Modelling Without a Modelling Language

Antti Valmari & Vesa Lappalainen
University of Jyväskylä

1 Introduction

The starting point was teaching state space methods to students who

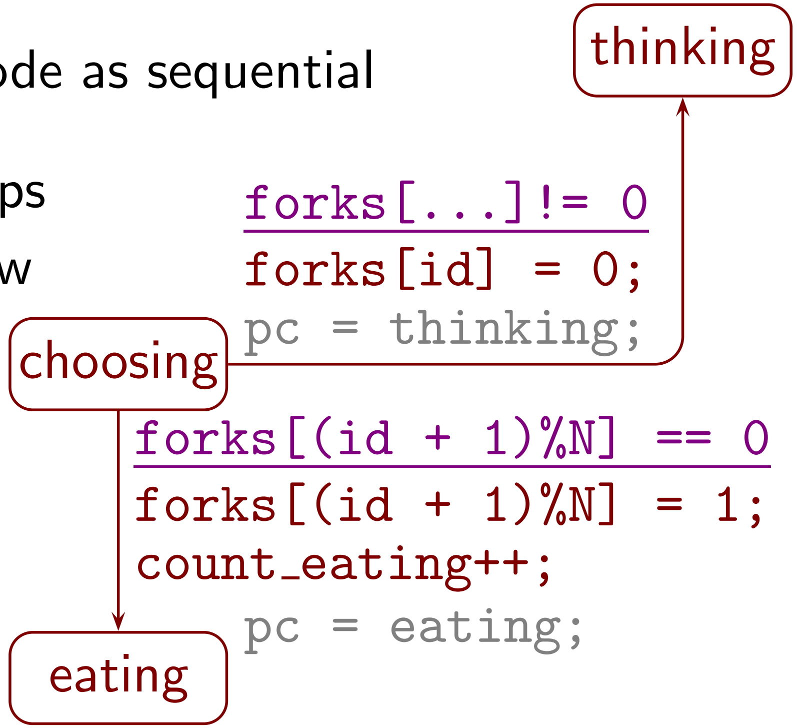
- know little theory
- know very little concurrency
- know C++

Pedagogical considerations

- wanted a clear, simple, but powerful enough abstract semantic model of concurrency

⇒ guarded transitions on shared variables

- students tend to incorrectly interpret concurrent code as sequential
  - it is hard to start thinking in a new way!
  - making concurrency aspects look unfamiliar helps
- outside concurrency, exploit what they already know
  - reduces unnecessary burden
  - ⇒ C++ for (atomic) sequential code



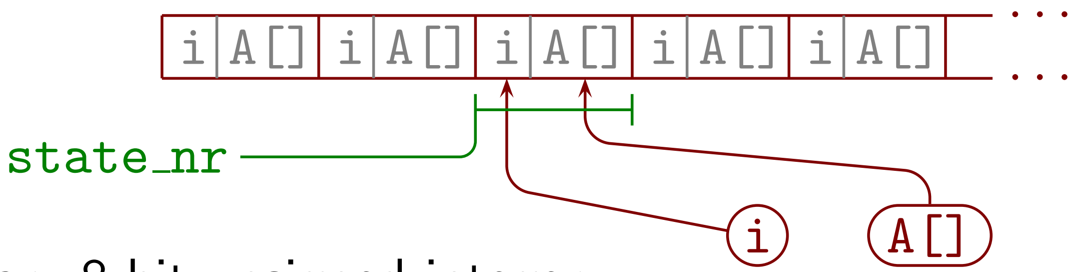
The tool had to be

- technically easy to start to use
  - download, *not* install
- not woefully slow

First version 2014

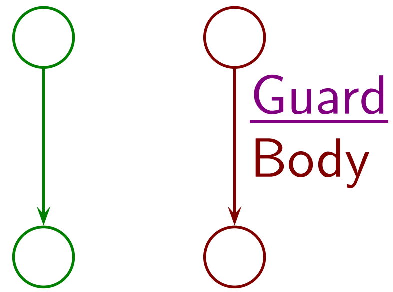
- quickly written
- the model is represented as a collection of C++ functions
  - `bool fire_transition( unsigned tr )` $\rightsquigarrow$ may change the global state
  - `void print_state()` $\rightsquigarrow$ readable error traces
  - `unsigned nr_transitions()` $\rightsquigarrow$ also set the initial state
  - `const char *check_state()` $\rightsquigarrow$ the string is the error message
  - `const char *check_deadlock()` $\rightsquigarrow$ illegal vs. legal termination
  - `bool is_may_progress()` $\rightsquigarrow$ **AG EF** progress
  - `bool is_must_progress()` $\rightsquigarrow$ **AG AF** progress
- download the tool, copy the model to `simple_mc.model`, compile, and run
- global state was a single (32-bit) unsigned integer
  $\Rightarrow$ models had to contain clumsy bit manipulation
- the transition relation was often a horrible mess of `if`'s and `switch`'s
- horrible — but did work pedagogically!

Improvements 2015: "ASSET"



- unrestricted global state size
- `state_var`
  - to the modeller, looks like an ordinary 8-bit unsigned integer
  - behind the scenes, accesses the data structure of found global states

Improvement 2018 (this talk)



- represent guards and bodies as C++ lambda functions
- develop (re-usable) classes for common usage patterns
  - `client_tr`: tail state and head state, indexed
  - `server_tr`: tail state, guard, body, and head state, indexed
  - cf. algorithms and data structures in libraries
  - more in the future?

$\Rightarrow$ rid of the messes of `if`'s and `switch`'s

Performance

- dining philosophers in Promela from [5]
- straightforward translation to ASSET
- re-use of `server_tr`

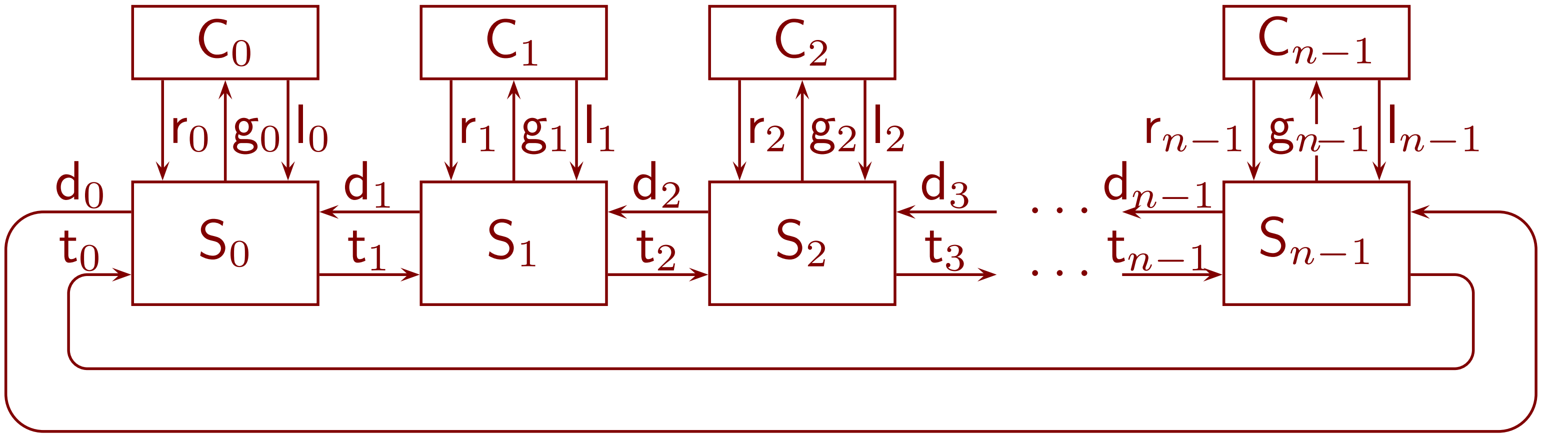
n	states	edges	time sec ASSET	time sec SPIN
10	154 450	1 145 997	0.6	0.3
11	510 116	4 153 629	0.9	1.2
12	1 684 801	14 936 051	2.1	4.7
13	5 564 522	53 351 654	7.2	18.4
14	18 378 370	189 489 700	32.4	80

2 Quick Comparison to Promela

```
bit forks[N];
byte count_eating;
proctype reset_phil(byte id) {
thinking:  ...
choosing:
  if
  ::atomic {
      forks[(id + 1)%N] == 0 ->
        forks[(id + 1)%N] = 1;
        count_eating++;
    };
  ::atomic {
      forks[(id + 1)%N] != 0 ->
        forks[id] = 0;
    }
    goto thinking;
  fi;
eating: ...
```

```
enum { thinking, choosing, eating, leaving };
state_bit forks[n];
state_var count_eating, S[n];
server_tr phils[] = {
  ...
  server_tr(
    choosing,
    GUARD( forks[(i + 1)%n] == 0 ),
    BODY( forks[(i+1)%n] = 1; ++count_eating; ),
    eating
  ),
  server_tr(
    choosing,
    GUARD( i == n-1 && forks[(i + 1)%n] != 0 ),
    BODY( forks[i] = 0; ),
    thinking
  )
};
```

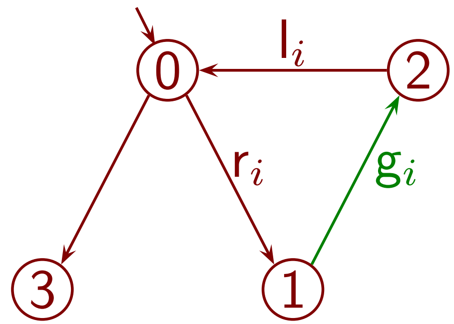
3 Demand-Driven Token Ring



State variables: `state_var C[n], S[n]; state_bit T[n];`

Clients

```
client_tr clients[] = {
  client_tr( idle, terminated ),  // termination
  client_tr( idle, requested ),   // request access
  client_tr( critical, idle )     // leave critical
};
```



Why termination branch?

- light-weight method to easily catch more progress errors
- [Valmari & Setälä 1996], [Valmari & Hansen 2018]

Servers

0: **wait until** C_i has requested or $S_{i \oplus 1}$ needs the token
goto 1

1: **wait** until I have the token
if C_i has requested **then** grant it permission; **goto** 2
else give the token to $S_{i \oplus 1}$; **goto** 0

2: **wait until** C_i has left its critical section
give the token to $S_{i \oplus 1}$; **goto** 0

```
server_tr(
  wait_token,
  GUARD( T[i] && C[i] == requested ),
  BODY( C[i] = critical; ),
  wait_client
),
server_tr(
  wait_token,
  GUARD( T[i] && C[i] != requested && S[ next(i) ] == wait_token ),
  BODY( T[i] = false; T[ next(i) ] = true; ),
  initial
),
```

direct access to `C[i]` and `S[next(i)]` made the model small enough to be fully shown in the paper

Example of Error Detection

```
#define chk_must_progress
bool is_must_progress(){ return C[0] != requested; }
```

- seed an error: after serving its client, the server does not automatically push the token forward

`print_state()`

```
-i*-i -i -i -i -i
-i*Ri -i -i -i -i
-i*Rt -i -i -i -i
-t*Rt -i -i -i -i
-i Rt*-i -i -i -i
==========
Ri Rt*-i -i -i -i
Rt Rt*-i -i -i -i
...
Rt  t  t -t -t -t*
Rt  t  t  t -t -t*
----------
Rt  t  t  t  t -t*
Rt  t  t  t  t Rt*
Rt  t  t  t  t Cc*
Rt  t  t  t  t -c*
Rt  t  t  t  t -i*
466560 states, 2954880 edges
!!!  Must-type non-progress error
```

Client 1 requests, so the token is fetched to Server 1

Client 0 requests in vain

the demand goes round, Client 1 is served, Clients 1 to 4 terminate

eternal loop where Client 5 is served again and again while Client 0 waits in vain

4 Simple Transition Classes

Nicer syntax for C++ lambda functions

```
typedef bool (*guard_type)( unsigned );
typedef void (*body_type)( unsigned );
#define GUARD(x) { [](unsigned i) {return x;} }
#define BODY(x) { [](unsigned i) {x} }
```

The essence of the server transition class

```
class server_tr{
  unsigned tail, head; guard_type guard; body_type body;
...
  bool operator()( unsigned i ) const {
    if( S[i] != tail || !guard( i ) ){ return false; }
    body( i ); S[i] = head; return true;
  }
};
```

- `i` is the index of the server
- transition is enabled $\Leftrightarrow$ control of the process is at its tail state and its guard holds
- transition fires $\Leftrightarrow$ its body is executed and the control moves to its head state

Firing of transitions

- not yet fully automated — this had to be written manually

```
inline bool fire_transition( unsigned i ){
  /* Servers */
  if( i < nr_server_tr ){
    return servers[ i % server_tr::cnt ]( i / server_tr::cnt );
  }
  /* Clients */
  i -= nr_server_tr;
  return clients[ i % client_tr::cnt ]( i / client_tr::cnt );
}
```

- the modeller wrote 3 client and 4 server transitions
- there are n clients and servers

$\Rightarrow$ ASSET uses $0, \ldots, 7n - 1$ as transition numbers

- the number is split to two parts:
  - a server or client transition is picked from an array of transitions
  - the index of the client or server goes as a parameter to the transition

A (non-)problem

- numerous calls of `fire_transition` with `S[i] != tail` or `C[i] != tail`

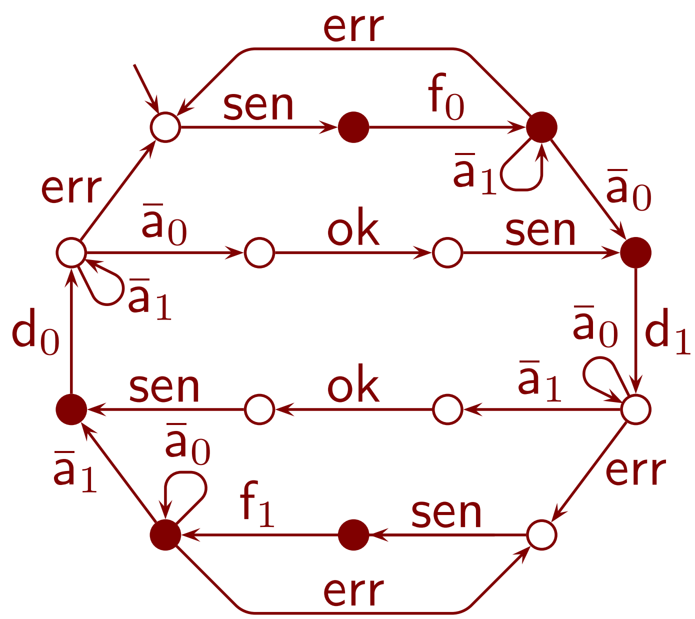
5 Faster Transition Classes

Transitions that are never simultaneously enabled may share their ASSET number

⇒ each process only needs as many transition numbers as its *degree of nondeterminism*
  – maximum number of simultaneously enabled transitions

- degree of nondeterminism $\leq$ outdegree of local state
- often $<<$ total number of transitions of the process

level					
0	sen	f_0	err	d_1	...
1	$\wp$	$\wp$	$\bar{a}_0$	$\wp$	
2	$\wp$	$\wp$	$\bar{a}_1$	$\wp$	



- local state `S[i]` selects the column

⇒ no need to test `S[i] == tail`

- $\wp$ is a transition whose guard is always `false`
- $\bar{a}_0$ and $\bar{a}_1$ can be a single transition, saving one level

In the measurements of this model, the manipulation of fifo's dominated analysis time

⇒ the simple solution was only little slower than the `if`'s and `switch`'s

⇒ there was no room for much improvement by the faster classes

6 Demand-Driven Token Ring Measurements

Demand-driven token ring state space size

n	7	8	9
states	2 939 328	20 155 392	136 048 896
edges	21 500 640	167 588 352	1 267 270 272

- edges $\approx$ 1.04 n states

$\Rightarrow$ most transitions are disabled in most states

(Relative) running times

			Faster trans. classes		Old technique		
n	hash	Simple	Lambda4	Lambda3	Switch7	Switch3	seconds
7	23	1.38	1.27	1.18	1.22	1.00	3.33
7	24	1.37	1.27	1.19	1.24	1.00	3.28
8	23	1.31	1.23	1.19	1.10	1.00	45.7
8	27	1.44	1.34	1.27	1.15	1.00	29.6
9	27	1.26	1.19	1.17	1.11	1.00	355
9	28	1.30	1.21	1.17	1.13	1.00	321
Old mini-laptop							
8	23	1.70	1.48	1.33	1.10	1.00	181

7 Conclusions

Guarded transition systems on shared variables, including process local state, can be expressed naturally using C++ lambda functions

The cost of lambda functions

- does not matter much on modern machines
  - hash table size matters more!
- is more significant on an old weak machine

Faster classes were indeed faster, but not very much

- they cannot be faster than `if`'s and `switch`'s
- simple classes were not much slower than `if`'s and `switch`'s

⇒ there was no room for much improvement

The following proved feasible:

- re-using a simple class in another model
- experimenting with a research idea without touching the core tool

Thank you for attention! Questions?