Synthesising Glue Logic Transactors, Multiplexors and Serialisors from Protocol Specifications.

David Greaves
Myoung Jin Nam
University of Cambridge
Computer Laboratory
UK

FDL 2010
Southampton
September 2010

The Design Exercise

- List participating interfaces and their protocols
- Connect them together, fully-reactively and without deadlock.
- Commonly just need data conservation, but
- Sometimes need other operations:

. Filtering

. Multiplexing

. Demultiplexing

. Buffering

. Serialising

. Deserialising

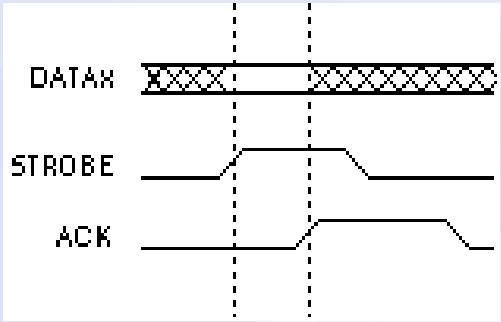
For any other processing:
Use an intermediate participant

Product Design Method

- Each participant protocol is defined by an NDFSA (automaton) over its nets
- Additional resources in our glue also have state (e.g. holding register is dead/live)
- Form full cross product of all participants
- Delete any arcs that violate data conservation or lead to deadlock
- Select a *preferred* direction at any remaining non-deterministic branch.

Four-Phase Handshake

Participant example:
Four-phase handshake (asynchronous parallel port protocol).

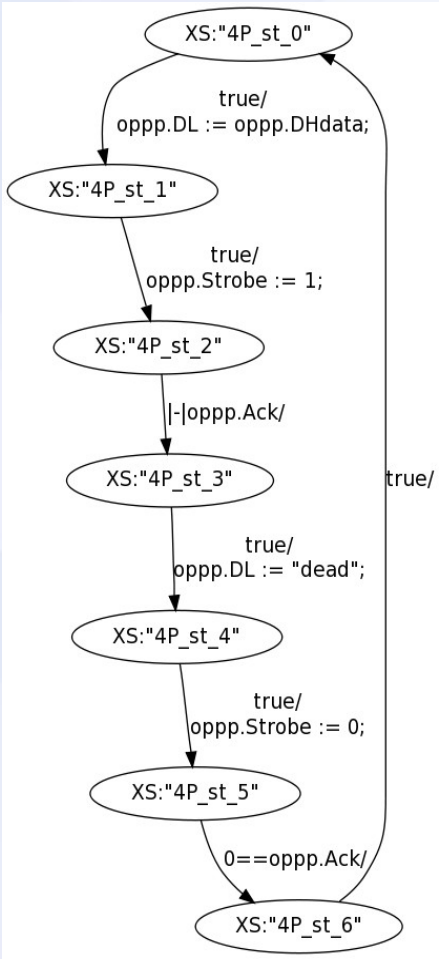


Interface:

```
output [7:0] data;
output Strobe;
input Ack;
```

Protocol:

```
four_phase_handshake_protocol(pred) =
   Seq[  Set(pred,        X_net "DL");
         Set(xi_num 1,    X_net "Strobe");
         Set(xi_num 1,    X_net "Ack");
         Set(deadval,     X_net "DL");
         Set(xi_num 0,    X_net "Strobe");
         Set(xi_num 0,    X_net "Ack")
      ]
;
```



Guards and commands shown are for output port and automatically Interchanged for input instance.

Protocol Description Language

Protocol P = Loop `of` (ρ) where

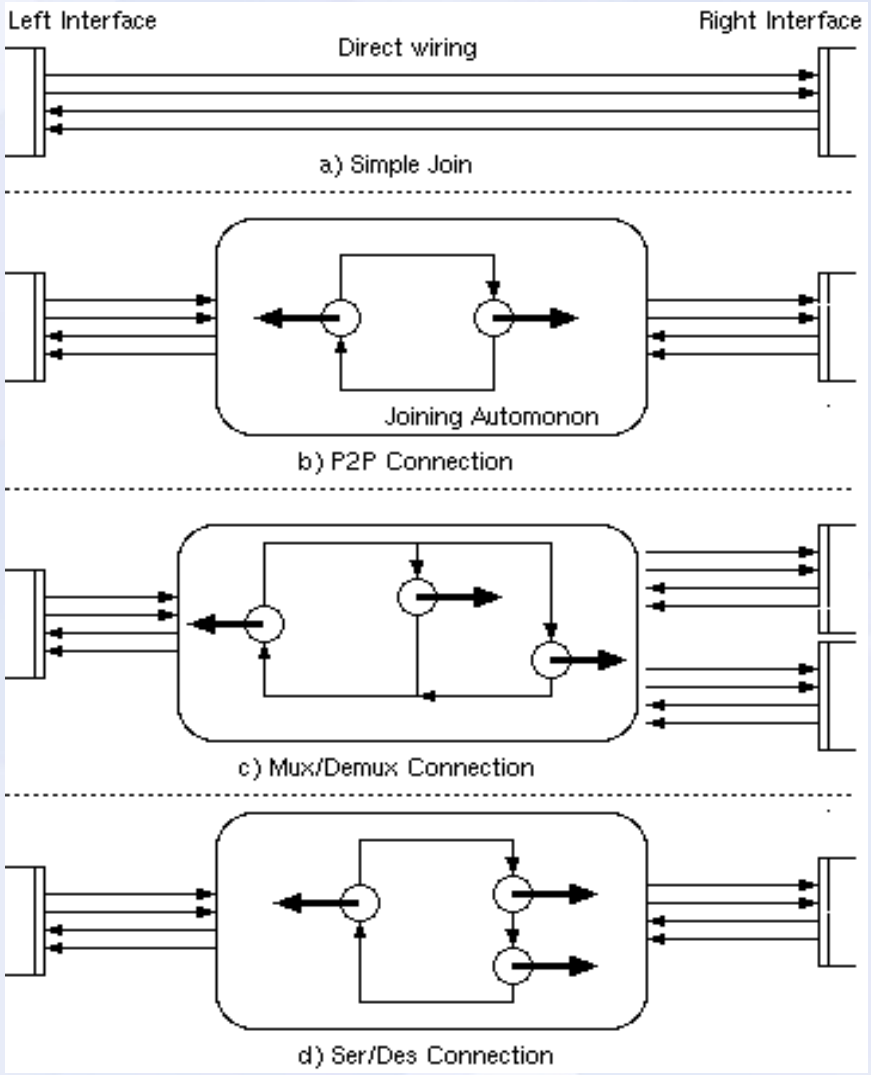
ρ =	Eq `of` $\alpha * \alpha$ `list`	// Parallel assignment
|	Seq `of` ρ `list`	// Sequencing
|	Disj `of` ρ `list`	// Non-deterministic branching
|	Next	// Wait one clock (same as Eq `nil`.)

where α is an integer expression ranging over the interface nets (Table II).

TABLE I

ABSTRACT SYNTAX FOR THE PROTOCOL CAPTURE LANGUAGE USED IN OUR EXPERIMENTS, GIVEN AS AN ML-LIKE DATASTRUCTURE.

Anything convertible to this form of non-deterministic FSA serves...



Typical Connection Patterns

- Can have more than 2 participants

- Some patterns require internal state

- Demultiplexing requires a routing predicate

- The filter pattern is a 1-to-2 demux with a /dev/null output port

- All patterns can be encoded using a simple algebra.

Interconnection Algebra

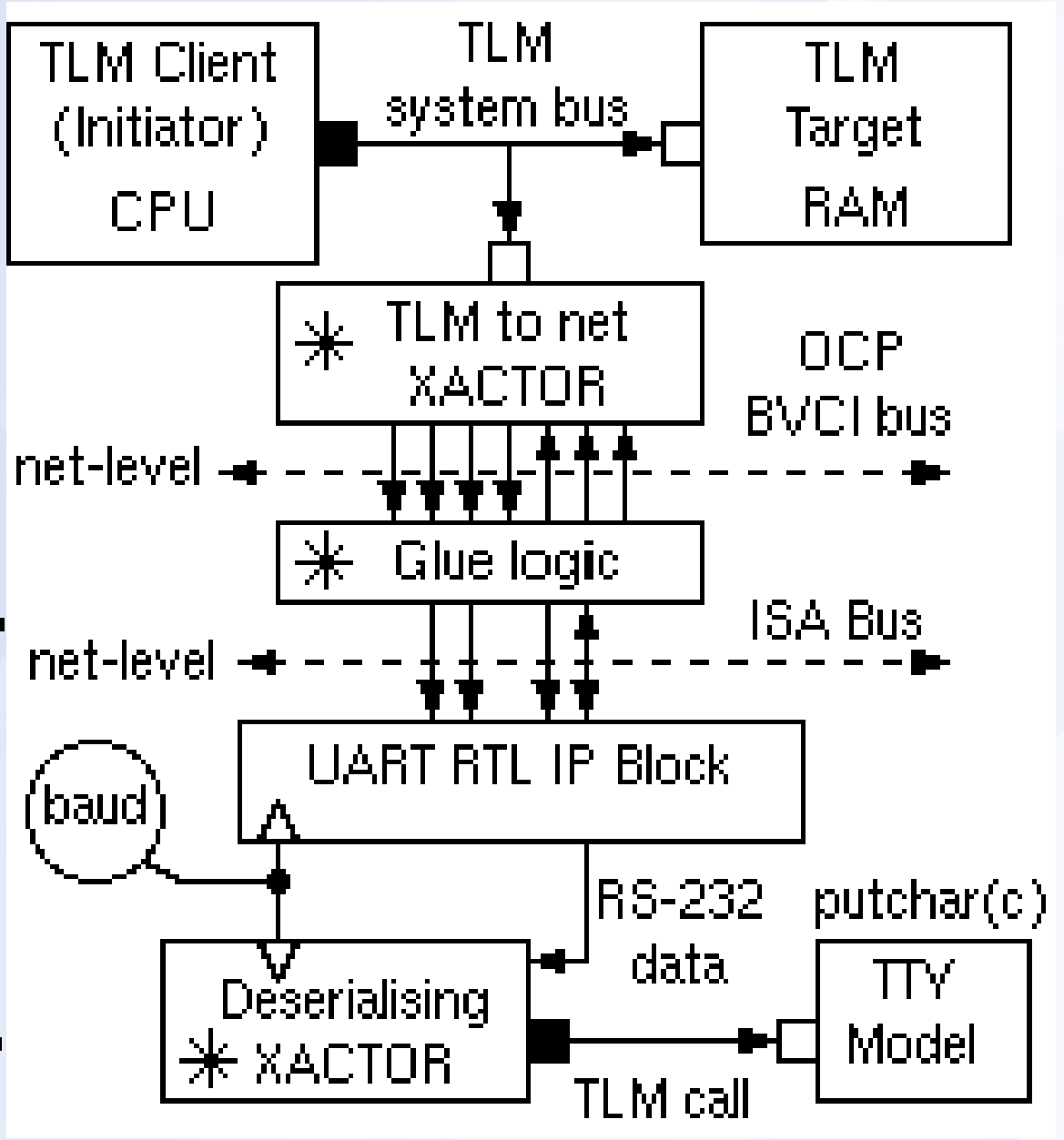
$\alpha =$	$\bot$	(dead)
$\mid$	D_n	(n-bit register)
$\mid$	$\alpha \mid \alpha'$	(bitwise OR)
$\mid$	$\alpha <<$ N	(constant left shift)
$\mid$	kill(α)	(kill expression)
$\mid$	$(\alpha, P_{\text{user}}(\alpha'))$	(expression guarded by predicate)

TABLE II

ABSTRACT SYNTAX FOR EXPRESSIONS HELD IN SYMBOLIC VALUES AT COMPILE TIME.

Typical Example showing three applications of our method.

Use for design exploration and synthesis.



Overall Tool Flow

Envisioned as an IP-XACT Eclipse Plugin

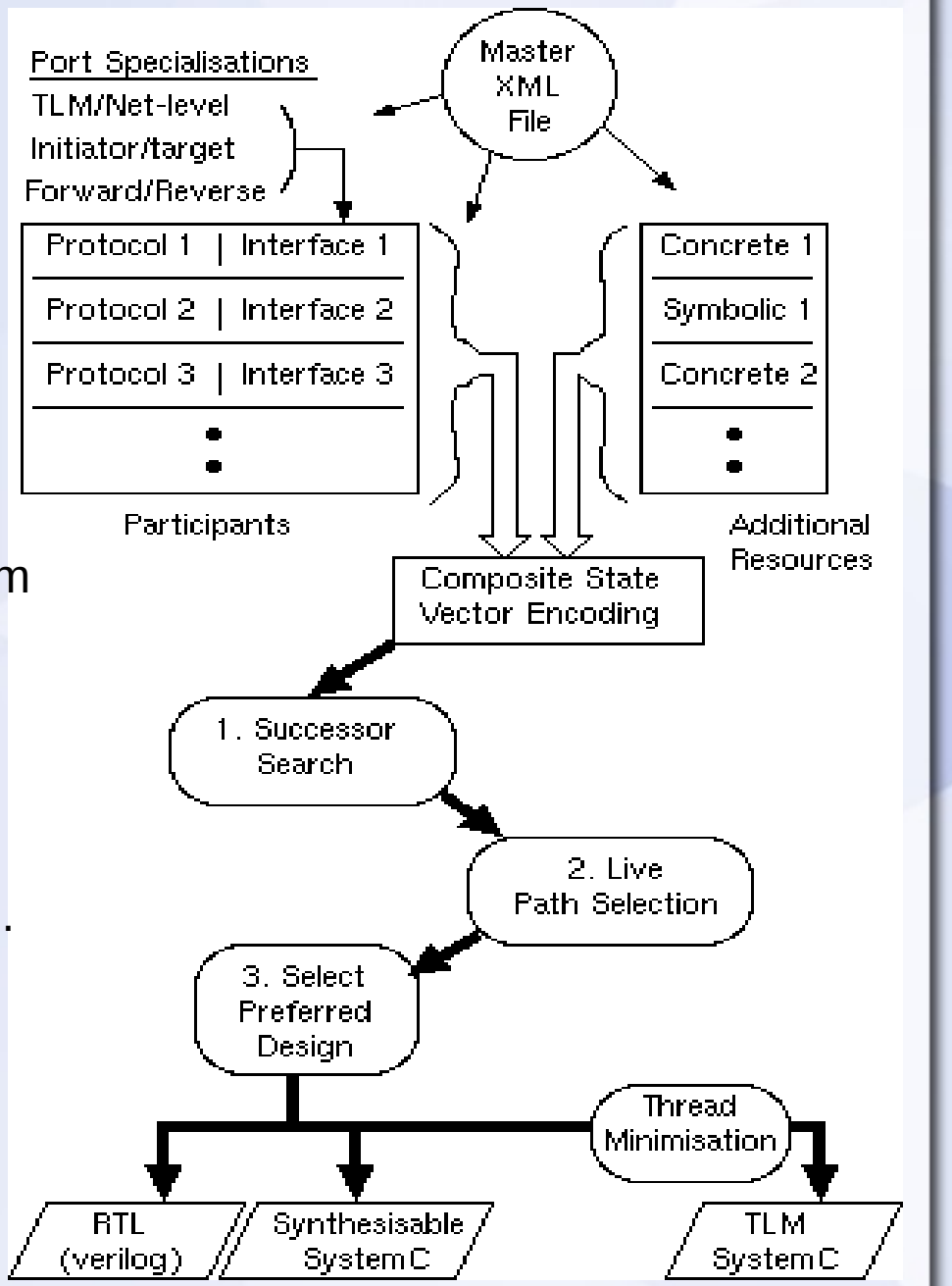
XML file pulls protocols and interfaces from library

Interfaces are parameterised with their direction and bus widths

XML file also contains glue equations (e.g. filter predicates)

Additional resources added by human.

Then an automatic procedure...



XML Example

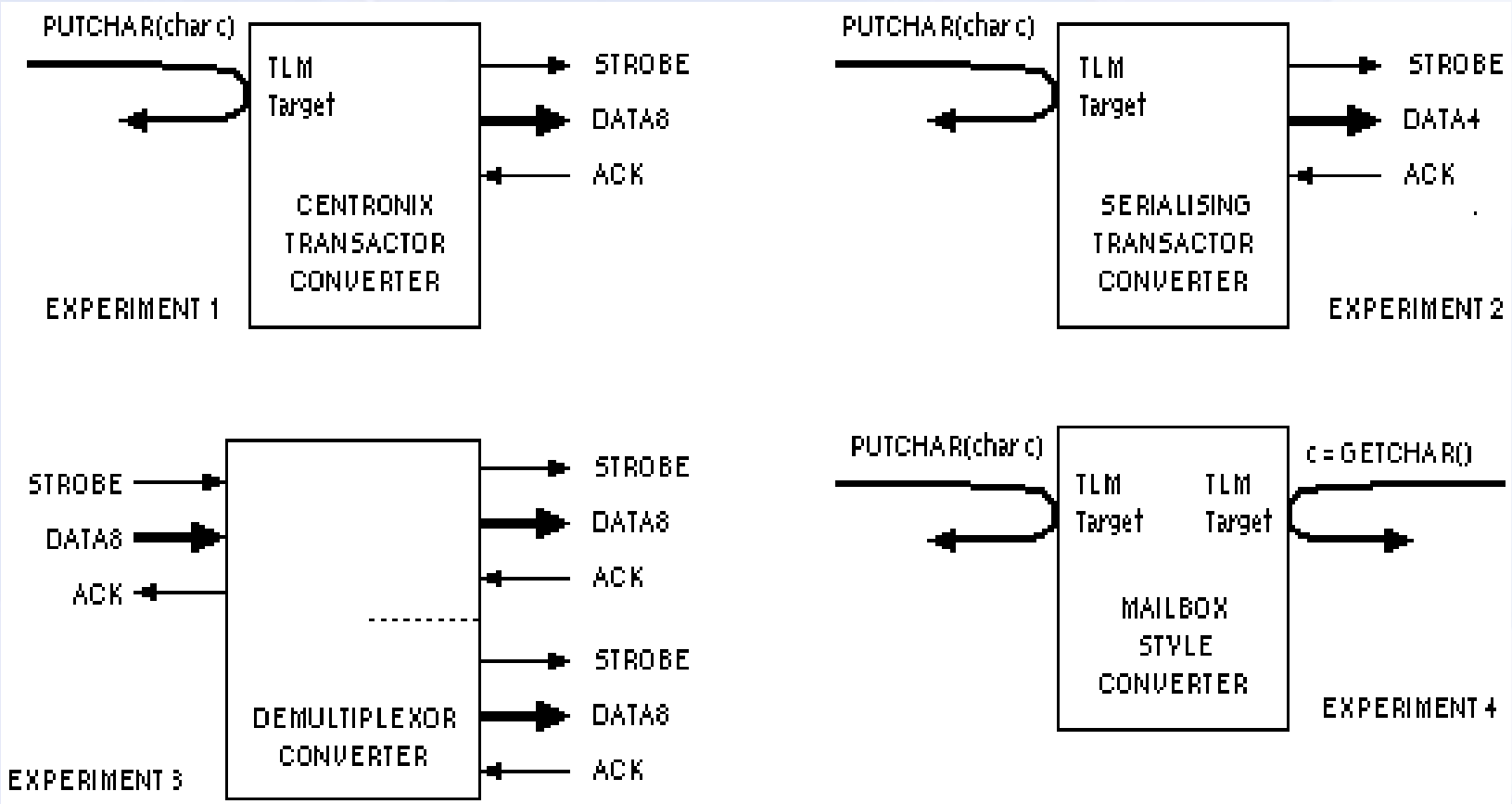
```
<?xml version='1.0' ?>
<joining name="bvisa">

  <participant>
    <iname>bv32</iname>
    <interface>bvci32</interface>
    <protocol>bvci32</protocol>
    <direction>reverse</direction>
  </participant>

  <equations>bvisa</equations>

  <participant>
    <iname>isa1620a</iname>
    <interface>isa1620</interface>
    <protocol>isa1620</protocol>
    <direction>forward</direction>
  </participant>

</joining>
```

Glue equations currently not yet in XML file – instead in a named F# source file...

But F# has some interesting domain-specific extensions ....

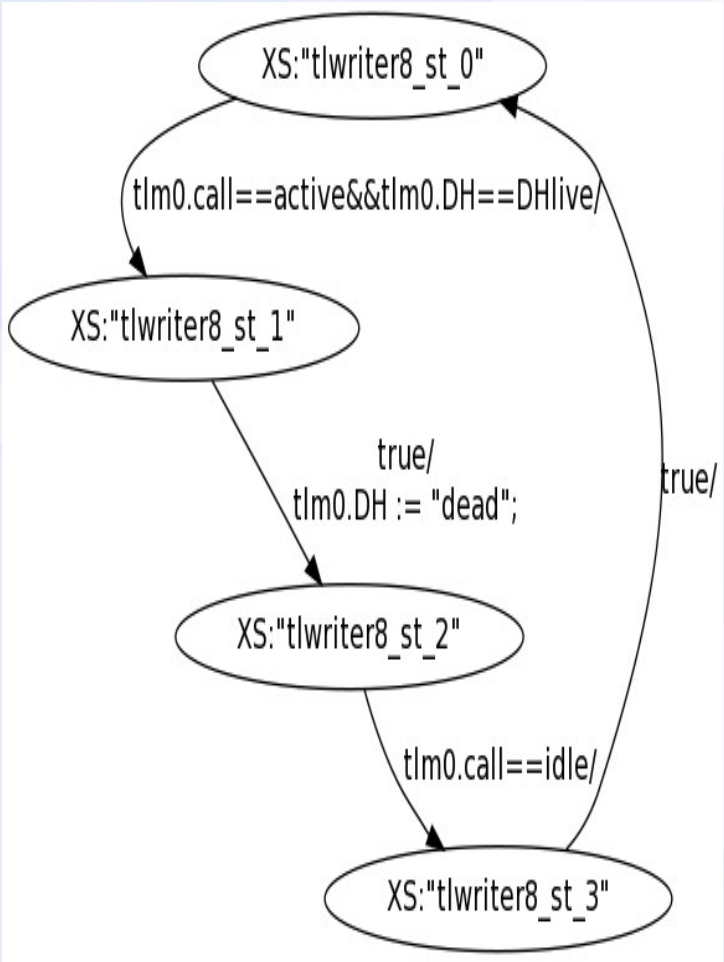
```
let  ioaddr = xi_bitor(ad_lo, lshift(predicate(kill ad_hi, xi_deqd(ad_hi, xi_num 17)), 20))
let  memaddr = xi_bitor(ad_lo, lshift(predicate(kill ad_hi, xi_deqd(ad_hi, xi_num 16)), 20))
;
```

Four examples



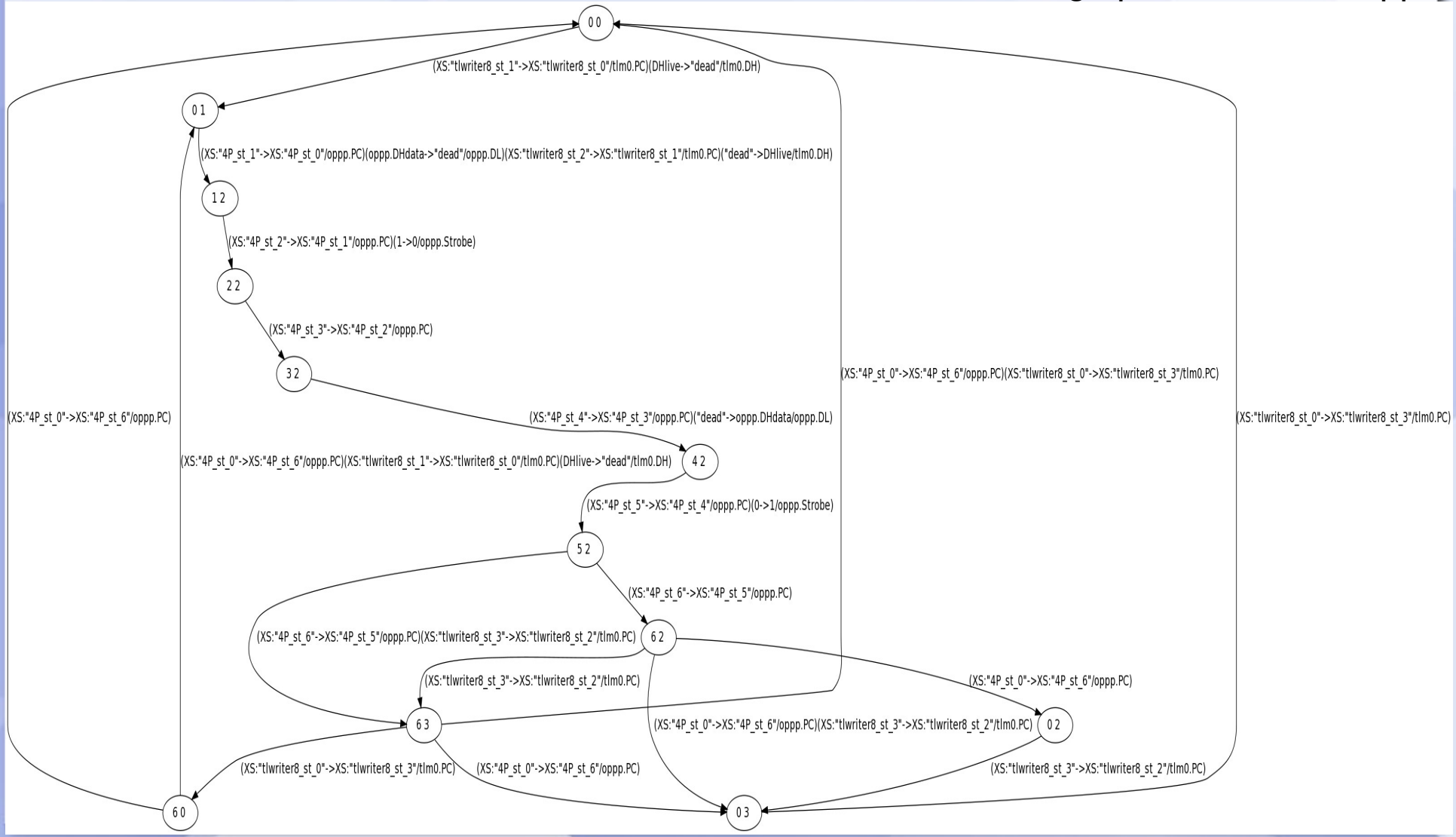
Ex 1: TLM Writer x 4/P

```
Protocol:
tlm_writer_handshake_protocol =
 Seq[
   Set(xi_uqstring "DHdata",  X_net "DH");
   Set(xi_uqstring "active",     X_net "call");
   Set(deadval,                  X_net "DH");
   Set(xi_uqstring "idle",       X_net "call");
       ]
;
```



Dot plot from intermediate stage of TLM to 4/P xactor:

- All data has been conserved
- Only live paths shown
- Non-deterministic choices retained
- Thread-trimming optimisation not applied

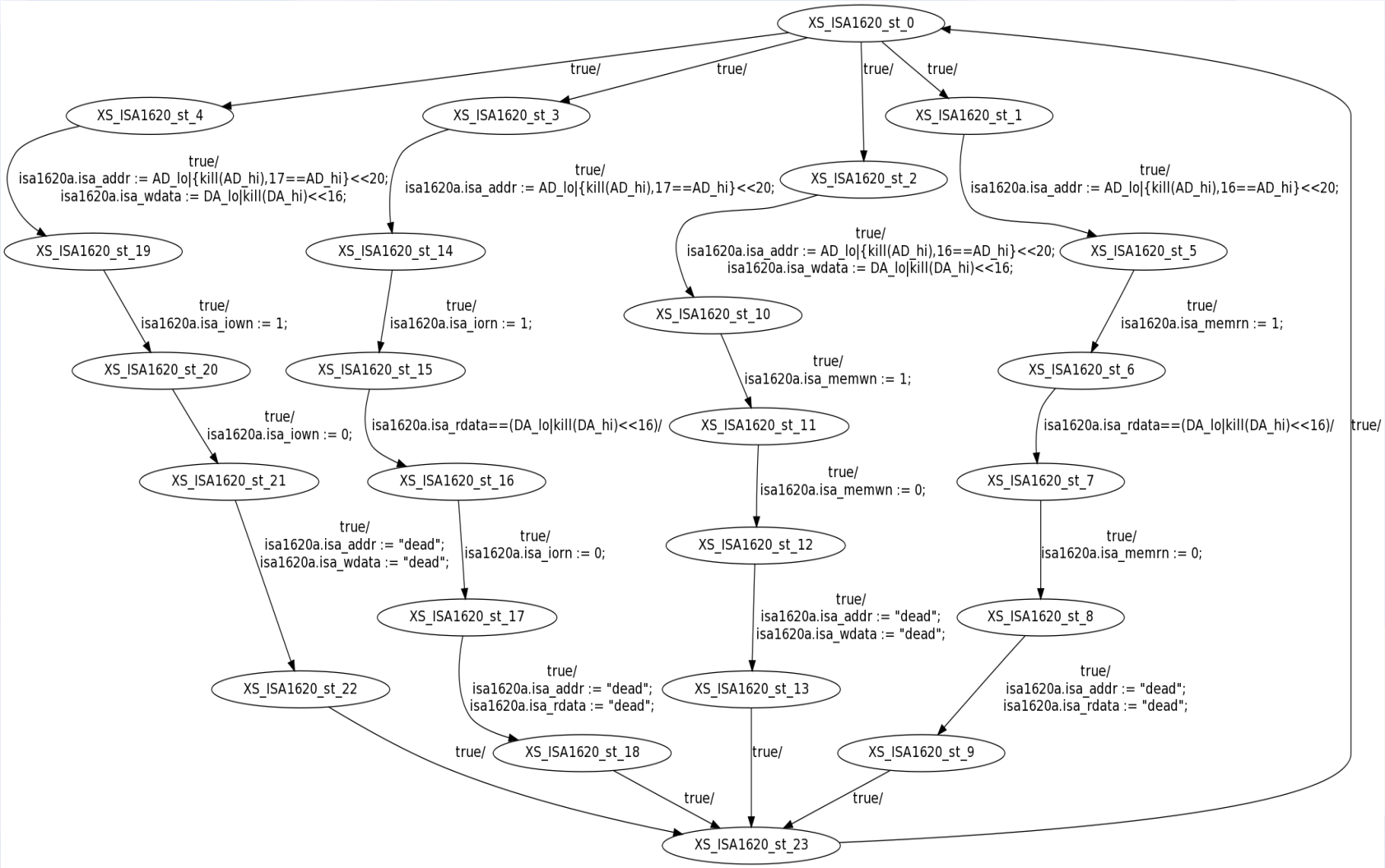


Concrete & Symbolic State

- Concrete state:
  - Eg: true/false, 0-9, {idle,read,write}
  - Set by driving component (output port)
- Symbolic state:
  - Eg: dead, D32, D8|(E8<<8)
  - Killed dead by receiving component
  - Set live by driving component
  - Chiseled at a serialisor until dead
  - Accumulated at deserialisor until correct width

Data Conserving Unification/ Congruence Rules

$$
\begin{array}{l}
\textbf{let rec } \mathbf{C} = \textbf{function} \\
\quad | \ (D_n,\ D'_m) \rightarrow ([D'_m := D_n],\ \mathbf{n{=}m},\ \bot) \ \text{// Width match} \\
\quad | \ (\alpha, P_u(\omega)) \rightarrow \\
\qquad\qquad \textbf{let } (c, g, \alpha') = \mathbf{C}(\alpha, \omega) \\
\qquad\qquad \textbf{in } (c, g \wedge P_u(\alpha'), \alpha') \ \text{// Predicate} \\
\quad | \ \mathbf{kill}(\alpha) \rightarrow \\
\qquad\qquad \textbf{let } (c, g, \alpha') = \mathbf{C}(\alpha, \omega) \\
\qquad\qquad \textbf{in } (c, g, \bot) \ \text{// Kill} \\
\quad | \ (\alpha_l \mid \alpha_r, \omega) \rightarrow \\
\qquad\qquad \textbf{let } (c, g, \alpha') = \mathbf{C}(\alpha_l, \omega) \\
\qquad\qquad \textbf{in } (c, g \wedge (\alpha' = \bot), \alpha_r) \ \text{// Serialise} \\
\quad | \ (\alpha_l,\ \omega_l \mid \omega_r) \rightarrow \\
\qquad\qquad \textbf{let } (c, g, \omega') = \mathbf{C}(\alpha,\ \omega_r) \\
\qquad\qquad \textbf{in } (c, g \wedge (\omega' = \bot), \omega_l) \ \text{// Deserialise} \\
\quad | \ (\alpha{<<}N,\ \omega) \rightarrow \\
\qquad\qquad \textbf{let } (c, g, \alpha') = \mathbf{C}(\alpha{>>}N\ ,\ \omega) \\
\qquad\qquad \textbf{in } ([(\alpha{>>}N)/\alpha]c, g, \bot) \ \text{// Shift out} \\
\quad | \ (\alpha, \omega{<<}N) \rightarrow \\
\qquad\qquad \textbf{let } (c, g, \alpha') = \mathbf{C}(\alpha,\ \omega) \\
\qquad\qquad \textbf{in } ([(\omega{<<}N)/\omega]c, g, \bot) \ \text{// Shift in}
\end{array}
$$



ISA Bus Participant: Non-det branch to four paths: mem/io read/write.

```
let isa_nets =
  [ (simplenet "isa_iown",       Ndi OUTPUT, gen_Concrete_enum [xi_num 1; xi_num 0]);
    (simplenet "isa_iorn",       Ndi OUTPUT, gen_Concrete_enum [xi_num 1; xi_num 0]);
    (simplenet "isa_memwn",      Ndi OUTPUT, gen_Concrete_enum [xi_num 1; xi_num 0]);
    (simplenet "isa_memrn",      Ndi OUTPUT, gen_Concrete_enum [xi_num 1; xi_num 0]);
    (vectornet_w("isa_addr", 20),     Ndi OUTPUT, gen_Symbolic_bitvec (16));
    (vectornet_w("isa_rdata", 16),    Ndi INPUT, gen_Symbolic_bitvec (16));
    (vectornet_w("isa_wdata", 16),    Ndi OUTPUT, gen_Symbolic_bitvec (16));
  ]
```

ISA A20/D16 Nets

```
let isa_mem_read =
 Seq[  Set(memaddr, X_net "isa_addr");
       Set(xi_num 1, X_net "isa_memrn");
       Set(data32, X_net "isa_rdata");
       Set(xi_num 0, X_net "isa_memrn");
       Setl [ (deadval, X_net "isa_addr"); (deadval, X_net "isa_rdata"); ];
  ];
```

ISA Read Mem Protocol

ISA Disjunction of Paths

```
    // Make an alternation of four basic cycles:
    let isa_legacy_protocol =
        Disjunction[       isa_mem_read; isa_mem_write; isa_io_read; isa_io_write;  ]

    let isa1620 = ("ISA1620", isa_nets,  isa_legacy_initial, isa_legacy_idle, Synch isa_legacy_protocol)
```

ISA A32/D32 embedding equations

```
    // Glue equations for A32/D32 mapping :
    let   ioaddr = xi_bitor(ad_lo, lshift(predicate(kill ad_hi, xi_deqd(ad_hi, xi_num 17)), 20))
    let  memaddr = xi_bitor(ad_lo, lshift(predicate(kill ad_hi, xi_deqd(ad_hi, xi_num 16)), 20))
    let data32 = xi_bitor(data_lo, lshift(kill data_hi, 16))
```

Guiding Heuristic

- With causal participants eliminating all but one decision at a non-det choice gives working design.
- Heuristic choice during synthesis: chose a design that:
  - Makes as many changes as possible.
  - Executes most rapidly (i.e. shortest path to idle),
  - Good for timing closure (at least one register delay per net),
  - Shares commonality as much as possible (lowest overall complexity),
  - For TLM: enables work to be packed on fewer threads.

Transactional Ports

- TLM modelling: subroutine calls instead of nets.
- Transactor: *glue logic* with some mix of TLM and net-level ports.
- TLM ports are some mix of initiator and target
- Transactor: may *multiplex* various TLM calls over various net-level interfaces.
- Many transactors have no work to do while between transactions... can save on threads.

Call Active Concrete Bit

- Assume TLM calls are non-reentrant.
- Additional boolean concrete state flag records call phase: active/idle.
- Then treat as a net-level port with access restrictions:

When idle (not active), initiator can:
. Write argument expressions
. Set active flag
. Return value can be read

When active, target can:
. Read arg values
. Write return value
. Clear active flag

- Thread reduction possible ?
  - For initiator if nothing to do while active
  - For target, if nothing to do while idle

Search Space Exponential?

- Most free inputs only connect to one participant, therefore
  - Search space around a component is exponential
  - Search space as number of participants increases grows more slowly
  - But we consider stuttering composition of components, which is exponential
  - Perhaps phrase as SAT problem ?

Four example results:

Exp	Participants concrete states no.	Product no. states explored	Converter no. states live paths	SystemC no. lines
1	$4 \times 6 = 24$	72	71	1070
2	$4 \times 6 = 24$	123	123	1848
3	$6 \times 6 \times 4 = 144$	575	575	14198
4	$4 \times 4 \times 2 = 32$	325	324	7534

```
$ mono ./joiner.exe -o bvisa.cpp bvisa.xml -cpp bvisa.cpp -vnl bvisa.vnl
Forming participant iname=bv32 protocol=bvci32 inteface=bvci32 direction=reverse
Forming participant iname=isa1620a protocol=isa1620 inteface=isa162 direction=forward
Considered 100 with 187 states to explore...
Considered 200 with 265 states to explore...
Considered 300 with 308 states to explore...
Considered 400 with 193 states to explore...
Considered 500 with 267 states to explore...
Considered 600 with 330 states to explore...
Considered 700 with 187 states to explore...
Considered 800 with 105 states to explore...
Considered 900 with 289 states to explore...
Considered 1000 with 291 states to explore...
Considered 1100 with 334 states to explore...
Considered 1200 with 176 states to explore...
Considered 1300 with 1204 states to explore...
Considered 1400 with 1237 states to explore...
Considered 1500 with 2068 states to explore...
Considered 1600 with 2022 states to explore...
Considered 1700 with 2108 states to explore...
Considered 1800 with 1973 states to explore...
Considered 1900 with 131 states to explore...
** Warning: The following command line args were unused -vnl, bvisa.vnl
Finished Product Construction (after considering 1933 states).
Finished Basic Live Path Determination MJN Join: states left=397

$ wc bvisa.cpp bvisa.h
  3282    7082 1234486 bvisa.cpp
    41     142    1078 bvisa.h
  3323    7224 1235564 total
```

Compile Time

Interpreted F# implementation

2K non dead-end states processed in 10 minutes with much logging turned on.

Conclusions

- Surprisingly versatile technique!
- Sometimes needs some branding e.g. to distinguish wdata32 from waddr32
- Selecting shortest path leads to lowest complexity in h/w or s/w, but, for h/w we may need at least one register for timing closure and stick with the longer path always when used at all is probably sensible.
- Not yet clear whether useable designs result without post-processing bisimulation reduction.