

Software Testing and Validation

A.A. 2023/2024

Corso di Laurea in Informatica

The Murphi Model Checker

Igor Melatti

Università degli Studi dell'Aquila

Dipartimento di Ingegneria e Scienze dell'Informazione e Matematica



UNIVERSITÀ
DEGLI STUDI
DELL'AQUILA



DISIM
Dipartimento di Ingegneria
e Scienze dell'Informazione
e Matematica

- Murphi or Mur φ , 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: https://igormelatti.github.io/sw_test_val/20232024/index.html



Murphi

- Formally, as all model checkers, Murphi needs the following input:
 - 1 a description of the system S you want to verify (i.e., the “model” you want to “check”)
 - as we will see, this is essentially a Kripke structure
 - 2 a property φ you want the system S to satisfy
- The output will be either OK or FAIL
 - if FAIL, it is possible to tell Murphi to print a *counterexample*



UNIVERSITÀ
DEGLI STUDI
DELL'AQUILA



DISIM
Dipartimento di Ingegneria
e Scienze dell'Informazione
e Matematica

Murphi

- 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 languages, especially Pascal (for statements) and C (for expressions)



UNIVERSITÀ
DEGLI STUDI
DELL'AQUILA



DISIM
Dipartimento di Ingegneria
e Scienze dell'Informazione
e Matematica

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*
 - there are only two simple types: *enumerations* and *integer subranges*
 - the *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



- 1. (Continuing)
 - 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 definitions may be mixed, of course keeping in mind variables scoping
 - e.g., if you need constant A to define type B of variable C , you must define constant A first, then type B and finally variable C
 - type B could also be used inline directly when declaring C



- 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
- E.g., if function F calls procedure G which calls function H , then G must be defined before F and H before G



- 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(*n application*) *guard*, defining if a rule is applicable (*fired*, as Murphi says) or not
 - a *body*, again formed by atomically executed Pascal-like statements
 - an optional string, working as a short comment for the rule
 - by the way, comments may be either with C syntax (*/**/*) or Pascal syntax (*--*)



Murphi

- Of course the guard must be a boolean expression
- Only global variables and constants may occur in a guard
 - actually, also ruleset indexes, we will be back on this
- It is possible to call functions (not procedures!)
- The body may contain the typical statements of imperative programming languages: assignments, cycles, ifs, functions and procedures calls
- Local variables may be defined and used



UNIVERSITÀ
DEGLI STUDI
DELL'AQUILA



DISIM
Dipartimento di Ingegneria
e Scienze dell'Informazione
e Matematica

- 3. (Continuing):
 - *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
 - exceptions are possible when `forall` or `exist` are used
 - it is of course possible to call functions
- Finally, at least one 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)

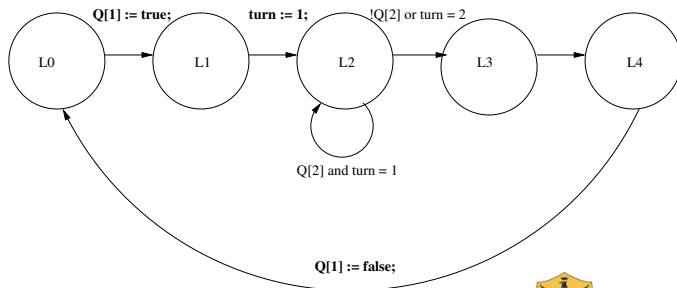
Peterson's Algorithm

```
boolean flag [2];
int turn;
void P0()
{
    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);
}
```



Murphi

- Example: G. L. Peterson protocol for mutual exclusion of 2 processes (1981)
- UML-like state diagram: this is the first process; the second may be obtained exchanging 1's with 2's and viceversa



UNIVERSITÀ
DEGLI STUDI
DELL'AQUILA



DISIM
Dipartimento di Ingegneria
e Scienze dell'Informazione
e Matematica

- Example: G. L. Peterson protocol for mutual exclusion of 2 processes (1981)
 - two identical processes
 - each applies Peterson protocol to access to the critical section L3
 - 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 $Q[(i + 1) \bmod 2]$
 - $turn$ is another global variable, which may be both read and modified by both processes



Murphi

- 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 the UML-like state diagram)
 - 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 fictitious: only 2 processes, not more
 - this version of Peterson protocol only works for 2 processes
- thus, the state space is
$$S = \text{label_t}^2 \times \{\text{true}, \text{false}\}^2 \times \{1, 2\}$$



UNIVERSITÀ
DEGLI STUDI
DELL'AQUILA



DISIM
Dipartimento di Ingegneria
e Scienze dell'Informazione
e Matematica

Variables for Murphi Model Describing Peterson Protocol

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 $v \in \{1..N\}$



UNIVERSITÀ
DEGLI STUDI
DELL'AQUILA



DISIM
Dipartimento di Ingegneria
e Scienze dell'Informazione
e Matematica

- Hence, $|S| = 5^2 \times 2^2 \times 2 = 200$ (there are 200 possible states)
 - as a matter of comparison, the “state” L0 in the UML-like state diagram actually contains $5^1 \times 2^2 \times 2 = 40$ states...
- However, as we will see, *reachable* states are about 10 times less
- 2 initial states: turn may be initialized 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*



Murphi

- If we want to check Peterson with 3 processes, currently we would have to add rules in the description
 - very similar to the ones already present, only changing the index to 3
- 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`, but invariant is still for two processes only



UNIVERSITÀ
DEGLI STUDI
DELL'AQUILA



DISIM
Dipartimento di Ingegneria
e Scienze dell'Informazione
e Matematica

- Finally, in `03.2.peterson.with_rulesets.parametric.m` also the invariant is parametric in N
 - `Exists $x:T$ $E(x)$ End` is equivalent to $\bigvee_{x \in T} E(x)$
 - `Forall $x:T$ $E(x)$ End` is equivalent to $\bigwedge_{x \in T} E(x)$
 - all types $T = \{x_1, \dots, x_{|T|}\}$ are finite, thus it is a finite formula

