Formal Methods in Software Development
Resume of the 21/10/2020 lesson

Igor Melatti and Ivano Salvo

Murphi Model Checker: Behind the Hood

- We have seen its input language syntax and semantics, now comes the verification algorithm
- Murphi needs 3 steps in order to verify or simulate a system
- Assume you have described your system $\mathcal{S}$ in the file `model.m`
  1. the Murphi compiler (`src/mu`) is invoked on `model.m` and outputs a `model.cpp` file (unless there are errors...)
  2. the file `model.cpp` is compiled with a C++ compiler, giving the directory `include/` as additional include directory; in this way, an executable file `model` is obtained
  3. just execute `model` (with option `-s` for simulation; option `-h` gives an overview of all possible options)
- Most important step 1 compilation options:

  -c enables hash compaction (see the verification part; typically combined with `-b`)

  -b values not aligned on bytes in the current state (verification will require less space, slightly more execution time)
- Step 1 is accomplished by means of the 25 files in the `src/` directory
- Standard compiler implementation, with Flex lexical analyzer (`mu.l`) and Yacc parser (`mu.y`)
- The main function, i.e. the one building `model.cpp`, is `program::generate_code` in `cpp_code.cpp` (called by `main`, in `mu.cpp`)
- In short, `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`, that as a consequence will contain the model-specific code

- Let's have a closer look to `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 (as in Figs. 2 and 4), `mu_v` retrieves the value from a fixed-address structure containing the current state value (`workingstate`; `in_world` is true)
- The main elements of class `mu__int` are listed in Fig. 1
- 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, `byteOffset(mu_v)` $= \sum_{i=1}^{n} B(M_i)$
- Note that `workingstate` has a fixed length, that is `BLOCKS_IN_WORLD` $= \sum_{i=1}^{N} B(M_i)$, being N the number of all global variables; namely, the `bits` attribute of the class `state` (of which `workingstate` is an instance) has `BLOCKS_IN_WORLD` unsigned chars
- We are now ready to have a glance at the Murphi assignment mapping in C++
  - As an example, `a := b;` becomes `mu_a = (mu_b);`
  - 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`

```
class mu__int {
 enum {undef_value=0xff};
 bool in_world;           /* local (false) or global (true) */
 int lb, ub;              /* upper and lower bound */
 int byteOffset;          /* in bytes */
 /* valptr points to workingstate->bits[byteOffset] for global
    variables, and to cvalue for local var */
 unsigned char *valptr;
 unsigned char cvalue;    /* value for local variables */
public:
 /* constructor, sets all the attributes (the variable is
    supposed to be local by default, with an undefined value);
    byteOffset is given as a parameter, so it 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;
 }
 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 defined(bool val) {if (!val) *valptr = undef_value;};
 bool defined() const {return (*valptr != undef_value);};
 void undefined() {*valptr = undef_value;};
 bool isundefined() const {return (*valptr == undef_value);};
 void to_state(state *thestate) {
  /* used to make the variable global */
  in_world = TRUE;
  valptr = (unsigned char *)&(workingstate->bits[byteOffset]);
 };
};
```

Figure 1: Class `mu__int` (from `include mu_util.h`)

- 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...

- We can now look at the translation of rules
- For each rule i (starting from 0 at the *end* of `model.m`!) there is a class named `RuleBase`i
- Example: the Murphi code in Fig. 2 is translated in the C++ code in Fig. 3
- Another example (with rulesets): the Murphi code in Fig. 4 is translated in the C++ code in Fig. 5
- 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 $j = 1, \ldots, N_i$
- Then, class `NextStateGenerator` is shown in Fig. 6
- Note that `Condition` simply calls its homonymous method of the RuleBase class corresponding the current `r`...
- Step 2 will compile the file in Fig. 7
- Step 3 will execute the result of the compilation of the file in Fig. 7
- Fig. 8 show how simulation is carried out
- Not very useful, SPIN simulation is much butter
- Fig. 9 show how verification is carried out
  - `next(s)` is computed using class `NextStateGenerator` from Figure 6
  - if 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`
  - try to follow all the code on the graph at Figure 10

```
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;
```

Figure 2: A Murphi rule

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

Figure 3: Translation of the Murphi rule in Fig. 2

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

Figure 4: A Murphi ruleset

```
class RuleBase0 {
public:
 bool Condition(unsigned r) {
  /* Condition will be called (u1 − l1 + 1)(u2 − l2 + 1) times for each
     state to be expanded (indeed, NextRule() is called, but
     it has nearly the same code), with r ranging from 0 to
     (u1 − l1 + 1)(u2 − l2 + 1) − 1 */
  static mu__subrange_7 mu_j;
  mu_j.value((r % (u2 − l2 + 1)) + l2);
  r = r / (u2 − l2 + 1);
  static mu__subrange_6 mu_i;
  mu_i.value((r % (u1 − l1 + 1)) + l1);
  /* useless, but it is automatically generated... */
  r = r / (u1 − l1 + 1);
  return (mu_i) < (mu_j);
 }
 void Code(unsigned r) {
  static mu__subrange_7 mu_j;
  mu_j.value((r % (u2 − l2 + 1)) + l2);
  r = r / (u2 − l2 + 1);
  static mu__subrange_6 mu_i;
  mu_i.value((r % (u1 − l1 + 1)) + l1);
  r = r / (u1 − l1 + 1);
  mu_v = ((mu_v) + (mu_i)) - (mu_j);
 };
 ⋮
};
```

Figure 5: Translation of the Murphi ruleset in Fig. 4

```
Let P(k) = Σ_{i=0}^{k-1} (Π_{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);
 ⋮
 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;
 }
 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);
```

Figure 6: Class NextStateGenerator

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

Figure 7: The file compiled in the step 2

```
/* Make a random walk in the NFSS described by MD */
void Make_a_run(MurphiDescription MD, AP φ)
{
 pick at random an initial state s among the ones in MD;
 if (!φ(s))
  return with error message;
 s_current = s;
 while (1) { /* loop forever (unless an error occurs) */
  s_next = s_current;
  rules_tried = 0;
  while (s_next == s_current && rules_tried <
   num_rules_MD) {
   pick at random a rule r never tried before;
   if (the r guard is satisfied by s_current)
    s_next = execution of the r body on s_current;
   rules_tried++;
  } /* while */
  if (!φ(s_next))
   return with error message;
  s_current = s_next;
 } /* while */
} /* Make_a_run() */
```

Figure 8: Murphi simulation

```
FIFO_Queue Q;
HashTable T;

/* Returns true iff φ holds in all the reachable states
 */
bool BFS(NFSS 𝒮, AP φ)
{
 let 𝒮 = (S, I, 𝒜, next);

 /* is there an initial state which is an error state? */
 foreach s in I {
  if (!φ(s))
   /* error found, 𝒮 does not satisfy φ */
   return false;
 }

 /* load Q with initial states */
 foreach s in I Enqueue(Q, s);
 /* mark the initial states as visited */
 foreach s in I HashInsert(T, s);

 /* visit */
 while (Q ≠ ∅) {
  /* take from Q the state to be expanded */
  s = Dequeue(Q);
  /* s expansion */
  foreach (s_next, a) in next(s) {
   if (!φ(s_next))
    /* error found, 𝒮 does not satisfy φ */
    return false;

   if (s_next is not in T) {
    /* s_next must be eventually expanded */
    Enqueue(Q, s_next);
    /* mark s_next as visited */
    HashInsert(T, s_next);
   } /* if */ } /* foreach */ } /* while */
 /* here, Q is empty and T contains all the reachable
  states */
 /* error not found, 𝒮 satisfies φ */
 return true;
} /* BFS() */
```

Figure 9: Murphi verification

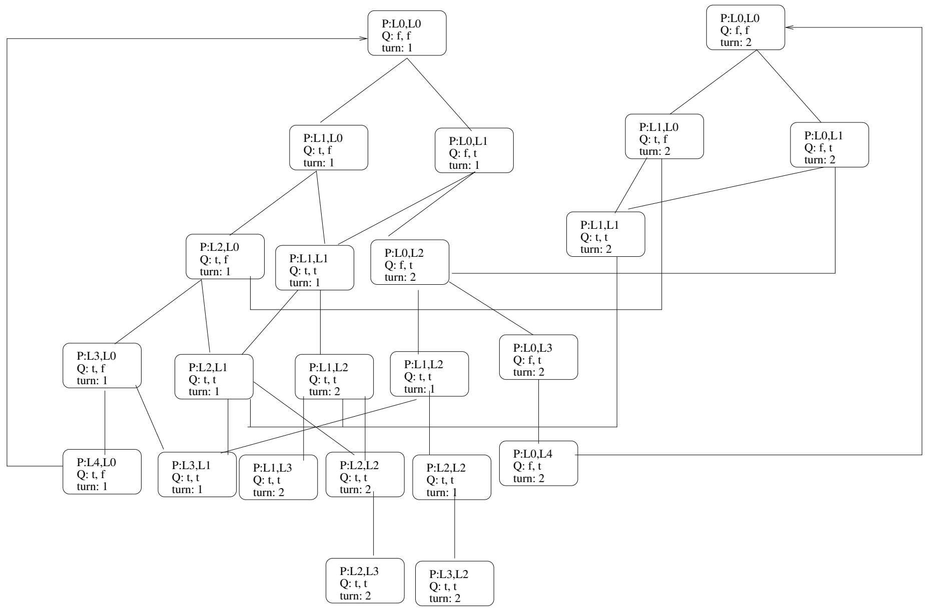

Figure 10: Part of the graph of Peterson's protocol (2 processes)

- The hash table `T` and the FIFOQueue `Q` is where state explosion strikes
- `Q` can be efficiently implemented with disk auxiliary storage, so it is not a problem
- Note that each `Q` entry is not a state, but a pointer to a state
- If hash compaction is not used, each entry of `Q` points to the slot in the hash table containing the desired state
- For `T`, one can try to use 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

- As for the hash compaction implementation, here is it
- 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 $H(i) = \{$`hashmatrix`$[3k + i] \mid$ the k-th bit of the uncompressed state s 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
- 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
- To save some (not much...) space, the Murphi compiler option `-b` may be used to compress states (*bit compression* in SPIN's parlance)
- 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...); see Figs. 11 and 12
- Of course, a more complex handling than the `valptr` and `byteOffset` one shown in Fig. 1 has to be used

• A trasversal technique is to use symmetry or multiset reduction

- 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`)

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

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

Figure 11: Murphi example for the bit compression

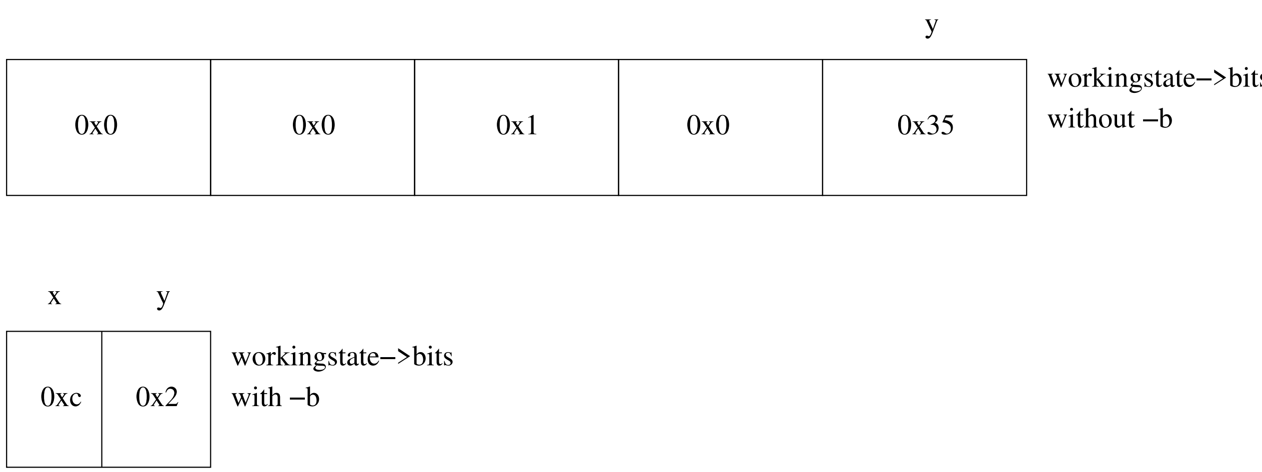

Figure 12: State occupation for Fig. 11 with and without bit compression

  - 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
  - Thus, only equivalence classes representatives are explored

- Most important verification options:

  -h prints all the options

  -bN N bits to be used for each signature in hash compaction

  -mN use exactly N MB of RAM for hash table ($0.9N$) and queue ($0.1N$); note however that with hash compaction more memory may be used

  -ndl disable deadlock detection

  -pN print progress reports after every 10^N states

1 Assignments

1. Find the corresponding code fragment of each instruction (or block of instructions) of Fig. 9 in the effective code in the `include/` directory of Murphi; if for some code fragments there is not an enough precise correspondence, point it out (in particular, explain how the current state expansion is effectively carried out)

2. Add to Fig. 9 the deadlock detection implemented in the effective code in the `include/` directory of Murphi

3. Add to Fig. 9 the symmetry reduction (only a couples of additional lines should be necessary, plus some rearrangments)

4. Write down the startstates analogous for Fig. 6 (hint: create a simple Murphi model and compile it...)

5. Modify the Murphi simulation so that

   (a) It stops after a given number N of transitions (N must be given via a new option `-simlim`N)

   (b) At each step it ask the user to choose between the currently enabled rules

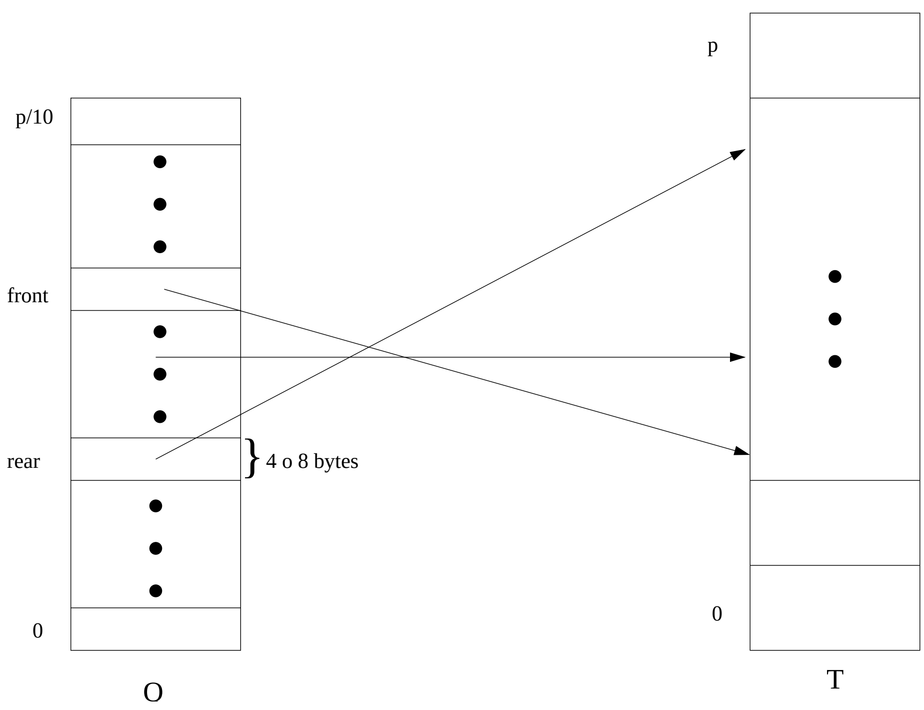

Figure 13: No compression option

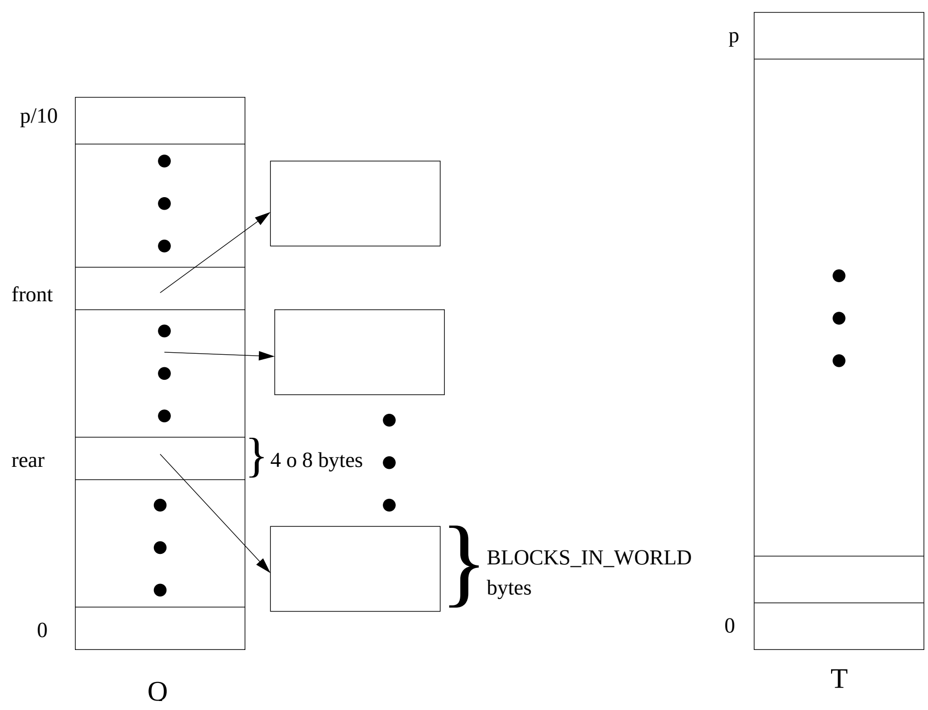

Figure 14: Murphi with options `-c` (hash compaction) only

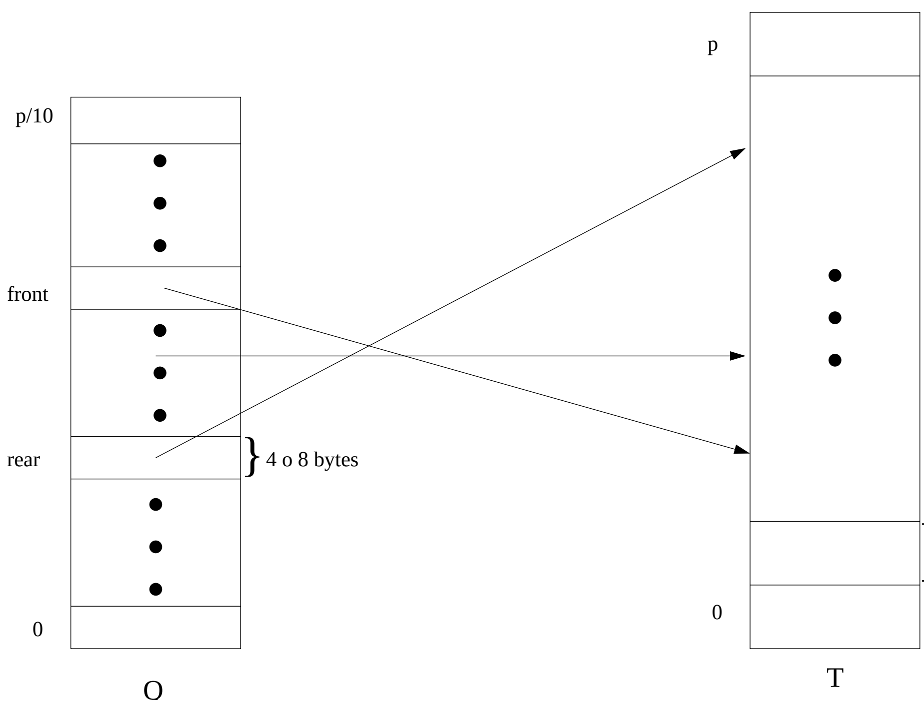

Figure 15: Murphi with options `-b` (bit compression) only

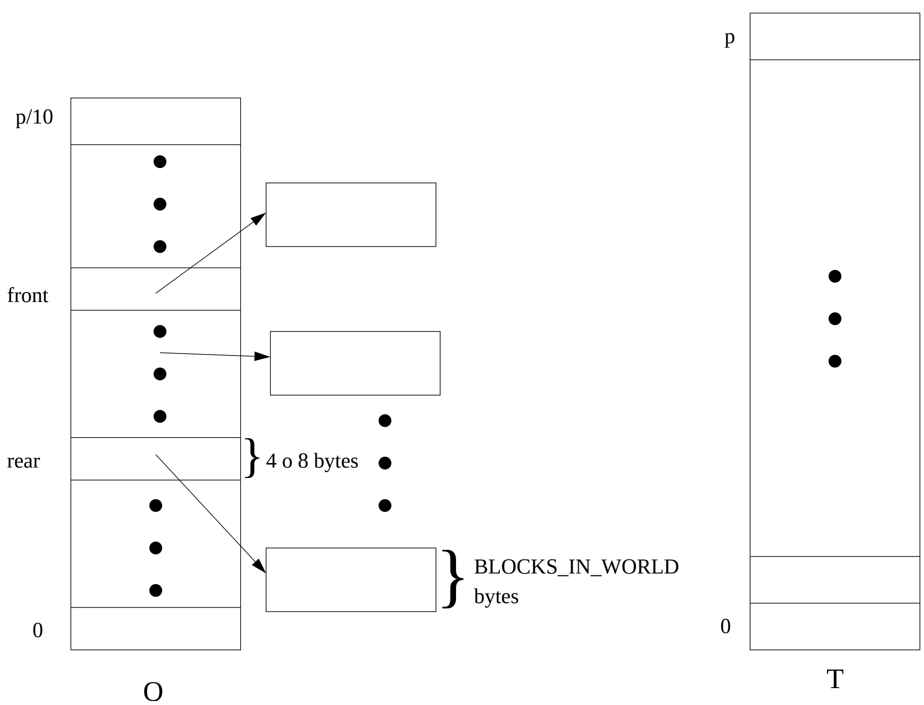

Figure 16: Murphi with both options `-c -b`