1*. Intro. This is a quick-and-dirty way to go from a slightly symbolic description of proposed mutual-exclusion algorithms to a corresponding set of clauses, so that I can use the clauses for bounded model checking.

In other words, I want to see whether the given concurrent algorithms can violate the mutex property by permitting simultaneous execution of two critical sections, or whether they can lead to livelock or starvation, in a given number of steps. To test this, I'll see if certain extensions of the clauses are satisfiable.

First I have to describe the input language. Each step/state of an algorithm is given a name, which begins with an uppercase letter and has at most four characters. Every shared variable is also given a number, which begins with a lowercase letter and has at most two characters.

Only four elementary kinds of primitive operations are permitted at each step:

1) Compute non-critically, then optionally go to step l. (Here l is a step name.)
2) Compute critically, then go to step l. (Likewise.)
3) Set $V \leftarrow v$, then goto l. (Here V is a shared variable and v is a constant.)
4) If $V = v$, goto l, else goto l'. (Likewise.)

These steps specify state transitions in an fairly obvious way; precise semantics will be explained later.

Here's a simple example of possible input:

```
~ separate locks
A0 maybe goto A1
A1 a=1 goto A2
A2 if b=1 goto A2 else A3
A3 critical goto A4
A4 a=0 goto A0
B0 maybe goto B1
B1 b=1 goto B2
B2 if a=1 goto B2 else B3
B3 critical goto B4
B4 b=0 goto B0
```

The first line, which begins with '~', is simply a comment that will be passed to the output file. It is followed by steps of types 1, 3, 4, 2, 3, 1, 3, 4, 2, 3, respectively. The shared variables are `a` and `b`. The concurrent occurrence of critical states should never occur.

(I do not claim that these programs solve the mutex problem; they simply provide an example.)

At present I assume that all step names begin with either A or B, and that all shared variables are Boolean. But those restrictions might well be lifted later, after I get some experience with this simpler scheme.

This version of the program does not try to find a violation of the mutual exclusion property. Instead, it tries to find a "starvation cycle," namely a cycle in which state X_k equals state X_0, for some $k \le r$, and for which the following conditions hold: (1) Both processes have been bumped at least once in the cycle. (2) At least one of the processes has executed neither a `maybe` command nor a `critical` command within the cycle.

Furthermore we assume that a set of lemmas is given on an auxiliary input file. These lemmas need not be invariant in the normal sense; but they must be true in every state that we allow. In particular, these lemmas will typically exclude states that are forbidden in a starvation cycle. (They might, for example, include the unit clause '`~A0`' if we are trying to starve process A.) Condition (2) is not actually enforced by this program; it's only enforced by the lemmas.

We don't assume that X_0 is an initial state. We only assume that it satisfies the given lemmas.

2* Here then is the basic outline of this program.

```
#define maxsteps 100   /* at most this many steps */
#define bufsize 1024   /* must exceed the length of the longest input line */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
  ⟨ Type definitions 4 ⟩;
  step state[maxsteps];   /* internal representation of the programs */
  char vars[maxsteps][2];   /* the distinct shared-variable names */
  int astep[maxsteps], bstep[maxsteps];   /* steps for processes A and B */
  int r;   /* command-line parameter, the number of time steps to emulate */
  FILE *lemma_file;
  char buf[bufsize];   /* input from stdin goes here */
  main(int argc, char *argv[])
  {
    register int i, j, k, m, n, t, ma, mb;
    ⟨ Process the command line 3* ⟩;
    ⟨ Parse the input into the state table 5 ⟩;
    for (t = 0; t < r; t++) ⟨ Generate the transitions from time t to time t + 1 17 ⟩;
    ⟨ Generate clauses that deal with the lemmas 25* ⟩;
    ⟨ Generate clauses to force a starvation cycle 27* ⟩;
  }
```

3* ⟨ Process the command line 3* ⟩ ≡

```
  if (argc ≠ 3 ∨ sscanf(argv[1], "%d", &r) ≠ 1) {
    fprintf(stderr, "Usage:␣%s␣r␣p␣foo.lemmas␣<␣foo.dat\n", argv[0]);
    exit(−1);
  }
  if (r ≤ 0) {
    fprintf(stderr, "Parameter␣r␣must␣be␣positive!\n");
    exit(−2);
  }
  lemma_file = fopen(argv[2], "r");
  if (¬lemma_file) {
    fprintf(stderr, "I␣can't␣open␣file␣`%s'␣for␣reading!\n", argv[2]);
    exit(−3);
  }
  printf("~␣sat-mutex-starve-lemmas␣%d␣%s\n", r, argv[2]);
```

This code is used in section 2*.

4. Every non-comment line of input is recorded in an abbreviated form.

```
⟨ Type definitions 4 ⟩ ≡
  typedef struct state_struct {
    char name[4], lab[4], elab[4];   /* the name of this step and its successors */
    char var[2];   /* the shared variable */
    char val;   /* its value */
    char crit;   /* is this a critical step? */
  } step;
```

This code is used in section 2*.

5. I don't attempt to provide much syntactic sugar for the user (since I expect to be the only user). If I need something fancier, I'll probably write a preprocessor to convert fancy output into the primitive form that is understood by this program.

```
⟨ Parse the input into the state table 5 ⟩ ≡
  for (m = n = ma = mb = 0; ; ) {
    if (¬fgets(buf, bufsize, stdin)) break;
    if (buf[0] ≡ '~') printf("%s", buf);
    else {
      char *curp = buf;

      if (m ≥ maxsteps) {
        fprintf(stderr, "Recompile␣me␣--␣I␣only␣have␣room␣for␣%d␣steps!\n", maxsteps);
        exit(−666);
      }
      ⟨ Scan the name field 6 ⟩;
      if (strncmp(curp, "maybe␣", 6) ≡ 0) ⟨ Scan a maybe step 7 ⟩
      else if (strncmp(curp, "critical␣", 9) ≡ 0) ⟨ Scan a critical step 8 ⟩
      else if (strncmp(curp, "if␣", 3) ≡ 0) ⟨ Scan an if step 10 ⟩
      else ⟨ Scan an assignment step 14 ⟩;
      m++;
    }
  }
  ⟨ Check for missing steps 15 ⟩;
  if (state[astep[0]].crit + state[bstep[0]].crit > 1) {
    fprintf(stderr, "Both␣processes␣are␣initially␣in␣critical␣sections!\n");
    exit(−555);
  }
  fprintf(stderr, "(%d+%d␣steps␣with␣%d␣shared␣variables␣successfully␣input)\n", ma, mb, n);
```

This code is used in section 2*.

6. #define abrt(m, t)
{ fprintf(stderr, "Oops,␣%s!\n>␣%s\n", m, buf); exit(t); }

```
⟨ Scan the name field 6 ⟩ ≡
  for (j = 0; *curp ∧ *curp ≠ '␣' ∧ *curp ≠ '\n'; j++, curp++)
    if (j < 4) state[m].name[j] = *curp;
  if (j > 4) abrt("the␣name␣is␣too␣long", −10);
  if (state[m].name[0] < 'A' ∨ state[m].name[0] > 'B')
    abrt("the␣step␣name␣must␣begin␣with␣A␣or␣B", −11);
  for (j = 0; j < m; j++)
    if (strncmp(state[j].name, state[m].name, 4) ≡ 0) abrt("that␣name␣has␣already␣been␣used", −12);
  if (state[m].name[0] ≡ 'A') astep[ma++] = m;
  else bstep[mb++] = m;
  if (*curp++ ≠ '␣') abrt("step␣is␣incomplete", −13);
```

This code is used in section 5.

7. ⟨ Scan a maybe step 7 ⟩ ≡

```
  {
    curp += 5;
    ⟨ Scan the lab field 9 ⟩;
    if (*curp ≠ '\n') abrt("maybe␣step␣ends␣badly", −14);
  }
```

This code is used in section 5.

8. ⟨ Scan a `critical` step 8 ⟩ ≡

```
{
  curp += 8;
  state[m].crit = 1;
  ⟨ Scan the lab field 9 ⟩;
  if (*curp ≠ '\n') abrt("critical␣step␣ends␣badly", −15);
}
```

This code is used in section 5.

9. ⟨ Scan the *lab* field 9 ⟩ ≡

```
if (strncmp(curp, "␣goto␣", 6) ≠ 0) abrt("missing␣goto", −16);
curp += 6;
for (j = 0; *curp ∧ *curp ≠ '␣' ∧ *curp ≠ '\n'; j++, curp++)
  if (j < 4) state[m].lab[j] = *curp;
if (j > 4) abrt("the␣label␣is␣too␣long", −17);
```

This code is used in sections 7, 8, 10, and 14.

10. ⟨ Scan an `if` step 10 ⟩ ≡

```
{
  curp += 3;
  ⟨ Scan the var field 11 ⟩;
  if (*curp++ ≠ '=') abrt("missing␣'='␣in␣an␣if␣step", −18);
  ⟨ Scan the val field 12 ⟩;
  ⟨ Scan the lab field 9 ⟩;
  ⟨ Scan the elab field 13 ⟩;
  if (*curp ≠ '\n') abrt("that␣if␣step␣ends␣badly", −19);
}
```

This code is used in section 5.

11. ⟨ Scan the *var* field 11 ⟩ ≡

```
for (j = 0; *curp ∧ *curp ≠ '=' ∧ *curp ≠ '\n'; j++, curp++)
  if (j < 2) state[m].var[j] = vars[n][j] = *curp;
if (j > 2) abrt("the␣variable␣name␣is␣too␣long", −20);
if (state[m].var[0] < 'a' ∨ state[m].var[0] > 'z')
  abrt("a␣variable␣name␣must␣begin␣with␣a␣lowercase␣letter", −21);
for (j = 0; j < n; j++)
  if (strncmp(vars[j], state[m].var, 2) ≡ 0) break;
if (j ≡ n) n++;
else vars[n][1] = 0;
```

This code is used in sections 10 and 14.

12. ⟨ Scan the *val* field 12 ⟩ ≡

```
if (*curp < '0' ∨ *curp > '1') abrt("the␣value␣must␣be␣0␣or␣1", −22);
state[m].val = *curp++ − '0';
```

This code is used in sections 10 and 14.

13. ⟨ Scan the *elab* field 13 ⟩ ≡

```
if (strncmp(curp, "␣else␣", 6) ≠ 0) abrt("missing␣else", −23);
curp += 6;
for (j = 0; ∗curp ∧ ∗curp ≠ '\n'; j++, curp++)
   if (j < 4) state[m].elab[j] = ∗curp;
if (j > 4) abrt("the␣else␣label␣is␣too␣long", −24);
```

This code is used in section 10.

14. ⟨ Scan an assignment step 14 ⟩ ≡

```
{
   ⟨ Scan the var field 11 ⟩;
   if (∗curp++ ≠ '=') abrt("missing␣'='␣in␣an␣assignment␣step", −25);
   ⟨ Scan the val field 12 ⟩;
   ⟨ Scan the lab field 9 ⟩;
   if (∗curp ≠ '\n') abrt("assignment␣step␣ends␣badly", −26);
}
```

This code is used in section 5.

15. ⟨ Check for missing steps 15 ⟩ ≡

```
if (ma ≡ 0) {
   fprintf(stderr, "There␣are␣no␣steps␣for␣process␣A!\n");
   exit(−99);
}
if (mb ≡ 0) {
   fprintf(stderr, "There␣are␣no␣steps␣for␣process␣B!\n");
   exit(−98);
}
for (k = t = 0; k < m; k++) {
   if (state[k].lab[0]) {
      for (j = 0; j < m; j++)
         if (strncmp(state[j].name, state[k].lab, 4) ≡ 0) break;
      if (j ≡ m) {
         fprintf(stderr, "Missing␣step␣%.4s!\n", state[k].lab);
         t++;
      }
   }
   if (state[k].elab[0]) {
      for (j = 0; j < m; j++)
         if (strncmp(state[j].name, state[k].elab, 4) ≡ 0) break;
      if (j ≡ m) {
         fprintf(stderr, "Missing␣step␣%.4s!\n", state[k].elab);
         t++;
      }
   }
}
if (t) exit(−30);
```

This code is used in section 5.

16. The generated clauses involve variables like '`2A1`', meaning that process A is in state `A1` at time 2; also variables like '`3b`', meaning that shared variable `b` is 1 (true) at time 3; also variables like '`1@`', meaning that process A took a turn at time 1. (The negations of these variables, namely `~2A1`, `~3b`, `~1@`, mean respectively that A is not in state `A1` at time 2, `b` is 0 (false) at time 3, and process B took a turn at time 1.)

At time 0, all shared variables are 0 and each process is in its first-mentioned state.

⟨ Generate the initial clauses 16 ⟩ ≡

```
{
  for (j = 0; j < n; j++) printf("~000%.2s\n", vars[j]);
  printf("000%.4s\n", state[astep[0]].name);
  for (j = 1; j < ma; j++) printf("~000%.4s\n", state[astep[j]].name);
  printf("000%.4s\n", state[bstep[0]].name);
  for (j = 1; j < mb; j++) printf("~000%.4s\n", state[bstep[j]].name);
}
```

17. Speaking of turns reminds me that I promised to define precise semantics.

At each time t one of the processes, chosen nondeterministically, is granted permission to take a turn, which means intuitively that it performs the step corresponding to its current state. We say that the selected process is "bumped."

Every process is in a unique state at time t. The state of a process remains the same at time $t+1$ if it's not bumped. But if it's bumped, the next state is (1) either the same or *lab*, nondeterministically, after a `maybe` step; (2) *lab* after a critical step or an assignment step; (2) either *lab* or *elab* after an `if` step, depending on whether or not the shared variable has the specified value.

The value of a shared variable at time $t + 1$ is the same as the value that it had at time t, unless the bumped process assigned another value to it. In particular, if two processes are trying to change the same shared variable, the bumped process changes it first.

When the bumped process executes an `if` statement at the same time as another process is trying to write the same variable, the other process does not influence the result of the `if`; the change it wants to make will have to wait. [This rule means that weaker algorithms can get by, but they need stronger (and presumably more expensive and/or slower) hardware support. I'm using this rule in all the early examples of mutex in TAOCP, because it is easier to explain; the harder rule can be considered later, after algorithms pass this simpler criterion.]

⟨ Generate the transitions from time t to time $t + 1$ 17 ⟩ ≡

```
{
  ⟨ Generate clauses to forbid nonunique states for A at time t + 1 18 ⟩;
  ⟨ Generate clauses to forbid nonunique states for B at time t + 1 19 ⟩;
  ⟨ Generate the state transition clauses when A is bumped 20 ⟩;
  ⟨ Generate the state transition clauses when B is bumped 22 ⟩;
  ⟨ Generate the variable transition clauses 24 ⟩;
}
```

This code is used in section 2*.

18. I introduce auxiliary variables here, using Heule's exclusion clauses, so that we don't have quadratic blowup when the programs are large.

```
#define printprevA()
        if (j) printf("%03d_A%d", t + 1, i − 1);
        else printf("~%03d%.4s", t + 1, state[astep[k − 1]].name);
```

```
⟨ Generate clauses to forbid nonunique states for A at time t + 1 18 ⟩ ≡
  k = ma;
  if (k > 1) {
    i = j = 0;
    if (k ≡ 2) printf("~%03d%.4s␣~%03d%.4s\n", t + 1, state[astep[0]].name, t + 1, state[astep[1]].name);
    while (k > 4) {
      printprevA();
      printf("␣~%03d%.4s\n", t + 1, state[astep[k − 2]].name);
      printprevA();
      printf("␣~%03d%.4s\n", t + 1, state[astep[k − 3]].name);
      printprevA();
      printf("␣~%03d_A%d\n", t + 1, i);
      printf("~%03d%.4s␣~%03d%.4s\n", t + 1, state[astep[k − 2]].name, t + 1, state[astep[k − 3]].name);
      printf("~%03d%.4s␣~%03d_A%d\n", t + 1, state[astep[k − 2]].name, t + 1, i);
      printf("~%03d%.4s␣~%03d_A%d\n", t + 1, state[astep[k − 3]].name, t + 1, i);
      i++, j = 1, k −= 2;
    }
    printprevA();
    printf("␣~%03d%.4s\n", t + 1, state[astep[k − 2]].name);
    printprevA();
    printf("␣~%03d%.4s\n", t + 1, state[astep[k − 3]].name);
    printf("~%03d%.4s␣~%03d%.4s\n", t + 1, state[astep[k − 2]].name, t + 1, state[astep[k − 3]].name);
    if (k > 3) {
      printprevA();
      printf("␣~%03d%.4s\n", t + 1, state[astep[k − 4]].name);
      printf("~%03d%.4s␣~%03d%.4s\n", t + 1, state[astep[k − 2]].name, t + 1, state[astep[k − 4]].name);
      printf("~%03d%.4s␣~%03d%.4s\n", t + 1, state[astep[k − 3]].name, t + 1, state[astep[k − 4]].name);
    }
  }
```

This code is used in section 17.

19. **#define** *printprevB*()

```
    if (j) printf("%03d_B%d", t + 1, i − 1);
    else printf("~%03d%.4s", t + 1, state[bstep[k − 1]].name);
```

⟨ Generate clauses to forbid nonunique states for B at time $t + 1$ 19 ⟩ ≡

```
  k = mb;
  if (k > 1) {
    i = j = 0;
    if (k ≡ 2) printf("~%03d%.4s␣~%03d%.4s\n", t + 1, state[bstep[0]].name, t + 1, state[bstep[1]].name);
    while (k > 4) {
      printprevB();
      printf("␣~%03d%.4s\n", t + 1, state[bstep[k − 2]].name);
      printprevB();
      printf("␣~%03d%.4s\n", t + 1, state[bstep[k − 3]].name);
      printprevB();
      printf("␣~%03d_B%d\n", t + 1, i);
      printf("~%03d%.4s␣~%03d%.4s\n", t + 1, state[bstep[k − 2]].name, t + 1, state[bstep[k − 3]].name);
      printf("~%03d%.4s␣~%03d_B%d\n", t + 1, state[bstep[k − 2]].name, t + 1, i);
      printf("~%03d%.4s␣~%03d_B%d\n", t + 1, state[bstep[k − 3]].name, t + 1, i);
      i++, j = 1, k −= 2;
    }
    printprevB();
    printf("␣~%03d%.4s\n", t + 1, state[bstep[k − 2]].name);
    printprevB();
    printf("␣~%03d%.4s\n", t + 1, state[bstep[k − 3]].name);
    printf("~%03d%.4s␣~%03d%.4s\n", t + 1, state[bstep[k − 2]].name, t + 1, state[bstep[k − 3]].name);
    if (k > 3) {
      printprevB();
      printf("␣~%03d%.4s\n", t + 1, state[bstep[k − 4]].name);
      printf("~%03d%.4s␣~%03d%.4s\n", t + 1, state[bstep[k − 2]].name, t + 1, state[bstep[k − 4]].name);
      printf("~%03d%.4s␣~%03d%.4s\n", t + 1, state[bstep[k − 3]].name, t + 1, state[bstep[k − 4]].name);
    }
  }
```

This code is used in section 17.

20. **#define** *tprime* $(t + 1)$

⟨ Generate the state transition clauses when A is bumped 20 ⟩ ≡

```
  for (k = 0; k < ma; k++) {
    printf("%03d@␣~%03d%.4s␣%03d%.4s\n", t, t, state[astep[k]].name, tprime, state[astep[k]].name);
    if (state[astep[k]].var[0] ≡ 0) {
      if (state[astep[k]].crit ≡ 0)      /* a maybe step */
        printf("~%03d@␣~%03d%.4s␣%03d%.4s␣%03d%.4s\n", t, t, state[astep[k]].name, tprime,
              state[astep[k]].name, tprime, state[astep[k]].lab);
      else printf("~%03d@␣~%03d%.4s␣%03d%.4s\n", t, t, state[astep[k]].name, tprime, state[astep[k]].lab);
            /* a critical step */
    } else if (state[astep[k]].elab[0] ≡ 0) {      /* an assignment step */
      printf("~%03d@␣~%03d%.4s␣%03d%.4s\n", t, t, state[astep[k]].name, tprime, state[astep[k]].lab);
    } else ⟨ Generate clauses for when A is bumped in an if step 21 ⟩;
  }
```

This code is used in section 17.

21. ⟨ Generate clauses for when A is bumped in an `if` step 21 ⟩ ≡

```
{
  printf("~%03d@␣~%03d%.4s", t, t, state[astep[k]].name);
  printf("␣%s%03d%.2s␣%03d%.4s\n", state[astep[k]].val ? "~" : "", t, state[astep[k]].var, tprime,
      state[astep[k]].lab);
  printf("~%03d@␣~%03d%.4s", t, t, state[astep[k]].name);
  printf("␣%s%03d%.2s␣%03d%.4s\n", state[astep[k]].val ? "" : "~", t, state[astep[k]].var, tprime,
      state[astep[k]].elab);
}
```

This code is used in section 20.

22. ⟨ Generate the state transition clauses when B is bumped 22 ⟩ ≡

```
for (k = 0; k < mb; k++) {
  printf("~%03d@␣~%03d%.4s␣%03d%.4s\n", t, t, state[bstep[k]].name, tprime, state[bstep[k]].name);
  if (state[bstep[k]].var[0] ≡ 0) {
    if (state[bstep[k]].crit ≡ 0)      /* a maybe step */
      printf("%03d@␣~%03d%.4s␣%03d%.4s␣%03d%.4s\n", t, t, state[bstep[k]].name, tprime,
          state[bstep[k]].name, tprime, state[bstep[k]].lab);
    else printf("%03d@␣~%03d%.4s␣%03d%.4s\n", t, t, state[bstep[k]].name, tprime, state[bstep[k]].lab);
        /* a critical step */
  } else if (state[bstep[k]].elab[0] ≡ 0) {      /* an assignment step */
    printf("%03d@␣~%03d%.4s␣%03d%.4s\n", t, t, state[bstep[k]].name, tprime, state[bstep[k]].lab);
  } else ⟨ Generate clauses for when B is bumped in an if step 23 ⟩;
}
```

This code is used in section 17.

23. ⟨ Generate clauses for when B is bumped in an `if` step 23 ⟩ ≡

```
{
  printf("%03d@␣~%03d%.4s", t, t, state[bstep[k]].name);
  printf("␣%s%03d%.2s␣%03d%.4s\n", state[bstep[k]].val ? "~" : "", t, state[bstep[k]].var, tprime,
      state[bstep[k]].lab);
  printf("%03d@␣~%03d%.4s", t, t, state[bstep[k]].name);
  printf("␣%s%03d%.2s␣%03d%.4s\n", state[bstep[k]].val ? "" : "~", t, state[bstep[k]].var, tprime,
      state[bstep[k]].elab);
}
```

This code is used in section 22.

24. ⟨ Generate the variable transition clauses 24 ⟩ ≡

```
for (k = 0; k < n; k++) {      /* first consider all cases where the value changes */
  for (j = 0; j < m; j++)
    if (strncmp(state[j].var, vars[k], 2) ≡ 0 ∧ state[j].elab[0] ≡ 0)
      printf("%s%03d@␣~%03d%.4s␣%s%03d%.2s\n", state[j].name[0] ≡ 'A' ? "~" : "", t, t,
          state[j].name, state[j].val ≡ 0 ? "~" : "", tprime, state[j].var);
        /* now consider all cases where the value doesn't change */
  printf("~%03d@␣%03d%.2s", t, t, vars[k]);      /* A bumped and val is 0 */
  for (j = 0; j < m; j++)
    if (strncmp(state[j].var, vars[k], 2) ≡ 0 ∧ state[j].elab[0] ≡ 0 ∧ state[j].name[0] ≡ 'A')
      printf("␣%03d%.4s", t, state[j].name);      /* not changed by A */
  printf("␣~%03d%.2s\n", tprime, vars[k]);      /* it stays 0 */
  printf("%03d@␣%03d%.2s", t, t, vars[k]);      /* B bumped and val is 0 */
  for (j = 0; j < m; j++)
    if (strncmp(state[j].var, vars[k], 2) ≡ 0 ∧ state[j].elab[0] ≡ 0 ∧ state[j].name[0] ≡ 'B')
      printf("␣%03d%.4s", t, state[j].name);      /* not changed by B */
  printf("␣~%03d%.2s\n", tprime, vars[k]);      /* it stays 0 */
  printf("~%03d@␣~%03d%.2s", t, t, vars[k]);      /* A bumped and val is 1 */
  for (j = 0; j < m; j++)
    if (strncmp(state[j].var, vars[k], 2) ≡ 0 ∧ state[j].elab[0] ≡ 0 ∧ state[j].name[0] ≡ 'A')
      printf("␣%03d%.4s", t, state[j].name);      /* not changed by A */
  printf("␣%03d%.2s\n", tprime, vars[k]);      /* it stays 1 */
  printf("%03d@␣~%03d%.2s", t, t, vars[k]);      /* B bumped and val is 1 */
  for (j = 0; j < m; j++)
    if (strncmp(state[j].var, vars[k], 2) ≡ 0 ∧ state[j].elab[0] ≡ 0 ∧ state[j].name[0] ≡ 'B')
      printf("␣%03d%.4s", t, state[j].name);      /* not changed by B */
  printf("␣%03d%.2s\n", tprime, vars[k]);      /* it stays 1 */
}
```

This code is used in section 17.

25*. If the ith lemma is $l_1 \vee \cdots \vee l_k$, we essentially output the clauses $l_{t1} \vee \cdots \vee l_{tk}$ for $0 \le t < r$, plus the clauses $\neg$#$i \vee \neg l_{rj}$. The effect is to assert that this lemma is true until time r, but if #i holds then it fails at time r. Finally we assert that at least one lemma does fail at time r.

⟨ Generate clauses that deal with the lemmas 25* ⟩ ≡

```
for (i = 1; ; i++) {
  register char *p, *q;
  char hold;

  if (¬fgets(buf, bufsize, lemma_file)) break;
  for (t = 0; t ≤ r; t++) ⟨ Generate the clauses for Φ(t) 26* ⟩;
}
```

This code is used in section 2*.

26* ⟨ Generate the clauses for $\Phi(t)$ 26* ⟩ ≡

```
{
  for (p = buf; *p ≡ '␣'; p++) ;
  while (*p ≠ '\n') {
    if (*p ≡ '~') j = 1, p++; else j = 0;
    for (q = p; *q ≠ '␣' ∧ *q ≠ '\n'; q++) ;
    hold = *q, *q = '\0';
    printf("␣%s%03d%s", j ? "~" : "", t, p);
    p = q, *p = hold;
    for ( ; *p ≡ '␣'; p++) ;
  }
  printf("\n");
}
```

This code is used in section 25*.

27* Here we introduce auxiliary variables tU, tV, and tW, *meaningrespectively*'$@_0 \lor \cdots \lor @_{t-1}$', '$\overline{@_0} \lor \cdots \lor \overline{@_{t-1}}$', and 'the first t steps are a starvation cycle'.

⟨ Generate clauses to force a starvation cycle 27* ⟩ ≡

```
printf("~000U\n");
for (t = 1; t ≤ r; t++) printf("~%03dU␣%03dU␣%03d@\n", t, t − 1, t − 1);
printf("~000V\n");
for (t = 1; t ≤ r; t++) printf("~%03dV␣%03dV␣~%03d@\n", t, t − 1, t − 1);
for (t = 2; t ≤ r; t++) {
  printf("~%03dW␣%03dU\n", t, t);
  printf("~%03dW␣%03dV\n", t, t);
  for (j = 0; j < m; j++) {
    printf("~%03dW␣000%.4s␣~%03d%.4s\n", t, state[j].name, t, state[j].name);
    printf("~%03dW␣~000%.4s␣%03d%.4s\n", t, state[j].name, t, state[j].name);
  }
  for (j = 0; j < n; j++) {
    printf("~%03dW␣000%.2s␣~%03d%.2s\n", t, vars[j], t, vars[j]);
    printf("~%03dW␣~000%.2s␣%03d%.2s\n", t, vars[j], t, vars[j]);
  }
}
for (t = 2; t ≤ r; t++) printf("␣%03dW", t);
printf("\n");
```

This code is used in section 2*.

28.* Index.

The following sections were changed by the change file: 1, 2, 3, 25, 26, 27, 28.

SAT-MUTEX-STARVE-LEMMAS