

SVA: using CSP and FDR to analyse shared variable programs

SVA: using CSP and FDR to analyse shared variable programs

Bill Roscoe
Oxford University Computing Laboratory
with thanks to David Hopkins

SVA: using CSP and FDR to analyse shared variable programs

SVA: Shared Variable Analyser

See Chapters 18 and 19 of [Understanding Concurrent Systems: new book on CSP](#) published by Springer.

This talk summarises parts of those chapters.

Tool and lots of examples down-loadable from

www.comlab.ox.ac.uk/ucs.

SVA: Shared Variable Analyser

- SVA is a front end for CSP that handles shared variable programs.
- Originally written in 2000 by AWR, inspired by tutorial exercises on mutual exclusion:

Which of the following mutual exclusion algorithms achieves its objective? ($i = 1, 2$.) (Hyman's and Petersen's algorithms.)

```
While true do
{b[i] := true;
While t!=i do
{While b[3-i] do Skip;
t := i};
{critical section};
b[i] := false}
```

```
While true do
{b[i] := true
t := 3-i;
While t==3-i and b[3-i] do Skip;
{critical section}
b[i] := false}
```

Original tool

- Written entirely in CSP: essentially a functional program simulating a given shared variable one. A separate process for every thread, every variable, and every non-trivial expression calculation.
- Programs written as members of a CSP datatype:

```
datatype Cmd = Skip | Sq.(Cmd, Cmd) | SQ.Seq(Cmd) |  
Iter.Cmd | While.(BExpr, Cmd) | Cond.(BExpr, Cmd, Cmd, Cmd)  
Iassign.(ivnames, IExpr) | Bassign.(bvnames, BExpr, Cmd)  
Sig.Signals | ISig.(ISignals, IExpr) | Atomic.Cmd
```

Debug information inscrutable in the extreme!

Consequently not usable except by cognoscenti.

Expressions (Original tool)

- 2 Types: boolean and $\{MinI \dots MaxI\}$.
- Obvious range of operators over these.
- Fixed variables and arrays where the index must be a constant (useful for defining an array of processes, whose index is treated as a constant when compiling it, but very restricting).

Compiling a thread (2009 version)

```
MainProc(Skip,j) = SKIP
```

```
MainProc(Sig.x,j) = x -> SKIP
```

```
MainProc(Sq.(p,q),j) = MainProc(p,j);MainProc(q,j)
```

```
MainProc(SQ.<>,j) = SKIP
```

```
MainProc(SQ.<p>^Ps,j) = MainProc(p,j);MainProc(SQ.Ps,j)
```

```
MainProc(Iter.p,j) = MainProc(p,j);MainProc(Iter.p,j)
```

```
MainProc(While.(b,p),j) =
```

```
  let P(x) = if x then MainProc(p,j);MainProc(While.(b,p),j)
```

```
    else SKIP
```

```
    within BExpEval(b,P,j)
```

SVA: using CSP and FDR to analyse shared variable programs

```
MainProc(Cond.(b,p,q),j) =
  let P(x) = if x then MainProc(p,j) else MainProc(q,j)
              within BExpEval(b,P,j)

MainProc(Bassign.(el,e),j) =
  BLvEval(el,\lv@BExpEval(e,\rv@bvwrite.j.lv.rv->SKIP,j),j)

MainProc(Iassign.(el,e),j) =
  let P(lv,rv) = if rv>=MinI and rv<=MaxI then
                  ivwrite.j.lv.rv -> SKIP
                  else error.j -> STOP
              within ILvEval(el, \lv@IExpEval(e,\rv.P(lv,rv),j),j)

MainProc(ISig.(c,e),j) = let P(x) = c!x -> SKIP
                          within IExpEval(e,P,j)

MainProc(Atomic.p,j) = start_at.j -> MainProc(p,j); end_at.j ->
```

2007 Tool

Undergraduate project by David Hopkins led to:

- ASCII input language with parser/translator into above form.
- GUI
- Interpreter for debug information on a thread-by-thread basis.
- Extensions to handle prioritised execution (not particularly successful) and “monitors” (which I now rename **overseers**).

Overseer: a process that runs with priority every time one of a specified set of variables is written to. E.g. used to model more sophisticated data-types such as linear order or queue.

New in 2009/10

- Dynamically indexed arrays.
- Option of **dirty** variables.
- Expression calculating processes merged into threads, using continuations (see above and next slide).
- Atomicity regulated by variables rather than separate process.
- Overseer concept simplified.
- Subtypes of integers.
- Uses state reduction technique: **compression**.
- Refinement between program fragments.

Evaluating an expression

```
IExpEval(e,P,j) =  
  let IXEF(NoF) =  
    let k=evaluatei(e) within  
      if ok(k) and num(k)>=MinI and num(k) <=MaxI then  
        P(num(k))  
      else error.j -> STOP  
  IXEF(ISV.v) = ival.j.v?x -> IExpEval(subsi(v,x,e),P,j)  
  within IXEF(fetchi(e))
```

Example 1: Simpson's 4-slot algorithm

- **Dirty variable:** gives a nondeterministic result when read clashes with write.
- Usually use something like mutual exclusion to avoid such clashes.
- Simpson's algorithm is designed for cases where we do not want either side to have to wait: assumes one writer and one reader.
- Uses four alternative slots to read from and write to, and four boolean flag variables.
- Many presentations assume flags are clean; we will not.
- Desiderata: no clashing reads/writes of slots, no waiting, all reads give timely data, sequential consistency.

Simpson's algorithm writer

Shared: slot[], index[], latest, reading

```
Writer() = iter
{
  int wpair, windex, inp, wval;
  wval := inp;
  wpair := 1 - reading;
  windex := 1 - index[wpair];
  slot[2*wpair + windex] := wval;
  index[wpair] := windex;
  latest := wpair; }
}
```

Simpson's algorithm reader

```
Reader() = iter
{
  int rpair, rindex, outp;
  rpair := latest;
  reading := rpair;
  rindex := index[rpair];
  outp := slot[2*rpair + rindex];
  if outp = 1 then sig(collision); }
}
```

Simpson's algorithm development

- Original algorithm achieves cleanliness but not recentness in the presence of dirty flags.
- Algorithm can be refined by only writing flags when their values are going to change. This achieves recentness but not sequential consistency.
- No algorithm with memory-less reader can achieve sequential consistency.
- Augmenting and data-refining write-when-different algorithm produces a version with counters which achieves all the desiderata.

Simpson's algorithm lessons

- Lesson 1: Model checking is a lot easier than manual analysis when it is applicable! (The latter is very popular with Simpson's algorithm.)
- Lesson 2: You can use properties derived from model checking as part of a manual development: cleanness is needed to make the counters work, and we inherit recentness.

Version 1 of bakery algorithm: original

```
PP(i) = iter
{
  int j,temp; // declarations of local variables
  choosing[i] := true;
  temp := 0; j := 1;
  while j <= N do
    {temp := max(temp,number[j]); j := j + 1}; // Maximum
  number[i] := temp+1; choosing[i] := false;
  j := 1;
  while j <= N do
  {while choosing[j] do skip;
  if j >= i then while number[j]>0 && number[j] < number[i]
    else while number[j]>0 && number[j] <= number[i] do skip;
  j := j + 1; };
  sig(css.i); sig(cse.i); // critical section
  number[i] := 0;}
```

Version 2 of the bakery algorithm: “simplified”

```
PP(i) = iter
{  int j,temp;    // declarations of local variables
  turn[i] := 1;
  temp := 0; j := 1;
  while j <= N do
    {temp := max(temp,turn[j]); j := j + 1};
  turn[i] := temp+1;
  j := 1;
  while j <= N do
    { if j >= i then while turn[j]>0 && turn[j] < turn[i] do skip
      else while turn[j]>0 && turn[j] <= turn[i] do skip;
        j := j + 1 };
    sig(css.i); sig(cse.i); // critical section
  turn[i] := 0; }
```

Bakery algorithm: initial conclusions

- Maximum algorithm arbitrarily chosen.
- Only proved for small number of lanes, with bounded ticket values.
- Original version works (apparently!) for dirty variables, simplified does not.
- Implementation details (particularly maximum computation) somewhat arbitrary.
- Compression really, really works!

Principles of compression

- If computing something about the semantic value of the CSP process $C[P]$, can replace P by any semantically equivalent process.
- So replace by $compress(P)$, where $compress$ with fewer states.
- Can produce spectacular effects when multiplied and used hierarchically.
- Most used compressions: $sbisim$, $normal$ (can expand), tau_loop_factor and $diamond$.
- $diamond(P)$ is formed by removing τ actions and all those states only reached by τ in $diamond(P)$, subject containing the root node.
- $sbisim(diamond(P))$, equivalent to $normal$ on deterministic P is usually the best pragmatic choice.
- Compression usually follows hiding.

Compression in SVA

- Cluster local variables, and most appropriate shared variables, with each thread. Priority:
 1. Overseer variables go with overseer.
 2. User can define associations.
 3. Otherwise link variable to a thread that can write it.Hide local interactions and compress.
- Beyond that we can structure the threads into a tree and compress each non-root node after hiding local interactions:

```
hierarchCompress <<PP(1),PP(2)>, <PP(3),PP(4)>,PP(5),PP(6)>>
```

Refinement

- Translating into CSP automatically gives a theory of refinement: to make it useful we have to translate at the right level of abstraction.
- By default, SVA translates into a “closed-world” program, with all the reads and writes of variables hidden from view. This is not suitable for a program fragment.
- We need allow external code to write and read variables – sequentially and in parallel – but only in ways allowed by context.
- We need to consider the possible imposition of atomicity both within the program fragment and externally.

Refinement interface

Contexts might be atomic, parallel or general.

- Which variables can be read/written by other code in sequence, or where this code is atomic? SeqReads and SeqWrites.
- Which variables can be read/written by other code running in parallel with this? ParReads and ParWrites.
- Can a parallel program run an atomic section? `ext_atomic`
- We do not need to see if **this** code can go atomic.

Need for `ext_atomic`

Parallel context with variables initialised to 0, `ParReads={x,y}`,

`ParWrites={}`.

```
PP: iter {x := 1;      QQ: iter {x := 1;
      x := 0;         y := 1;
      y := 1;        x := 0;
      y := 0;        y := 0}
```

Equivalent if other threads do not use `atomic`. If they do only `QQ` allows the reads of `x` and `y` both as 1 in the same atomic section.

Four Maxes for the bakery

```
Maximum A:
temp := 0; j := 0;
while j <= N do
  {temp := max(temp,
    number[j]);
   j := j+1};
number[i] := temp+1;

Maximum B:
j := 0;
while j <= N do
  {number[i] := max(number[i],
    number[j]);
   j := j+1};
number[i] := number[i]+1;

Maximum C:
j := 0;
while j <= N do
  {temp := number[j];
   if temp > number[i]
   then number[i] := temp;
   j := j+1};
number[i] := number[i]+1;

Maximum D:
j := 0;
while j <= N do
  {if number[j] > number[i]
   then number[i] := number[j];
   j := j+1};
number[i] := number[i]+1;
```

Comparing Maxes

Equivalent as sequential programs.

In context, naturally for thread i :

- SeqWrites = SeqReads = {number[i]}
- ParWrites = {number[j] | $i \neq j$ }
- ParReads = {number[i]}
- ext_atomic = false

Result: Maximum A incomparable to others. B and C are equivalent and both refine D.

So proving for A establishes nothing for the others.....

Maxes in context

A, B, and C all appear to work in both versions of bakery algorithm.

But D is rather dodgy! Fails in simplified algorithm and substituting

```
j := 0;
while j <= N do
  {if number[j] > number[i] .. by.. if number[i] >= number[j]
   then number[i] := number[j];
   j := j+1};
number[i] := number[i]+1;
```

makes it fail in both!

D is not a reasonable max algorithm in a shared variable environment.

More about the bakery

Can prove the bakery algorithm for arbitrary number of lanes and unbounded ticket values: see book.

- Show two lanes achieve mutual exclusion no matter what the others do to their variables: even when there are a nondeterministic number of other lanes.
- Replace actual ticket values by their relative positions in the linear order, using overseer.

SVA: using CSP and FDR to analyse shared variable programs

SVA future

- Adding time and/or other barrier synchronisations?
- Mutual exclusion or Hoare-style monitors as primitive?
- Linking in to LTL checker?
- Further data types? Pointers?
- Version for practical languages?

SVA conclusions

- Compression makes this a very effective tool for safety conditions.
- Present functionality of FDR means it cannot support many liveness/fairness issues.
- Should be good for teaching.
- Description in CSP makes semantics extremely clear.
- Shared variable programs can be very hard to understand.

General conclusions

- CSP is an excellent language for implementing concurrent models.
- CSP_M makes it possible to write compilers/simulators in the language.
- Allows derivation of refinement theories.
- Future of FDR may well be as a back end for tools addressing other languages: statecharts, security, state machine descriptions (Verum etc)....