SPIN VERSION 0 VALIDATOR SOURCE E

The program listings that follow are the program segments that are added to the simulator code described in Chapter 12 and listed in Appendix D. The code from this appendix is used to generate a protocol-specific validator for any protocol validation model that is described in PROMELA. The extensions are discussed in Chapter 13.

The new *makefile* for this version of SPIN looks as follows.

```
CC=cc           # ANSI C compiler
CFLAGS=-O       # optimizer
YFLAGS=-v -d -D # create y.output, y.debug, and y.tab.h
OFILES= spin.o lex.o sym.o vars.o main.o debug.o \
        mesg.o flow.o sched.o run.o pangen1.o pangen2.o \
        pangen3.o pangen4.o pangen5.o

spin:   $(OFILES)
        $(CC) $(CFLAGS) -o spin $(OFILES) -lm

%.o:    %.c spin.h
        $(CC) $(CFLAGS) -c $%.c

pangen1.o:      pangen1.c pangen1.h pangen3.h
pangen2.o:      pangen2.c pangen2.h
```

The remainder of this Appendix lists the contents of the 8 additional source files (see Table E.1). A large part of the code is contained in header files and copied into a protocol specific validator generated with SPIN.

Two pre-processor directives are generated for optional manipulation by the user. By default, all validators generated by SPIN perform an exhaustive search. If the name `BITSTATE` is defined at compile-time, this search strategy is replaced with a supertrace analysis (see Chapter 14 for examples). Similarly, by default there is no predefined maximum to the amount of memory that an exhaustive analysis can use. If, however, the name `MEMCNT` is defined at compile-time, it numeric value will be used to set an upper-bound. If, for instance, `MEMCNT=20` the upper-bound used is 2^{20} bytes (see also Chapter 14 for examples).

Table E.1 – Source File Index

File	Line Number
pangen1.c	1148
pangen1.h	1
pangen2.h	909
pangen2.c	1574
pangen3.c	2096
pangen3.h	1038
pangen4.c	2201
pangen5.c	2381

ONLINE VERSION OF SPIN

The source code listed in Appendices D and E of this book document the version 0 sources of SPIN. These sources were originally distributed only through AT&T's Toolchest software distribution system, for a fee. The most recent, extended, version of SPIN is available without fee for research and educational use from the web via SPIN's homepage:

```
http://netlib.bell-labs.com/netlib/spin/whatispin.html
```

More SPIN related information, about workshops, newsletters, and online documentation, is available through this page.

Table E.2 – Procedures Listed – Appendix E

Procedure	Line	Procedure	Line
any_proc(now)	2298	any_undo(now)	2281
blurb(fd, t, n)	2071	check_proc(now, m)	2308
d_eval_sub(s, pno, nst)	2501	do_init(sp)	1325
do_var(dowhat, s, sp)	1299	doglobal(dowhat)	1288
dolocal(dowhat, p, s)	1271	dumpskip(n, m)	2143
dumpsrc(n, m)	2167	end_labs(s, i)	1240
genaddproc()	1190	genaddqueue()	1475
genheader()	1171	genother(cnt)	1210
genunio()	2322	getweight(n)	2048
has_tau(n)	2060	huntele(f, o)	1403
huntstart(f)	1388	lost_trail()	2459
match_trail()	2395	ncases(fd, p, n, m, c)	1462
ntimes(fd, n, m, c)	1258	put_pinit(e, s, p, i)	1363
put_ptype(s, p, i, m0, m1)	1343	putnr(n)	2192
putstmnt(fd, now, m)	1825	typ2c(sp)	1432
undostmnt(now, m)	2214	walk_sub(e, pno, nst)	2469

Table E.3 – Procedures Explained – Chapter 13

Procedure	Page	Procedure	Page
addproc()	306	assert()	307
checkchan()	309	d_hash()	300
d_hash()	307	delproc()	306
endstate()	307	gensrc()	298
gensrc()	308	hstore()	306
huntini()	308	match_trail()	298
match_trail()	310	new_state()	306
new_state()	300	new_state()	300
new_state()	305	p_restor()	306
putproc()	308	putseq()	308
putstmnt()	308	putstmnt()	309
q_restor()	305	qrecv()	305
qsend()	305	r_ck()	307
retrans()	307	s_hash()	300
s_hash()	307	settable()	307
uerror()	300	uerror()	303
undostmnt()	308	undostmnt()	309
unrecv()	306	unsend()	306

```
/***** spin: pangen1.h *****/

char *Header[] = {
    "#define qptr(x)        (((uchar *)&now)+q_offset[x])",
    "#define pptr(x)        (((uchar *)&now)+proc_offset[x])",
    "#define Pptr(x)        ((proc_offset[x])?pptr(x):noptr)",
    "#define q_sz(x)        (((Q0 *)qptr(x))->Qlen)\n",
    "#define MAXQ           255",
    "#define MAXPROC        255",
    "#define WS             sizeof(long)   /* word size in bytes */",
    "#ifndef VECTORSZ",
    "#define VECTORSZ       1024           /* sv   size in bytes */",
    "#endif",
    "extern char *malloc(), *memcpy(), *memset();",
    "extern void exit();",
    "extern int abort();\n",
    "typedef struct Stack  {         /* for queues and processes */",
    "       short o_delta;",
    "       short o_offset;",
    "       short o_skip;",
    "       short o_delqs;",
    "       char *body;",
    "       struct Stack *nxt;",
    "       struct Stack *lst;",
    "} Stack;\n",
    "typedef struct Svtack { /* for complete state vector */",
    "       short o_delta;   /* current size of frame */",
    "       short m_delta;   /* maximum size of frame */",
    "#if SYNC",
    "       short o_boq;",
    "#endif",
    "       int j1, j2;     /* loop detection */",
    "       char *body;",
    "       struct Svtack *nxt;",
    "       struct Svtack *lst;",
    "} Svtack;\n",
#ifdef VARSTACK
    "typedef struct Varstack {",
    "       int val;",
    "       int cksum;      /* debugging only */",
    "       struct Varstack *nxt;",
    "       struct Varstack *lst;",
    "} Varstack;\n",
#endif
#ifdef GODEF
    "#define UNUSED 0",
    "#define R_LOCK         0",
    "#define W_LOCK         1",
    "#define Snd_LOCK       2",
    "#define Rcv_LOCK       3",
    "#define NLOCKS         4",
    "#define BLOCK  1",
    "#define REL    2",
```

```
 54     "typedef struct CS_stack {",
 55     "       short status;   /* -1,0,1,2 = pending, unused, blocked, released */",
 56     "       short reason;   /*  0..NLOCKS = blocked by R,W,Snd, or Rcv */",
 57     "       short delta;    /* the amount of an increment or decrement */",
 58     "       short pid, stmnt, cs;",
 59     "       int depth;",
 60     "       struct CS_stack *nxt;",
 61     "       struct CS_stack *lst;",
 62     "} CS_stack;\n",
 63 #endif
 64     "typedef struct Trans {",
 65     "       short atom;     /* is this an atomic transition */",
 66     "       short st;       /* the nextstate */",
 67     "       short ist;      /* intermediate state */",
 68 #ifdef GODEF
 69     "       short local;    /* 1 iff this option is local */",
 70     "       short Local;    /* 1 iff all other options are also local */",
 71 #endif
 72     "       char *tp;       /* source text of the forward move */",
 73     "       char ntp;       /* ntyp of the state, e.g. 'r', 'c' etc */",
 74     "       int forw;       /* index for forward transition */",
 75     "       int back;       /* index for return  transition */",
 76     "       struct Trans *nxt;",
 77     "} Trans;\n",
 78     "Trans ***trans;        /* 1 ptr per state per proctype */\n",
 79     "int depthfound = -1;   /* loop detection */",
 80     "short proc_offset[MAXPROC], proc_skip[MAXPROC];",
 81     "short q_offset[MAXQ], q_skip[MAXQ];",
 82     "short vsize;           /* vector size in bytes */",
 83     "short boq = -1;        /* blocked_on_queue status */",
 84 #ifdef GODEF
 85     "short tratable[MAXPROC]; /* no of 1st trans of each proctype */",
 86 #endif
 87     "typedef struct State {",
 88     "       uchar _nr_pr;",
 89     "       uchar _nr_qs;",
 90     "       uchar _p_t; /* loop detection */",
 91     "       uchar _a_t; /* acceptance cycle dectection */",
 92     "    0,
 93 };
 94
 95 char *Addp0[] = {
 96     /* addproc(....parlist... */ ")",
 97     "{",
 98     "       int j, h = now._nr_pr;",
 99     "       if (h >= MAXPROC)",
100     "               Uerror(\"too many processes\");",
101     "       switch (n) {",
102     "       case 0: j = sizeof(P0); break;",
103     "    0,
104 };
105
106 char *Addp1[] = {
107     "       default: Uerror(\"bad proc - addproc\");",
```

```
108     "       }",
109     "       if (vsize%%WS && (j > WS-(vsize%%WS)))",
110     "       {       proc_skip[h] = WS-(vsize%%WS);",
111     "               vsize += proc_skip[h];",
112     "       } else",
113     "               proc_skip[h] = 0;",
114     "       proc_offset[h] = vsize;",
115     "       now._nr_pr += 1;",
116     "       vsize += j;",
117     "       hmax = max(hmax, vsize);",
118     "       if (vsize >= VECTORSZ)",
119     "               Uerror(\"VECTORSZ is too small, edit pan.h\");",
120     "       memset((char *)pptr(h), 0, j);",
121     "       switch (n) {",
122     0,
123 };
124
125 char *Addq0[] = {
126     "addqueue(n)",
127     "{      int j=0, i = now._nr_qs;",
128     "       if (i >= MAXQ)",
129     "               Uerror(\"too many queues\");",
130     "       switch (n) {",
131     0,
132 };
133
134 char *Addq1[] = {
135     "       default: Uerror(\"bad queue - addqueue\");",
136     "       }",
137     "       if (vsize%%WS && (j > WS-(vsize%%WS)))",
138     "       {       q_skip[i] = WS-(vsize%%WS);",
139     "               vsize += q_skip[i];",
140     "       } else",
141     "               q_skip[i] = 0;",
142     "       q_offset[i] = vsize;",
143     "       now._nr_qs += 1;",
144     "       vsize += j;",
145     "       hmax = max(hmax, vsize);",
146     "       if (vsize >= VECTORSZ)",
147     "               Uerror(\"VECTORSZ is too small, edit pan.h\");",
148     "       memset((char *)qptr(i), 0, j);",
149     "       ((Q0 *)qptr(i))->_t = n;",
150     "       return i+1;",
151     "}\n",
152     0,
153 };
154
155 char *Addq11[] = {
156     "{      int j; uchar *z;\n",
157     "       if (!into--)",
158     "       uerror(\"reference to uninitialized chan name (sending)\");",
159     "       if (into >= now._nr_qs || into < 0)",
160     "               Uerror(\"qsend bad queue#\");",
161     "       z = qptr(into);",
```

```
162     "       switch (((Q0 *)qptr(into))->_t) {",
163     0,
164 };
165
166 char *Addq2[] = {
167     "       case 0: printf(\"queue was deleted\\n\");",
168     "       default: Uerror(\"bad queue - qsend\");",
169     "       }",
170     "#endif",
171     "}\n",
172     "#if SYNC==0",
173     "q_zero(from) { /* for picky compilers */ }",
174     "#endif",
175     "#if SYNC",
176     "q_zero(from)",
177     "{",
178     "       if (!from--)",
179     "       uerror(\"reference to uninitialized chan name (receiving)\");",
180     "       switch(((Q0 *)qptr(from))->_t) {",
181     0,
182 };
183
184 char *Addq3[] = {
185     "       case 0: printf(\"queue was deleted\\n\");",
186     "       }",
187     "       Uerror(\"bad queue q-zero\");",
188     "}",
189     "#endif",
190     "q_len(x)",
191     "{      if (!x--) uerror(\"reference to uninitialized chan name\");",
192     "       return ((Q0 *)qptr(x))->Qlen;",
193     "}\n",
194     "q_full(from)",
195     "{      if (!from--)",
196     "       uerror(\"reference to uninitialized chan name (sending)\");",
197     "       switch(((Q0 *)qptr(from))->_t) {",
198     0,
199 };
200
201 char *Addq4[] = {
202     "       case 0: printf(\"queue was deleted\\n\");",
203     "       }",
204     "       Uerror(\"bad queue - q_full\");",
205     "}\n",
206     "qrecv(from, slot, fld, done)",
207     "{      uchar *z;",
208     "       int j, k, r=0;",
209     "       if (!from--)",
210     "       uerror(\"reference to uninitialized chan name (receiving)\");",
211     "       if (from >= now._nr_qs || from < 0)",
212     "               Uerror(\"qrecv bad queue#\");",
213     "       z = qptr(from);",
214     "       switch (((Q0 *)qptr(from))->_t) {",
215     0,
```

```
216 };
217
218 char *Addq5[] = {
219     "       case 0: printf(\"queue was deleted\\n\");",
220     "       default: Uerror(\"bad queue - qrecv\");",
221     "       }",
222     "       return r;",
223     "}\n",
224     0,
225 };
226
227 char *Code0[] = {
228     "run()",
229     "{      memset((char *)&now, 0, sizeof(State));",
230     "       vsize = sizeof(State) - VECTORSZ;",
231     "       settable();",
232     0,
233 };
234 char *Code1[] = {
235     "#define CONNECT        %d /* accept labels */",
236     0,
237 };
238 char *Code2[] = {
239     "       UnBlock;        /* disable rendez-vous */",
240     "#ifdef BITSTATE",
241     "       SS = (uchar *) emalloc(1<<(ssize-3));",
242     "       if (loops)",
243     "       LL = (uchar *) emalloc(1<<(ssize-3));",
244     "#else",
245     "       hinit();",
246     "#endif",
247     "       stack   = ( Stack *) emalloc(sizeof(Stack));",
248     "       svtack  = (Svtack *) emalloc(sizeof(Svtack));",
249 #ifdef VARSTACK
250     "       varstack= (Varstack *) emalloc(sizeof(Varstack));",
251 #endif
252 #ifdef GODEF
253     "       cs_stack= (CS_stack *) emalloc(sizeof(CS_stack));",
254     "       cs_stack->depth = -1;   /* avoid a false match */",
255 #endif
256     "       /* a place to point for Pptr of non-running procs: */",
257     "       noptr   = (uchar *) emalloc(Maxbody * sizeof(char));",
258     "       addproc(0);     /* init */",
259     "       depth=mreached=0;",
260     "       trpt = &trail[depth];",
261     "       new_state();",
262     "}\n",
263
264     "#ifdef JUMBO",
265     "/** EXPERIMENTAL **/",
266     "Trans *",
267     "jumbostep(short II)",
268     "{      register Trans *t, *T = 0; char m, ot; short tt;",
269     "       /* assume this has already been set */",
```

```
270     "#ifdef ALG3",
271     "       printf(\"sorry: cannot combine -DJUMBO with -DALG3\\n\");",
272     "       exit(1);",
273     "#endif",
274
275     "       sv_save();      /* remember where we came from */",
276     "       tt = (short) ((P0 *)this)->_p;",
277     "       ot = (uchar) ((P0 *)this)->_t;",
278     "chain:",
279     "       for (t = trans[ot][tt]; t; t = t->nxt)",
280     "#include \"pan.m\"",
281     "P999:",
282     "       if (m == 0)",
283     "       {       printf(\"cannot happen - jumbostep\\n\");",
284     "               return;",
285     "       }",
286     "       if (!T) T = t;",
287     "       if (t->st)",
288     "       {       tt = ((P0 *)this)->_p = t->st;",
289     "               reached[ot][t->st] = 1;",
290     "               if (trans[ot][tt]->Local > 1)",
291     "                       goto chain;",
292     "       }",
293     "       return T;",
294     "}",
295     "/** END **/",
296     "#endif",
297
298     "new_state()",
299     "{      register Trans *t;",
300     "       char n, m, ot, match_type;",
301     "       short II, tt;\n",
302     "       short From = now._nr_pr-1;",
303     "       short To = 0;",
304 #ifdef GODEF
305     "       char presel;",
306 #endif
307     "Down:",
308 #ifdef GODEF
309     "       presel=0;",
310 #endif
311     "       if (now._p_t && prognow()) /* loop detection */",
312 #ifdef GODEF
313     "       {       trpt->tau |= 16; /* pm for 1 level up */",
314     "               goto Up;",
315     "       }",
316 #else
317     "               goto Up;",
318 #endif
319     "       if (depth >= maxdepth)",
320     "       {       truncs++;",
321     "#if SYNC",
322     "               (trpt+1)->o_n = 1; /* not a deadlock */",
323     "#endif",
```

```
324     "               goto Up;",
325     "       }",
326     "#ifdef VERI",
327     "       if (!(trpt->tau&4))     /* if no claim move */",
328     "#endif",
329     "#if SYNC>0",
330     "       if (boq == -1)          /* if not mid-rv */",
331     "#endif",
332     "       if (!(trpt->tau&8))     /* if no atomic move */",
333     "       {",
334     "#ifdef BITSTATE",
335     "               d_hash((uchar *) &now, vsize);",
336     "               j3 = (1<<(J1&7)); j1 = J1>>3;",
337     "               j4 = (1<<(J2&7)); j2 = J2>>3;",
338     "               if ((SS[j2]&j3) && (SS[j1]&j4))",
339     "#else",
340     
341     "#ifdef CACHE",
342     "               if ((match_type = nh_store((char *)&now, vsize)) != 0)",
343     "#else",
344     "               if ((match_type = hstore((char *)&now, vsize)) != 0)",
345     "#endif",
346     
347     "#endif",
348     "               {       truncs++;",
349     "                       if (match_type == 2)",
350     "                               trpt->tau |= 16; /* pm for 1 level up */",
351     
352     "#if CONNECT>0",
353     "                       if (now._a_t && depth > A_depth)",
354     "                       {       if (memcmp((char *)&A_Root, (char *)&now, vsize) == 0)",
355     "                               {       if (fair_cycle())",
356     "                                       uerror(\"acceptance cycle\");",
357     "                                       if (depth > 0) goto Up; else return;",
358     "                               }",
359     "                       }",
360     "#endif",
361     
362     "#ifdef BITSTATE",
363     "                       if (loops && now._p_t",
364     "                       && LL[j1] && LL[j2] && onstack())",
365     "                       {       if (fair_cycle())",
366     "                               uerror(\"non-progress cycle\");",
367     "                       }",
368     "#endif",
369     "                       if (depth > 0) goto Up; else return;",
370     "               }",
371     "#ifdef BITSTATE",
372     "               SS[j2] |= j3; SS[j1] |= j4;",
373     "               if (loops)",
374     "               {       sv_save();",
375     "                       LL[j1]++; LL[j2]++;",
376     "                       svtack->j1 = J1;",
377     "                       svtack->j2 = J2;",
```

```
378     "               }",
379     "#endif",
380     "               nstates++;",
381     "       }",
382     "       if (depth > mreached)",
383     "               mreached = depth;",
384     "       n = 0;",
385     "#if SYNC",
386     "       (trpt+1)->o_n = 0;",
387     "#endif",
388     "#ifdef VERI",
389     "       if (now._nr_pr < 2",
390     "       || ((P0 *)pptr(1))->_p == endclaim)",
391     "               uerror(\"claim violated!\");",
392     "       if (stopstate[VERI][((P0 *)pptr(1))->_p])",
393     "               uerror(\"endstate in claim reached\");",
394     "Stutter:",
395     "       if (trpt->tau&4)        /* must make a claimmove */",
396     "       {       II = 1;",
397     "               goto Veri0;",
398     "       }",
399     "#endif",
400 #ifdef GODEF
401     "       if (boq != -1) nlinks++;        /* compatibility with patrice */",
402     "#ifndef NOALG2",
403     "       if (boq == -1 && From != To)",
404     "       for (II = From; II >= To; II -= 1)      /* pre-scan */",
405     "       {",
406     "Resume:        /* pick up here when a first pre-selection failed */",
407     "#ifdef VERI",
408     "               if (II == 1) continue;",
409     "#endif",
410     "               this = pptr(II);",
411     "               tt = (short) ((P0 *)this)->_p;",
412     "               ot = (uchar) ((P0 *)this)->_t;",
413     "               for (t = trans[ot][tt]; t; t = t->nxt)",
414     "               {       if (!t->local)",
415     "                               goto Trynext;",
416     "               }",
417     "               From = To = II; /* all moves are local */",
418     "               presel = 1;     /* in case we get stuck */",
419     "               break;",
420     "Trynext:       ;",
421     "       }",
422     "#endif",
423 #endif
424     "\nAgain:",
425     "       for (II = From; II >= To; II -= 1)",
426     "       {",
427     "#ifdef VERI",
428     "               if (II == 1) continue;",
429     "#endif",
430     "Veri0:         this = pptr(II);",
431     "               tt = (short) ((P0 *)this)->_p;",
```

```
432     "               ot = (uchar) ((P0 *)this)->_t;",
433     "#ifdef JUMBO",
434     "/** EXPERIMENTAL **/",
435     "               if(trans[ot][tt]->Local > 1)",
436     "               {       t = jumbostep(II);",
437     "                       m = 3;",
438     "                       depth++; trpt++;",
439     "                       trpt->pr = II;",
440     "                       trpt->st = tt;",
441     "                       goto Q999;",
442     "               }",
443     "/** END **/",
444     "#endif",
445     "               for (t = trans[ot][tt]; t; t = t->nxt)",
446     "               {",
447 #ifdef GODEF
448     "#ifdef ALG3",
449     "                       if (now._p_t == 0)",
450     "                       if (csets[II][t->forw] > 0)",
451     "                       {",
452     "                               continue;",
453     "                       }",
454     "#endif",
455 #endif
456     "#include \"pan.m\"",
457     "P999:                  /* jumps here when move succeeds */",
458     "#ifdef ALG3",
459     "                       if (Nwait > 0)",
460     "                               rel_all_blocks(II);",
461     "#endif",
462     "#ifdef VERBOSE",
463     "       printf(\"%%3d: proc %%d exec %%d, from %%d to %%d %%s\\n\", ",
464     "               depth, II, t->forw, tt, t->st, Moves[t->forw]);",
465     "#ifdef ALG3",
466     "                       dumpsleep(\"new_state\");",
467     "#endif",
468     "#endif",
469     "                       depth++; trpt++;",
470     "                       trpt->pr = II;",
471     "                       trpt->st = tt;",
472     "                       if (t->st)",
473     "                       {       ((P0 *)this)->_p = t->st;",
474 #if 0
475                                     XXXXX WRITING _p XXXXX
476 #endif
477     "                               reached[ot][t->st] = 1;",
478     "                       }",
479     "Q999:                  trpt->o_t  =  t; trpt->o_n  = n;",
480     "                       trpt->o_ot = ot; trpt->o_tt = tt;",
481     "                       trpt->o_To = To; trpt->o_m  = m;",
482     "                       trpt->tau = 0;",
483     "                       if (t->atom&2)",
484     "                       {       trpt->tau |= 8;",
485     "#ifdef VERI",
```

```
486     "                               if((trpt-1)->tau&4)",
487     "                                 trpt->tau |= 4;",
488     "                               else",
489     "                                 trpt->tau &= ~4;",
490     "                       } else",
491     "                       {       if ((trpt-1)->tau&4)",
492     "                                       trpt->tau &= ~4;",
493     "                               else",
494     "                                       trpt->tau |= 4;",
495     "                       }",
496     "#else",
497     "                       } else",
498     "                               trpt->tau &= ~8;",
499     "#endif",
500     "                       if (boq == -1 && t->atom&2)",
501     "                       {       From = To = II; nlinks++;",
502     "                       } else",
503     "                       {       From = now._nr_pr-1; To = 0;",
504     "                       }",
505 #ifdef GODEF
506     "                       if (presel)",
507     "                       {",
508     "                               (trpt-1)->tau |= 32;",
509     "                       } else {",
510     "                               (trpt-1)->tau &= ~32;",
511     "                       }",
512 #endif
513     "                       goto Down;      /* pseudo-recursion */",
514     "Up:",
515 #ifdef GODEF
516     "                       presel=0;",
517 #endif
518     "#if CONNECT>0",
519     "                       if (now._a_t && depth <= A_depth)",
520     "                       {",
521     "                               return; /* we came from checkaccept() */",
522     "                       }",
523     "#endif",
524     "                       t  = trpt->o_t;  n  = trpt->o_n;",
525     "                       ot = trpt->o_ot; II = trpt->pr;",
526     "                       tt = trpt->o_tt; this = pptr(II);",
527     "                       To = trpt->o_To; m  = trpt->o_m;",
528     "#ifdef VERI",
529     "#if SYNC",
530     "/* preserve rendez-vous completion status: */",
531     "/* if the next level was a claim, copy through */",
532     "                       if (trpt->tau&4)",
533     "                               trpt->o_n = (trpt+1)->o_n;",
534     "#endif",
535     "#endif",
536
537     "#ifdef JUMBO",
538     "/** EXPERIMENTAL **/",
539     "                       if (trans[ot][tt]->Local > 1)",
```

```
540     "                       {",
541     "                               sv_restor();",
542     "                               goto R999;",
543     "                       }",
544     "/** END **/",
545     "#endif",
546
547     "#include \"pan.b\"",
548     "R999:                  /* jumps here when done */",
549     "#ifdef VERBOSE",
550     "       printf(\"%%3d: proc %%d reverses %%d, from %%d to %%d\", ",
551     "               depth, II, t->forw, tt, t->st);",
552     "       printf(\" %%s   tau %%d tau-1 %%d\\n\", Moves[t->forw],",
553     "               trpt->tau, (trpt-1)->tau);",
554     "#endif",
555 #ifdef GODEF
556     "#ifdef ALG3",
557     "                       unrelease(); /* undo status 2 forward releases */",
558     "#endif",
559     "                       /* truncated on stack or on a      */",
560     "                       /* progress state with now._p_t==1 */",
561     "                       if (trpt->tau&16)",
562     "                       {       if ((trpt-1)->tau&8)    /* atomic */",
563     "                               {       (trpt-1)->tau |= 16;",
564     "                               }",
565     "                       } else",
566     "                       {       (trpt-1)->tau |= 64;    /* remember it */",
567     "                       }",
568 #endif
569     "                       depth--; trpt--;",
570     "                       if (m > n) n = m;",
571     "                       ((P0 *)this)->_p = tt;",
572     "               } /* all options */",
573 #ifdef GODEF
574     "               push_commit();  /* activate process blocks */",
575 #endif
576     "#ifdef VERI",
577     "               if (II == 1) break;",
578     "#endif",
579     "       } /* all processes */",
580 #ifdef GODEF
581     "#ifdef ALG3",
582     "       unpush();       /* unpush status 1 blocks */",
583     "#endif",
584     "#ifndef NOALG2",
585
586     "       if (!(trpt->tau&64)  /* no nxtstates outside stack  */",
587     "       &&    trpt->tau&32)  /* last moves were preselected */",
588     "       {",
589     "                       presel = 0;",
590     "                       From = now._nr_pr-1; To = 0;",
591     "                       II--;   /* next preselection victim */",
592     "                       if (II >= 0)",
593     "                               goto Resume;",
```

```
594     "                       else",
595     "                               goto Again;",
596     "       }",
597
598     "       if (presel == 1)",
599     "       {       if (n == 0) /* preselected process could not move */",
600     "               {",
601     "                       presel = 0;",
602     "                       From = now._nr_pr-1; To = 0;",
603     "                       II--;   /* next preselection victim */",
604     "                       if (II >= 0)",
605     "                               goto Resume;",
606     "                       else",
607     "                               goto Again;",
608     "               } else if (loops && now._p_t == 0)",
609     "               {       /* must still run progress checker */",
610     "                       From = To = 1;  /* it has pid 1 */",
611     "                       goto Again;",
612     "               }",
613     "       }",
614     "#endif",
615     "#ifdef ALG3",
616     "       if (Nwait == nwait[CS_timeout])",
617     "#endif",
618 #endif
619     "       if (n == 0)",
620     "       {",
621     "#ifdef VERI",
622     "               if (trpt->tau&4) goto Done;     /* ok if a claim blocks */",
623     "#endif",
624     "#if SYNC",
625     "               if (boq == -1)",
626     "#endif",
627     "               if (!endstate() && now._nr_pr ",
628     "               &&  depth < maxdepth-1)",
629     "               {       if (!((trpt->tau)&1))   /* timeout */",
630     "                       {       trpt->tau |= 1;",
631     "                               push_act(0, W_LOCK, REL, 0, CS_timeout);",
632     "       /* if this releases any procs - they are automatically",
633     "          unreleased by the first process returning to this level",
634     "        */",
635     "                               goto Again;",
636     "                       }",
637     "#ifdef VERI",
638     "                       if (n >= 0)     /* Claim Stutter */",
639     "#ifndef NOSTUTTER",
640     "                       {       trpt->tau |= 4;",
641     "                       {       trpt->tau |= 128;",
642     "                               goto Stutter;",
643     "                       }",
644     "#else",
645     "                               goto Done;      /* i.e., always */",
646     "#endif",
647     "#else",
```

```
648     "                       if (loops) goto Done;   /* do loop det. only */",
649     "#endif",
650     "                       if (!(trpt->tau&8))     /* not an atomic move */",
651     "                       {",
652     "#ifdef VERI",
653     "                               printf(\"claim at\");",
654     "                               xrefsrc(claimline,1,((P0 *)pptr(1))->_p);",
655     "#endif",
656     "                               uerror(\"invalid endstate\");",
657     "                       } else",
658     "                               Uerror(\"atomic seq blocks\");",
659     "               }",
660     "#ifdef VERI",
661     "#ifndef NOSTUTTER",
662     "               else",
663     "               {       trpt->tau |= 4;",
664     "                       trpt->tau |= 128;       /* Stutter mark */",
665     "                       goto Stutter;",
666     "               }",
667     "#endif",
668     "#endif",
669     "       }",
670     "Done:",
671 #ifdef GODEF
672     "       if (!(trpt->tau&8))",
673 #else
674     "#ifdef CACHE",
675     "       if (!(trpt->tau&8))",
676     "#else",
677     "       if (loops && !(trpt->tau&8))",
678     "#endif",
679 #endif
680     "#ifdef VERI",
681     "       if (!(trpt->tau&4))",
682     "#endif",
683     "#if SYNC>0",
684     "       if (boq == -1)",
685     "#endif",
686     "       {",
687     "#ifdef BITSTATE",
688     "               LL[(svtack->j1)>>3]--;",
689     "               LL[(svtack->j2)>>3]--;",
690     "               svtack = svtack->lst;",
691     "               if (trpt->tau&2)        /* state marked dirty: remove */",
692     "               {       SS[(svtack->j2)>>3] &= ~(1<<((svtack->j1)&7));",
693     "                       SS[(svtack->j1)>>3] &= ~(1<<((svtack->j2)&7));",
694     "               }",
695     "#else",
696     "               htag((char *)&now, vsize);",
697     "#endif",
698     "       }",
699     "#if CONNECT>0",
700     "#ifdef VERI",
701     "       if ((!(trpt->tau&4))",
```

```
    "       ||  (trpt->tau&128))    /* no claim move, unless Stutter */",
    "#endif",
    "       if (acycles             /* -a option is used */",
    "       &&  !(trpt->tau&8))     /* not an atomic move */",
    "           checkaccept();  /* check for acceptance-cycles */",
    "#endif",
    "       if (depth > 0) goto Up;",
    "}\n",
#ifdef GODEF
    "#ifdef ALG3",
    "rel_all_blocks(pid)    /* not thoroughly tested */",
    "{      int kk, mm, k, s, r, F, T, Cn, effect=0;",
    "       F = tratable[((P0 *)pptr(pid))->_t];",
    "       T = tratable[((P0 *)pptr(pid))->_t+1];",
    "       for (s = F; s < T; s++)",
    "       {       if (csets[pid][s] == 0) continue;",
    "               for (kk = 1; kk < 1+Csels_c[s][0]; kk++)",
    "               {       if (Csels_p[s][kk] != pid) continue;",
    "                       k = Csels_c[s][kk];",
    "                       r = Csels_r[s][kk];",
    "                       Cn = Csels_c[s][0]--;",
    "                       if (Cn < 1) Uerror(\"cannot happen - rel_all\");",
    "                       for (mm = kk; mm < Cn; mm++)",
    "                       {       Csels_c[s][mm] = Csels_c[s][mm+1];",
    "                               Csels_r[s][mm] = Csels_r[s][mm+1];",
    "                               Csels_p[s][mm] = Csels_p[s][mm+1];",
    "                       }",
    "                       csems[pid][k]--;",
    "                       csets[pid][s]--;",
    "                       if (nwait[k] <= 0)",
    "                       {       printf(\"nwait[%%d] = %%d (%%d)\\n\", ",
    "                                       k, nwait[k], Nwait);",
    "                               Uerror(\"nWait\");",
    "                       }",
    "                       nwait[k]--; Nwait--; effect=1;",
    "                       push_cs_el(pid,s,k,depth+1,2,r,1);",
    "                       kk--;",
    "               }",
    "       }",
    "#ifdef VERBOSE",
    "       if (effect) dumpsleep(\"rel_blocks\");",
    "#endif",
    "}",
    "       char *LCK[] = { \"Read\", \"Write\", \"Send\", \"Recv\" };",
    "dumpsleep(str)",
    "       char *str;",
    "{      int pid, xx, yy, zz, kk, F, T;",
    "       for (pid = 0; pid < now._nr_pr; pid++)",
    "       { F = tratable[((P0 *)pptr(pid))->_t];",
    "         T = tratable[((P0 *)pptr(pid))->_t+1];",
    "         for (xx = F; xx < T; xx++)",
    "         {     if (csets[pid][xx] == 0) continue;",
    "               printf(\"sleepset proc %%d: \", pid);",
    "               printf(\" trans %%2d, cs { \", xx);",
```

```
756     "               for (kk = 1; kk < 1+Csels_c[xx][0]; kk++)",
757     "               {       yy = Csels_r[xx][kk];",
758     "                       zz = Csels_c[xx][kk];",
759     "                       if (pid == Csels_p[xx][kk])",
760     "                       {       if (zz < MAXCONFL)",
761     "                                       printf(\"%%s on var %%s, \",",
762     "                                       LCK[yy], CS_names[zz]);",
763     "                               else if (zz == MAXCONFL)",
764     "                                       printf(\"<local>, \");",
765     "                               else",
766     "                                       printf(\"%%s on qid %%d, \",",
767     "                                       LCK[yy], zz);",
768     "                       }",
769     "               }",
770     "       } }",
771     "}",
772     "push_cs_el(pid, stmnt, cs, dp, st, rs, dt)",
773     "{",
774     "       if (stmnt == 0) return; /* timeouts map onto 0 */",
775     "       if (cs_stack->depth > dp)",
776     "       {       push2_cs_el(pid, stmnt, cs, dp, st, rs, dt);",
777     "               return;",
778     "       }",
779     "       if (!cs_stack->nxt)",
780     "       {       cs_stack->nxt = (CS_stack *)",
781     "                       emalloc(sizeof(CS_stack));",
782     "               cs_stack->nxt->lst = cs_stack;",
783     "               cs_max++;",
784     "       }",
785     "       cs_stack = cs_stack->nxt;",
786     "       cs_stack->pid    = pid;",
787     "       cs_stack->stmnt  = stmnt;",
788     "       cs_stack->cs     = cs;",
789     "       cs_stack->delta  = dt;",
790     "       cs_stack->depth  = dp;",
791     "       cs_stack->status = st;",
792     "       cs_stack->reason = rs;",
793     "}\n",
794     /* reach up in cs_stack and insert at correct depth */
795     "push2_cs_el(pid, stmnt, cs, dp, st, rs, dt)",
796     "{      CS_stack *k, *twiddle;",
797     "       cs_max++;",
798     "       twiddle = (CS_stack *) emalloc(sizeof(CS_stack));",
799     "       twiddle->pid    = pid;",
800     "       twiddle->stmnt  = stmnt;",
801     "       twiddle->cs     = cs;",
802     "       twiddle->delta  = dt;",
803     "       twiddle->depth  = dp;",
804     "       twiddle->status = st;",
805     "       twiddle->reason = rs;",
806     "       for (k = cs_stack; k && k->depth > dp; k = k->lst)",
807     "               ;",
808     "       if (k)",
809     "       {       twiddle->nxt = k->nxt;",
```

```
810     "               k->nxt->lst = twiddle;",
811     "               twiddle->lst = k;",
812     "               k->nxt = twiddle;",
813     "       } else",
814     "               cs_stack = twiddle;",
815     "}\n",
816     "push_commit()  /* commit to a pending lock */",
817     "{      CS_stack *k; int mv, Cn, effect=0;",
818     "       for (k = cs_stack; k && k->depth == depth+1; k = k->lst)",
819     "       {       if (k->status != -1) continue;",
820     "               k->status = 1; mv = k->stmnt;",
821     "               Cn = ++Csels_c[mv][0];",
822     "               if (Cn > MULT_MAXCS)",
823     "               {       printf(\"error: recompile with MULT>%%d\\n\",MULT);",
824     "                       exit(1);",
825     "               }",
826     "               Csels_c[mv][Cn] = k->cs;",
827     "               Csels_r[mv][Cn] = k->reason;",
828     "               Csels_p[mv][Cn] = k->pid;",
829     "               csems[k->pid][k->cs]++;",
830     "               csets[k->pid][mv]++;",
831     "               nwait[k->cs]++;",
832     "               Nwait++; effect=1;",
833     "       }",
834     "#ifdef VERBOSE",
835     "       if (effect) dumpsleep(\"push_commit\");",
836     "#endif",
837     "}\n",
838     "char Conflict[NLOCKS][NLOCKS] = { /* 1 == DEP, 0 == IND */",
839     " /*                       R_LOCK, W_LOCK, Snd_LOCK, Rcv_LOCK */",
840     " /* R_LOCK */          {   0,      1,          1,          1 },",
841     " /* W_LOCK */          {   1,      1,          1,          1 },",
842     " /* Snd_LOCK */        {   1,      1,          1,          M_LOSS },",
843     " /* Rcv_LOCK */        {   1,      1,          M_LOSS,     1 },",
844     "};",
845     "/* when m_loss is set (on SPIN's -m flag) sends and receives",
846     " * on the same queue are really only dependent when the queue",
847     " * is full - the above version is therefore a little conservative",
848     " */",
849     "push_act(pid, what, when, stmnt, cs)   /* log a global action */",
850     "{      int i, j, k, r, F, R, T, maxk, delta, kk, own; int effect=0;",
851     "",
852     "       if (when == BLOCK)      /* set a pending lock */",
853     "#ifndef NOPELED",
854     "       {       if (!(trpt->tau&16))    /* PELED's PROVISO */",
855     "#else",
856     "       {",
857     "#endif",
858     "               push_cs_el(pid, stmnt, cs, depth, -1, what, 1);",
859     "               return;",
860     "       } /* else release */",
861     "       maxk = 1+MAXCONFL+now._nr_qs;",
862     "       if (nwait[cs] > 0)",
863     "       for (i = 0; i < now._nr_pr; i++)",
```

```
864     "       if (csems[i][cs] > 0)",
865     "       { F = tratable[((P0 *)pptr(i))->_t];",
866     "         T = tratable[((P0 *)pptr(i))->_t+1];",
867     "         for (j = F; j < T; j++)",
868     "         {     for (kk = 1; kk < 1+Csels_c[j][0]; kk++)",
869     "               {       if (Csels_c[j][kk] == cs",
870     "                       &&  Conflict[what][Csels_r[j][kk]])",
871     "                       {       /* clear all blocks on j */",
872     "                               for (kk = 1; kk < 1+Csels_c[j][0]; kk++)",
873     "                               {       r   = Csels_r[j][kk];",
874     "                                       k   = Csels_c[j][kk];",
875     "                                       own = Csels_p[j][kk];",
876     "                                       csems[own][k]--;",
877     "                                       csets[own][j]--;",
878     "                                       nwait[k]--;",
879     "                                       Nwait--;",
880     "                                       push_cs_el(own,j,k,depth+1,2,r,1);",
881     "                                       effect = 1;",
882     "                               }",
883     "                               Csels_c[j][0] = 0;",
884     "                               break;",
885     "               }       }",
886     "         }",
887     "       }",
888     "out:   if (nwait[cs] < 0)",
889     "               Uerror(\"nwait negative\");",
890     "#ifdef VERBOSE",
891     "       if (effect) dumpsleep(\"act\");",
892     "#endif",
893     "}\n",
894     "unrelease()",
895     "{      int k, p, s, dt, Cn, effect=0;",
896     "       CS_stack *K;",
897     "       for (K = cs_stack; K && K->depth == depth; K = K->lst)",
898     "       {       k = K->cs;",
899     "               p = K->pid;",
900     "               s = K->stmnt;",
901     "               if (K->status == 2)",
902     "               {",
903     "                       for (dt = 0; dt < K->delta; dt++)",
904     "                       {       Cn = ++Csels_c[s][0];",
905     "                               if (Cn > MULT_MAXCS)",
906     "                               Uerror(\"cannot happen - Csels1\");",
907     "                               Csels_c[s][Cn] = k;",
908     "                               Csels_r[s][Cn] = K->reason;",
909     "                               Csels_p[s][Cn] = p;",
910     "                               csems[p][k]++;",
911     "                               csets[p][s]++;",
912     "                               nwait[k]++; Nwait++;",
913     "                       }",
914     "                       K->status = 3;",
915     "                       effect=1;",
916     "               }",
917     "       }",
```

```
    "#ifdef VERBOSE",
    "       if (effect) dumpsleep(\"unrelease\");",
    "#endif",
    "}\n",
    "unpush()",
    "{      int k, p, r, s, kk, mm, Cn, oCn, effect=0;",
    "       while (cs_stack && cs_stack->depth == depth+1)",
    "       {       k = cs_stack->cs;",
    "               p = cs_stack->pid;",
    "               s = cs_stack->stmnt;",
    "               r = cs_stack->reason;",
    "               if (cs_stack->status == 1)",
    "               {",
    "                       oCn = Csels_c[s][0];",
    "                       for (kk = 1; kk < 1+Csels_c[s][0]; kk++)",
    "                       if (Csels_r[s][kk] == r",
    "                       &&  Csels_c[s][kk] == k",
    "                       &&  Csels_p[s][kk] == p)",
    "                       {       Cn = Csels_c[s][0]--;",
    "                               if (Cn < 1)",
    "                               Uerror(\"cannot happen - Csels2\");",
    "                               for (mm = kk; mm < Cn; mm++)",
    "                               { Csels_c[s][mm] = Csels_c[s][mm+1];",
    "                                 Csels_r[s][mm] = Csels_r[s][mm+1];",
    "                                 Csels_p[s][mm] = Csels_p[s][mm+1];",
    "                               }",
    "                               break;",
    "                       }",
    "                       if (oCn == Csels_c[s][0])",
    "                       {",
    "       printf(\"cannot find %%d,%%d in\\n\", r, k);",
    "       for (kk = 1; kk < 1+Csels_c[s][0]; kk++)",
    "       printf(\"\t%%d,%%d\\n\", Csels_r[s][kk], Csels_c[s][kk]);",
    "                               Uerror(\"cannot happen Cs unpush\");",
    "                       }",
    "                       csems[p][k]--;",
    "                       csets[p][s]--;",
    "                       if (nwait[k] <= 0)",
    "                       {       printf(\"nwait[%%d] = %%d (%%d)\\n\", ",
    "                                       k, nwait[k], Nwait);",
    "                               Uerror(\"nwait\");",
    "                       }",
    "                       nwait[k]--; Nwait--; effect=1;",
    "               } else if (cs_stack->status != 3)",
    "               {       printf(\"cs = %%d, mv = %%d\\n\", ",
    "                               cs_stack->cs, cs_stack->stmnt);",
    "                       printf(\"Bad status: %%d\\n\", cs_stack->status);",
    "                       Uerror(\"unpush\");",
    "               }",
    "               cs_stack->status = cs_stack->reason = 0;",
    "               cs_stack = cs_stack->lst;",
    "       }",
    "#ifdef VERBOSE",
    "       if (effect) dumpsleep(\"unpush\");",
```

```
    "#endif",
    "}\n",
    "#endif",
#endif
    "assert(a, s, ii, tt, t)",
    "       char *s;",
    "       Trans *t;",
    "{      if (!a)",
    "       {       printf(\"assertion violated %%s\", s);",
    "               depth++; trpt++;",
    "               trpt->pr = ii;",
    "               trpt->st = tt;",
    "               trpt->o_t = t;",
    "               uerror(\"aborted\");",
    "               depth--; trpt--;",
    "       }",
    "}",
    "#ifndef NOBOUNDCHECK",
    "Boundcheck(x, y, a1, a2, a3)",
    "       Trans *a3;",
    "{      assert((x >= 0 && x < y), \"- invalid array index\\n\", a1, a2, a3);",
    "       return x;",
    "}",
    "#endif",
    "#ifdef MEMCNT",
    "int memcnt=0;",
    "#endif",
    "void",
    "wrapup()",
    "{",
    "#ifdef BITSTATE",
    "       double a, b;\n",
    "       printf(\"bit statespace search \");",
    "#else",
    "       printf(\"full statespace search \");",
    "#endif",
    "#ifdef VERI",
    "       printf(\"on behavior restricted to claim \");",
    "#endif",
    "       printf(\"for:\\n\tassertion violations\");",
    "#ifndef VERI",
    "       if (loops)",
    "               printf(\" and %%s non-progress loops\",",
    "                       fairness?\"FAIR\":\"\");",
    "       else",
    "               printf(\" and invalid endstates\");",
    "#endif",
    "#if CONNECT>0",
    "       if (acycles && !loops)",
    "               printf(\"\\n\\tand %%s acceptance cycles\",",
    "                       fairness?\"FAIR\":\"\");",
    "#endif",
    "       if (!done) printf(\"\\nsearch was not completed\");",
    "       printf(\"\\nvector %%d byte, depth reached %%d\", ",
```

```
1026    "                                       hmax, mreached);",
1027    "       if (loops)",
1028    "       {  printf(\", non-progress loops: %%d\\n\", errors);",
1029    "       } else",
1030    "         printf(\", errors: %%d\\n\", errors);",
1031    "       printf(\"%%8d states, stored\", nstates - recycled);",
1032    "       if (recycled) printf(\" (%%d recycled)\", recycled);",
1033    "       printf(\"\\n%%8d states, linked\\n\", nlinks);",
1034    "       printf(\"%%8d states, matched\t   total: %%8d\\n\",",
1035    "       truncs, nstates+nlinks+truncs);",
1036    "#ifdef BITSTATE",
1037    "       a = (double) (1<<ssize);",
1038    "       b = (double) nstates+1.;",
1039    "       printf(\"hash factor: %%f \", a/b);",
1040    "       printf(\"(best coverage if >100)\\n\");",
1041    "#else",
1042    "       printf(\"hash conflicts: %%d (resolved)\\n\", hcmp);",
1043    "#endif",
1044    "       printf(\"(max size 2^%%d states, \", ssize);",
1045 #ifdef VARSTACK
1046    "       printf(\"varstack: %%d, \", vmax);",
1047 #endif
1048 #ifdef GODEF
1049    "       printf(\"cs_stack: %%d, \", cs_max);",
1050 #endif
1051    "       printf(\"stackframes: %%d/%%d)\\n\\n\", smax, svmax);",
1052    "       if (M_LOSS) printf(\"total messages lost: %%d\\n\\n\", loss);",
1053    "#ifdef MEMCNT",
1054    "       printf(\"memory used: %%d\\n\", memcnt);",
1055    "#endif",
1056    "       if (done && !loops) do_reach();",
1057 #ifdef GODEF
1058    "#ifdef ALG3",
1059    "#ifdef VERBOSE",
1060    "       if (done)",
1061    "       {       int i,j,k,r;",
1062    "               for (j = 0; j < MAXSTATE; j++)",
1063    "               {       if (Csels_c[j][0] != 0)",
1064    "                       printf(\"Csels_c[%%d][0] = %%d\\n\",",
1065    "                               j, Csels_c[j][0]);",
1066    "               }",
1067    "               for (i = 0; i < MAXPROC; i++)",
1068    "                       for (k = 0; k < TOPQ; k++)",
1069    "                               if (csems[i][k] != 0)",
1070    "                               printf(\"\tcsem %%d,%%d = %%d\\n\", ",
1071    "                               i,k, csems[i][k]);",
1072
1073    "               for (j = 0; j < TOPQ; j++)",
1074    "                       if (nwait[j] != 0)",
1075    "                               printf(\"\tnwait %%d = %%d\\n\", ",
1076    "                               j, nwait[j]);",
1077    "               if (Nwait != 0)",
1078    "                       printf(\"Nwait = %%d\\n\", Nwait);",
1079    "       }",
```

```
1080    "#endif",
1081    "#endif",
1082 #endif
1083    "       exit(0);",
1084    "}\n",
1085    "d_hash(cp, om)",
1086    "       uchar *cp;",
1087    "{",
1088    "       register long z = 0x88888EEFL;",
1089    "       register long *q, *r;",
1090    "       register int h;",
1091    "       register m, n;\n",
1092    "       h = (om+3)/4;",
1093    "       m = n = -1;",
1094    "       q = r = (long *) cp;",
1095    "       r += (long) h;",
1096    "       do {",
1097    "               m += m;",
1098    "               if (m < 0)",
1099    "                       m ^= z;",
1100    "               m ^= *q++;",
1101    "               n += n;",
1102    "               if (n < 0)",
1103    "                       n ^= z;",
1104    "               n ^= *--r;",
1105    "       } while (--h > 0);",
1106    "       J1 = (m ^ (m>>(8*sizeof(unsigned)-ssize)))&mask;",
1107    "       J2 = (n ^ (n>>(8*sizeof(unsigned)-ssize)))&mask;",
1108    "}\n",
1109    "s_hash(cp, om)",
1110    "       uchar *cp;",
1111    "{",
1112    "#ifdef ALTHASH",
1113    "       d_hash(cp,om);",
1114    "       j1 = (J1^J2)&mask;",
1115    "#else",
1116    "       register long z = 0x88888EEFL;",
1117    "       register long *q;",
1118    "       register int h;\n",
1119    "       register m = -1;",
1120    "       h = (om+3)/4;",
1121    "       q = (long *) cp;",
1122    "       do {",
1123    "               m += m;",
1124    "               if (m < 0)",
1125    "                       m ^= z;",
1126    "               m ^= *q++;",
1127    "       } while (--h > 0);",
1128    "       j1 = (m ^ (m>>(8*sizeof(unsigned)-ssize)))&mask;",
1129    "#endif",
1130    "}\n",
1131    "main(argc, argv)",
1132    "       char *argv[];",
1133    "{",
```

```
1134    "       while (argc > 1 && argv[1][0] == '-')",
1135    "       {       switch (argv[1][1]) {",
1136    "#if CONNECT>0",
1137    "               case 'a': acycles = 1; break;",
1138    "#endif",
1139    "               case 'c': upto  = atoi(&argv[1][2]); break;",
1140    "               case 'd': state_tables++; break;",
1141    "               case 'f': fairness = 1; break;",
1142    "               case 'H': homomorphism = 1;",
1143    "                         if (argc < 4) { usage(); exit(); }",
1144    "                         hom_target = argv[2]; hom_source = argv[3];",
1145    "                         printf(\"short trans;\\n\");",
1146    "                         break;",
1147    "#ifndef VERI",
1148    "               case 'l': loops = 1; break;",
1149    "#endif",
1150    "               case 'm': maxdepth = atoi(&argv[1][2]); break;",
1151    "               case 'w': ssize = atoi(&argv[1][2]); break;",
1152 #ifdef PAIRS
1153    "               case 't': tree_before=1; break;",
1154 #endif
1155    "               default : usage(); exit(1);",
1156    "               }",
1157    "               argc--; argv++;",
1158    "       }",
1159    "       if (acycles && loops)",
1160    "       {       fprintf(stderr, \"sorry: cannot combine -a and -l\\n\");",
1161    "               usage(); exit(1);",
1162    "       }",
1163    "       if (fairness && !acycles && !loops)",
1164    "       {       fprintf(stderr, \"sorry: option -f only has effect when\");",
1165    "               fprintf(stderr, \" combined with -a or -l\\n\");",
1166    "               usage(); exit(1);",
1167    "       }",
1168    "       signal(SIGINT, wrapup);",
1169    "       mask = ((1<<ssize)-1);  /* hash init */",
1170    "       trail = (Trail *) emalloc((maxdepth+2)*sizeof(Trail));",
1171    "       run();",
1172    "       done = 1;",
1173    "       wrapup();",
1174    "}\n",
1175    "usage()",
1176    "{      fprintf(stderr, \"unknown option\\n\");",
1177    "#if CONNECT>0",
1178    "       fprintf(stderr, \"-a  find acceptance cycles\\n\");",
1179    "#else",
1180    "       fprintf(stderr, \"-a  disabled (no accept labels are defined)\\n\");",
1181    "#endif",
1182    "       fprintf(stderr, \"-cN stop at Nth error \");",
1183    "       fprintf(stderr, \"(default=1)\\n\");",
1184    "       fprintf(stderr, \"-d  print state tables and stop\\n\");",
1185    "       fprintf(stderr, \"-d -d  print un-optimized state tables\\n\");",
1186    "       fprintf(stderr, \"-f  enforce weak fairness in cycles\\n\");",
1187    "       fprintf(stderr, \"-H  target_proctype source_proctype\\n\");",
```

```
1188    "       fprintf(stderr, \"    produce a model for proving homomorphism\\n\");",
1189    "#ifndef VERI",
1190    "       fprintf(stderr, \"-l  find non-progress cycles\\n\");",
1191    "#else",
1192    "       fprintf(stderr, \"-l  disabled (by presence of never claim)\\n\");",
1193    "#endif",
1194    "       fprintf(stderr, \"-mN max depth N (default=10k)\\n\");",
1195    "       fprintf(stderr, \"-wN hashtable of 2^N entries \");",
1196    "       fprintf(stderr, \"(default=%%d)\\n\", ssize);",
1197    "}\n",
1198 #if 0
1199    "char *",
1200    "emalloc(n)",
1201    "{      char *tmp = malloc(n);",
1202    "#ifdef MEMCNT",
1203    "       if (!tmp || memcnt > 1<<MEMCNT)",
1204    "#else",
1205    "       if (!tmp)",
1206    "#endif",
1207    "       {       printf(\"pan: out of memory\\n\");",
1208    "               wrapup();",
1209    "       }",
1210    "#ifdef MEMCNT",
1211    "       memcnt += n;",
1212    "#endif",
1213    "       memset(tmp, 0, n);",
1214    "       return tmp;",
1215    "}\n",
1216 #else
1217    "/* include realloc and free to keep sysV libc",
1218    " * from including them and",
1219    " * finding multiple references",
1220    " */",
1221    "char *",
1222    "realloc(s)",
1223    "       char *s;",
1224    "{      printf(\"aborting: cannot happen - call on realloc()\\n\");",
1225    "       wrapup();",
1226    "}",
1227    "",
1228    "free(s)",
1229    "       char *s;",
1230    "{      /* never called - simply ignore it */",
1231    "}",
1232    "",
1233    "char *",
1234    "malloc(n)",
1235    "       unsigned n;",
1236    "{",
1237    "       char *tmp;",
1238    "       extern char *sbrk();",
1239    "       tmp = sbrk(n);",
1240    "#ifdef MEMCNT",
1241    "       if ((int) tmp == -1 || memcnt > 1<<MEMCNT)",
```

```
1242    "#else",
1243    "       if ((int) tmp == -1)",
1244    "#endif",
1245    "       {       printf(\"aborting: out of memory\\n\");",
1246    "               wrapup();",
1247    "       }",
1248    "#ifdef MEMCNT",
1249    "       memcnt += n;",
1250    "#endif",
1251    "       return tmp;",
1252    "}",
1253    "",
1254    "#define CHUNK  4096",
1255    "",
1256    "char *",
1257    "emalloc(n)      /* memory is never released or reallocated */",
1258    "       unsigned n;",
1259    "{",
1260    "       char *tmp;",
1261    "       static char *have;",
1262    "       static long left = 0L;",
1263    "       static long fragment = 0L;",
1264    "",
1265    "       if (n == 0)",
1266    "               return (char *) NULL;",
1267    "       if (n&3)",
1268    "               n += 4-(n&3);   /* for proper alignment */",
1269    "       if (left < n)",
1270    "       {       unsigned grow = (n < CHUNK) ? CHUNK : n;",
1271    "               have = malloc(grow);",
1272    "               fragment += left;",
1273    "               left = grow;",
1274    "       }",
1275    "       tmp = have;",
1276    "       have += (long) n;",
1277    "       left -= (long) n;",
1278    "       memset(tmp, 0, n);",
1279    "       return tmp;",
1280    "}",
1281 #endif
1282    "Uerror(str)",
1283    "       char *str;",
1284    "{      /* always fatal */",
1285    "       errors = upto-1;",
1286    "       uerror(str);",
1287    "       wrapup();",
1288    "}\n",
1289    "uerror(str)",
1290    "       char *str;",
1291    "{",
1292    "       if (++errors == upto)",
1293    "       {       printf(\"pan: %%s (at depth %%d)\\n\", str,",
1294    "               (depthfound==-1)?depth:depthfound);",
1295    "               putrail();",
```

```
1296    "               wrapup();",
1297    "       }",
1298    "       return 1;",
1299    "}\n",
1300    "r_ck(which, N, M, src)",
1301    "       uchar *which;",
1302    "       short *src;",
1303    "{      int i, m=0;\n",
1304    "#ifdef VERI",
1305    "       if (M == VERI) return;  /* no useful info there */",
1306    "#endif",
1307    "       printf(\"unreached in proctype %%s:\\n\", procname[M]);",
1308    "       for (i = 1; i < N; i++)",
1309    "         if (which[i] == 0)",
1310    "               xrefsrc(src[i], M, i);",
1311    "         else",
1312    "               m++;",
1313    "       printf(\"\t(%%d of %%d states)\\n\", N-1-m, N-1);",
1314    "}\n",
1315    "xrefsrc(lno, M, i)",
1316    "{",
1317    "       printf(\"\\tline %%d (state %%d)\", lno, i);",
1318    "       xrefstmnt(M, i);",
1319    "}",
1320    "xrefstmnt(M, i)",
1321    "{",
1322    "       if (trans[M][i] && trans[M][i]->tp)",
1323    "       {       if (strcmp(trans[M][i]->tp, \"\") != 0)",
1324    "                       printf(\", \\\"%%s\\\"\", trans[M][i]->tp);",
1325    "               else if (stopstate[M][i])",
1326    "                       printf(\", -endstate-\");",
1327    "       } else",
1328    "               printf(\", ?\");",
1329    "       printf(\"\\n\");",
1330    "}\n",
1331    "putrail()",
1332    "{      int fd, i, j, q;",
1333    "       char snap[64];\n",
1334    "       if ((fd = creat(\"pan.trail\", 0666)) <= 0)",
1335    "       {       printf(\"cannot create pan.trail\\n\");",
1336    "               return;",
1337    "       }",
1338    "#ifdef VERI",
1339    "       sprintf(snap, \"-2:%%d:-2:-2\\n\", VERI);",
1340    "       write(fd, snap, strlen(snap));",
1341    "#endif",
1342    "       for (i = 1, j = 0; i <= depth; i++)",
1343    "       {       q = trail[i].pr;",
1344    "               if (i == depthfound)",
1345    "                       write(fd, \"-1:-1:-1:-1\\n\", 12);",
1346    "               if (loops)",
1347    "#ifdef VERI",
1348    "               {       if (q == 2) continue;",
1349    "                       if (q  > 2) q -= 2;",
```

```
1350    "               }",
1351    "#else",
1352    "               {       if (q == 1) continue;",
1353    "                       if (q  > 1) q--;",
1354    "               }",
1355    "#endif",
1356    "               if (trail[i].o_t->ist)",
1357    "               {  sprintf(snap, \"%%d:%%d:%%d:%%d\\n\", j++,",
1358    "                                  q, trail[i].o_t->ist, i);",
1359    "                  write(fd, snap, strlen(snap));",
1360    "               }",
1361    "               sprintf(snap, \"%%d:%%d:%%d:%%d\\n\", j++, ",
1362    "                       q, trail[i].o_t->st, i);",
1363    "               write(fd, snap, strlen(snap));",
1364    "       }",
1365    "       printf(\"pan: wrote pan.trail\\n\");",
1366    "       close(fd);",
1367    "}\n",
1368    "sv_save()      /* push state vector onto save stack */",
1369    "{      if (!svtack->nxt)",
1370    "       {  svtack->nxt = (Svtack *) emalloc(sizeof(Svtack));",
1371    "          svtack->nxt->body = emalloc(vsize*sizeof(char));",
1372    "          svtack->nxt->lst = svtack;",
1373    "          svtack->nxt->m_delta = vsize;",
1374    "          svmax++;",
1375    "       } else if (vsize > svtack->nxt->m_delta)",
1376    "       {  svtack->nxt->body = emalloc(vsize*sizeof(char));",
1377    "          svtack->nxt->lst = svtack;",
1378    "          svtack->nxt->m_delta = vsize;",
1379    "          svmax++;",
1380    "       }",
1381    "       svtack = svtack->nxt;",
1382    "#if SYNC",
1383    "       svtack->o_boq = boq;",
1384    "#endif",
1385    "       svtack->o_delta = vsize;",
1386    "       memcpy((char *)(svtack->body), (char *)&now, vsize);",
1387    "}\n",
1388    "sv_restor()    /* pop state vector from save stack */",
1389    "{      memcpy((char *)&now, svtack->body, svtack->o_delta);",
1390    "#if SYNC",
1391    "       boq = svtack->o_boq;",
1392    "#endif",
1393    "       if (vsize != svtack->o_delta)",
1394    "               Uerror(\"sv_restor\");",
1395    "       if (!svtack->lst)",
1396    "               Uerror(\"error: v_restor\");",
1397    "       svtack  = svtack->lst;",
1398    "}\n",
1399    "p_restor(h)",
1400    "{      int i; char *z = (char *) &now;",
1401    "       proc_offset[h] = stack->o_offset;",
1402    "       proc_skip[h]   = stack->o_skip;",
1403    "       vsize += stack->o_skip;",
```

```
1404    "       memcpy(z+vsize, stack->body, stack->o_delta);",
1405    "       vsize += stack->o_delta;",
1406    "       i = stack->o_delqs;",
1407    "       now._nr_pr += 1;",
1408    "       if (!stack->lst)        /* debugging */",
1409    "               Uerror(\"error: p_restor\");",
1410    "       stack = stack->lst;",
1411    "       this = pptr(h);",
1412    "       while (i-- > 0)",
1413    "               q_restor();",
1414    "}\n",
1415    "q_restor()",
1416    "{      char *z = (char *) &now;",
1417    "       q_offset[now._nr_qs] = stack->o_offset;",
1418    "       q_skip[now._nr_qs]   = stack->o_skip;",
1419    "       vsize += stack->o_skip;",
1420    "       memcpy(z+vsize, stack->body, stack->o_delta);",
1421    "       vsize += stack->o_delta;",
1422    "       now._nr_qs += 1;",
1423    "       if (!stack->lst)        /* debugging */",
1424    "               Uerror(\"error: q_restor\");",
1425    "       stack = stack->lst;",
1426    "}\n",
1427    "delproc(sav, h)",
1428    "{      int d, i=0;",
1429    "",
1430    "       if (h+1 != now._nr_pr) return 0;",
1431    "",
1432    "       while (now._nr_qs",
1433    "       &&     q_offset[now._nr_qs-1] > proc_offset[h])",
1434    "       {       delq(sav);",
1435    "               i++;",
1436    "       }",
1437    "       d = vsize - proc_offset[h];",
1438    "       if (sav)",
1439    "       {       if (!stack->nxt)",
1440    "               {       stack->nxt = (Stack *)",
1441    "                               emalloc(sizeof(Stack));",
1442    "                       stack->nxt->body = ",
1443    "                               emalloc(Maxbody*sizeof(char));",
1444    "                       stack->nxt->lst = stack;",
1445    "                       smax++;",
1446    "               }",
1447    "               stack = stack->nxt;",
1448    "               stack->o_offset = proc_offset[h];",
1449    "               stack->o_skip   = proc_skip[h];",
1450    "               stack->o_delta  = d;",
1451    "               stack->o_delqs  = i;",
1452    "               memcpy(stack->body, (char *)pptr(h), d);",
1453    "       }",
1454    "       vsize = proc_offset[h];",
1455    "       now._nr_pr = now._nr_pr - 1;",
1456    "       memset((char *)pptr(h), 0, d);",
1457    "       vsize -= proc_skip[h];",
```

```
1458    "       return 1;",
1459    "}\n",
1460 #ifdef VARSTACK
1461    "pushvarval(v, ck)",
1462    "{      if (!varstack->nxt)",
1463    "       {       varstack->nxt = (Varstack *)",
1464    "                       emalloc(sizeof(Varstack));",
1465    "               varstack->nxt->lst = varstack;",
1466    "               vmax++;",
1467    "       }",
1468    "       varstack = varstack->nxt;",
1469    "       varstack->val = v;",
1470    "       varstack->cksum = ck;",
1471    "}\n",
1472    "popvarval(ck)",
1473    "{      if (!varstack->lst)",
1474    "               Uerror(\"error: popvar\");",
1475    "       if (varstack->cksum != ck)",
1476    "       {       printf(\"%%d <-> %%d\\n\", varstack->cksum, ck);",
1477    "               Uerror(\"mismatch varstack\");",
1478    "       }",
1479    "       varstack = varstack->lst;",
1480    "       return varstack->nxt->val;",
1481    "}\n",
1482 #endif
1483    "delq(sav)",
1484    "{      int h = now._nr_qs - 1;",
1485    "       int d = vsize - q_offset[now._nr_qs - 1];",
1486    "       if (sav)",
1487    "       {       if (!stack->nxt)",
1488    "               {       stack->nxt = (Stack *)",
1489    "                               emalloc(sizeof(Stack));",
1490    "                       stack->nxt->body = ",
1491    "                               emalloc(Maxbody*sizeof(char));",
1492    "                       stack->nxt->lst = stack;",
1493    "                       smax++;",
1494    "               }",
1495    "               stack = stack->nxt;",
1496    "               stack->o_offset = q_offset[h];",
1497    "               stack->o_skip   = q_skip[h];",
1498    "               stack->o_delta  = d;",
1499    "               memcpy(stack->body, (char *)qptr(h), d);",
1500    "       }",
1501    "       vsize = q_offset[h];",
1502    "       now._nr_qs = now._nr_qs - 1;",
1503    "       memset((char *)qptr(h), 0, d);",
1504    "       vsize -= q_skip[h];",
1505    "}\n",
1506    "prognow()",
1507    "{",
1508    "       int i; P0 *ptr;",
1509    "       for (i = 0; i < now._nr_pr; i++)",
1510    "       {       ptr = (P0 *) pptr(i);",
1511    "               if (progstate[ptr->_t][ptr->_p])",
```

```
1512    "                       return 1;",
1513    "       }",
1514    "       return 0;",
1515    "}\n",
1516    "endstate()",
1517    "{      int i; P0 *ptr;",
1518    "       for (i = 0; i < now._nr_pr; i++)",
1519    "       {",
1520    "#ifdef VERI",
1521    "               if (i == 1) continue;",
1522    "#endif",
1523    "               ptr = (P0 *) pptr(i);",
1524    "               if (!stopstate[ptr->_t][ptr->_p])",
1525    "                       return 0;",
1526    "       }",
1527    "       if (loops)",
1528    "               uerror(\"non progress sequence\");",
1529    "       return 1;",
1530    "}\n",
1531    "onstack()",
1532    "{      register Svtack *ptr;",
1533    "       register char *won = (char *)&now;",
1534    "       register int j=depth;",
1535    "       for (ptr = svtack; ptr; ptr = ptr->lst, j--)",
1536    "       if (ptr->o_delta == vsize",
1537    "       &&  ptr->j1 == J1 && ptr->j2 == J2",
1538    "       &&  memcmp(ptr->body, won, vsize) == 0)",
1539    "       {       depthfound = j;",
1540    "               return 1;",
1541    "       }",
1542    "       return 0;",
1543    "}\n",
1544    "fair_cycle()",
1545    "{      int i, j, q, II;",
1546    "       Trans *t;",
1547    "       short tt;",
1548    "       char ot;",
1549    "       uchar moved[MAXPROC];",
1550    "",
1551    "       if (!fairness) return 1;",
1552    "       memset(moved, 0, MAXPROC);",
1553    "#ifdef VERI",
1554    "       moved[1] = 1;",
1555    "       if (loops) moved[2] = 1;",
1556    "#else",
1557    "       if (loops) moved[1] = 1;",
1558    "#endif",
1559    "       for (i = depthfound; i <= depth; i++)",
1560    "       {       q = trail[i].pr;",
1561    "#ifdef VERI",
1562    "               if (q == 1 || (loops && q == 2)) continue;",
1563    "#else",
1564    "               if (loops && q == 1) continue;",
1565    "#endif",
```

```
1566    "               moved[q] = 1;",
1567    "       }",
1568    "       for (II = 0; II < now._nr_pr; II++)",
1569    "       {       if (!moved[II])",
1570    "               {       this = pptr(II);",
1571    "                       tt = (short) ((P0 *)this)->_p;",
1572    "                       ot = (uchar) ((P0 *)this)->_t;",
1573    "                       for (t = trans[ot][tt]; t; t = t->nxt)",
1574    "                       {",
1575    "#include \"pan.f\"",
1576    "                               goto not_fair;",
1577    "                       }",
1578    "               }",
1579    "       }",
1580    "       /* a fair cycle was detected */",
1581    "       for (i = depthfound-1; i <= depth; i++)",
1582    "               trail[i].tau &= ~2;     /* unmark states in SCC */",
1583    "       return 1;",
1584    "not_fair:",
1585    "       /* mark all states in the SCC dirty - to avoid missing fair */",
1586    "       /* traversals of the same SCC that could be generated later */",
1587    "       for (i = depthfound-1; i <= depth; i++)",
1588    "               trail[i].tau |= 2;",
1589    "       return 0;",
1590    "}\n",
1591
1592    "#if CONNECT>0",
1593    "checkaccept()",
1594    "{      int i;",
1595    "       for (i = 0; i < now._nr_pr; i++)",
1596    "       {       P0 *ptr = (P0 *) pptr(i);",
1597    "               if (accpstate[ptr->_t][ptr->_p])",
1598    "                       break;",
1599    "       }",
1600    "       if (i == now._nr_pr)",
1601    "               return;",
1602    "       if (now._a_t)",
1603    "       {",
1604    "               return;",
1605    "       }",
1606    "       now._a_t = 13;  /* 13 to help the hasher */",
1607    "       A_depth = depth;",
1608    "       memcpy((char *)&A_Root, (char *)&now, vsize);",
1609    "       depthfound = depth;",
1610    "       new_state();    /* the 2nd search */",
1611    "       depthfound = -1;",
1612    "       now._a_t = 0;",
1613    "}",
1614    "#endif\n",
1615
1616    "#ifndef BITSTATE",
1617    "struct H_el {",
1618    "       struct H_el *nxt;",
1619 #if 0
```

```
1620    "#ifdef CACHE",
1621    "       struct H_el *lst, *unxt;",
1622    "       unsigned hslot;",       /* hash table slot   */
1623    "#endif",
1624 #endif
1625    "       unsigned sz;",          /* state vector size */
1626    "       unsigned tagged;",      /* bits 30 and 31 are special */
1627    "       unsigned state;",
1628    "} **H_tab;\n",
1629
1630    "hinit()",
1631    "{      H_tab = (struct H_el **)",
1632    "               emalloc((1<<ssize)*sizeof(struct H_el *));",
1633 #if 0
1634    "       printf(\"hash table uses %%d bytes\\n\", ",
1635    "               (1<<ssize)*sizeof(struct H_el *));",
1636    "       fflush(stdout);",
1637 #endif
1638    "}\n",
1639    "#ifdef CACHE",
1640    "#include \"nh_store.c\"",
1641    "#endif",
1642
1643    "struct H_el *Free_list = 0;    /* recycles removed states  */",
1644    "",
1645    "recycle_state(v, n)",
1646    "       struct H_el *v;",
1647    "       short n;",
1648    "{",
1649    "       struct H_el *tmp, *last = 0;",
1650    "       v->tagged = n;",
1651    "       v->nxt = 0;",
1652    "       for (tmp = Free_list; tmp; last = tmp, tmp = tmp->nxt)",
1653    "       {       if (tmp->tagged >= n)",
1654    "               {       if (last)",
1655    "                       {       v->nxt = tmp->nxt;",
1656    "                               last->nxt = v;",
1657    "                       } else",
1658    "                       {       v->nxt = Free_list;",
1659    "                               Free_list = v;",
1660    "                       }",
1661    "                       return;",
1662    "               }",
1663    "               if (!tmp->nxt)",
1664    "               {       tmp->nxt = v;",
1665    "                       return;",
1666    "       }       }",
1667    "       Free_list = v;",
1668    "}",
1669    "",
1670    "struct H_el *",
1671    "grab_state(n)",
1672    "{      struct H_el *tmp, *last = 0;",
1673    "",
```

```
1674    "       for (tmp = Free_list; tmp; last = tmp, tmp = tmp->nxt)",
1675    "               if (tmp->tagged == n)",
1676    "               {       if (last)",
1677    "                               last->nxt = tmp->nxt;",
1678    "                       else",
1679    "                               Free_list = tmp->nxt;",
1680    "                       tmp->nxt = 0;",
1681    "                       recycled++;",
1682    "                       return tmp;",
1683    "               }",
1684    "       return (struct H_el *)",
1685    "             emalloc(sizeof(struct H_el)+n-sizeof(unsigned));",
1686    "}",
1687    "",
1688
1689    "htag(V, N)",
1690    "       char *V;",
1691    "       short N;",
1692    "{",
1693    "       register struct H_el *tmp, *last = 0;",
1694    "       char *v; short n;",
1695    "#ifdef COMPRESS",
1696    "       n = compress(&v, V, N);",
1697    "#else",
1698    "       n = N; v = V;",
1699    "#endif",
1700    "       s_hash(v, n);",
1701    "       for (tmp = H_tab[j1]; tmp; last = tmp, tmp = tmp->nxt)",
1702    "       {       if (",
1703    "#ifdef CACHE",
1704    "                       tmp->sz == n &&",
1705    "#endif",
1706    "                       memcmp(((char *)&(tmp->state)), v, n) == 0)",
1707    "               {",
1708    "#ifdef CACHE",
1709    "       if (tmp->tagged & (1<<31)) Uerror(\"Double Htag\");",
1710    "#endif",
1711    "                       tmp->tagged &= (1<<30); /* preserve only bit 30 */",
1712    "#if CONNECT==0",
1713    "#ifdef CACHE",
1714    "                       tmp->tagged |= (1<<31); /* set bit 31 */",
1715    "#endif",
1716    "#endif",
1717    "                       if (trpt->tau&2) /* state marked dirty: remove */",
1718    "                       {       if (last)",
1719    "                                       last->nxt = tmp->nxt;",
1720    "                               else",
1721    "                                       H_tab[j1] = tmp->nxt;",
1722    "                               recycle_state(tmp, n);",
1723    "                       }",
1724    "                       return;",
1725    "               }",
1726    "       }",
1727    "       for (tmp = H_tab[j1], n=0; tmp; tmp = tmp->nxt)",
```

```
1728    "               n++;",
1729    "       printf(\"cannot happen, htag, length of list is %%d\\n\", n);",
1730    "       fflush(stdout);",
1731    "/*     Uerror(\"cannot happen, htag\");        */",
1732    "}\n",
1733    "#ifdef COMPRESS",
1734 #include "compress.h"
1735    "#endif",
1736
1737    "#ifndef CACHE",
1738    "hstore(V, N)",
1739    "       char *V;",
1740    "       short N;",
1741    "{",
1742    "       register struct H_el *tmp;\n",
1743    "       char *v; short n;",
1744    "#ifdef COMPRESS",
1745    "       n = compress(&v, V, N);",
1746    "#else",
1747    "       n = N; v = V;",
1748    "#endif",
1749    "       if (Normalize) return 1;",
1750    "       s_hash((uchar *)v, n);",
1751    "       tmp = H_tab[j1];",
1752    "       if (!tmp)",
1753    "       {  tmp = grab_state(n);",
1754    "          H_tab[j1] = tmp;",
1755    "       } else",
1756    "       {  for (;; hcmp++)",
1757    "          {   if (memcmp(&(tmp->state), v, n) == 0)",
1758    "              {        if (tmp->tagged & ~((1<<30)|(1<<31)))",
1759    "                       {       if (loops && now._p_t)",
1760    "                               {       depthfound = tmp->tagged&~(1<<30);",
1761    "                                       if (fair_cycle())",
1762    "                                       uerror(\"non-progress cycle\");",
1763    "                               }",
1764    "                               return 2; /* match on stack */",
1765    "                       } else",
1766    "                               return 1; /* match outside stack */",
1767    "              }",
1768    "              if (!tmp->nxt) break;",
1769    "              tmp = tmp->nxt;",
1770    "          }",
1771    "          tmp->nxt = grab_state(n);",
1772    "          tmp = tmp->nxt;",
1773    "       }",
1774    "       tmp->tagged = depth+1; /* non-zero while on stack */",
1775    "       memcpy(((char *)&(tmp->state)), v, n);",
1776 #ifdef PAIRS
1777    "#ifdef PAIRS",
1778    "       if (boq == -1) pairs();",
1779    "#endif",
1780 #endif
1781    "       return 0;",
```

```
1782    "}",
1783    "#endif",
1784    "#endif",
1785    "#include \"pan.t\"",
1786    "",
1787    "do_reach()",
1788    "{",
1789    0,
1790 };
1791
1792 /***** spin: pangen2.h *****/
1793
1794 char *Preamble[] = {
1795    "#include       <stdio.h>",
1796    "#include       <signal.h>",
1797    "#include       \"pan.h\"\n",
1798    "#define max(a,b) (((a)<(b)) ? (b) : (a))",
1799    "typedef struct Trail {",
1800    "       short pr;       /* process id    */",
1801    "       short st;       /* current state */",
1802    "       uchar tau;      /* status flags */",
1803    "       char  o_n, o_ot, o_m;", /* to save locals */
1804    "       short o_tt, o_To;", /* used in new_state()  */
1805    "       Trans *o_t;",   /* transition fct, next state   */
1806    "       int oval;",     /* backup value of a variable */
1807    "} Trail;",
1808    "Trail *trail, *trpt;",
1809    "uchar *this;\n",
1810    "int maxdepth=10000;",
1811    "uchar *SS, *LL;",
1812    "char *emalloc(), *malloc(), *memset();",
1813    "int mreached=0, done=0, errors=0;",
1814    "long nstates=0, recycled=0;",
1815    "long nlinks=0, truncs=0, loss=0;",
1816    "int mask, hcmp=0, loops=0, acycles=0, upto=1;",
1817    "int state_tables=0, fairness=0, homomorphism=0;",
1818    "char *hom_target, *hom_source;",
1819 #ifdef PAIRS
1820    "int tree_before=0;",
1821 #endif
1822    "#ifdef BITSTATE",
1823    "int ssize=22;",
1824    "#else",
1825    "int ssize=18;",
1826    "#endif",
1827    "int hmax=0, svmax=0, smax=0;",
1828 #ifdef VARSTACK
1829    "int vmax=0;",
1830 #endif
1831 #ifdef GODEF
1832    "int cs_max=0;",
1833 #endif
1834    "int Maxbody=0;",
1835    "uchar *noptr;          /* used by macro Pptr(x) */",
```

```
1836    "State  A_Root;         /* root of acceptance cycles */",
1837    "State  now;            /* the full state vector */",
1838    "Stack  *stack;         /* for queues, processes */",
1839    "Svtack *svtack;        /* for old state vectors */",
1840 #ifdef VARSTACK
1841    "Varstack *varstack;    /* for old variable vals */",
1842 #endif
1843 #ifdef GODEF
1844    "CS_stack *cs_stack;    /* conflict sets */",
1845 #endif
1846    "int J1, J2, j1, j2, j3, j4;",
1847    "int A_depth=0;\n",
1848    "int depth=0;\n",
1849    "#if SYNC",
1850    "#define IfNotBlocked   if (boq != -1) continue;",
1851    "#define UnBlock        boq = -1",
1852    "#else",
1853    "#define IfNotBlocked   /* cannot block */",
1854    "#define UnBlock        /* don't bother */",
1855    "#endif\n",
1856    0,
1857 };
1858 char *Tail[] = {
1859    "Trans *",
1860    "settr(a, b, c, d, t, l, ntp)",
1861    "       char *t;",
1862    "{      Trans *tmp = (Trans *) emalloc(sizeof(Trans));\n",
1863    "       tmp->atom = a&6;",
1864    "       tmp->st = b;",
1865    "       tmp->local = l;",
1866    "       tmp->tp = t;",
1867    "       tmp->ntp = ntp;",
1868    "       tmp->forw = c;",
1869    "       tmp->back = d;",
1870 #ifdef GODEF
1871    "#ifdef VERBOSE",
1872    "       Moves[c] = t;",
1873    "#endif",
1874 #endif
1875    "       return tmp;",
1876    "}\n",
1877 #ifdef PAIRS
1878    "#define Visited 1<<12",
1879    "dfs(p, t, srcln)",
1880    "       short srcln[];",
1881    "{      Trans *n; char *wtyp();",
1882    "",
1883    "       if (t == 0 || (trans[p][t]->atom & Visited))",
1884    "       {       printf(\"%%dl\", t);",
1885    "               return;",
1886    "       }",
1887    "       trans[p][t]->atom |= Visited;",
1888    "       printf(\"%%d\\\"%%d%%s%%s\\\"d\", ",
1889    "               t, srcln[t],",
```

```
1890    "               (trans[p][t]->atom&2)?\" *\":\" \",",
1891    "               trans[p][t]->tp);",
1892    "       if (trans[p][t]->st)",
1893    "               dfs(p, trans[p][t]->st, srcln);",
1894    "       for (n = trans[p][t]->nxt; n; n = n->nxt)",
1895    "               dfs(p, n->st, srcln);",
1896    "       printf(\"u\");",
1897    "}\n",
1898    "Tree(p, strt, srcln)",
1899    "       short srcln[];",
1900    "{      printf(\"echo \");",
1901    "       dfs(p, strt, srcln);",
1902    "       printf(\" | 2.tree\\n\");",
1903    "}\n",
1904 #endif
1905    "#ifdef JUMBO",
1906    "#define Visited 1<<12",
1907    "#define Completed 1<<13",
1908    "int",
1909    "jumbo_list(p, t)",
1910    "{      int all_local; Trans *n;",
1911    "       int nl, nxt_local;",
1912
1913    "       if (t == 0) return 0;",
1914    "       if (trans[p][t]->atom & Visited)",
1915    "       {       if (trans[p][t]->atom & Completed)",
1916    "                       return trans[p][t]->Local;",
1917    "               else",
1918    "                       return 0;",
1919    "       }",
1920    "       n = trans[p][t];",
1921    "       n->atom |= Visited;",
1922
1923    "       nxt_local = jumbo_list(p, n->st);",
1924
1925    "       if (n->nxt && n->ntp != 'c')",
1926    "               all_local = 0;",
1927    "       else",
1928    "               all_local = n->local;",
1929
1930    "       for (n = n->nxt; n; n = n->nxt)",
1931    "       {       nl = jumbo_list(p, n->st);",
1932    "               if (nl < nxt_local) nxt_local = nl;",
1933    "               if (!n->local || n->ntp != 'c') all_local = 0;",
1934    "       }",
1935    "       if (all_local != 0)",
1936    "               all_local  = nxt_local + 1;",
1937    "       trans[p][t]->Local = all_local;",
1938    "       trans[p][t]->atom |= Completed;",
1939    "       return all_local;",
1940    "}",
1941    "#endif",
1942    "Trans *",
1943    "cpytr(a)",
```

```
1944    "       Trans *a;",
1945    "{      Trans *tmp = (Trans *) emalloc(sizeof(Trans));\n",
1946    "       tmp->atom = a->atom;",
1947    "       tmp->st = a->st;",
1948    "       tmp->ist = a->ist;",
1949    "       tmp->local = a->local;",
1950    "       tmp->forw = a->forw;",
1951    "       tmp->back = a->back;",
1952    "       tmp->tp = a->tp;",
1953    "       tmp->ntp = a->ntp;",
1954    "       return tmp;",
1955    "}\n",
1956    "int cnt;",
1957    "retrans(n, m, is, srcln, reach) /* proc n, m states, is=initial state */",
1958    "       short srcln[];",
1959    "       uchar reach[];",
1960    "{      Trans *T0, *T1, *T2, *T3;",
1961    "       int i, j=0;",
1962    "       if (state_tables == 2)",
1963    "       {       printf(\"RAW proctype %%s\\n\", ",
1964    "                       procname[n]);",
1965    "               for (i = 1; i < m; i++)",
1966    "                       reach[i] = 1;",
1967    "#ifdef JUMBO",
1968    "               jumbo_list(n, is);",
1969    "#endif",
1970    "               tagtable(n, m, is, srcln, reach);",
1971    "               return;",
1972    "       }",
1973    "       do {    j++;",
1974    "               for (i = 1, cnt = 0; i < m; i++)",
1975    "               {       T1 = trans[n][i]->nxt;",
1976    "                       T2 = trans[n][i];",
1977    "/* prescan: */         for (T0 = T1; T0; T0 = T0->nxt)",
1978    "/* choice inside choice */     if (trans[n][T0->st]->nxt)",
1979    "                                       break;",
1980    "                       if (T0)",
1981    "                       for (T0 = T1; T0; T0 = T0->nxt)",
1982    "                       {       T3 = trans[n][T0->st];",
1983    "                               if (!T3->nxt)",
1984    "                               {       T2->nxt = cpytr(T0);",
1985    "                                       T2 = T2->nxt;",
1986    "                                       imed(T2, T0->st, n);",
1987    "                                       continue;",
1988    "                               }",
1989    "                               do {    T3 = T3->nxt;",
1990    "                                       T2->nxt = cpytr(T3);",
1991    "                                       T2 = T2->nxt;",
1992    "                                       imed(T2, T0->st, n);",
1993    "                               } while (T3->nxt);",
1994    "                               cnt++;",
1995    "                       }",
1996    "               }",
1997    "       } while (cnt);",
```

```
1998    "       for (i = 1; i < m; i++)",
1999    "       if (trans[n][i]->nxt)   /* optimize the list a bit */",
2000    "       {       T1 = trans[n][i]->nxt;",
2001    "               T0 = trans[n][i] = cpytr(trans[n][T1->st]);",
2002    "               imed(T0, T1->st, n);",
2003    "               for (T1 = T1->nxt; T1; T1 = T1->nxt)",
2004    "               {       T0->nxt = cpytr(trans[n][T1->st]);",
2005    "                       T0 = T0->nxt;",
2006    "                       imed(T0, T1->st, n);",
2007    "       }       }",
2008    "more:",
2009    "       if (state_tables",
2010    "       || (homomorphism == 1 && strcmp(hom_target, procname[n]) == 0)",
2011    "       || (homomorphism == 2 && strcmp(hom_source, procname[n]) == 0))",
2012    "       {       if (n == 0 && homomorphism == 1)",
2013    "                       printf(\"\");",
2014    "               printf(\"proctype %%s%%s%%s%%s\",",
2015    "                       homomorphism==1?\"O\":\"\",",
2016    "                       homomorphism==2?\"R\":\"\",",
2017    "                       procname[n],",
2018    "                       homomorphism?\"()\\n{\\n\":\"\\n\");",
2019    "               for (i = 1; i < m; i++)",
2020    "                       reach[i] = 1;",
2021    "#ifdef JUMBO",
2022    "               jumbo_list(n, is);",
2023    "#endif",
2024    "               tagtable(n, m, is, srcln, reach);",
2025    "               if (!state_tables)",
2026    "                       printf(\"S0:    skip\\n}\\n\");",
2027    "       }",
2028    "       switch (homomorphism) {",
2029    "       case  1: homomorphism = 2; goto more;",
2030    "       case  2: homomorphism = 1; break;",
2031    "       default: break;",
2032    "       }",
2033    "}",
2034    "imed(T, v, n)  /* set intermediate state */",
2035    "       Trans *T;",
2036    "{      static uchar warned=0;",
2037    "       if (T->ist && !warned)",
2038    "       {       warned=1;",
2039    "               printf(\"warning: %%s has \", procname[n]);",
2040    "               printf(\"ambiguous flow ctl structures, \");",
2041    "               printf(\"revise model\\n\");",
2042    "       }",
2043    "       progstate[n][T->st] |= progstate[n][v];",
2044    "       accpstate[n][T->st] |= accpstate[n][v];",
2045    "       stopstate[n][T->st] |= stopstate[n][v];",
2046    "       T->ist = v;",
2047    "}",
2048    "tagtable(n, m, is, srcln, reach)",
2049    "       short srcln[];",
2050    "       uchar reach[];",
2051    "{",
```

```
2052    "       Trans *z;",
2053    "       if (is >= m || !trans[n][is]",
2054    "       ||  is <= 0 || reach[is] == 0)",
2055    "               return;",
2056    "       reach[is] = 0;",
2057    "       if (homomorphism)",
2058    "       {       if (accpstate[n][is])",
2059    "                       printf(\"accept_%%d:\\n\", is);",
2060    "               if (stopstate[n][is])",
2061    "                       printf(\"end_%%d:\\n\", is);",
2062    "               if (progstate[n][is])",
2063    "                       printf(\"progress_%%d:\\n\", is);",
2064    "               printf(\"S%%d:\", is);",
2065    "               if (homomorphism == 1)",
2066    "                       printf(\"\t!trans ->\\n\");",
2067    "               printf(\"\tif\\n\");",
2068    "               for (z = trans[n][is]; z; z = z->nxt)",
2069    "               {       printf(\"\t:: \");",
2070    "                       Crack(n, is, z, srcln);",
2071    "               }",
2072    "               printf(\"\tfi;\\n\");",
2073    "       } else",
2074    "               if (state_tables)",
2075    "               for (z = trans[n][is]; z; z = z->nxt)",
2076    "                       crack(n, is, z, srcln);",
2077    "       for (z = trans[n][is]; z; z = z->nxt)",
2078    "               tagtable(n, m, z->st, srcln, reach);",
2079    "}",
2080    "uniq_trans(str)",
2081    "       char *str;",
2082    "{      int j;",
2083    "       static int n_have=0;",
2084    "       typedef struct HAVE {",
2085    "               char *s;",
2086    "               struct HAVE *n;",
2087    "       } HAVE;",
2088    "       HAVE *t, *tt;",
2089    "       static HAVE *h = 0;",
2090    "       for (t = h, tt = 0, j = 0; t; tt = t, t = t->n, j++)",
2091    "               if (strcmp(t->s, str) == 0)",
2092    "                       return j;",
2093    "       t = (HAVE *) emalloc(sizeof(HAVE));",
2094    "       t->s = str;",
2095    "       if (!h)",
2096    "               h = t;",
2097    "       else",
2098    "               tt->n = t;",
2099    "       return j;",
2100    "}",
2101    "putsource(s)",
2102    "       char *s;",
2103    "{      int i;",
2104    "       for (i = 0; s[i]; i++)",
2105    "               if (s[i] == \'\\n\')",
```

```
2106    "                       printf(\"\\\\n\");",
2107    "               else",
2108    "                       putchar(s[i]);",
2109    "}",
2110    "Crack(n, j, z, srcln)",
2111    "       Trans *z;",
2112    "       short srcln[];",
2113    "{      int i;",
2114    "       if (!z) return;",
2115    "       i = 1+uniq_trans(z->tp);",
2116    "       if (z->atom & 6) i = -i;",
2117    "       if (strcmp(z->tp, \"(1)\") == 0)",
2118    "       {       printf(\"skip; goto S%%d        \", z->st);",
2119    "               printf(\"/* line %%3d */\\n\", srcln[j]);",
2120    "               return;",
2121    "       }",
2122    "#if 0",
2123    "       if (z->local && strcmp(z->tp, \"@\") != 0)",
2124    "       {       putsource(z->tp);",
2125    "               printf(\"; goto S%%d    \", z->st);",
2126    "               printf(\"/* line %%3d */\\n\", srcln[j]);",
2127    "               return;",
2128    "       }",
2129    "#endif",
2130    "       if (homomorphism == 1)",
2131    "               printf(\"atomic { trans = %%2d; \", i);",
2132    "       else /* homomorphism == 2 */",
2133    "               printf(\"atomic { (trans == %%2d); trans = 0; \", i);",
2134    "       printf(\"goto S%%d }\", z->st);",
2135    "       printf(\"       /* line %%3d, \", srcln[j]);",
2136    "       putsource(z->tp);",
2137    "       printf(\" */\\n\");",
2138    "       fflush(stdout);",
2139    "}",
2140    "crack(n, j, z, srcln)",
2141    "       Trans *z;",
2142    "       short srcln[];",
2143    "{      int i;",
2144    "       if (!z) return;",
2145    "printf(\"\\tstate %%2d --[%%2d]--> state %%2d [%%s%%s%%s%%s%%s] (%%d) line %%3d => \",",
2146    "       j, z->forw, z->st,",
2147    "       z->atom & 6 ?\"A\":\"-\",",
2148    "       z->local?\"L\" :\"-\",",
2149    "       accpstate[n][j]?\"a\" :\"-\",",
2150    "       stopstate[n][j]?\"e\" : \"-\",",
2151    "       progstate[n][j]?\"p\" : \"-\",",
2152    "       z->Local,",
2153    "       srcln[j]);",
2154    "       putsource(z->tp);",
2155    "       printf(\"\\n\");",
2156    "       fflush(stdout);",
2157    "}",
2158    0,
2159 };
```

```
2160
2161 /***** spin: pangen3.h *****/
2162
2163 char *R0[] = {
2164    "       Maxbody = max(Maxbody, sizeof(P%d));",
2165    "       reached[%d] = reached%d;",
2166    "       accpstate[%d] = (uchar *) emalloc(nstates%d);",
2167    "       progstate[%d] = (uchar *) emalloc(nstates%d);",
2168    "       stopstate[%d] = (uchar *) emalloc(nstates%d);",
2169    "       stopstate[%d][endstate%d] = 1;",
2170    0,
2171 };
2172 char *R0a[] = {
2173 #ifdef PAIRS
2174    "       if (tree_before) Tree(%d, start%d, src_ln%d);",
2175 #endif
2176    "       retrans(%d, nstates%d, start%d, src_ln%d, reached%d);",
2177    0,
2178 };
2179 char *R0b[] = {
2180    "       if (state_tables)",
2181    "       {       printf(\"\\nTransition Types:    \");",
2182    "               printf(\"A=atomic; L=local;\\n\");",
2183    "               printf(\"Source-State Labels: \");",
2184    "               printf(\"p=progress; e=end; a=accept;\\n\");",
2185    "       }",
2186    "       if (homomorphism)",
2187    "       printf(\"init { atomic { run O%%s(); run R%%s() } }\\n\",",
2188    "               hom_target, hom_source);",
2189    "       if (state_tables || homomorphism)",
2190    "               exit();",
2191    0,
2192 };
2193 char *R1[] = {
2194    "       reached[%d] = (uchar *) emalloc(4*sizeof(uchar));",
2195    "       stopstate[%d] = (uchar *) emalloc(4*sizeof(uchar));",
2196    "       progstate[%d] = stopstate[%d];",
2197    "       accpstate[%d] = stopstate[%d];",
2198    0,
2199 };
2200 char *R2[] = {
2201    "uchar *accpstate[%d];",
2202    "uchar *progstate[%d];",
2203    "uchar *reached[%d];",
2204    "uchar *stopstate[%d];",
2205    0,
2206 };
2207 char *R3[] = {
2208    "       Maxbody = max(Maxbody, sizeof(Q%d));",
2209    0,
2210 };
2211 char *R4[] = {
2212    "       r_ck(reached%d, nstates%d, %d, src_ln%d);",
2213    0,
```

```
2214 };
2215 char *R5[] = {
2216    "       case %d: j = sizeof(P%d); break;",
2217    0,
2218 };
2219 char *R6[] = {
2220    "       case %d: /* progress checker */",
2221    "               ((P%d *)pptr(h))->_t = %d;",
2222    "               ((P%d *)pptr(h))->_p = 1;",
2223    "               now._p_t = 0;",
2224    "               break;",
2225    "       }",
2226    "#ifdef VERI",
2227    "       if (h == 0 && !addproc(VERI))",
2228    "               return 0;",
2229    "#endif",
2230    "       if (h == 0 && loops && !addproc(%d))",
2231    "               return 0;",
2232    "#ifdef VERI",
2233    "       return (h>0)?h-loops-1:0;",
2234    "#else",
2235    "       return (h>0)?h-loops:0;",
2236    "#endif",
2237    "}\n",
2238    0,
2239 };
2240 char *R8[] = {
2241    "       case %d: j = sizeof(Q%d); break;",
2242    0,
2243 };
2244 char *R9[] = {
2245    "typedef struct Q%d {",
2246    "       uchar Qlen;     /* q_size */",
2247    "       uchar _t;       /* q_type */",
2248    "       struct {",
2249    0,
2250 };
2251 char *R10[] = {
2252    "typedef struct Q0 {\t/* generic q */",
2253    "       uchar Qlen, _t;",
2254    "} Q0;",
2255    0,
2256 };
2257 char *R12[] = {
2258    "\t\tcase %d: r = ((Q%d *)z)->contents[slot].fld%d; break;",
2259    0,
2260 };
2261 char *R13[] = {
2262    "unsend(into)",
2263    "{      int m=0, j; uchar *z;",
2264    "       if (!into--) uerror(\"reference to uninitialized chan name\");",
2265    "       z = qptr(into);",
2266    "       j = ((Q0 *)z)->Qlen;",
2267    "       ((Q0 *)z)->Qlen = --j;",
```

```
2268    "       switch (((Q0 *)qptr(into))->_t) {",
2269    0,
2270 };
2271 char *R14[] = {
2272    "       default: Uerror(\"bad queue - unsend\");",
2273    "       }",
2274    "       return m;",
2275    "}",
2276    "",
2277    "unrecv(from, slot, fld, fldvar, strt)",
2278    "{      int j; uchar *z;",
2279    "       if (!from--) uerror(\"reference to uninitialized chan name\");",
2280    "       z = qptr(from);",
2281    "       j = ((Q0 *)z)->Qlen;",
2282    "       if (strt) ((Q0 *)z)->Qlen = j+1;",
2283    "       switch (((Q0 *)qptr(from))->_t) {",
2284    0,
2285 };
2286 char *R15[] = {
2287    "       default: Uerror(\"bad queue - qrecv\");",
2288    "       }",
2289    "}",
2290    0,
2291 };
2292
2293 /***** spin: pangen1.c *****/
2294
2295 #include <stdio.h>
2296 #include <math.h>
2297 #include "spin.h"
2298 #include "y.tab.h"
2299 #include "pangen1.h"
2300 #include "pangen3.h"
2301
2302 extern FILE        *tc, *th;
2303 extern Node        *Mtype;
2304 extern ProcList    *rdy;
2305 extern Queue       *qtab;
2306 extern RunList     *run;
2307 extern Symbol      *symtab[Nhash+1];
2308 extern int nqs, nps, mst, Mpars;
2309 extern char        *claimproc;
2310
2311 enum { INIV, PUTV };
2312
2313 short Types[] = { BIT, BYTE, CHAN, SHORT, INT };
2314 int Npars=0, u_sync=0, u_async=0;
2315 int acceptors=0;
2316
2317 void
2318 genheader()
2319 {  ProcList *p;
2320    int i;
2321
```

```
2322    fprintf(th, "#define SYNC       %d\n", u_sync);
2323    fprintf(th, "#define ASYNC      %d\n\n", u_async);
2324    fprintf(tc, "char *procname[] = {\n");
2325    put_ptype(run->n->name, (Node *) 0, 0, mst, nps);
2326    for (p = rdy, i = 1; p; p = p->nxt, i++)
2327            put_ptype(p->n->name, p->p, i, mst, nps);
2328    put_ptype("_progress", (Node *) 0, i, mst, nps);
2329    fprintf(tc, "};\n\n");
2330    ntimes(th, 0, 1, Header);
2331    doglobal(PUTV);
2332    fprintf(th, "   uchar sv[VECTORSZ];\n");
2333    fprintf(th, "} State;\n");
2334 #ifdef GODEF
2335    {       Symbol *sp; extern int uniq, Maxcs;
2336            int j, k=0;
2337            fprintf(th, "\n/*** Conflict Set Numbers ***/\n");
2338            fprintf(th, "#define CS_timeout\t%d\n", k++);
2339            for (j = 0; j < 5; j++) /* for each data type */
2340            for (i = 0; i <= Nhash; i++)
2341            for (sp = symtab[i]; sp; sp = sp->next)
2342                    if (sp->type == Types[j])
2343                    {       if (sp->context)
2344                                    continue;
2345                            fprintf(th, "#define CS_%s\t%d\n", sp->name, k);
2346                            k += sp->nel;
2347                    }
2348            fprintf(th, "\nchar *CS_names[] = {\n");
2349            fprintf(th, "   \"timeout\",\n");
2350            if (k > 1)
2351            {       int a=0;
2352                    for (j = 0; j < 5; j++)
2353                    for (i = 0; i <= Nhash; i++)
2354                    for (sp = symtab[i]; sp; sp = sp->next)
2355                    if (sp->type == Types[j])
2356                    {       if (sp->context)
2357                                    continue;
2358                            if (sp->nel == 1)
2359                                    fprintf(th, "   \"%s\",\n", sp->name);
2360                            else
2361                                    for (a = 0; a < sp->nel; a++)
2362                                            fprintf(th, "   \"%s[%d]\",\n",
2363                                                    sp->name, a);
2364                    }
2365            }
2366            fprintf(th, "};\n");
2367            fprintf(th, "#define MAXSTATE   %d\n", uniq+2);
2368            /* added 2 for the two progress checker's states */
2369            fprintf(th, "#define TOPQ       (1+MAXCONFL+MAXQ)\n");
2370            fprintf(th, "/* Maxcs =\n");
2371            fprintf(th, " * max nr of cs that any 1 statement\n");
2372            fprintf(th, " * can be waiting for at any one time\n");
2373            fprintf(th, " */\n");
2374            fprintf(th, "#define MAXCS      %d\n", Maxcs);
2375            fprintf(th, "#define MAXCONFL   %d\n", k);
```

```
2376            fprintf(th, "#ifndef MULT\n");
2377            fprintf(th, "#define MULT       1\t/* max nr forks of a proc */\n");
2378            fprintf(th, "#endif\n");
2379            fprintf(th, "#if SYNC == 0\n");
2380            fprintf(th, "#define MULT_MAXCS (MULT*MAXCS)\n");
2381            fprintf(th, "#else\n");
2382            fprintf(th, "#define MULT_MAXCS (2*MULT*MAXCS)\n");
2383            fprintf(th, "#endif\n");
2384
2385            fprintf(tc, "#ifdef ALG3\n");
2386            fprintf(tc, "unsigned char Csels_c[MAXSTATE][MULT_MAXCS+1];\n");
2387            fprintf(tc, "unsigned char Csels_r[MAXSTATE][MULT_MAXCS+1];\n");
2388            fprintf(tc, "unsigned char Csels_p[MAXSTATE][MULT_MAXCS+1];\n");
2389            fprintf(tc, "char  csems[MAXPROC][TOPQ];\n");
2390            fprintf(tc, "short csets[MAXPROC][MAXSTATE];\n");
2391            fprintf(tc, "short Nwait=0, nwait[TOPQ];\n\n");
2392            fprintf(tc, "#endif\n");
2393            fprintf(th, "#ifdef VERBOSE\n");
2394            fprintf(th, "char *Moves[MAXSTATE];\n");
2395            fprintf(th, "#endif\n");
2396            fprintf(th, "#ifndef ALG3\n");
2397            fprintf(th, "#define push_act(p,s,w,h,t)        /* skip */\n");
2398            fprintf(th, "#define unrelease()               /* skip */\n");
2399            fprintf(th, "#define unpush()                  /* skip */\n");
2400            fprintf(th, "#define push_commit()             /* skip */\n");
2401            fprintf(th, "#define un_commit(p)              /* skip */\n");
2402            fprintf(th, "#endif\n");
2403    }
2404 #endif
2405 }
2406
2407 void
2408 genaddproc()
2409 {  ProcList *p;
2410    int i;
2411
2412    fprintf(tc, "addproc(n");
2413    for (i = 0; i < Npars; i++)
2414            fprintf(tc, ", par%d", i);
2415
2416    ntimes(tc, 0, 1, Addp0);
2417    ntimes(tc, 1, nps, R5);
2418    ntimes(tc, 0, 1, Addp1);
2419
2420    put_pinit(run->pc, run->n, (Node *) 0, 0);
2421    for (p = rdy, i = 1; p; p = p->nxt, i++)
2422            put_pinit(p->s->frst, p->n, p->p, i);
2423
2424    ntimes(tc, i, i+1, R6);
2425 }
2426
2427 void
2428 genother(cnt)
2429 {  ProcList *p;
```

```
2430    int i;
2431
2432    ntimes(tc,   0,   1, Code0);
2433    ntimes(tc,   0, cnt,   R0);
2434    ntimes(tc, cnt, cnt+1, R1);
2435    end_labs(run->n, 0);
2436    for (p = rdy, i = 1; p; p = p->nxt, i++)
2437            end_labs(p->n, i);
2438    ntimes(tc, 0, cnt, R0a);
2439    ntimes(tc, 0,   1, R0b);
2440
2441 #ifdef GODEF
2442    fprintf(tc, "\ttratable[%d] = _TRA_%d;  /* progress */\n", i, i);
2443 #endif
2444 #ifdef PAIRS
2445    fprintf(tc, "   if (tree_before) exit(0);\n");
2446 #endif
2447    ntimes(th, acceptors, acceptors+1, Code1);
2448    ntimes(th, i+1, i+2, R2);
2449
2450    doglobal(INIV);
2451    ntimes(tc, 1, nqs+1, R3);
2452    ntimes(tc, 0,     1, Code2);
2453    ntimes(tc, 0,     i, R4);
2454    fprintf(tc, "}\n\n");
2455 #ifdef PAIRS
2456    putpairs();
2457 }
2458
2459 putpairs()
2460 {  ProcList *p; int i;
2461    fprintf(tc, "#ifdef PAIRS\n");
2462    fprintf(tc, "pairs()\n");
2463    fprintf(tc, "{  int i,j; P0 *ptr;\n");
2464    fprintf(tc, "   for (i=1, j=0; i < now._nr_pr; i++)\n");
2465    fprintf(tc, "   {       ptr = (P0 *) pptr(i);\n");
2466    fprintf(tc, "#ifdef VERI\n");
2467    fprintf(tc, "           if (i == 1) continue;\n");
2468    fprintf(tc, "           if (i > 2) printf(\" \");\n");
2469    fprintf(tc, "#else\n");
2470    fprintf(tc, "           if (i > 1) printf(\" \");\n");
2471         fprintf(tc, "           else printf(\"NEW state %%d: \", nstates);\n");
2472            fprintf(tc, "#endif\n");
2473    fprintf(tc, "           switch(ptr->_t) {\n");
2474    for (p = rdy, i = 1; p; p = p->nxt, i++)
2475    {       fprintf(tc, "\t\t\tcase %d:", i);
2476            fprintf(tc, " printf(\"%%d\", ptr->_p /* src_ln%d[ptr->_p] */);", i);
2477            fprintf(tc, " j++; break;\n");
2478    }
2479    fprintf(tc, "   }       }\n");
2480    fprintf(tc, "   if (j) printf(\"\\n\");\n");
2481    fprintf(tc, "}\n");
2482    fprintf(tc, "#endif\n\n");
2483 #endif
```

```
2484 }
2485
2486 static struct {
2487    char *s, *t; int n, m;
2488 } ln[] = {
2489    "end",          "stopstate",    3,      0,
2490    "progress",     "progstate",    8,      0,
2491    "accept",       "accpstate",    6,      1,
2492    0,              0,              0,      0,
2493 };
2494
2495 void
2496 end_labs(s, i)
2497    Symbol *s;
2498 {
2499    extern Label *labtab;
2500    Label *l;
2501    int j;
2502 #ifdef GODEF
2503    fprintf(tc, "\ttratable[%d] = _TRA_%d;  /* %s */\n", i, i, s->name);
2504 #endif
2505    for (l = labtab; l; l = l->nxt)
2506    for (j = 0; ln[j].n; j++)
2507            if (strncmp(l->s->name, ln[j].s, ln[j].n) == 0
2508            &&  strcmp(l->s->context->name, s->name) == 0)
2509            {       fprintf(tc, "\t%s[%d][%d] = 1;\n",
2510                            ln[j].t, i, l->e->seqno);
2511                    acceptors += ln[j].m;
2512            }
2513 }
2514
2515 void
2516 ntimes(fd, n, m, c)
2517    FILE *fd;
2518    char *c[];
2519 {
2520    int i, j;
2521    for (j = 0; c[j]; j++)
2522    for (i = n; i < m; i++)
2523    {       fprintf(fd, c[j], i, i, i, i, i);
2524            fprintf(fd, "\n");
2525    }
2526 }
2527
2528 void
2529 dolocal(dowhat, p, s)
2530    char *s;
2531 {
2532    int i, j;
2533    Symbol *sp;
2534    char buf[64];
2535
2536    for (j = 0; j < 5; j++)
2537    for (i = 0; i <= Nhash; i++)
```

```
2538    for (sp = symtab[i]; sp; sp = sp->next)
2539            if (sp->context && sp->type == Types[j]
2540            &&  strcmp(s, sp->context->name) == 0)
2541            {       sprintf(buf, "((P%d *)pptr(h))->", p);
2542                    do_var(dowhat, buf, sp);
2543            }
2544 }
2545
2546 void
2547 doglobal(dowhat)
2548 {  Symbol *sp;
2549    int i, j;
2550
2551    for (j = 0; j < 5; j++)
2552    for (i = 0; i <= Nhash; i++)
2553    for (sp = symtab[i]; sp; sp = sp->next)
2554            if (!sp->context && sp->type == Types[j])
2555                    do_var(dowhat, "now.", sp);
2556 }
2557
2558 void
2559 do_var(dowhat, s, sp)
2560    char *s;
2561    Symbol *sp;
2562 {
2563    int i;
2564
2565    switch(dowhat) {
2566    case PUTV:
2567            typ2c(sp);
2568            break;
2569    case INIV:
2570            if (!sp->ini)
2571                    break;
2572            if (sp->nel == 1)
2573            {       fprintf(tc, "\t\t%s%s = ", s, sp->name);
2574                    do_init(sp);
2575            } else
2576            for (i = 0; i < sp->nel; i++)
2577            {       fprintf(tc, "\t\t%s%s[%d] = ", s, sp->name, i);
2578                    do_init(sp);
2579            }
2580            break;
2581    }
2582 }
2583
2584 void
2585 do_init(sp)
2586    Symbol *sp;
2587 {
2588    int i;
2589
2590    if (sp->type == CHAN && ((i = qmake(sp)) > 0))
2591            fprintf(tc, "addqueue(%d);\n", i);
```

```
2592    else
2593            fprintf(tc, "%d;\n", eval(sp->ini));
2594 }
2595
2596 blog(n)    /* for small log2 without rounding problems */
2597 {  int m=1, r=2;
2598    while (r < n) { m++; r *= 2; }
2599    return 1+m;
2600 }
2601
2602 void
2603 put_ptype(s, p, i, m0, m1)
2604    char *s;
2605    Node *p;
2606 {
2607    Node *fp, *fpt;
2608    int j;
2609    fprintf(tc, "   \"%s\",\n", s);
2610    fprintf(th, "typedef struct P%d { /* %s */\n", i, s);
2611    fprintf(th, "   unsigned _t : %d; /* proctype */\n", blog(m1));
2612    fprintf(th, "   unsigned _p : %d; /* state    */\n", blog(m0));
2613    dolocal(PUTV, i, s);    /* includes pars */
2614    fprintf(th, "} P%d;\n", i);
2615
2616    for (fp  = p, j = 0;  fp;  fp = fp->rgt)
2617    for (fpt = fp->lft;  fpt; fpt = fpt->rgt)
2618            j++;    /* count # of parameters */
2619    Npars = max(Npars, j);
2620 }
2621
2622 void
2623 put_pinit(e, s, p, i)
2624    Element *e;
2625    Symbol *s;
2626    Node *p;
2627 {
2628    Node *fp, *fpt;
2629    int ini, j;
2630
2631    ini = huntele(e, e->status)->seqno;
2632    fprintf(th, "#define start%d    %d\n", i, ini);
2633
2634    fprintf(tc, "\tcase %d: /* %s */\n", i, s->name);
2635    fprintf(tc, "\t\t((P%d *)pptr(h))->_t = %d;\n", i, i);
2636    fprintf(tc, "\t\t((P%d *)pptr(h))->_p = %d;", i, ini);
2637    fprintf(tc, " reached%d[%d]=1;\n", i, ini);
2638    dolocal(INIV, i, s->name);
2639    for (fp  = p, j=0; fp; fp = fp->rgt)
2640    for (fpt = fp->lft; fpt; fpt = fpt->rgt, j++)
2641    {       if (fpt->nsym->nel != 1)
2642            fatal("array in parameter list, %s", fpt->nsym->name);
2643            fprintf(tc, "\t\t((P%d *)pptr(h))->%s = par%d;\n",
2644                            i, fpt->nsym->name, j);
2645    }
```

```
2646    fprintf(tc, "\t break;\n");
2647 }
2648
2649 huntstart(f)
2650    Element *f;
2651 {
2652    Element *e = f;
2653
2654    if (e->n)
2655    {       if (e->n->ntyp==’.’ && e->nxt)
2656                    e = e->nxt;
2657            else if (e->n->ntyp == ATOMIC)
2658                    e->n->seql->this->last->nxt = e->nxt;
2659    }
2660    return e->seqno;
2661 }
2662
2663 Element *
2664 huntele(f, o)
2665    Element *f;
2666 {
2667    Element *g, *e = f;
2668    int cnt; /* a precaution against loops */
2669    for (cnt=0; cnt < 10 && e->n; cnt++)
2670    {       switch (e->n->ntyp) {
2671            case GOTO:
2672                    g = get_lab(e->n->nsym);
2673                    break;
2674            case ’.’:
2675            case BREAK:
2676                    if (!e->nxt)
2677                            return e;
2678                    g = e->nxt;
2679                    break;
2680            case ATOMIC:
2681                    e->n->seql->this->last->nxt = e->nxt;
2682            default:        /* fall through */
2683                    return e;
2684            }
2685            if ((o & ATOM) && !(g->status & ATOM))
2686                    return e;
2687            e = g;
2688    }
2689    return e;
2690 }
2691
2692 void
2693 typ2c(sp)
2694    Symbol *sp;
2695 {
2696    switch (sp->type) {
2697    case BIT:
2698            if (sp->nel == 1)
2699            {       fprintf(th, "\tunsigned %s : 1", sp->name);
```

```
2700                    break;
2701            } /* else fall through */
2702    case CHAN:      /* good for up to 255 channels */
2703    case BYTE:
2704            fprintf(th, "\tuchar %s", sp->name);
2705            break;
2706    case SHORT:
2707            fprintf(th, "\tshort %s", sp->name);
2708            break;
2709    case INT:
2710            fprintf(th, "\tint %s", sp->name);
2711            break;
2712    case PREDEF:
2713            return;
2714    default:
2715            fatal("variable %s undeclared", sp->name);
2716    }
2717    if (sp->nel != 1)
2718            fprintf(th, "[%d]", sp->nel);
2719    fprintf(th, ";\n");
2720 }
2721
2722 void
2723 ncases(fd, p, n, m, c)
2724    FILE *fd;
2725    char *c[];
2726 {
2727    int i, j;
2728    for (j = 0; c[j]; j++)
2729    for (i = n; i < m; i++)
2730    {       fprintf(fd, c[j], i, p, i);
2731            fprintf(fd, "\n");
2732    }
2733 }
2734
2735 void
2736 genaddqueue()
2737 {  char *buf0;
2738    int j;
2739    Queue *q;
2740
2741    buf0 = (char *) emalloc(32);
2742    ntimes(tc, 0, 1, Addq0);
2743    for (q = qtab; q; q = q->nxt)
2744    {       ntimes(tc, q->qid, q->qid+1, R8);
2745            ntimes(th, q->qid, q->qid+1, R9);
2746            for (j = 0; j < q->nflds; j++)
2747            {       switch (q->fld_width[j]) {
2748                    case BIT:
2749                            if (q->nflds != 1)
2750                            {       fprintf(th, "\t\tunsigned");
2751                                    fprintf(th, " fld%d : 1;\n", j);
2752                                    break;
2753                            } /* else fall through: gives smaller struct */
```

```
2754                    case CHAN:
2755                    case BYTE:
2756                            fprintf(th, "\t\tuchar fld%d;\n", j);
2757                            break;
2758                    case SHORT:
2759                            fprintf(th, "\t\tshort fld%d;\n", j);
2760                            break;
2761                    case INT:
2762                            fprintf(th, "\t\tint fld%d;\n", j);
2763                            break;
2764                    default:
2765                            fatal("bad channel spec", "");
2766                    }
2767            }
2768            fprintf(th, "   } contents[%d];\n", max(1, q->nslots));
2769            fprintf(th, "} Q%d;\n", q->qid);
2770    }
2771    ntimes(th, 0, 1, R10);
2772    ntimes(tc, 0, 1, Addq1);
2773
2774    fprintf(tc, "qsend(into");
2775    for (j = 0; j < Mpars; j++)
2776            fprintf(tc, ", fld%d", j);
2777    fprintf(tc, ")\n");
2778    ntimes(tc, 0, 1, Addq11);
2779
2780    for (q = qtab; q; q = q->nxt)
2781    {       sprintf(buf0, "((Q%d *)z)->", q->qid);
2782            fprintf(tc, "\tcase %d: j = %sQlen;\n", q->qid, buf0);
2783            fprintf(tc, "\t\t%sQlen = j+1;\n", buf0);
2784            if (q->nslots == 0)     /* reset handshake point */
2785                    fprintf(tc, "\t\t(trpt+2)->o_m = 0;\n");
2786            sprintf(buf0, "((Q%d *)z)->contents[j].fld", q->qid);
2787            for (j = 0; j < q->nflds; j++)
2788                    fprintf(tc, "\t\t%s%d = fld%d;\n", buf0, j, j);
2789            fprintf(tc, "\t\tbreak;\n");
2790    }
2791    ntimes(tc, 0, 1, Addq2);
2792
2793    for (q = qtab; q; q = q->nxt)
2794    fprintf(tc, "\tcase %d: return %d;\n", q->qid, (!q->nslots));
2795
2796    ntimes(tc, 0, 1, Addq3);
2797
2798    for (q = qtab; q; q = q->nxt)
2799    fprintf(tc, "\tcase %d: return (q_sz(from) == %d);\n",
2800                    q->qid, max(1, q->nslots));
2801
2802    ntimes(tc, 0, 1, Addq4);
2803    for (q = qtab; q; q = q->nxt)
2804    {       sprintf(buf0, "((Q%d *)z)->", q->qid);
2805            fprintf(tc, "   case %d:", q->qid);
2806            if (q->nflds == 1)
2807            {       fprintf(tc, "\tif (fld == 0) r = %s", buf0);
```

```
2808                    fprintf(tc, "contents[slot].fld0;\n");
2809            } else
2810            {       fprintf(tc, "\tswitch (fld) {\n");
2811                    ncases(tc, q->qid, 0, q->nflds, R12);
2812                    fprintf(tc, "\t\t}\n");
2813            }
2814            fprintf(tc, "\t\tif (done)\n");
2815            fprintf(tc, "\t\t{      j = %sQlen;\n",  buf0);
2816            fprintf(tc, "\t\t       %sQlen = --j;\n", buf0);
2817            fprintf(tc, "\t\t       for (k=0; k<j; k++)\n", q->qid);
2818            fprintf(tc, "\t\t       {\n");
2819
2820            sprintf(buf0, "\t\t\t((Q%d *)z)->contents", q->qid);
2821            for (j = 0; j < q->nflds; j++)
2822            {       fprintf(tc, "\t%s[k].fld%d = \n", buf0, j);
2823                    fprintf(tc, "\t\t%s[k+1].fld%d;\n", buf0, j);
2824            }
2825            fprintf(tc, "\t\t       }\n");
2826            for (j = 0; j < q->nflds; j++)
2827                    fprintf(tc, "%s[j].fld%d = 0;\n", buf0, j);
2828            fprintf(tc, "\t\t\tif (fld+1 != %d)\n\t\t\t", q->nflds);
2829            fprintf(tc, "\tuerror(\"missing pars in receive\");\n");
2830            /* incompletely received msgs cannot be unrecv'ed */
2831            fprintf(tc, "\t\t}\n");
2832            fprintf(tc, "\t\tbreak;\n");
2833    }
2834    ntimes(tc, 0, 1, Addq5);
2835 }
2836
2837 /***** spin: pangen2.c *****/
2838
2839 #include <stdio.h>
2840 #include "spin.h"
2841 #include "y.tab.h"
2842 #include "pangen2.h"
2843
2844 extern ProcList    *rdy;
2845 extern RunList     *run;
2846 extern Symbol      *Fname;
2847 extern char        *claimproc;
2848 extern int lineno;
2849 extern int Mpars;
2850 extern int m_loss;
2851 int                Globalname;
2852
2853 #ifdef GODEF
2854 void push_cs();
2855 void push_loss();
2856 void putbase();
2857 void putindex();
2858 void coll_global();
2859 void coll_base();
2860 void coll_cs();
2861 void coll_indx();
```

```
2862 int aMarked=0, Marked=0, Countm=0, Maxcs=0;
2863 #endif
2864
2865 FILE       *tc, *th, *tt, *tm, *tb, *tf;
2866 int        uniq=1;
2867 int        nocast=0;       /* to turn off casts in lvalues */
2868 int        terse=0;        /* terse printing of varnames */
2869 int        nps=0;          /* number of processes */
2870 int        mst=0;          /* max nr of state/process */
2871 int        claimnr = -1;   /* claim process, if any */
2872 int        Pid;            /* proc currently processed */
2873 int        EVAL_runs = 0;  /* used in fairness checks */
2874
2875 #ifdef VARSTACK
2876 int        Cksum;          /* debugging only */
2877 #endif
2878
2879 fproc(s)
2880    char *s;
2881 {
2882    ProcList *p;
2883    int i;
2884
2885    if (strcmp("_init", s) == 0)
2886            return 0;
2887    for (p = rdy, i = 1; p; p = p->nxt, i++)
2888            if (strcmp(p->n->name, s) == 0)
2889                    return i;
2890    fatal("proctype %s not found", s);
2891 }
2892
2893 void
2894 gensrc()
2895 {  ProcList *p;
2896    int i;
2897
2898    if (!(tc = fopen("pan.c", "w"))         /* main routines */
2899    ||  !(th = fopen("pan.h", "w"))         /* header file   */
2900    ||  !(tt = fopen("pan.t", "w"))         /* transition matrix */
2901    ||  !(tm = fopen("pan.m", "w"))         /* forward  moves */
2902    ||  !(tf = fopen("pan.f", "w"))         /* fairness checks */
2903    ||  !(tb = fopen("pan.b", "w")))        /* backward moves */
2904    {       printf("spin: cannot create pan.[chtmb]\n");
2905            exit(1);
2906    }
2907    fprintf(th, "/*** %s ***/\n", Fname->name);
2908    fprintf(th, "#define uchar      unsigned char\n");
2909    if (claimproc)
2910    {       claimnr = fproc(claimproc);
2911            fprintf(th, "#define VERI       %d\n",  claimnr);
2912            fprintf(th, "#define claimline");
2913            fprintf(th, "   src_ln%d[((P0 *)pptr(1))->_p]\n",
2914                                                    claimnr);
2915    }
```

```
2916    fprintf(th, "#define M_LOSS     %d\n", m_loss);
2917    fprintf(th, "#define endclaim   endstate%d\n",  claimnr);
2918    ntimes(tc, 0, 1, Preamble);
2919
2920    fprintf(tc, "#ifndef NOBOUNDCHECK\n");
2921    fprintf(tc, "#define Index(x, y)        Boundcheck(x, y, II, tt, t)\n");
2922    fprintf(tc, "#else\n");
2923    fprintf(tc, "#define Index(x, y)        x\n");
2924    fprintf(tc, "#endif\n");
2925
2926    mst = (run)?run->maxseq:0;
2927    for (p = rdy, i = 1; p; p = p->nxt, i++)
2928            mst = max(p->s->last->seqno, mst);
2929    nps = i+1;      /* add progress checker */
2930
2931    fprintf(tt, "settable()\n{\tTrans *T, *settr();\n\n");
2932    fprintf(tt, "#ifdef VERBOSE\n");
2933    fprintf(tt, "\tMoves[0] = \"bad move\";\n");
2934    fprintf(tt, "#endif\n");
2935    fprintf(tt, "\ttrans = (Trans ***) ");
2936    fprintf(tt, "emalloc(%d*sizeof(Trans **));\n", nps);
2937
2938    fprintf(tm, "   switch (t->forw) {\n");
2939    fprintf(tm, "   default: Uerror(\"bad forward move\");\n");
2940
2941    fprintf(tb, "   switch (t->back) {\n");
2942    fprintf(tb, "   default: Uerror(\"bad return move\");\n");
2943    fprintf(tb, "   case  0: goto R999; /* nothing to undo */\n");
2944
2945    fprintf(tf, "   switch (t->forw) {\n");
2946    fprintf(tf, "   default: continue;\n");
2947
2948    if (!run) fatal("no runable process", (char *)0);
2949
2950    putproc(run->n, run->pc, 0, run->maxseq);
2951    for (p = rdy, i = 1; p; p = p->nxt, i++)
2952            putproc(p->n, p->s->frst, i, p->s->last->seqno);
2953    putprogress(i, 2);
2954 #ifdef GODEF
2955    fprintf(th, "#define _TRA_%d    %d      /* progress */\n", i, uniq);
2956    fprintf(th, "#define _TRA_%d    %d      /* end */\n", i+1, uniq+2);
2957 #endif
2958    ntimes(tt, 0, 1, Tail);
2959    genheader();
2960    genaddproc();
2961    genother(i);
2962    genaddqueue();
2963    genunio();
2964
2965    putsyms(tc, th);
2966 }
2967
2968 void
2969 putproc(n, e, i, j)
```

```
2970    Symbol *n;
2971    Element *e;
2972 {
2973    Pid = i;
2974 #ifdef GODEF
2975    fprintf(th, "#define _TRA_%d    %d      /* %s */\n", i, uniq, n->name);
2976 #endif
2977    fprintf(th, "\nshort nstates%d=%d;\t/* %s */\n", i,j+1,n->name);
2978    fprintf(tm, "\n          /* PROC %s */\n", n->name);
2979    fprintf(tb, "\n          /* PROC %s */\n", n->name);
2980    fprintf(tt, "\n /* proctype %d: %s */\n", i, n->name);
2981    fprintf(tt, "\n trans[%d] = (Trans **)", i);
2982    fprintf(tt, " emalloc(%d*sizeof(Trans *));\n\n", j+1);
2983    putseq(e, 0);
2984    dumpsrc(j, i);
2985 }
2986
2987 void
2988 putprogress(i, j)  /* loop detector */
2989 {
2990    fprintf(th, "\nshort nstates%d=%d;\t/* _progress */\n", i, j+1);
2991
2992    fprintf(tt, "\n /* proctype %d: _progress */\n", i);
2993    fprintf(tt, "\n trans[%d] = (Trans **)", i);
2994    fprintf(tt, " emalloc(%d*sizeof(Trans *));\n\n", j+1);
2995    fprintf(tt, "   trans[%d][1]    = settr(1,2,%d,%d,\"-\",0,0);\n",
2996                                        i, uniq, uniq);
2997    fprintf(tt, "   trans[%d][2]    = settr(1,0,%d,%d,\"-\",0,0);\n",
2998                                        i, uniq+1, uniq+1);
2999    fprintf(tt, "}\n");
3000
3001    fprintf(tm, "\n          /* _progress */\n");
3002    fprintf(tm, "   case %d:        /* progress */\n", uniq);
3003    fprintf(tm, "           IfNotBlocked\n");
3004    fprintf(tm, "           now._p_t = 13;  /* 13 to help the hasher */\n");
3005    fprintf(tm, "           m = 3; goto P999;\n");
3006    fprintf(tm, "   case %d:\n", uniq+1);
3007    fprintf(tm, "           continue;\n");
3008    fprintf(tm, "   }\n\n");
3009
3010    fprintf(tb, "\n          /* _progress */\n");
3011    fprintf(tb, "   case %d:        /* progress */\n", uniq);
3012    fprintf(tb, "           now._p_t = 0;\n");
3013    fprintf(tb, "           goto R999;\n");
3014    fprintf(tb, "   case %d:\n", uniq+1);
3015    fprintf(tb, "           goto R999;\n");
3016    fprintf(tb, "   }\n\n");
3017
3018    fprintf(tf, "   }\n");
3019 }
3020
3021 void
3022 putseq(f, level)
3023    Element *f;
```

```
3024 {
3025    Element *e;
3026
3027    for (e = f; e; e = e->nxt)
3028            putseq_el(e, level+1);
3029 }
3030
3031 void
3032 putseq_lst(s, level)
3033    Sequence *s;
3034 {
3035    Element *g;
3036
3037    for (g = s->frst; ; g = g->nxt)
3038    {       if (!g)
3039            {       fprintf(stderr, "cannot happen seq_lst\n");
3040                    exit(1);
3041            }
3042            putseq_el(g, level+1);
3043            if (g == s->last)
3044                    break;
3045    }
3046 }
3047
3048 void
3049 putseq_el(e, level)
3050    Element *e;
3051 {
3052    SeqList *h;
3053    int n, a, bu;
3054
3055    if (e->status & DONE)
3056            return;
3057    e->status |= DONE;
3058    if (e->n->nval)
3059            putsrc(e->n->nval, e->seqno);
3060    if (e->sub)
3061    {       int oMarked, oaMarked; atom_stack *oCS, *save_ast();
3062            fprintf(tt, "\tT = trans[%d][%d] = ",
3063                                            Pid, e->seqno);
3064            fprintf(tt, "settr(%d,0,0,0,\"", e->status);
3065            comment(tt, e->n, e->seqno);
3066            fprintf(tt, "\",0,%d);\t/* line %d (%d,%d) */\n",
3067                    e->n->ntyp,
3068                    e->n->nval, Marked, level);
3069            for (h = e->sub; h; h = h->nxt)
3070            {       putskip(h->this->frst->seqno);
3071                    a = huntstart(h->this->frst);
3072                    fprintf(tt, "\tT = T->nxt\t= ");
3073                    fprintf(tt, "settr(%d,%d,0,0,\"",
3074                                            e->status, a, e->n->ntyp);
3075                    comment(tt, e->n, e->seqno);
3076                    fprintf(tt, "\", %d, %d);\t/* line %d (%d,%d) */\n",
3077                            1-Marked, e->n->ntyp,
```

```
3078                            e->n->nval, Marked, level);
3079            }
3080 #if 0
3081            oMarked = Marked;
3082            oCS = save_ast();
3083 #endif
3084            oaMarked = aMarked;
3085            for (h = e->sub; h; h = h->nxt)
3086            {
3087                    aMarked = oaMarked;
3088 #if 1
3089                    if (aMarked)
3090                    {       clear_ast();
3091                            coll_global(h->this, 0);
3092                            Marked = has_ast();
3093                    }
3094 #else
3095                    Marked  = oMarked;
3096                    restor_ast(oCS);
3097 #endif
3098                    putseq_lst(h->this, level);
3099            }
3100    } else
3101    {       if (e->n && e->n->ntyp == ATOMIC)
3102            {       patch_atomic(e->n->seql->this);
3103                    putskip(e->n->seql->this->frst->seqno);
3104                    a = huntstart(e->n->seql->this->frst);
3105                    fprintf(tt, "\tT = trans[%d][%d] = ", Pid, e->seqno);
3106                    fprintf(tt, "settr(%d,0,0,0,\"", ATOM, e->n->ntyp);
3107                    comment(tt, e->n, e->seqno);
3108                    fprintf(tt, "\", 0, %d);\t/* line %d */\n",
3109                            e->n->ntyp, e->n->nval);
3110                    fprintf(tt, "\t    T->nxt\t= ");
3111
3112                    fprintf(tt, "settr(%d,%d,0,0,\"", ATOM, a, e->n->ntyp);
3113                    comment(tt, e->n, e->seqno);
3114                    fprintf(tt, "\", 0, %d);\t/* line %d (%d,%d) */\n",
3115                            e->n->ntyp, e->n->nval, Marked, level);
3116                    e->n->seql->this->last->nxt = e->nxt;
3117 #ifdef GODEF
3118                    /*
3119                     * if the statements in an atomic sequence
3120                     * touch global objects, the guard(s) must
3121                     * be labeled with all conflict sets touched
3122                     */
3123                    if (has_ast())
3124                    {       fprintf(stderr, "internal error: ast stack\n");
3125                            pop_ast(stderr, 0);
3126                            exit(1);
3127                    }
3128                    coll_global(e->n->seql->this, 0);
3129                    Marked = has_ast();
3130                    aMarked = 1;
3131 #endif
```

```
3132                    putseq_lst(e->n->seql->this, level);
3133 #ifdef GODEF
3134                    Marked = 0; aMarked = 0;
3135 #endif
3136                    return;
3137            }
3138            if (e->n->ntyp == GOTO)
3139                    a = huntele(get_lab(e->n->nsym),
3140                                    e->status)->seqno;
3141            else if (e->nxt)
3142                    a = huntele(e->nxt, e->status)->seqno;
3143            else
3144                    a = 0;
3145            fprintf(tt, "\tT = trans[%d][%d]\t= ",
3146                                    Pid, e->seqno);
3147
3148            putfair(tf, e->n, e->seqno, uniq);
3149
3150            fprintf(tm, "\tcase  %d: /* STATE ", uniq++);
3151            fprintf(tm, "%d, ", e->seqno);
3152            comment(tm, e->n, e->seqno);
3153            fprintf(tm, ", line %d (Marked %d, level %d, status %d/%d) */\n\t\t",
3154                    e->n->nval, Marked, level, e->status, e->status&ATOM);
3155            if (e->n && e->n->ntyp != 'r' && Pid != claimnr)
3156                    fprintf(tm, "IfNotBlocked\n\t\t");
3157            putstmnt(tm, e->n, e->seqno);
3158 #ifdef VARSTACK
3159 /*
3160  * warning: checklast() in dflow.c
3161  * is an untrusted optimization
3162  *                 Cksum=rand();
3163  *                 if (Pid != claimnr)
3164  *                         bu = checklast(tm, e->n, e->nxt, e->seqno, 1);
3165  *                 else
3166  */
3167 #endif
3168                    bu = 0;
3169            n = getweight(e->n);
3170            fprintf(tm, ";\n\t\tm = %d;\n\t\t", n);
3171 #ifdef GODEF
3172            Countm = 0;
3173            if (any_cs(e->n)) push_cs(tm, e->n, 0);
3174            Maxcs = max(Countm, Maxcs);
3175 #endif
3176            if (bu == 0) bu = 2;
3177            fprintf(tm, "goto P999;\n", n);
3178            if (bu || any_undo(e->n))
3179            {
3180                    fprintf(tb, "\tcase  %d: ", uniq-1);
3181                    fprintf(tb, "/* STATE ");
3182                    fprintf(tb, "%d, ", e->seqno);
3183                    comment(tb, e->n, e->seqno);
3184            fprintf(tb, ", line %d (Marked %d, level %d, status %d/%d) */\n\t\t",
3185            e->n->nval, Marked, level, e->status, e->status&ATOM);
```

```
3186 #ifdef VARSTACK
3187                     if (bu == 1)
3188                             checklast(tb, e->n, e->nxt, e->seqno, 0);
3189 #endif
3190                     if (any_undo(e->n))
3191                             undostmnt(e->n, e->seqno);
3192                     fprintf(tb, ";\n\t\t");
3193 #ifdef GODEF
3194             Countm = 0;
3195             /* DO combine the conflict sets in the backward move */
3196             if (Marked)     /* guard of an atomic sequence -with globals- */
3197                     pop_ast(tb, 1);
3198             else
3199             {       if ((e->status&ATOM) == 0 && (e->status&L_ATOM) == 0)
3200                     {       if (any_cs(e->n))       /* globals touched */
3201                                     push_cs(tb, e->n, 1);
3202                             else                    /* locals only */
3203                             {       Countm++;
3204                                     fprintf(tb, "push_act(II, R_LOCK, BLOCK, ");
3205                                     fprintf(tb, "t->forw, MAXCONFL);\n\t\t");
3206                             }
3207                     } else if (aMarked)     /* guard of local at.seq. */
3208                     {       Countm++;
3209                             fprintf(tb, "push_act(II, R_LOCK, BLOCK, ");
3210                             fprintf(tb, "t->forw, MAXCONFL);\n\t\t");
3211                     }
3212             }
3213             Maxcs = max(Countm, Maxcs);
3214 #endif
3215                     fprintf(tb, "goto R999;\n");
3216                     fprintf(tt, "settr(%d,%d,%d,%d,\"",
3217                             e->status, a, uniq-1, uniq-1, e->n->ntyp);
3218             } else
3219                     fprintf(tt, "settr(%d,%d,%d,0,\"",
3220                             e->status, a, uniq-1, e->n->ntyp);
3221             comment(tt, e->n, e->seqno);
3222             if (Marked)     /* globals are touched later in an atomic s. */
3223                     Globalname=1;
3224             fprintf(tt, "\", %d, %d);\t/* line %d (%d,%d) */\n",
3225                     1-Globalname, e->n->ntyp, e->n->nval, Marked, level);
3226             Marked = 0; aMarked = 0;
3227     }
3228 }
3229
3230 void
3231 patch_atomic(s)
3232     Sequence *s;
3233 {  /* catch goto's that break the chain */
3234     Element *f, *g;
3235     SeqList *h;
3236     for (f = s->frst; ; f = f->nxt)
3237     {       if (f->n && f->n->ntyp == GOTO)
3238             {       g = get_lab(f->n->nsym);
3239                     if ((f->status & ATOM)
```

```
3240                    && !(g->status & ATOM))
3241                    {       f->status &= ~ATOM;
3242                            f->status |= L_ATOM;
3243                    }
3244            } else
3245            for (h = f->sub; h; h = h->nxt)
3246                    patch_atomic(h->this);
3247            if (f == s->last)
3248                    break;
3249    }
3250 }
3251
3252 #ifdef GODEF
3253
3254 int Mustwrite = 0;
3255
3256 any_cs(now)
3257    Node *now;
3258 {
3259    Node *v;
3260
3261    if (!now) { return; }
3262    switch (now->ntyp) {
3263
3264    case CONST:  case  'q': case '.':
3265    case BREAK:  case GOTO: case '@':
3266    case ATOMIC: case   IF: case DO:
3267                    return 0;
3268
3269    case 'p':       /* XXXXX forbid rem ref of locals - handle _p */
3270            return 0;
3271
3272    case 'c': case '!': case LEN:
3273    case UMIN: case ASSERT:
3274    case '~':       return any_cs(now->lft);
3275
3276    case '/': case '*': case '-': case '+':
3277    case '%': case '<': case '>': case '&':
3278    case '|': case LE: case GE: case NE:
3279    case EQ: case OR: case AND: case LSHIFT: case RSHIFT: case ASGN:
3280                    return any_cs(now->lft)|any_cs(now->rgt);
3281
3282    case RUN:
3283    case PRINT:     for (v = now->lft; v; v = v->rgt)
3284                            if (any_cs(v->lft))
3285                                    return 1;
3286                    return 0;
3287
3288    case TIMEOUT:
3289    case 'r':
3290    case 's':
3291    case 'R':       return 1;
3292
3293    case NAME:      if (!now->nsym->context
```

```
3294                    ||  now->nsym->type == CHAN)    /* global or chan */
3295                            return 1;
3296                    if (now->nsym->nel != 1)
3297                            return any_cs(now->lft);
3298                    return 0;
3299    }
3300    fprintf(stderr, "cannot happen %d\n", now->ntyp);
3301    return 0;
3302 }
3303
3304 void
3305 coll_cs(now)
3306    Node *now;
3307 {
3308    Node *v;
3309
3310    if (!now) { return; }
3311    switch (now->ntyp) {
3312    case 'c': case '!':
3313    case UMIN: case ASSERT:
3314    case '~':       coll_cs(now->lft);
3315                    break;
3316
3317    case '/': case '*': case '-': case '+':
3318    case '%': case '<': case '>': case '&':
3319    case '|': case LE: case GE: case NE:
3320    case EQ: case OR: case AND: case LSHIFT: case RSHIFT:
3321                    coll_cs(now->lft);
3322                    coll_cs(now->rgt);
3323                    break;
3324
3325    case PRINT:
3326    case RUN:       for (v = now->lft; v; v = v->rgt)
3327                            coll_cs(v->lft);
3328                    break;
3329
3330    case ASGN:      coll_base("W_LOCK", Direct, now->ntyp, now->lft);
3331                    coll_indx(now->lft);
3332                    coll_cs(now->rgt);
3333                    break;
3334
3335    case 'r':       coll_base("R_LOCK", Direct, now->ntyp, now->lft);
3336                    coll_base("Rcv_LOCK", Indirect, now->ntyp, now->lft);
3337                    coll_indx(now->lft);
3338
3339                    Mustwrite = 1;
3340                    for (v = now->rgt; v; v = v->rgt)
3341                            coll_cs(v->lft);
3342                    Mustwrite = 0;
3343                    break;
3344
3345    case 's':       coll_base("R_LOCK", Direct, now->ntyp, now->lft);
3346
3347                    coll_base("Snd_LOCK", Indirect, now->ntyp, now->lft);
```

```
3348                    coll_indx(now->lft);
3349
3350                    for (v = now->rgt; v; v = v->rgt)
3351                            coll_cs(v->lft);
3352                    break;
3353
3354    case 'R':       coll_base("R_LOCK", Direct, now->ntyp, now->lft);
3355
3356                    coll_base("R_LOCK", Indirect, now->ntyp, now->lft);
3357                    coll_indx(now->lft);
3358
3359                    for (v = now->rgt; v; v = v->rgt)
3360                            coll_cs(v->lft);
3361                    break;
3362    case TIMEOUT:   coll_base("R_LOCK", Direct, now->ntyp, 0);
3363                    break;
3364    case LEN:       coll_base("R_LOCK", Direct, now->ntyp, now->lft);
3365
3366                    coll_base("R_LOCK", Indirect, now->ntyp, now->lft);
3367                    coll_indx(now->lft);
3368                    break;
3369
3370    case NAME:      coll_base((Mustwrite)?"W_LOCK":"R_LOCK", Direct, now->ntyp, now);
3371                    coll_indx(now);
3372                    break;
3373
3374    case CONST:
3375    case   'p':
3376    case   'q':
3377    default   :     break;
3378    }
3379 }
3380
3381 void
3382 push_loss(fd, now, How)
3383    FILE *fd;
3384    Node *now;
3385 {
3386    Node *v;
3387    /* special case: update the conflict sets when
3388     * a message loss on option -m occurs
3389     * it counts as a read on the channel id
3390     */
3391    putbase(fd, "R_LOCK", How, now->lft);
3392    fprintf(fd, "push_act(II, R_LOCK, %s, t->forw, ",
3393            (How == 0)?"REL":"BLOCK");
3394    putname(fd, "1+MAXCONFL+", now->lft, 0, ");\n\t\t");
3395    putindex(fd, now->lft, How);
3396    for (v = now->rgt; v; v = v->rgt)
3397            push_cs(fd, v->lft, How);
3398 }
3399
3400 void
3401 push_cs(fd, now, How)      /* How = 0, before; How = 1, after */
```

```
3402    FILE *fd;
3403    Node *now;
3404 {
3405    Node *v;
3406
3407    if (!now) { return; }
3408    switch (now->ntyp) {
3409    case 'c': case '!':
3410    case UMIN: case ASSERT:
3411    case '~':       push_cs(fd,now->lft, How);
3412                    break;
3413
3414    case '/': case '*': case '-': case '+':
3415    case '%': case '<': case '>': case '&':
3416    case '|': case LE: case GE: case NE:
3417    case EQ: case OR: case AND: case LSHIFT: case RSHIFT:
3418                    push_cs(fd, now->lft, How);
3419                    push_cs(fd, now->rgt, How);
3420                    break;
3421
3422    case PRINT:
3423    case RUN:       for (v = now->lft; v; v = v->rgt)
3424                            push_cs(fd, v->lft, How);
3425                    break;
3426
3427    case ASGN:      putbase(fd, "W_LOCK", How, now->lft);
3428                    if (now->lft->nsym->nel != 1)
3429                            push_cs(fd, now->lft->lft, How);
3430                    push_cs(fd, now->rgt, How);
3431                    break;
3432
3433    case 'r':       fprintf(fd, "{ int L_typ = Rcv_LOCK;\n");
3434                    fprintf(fd, "#if SYNC\n");
3435                    fprintf(fd, "\t\tint od=depth;\n");
3436                    fprintf(fd, "#if ASYNC\n");
3437                    putname(fd, "\t\tif (q_zero(", now->lft, 0, "))\n");
3438                    fprintf(fd, "#endif\n");
3439                    fprintf(fd, "\t\t{      depth--; L_typ = Snd_LOCK; }\n");
3440                    /*
3441                     * depth is decremented here to make sure these
3442                     * blocks are committed to and unpushed by the
3443                     * send half of the rendezvous
3444                     */
3445                    fprintf(fd, "#endif\n\t\t");
3446                    putbase(fd, "R_LOCK", How, now->lft);
3447                    Countm++;
3448                    fprintf(fd, "push_act(II, L_typ, %s, t->forw, ",
3449                            (How == 0)?"REL":"BLOCK");
3450                    putname(fd, "1+MAXCONFL+", now->lft, 0, ");\n\t\t");
3451                    putindex(fd, now->lft, How);
3452                    Mustwrite = 1;
3453                    for (v = now->rgt; v; v = v->rgt)
3454                            push_cs(fd, v->lft, How);
3455                    Mustwrite = 0;
```

```
3456                    fprintf(fd, "\n#if SYNC\n");
3457                    fprintf(fd, "           depth = od;\n");
3458                    fprintf(fd, "#endif\n");
3459                    fprintf(fd, "           }\n\t\t");
3460                    break;
3461
3462    case 's':       if (How == 1)   /* lock rv's only at receive point */
3463                    {       fprintf(fd, "if (SYNC == 0 || !q_zero");
3464                            putname(fd, "(", now->lft, 0, ")) {\n\t\t");
3465                    }
3466                    putbase(fd, "R_LOCK", How, now->lft);
3467                    Countm++;
3468                    fprintf(fd, "push_act(II, Snd_LOCK, %s, t->forw, ",
3469                            (How == 0)?"REL":"BLOCK");
3470                    putname(fd, "1+MAXCONFL+", now->lft, 0, ");\n\t\t");
3471                    putindex(fd, now->lft, How);
3472                    for (v = now->rgt; v; v = v->rgt)
3473                            push_cs(fd, v->lft, How);
3474                    if (How == 1)
3475                            fprintf(fd, "}\n\t\t");
3476                    break;
3477
3478    case 'R':       putbase(fd, "R_LOCK", How, now->lft);
3479                    Countm++;
3480                    fprintf(fd, "push_act(II, R_LOCK, %s, t->forw, ",
3481                            (How == 0)?"REL":"BLOCK");
3482                    putname(fd, "1+MAXCONFL+", now->lft, 0, ");\n\t\t");
3483                    putindex(fd, now->lft, How);
3484                    for (v = now->rgt; v; v = v->rgt)
3485                            push_cs(fd, v->lft, How);
3486                    break;
3487
3488    case LEN:       putbase(fd, "R_LOCK", How, now->lft);
3489                    Countm++;
3490                    fprintf(fd, "push_act(II, R_LOCK, %s, t->forw, ",
3491                            (How == 0)?"REL":"BLOCK");
3492                    putname(fd, "1+MAXCONFL+", now->lft, 0, ");\n\t\t");
3493                    putindex(fd, now->lft, How);
3494                    break;
3495
3496    case NAME:      putbase(fd, (Mustwrite)?"W_LOCK":"R_LOCK", How, now);
3497                    if (now->nsym->nel != 1)
3498                            push_cs(fd, now->lft, How);
3499                    break;
3500    case TIMEOUT:   fprintf(fd, "push_act(II, R_LOCK, %s, t->forw, ",
3501                            (How == 0)?"REL":"BLOCK");
3502                    fprintf(fd, "CS_timeout);\n\t\t");
3503                    break;
3504    case   'p':
3505    case   'q':
3506    case CONST:
3507    default   :     break;
3508    }
3509 }
```

```
3510 #endif
3511
3512 #define cat0(x)    putstmnt(fd,now->lft,m); fprintf(fd, x); \
3513                    putstmnt(fd,now->rgt,m)
3514 #define cat1(x)            fprintf(fd,"("); cat0(x); fprintf(fd,")")
3515 #define cat2(x,y)          fprintf(fd,x); putstmnt(fd,y,m)
3516 #define cat3(x,y,z)        fprintf(fd,x); putstmnt(fd,y,m); fprintf(fd,z)
3517
3518 void
3519 putstmnt(fd, now, m)
3520    FILE *fd;
3521    Node *now;
3522 {
3523    Node *v;
3524    int i, j;
3525
3526    if (!now) { fprintf(fd, "0"); return; }
3527    if (now->ntyp != CONST) lineno = now->nval;
3528    switch (now->ntyp) {
3529    case CONST:     fprintf(fd, "%d", now->nval); break;
3530    case '!':       cat3("!(", now->lft, ")"); break;
3531    case UMIN:      cat3("-(", now->lft, ")"); break;
3532    case '~':       cat3("~(", now->lft, ")"); break;
3533
3534    case '/':       cat1("/");  break;
3535    case '*':       cat1("*");  break;
3536    case '-':       cat1("-");  break;
3537    case '+':       cat1("+");  break;
3538    case '%':       cat1("%%"); break;
3539    case '<':       cat1("<");  break;
3540    case '>':       cat1(">");  break;
3541    case '&':       cat1("&");  break;
3542    case '|':       cat1("|");  break;
3543    case LE:        cat1("<="); break;
3544    case GE:        cat1(">="); break;
3545    case NE:        cat1("!="); break;
3546    case EQ:        cat1("=="); break;
3547    case OR:        cat1("||"); break;
3548    case AND:       cat1("&&"); break;
3549    case LSHIFT:    cat1("<<"); break;
3550    case RSHIFT:    cat1(">>"); break;
3551
3552    case TIMEOUT:   fprintf(fd, "((trpt->tau)&1)"); break;
3553
3554    case RUN:       if (claimproc
3555                    &&  strcmp(now->nsym->name, claimproc) == 0)
3556                            fatal("%s is claim, not runnable",
3557                                                    claimproc);
3558                    if (EVAL_runs)
3559                    {       fprintf(fd, "(now._nr_pr < MAXPROC)");
3560                            break;
3561                    }
3562                    fprintf(fd,"addproc(%d", fproc(now->nsym->name));
3563                    for (v = now->lft; v; v = v->rgt)
```

```
3564                    {       cat2(", ", v->lft);
3565                    }
3566                    fprintf(fd, ")");
3567                    break;
3568    case LEN:       putname(fd, "q_len(", now->lft, m, ")");
3569                    break;
3570
3571    case 's':       fprintf(fd, "\n#if (SYNC>0 && ASYNC==0)\n\t\t");
3572                    putname(fd, "if (q_len(", now->lft, m, "))\n");
3573                    fprintf(fd, "#else\n\t\t");
3574                    putname(fd, "if (q_full(", now->lft, m, "))\n");
3575                    fprintf(fd, "#endif\n");
3576                    if (m_loss)
3577                    {       fprintf(fd, "\t\t{\n\t\tm=3; loss++;\n");
3578                            push_loss(fd, now, 0);
3579                            fprintf(fd, "goto P999;\n\t\t}\n\t\t");
3580                    } else
3581                            fprintf(fd, "\t\t\tcontinue;\n\t\t");
3582                    putname(fd, "qsend(", now->lft, m, "");
3583                    for (v = now->rgt, i = 0; v; v = v->rgt, i++)
3584                    {       cat2(", ", v->lft);
3585                    }
3586                    if (i > Mpars)
3587                            fatal("too many pars in send", "");
3588                    for ( ; i < Mpars; i++)
3589                            fprintf(fd, ", 0");
3590                    fprintf(fd, ");\n");
3591                    fprintf(fd, "#if SYNC\n#if ASYNC==0\n");
3592                    putname(fd, "\t\tboq = ", now->lft, m, ";\n");
3593                    fprintf(fd, "#else\n\t\t");
3594                    putname(fd, "if (q_zero(", now->lft, m, ")) ");
3595                    putname(fd, "boq = ",now->lft,m,";\n");
3596                    fprintf(fd, "#endif\n#endif\n\t\t");
3597                    break;
3598    case 'r':       fprintf(fd, "\n#if SYNC\n#if ASYNC==0\n");
3599                    putname(fd, "\t\tif (boq != ", now->lft,m,")");
3600                    fprintf(fd, " continue;\n#else\n");
3601                    putname(fd, "\t\tif (q_zero(", now->lft,m,"))");
3602                    fprintf(fd, "\n\t\t");
3603                    putname(fd, "{  if (boq != ",  now->lft,m,")");
3604                    fprintf(fd, " continue;\n\t\t} else\n\t\t");
3605                    fprintf(fd, "{  if (boq != -1) continue;\n\t\t");
3606                    fprintf(fd, "}\n#endif\n#endif\n\t\t");
3607                    putname(fd, "if (q_len(", now->lft, m, ") == 0)");
3608                    fprintf(fd, " continue");
3609    /* test */      for (v = now->rgt, i=j=0; v; v = v->rgt, i++)
3610                    {       if (v->lft->ntyp != CONST)
3611                            {       j++; continue;
3612                            }
3613                            fprintf(fd, ";\n\t\t");
3614                            cat3("if (", v->lft, " != ");
3615                            putname(fd, "qrecv(", now->lft, m, ", ");
3616                            fprintf(fd, "0, %d, 0)) continue", i);
3617                    }
```

```
3618                    if (j > 0)
3619                            fprintf(fd, ";\n\t\tsv_save()");
3620    /* set */       for (v = now->rgt, i = 0; v; v = v->rgt, i++)
3621                    {       if (v->lft->ntyp == CONST && v->rgt)
3622                                    continue;
3623                            fprintf(fd, ";\n\t\t");
3624                            if (v->lft->ntyp != CONST)
3625                            {       nocast=1;
3626                                    putstmnt(fd, v->lft, m);
3627                                    nocast=0; fprintf(fd, " = ");
3628                            }
3629                            putname(fd, "qrecv(", now->lft, m, "");
3630                            fprintf(fd, ", 0, %d, ", i);
3631                            fprintf(fd, "%d)", (v->rgt)?0:1);
3632                    }
3633                    fprintf(fd, ";\n#if SYNC\n");
3634                    putname(fd, "\t\tif (q_zero(", now->lft, m, "");
3635                    fprintf(fd, ")) boq = -1;\n#endif\n\t\t");
3636                    break;
3637    case 'R':       putname(fd, "(q_len(", now->lft, m, ") > 0");
3638    /* test */      for (v = now->rgt, i=j=0; v; v = v->rgt, i++)
3639                    {       if (v->lft->ntyp != CONST)
3640                            {       j++; continue;
3641                            }
3642                            fprintf(fd, "\n\t\t&& qrecv(");
3643                            putname(fd, "", now->lft, m, "");
3644                            fprintf(fd, ", 0, %d, 0) == ", i);
3645                            putstmnt(fd, v->lft, m);
3646                    }
3647                    fprintf(fd, ")");
3648                    break;
3649
3650    case 'c':       cat3("if (!(", now->lft, "))\n");
3651                    fprintf(fd, "\t\t\tcontinue");
3652                    break;
3653    case ASGN:      cat3("(trpt+1)->oval = ", now->lft, ";\n\t\t");
3654                    nocast = 1; putstmnt(fd,now->lft,m); nocast = 0;
3655                    fprintf(fd," = ");
3656                    putstmnt(fd,now->rgt,m);
3657                    break;
3658    case PRINT:     fprintf(fd, "printf(%s", now->nsym->name);
3659                    for (v = now->lft; v; v = v->rgt)
3660                    {       cat2(", ", v->lft);
3661                    }
3662                    fprintf(fd, ")");
3663                    break;
3664    case NAME:      if (!nocast && now->nsym
3665                    &&  now->nsym->type < SHORT)
3666                            putname(fd, "((int)", now, m, ")");
3667                    else
3668                            putname(fd, "", now, m, "");
3669                    break;
3670    case   'p':     putremote(fd, now, m);
3671                    break;
```

```
3672    case   'q':     if (terse)
3673                            fprintf(fd, "%s", now->nsym->name);
3674                    else
3675                            fprintf(fd, "%d", remotelab(now));
3676                    break;
3677    case ASSERT:    cat3("assert(", now->lft, ", ");
3678                    terse = nocast = 1;
3679                    cat3("\"", now->lft, "\\n\", II, tt, t)");
3680                    terse = nocast = 0;
3681                    break;
3682    case   '.':
3683    case BREAK:
3684    case  GOTO:     putskip(m);
3685                    break;
3686    case   '@':     if (EVAL_runs)
3687                    {       fprintf(fd, "if (i+1 != now._nr_pr) continue");
3688                            break;
3689                    }
3690                    fprintf(fd, "if (!delproc(1, II)) continue");
3691                    fprintf(th, "#define endstate%d %d\n", Pid, m);
3692                    break;
3693    default   :     printf("spin: bad node type %d (.m)\n", now->ntyp);
3694                    fflush(tm); fflush(tb);
3695                    fflush(tc); fflush(th); fflush(tt);
3696                    exit(1);
3697    }
3698 }
3699
3700 putfair(fd, now, m, u)
3701    FILE *fd;
3702    Node *now;
3703 {
3704    Node *v;
3705    int i, j;
3706
3707
3708    if (!now) { fprintf(fd, "0"); return; }
3709    switch (now->ntyp) {
3710    default:        fprintf(fd, "\tcase  %d:\n\t\t", u);
3711                    EVAL_runs = 1;  /* don't execute a RUN or @ */
3712                    putstmnt(fd, now, m);
3713                    EVAL_runs = 0;
3714                    fprintf(fd, ";\n\t\tbreak;\n");
3715                    break;
3716    case ASSERT:
3717    case   '.':
3718    case BREAK:
3719    case  GOTO:
3720    case ASGN:
3721    case PRINT:     fprintf(fd, "\tcase  %d: break;\n", u);
3722                    break;
3723
3724    case 's':       if (m_loss)
3725                    {       fprintf(fd, "\tcase  %d: break;\n", u);
```

```
3726                            break;
3727                    }
3728                    fprintf(fd, "\tcase  %d:", u);
3729                    fprintf(fd, "\n#if (SYNC>0 && ASYNC==0)\n\t\t");
3730                    putname(fd, "if (q_len(", now->lft, m, ")) ");
3731                    fprintf(fd, "\n#else\n\t\t");
3732                    putname(fd, "if (q_full(", now->lft, m, ")) ");
3733                    fprintf(fd, "\n#endif\n\t\t\t");
3734                    fprintf(fd, "continue;\n");
3735                    fprintf(fd, "\t\tbreak;\n");
3736                    break;
3737    case 'r':       fprintf(fd, "\tcase  %d:", u);
3738                    fprintf(fd, "\n#if SYNC\n#if ASYNC==0\n");
3739                    putname(fd, "\t\tif (boq != ", now->lft,m,")");
3740                    fprintf(fd, " continue;\n#else\n");
3741                    putname(fd, "\t\tif (q_zero(", now->lft,m,"))");
3742                    fprintf(fd, "\n\t\t");
3743                    putname(fd, "{  if (boq != ",  now->lft,m,")");
3744                    fprintf(fd, " continue;\n\t\t} else\n\t\t");
3745                    fprintf(fd, "{  if (boq != -1) continue;\n\t\t");
3746                    fprintf(fd, "}\n#endif\n#endif\n\t\t");
3747                    putname(fd, "if (q_len(", now->lft, m, ") == 0)");
3748                    fprintf(fd, " continue");
3749    /* test */      for (v = now->rgt, i=j=0; v; v = v->rgt, i++)
3750                    {       if (v->lft->ntyp != CONST)
3751                            {       j++; continue;
3752                            }
3753                            fprintf(fd, ";\n\t\t");
3754                            cat3("if (", v->lft, " != ");
3755                            putname(fd, "qrecv(", now->lft, m, ", ");
3756                            fprintf(fd, "0, %d, 0)) continue", i);
3757                    }
3758                    fprintf(fd, ";\n\t\tbreak;\n");
3759                    break;
3760    }
3761 }
3762
3763 void
3764 putbase(fd, what, when, n)
3765    FILE *fd;
3766    char *what;
3767    Node *n;
3768 {
3769    if (n->nsym->context)
3770            return; /* not a global */
3771
3772    Countm++;
3773    fprintf(fd, "push_act(II, %s, %s, t->forw, CS_",
3774            what, (when == 0)?"REL":"BLOCK");
3775    fprintf(fd, "%s", n->nsym->name);
3776    if (n->nsym->nel > 1)
3777    {       fprintf(fd, "+");
3778            putstmnt(fd, n->lft, 0);
3779    }
```

```
3780    fprintf(fd, ");\n\t\t");
3781 }
3782
3783 void
3784 putindex(fd, n, How)
3785    FILE *fd;
3786    Node *n;
3787 {
3788    if (Mustwrite != 0)
3789            fprintf(stderr, "cannot happen putindex\n");
3790    if (n->nsym->nel != 1)
3791            push_cs(fd, n->lft, How);
3792
3793 }
3794
3795 void
3796 coll_indx(n)
3797    Node *n;
3798 {
3799    if (n->nsym->nel != 1)
3800            coll_cs(n->lft);
3801 }
3802
3803 void
3804 coll_base(what, when, cause, n)
3805    char *what;
3806    Node *n;
3807 {
3808    if (n && n->nsym->context)
3809            return; /* not a global */
3810    push_ast(what, when, cause, n);
3811 }
3812
3813 void
3814 putname(fd, pre, n, m, suff)       /* varref */
3815    FILE *fd;
3816    Node *n;
3817    char *pre, *suff;
3818 {
3819    Symbol *s = n->nsym;
3820    if (!s)
3821            fatal("no name - putname", "");
3822    if (!s->type)
3823            yyerror("undeclared name ‘%s’", s->name);
3824
3825    if (!s->context || s->type == CHAN)
3826            Globalname = 1;
3827    fprintf(fd, pre);
3828    if (s->context || !strcmp(s->name, "_p"))
3829    {       if (!terse) fprintf(fd, "((P%d *)this)->", Pid);
3830            fprintf(fd, "%s", s->name);
3831    } else
3832    {       if (!terse) fprintf(fd, "now.");
3833            fprintf(fd, "%s", s->name);
```

```
3834    }
3835    if (s->nel != 1)
3836    {       cat3("[ Index(", n->lft, ", "); /* BOUNDCHECK */
3837            fprintf(fd, "%d) ]", s->nel);   /* BOUNDCHECK */
3838    }
3839    fprintf(fd, suff);
3840 }
3841
3842 void
3843 putremote(fd, n, m)               /* remote reference */
3844    FILE *fd;
3845    Node *n;
3846 {
3847    int promoted = 0;
3848
3849    if (terse)
3850    {       fprintf(fd, "%s[", n->lft->nsym->name);
3851            putstmnt(fd, n->lft->lft, m);
3852            if (strcmp(n->nsym->name, "_p") == 0)
3853                    fprintf(fd, "]:");
3854            else
3855                    fprintf(fd, "].%s", n->nsym->name);
3856    } else
3857    {       if (n->nsym->type < SHORT && !nocast)
3858            {       promoted = 1;
3859                    fprintf(fd, "((int)");
3860            }
3861            fprintf(fd, "((P%d *)Pptr(loops+",
3862                    fproc(n->lft->nsym->name));
3863            if (claimproc) fprintf(fd, "1+");
3864            putstmnt(fd, n->lft->lft, m);
3865            fprintf(fd, "))->%s", n->nsym->name);
3866 #if 0
3867            if (strcmp(n->nsym->name, "_p") == 0)
3868                    XXXXX READING _p XXXXX
3869 #endif
3870    }
3871    if (n->rgt)
3872    {       fprintf(fd, "[");       /* cannot do BOUNDCHECK */
3873            putstmnt(fd, n->rgt, m);
3874            fprintf(fd, "]");
3875    }
3876    if (promoted) fprintf(fd, ")");
3877 }
3878
3879 getweight(n)
3880    Node *n;
3881 {
3882    switch (n->ntyp) {
3883    case 'r':     return 4;
3884    case 's':     return 2;
3885    case TIMEOUT: return 1; /* lowest priority */
3886    case 'c':     if (has_typ(n->lft, TIMEOUT)) return 1;
3887    }
```

```
3888    return 3;
3889 }
3890
3891 has_typ(n, m)
3892    Node *n;
3893    short m;
3894 {
3895    if (!n) return 0;
3896    if (n->ntyp == m) return 1;
3897    return (has_typ(n->lft, m) || has_typ(n->rgt, m));
3898 }
3899
3900 /***** spin: pangen3.c *****/
3901
3902 #include <stdio.h>
3903 #include <ctype.h>
3904 #include "spin.h"
3905 #include "y.tab.h"
3906
3907 extern FILE        *th;
3908 #ifdef GODEF
3909 extern int Globalname;
3910 extern int Countm;
3911 #endif
3912
3913 typedef struct SRC {
3914    short ln, st;
3915    struct SRC *nxt;
3916 } SRC;
3917
3918 SRC        *frst = (SRC *) 0;
3919 SRC        *skip = (SRC *) 0;
3920 int        col;
3921
3922 void
3923 putskip(m) /* states that need not be reached */
3924 {  SRC *tmp;
3925
3926    for (tmp = skip; tmp; tmp = tmp->nxt)
3927            if (tmp->st == m)
3928                    return;
3929    tmp = (SRC *) emalloc(sizeof(SRC));
3930    tmp->st = (short) m;
3931    tmp->nxt = skip;
3932    skip = tmp;
3933 }
3934
3935 void
3936 putsrc(n, m)       /* match states to source lines */
3937 {  SRC *tmp;
3938
3939    for (tmp = frst; tmp; tmp = tmp->nxt)
3940            if (tmp->st == m)
3941            {       if (tmp->ln != n)
```

```
3942                            printf("putsrc mismatch %d - %d\n");
3943                    return;
3944            }
3945    tmp = (SRC *) emalloc(sizeof(SRC));
3946    tmp->ln = (short) n;
3947    tmp->st = (short) m;
3948    tmp->nxt = frst;
3949    frst = tmp;
3950 }
3951
3952 void
3953 dumpskip(n, m)
3954 {  SRC *tmp, *lst;
3955    int j;
3956
3957    fprintf(th, "uchar reached%d [] = {\n\t", m);
3958    for (j = 0, col = 0; j <= n; j++)
3959    {       lst = (SRC *) 0;
3960            for (tmp = skip; tmp; lst = tmp, tmp = tmp->nxt)
3961                    if (tmp->st == j)
3962                    {       putnr(1);
3963                            if (lst)
3964                                    lst->nxt = tmp->nxt;
3965                            else
3966                                    skip = tmp->nxt;
3967                            break;
3968                    }
3969            if (!tmp)
3970                    putnr(0);
3971    }
3972    fprintf(th, "};\n");
3973    skip = (SRC *) 0;
3974 }
3975
3976 void
3977 dumpsrc(n, m)
3978 {  SRC *tmp, *lst;
3979    int j;
3980
3981    fprintf(th, "short src_ln%d [] = {\n\t", m);
3982    for (j = 0, col = 0; j <= n; j++)
3983    {       lst = (SRC *) 0;
3984            for (tmp = frst; tmp; lst = tmp, tmp = tmp->nxt)
3985                    if (tmp->st == j)
3986                    {       putnr(tmp->ln);
3987                            if (lst)
3988                                    lst->nxt = tmp->nxt;
3989                            else
3990                                    frst = tmp->nxt;
3991                            break;
3992                    }
3993            if (!tmp)
3994                    putnr(0);
3995    }
```

```
3996    fprintf(th, "};\n");
3997    frst = (SRC *) 0;
3998    dumpskip(n, m);
3999 }
4000
4001 void
4002 putnr(n)
4003 {
4004    if (col++ == 8)
4005    {       fprintf(th, "\n\t");
4006            col = 1;
4007    }
4008    fprintf(th, "%3d, ", n);
4009 }
4010
4011 #define Cat0(x)    comwork(fd,now->lft,m); fprintf(fd, x); \
4012                    comwork(fd,now->rgt,m)
4013 #define Cat1(x)            fprintf(fd,"("); Cat0(x); fprintf(fd,")")
4014 #define Cat2(x,y)          fprintf(fd,x); comwork(fd,y,m)
4015 #define Cat3(x,y,z)        fprintf(fd,x); comwork(fd,y,m); fprintf(fd,z)
4016
4017 void
4018 symbolic(fd, v)
4019    FILE *fd;
4020 {
4021    Node *n; extern Node *Mtype;
4022    int cnt = 1;
4023
4024    for (n = Mtype; n; n = n->rgt, cnt++)
4025            if (cnt == v)
4026            {       fprintf(fd, "%s", n->lft->nsym->name);
4027                    break;
4028            }
4029    if (!n)
4030            fprintf(fd, "%d", v);
4031 }
4032
4033 void
4034 comwork(fd, now, m)
4035    FILE *fd;
4036    Node *now;
4037 {
4038    Node *v;
4039    int i, j; extern int Mpars;
4040
4041    if (!now) { fprintf(fd, "0"); return; }
4042    switch (now->ntyp) {
4043    case CONST:     fprintf(fd, "%d", now->nval); break;
4044    case '!':       Cat3("!(", now->lft, ")"); break;
4045    case UMIN:      Cat3("-(", now->lft, ")"); break;
4046    case '~':       Cat3("~(", now->lft, ")"); break;
4047
4048    case '/':       Cat1("/");  break;
4049    case '*':       Cat1("*");  break;
```

```
4050    case '-':       Cat1("-");  break;
4051    case '+':       Cat1("+");  break;
4052    case '%':       Cat1("%%"); break;
4053    case '<':       Cat1("<");  break;
4054    case '>':       Cat1(">");  break;
4055    case '&':       Cat1("&");  break;
4056    case '|':       Cat1("|");  break;
4057    case LE:        Cat1("<="); break;
4058    case GE:        Cat1(">="); break;
4059    case NE:        Cat1("!="); break;
4060    case EQ:        Cat1("=="); break;
4061    case OR:        Cat1("||"); break;
4062    case AND:       Cat1("&&"); break;
4063    case LSHIFT:    Cat1("<<"); break;
4064    case RSHIFT:    Cat1(">>"); break;
4065
4066    case RUN:       fprintf(fd, "run %s(", now->nsym->name);
4067                    for (v = now->lft; v; v = v->rgt)
4068                            if (v == now->lft)
4069                            {       Cat2("", v->lft);
4070                            } else
4071                            {       Cat2(",", v->lft);
4072                            }
4073                    fprintf(fd, ")");
4074                    break;
4075
4076    case LEN:       putname(fd, "len(", now->lft, m, ")");
4077                    break;
4078
4079    case 's':       putname(fd, "", now->lft, m, "!");
4080                    for (v = now->rgt, i = 0; v; v = v->rgt, i++)
4081                    {       if (v != now->rgt) fprintf(fd,",");
4082                            if (v->lft->ntyp == CONST)
4083                                    symbolic(fd,v->lft->nval);
4084                            else
4085                                    comwork(fd,v->lft,m);
4086                    }
4087                    for ( ; i < Mpars; i++)
4088                            fprintf(fd, ", 0");
4089                    break;
4090    case 'r':       putname(fd, "", now->lft, m, "?");
4091                    for (v = now->rgt, i=j=0; v; v = v->rgt, i++)
4092                    {       if (v != now->rgt) fprintf(fd,",");
4093                            if (v->lft->ntyp == CONST)
4094                                    symbolic(fd,v->lft->nval);
4095                            else
4096                                    comwork(fd,v->lft,m);
4097                    }
4098                    break;
4099    case 'R':       putname(fd, "", now->lft, m, "?[");
4100                    for (v = now->rgt, i=j=0; v; v = v->rgt, i++)
4101                    {       if (v != now->rgt) fprintf(fd,",");
4102                            if (v->lft->ntyp == CONST)
4103                                    symbolic(fd,v->lft->nval);
```

```
4104                            else
4105                                    comwork(fd,v->lft,m);
4106                    }
4107                    fprintf(fd, "]");
4108                    break;
4109
4110    case 'c':       Cat3("(", now->lft, ")");
4111                    break;
4112    case ASGN:      comwork(fd,now->lft,m);
4113                    fprintf(fd," = ");
4114                    comwork(fd,now->rgt,m);
4115                    break;
4116    case PRINT:     {       char buf[512];
4117                            strncpy(buf, now->nsym->name, 510);
4118                            for (i = strlen(buf)-1; i >= 0; i--)
4119                                    if (buf[i] == '\"')
4120                                            buf[i] = '\'';
4121                            fprintf(fd, "printf(%s", buf);
4122                    }
4123                    for (v = now->lft; v; v = v->rgt)
4124                    {       Cat2(",", v->lft);
4125                    }
4126                    fprintf(fd, ")");
4127                    break;
4128    case NAME:      putname(fd, "", now, m, "");
4129                    break;
4130    case   'p':     putremote(fd, now, m);
4131                    break;
4132    case   'q':     fprintf(fd, "%s", now->nsym->name);
4133                    break;
4134    case ASSERT:    Cat3("assert(", now->lft, ")");
4135                    break;
4136    case   '.':
4137    case BREAK:
4138    case  GOTO:     fprintf(fd, "goto", m); break;
4139    case   '@':
4140                    fprintf(fd, "@", m); break;
4141
4142    case ATOMIC:
4143                    fprintf(fd, "ATOMIC");
4144                    break;
4145    case IF:
4146                    fprintf(fd, "IF");
4147                    break;
4148    case DO:
4149                    fprintf(fd, "DO");
4150                    break;
4151    case TIMEOUT:
4152 #ifdef GODEF
4153                    Globalname = 1; /* don't consider this local */
4154 #endif
4155                    fprintf(fd, "timeout");
4156                    break;
4157    default:        if (isprint(now->ntyp))
```

```
4158                            fprintf(fd, "'%c'", now->ntyp);
4159                    else
4160                            fprintf(fd, "%d", now->ntyp);
4161                    break;
4162    }
4163 }
4164
4165 void
4166 comment(fd, now, m)
4167    FILE *fd;
4168    Node *now;
4169 {
4170    extern int terse, nocast;
4171 #ifdef GODEF
4172    Globalname = 0;
4173 #endif
4174    terse=nocast=1;
4175    comwork(fd, now, m);
4176    terse=nocast=0;
4177 }
4178
4179 #ifdef GODEF
4180
4181 atom_stack *top_ast = 0;
4182
4183 push_ast(what, when, cause, n)
4184    char *what;
4185    Node *n;
4186 {
4187    atom_stack *tmp, *lst;
4188
4189    for (tmp = top_ast; tmp; tmp = tmp->nxt)
4190            if (strcmp(tmp->what, what) == 0
4191            &&  tmp->when == when
4192            &&  tmp->n && n
4193            &&  tmp->n->nsym == n->nsym)
4194                    return;
4195    tmp = (atom_stack *) emalloc(sizeof(atom_stack));
4196    tmp->what  = (char *) emalloc(strlen(what) + 1);
4197    strcpy(tmp->what, what);
4198    tmp->when  = when;
4199    tmp->cause = cause;
4200    tmp->n     = n;
4201    if (cause == 'r' || cause == 's' || !top_ast)
4202    {       tmp->nxt   = top_ast;
4203            top_ast    = tmp;
4204    } else  /* tail add */
4205    {       for (lst = top_ast; lst->nxt; lst = lst->nxt)
4206                    ;
4207            lst->nxt = tmp;
4208    }
4209 }
4210
4211 static int rHeader;
```

```
4212 static int sHeader;
4213
4214 lastfirst(fd, tmp)
4215    FILE *fd;
4216    atom_stack *tmp;
4217 {
4218    if (!tmp) return;
4219    lastfirst(fd, tmp->nxt);
4220    switch (tmp->cause) {
4221    case 'r':
4222            if (!rHeader) break;
4223            rHeader = 0;
4224            fprintf(fd, "\n#if SYNC\n");
4225            fprintf(fd, "           depth = od;\n");
4226            fprintf(fd, "#endif\n");
4227            fprintf(fd, "           } /*rH*/\n\t\t");
4228            break;
4229    case 's':
4230            if (!sHeader) break;
4231            sHeader = 0;
4232            fprintf(fd, "} /*sH*/\n\t\t");
4233    default :
4234            break;
4235    }
4236 }
4237
4238 pop_ast(fd, how)
4239    FILE *fd;
4240 {
4241    atom_stack *tmp;
4242
4243    rHeader = 0;
4244    sHeader = 0;
4245    for (tmp = top_ast; tmp; tmp = tmp->nxt)
4246    {       Countm++;
4247            switch (tmp->cause) {
4248            case 'r':
4249                    if (rHeader == 0)
4250                    {       fprintf(fd, "{ int L_typ = Rcv_LOCK;\n");
4251                            fprintf(fd, "#if SYNC\n");
4252                            fprintf(fd, "\t\tint od=depth;\n");
4253                            fprintf(fd, "#if ASYNC\n");
4254                            putname(fd, "\t\tif (q_zero(", tmp->n, 0, "))\n");
4255                            fprintf(fd, "#endif\n");
4256                            fprintf(fd, "\t\t{\tdepth--; L_typ = Snd_LOCK; }\n");
4257                            fprintf(fd, "#endif\n\t\t");
4258                            rHeader++;
4259                    }
4260                    pop_common(fd, tmp, how);
4261                    break;
4262
4263            case 's':
4264                    if (sHeader == 0 && how == 1)
4265                    {       fprintf(fd, "if (SYNC == 0 || !q_zero");
```

```
4266                            putname(fd, "(", tmp->n, 0, ")) {\n\t\t");
4267                            sHeader++;
4268                    }
4269                    pop_common(fd, tmp, how);
4270                    break;
4271            case TIMEOUT:
4272                    fprintf(fd, "push_act(II, R_LOCK, %s, t->forw, ",
4273                            (how == 0)?"REL":"BLOCK");
4274                    fprintf(fd, "CS_timeout); /* + */\n\t\t");
4275                    break;
4276            default:
4277                    pop_common(fd, tmp, how);
4278                    break;
4279            }
4280    }
4281    lastfirst(fd, top_ast);
4282    if (how == 1) clear_ast();
4283 }
4284
4285 pop_common(fd, tmp, how)
4286    FILE *fd;
4287    atom_stack *tmp;
4288 {
4289    if (tmp->when == Direct)
4290    {       fprintf(fd, "push_act(II, %s, ", tmp->what);
4291            fprintf(fd, "%s, t->forw, CS_",
4292                    (how == 0)?"REL":"BLOCK");
4293            fprintf(fd, "%s", tmp->n->nsym->name);
4294            if (tmp->n->nsym->nel > 1)
4295            {       fprintf(fd, "+");
4296                    putstmnt(fd, tmp->n->lft, 0);
4297            }
4298            fprintf(fd, "); /* + */\n\t\t");
4299    } else if (tmp->when == Indirect)
4300    {       if (tmp->cause == 'r')
4301                    fprintf(fd, "push_act(II, L_typ, %s, t->forw, ",
4302                            (how == 0)?"REL":"BLOCK");
4303            else
4304                    fprintf(fd, "push_act(II, %s, %s, t->forw, ",
4305                            tmp->what, (how == 0)?"REL":"BLOCK");
4306            putname(fd, "1+MAXCONFL+", tmp->n, 0, "); /* + */\n\t\t");
4307    } else
4308    {       fprintf(stderr, "cannot happen pop_ast\n");
4309            abort();
4310    }
4311 }
4312
4313 has_ast()
4314 {
4315    return (top_ast != 0);
4316 }
4317
4318 clear_ast()
4319 {
```

```
4320    /* don't call free, avoid wasting time in malloc */
4321    top_ast = (atom_stack *) 0;
4322 }
4323
4324 atom_stack *
4325 save_ast()
4326 {
4327    return top_ast;
4328 }
4329
4330 restor_ast(oCS)
4331    atom_stack *oCS;
4332 {
4333    top_ast = oCS;
4334 }
4335
4336 coll_global(s, how)
4337    Sequence *s;
4338 {
4339    Element *f, *g;
4340    SeqList *h;
4341
4342    if (!s) return;
4343    for (f = s->frst; ; f = f->nxt)
4344    {       coll_cs(f->n);
4345            for (h = f->sub; h; h = h->nxt)
4346                    coll_global(h->this, how);
4347            if (f == s->last)
4348                    break;
4349    }
4350 }
4351 #endif
4352
4353 /***** spin: pangen4.c *****/
4354
4355 #include <stdio.h>
4356 #include "spin.h"
4357 #include "y.tab.h"
4358
4359 extern FILE        *tc, *tb;
4360 extern Queue       *qtab;
4361 extern int nocast;
4362 extern int lineno;
4363 extern char        *R13[], *R14[], *R15[];
4364
4365 void
4366 undostmnt(now, m)
4367    Node *now;
4368 {
4369    Node *v;
4370    int i, j; extern int m_loss;
4371
4372    if (!now)
4373    {       fprintf(tb, "0");
```

```
4374            return;
4375    }
4376    lineno = now->nval;
4377    switch (now->ntyp) {
4378    case CONST:     case '!':       case UMIN:
4379    case '~':       case '/':       case '*':
4380    case '-':       case '+':       case '%':
4381    case '<':       case '>':       case '&':
4382    case '|':       case LE:        case GE:
4383    case NE:        case EQ:        case OR:
4384    case AND:       case LSHIFT:    case RSHIFT:
4385    case TIMEOUT:   case LEN:       case NAME:
4386    case 'R':       putstmnt(tb, now, m);
4387                    break;
4388    case RUN:       fprintf(tb, "delproc(0, now._nr_pr-1)");
4389                    break;
4390    case 's':       if (m_loss)
4391                    {       fprintf(tb, "if (m != 2) /* msg was lost */\n\t\t");
4392                            fprintf(tb, "{\n\t\t");
4393                            push_loss(tb, now, 1);
4394                            fprintf(tb, "goto R999;\n\t\t");
4395                            fprintf(tb, "}\n\t\t");
4396                    }
4397                    fprintf(tb, "m = unsend");
4398                    putname(tb, "(", now->lft, m, ")");
4399                    break;
4400    case 'r':       for (v = now->rgt, j = 0; v; v = v->rgt)
4401                            if (v->lft->ntyp != CONST)
4402                                    j++;
4403                    if (j > 0)      /* variables were set */
4404                    {       fprintf(tb, "sv_restor()");
4405                            break;
4406                    }
4407                    for (v = now->rgt, i = 0; v; v = v->rgt, i++)
4408                    {       fprintf(tb, "unrecv");
4409                            putname(tb, "(", now->lft, m, ", 0, ");
4410                            fprintf(tb, "%d, ", i);
4411                            undostmnt(v->lft, m);
4412                            fprintf(tb, ", %d);\n\t\t", (i==0)?1:0);
4413                    }
4414                    break;
4415    case '@':       fprintf(tb, "p_restor(II)");
4416                    break;
4417    case ASGN:      nocast=1; putstmnt(tb,now->lft,m);
4418                    nocast=0; fprintf(tb, " = trpt->oval");
4419                    check_proc(now->rgt, m);
4420                    break;
4421    case 'c':       check_proc(now->lft, m);
4422                    break;
4423    case '.':
4424    case GOTO:
4425    case BREAK:     break;
4426    case ASSERT:
4427    case PRINT:     check_proc(now, m);
```

```
4428                    break;
4429    default:        printf("spin: bad node type %d (.b)\n",
4430                                                    now->ntyp);
4431                    exit(1);
4432    }
4433 }
4434
4435 any_undo(now)
4436    Node *now;
4437 {  /* is there anything to undo on a return move? */
4438
4439    if (!now) return 1;
4440    switch (now->ntyp) {
4441    case 'c':       return any_proc(now->lft);
4442    case ASSERT:
4443    case PRINT:     return any_proc(now);
4444
4445    case '.':
4446    case GOTO:
4447    case BREAK:     return 0;
4448    default:        return 1;
4449    }
4450 }
4451
4452 any_proc(now)
4453    Node *now;
4454 {  /* check if an expression refers to a process */
4455    if (!now) return 0;
4456    if (now->ntyp == '@' || now->ntyp == RUN)
4457            return 1;
4458    return (any_proc(now->lft) || any_proc(now->rgt));
4459 }
4460
4461 void
4462 check_proc(now, m)
4463    Node *now;
4464 {
4465    if (!now)
4466            return;
4467    if (now->ntyp == '@' || now->ntyp == RUN)
4468    {       fprintf(tb, ";\n\t\t");
4469            undostmnt(now, m);
4470    }
4471    check_proc(now->lft, m);
4472    check_proc(now->rgt, m);
4473 }
4474
4475 void
4476 genunio()
4477 {  char *buf1;
4478    Queue *q; int i;
4479
4480    buf1 = (char *) emalloc(128);
4481    ntimes(tc, 0, 1, R13);
```

```
4482    for (q = qtab; q; q = q->nxt)
4483    {       sprintf(buf1, "((Q%d *)z)->contents[j].fld", q->qid);
4484            fprintf(tc, "   case %d:\n", q->qid);
4485            for (i = 0; i < q->nflds; i++)
4486                    fprintf(tc, "\t\t%s%d = 0;\n", buf1, i);
4487            if (q->nslots==0)
4488            {       /* check if rendezvous succeeded, 1 level down */
4489                    fprintf(tc, "\t\tm = (trpt+1)->o_m;\n");
4490                    fprintf(tc, "\t\tUnBlock;\n");
4491            } else
4492                    fprintf(tc, "\t\tm = trpt->o_m;\n");
4493            fprintf(tc, "\t\tbreak;\n");
4494    }
4495    ntimes(tc, 0, 1, R14);
4496    for (q = qtab; q; q = q->nxt)
4497    {       sprintf(buf1, "((Q%d *)z)->contents", q->qid);
4498            fprintf(tc, "   case %d:\n", q->qid);
4499            if (q->nslots == 0)
4500                    fprintf(tc, "\t\tif (strt) boq = from;\n");
4501            else if (q->nslots > 1) /* shift */
4502            {       fprintf(tc, "\t\tif (strt && slot<%d)\n",
4503                                                    q->nslots-1);
4504                    fprintf(tc, "\t\t{\tfor (j--; j>=slot; j--)\n");
4505                    fprintf(tc, "\t\t\t{");
4506                    for (i = 0; i < q->nflds; i++)
4507                    {       fprintf(tc, "\t%s[j+1].fld%d =\n\t\t\t",
4508                                                    buf1, i);
4509                            fprintf(tc, "\t%s[j].fld%d;\n\t\t\t",
4510                                                    buf1, i);
4511                    }
4512                    fprintf(tc, "}\n\t\t}\n");
4513            }
4514            strcat(buf1, "[slot].fld");
4515            fprintf(tc, "\t\tif (strt) {\n");
4516            for (i = 0; i < q->nflds; i++)
4517                    fprintf(tc, "\t\t\t%s%d = 0;\n", buf1, i);
4518            fprintf(tc, "\t\t}\n");
4519            if (q->nflds == 1)      /* set */
4520                    fprintf(tc, "\t\tif (fld == 0) %s0 = fldvar;\n",
4521                                                    buf1);
4522            else
4523            {       fprintf(tc, "\t\tswitch (fld) {\n");
4524                    for (i = 0; i < q->nflds; i++)
4525                    {       fprintf(tc, "\t\tcase %d:\t%s", i, buf1);
4526                            fprintf(tc, "%d = fldvar; break;\n", i);
4527                    }
4528                    fprintf(tc, "\t\t}\n");
4529            }
4530            fprintf(tc, "\t\tbreak;\n");
4531    }
4532    ntimes(tc, 0, 1, R15);
4533 }
4534
4535 /***** spin: pangen5.c *****/
```

```
4536
4537 #include <stdio.h>
4538 #include <sys/types.h>
4539 #include <sys/stat.h>
4540 #include "spin.h"
4541 #include "y.tab.h"
4542
4543 extern int nproc, nstop, Tval, Rvous, Have_claim;
4544 extern RunList     *run, *X;
4545 extern int verbose, lineno;
4546 extern int depth;
4547
4548 FILE *fd;
4549
4550 void
4551 whichproc(p)
4552 {  RunList *oX;
4553
4554    for (oX = run; oX; oX = oX->nxt)
4555            if (oX->pid == p)
4556            {       printf("(%s) ", oX->n->name);
4557                    break;
4558            }
4559 }
4560
4561 int
4562 newer(f1, f2)
4563    char *f1, *f2;
4564 {
4565    struct stat x, y;
4566
4567    if (stat(f1, (struct stat *)&x) < 0) return 0;
4568    if (stat(f2, (struct stat *)&y) < 0) return 1;
4569    if (x.st_mtime < y.st_mtime) return 0;
4570    return 1;
4571 }
4572
4573 void
4574 match_trail()
4575 {  int i, pno, nst, lv0=0, lv1=0;
4576    extern Symbol *Fname;
4577
4578    if (Fname->name[0] == '\"')
4579    {       i = strlen(Fname->name);
4580            Fname->name[i-1] = '\0';
4581            Fname = lookup(&Fname->name[1]);
4582    }
4583
4584    if (newer(Fname->name, "pan.trail"))
4585    printf("Warning, file %s modified since trail was written\n",
4586                    Fname->name);
4587
4588    if (!(fd = fopen("pan.trail", "r")))
4589    {       printf("spin -t: cannot find ‘pan.trail’\n");
```

```
4590            exit(1);
4591    }
4592    Tval = 1; /* timeouts may be part of the trail */
4593    while (fscanf(fd, "%d:%d:%d:%d\n", &depth, &pno, &nst, &lv0)
4594                                                        == 4)
4595    {       if (lv1 >= 0 && depth > 0 && (verbose&32 || lv1 != lv0))
4596                    talk(X->pc, X->symtab);
4597            lv1=lv0;        /* non-verbose in intermediate steps */
4598            if (depth == -1)
4599            {       if (verbose)
4600                    printf("<<<<<START OF CYCLE>>>>>\n");
4601                    continue;
4602            }
4603            if (depth == -2)
4604            {
4605                    start_claim(pno);
4606                    continue;
4607            }
4608            i = nproc - nstop;
4609            if (nst == 0)
4610            {       if (pno == i-1 && run->pc->n->ntyp == '@')
4611                    {       run = run->nxt;
4612                            nstop++;
4613                            continue;
4614                    } else
4615                    {       printf("step %d: stop error, %d %d %c\n",
4616                                    depth, pno, i, run->pc->n->ntyp);
4617                            exit(1);
4618            }       }
4619            for (X = run; X; X = X->nxt)
4620            {
4621                    if (--i == pno)
4622                            break;
4623            }
4624            if (!X)
4625            {       int k=0;
4626                    printf("step %d: lost trail ", depth); whichproc(pno);
4627                    if (Have_claim)
4628                    {       if (pno == 1)
4629                                    printf("(state %d)\n", nst);
4630                            else
4631                            {       if (pno > 1) k = 1;
4632                                    printf("(proc %d state %d)\n", pno-k, nst);
4633                            }
4634                    } else
4635                            printf("(proc %d state %d)\n", pno-k, nst);
4636                    lost_trail();
4637                    wrapup();
4638                    exit(1);
4639            }
4640            lineno = X->pc->n->nval;
4641            do
4642            {       X->pc = d_eval_sub(X->pc, pno, nst);
4643            } while (X && X->pc && X->pc->seqno != nst);
```

```
4644            if (!X || !X->pc)
4645            {       int k = 0;
4646                    printf("step %d: lost trail ", depth); whichproc(pno);
4647                    if (Have_claim)
4648                    {       if (pno == 1)
4649                                    printf("(state %d)\n", nst);
4650                            else
4651                            {       if (pno > 1) k = 1;
4652                                    printf("(proc %d state %d.)\n", pno-k, nst);
4653                            }
4654                    } else
4655                            printf("(proc %d state %d.)\n", pno, nst);
4656                    lost_trail();
4657                    wrapup();
4658                    exit(1);
4659            }
4660    }
4661    talk(X->pc, X->symtab);
4662    printf("spin: trail ends after %d steps\n", depth);
4663    wrapup();
4664 }
4665
4666 void
4667 lost_trail()
4668 {  int d, p, n, l;
4669
4670    while (fscanf(fd, "%d:%d:%d:%d\n", &d, &p, &n, &l) == 4)
4671    {       printf("step %d: proc  %d ", d, p); whichproc(p);
4672            printf("(state %d) - d %d\n", n, l);
4673    }
4674 }
4675
4676 int Depth=0;
4677
4678 Element *
4679 walk_sub(e, pno, nst)
4680    Element *e;
4681 {
4682    SeqList *z;
4683    Element *f;
4684
4685    if (Depth > 32) /* very likely circular */
4686            return (Element *) 0;
4687    Depth++;
4688    for (z = e->sub; z; z = z->nxt)
4689    {
4690            if (z->this->frst->seqno == nst)
4691            {       Depth--; return z->this->frst; }
4692            if (!z->this->frst->nxt)
4693                    fatal("cannot happen", "walk_sub");
4694            if (z->this->frst->sub)
4695            {       f = walk_sub(z->this->frst, pno, nst);
4696                    if (f) { Depth--; return f; }
4697            }
```

```
4698            f = huntele(z->this->frst, z->this->frst->status);
4699            if (f->seqno == nst)
4700            {       Depth--; return f; }
4701            if (f->seqno == X->pc->seqno)           /* looping */
4702                    continue;                       /* fails */
4703            if (f->sub && (f = walk_sub(f, pno, nst)))
4704            {       Depth--; return f; }
4705            if (f && f->n->ntyp == ATOMIC)
4706            {       f = f->n->seql->this->frst;
4707                    if (f->seqno == nst)
4708                    {       Depth--; return f; }
4709            }
4710    }
4711    Depth--;
4712    return (Element *) 0;
4713 }
4714
4715 Element *
4716 d_eval_sub(s, pno, nst)
4717    Element *s;
4718 {
4719    Element *e=s;
4720
4721    if (e->n->ntyp == GOTO)
4722    {
4723            return get_lab(e->n->nsym);
4724    }
4725    if (e->sub)
4726    {       if (e = walk_sub(e, pno, nst))
4727            {
4728                    return e;
4729            }
4730    } else if (e->n && e->n->ntyp == ATOMIC)
4731    {       e->n->seql->this->last->nxt = e->nxt;
4732            if (e->n->seql->this->frst->seqno == nst)
4733                    return e->n->seql->this->frst;
4734            return d_eval_sub(e->n->seql->this->frst, pno, nst);
4735    } else if (eval(e->n))
4736    {
4737            return e->nxt;
4738    }
4739    if (e && (nst == e->seqno))
4740            return e;
4741    if (s && (nst == s->seqno))
4742            return s;
4743    printf("step %d: lost trail ", depth);
4744    if (Have_claim)
4745    {       int k=0;
4746            if (pno == 1)
4747                    printf("(");
4748            else
4749            {       if (pno > 1) k = 1;
4750                    printf("(proc %d ", pno-k);
4751            }
```

```
4752    } else
4753            printf("(proc %d ", pno);
4754    whichproc(pno);
4755    printf("state .%d) [stuck in %d]\n", nst, (e)?e->seqno:-1);
4756    lost_trail();
4757    wrapup();
4758    exit(1);
4759 }
```