only two: *enumerations* and *integer subranges*
    - the *boolean* boolean data type is predefined as an enumeration (true, false)
    - the composed types are formed using *array* and/or *records* (structs), possibly mixed, following the Pascal syntax
- *global variables*, each having one of the types above
    - global variables are fundamental, as they define the *states space* S
    - that is, S is defined by all possible values of all global variables
        - thus, is defined by the Cartesian product of all types of all global variables defined
        - as all types are *finite*, S may be huge but it is always finite
        - see example below
- note that such types of definitions may be mixed, of course keeping in mind variables scoping (e.g., if you need constant A to define variable B, you must define constant A before B)

2. definitions of:
    - *functions*
        - return a value
        - may have side effects (i.e., modify a global variable)
        - may modify input arguments, but must be explicitly stated as in Pascal (parameter passed as *reference*)
    - *procedures*
        - do not return a value
        - may have side effects (i.e., modify a global variable)
        - may modify input arguments, but must be explicitly stated as in Pascal (parameter passed as *reference*)
    - for both functions and procedures:
        - Pascal-like syntax
        - it is possible to define and use *local* variables
        - local variables *must not* be considered in the definition of the state space S
    - again, you can mix them, provided scoping is respected (if function F calls function G, then G must be defined before F)
3. definitions (mixed as you like it) of:
    - *start states*, defined as Pascal-like statements, intended as atomically executed

            * may contain the typical statements of imperative programming languages: assignments, cycles, ifs, functions and procedures calls
            * local variables may be defined
        - *rules*, each defined by:
            * a *(application) guard*, defining if a rule is applicable (*fired*, as Murphi says) or not
                · of course it must be a boolean expression
                · only global variables and constants may occur in a guard
                · it is of course possible to call functions
            * a *body*, again formed by atomically executed Pascal-like statements
                · may contain the typical statements of imperative programming languages: assignments, cycles, ifs, functions and procedures calls
                · local variables may be defined
            * an optional string, working as a short comment for the rule
            * by the way, comments may be either with C syntax (`/**/`) or Pascal syntax (`--`)
        - *invariants*, each of them defines a property to be checked
            * same as guards: it must be a boolean expression
            * only global variables and constants may occur in a guard
            * it is of course possible to call functions

- Finally, at least an initial state and one rule must be present (see `00.minimal_model.m`)

- Murphi checks that all reachable states of S satisfy all invariants

    - a state $s \in S$ is *reachable* if there exists a path in the transition graph from an initial state to s
    - that is: starting from an initial state, there exists a chain of rules, each applied to the state obtained from the preceding one, leading to s
    - this is a *safety* property

- Example: G. L. Peterson protocol for mutual exclusion of 2 processes (1981)

    - two identical processes
        * the first is as in Figure 2, for the second it is necessary to exchange 1's with 2's and viceversa
        * each applies Peterson protocol to access to the critical section L3

```
boolean flag [2];
int turn;
void P0()                                   Peterson's Algorithm
{
     while (true) {
           flag [0] = true;
           turn = 1;
           while (flag [1] && turn == 1) /* do nothing */;
           /* critical section */;
           flag [0] = false;
           /* remainder */;
     }
}
void P1()
{
     while (true) {
           flag [1] = true;
           turn = 0;
           while (flag [0] && turn == 0) /* do nothing */;
           /* critical section */;
           flag [1] = false;
           /* remainder */
     }
}
void main()
{
     flag [0] = false;
     flag [1] = false;
     parbegin (P0, P1);
}
```

Figure 1: Peterson's protocol in pseudo-code

    * the first issuing the request enters L3
    * `Q` is a global variable, defined as an array of two integers
        · each process i may modify `Q`[i] and read `Q`[$(i+1) mod 2$]
    * `turn` is another global variable, which may be both read and modified by both processes

- Murphi description for Peterson protocol: let's start with the variables
    * of course `turn` and `Q`, but also two variables `P` for the modality ("states" in Figure 2)
    * see `01.2_peterson.no_rulesets.no_parametric.m`
    * to this aim, we define constants and types
    * the `N` constant (number of processes) is here fictious: only 2 processes, not more
    * this version of Peterson protocol only works for 2 processes
- thus, the state space is $S = \texttt{label\_t}^2 \times \{\texttt{true}, \texttt{false}\}^2 \times \{1, 2\}$
- see Figure 3
- hence, $|S| = 5^2 \times 2^2 \times 2 = 200$ (there are 200 possible states)
    * as a matter of comparison, the "state" L0 in Figure 2 actually contains $5^1 \times 2^2 \times 2 = 40$ states...

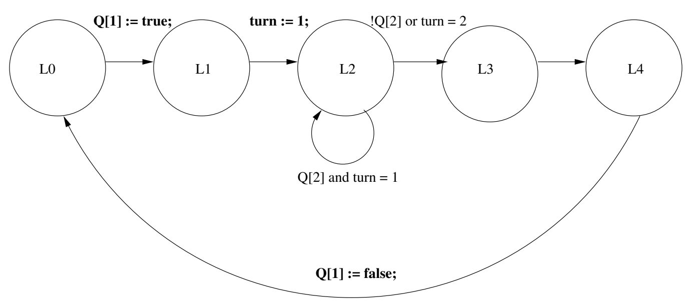

Figure 2: Peterson's protocol for process 1

P	$v \in \{L0, L1, L2, L3, L4\}$	$v \in \{L0, L1, L2, L3, L4\}$
Q	$v \in \{true, false\}$	$v \in \{true, false\}$
turn	$v \in \{1..N\}$	

Figure 3: Variables for Murphi model describing Peterson protocol

- however, as we will see, *reachable* states are about 10 times less
- 2 initial states: `turn` may be initialied with any value in its domain
- note that `01.2_peterson.no_rulesets.no_parametric.m` we have rules repeated 2 times in a nearly equal fashion
- this can be done in this very simple model, but in general descriptions must be *parametric*
- that is, if we want to check Peterson with 3 processi, currently we would have to add one more rule in the desciprion
- instead, it must be possible to only change the value of `N` from 2 to 3
- to write parametric descriptions in Murphi, rules are grouped with *rulesets*
  * an index will allow to describe the behavior of the generic process i
  * see `02.2_peterson.with_rulesets.no_parametric.m`
- invariant: of course, at any execution instant, there must be only one state in L3 (mutual exclusion)
  * in a first order logic, it would be something like:

    $$\forall k \in \{1, \ldots, \mathtt{N}\}.\ \forall k' \in \{1, \ldots, \mathtt{N}\}.\ (k \neq k' \wedge \mathtt{P}[k] = L3) \Rightarrow \mathtt{P}[k'] \neq L3$$

  * or, as a reverse:

    $$\neg(\exists k \in \{1, \ldots, \mathtt{N}\}.\ \exists k' \in \{1, \ldots, \mathtt{N}\}.\ k \neq k' \wedge \mathtt{P}[k] = L3 \wedge \mathtt{P}[k'] = L3)$$

  * in the first version, it is stated what is correct to happen
  * in the first version, it is stated what is wrong to happen
  * in both `00.2_peterson.with_rulesets.no_parametric.m` and `02.2_peterson.no_rulesets.no_parametric.m` invariant is not parametric
  * see `03.2_peterson.with_rulesets.parametric.m`
  * note that in S there surely are states violating mutual exclusion, e.g., $\langle L3, L3, false, false, 1\rangle$
  * but they are not *reachable*