Software Testing and Validation

A.A. 2022/2023

Corso di Laurea in Informatica

Kriepke Structures and Murphi Verification Algorithm(s)

Igor Melatti

Università degli Studi dell'Aquila

Dipartimento di Ingegneria e Scienze dell'Informazione e Matematica

Kripke Structures

- Let AP be a set of "atomic propositions"
  - in the sense of first-order logic: each atomic proposition is either true or false
  - tipically identified with lower case letters $p, q, \ldots$
- A *Kripke Structure* (KS) over AP is a 4-tuple $\langle S, I, R, L \rangle$
  - S is a finite set, its elements are called *states*
  - $I \subseteq S$ is a set of *initial states*
  - $R \subseteq S \times S$ is a *transition relation*
  - $L : S \to 2^{AP}$ is a *labeling function*

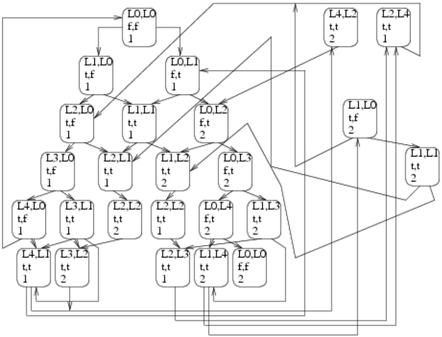
Labeled Transition Systems

- A *Labeled Transition System* (LTS) is a 4-tuple $\langle S, I, \Lambda, \delta \rangle$
  - S is a finite set of states as before
  - $I \subseteq S$ is a set of initial states as before (not always included)
  - Λ is a finite set of *labels*
  - $\delta \subseteq S \times \Lambda \times S$ is a *labeled transition relation*

Peterson's Mutual Exclusion as a Kripke Structure

- $S = \{(p_1, p_2, q_1, q_2, t) \mid p_1, p_2 \in \{\mathrm{L0}, \mathrm{L1}, \mathrm{L2}, \mathrm{L3}, \mathrm{L4}\}, q_1, q_2 \in \{0, 1\}, t \in \{1, 2\}\} = \{\mathrm{L0}, \mathrm{L1}, \mathrm{L2}, \mathrm{L3}, \mathrm{L4}\}^2 \times \{0, 1\}^2 \times \{1, 2\}$
- $I = \{\mathrm{L0}\}^2 \times \{0\}^2 \times \{1, 2\}$
- R: see next slide
- $AP = \{(\mathtt{P}_1 = v) \mid v \in \{\mathrm{L0}, \mathrm{L1}, \mathrm{L2}, \mathrm{L3}, \mathrm{L4}\}\} \cup \{(\mathtt{P}_2 = v) \mid v \in \{\mathrm{L0}, \mathrm{L1}, \mathrm{L2}, \mathrm{L3}, \mathrm{L4}\}\} \cup \{(\mathtt{Q}_1 = v) \mid v \in \{0, 1\}\} \cup \{(\mathtt{Q}_2 = v) \mid v \in \{0, 1\}\} \cup \{(\mathtt{turn} = v) \mid v \in \{1, 2\}\}$
  - e.g.: $L(\mathrm{L0}, \mathrm{L0}, 0, 0, 1)) = \{(\mathtt{P}_1 = \mathrm{L0}), (\mathtt{P}_2 = \mathrm{L0}), (\mathtt{Q}_1 = 0), (\mathtt{Q}_2 = 0), (\mathtt{turn} = 1)\}$

Peterson's Mutual Exclusion as a Kripke Structure



E.g.: $((\mathrm{L0}, \mathrm{L0}, 0, 0, 1), (\mathrm{L1}, \mathrm{L0}, 1, 0, 1)) \in R$, whilst

$((\mathrm{L0}, \mathrm{L0}, 0, 0, 1), (\mathrm{L2}, \mathrm{L0}, 0, 0, 1)) \notin R$

Of course, $|R|$ = number of arrows in figure above

Kripke Structure vs Labeled Transition Systems

- KSs have atomic propositions on states, LTSs have labels on transitions
- In model checking, atomic propositions are mandatory
  - to specify the formula to be verified, as we will see
  - a first example was the invariant in Murphi
- Instead, it is not required to have a label on transitions
  - Murphi allows to do so, but it is optional
  - may be easily added automatically, if needed
- Labels are typically needed when:
  - we deal with macrostates, as in UML state diagrams
  - when we are describing a complex system by specifying its sub-components, so labels are used for synchronization

Total Transition Relation

- In many cases, the transition relation R is required to be *total*
- $\forall s \in S. \exists s' \in S : (s, s') \in R$
  - this of course allows also $s = s'$ (*self loop*)
- In the Peterson's example, the relation is actually total
  - Murphi allows also non-total relations, by using option `-ndl`
  - note however that not giving option `-ndl` is stronger: $\forall s \in S. \exists s' \in S : s \neq s' \wedge (s, s') \in R$
  - otherwise, if s is s.t. $\forall s'.\ s = s' \vee (s, s') \notin R$, Murphi calls s a *deadlock* state
  - that is, you cannot go anywhere, except possibly self looping on s
- By deleting any rule, we will obtain a non-total transition relation

Non-Determinism

- The transition relation is, as the name suggests, a relation
- Thus, starting from a given state, it is possible to go to many different states
  - in a deterministic system, $\forall s_1, s_2, s_3 \in S.\ (s_1, s_2) \in R \land (s_1, s_3) \in R \rightarrow s_2 = s_3$
  - this does not hold for KSs
- This means that, starting from state s_1, the system may *non-deterministically* go either to s_2 or to s_3
  - or many other states
- Motivations for non-determinism: modeling choices!
  - underspecified subsystems
  - unpredictable interleaving
  - interactions with an uncontrollable environment
  - ...

UNIVERSITÀ DEGLI STUDI DELL'AQUILA
DISIM Dipartimento di Ingegneria e Scienze dell'Informazione e Matematica

Some Useful Notation

- Given a KS $\mathcal{S} = \langle S, I, R, L \rangle$, we can define:
  - the *predecessor* function $\mathrm{Pre}_{\mathcal{S}} : S \to 2^S$
    - defined as $\mathrm{Pre}_{\mathcal{S}}(s) = \{s' \in S \mid (s', s) \in R\}$
    - we will write simply $\mathrm{Pre}(s)$ when $\mathcal{S}$ is understood
  - the *successor* function $\mathrm{Post} : S \to 2^S$
    - defined as $\mathrm{Post}(s) = \{s' \in S \mid (s, s') \in R\}$
- Note that, if $\mathcal{S}$ is deterministic, $\forall s \in S.\ |\mathrm{Post}(s)| \leq 1$

UNIVERSITÀ DEGLI STUDI DELL'AQUILA

DISIM Dipartimento di Ingegneria e Scienze dell'Informazione e Matematica

Paths in KSs

- A path (or *execution*) on a KS $\mathcal{S} = \langle S, I, R, L \rangle$ is a sequence $\pi = s_0 s_1 s_2 \ldots$ such that:
  - $\forall i \geq 0.\ s_i \in S$ (it is composed by states)
  - $\forall i \geq 0.\ (s_i, s_{i+1}) \in R$ (it only uses valid transitions)
- We will denote i-th state of a path as $\pi(i) = s_i$
- Note that paths in LTSs also have actions: $\pi = s_0 a_0 s_1 a_1 \ldots$ s.t. $(s_i, a_i, s_{i+1} \in \delta)$

Paths in KSs

- The *length* of a path π is the number of states in π
  - paths can be either finite $\pi = s_0 s_1 \ldots s_n$, in which case $|\pi| = n + 1$
  - or infinite $\pi = s_0 s_1 \ldots$, in which case $|\pi| = \infty$
- We will denote the prefix of a path up to i as $\pi|_i = s_0 \ldots s_i$
  - a prefix of a path is always a finite path
- A path π is *maximal* iff one of the following holds
  - $|\pi| = \infty$
  - $|\pi| = n + 1$ and $|\mathrm{Post}(\pi(n))| = 0$
    - that is, $\forall s \in S.\ (\pi(n), s) \notin R$
    - i.e., the last state of the path has no successors
    - often called *terminal state*
- If R is total, maximal paths are always infinite
  - for many model checking algorithms, this is required

Reachability

- The set of paths of $\mathcal{S}$ starting from $s \in S$ is denoted by $\mathrm{Path}(\mathcal{S}, s) = \{\pi \mid \pi \text{ is a path in } \mathcal{S} \wedge \pi(0) = s\}$
- The set of paths of $\mathcal{S}$ is denoted by $\mathrm{Path}(\mathcal{S}) = \cup_{s \in I} \mathrm{Path}(\mathcal{S}, s)$
  - that is, they must start from an initial state
- A state $s \in S$ is *reachable* iff $\exists \pi \in \mathrm{Path}(\mathcal{S}), k \leq |\pi| : \ \pi(k) = s$
  - i.e., there exists a path from an initial state leading to s through valid transitions
- The set of reachable states is defined by $\mathrm{Reach}(\mathcal{S}) = \{\pi(i) \mid \pi \in \mathrm{Path}(\mathcal{S}), i \leq |\pi|\}$

Safety Property Verification

- Verification of *invariants*: nothing bad happens
- The property is a formula $\varphi : S \to \{0, 1\}$
  - built using boolean combinations of atomic propositions in $p \in AP$
  - i.e., the syntax is

$$\Phi : (\Phi) \mid \Phi \wedge \Phi \mid \Phi \vee \Phi \mid \neg\Phi \mid p$$

- The KS $\mathcal{S}$ satisfies φ iff φ holds on all reachable states
  - $\forall s \in \mathrm{Reach}(\mathcal{S}).\ \varphi(s) = 1$
- Note that it may happen that $\varphi(s) = 0$ for some $s \in S$: never mind, if $s \notin \mathrm{Reach}(\mathcal{S})$

From Murphi Description $\mathcal{M}$ to KS $\mathcal{S}$

- First, we mathematically define a Murphi description $\mathcal{M}$
- $V = \langle v_1, \ldots, v_n \rangle$ is the set of global variables of $\mathcal{M}$, with domains $\langle D_1, \ldots, D_n \rangle$
  - all variables are *unfolded* to the Murphi simple types
    - integer subranges
    - enumerations
    - the special "undefined" value should be added to all simple types
  - that is, if a variable is an array with q elements, then it is actually to be considered as q different variables
  - the same for records (and any nesting of arrays and records)
  - as an example: `var a : array [1..n] of record begin b : 1..m; c: 1..k; endrecord`
  - then there will be $2n$ variables as follows: $a1b, \ldots, anb, a1c, \ldots, anc$
  - the first n with type `1..m`, the other with type `1..k`

UNIVERSITÀ DEGLI STUDI DELL'AQUILA
DISIM Dipartimento di Ingegneria e Scienze dell'Informazione e Matematica

From Murphi Description $\mathcal{M}$ to KS $\mathcal{S}$

- $I = \{I_1, \ldots, I_k\}$ is the set of `startstate` sections in $\mathcal{M}$
  - startstates may be defined inside rulesets; again, all rulesets are *unfolded*
  - thus, if a startstate I is inside m nested rulesets $\mathcal{R}_1, \ldots, \mathcal{R}_m$...
  - and each ruleset $\mathcal{R}_i$ is defined on an index j_i spanning on a domain $\mathcal{D}_i$ (note that $\mathcal{D}_i$ must be a simple type)...
  - then there actually are $\prod_{l=1}^{m} |\mathcal{D}_l|$ startstates to be considered, instead of just one
  - of course, in each of these startstates definitions, the tuple $j_1, \ldots, j_m$ takes all possible values of $\mathcal{R}_1 \times \ldots \times \mathcal{R}_m$
- $T = \{T_1, \ldots, T_p\}$ is the set of `rule` sections in $\mathcal{M}$
  - again, if rulesets are present, they are *unfolded*

From Murphi Description $\mathcal{M}$ to KS $\mathcal{S}$

- The Kriepke structure $\mathcal{S} = \langle S, I, R, L \rangle$ described by $\mathcal{M}$ is such that:
  - $S = D_1 \times \ldots \times D_n$
  - $s \in I$ iff there is a startstate $I_i \in I$ s.t. s may be obtained by applying the body of I_i
  - $(s, t) \in R$ iff there is a rule $T_i \in T$ s.t. T_i guard is true in s and T_i body changes s to t
  - $AP = \{(v = d) \mid v = v_i \in V \wedge d \in D_i\}$
  - $(v = d) \in L(s)$ iff variable v has value d in s

UNIVERSITÀ DEGLI STUDI DELL'AQUILA
DISIM Dipartimento di Ingegneria e Scienze dell'Informazione e Matematica

From Murphi Description $\mathcal{M}$ to KS $\mathcal{S}$

- We also assume to have a function defining the semantics of Murphi (sequence of) statements
  - those in bodies of rules and startstates
- Let $\mathcal{P}$ be the set of all possible (syntactically legal) Murphi statements
  - including `while`, `if`, `for`, assignments...
- Thus, let $\eta : \mathcal{P} \times D_1 \times \ldots \times D_n \to D_1 \times \ldots \times D_n$ be our evaluation function
  - it takes a Murphi statement $P \in \mathcal{P}$ and the state s preceding such statement
  - it returns the new state s' obtained by executing P on s
  - e.g., $\eta(\mathtt{a} := \mathtt{a} + 1; \mathtt{b} := \mathtt{b} - 1, (1, 2, 3)) = (2, 1, 3)$
  - η may be defined, e.g., using operational semantics

From Murphi Description $\mathcal{M}$ to KS $\mathcal{S}$

- We also assume to have a function defining the semantics of Murphi boolean expression
  - those in guards of rules
  - and in invariants!
- Let $\mathcal{Q}$ be the set of all possible (syntactically legal) Murphi boolean expressions
  - including `forall`, `exists`, equality checks...
- Thus, let $\zeta : \mathcal{Q} \times D_1 \times \ldots \times D_n \to \{0, 1\}$ be our evaluation function
  - it takes a Murphi boolean expression $Q \in \mathcal{Q}$ and the state s to be evaluated
  - it returns 1 iff Q is true in s
  - e.g., $\zeta((\mathtt{a} = 3 | \mathtt{b} = 4), (1, 4, 3)) = 1$
  - ζ may be defined using atomic propositions in AP (see below)

From Murphi Description $\mathcal{M}$ to KS $\mathcal{S}$

- Let $Q \in \mathcal{Q}$ be a Murphi boolean expression
- Flatten Q w.r.t. `Forall` and `Exists`
  - `Forall` is replaced by ANDs, `Exists` by ORs
  - e.g., from `Exists i1: pid Do Exists i2: pid Do (i1 != i2 & P[i1] = L3 & P[i2] = L3) End End` ...
  - ... to `(1 != 1 & P[1] = L3 & P[1] = L3) | (2 != 1 & P[2] = L3 & P[1] = L3) | (1 != 2 & P[1] = L3 & P[2] = L3) | (2 != 2 & P[2] = L3 & P[2] = L3)`
- If we replace each variable $v_i \in V$ occurring in Q with a value $w_{j_i} \in D_i$, we obtain a boolean value (true or false)
  - e.g., the former evaluates to true by setting `P[1] = L3` and `P[2] = L3`
- Thus, $\zeta(Q, s) = 1$ iff $Q(w_{j_1}, \ldots, w_{j_n}) = 1$
  - where each w_{j_i} is such that $(v_i = w_{j_i}) \in L(s)$
  - $Q(w_{j_1}, \ldots, w_{j_n})$ is the result of replacing variable v_i with value w_{j_i}

UNIVERSITÀ DEGLI STUDI DELL'AQUILA

DISIM Dipartimento di Ingegneria e Scienze dell'Informazione e Matematica

From Murphi Description $\mathcal{M}$ to KS $\mathcal{S}$

- $(s, t) \in R$ iff there is a rule $T_i \in T$ s.t. T_i guard is true in s and T_i body changes s to t
- By using η and ζ, we can be more precise:
  - "T_i guard is true" means $\zeta(G(T_i), s) = 1$, being $G(T_i)$ the Murphi expression used as guard of rule T_i
  - "T_i body changes s to t" means $\eta(B(T_i), s) = t$, being $B(T_i)$ the Murphi statement used as body of rule T_i
- $s \in I$ iff there is a startstate $I_i \in I$ s.t. s may be obtained by applying the body of I_i
  - "s may be obtained by applying the body of I_i" means $\eta(B(I_i), (\perp, \ldots, \perp)) = s$, being $B(T_i)$ the Murphi statement used as body of startstate I_i

From Murphi Description $\mathcal{M}$ to KS $\mathcal{S}$

- $(s, t) \in R$ iff there is a rule $T_i \in T$ s.t. T_i guard is true in s and T_i body changes s to t:
  - that is: in the body of T_i, variables starting values are those of s
  - note that there may be two or more rules defining the same transition from s to t; no problem with this
  - simply, the same transition is described by multiple rules
- A state s is a deadlock state for two possible reasons:
  1. $(s, t) \notin R$ for all $t \in S$, i.e., the values for the variables in s do not satisfy any ruleset guard
  2. $(s, t) \in R \to t = s$, i.e., there is some ruleset guard which is satisfied by s, but its body do not change any of the global variables (e.g., the body is empty)

How to Verify a Murphi Description $\mathcal{M}$

- Theoretically, extract KS $\mathcal{S}$ and property φ from $\mathcal{M}$ as described above
  - for a given invariant I in $\mathcal{M}$, $\varphi(s) = \zeta(I, s)$ for all $s \in S$
- Then, KS $\mathcal{S}$ satisfies φ iff φ holds on all reachable states
  - $\forall s \in \mathrm{Reach}(\mathcal{S}).\ \varphi(s) = 1$
- Thus, consider KS as a graph and perform a visit
  - states are nodes, transitions are edges
- If a state e s.t. $\varphi(e) = 0$ is found, then we have an error
- Otherwise, all is ok

How to Verify a Murphi Description $\mathcal{M}$

- From a practical point of view, many optimization may be done, but let us stick to the previous scheme
- The worst case time complexity for a DFS or a BFS is $O(|V| + |E|)$ (and same for space complexity)
- For KSs, this means $O(|S| + |R|)$, thus it is linear in the size of the KS
- Is this good? NO! Because of the *state space explosion problem*
- Assuming that B bits are needed to encode each state
  - i.e., $B = \sum_{i=1}^{n} b_i$, being b_i the number of bits to encode domain D_i
- We have that $|S| = O(2^B)$

State Space Explosion

- The "practical" input dimension is B, rather than $|S|$ or $|R|$
- Typically, for a system with N components, we have $O(N)$ variables, thus $O(B)$ encoding bits
- It is very common to verify a system with N components, and then (if N is ok) also for $N + 1$ components
  - verifying a system with a generic number N of components is a typically proof checker task...
- This entails an esponential increase in the size of $|S|$
- Thus we need "clever" versions of BFS/DFS

Standard BFS: No Good for Model Checking

- Assumes that all graph nodes are in RAM
- For KSs, graph nodes are states, and we now there are too many
  - state space explosion
- You also need a full representation of the graph, thus also edges must be in RAM
  - using adjacency matrices or lists does not change much
  - for real-world systems, you may easily need TB of RAM
- Even if you have all the needed RAM, there is a huge preprocessing time needed to build the graph from the Murphi specification
- Then, also BFS itself may take a long time

Murphi BFS

- We need a definition inbetween the model and the KS: NFSS (Nondeterministic Finite State System)
- $\mathcal{N} = \langle S, I, \mathrm{Post} \rangle$, plus the invariant φ
  - S is the set of states, $I \subseteq S$ the set of initial states
  - $\mathrm{Post} : S \to 2^S$ is the successor function as defined before
    - given a state s, it returns T s.t. $t \in T \to (s, t) \in R$
  - no labeling, we already have φ

Murphi BFS

- KSs and NFSSs differ on having Post instead of R
- Post may easily be defined from the Murphi specification
- Such definition is implicit, as programming code, thus avoiding to store adjacency matrices or lists
  - $t \in \mathrm{Post}(s)$ iff there is a rule $T_i \in T$ s.t. T_i guard is true in s and T_i body changes s to t
    - see above for using η and ζ
  - Essentially, if the current state is s, it is sufficient to inspect all (flattened) rules in the Murphi specification $\mathcal{M}$
    - for all guards which are enabled in s, execute the body so as to obtain t, and add t to $\mathrm{next}(s)$
  - This is done "on the fly", only for those states s which must be explored

UNIVERSITÀ DEGLI STUDI DELL'AQUILA
DISIM Dipartimento di Ingegneria e Scienze dell'Informazione e Matematica

Murphi Simulation

```
void Make_a_run(NFSS 𝒩, invariant φ)
{
 let 𝒩 = ⟨S, I, Post⟩;
 s_curr = pick_a_state(I);
 if (!φ(s_curr))
  return with error message;
 while (1) { /* loop forever */
  s_next = pick_a_state(Post(s_curr));
  if (!φ(s_next))
   return with error message;
  s_curr = s_next;
 }
}
```

```
void Make_a_run(NFSS 𝒩, invariant φ)
{
 let 𝒩 = ⟨S, I, Post⟩;
 s_curr = pick_a_state(I);
 if (!φ(s_curr))
  return with error message;
 while (1) { /* loop forever */
  if (Post(s_curr) = ∅)
   return with deadlock message;
  s_next = pick_a_state(Post(s_curr));
  if (!φ(s_next))
   return with error message;
  s_curr = s_next;
 }
}
```

Murphi Simulation

- Similar to testing
- If an error is found, the system is bugged
  - or the model is not faithful
  - actually, Murphi simulation is used to understand if the model itself contains errors
- If an error is not found, we cannot conclude anything
- The error state may lurk somewhere, out of reach for the random choice in `pick_a_state`

Standard BFS (Cormen-Leiserson-Rivest)

```
BFS(G, s)
for ogni vertice u ∈ V[G] − {s}
      do  color[u] ← WHITE
          d[u] ← ∞
          π[u] ← NIL
color[s] ← GRAY
d[s] ← 0
π[s] ← NIL
Q ← {s}
while Q ≠ ∅
      do  u ← head[Q]
          for ogni v ∈ Adj[u]
                do if color[v] = WHITE
                      then  color[v] ← GRAY
                            d[v] ← d[u] + 1
                            π[v] ← u
                            ENQUEUE(Q, v)
          DEQUEUE(Q)
          color[u] ← BLACK
```

UNIVERSITÀ DEGLI STUDI DELL'AQUILA
DISIM Dipartimento di Ingegneria e Scienze dell'Informazione e Matematica

Murphi BFS

```
FIFO_Queue Q;
HashTable T;

bool BFS(NFSS 𝒩, AP φ)
{
 let 𝒩 = (S, I, Post);
 foreach s in I {
  if (!φ(s))
   return false;
 }
 foreach s in I
  Enqueue(Q, s);
 foreach s in I
  HashInsert(T, s);
```

Murphi BFS

```
while (Q ≠ ∅) {
 s = Dequeue(Q);
 foreach s_next in Post(s) {
  if (!φ(s_next))
   return false;
  if (s_next is not in T) {
   Enqueue(Q, s_next);
   HashInsert(T, s_next);
  } /* if */ } /* foreach */ } /* while */
return true;
}
```

Murphi BFS

- Edges are never stored in memory
- (Reachable) states are stored in memory only at the end of the visit
  - inside hashtable `T`
- This is called *on-the-fly* verification
- States are marked as visited by putting them inside an hashtable
  - rather than coloring them as gray or black
  - which needs the graph to be already in memory

State Space Explosion

- State space explosion hits in the FIFO queue `Q` and in the hashtable `T`
  - and of course in running time...
- However, `Q` is not really a problem
  - it is accessed *sequentially*
  - always in the front for extraction, always in the rear for insertion
  - can be efficiently stored using disk, much more capable of RAM
- `T` is the real problem
  - random access, not suitable for a file
  - what to do?
  - before answering, let's have a look at Murphi code

Murphi Usage

- As for all *explicit* model checker, a Murphi verification has the following steps:
  0. compile Murph source code and write a Murphi model `model.m`
  1. invoke Murphi compiler on `model.m`: this generates a file `model.cpp`
     - `mu options model.m`
     - see `mu -h` for available options
  2. invoke C++ compiler on `model.cpp`: this generates an executable file
     - `g++ -Ipath_to_include model.cpp -o model`
     - `path_to_include` is the `include` directory inside Murphi distribution
  3. invoke the executable file
     - `./model options`
     - see `./model -h` for available options

UNIVERSITÀ DEGLI STUDI DELL'AQUILA

DISIM Dipartimento di Ingegneria e Scienze dell'Informazione e Matematica

Murphi compiler

- Executable mu is in src directory of Murphi distribution
- Obtained by compiling the 25 source files in src
  - of course, a Makefile is provided for this
- Standard compiler implementation, with Flex lexical analyzer (mu.l) and Yacc parser (mu.y)
- The main function which builds model.cpp is program::generate_code in cpp_code.cpp (called by main, in mu.cpp)
- program::generate_code uses the parse tree generated by Yacc to "implement" in C++ the guards and the bodies of the rules
- The result goes in model.cpp: model-specific code

Organization of model.cpp

- Each Murphi variable v (local or global) corresponds to a C++ instance mu_v of the class mu__int (possibly through class generalizations)
- Class mu__int is used to handle variables with max value 254 (255 is used for the undefined value)
- For integer subranges with greater values, class mu__long is used; also mu__byte (equal to mu__int...) and mu__boolean exist
- If v is a local variable, mu_v directly contains the value (attribute cvalue, in_world is false)
- Otherwise, if v is global, mu_v retrieves the value from a fixed-address structure containing the current state value (workingstate; in_world is true)

Organization of `model.cpp`

```
class mu__int {
 enum {undef_value=0xff};
 bool in_world;        /* local iff false */
 int lb, ub;           /* bounds */
 int byteOffset;       /* in bytes */
 /* points to workingstate->bits[byteOffset]
    for global variables, to cvalue for
     local
 */
 unsigned char *valptr;
 unsigned char cvalue;
```

Organization of `model.cpp`

```
public:
 /* constructor, sets all attributes (the
    variable is supposed to be local by
    default, with an undefined value);
    byteOffset is computed by generate_code
 */
 mu__int(int lb, int ub, int size, char *n,
         int byteOffset);
 /* other useful functions */
 int operator= (int val) {
  if (val <= ub && val >= lb) value(val);
  else boundary_error(val);
  return val;
 }
```

Organization of model.cpp

```
operator int() const {
 if (isundefined()) return undef_error();
 return value();
};
const int value() const {return *valptr;};
int value(int val) {
 *valptr = val; return val;};
void to_state(state *thestate) {
 /* used to make the variable global */
 in_world = TRUE;
 valptr = (unsigned char *)&(workingstate->
  bits[byteOffset]);
};
};
```

Organization of `model.cpp`

- As for the `byteOffset` computation, `program::generate_code` simply computes the one for a variable `mu_v` mapping a Murphi variable `v` in the following way
  - Let $M_1, \ldots, M_n$ be the upper bounds of the n variables preceeding the declaration of `v`
  - Let $b(x) = \lfloor \log_2(x+1) \rfloor + 1$ be the number of bits required to represent the maximum value x (plus the undefined value)
  - Let $B(x) = 1$ if $b(x) \leq 8$, 4 otherwise (i.e. only 1-byte or 4-bytes integers may be used)
  - Then, $\texttt{byteOffset}(\texttt{mu\_v}) = \sum_{i=1}^{n} B(M_i)$

Organization of `model.cpp`: `workingstate`

- Structure containing the current global state, is an instance of class `state`
- Essentially, it consists of an array of unsigned characters, named `bits`
  - so that any value of any global variable may be casted inside it
  - at a precise location, pointed to by `valptr` from `mu__int`
- Note that `workingstate` has a fixed length, that is $\texttt{BLOCKS\_IN\_WORLD} = \sum_{i=1}^{N} B(M_i)$
  - being N the number of all global variables
  - namely, `bits` has `BLOCKS_IN_WORLD` unsigned chars

UNIVERSITÀ DEGLI STUDI DELL'AQUILA
DISIM Dipartimento di Ingegneria e Scienze dell'Informazione e Matematica

Translation of Murphi Model Statements

- Straightforward for `ifs`, `whiles` and so on: the "difficult" part is assignments (and expressions evaluation)
- Essentially, `a := b;` in `model.m` becomes `mu_a = (mu_b);` in `model.cpp`
- The operator `()` is redefined so that `mu_b` retrieves the value for `b`, either from itself (attribute `cvalue`) or from `workingstate` (thanks to `valptr`)
- Then, the redefined operator = is called, so that `mu_a` updates the value for a to be equal to that of `b`, either from itself (attribute `cvalue`) or from `workingstate`
- If the right side of the assignment has a generic expression, it is evaluated in a similar way (the operator `()` solves the Murphi variable references, the other values will be integer constants or function calls...)
- BTW, functions are mapped as C++ methods...

Translation of Murphi Rules

- For each rule *i* (starting from 0 at the *end* of `model.m`!) there is a class named `RuleBase`*i*
- Such class has `Code` method for the body and `Condition` method for the guard
- Startstates are similar, but they only have the body
- A suitable C++ code flattens rulesets, if present

Translation of Murphi Rules: From This...

```
Const VAL_LIM: 5;

Type val_t : 0..VAL_LIM;

Var v : val_t;

Rule "incBy1"
 v <= VAL_LIM - 1 ==>
 Var useless : val_t;
 Begin
  useless := 1;
  v := v + useless;
 End;
```

Translation of Murphi Rules: ... To This

```
class RuleBase1 {
public:
 ⋮
 bool Condition(unsigned r) { /* guard */
  return (mu_v) <= (4);
 }
 ⋮
 void Code(unsigned r) {       /* body */
  mu_1_val_t mu_useless("useless", 0);
  mu_useless = 1;
  mu_v = (mu_v) + (mu_useless);
 };
 ⋮
}
```

UNIVERSITÀ DEGLI STUDI DELL'AQUILA

DISIM Dipartimento di Ingegneria e Scienze dell'Informazione e Matematica

Translation of Murphi Rules: From This...

```
ruleset i: l1..u1 do
 ruleset j: l2..u2 do
  Rule "incBy1"
   i < j ==>
   Begin v := v + i - j; End;
Endruleset; Endruleset;
```

Translation of Murphi Rules: ... To This

```
class RuleBase0 {
public:
 bool Condition(unsigned r) {
  /* called (u_1 - l_1 + 1)(u_2 - l_2 + 1) with r ranging
     from 0 to (u_1 - l_1 + 1)(u_2 - l_2 + 1) - 1 */
  static mu__subrange_7 mu_j;
  mu_j.value((r % (u_2 - l_2 + 1)) + l_2);
  r = r / (u_2 - l_2 + 1);
  static mu__subrange_6 mu_i;
  mu_i.value((r % (u_1 - l_1 + 1)) + l_1);
  /* useless, but it is automatically
     generated... */
  r = r / (u_1 - l_1 + 1);
  return (mu_i) < (mu_j);
 }
```

UNIVERSITÀ DEGLI STUDI DELL'AQUILA
DISIM Dipartimento di Ingegneria e Scienze dell'Informazione e Matematica

Translation of Murphi Rules: ... To This

```
void Code(unsigned r) {
  static mu__subrange_7 mu_j;
  mu_j.value((r % ($u_2 - l_2 + 1$)) + $l_2$);
  r = r / ($u_2 - l_2 + 1$);
  static mu__subrange_6 mu_i;
  mu_i.value((r % ($u_1 - l_1 + 1$)) + $l_1$);
  r = r / ($u_1 - l_1 + 1$);
  mu_v = ((mu_v) + (mu_i)) - (mu_j);
 };
 ⋮
};
```

Translation of Murphi Rules: ... To This

```
void Code(unsigned r) {
 static mu__subrange_7 mu_j;
 mu_j.value((r % (u_2 - l_2 + 1)) + l_2);
 r = r / (u_2 - l_2 + 1);
 static mu__subrange_6 mu_i;
 mu_i.value((r % (u_1 - l_1 + 1)) + l_1);
 r = r / (u_1 - l_1 + 1);
 mu_v = ((mu_v) + (mu_i)) - (mu_j);
};
 ⋮
};
```

Murphi Overall Translation

- Note that the first part of `Condition` and `Code` is meant to translate an integer from 0 to $(u_1 - l_1 + 1)(u_2 - l_2 + 1) - 1$ in 2 values for the rulesets indeces
- The interface class for the verification algorithm is `NextStateGenerator`
- Suppose there are R rules $r_0, \ldots, r_{R-1}$, and that each r_i is contained in N_i nested rulesets having upper bound u_{ij} and lower bound l_{ij}, for $j = 1, \ldots, N_i$
- Note that `Condition` simply calls its homonymous method of the RuleBase class corresponding the current `r`...

Murphi Overall Translation

Let $P(k) = \sum_{i=0}^{k-1}(\prod_{j=1}^{N_i}(u_{ij} - l_{ij} + 1)) + 1$ be the number of flattened rules preceding the rule r_k;

```
class NextStateGenerator {
 RuleBase0 R0;
 ⋮
 RuleBase(R−1) R(R−1);
public:
 void SetNextEnabledRule(unsigned &
  what_rule);
```

Murphi Overall Translation

```
bool Condition(unsigned r) { /* r will
 range from 0 to P(R) */
  category = CONDITION;
  if (what_rule < P(1))
   return R0.Condition(r - 0);
  if (what_rule >= P(1) && what_rule < P(2))
   return R1.Condition(r - P(1));
  ⋮
  if (what_rule >= P(R−1) && what_rule <
   P(R))
   return R(R−1).Condition(r - P(R−1));
  return Error;
}
```

Murphi Overall Translation

```
void Code(unsigned r) {
  if (what_rule < P(1)) {
   R0.Code(r - 0); return;
  }
  if (what_rule >= P(1) && what_rule < P(2)) {
   R1.Code(r - P(1)); return;
  }
  ⋮
  if (what_rule >= P(R−1) && what_rule <
   P(R)) {
   R(R−1).Code(r - P(R−1)); return;
  } }
};
const unsigned numrules = P(R);
```

UNIVERSITÀ DEGLI STUDI DELL'AQUILA

DISIM Dipartimento di Ingegneria e Scienze dell'Informazione e Matematica

Step 2: What Is Actually Compiled by C++ Compiler

Concatenation of include/*.h
model.cpp
Concatenation of include/*.C

Murphi BFS

```
FIFO_Queue Q;
HashTable T;

bool BFS(NFSS 𝒩, AP φ)
{
 let 𝒩 = (S, I, Post);
 foreach s in I {
  if (!φ(s))
   return false;
 }
 foreach s in I
  Enqueue(Q, s);
 foreach s in I
  HashInsert(T, s);
```

Murphi BFS

```
while (Q ≠ ∅) {
 s = Dequeue(Q);
 foreach s_next in Post(s) {
  if (!φ(s_next))
   return false;
  if (s_next is not in T) {
   Enqueue(Q, s_next);
   HashInsert(T, s_next);
  } /* if */ } /* foreach */ } /* while */
return true;
}
```

BFS in Murphi

- `Post(s)` is computed using class `NextStateGenerator`
- It is equivalent to a `for` loop on all flattened rules
- For each flattened rule index r, `Condition(`r`)` tells if the current state `workingstate` enables the guard of r
- If so, the next state is obtained via `Code(`r`)`, by directly modifying `workingstate`

Hashtable in Murphi

- Open addressing ...
  - insert: repeatedly call $e = h(s, i)$ (for $i = 1, 2, \ldots$) till $\texttt{T}[e] = \varnothing$, then insert s in $\texttt{T}[e]$
  - search: repeatedly call $e = h(s, i)$ (for $i = 1, 2, \ldots$) till either:
    - $\texttt{T}[e] = \varnothing \to s$ is not present
    - $\texttt{T}[e] = s \to s$ is present
- ... with double hasing
  - there are two hash functions h_1, h_2
  - $h(s, i) = (h_1(s) + ih_2(s)) \bmod m$
  - m is the size of T, and is a prime number

Reducing Hashtable in Murphi

- States must be stored in `T`
- For efficiency reasons, `T` is a fixed-length array, each entry is an instance of `state` class
  - if `T` becomes full, the verification is terminated and you have to run it again with more memory
  - option `-m` of `model` executable
- Thus, `T` stores workingstates
- Two possible ways (also together):
  1. use less memory for each state
  2. store less states

Hash Compaction

- Enabled by compiling the Murphi model with -c
- When dealing with hash table insertions and searches, state "signatures" are used instead of the whole states
- The idea is that it is unlikely to happen that two different states have the same signature
- If this happens, some states may be never reached, even if they are indeed reachable
- Thus, there may be "false positives": the verification terminates with an OK messages, while the system was buggy instead
- However, this is very unlikely to happen, and in every case it is much better than testing, which may miss whole classes of bugs

UNIVERSITÀ DEGLI STUDI DELL'AQUILA

DISIM Dipartimento di Ingegneria e Scienze dell'Informazione e Matematica

Hash Compaction

- At the beginning of the verification, a vector `hashmatrix` of 24*`BLOCKS_IN_WORLD` longs (4 byte per each long) is created and initialized with *random* values (`hashmatrix` will never be modified)
- Then, given a state s to be sought/inserted, 3 longs `l0`, `l1` and `l2` are computed from `hashmatrix`
- Namely, `l`i, for $i = 0, 1, 2$, is the bit-to-bit xor of the longs in the set $H(i) = \{\texttt{hashmatrix}[3k + i] \mid \text{the } k\text{-th bit of the uncompressed state } s \text{ is } 1\}$;
- That is to say, every bit of s is used to determine if a given element of `hashmatrix` has or hasn't to be used in the signature computation

Hash Compaction

- This is accomplished in the functions of file `include/mu_hash.cpp`, where to avoid to compute `8*BLOCKS_IN_WORLD` bit-to-bit xor operations, some xor properties allow to use the preceeding computed signature and save some xor computation (`oldvec` variable)
- Then, `l0` is used as a hash value (index in the hash table)
- The concatenation of `l1` and `l2` (truncated to a given number of bits by option `-b`) gives the signature (the value to be sought/inserted in `T`)
- It should be obvious, now, that a signature cannot be used to generate states, so that's why `Q` entries do not point to hash table entries any more
- Thus, if current `workingstate` state is found to be new, and so its signature is put inside the hash table, a new memory block is allocated to be assigned to the current from of the queue, and `workingstate` is copied into that

Bit Compression

- To save some (not much...) space, the Murphi compiler option `-b` may be used to compress states (*bit compression* in SPIN's parlance)
- Whilst hashcompaction is a lossy compression, this is lossless
- But very less efficient
- In this way, `workingstate` contents are not forced to be aligned to byte boundaries, so it occupies less space
- Moreover, effective subranges size is used (remember we store the lower bound...)
- Of course, a more complex handling than the `valptr` and `byteOffset` one has to be used

Murphi BFS

```
Var
 x : 255..261;
 y : 30..53;

StartState
 x := 256;
 y := 53;
End;
```

Bit Compression

				y
0x0	0x0	0x1	0x0	0x35

workingstate−>bits
without −b

x	y
0xc	0x2

workingstate−>bits
with −b

Symmetry and Multiset Reductions

- Differently from SPIN's partial order reduction, these techniques are not transparent to the user
- In fact, symmetry reduction are applicable only if some types have been declared using the `scalarset` keyword (for multiset reduction, the keyword is `multiset`)
- Not all systems are symmetric
- However, when it is possible to apply symmetry reduction, only a subset of the state space is (safely) explored
- To be more precise, symmetry reduction induces a partition of the state space in equivalence classes
- A functions chain (implemented in the model-dependent part in `model.cpp`) is able to return the representative of the equivalence class of a given state

Symmetry and Multiset Reductions

- Rules for scalarset:
  - the values are not used in any comparison operation except equality testing
  - the values are not used in any arithmetic operation
  - the result from the for loop with the subrange as index does not depend on the order of the iteration
  - cannot be directly assigned to some value: either it is used on a forall, exists, for, ruleset, or it is used an assignment with some other scalarset value

Murphi BFS with Symmetry Reduction

```
FIFO_Queue Q;
HashTable T;

bool BFS(NFSS 𝒩, AP φ)
{
 let 𝒩 = (S, I, Post);
 foreach ss in I {
  s = Normalize(ss);
  if (!φ(s))
   return false;
 }
 foreach s in I
  Enqueue(Q, s);
 foreach s in I
  HashInsert(T, s);
```

Murphi BFS with Symmetry Reduction

```
while (Q ≠ ∅) {
 s = Dequeue(Q);
 foreach ss_next in Post(s) {
  s_next = Normalize(ss_next);
  if (!φ(s_next))
   return false;
  if (s_next is not in T) {
   Enqueue(Q, s_next);
   HashInsert(T, s_next);
  } /* if */ } /* foreach */ } /* while */
return true;
}
```

UNIVERSITÀ DEGLI STUDI DELL'AQUILA

DISIM Dipartimento di Ingegneria e Scienze dell'Informazione e Matematica

Symmetry Reduction

- How is `Normalize` implemented? Here are the main ideas
- Suppose that variable v is a `scalarset(N)`, and $v = \tilde{v}$ in a state $s \in S$
- Then, any *permutation* of the set $\{1, \ldots, N\}$ brings to an *equivalent* state
- Thus, all possible permutations are generated, and the lexicographically smaller state is chosen as the representative
  - apply a permutation means: change the value of v, and reorder any array or ruleset or for which depends on v
- Could be expensive, heuristics are also used to perform faster but potentially not complete normalizations
  - i.e., two symmetric states may be declared different
  - this does not hinders verification correctness, only its efficiency