Spin Tutorial Part II
*Model Extraction*
with FeaVer/Modex

gerard@spinroot.com

LARS
NASA/JPL Laboratory for Reliable Software
a simple example
to get started

here is a typical little, inscrutable, C program. does it have bugs, and if so how many?

can we use model checking to find them?

how much work do we have to do to extract a model and see the result?

step 1: define a Modex test harness:

```
$ cat example.prx
%F example.c
%X -x
%L
Import    _all_    _all_
%
```

```
$ cat example.c
#define N 5
#define M 4

int main(void)
{
    int i, x = 12, y = 34, z, w;
    int *p, *q, *h, *j, *k;
    int *****a, *b[N][M], c[M];

    for (i = 0; i < N; i++)
        b[i][3] = &c[i];

    p = &z;
    q = &x;
    x = 56;
    *p = *q;
    h = p;
    p = &y;
    *****a = h;
    j = *****a;
    h = q;
}
```
run modex

run modex

view model

spin2005
original C code:

```c
$ cat example.c
#define N  5
#define M  4

int main(void) {
    int i, x = 12, y = 34, z, w;
    int *p, *q, *h, *j, *k;
    int *****a, *b[N][M], c[M];
    for (i = 0; i < N; i++)
        b[i][3] = &c[i];
    p = &z;
    q = &x;
    x = 56;
    *p = *q;
    h = p;
    p = &y;
    ****a = h;
    j = ****a;
    h = q;
}
```

the extracted model:

```c
$ cat model
c_state "int ***** a" "Local main"
c_state "int * b[N][M]" "Local main"
c_state "int * p" "Local main"
c_state "int * q" "Local main"
c_state "int * h" "Local main"
c_state "int * j" "Local main"
c_state "int * k" "Local main"

active proctype main( )
{
    int c[M];
    int i;
    int x = 12;
    int y = 34;
    int z, w;

c_code { Pmain->i=0; }
    do
    :: c_expr { (Pmain->i<5) }
    c_code { (Pmain->i < 5) }
    c_code [Pmain->i][3]=&(Pmain->c[Pmain->i]);
    c_code { Pmain->i++; }
    :: else ->
    c_code { Pmain->p = &(Pmain->z); }
    c_code { Pmain->q = &(Pmain->x); }
    c_code { Pmain->x = 56; }
    c_code { Pmain->p = &Pmain->y; }
    c_code [Pmain->q && Pmain->p] { (**Pmain->p) = (**Pmain->q); }
    c_code { Pmain->h = Pmain->p; }
    c_code { Pmain->p = &Pmain->y; }
    c_code [Pmain->a&&(*Pmain->a) && (((**Pmain->a))&&(*(((**Pmain->a)))))
    { (**((**Pmain->a)))=Pmain->h; }
    c_code [Pmain->a&&(*Pmain->a) && (((**Pmain->a))&&(*(((**Pmain->a)))))
    { Pmain->j = (((**Pmain->a)))))); }
    c_code { Pmain->h = Pmain->q; }
}
```

some variables could be represented as Promela data object; others are imported with c_state declarations

modx generated correctness assertions
verification of the model with Spin

$ spin -a model
$ cc -o pan pan.c
$ ./pan
pan: (Pmain→i < 4) (at depth 15)
pan: wrote model.trail
(Spin Version 4.2.5 -- 2 April 2005)
Warning: Search not completed
   + Partial Order Reduction

Full statespace search for:
   never claim       - (none specified)
   assertion violations  +
   acceptance cycles     - (not selected)
   invalid end states    +

State-vector 160 byte, depth reached 14, errors: 1
  15 states, stored
  0 states, matched
  15 transitions (= stored+matched)
  0 atomic steps
hash conflicts: 0 (resolved)

2.622  memory usage (Mbyte)
$  ./pan -r

one of the generated preconditions fails
the counter-example

the counter-example is generated by pan, not spin, because we use embedded C code, which Spin cannot simulate

$ ./pan -r

1:    proc 0 (main) line 25 "pan_in" (state 1) [ Pmain->i=0; ]
2:    proc 0 (main) line 27 "pan_in" (state 7) [(( Pmain->i<5 ))]
3:    proc 0 (main) line 29 "pan_in" (state 3) [ Pmain->b[Pmain->i][3]==(Pmain->c[Pmain->i]); ]
4:    proc 0 (main) line 30 "pan_in" (state 4) [ Pmain->i++; ]
5:    proc 0 (main) line 27 "pan_in" (state 7) [(( Pmain->i<5 ))]
6:    proc 0 (main) line 29 "pan_in" (state 3) [ Pmain->b[Pmain->i][3]==(Pmain->c[Pmain->i]); ]
7:    proc 0 (main) line 30 "pan_in" (state 4) [ Pmain->i++; ]
8:    proc 0 (main) line 27 "pan_in" (state 7) [(( Pmain->i<5 ))]
9:    proc 0 (main) line 29 "pan_in" (state 3) [ Pmain->b[Pmain->i][3]==(Pmain->c[Pmain->i]); ]
10:   proc 0 (main) line 30 "pan_in" (state 4) [ Pmain->i++; ]
11:   proc 0 (main) line 27 "pan_in" (state 7) [(( Pmain->i<5 ))]
12:   proc 0 (main) line 29 "pan_in" (state 3) [ Pmain->b[Pmain->i][3]==(Pmain->c[Pmain->i]); ]
13:   proc 0 (main) line 30 "pan_in" (state 4) [ Pmain->i++; ]
14:   proc 0 (main) line 27 "pan_in" (state 7) [(( Pmain->i<5 ))]
15:   proc 0 (main) line 29 "pan_in" (state 3) [ Pmain->b[Pmain->i][3]==(Pmain->c[Pmain->i]); ]

pan: precondition false: (Pmain->i < 4)
spin: trail ends after 15 steps

local vars proc 0 (main):
    int c[0]: 0
    int c[1]: 0
...

$
example2.c

```sh
$ sed 's/c\[M\]/c[N]/' example.c > example2.c  # correct the array size of c[]
$ modex example2.c
$ cpp –E –P _modex_.drv > model
$ spin –a model
$ cc –o pan pan.c
$ ./pan
pan: Pmain->a && (*Pmain->a) && (**(*Pmain->a))) && (**(**(*Pmain->a))) (at depth 24)
pan: wrote model.trail
(Spin Version 4.2.5 -- 2 April 2005)
Warning: Search not completed
  + Partial Order Reduction
Full statespace search for:
  never claim       - (none specified)
  assertion violations +
  acceptance      cycles   - (not selected)
  invalid end states +
State-vector 164 byte, depth reached 23, errors: 1
  24 states, stored
  0 states, matched
  24 transitions (= stored+matched)
  0 atomic steps
```
new counter-example
a nil-pointer dereference error

$ ./pan -r

1: proc 0 (main) line 25 "pan_in" (state 1) [ Pmain->i=0; ]
2: proc 0 (main) line 27 "pan_in" (state 7) [ ( (Pmain->i<5) )]
3: proc 0 (main) line 29 "pan_in" (state 3) [ Pmain->b[Pmain->i][3]=&(Pmain->c[Pmain->i]); ]
4: proc 0 (main) line 30 "pan_in" (state 4) [ Pmain->i++; ]
5: proc 0 (main) line 27 "pan_in" (state 7) [ ( (Pmain->i<5) )]
6: proc 0 (main) line 29 "pan_in" (state 3) [ Pmain->b[Pmain->i][3]=&(Pmain->c[Pmain->i]); ]
7: proc 0 (main) line 30 "pan_in" (state 4) [ Pmain->i++; ]
8: proc 0 (main) line 27 "pan_in" (state 7) [ ( (Pmain->i<5) )]
9: proc 0 (main) line 29 "pan_in" (state 3) [ Pmain->b[Pmain->i][3]=&(Pmain->c[Pmain->i]); ]
10: proc 0 (main) line 30 "pan_in" (state 4) [ Pmain->i++; ]
11: proc 0 (main) line 27 "pan_in" (state 7) [ ( (Pmain->i<5) )]
12: proc 0 (main) line 29 "pan_in" (state 3) [ Pmain->b[Pmain->i][3]=&(Pmain->c[Pmain->i]); ]
13: proc 0 (main) line 30 "pan_in" (state 4) [ Pmain->i++; ]
14: proc 0 (main) line 27 "pan_in" (state 7) [ ( (Pmain->i<5) )]
15: proc 0 (main) line 29 "pan_in" (state 3) [ Pmain->b[Pmain->i][3]=&(Pmain->c[Pmain->i]); ]
16: proc 0 (main) line 30 "pan_in" (state 4) [ Pmain->i++; ]
17: proc 0 (main) line 27 "pan_in" (state 7) [else]
18: proc 0 (main) line 34 "pan_in" (state 10) [ Pmain->p=&(Pmain->z); ]
19: proc 0 (main) line 35 "pan_in" (state 11) [ Pmain->q=&(Pmain->x); ]
20: proc 0 (main) line 36 "pan_in" (state 12) [ Pmain->x=56; ]
21: proc 0 (main) line 37 "pan_in" (state 13) [ (*Pmain->p)=(*Pmain->q); ]
22: proc 0 (main) line 38 "pan_in" (state 14) [ Pmain->h=Pmain->p; ]
23: proc 0 (main) line 39 "pan_in" (state 15) [ Pmain->p=&(Pmain->y); ]

pan: precondition false: Pmain->a && (*Pmain->a) && (((*Pmain->a)) && (*(((Pmain->a)))))
spin: trail ends after 24 steps
an example with concurrency

int main(void)
{
    thread_t thread_id, main_id;

    main_id = thr_self();
    thr_setconcurrency(2);
    thr_create(NULL, 0, thread_sub, (void *)main_id, THR_SUSPENDED, &thread_id);

    while(1) {
        printf("MAIN: continuing subroutine thread\n"); fflush(stdout);
        thr_continue(thread_id);
        printf("MAIN: suspending self\n"); fflush(stdout);
        thr_suspend(main_id);
    }
    return(0);
}

void *thread_sub(void *arg)
{
    thread_t thread_id;
    thread_t main_id = (thread_t) arg;

    thread_id = thr_self();

    while(1) {
        printf("THREAD: continuing main thread\n"); fflush(stdout);
        thr_continue(main_id);
        printf("THREAD: suspending self\n"); fflush(stdout);
        thr_suspend(thread_id);
    }
    return((void *)0);
}
$ modex sw_race.c  # Extract Model
$ sh _modex_.run  # Compile and Run
pan: invalid end state (at depth 17)
pan: wrote model.trail
(Spin Version 4.2.5 -- 2 April 2005)
...
State-vector 16 byte, depth reached 18, errors: 1
...
$ pan -C  # Replay Error Trace
    THREAD: continuing main thread
    THREAD: suspending self
    MAIN: continuing subroutine thread
    THREAD: continuing main thread
    THREAD: suspending self
    MAIN: suspending self
18: main(0):[Suspend_main = 1]
spin: trail ends after 18 steps
#processes 2:
18: proc 0 (main) line 5 (state 7) (invalid end state)
    Printf("MAIN: continuing subroutine thread\n");
18: proc 1 (thread) line 20 (state 7) (invalid end state)
    Printf("THREAD: continuing main thread\n");
global vars:
    bit Suspend_main: 1
    bit Suspend_thread: 1
    bit arg: 0
...
test harness definition for sw_race

$ cat sw_race.prx
%F sw_race.c
%X -n main
%X -n thread_sub
$L
thr_setconcurrency(...) hide
thr_create(...) hide

thr_suspend(main_id) Suspend_main = 1
thr_continue(main_id) Suspend_main = 0

thr_suspend(thread_id) Suspend_thread = 1
thr_continue(thread_id) Suspend_thread = 0

main_id=... hide
thread_id=... hide
%
P
#define thread_t int

bool Suspend_main, Suspend_thread, arg;

active proctype main() provided (!Suspend_main) {
#include "_modex_main.spn"
}

active proctype thread() provided (!Suspend_thread) {
#include "_modex_thread_sub.spn"
}

special keyword that can appear on the rhs in the lookup table (defining the mapping/abstractions for the model extraction)

optionally define an explicit Promela context for the extracted models for each function in a %P segment in test harness
special keywords that can appear on the rhs in a modex lookup table definition

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>comment</td>
<td>Includes the statement as a comment in the model.</td>
</tr>
<tr>
<td>hide</td>
<td>Omits the statement from the test model.</td>
</tr>
<tr>
<td>skip</td>
<td>Like hide, but replaces statement with a null-step.</td>
</tr>
<tr>
<td>keep</td>
<td>Copies the statement without syntax conversions.</td>
</tr>
<tr>
<td>print</td>
<td>Includes the statement text in a print statement.</td>
</tr>
<tr>
<td>warn</td>
<td>Issues a warning if the left-hand side is encountered.</td>
</tr>
<tr>
<td>true</td>
<td>define an expression to always evaluate to true.</td>
</tr>
<tr>
<td>false</td>
<td>define an expression to always evaluate to false.</td>
</tr>
<tr>
<td>C_code</td>
<td>Encapsulates the statement as is inside the model.</td>
</tr>
<tr>
<td>C_expr</td>
<td>Encapsulates and treats as an expression.</td>
</tr>
</tbody>
</table>

examples:

\[\text{printf(...)} \quad \text{keep} \quad \text{exit(...)} \quad \text{warn}\]
# list of the 10 possible test harness commands

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Single- or Multi-Line</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>%F</td>
<td>single</td>
<td>Set target filename for ANSI C-source.</td>
</tr>
<tr>
<td>%X</td>
<td>single</td>
<td>Defines C source procedures to be extracted.</td>
</tr>
<tr>
<td>%G</td>
<td>multi</td>
<td>Sets parameters used by the FeaVer GUI, feaver_gui.</td>
</tr>
<tr>
<td>%L</td>
<td>multi</td>
<td>Defines a filter (i.e., mapping table) to be used in model extractions via one or more %X commands.</td>
</tr>
<tr>
<td>%H</td>
<td>single/multi</td>
<td>Defines optional header information for generated code.</td>
</tr>
<tr>
<td>%D</td>
<td>single/multi</td>
<td>Declares C data types used in the generated code.</td>
</tr>
<tr>
<td>%C</td>
<td>single/multi</td>
<td>Defines C data declarations used in the generated code.</td>
</tr>
<tr>
<td>%P</td>
<td>single/multi</td>
<td>Defines process infrastructure in Spin modeling language.</td>
</tr>
<tr>
<td>%O</td>
<td>single</td>
<td>Defines extra directives and linked external C-files for final compilation of the generated model checker.</td>
</tr>
<tr>
<td>%Q</td>
<td>single</td>
<td>Defines extra directives for preprocessing source files from which models are to be extracted.</td>
</tr>
</tbody>
</table>
### valid parameters to an %X command

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>-x</td>
<td>Extract all procedures from the target C-source file.</td>
</tr>
<tr>
<td>-a pname</td>
<td>Extract procedure pname as an active proctype.</td>
</tr>
<tr>
<td>-i pname</td>
<td>Extract procedure pname as an inline definition.</td>
</tr>
<tr>
<td>-p pname</td>
<td>Extract procedure pname as a passive proctype.</td>
</tr>
<tr>
<td>-e pname</td>
<td>Extract procedure as an extended active proctype (instrumented to support a procedure call mechanism).</td>
</tr>
<tr>
<td>-E pname</td>
<td>Extract procedure as an extended passive proctype.</td>
</tr>
<tr>
<td>-n pname</td>
<td>Extract procedure pname without proctype encapsulation.</td>
</tr>
<tr>
<td>-L map-name</td>
<td>Define a map to be used in subsequent model extractions.</td>
</tr>
</tbody>
</table>
the main steps in the model extraction process

- lexical analysis & parsing
  - produce control flow graph
- complexity control:
  - slicing
  - abstraction
  - restriction etc.
- model generation
  - interprets the cfg
  - applies the controls
  - converts into a Spin automata model
model extraction...

```c
int main(int argc, char *argv[]) {
    char buf[32];
    int n;
    if (!socket_setup((unsigned short) 78 SUB))
        exit(1); /* err msg already given */
    if (argc > 1)
        preload();
    printf("ch_sched: ready...\n");
    flush(stdin);
    for (;;) {
     sanity();
     fflush(stdout);
     n = wait_for_request(buf, sizeof(buf));
     update count = 0 if it wraps */
     if (strcmp(buf, "offer ", strlen("offer ") == 0)
         (char *) = char[strlen("offer ")];
     char * = char[strlen(buf) + 1];
     if (g) {
         g--; g++;
         add socket(fromaddress, n, g);
     }
    else if (strcmp(buf, "withdraw", strlen("withdraw")) == 0)
        del worker(fromaddress);
    else if (strcmp(buf, "jobdone", strlen("jobdone")) == 0)
        job done(fromaddress); /* from prop */
    else if (strcmp(buf, "done ", strlen("done ") == 0)
        task done(fromaddress, char[strlen("done ")]);
    else if (strcmp(buf, "trail ", strlen("trail ") == 0)
        do_wrap(fromaddress, char[strlen("trail ")]);
    else if (strcmp(buf, "status", strlen("status")) == 0)
        status);
    else if (strcmp(buf, "restrict", strlen("restrict")) == 0)
        restricted = 1;
    else if (strcmp(buf, "release", strlen("release")) == 0)
        restricted = 0;
    else if (strcmp(buf, "exit", strlen("exit")) == 0)
        break;
    else /* default */
        add task(buf);
    beginthread(assignments, G, NULL);
    closesocket((SOCKET) (n-1));
    printf("ch_sched: exits...\n");
    exit(0);
}
```
extensions in Spin v4 that make this possible

five new Promela primitives:

- **c_decl**: to define new C data *types* that can be used later in c_state or c_track primitives
- **c_state**: to declare new C variables *inside* state vector scoped as: Global, Local, or Hidden *(meant for used by modex only, not humans)*
- **c_track**: to report C data objects, declared outside the model (in C code), as holding *state information*
- **c_expr**: blocking: execute a C expression and use the boolean return value in the model
- **c_code**: non-blocking: execute a fragment of C code atomically and deterministically (like a d_step)
so does it really work on larger applications?

Protium system (Bell Labs): ~20 process threads

(server-side) (client-side)
proctype frsssthread() provided (go == _pid) {
    eventchan10;
    frsschan_t70:
    c_code [frssinfo] { cpst(*frssinfo, &(Pfrssthread->ltmpstate)); };
    if
      :: c_expr { (Pfrssthread->ltmpstate, stdid=SVR_INIT_ST) };
      c_code { now.lstate, msg_type=802; };
      c_code { now.lstate, stdid=(SVR_INIT_ST+1); };
      c_code { cpst(now.lstate, tossinfo); };
    c_code [tossinfo] { tossinfo->msg_code=202; };
      assert(!tossready); tossready=1;
      :: else:
        c_code { cpst(Pfrssthread->ltmpstate, &(now.lstate)); };
        c_code { now.curnstate=NEW_ST; };
    fi;
    frsschan_pl0;
    L_6:
    do
        :: frsschan_t70:
        c_code [frssinfo] { cpst(*frssinfo, &(Pfrssthread->ltmpstate)); };
        frsschan_pl0;
    do
      :: c_expr { 807 == Pfrssthread->ltmpstate, msg_type 3;
      c_code { cpst(Pfrssthread->ltmpstate, &now.lstate)); };
      c_code { now.curnstate=NEW_ST; };
    break;
goto C_29
}

#processes 10:
528: proc 0 (init:) line 449 (state 53)
    netfd_out?801.101_.
    netfd_out?802.202_.
    netfd_in?805.0.0
528: proc 1 (frssthread) line 113 (state 56) (invalid endstate)
    frsschan_t70
528: proc 2 (tossthread) line 168 (state 10) (invalid endstate)
    tosschan_t70
528: proc 3 (iowthread) line 200 (state 12) (invalid endstate)
    iochan_t10msg
528: proc 4 (mousethread) line 295 (state 81) (invalid endstate)
    assert(!(tossready))
FeaVer Test Harness
circular wait problem
detected in ~2 minutes
example scenario of 931 steps
the PathStar verification (1998-2000)

C code
Spin model
"test" drivers

environment model

model extraction
bug reporting

filter

Spin model

test automata

feature requirements
feature verification

PathStar

Call Processing Software Verification

extract model of source code

extract model of property

check

document error-traces

Standard Web Browser Interface

web-based interface

full automation

(Holzmann&Smith BLTJ2000)
FeaVer 1.0 User Guide
Gerard J. Holzmann and Margaret H. Smith

http://cm.bell-labs.com/cm/cs/what/modex
http://cm.bell-labs.com/cm/cs/what/feaver

---

Preface

FeaVer is a tool that can be used for the verification of distributed systems or code written in ANSI-standard C. This guide is meant to be read as a first introduction to the tool and to the methodology on which it is based.

We begin with a brief explanation of the need for a tool that can trap concurrency-related errors, followed by an overview of FeaVer's basic mode of operation. The main theme introduced here, and that will return throughout this document, is the definition of a verification test harness. The working of FeaVer is always determined by the test harness definition. Learning to use FeaVer, therefore, means learning to design and debug test harness descriptions.

FeaVer can be used either with a basic command-line interface or with the help of a more refined graphical user interface. In some cases the command-line interface is preferable, giving slightly more insight into the details of FeaVer's operation. But undeniably, especially when working with a fully developed test harness description, the graphical user interface can be more convenient. For first-time use, it is recommended to begin by developing a good basic understanding of the main processing steps that FeaVer performs when it runs. In the first few chapters of this guide, therefore, we will focus primarily on the command-line interface and a textual description of the main elements of a test harness. In a later chapter the graphical user interface is covered. A comprehensive review section at the end of this guide, is meant to be a point of reference for the more experienced user of FeaVer when constructing test harness definitions.

At the time of writing the FeaVer tool is being developed as a research prototype, and it continues to evolve. If you detect any information that is confusing or just plain wrong, please let us know and we will correct it.

---

64 pgs
some papers on modex/feaver
(pdf’s are on http://spinroot.com/gerard/pubs.html)

synopsis