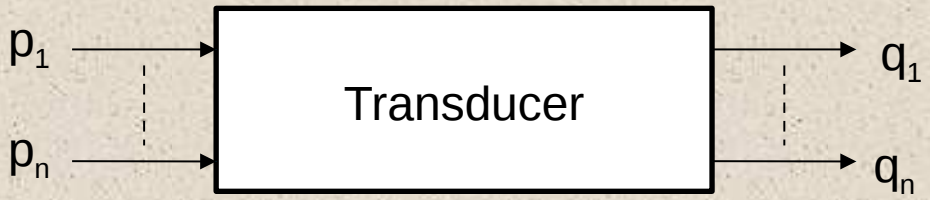
Guided Reactive Synthesis with Soft Requirements

Reactive Synthesis from QDDC

- Specify the input-output behaviour in a high-level language e.g. a temporal logic (QDDC)
- *Problem Statement*: Given an *interface* definition and *temporal spec Φ* over the set of input and output variables, automatically synthesize transducer M (Implementation), s.t. all behaviours of M satisfy Φ (or conclude that such transducer does not exist.)
- Synthesize an Implementation, that generates outputs from the given set of inputs (Transducer).



- Designed & implemented an tool DCSynthG which constructs such a transducer in SCADE/SMV format from QDDC.

Running Example: Minepump

- Minepump to keeps the water level in a mine under control for the safety of miners using a pump driven by a *controller*.
  - ❑ Mines are prone to methane leakage trapped underground
- Interface: HH2O, HCH4(Inputs),
  - Inputs: HH2O, HCH4
  - Outputs: Alarm, PumpOn
- Assumptions (*QDDC+NL*):
  - Sensor reliability assumption: *ppref* (DH2O ⇒ HH2O)
  - Water seepage assumptions: *tracks*(HH2O, DH2O, k_1)
  - Pump capacity assumption: *lags*(PumpOn, !HH2O, k_2)
  - Initial condition assumption: *init*(<!HH2O> && <!HCH4>, slen = 0)

Running Example: Minepump

- Commitments (*QDDC+NL*):
  - Alarm control: *lags*(HH2O, Alarm, k_5) and *lags*(HCH4, Alarm, k_6) and *lags*(! HH2O && !HCH4, !ALARM, k_7)
  - Safety condition*: ppref (!DH2O && (HCH4 ⇒ !PumpOn)).*
- Assumption and Commitments requirement specifications are given as timing diagram.
- Goal: Automatically synthesize an implementation that guarantees
  - Assumptions => Requirements
- Multiple correct implementation that satisfies the specification
  - Guided synthesis based on soft requirements.

Our Work

- Generate a Finite State Automaton for the requirement specification given as logic formula (QDDC).
- Apply Symbolic search technique for given input output partitioning
  - To find the behaviours in FSM that satisfy the safety specification (Maximally permissible non-deterministic automaton: MPNC)
  - Symbolic Algorithm (Without automaton spreading)
  - User guided formula based strategy taken as soft requirements, to determinize MPNC and generate LODC.
  - Encode LODC in required language (SCADE/SMV/Verilog)
- Generate Explanation, if specification is unrealizable

Algorithm for Safety Specification

- $\phi_{safe} = \Box(\Lambda_{i \in I1} D_i)$ with input/output partitioning.
- Compute Automaton $A(\phi_{safe}) = \{S, S_{0,} \delta, G, \sum^{(<,I,O)} \}$
  - ***Theorem (Decidability of QDDC)*** : For every QDDC formula *D*, we can effectively construct a finite state automaton *A(D)* over alphabet $VAL(P_{var})$, s.t. $\forall \sigma \in VAL(P_{var})^+$, $\sigma \models D$ *iff* $\sigma \in L(A(D))$, *where* P_{var} *are the variables used in the QDDC formulae D*
- C_{step}: $S \times 2^S \rightarrow \{1, 0\}$, for $i \in I$ and $o \in O$ and $s \in S$.
  - $C_{step}(s, G) = 1$, if $\forall i. \exists o : \delta(s, (i, o)) \in G$
  - $C_{step}(s, G) = 0$, if $\exists i. \forall o : \delta(s, (i, o)) \in (S-G)$
- V : Q -> {1, 0}, Value function from a state in the automaton A(D) to boolean.
- Reach(A(D), G): Set of states $G_{cntr} \subseteq S$ s.t. it is possible to controllably reach G.

Algorithm for ϕ_{safe} i.e. $\Box(\Lambda_{i \in I1} D_i)$

Algorithm: Computation of Reach(A(D), G):

Input: A(D), G, Input-output partitioning

Output: Reach(A(D), G)

Set $V'(s) = 1, \forall s \in G$ and $V'(s) = 0, \forall s \in (S-G)$

Set $G_{reach} = G$

Do

for each $s \in G_{reach}$ do

If $Cstep(s, G_{reach}) = 0$ then

$V'(s) = 0$

$G_{reach} = G_{reach} - s$

While (*previous* $V' \neq V'$ *i.e. we reach a fix-point*)

Return (G_{reach})

Algorithm for ϕ_{safe} i.e. $\Box(\Lambda_{i \in I1} D_i)$

- Realiazability Check
  - If initial state $s_0 \in$ Reach(A(D), G) then requirement specified by D are realizable.
- Compute MPNC (If specification is realizable)
  - Starting from s_0, only those paths in the automaton A(D), which keep us in Reach(A(D), G), where G is set of acceptable state.
- Compute LODC
  - Determinize the MPNC based on the soft requirements.

Soft Requirements

- Hard requirements are the requirement, which must always be obliged by the implementation.
- Mostly hard requirements are incomplete leading to non-deterministic implementation (with several output choices). but we need deterministic implementation.
  - One of the naive way is to randomly select one of the possible correct implementations.(Unsatisfactory!)
  - Choice of output can have major impact on performance.
- ***Soft Requirements***: soft requirement is a ordered list of propositional formulae $L=[P_0, \cdots, P_n]$
- **Locally Optimal Controller**: Algorithm to extract a deterministic implementation that satisfies the maximal prefix of soft requirements at each step.

DCSynthG : Soft Requirements based synthesis for Minepump

- Minepump_Soft_PumpOff [!PumpOn >> Alarm] : Implementation that tries to keep pump off as much as possible i.e. switch on the pump only when it cannot be avoided. (87 States)
- Minepump_Soft_PumpOn [PumpOn >> !Alarm] : Implementation that switch on the pump as soon as possible. (32 States)
- Minepump_Soft_MethanSafe [(HCH4_2 => !PumpOn) >> PumpOn)] : Implementation that tries to keep pump off if there is a methane leak in last 2 cycles otherwise switch on the pump. (43 States)

Performance Measurement Example: Maximum time for which water can remain high (HH2O) for each of these implementation?

Results: 8, 4 and 6 cycles respectively.

Generalization of Logic

- We propose to generalize the synthesis algorithm subset of LTL[DC] specification, which allows specification of unbounded liveness requirements in the logic.
- The Fragment of LTL[DC] considered for synthesis

$$\phi = \phi_{safe} \wedge \phi_{persistent} \wedge \phi_{surveillance}$$

where

$\phi_{safe} = \Box(\bigwedge_{i \in I1} D1_i)$ (Safety Specification)

$\phi_{persistent} = \Diamond \Box (\bigwedge_{i \in I2} D2_i)$

$\phi_{surveillance} = \bigwedge_{i \in I3} (\Box \Diamond D3_i)$

Here $D1_i$, $D2_i$, $D3_i$ are QDDC formulae and I_1, I_2, I_3 are finite index sets.

Experimentation

- The tool for synthesizing specification encoded in the form of ϕ_{safe}, has been used to synthesize a few examples.
- Algorithm produces Transducer as SCADE/SMV program realizing the given specification.
- Automatically synthesized program for few examples
  - N-cell Arbiter ((With soft requirements))
  - Minepump (With soft requirements)
  - Traffic Light
  - Alarm Annunciation Logic

N-Cell Arbiter

-- we consider specification for the arbiter with atmost 3 requests at a time. There are corresponding 3 acknowledgement lines also.
var req1, req2, req3, ack1, ack2, ack3;
const n=3 ;

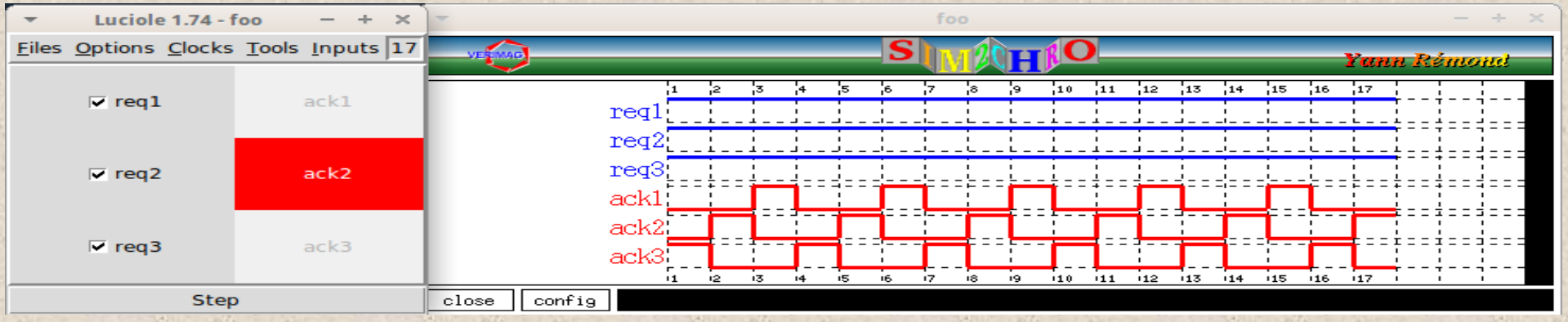
--**Spec1**: Following formula (Exclusion) says that atmost 1 acknowledgement can be given at a time.
define Exclusion as
[[(ack1 => !(ack2 || ack3)) && (ack2 => !(ack1 || ack3)) && (ack3 => !(ack1 || ack2))]];

-- **Spec2**: Formula 'Noloss' says that whenever there is atleast one request, then one of then should be grated the access i.e. the cycle should not be lost.
define Noloss as
[[(req1 || req2 || req3) => (ack1 || ack2 || ack3)]];

-- **Spec3**: The formula 'NoSpuriousAck' says that a request should be granted access to the bus only if it has requested it.
define NoSpuriousAck[req, ack] as
[[ack => req]];

-- **Spec4:** One of the most important property of an arbiter is that it should be fair to all the request i.e. it should grant access to the request if it is continuously true for some bounded amount of time (taken as 'n' is the following formula)
define Response[req, ack] as
[] ([[req]] && slen=n-1 => <> <ack>);

infer
Exclusion && Response[req1,ack1] && Response[req2,ack2] && Response[req3,ack3] && Noloss && NoSpuriousAck[req1, ack1] && NoSpuriousAck[req2, ack2] && NoSpuriousAck[req3, ack3]

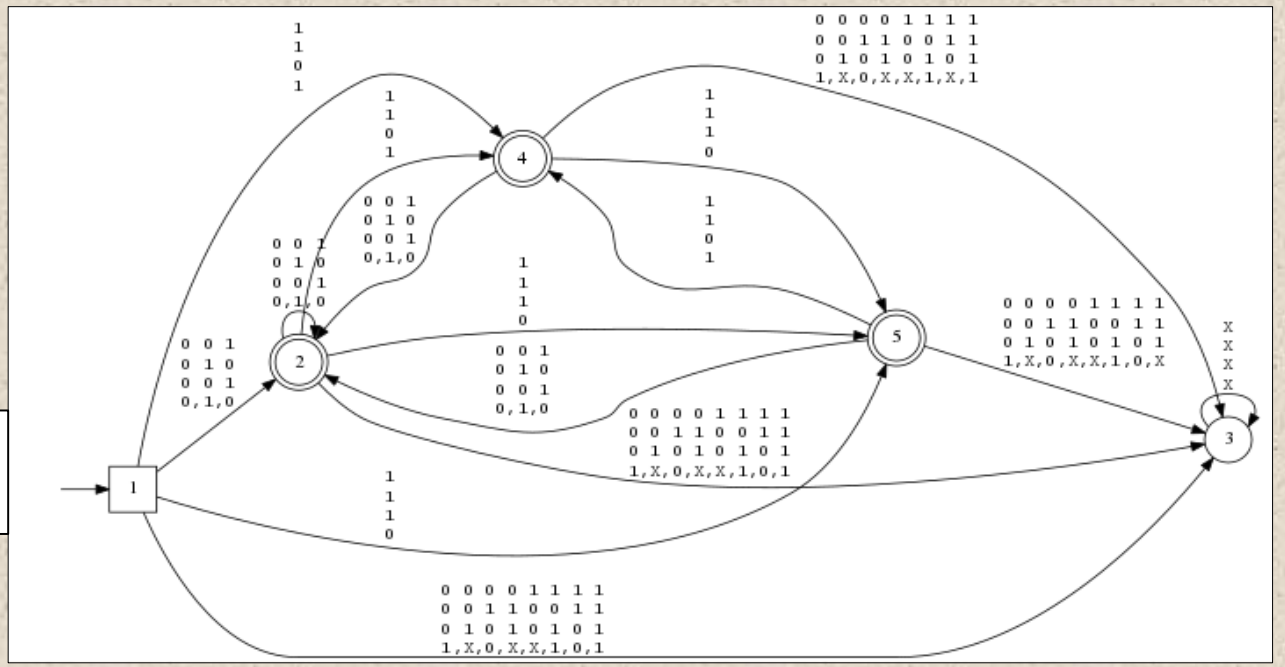


Workflow Example: 2 Cell Arbiter

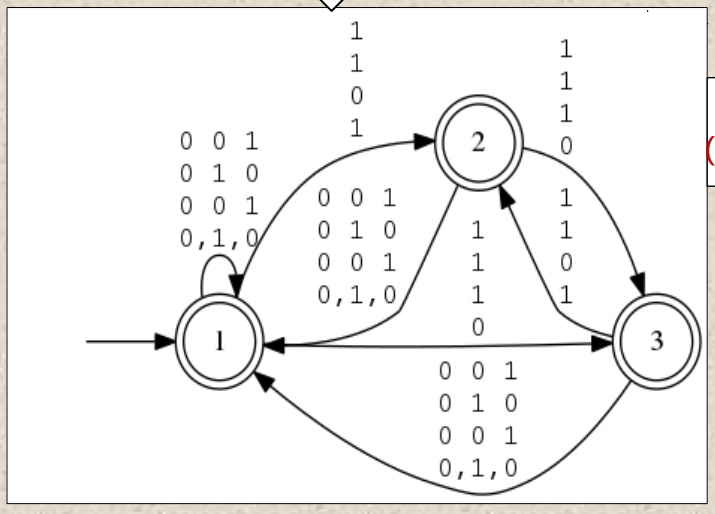
2 Cell: QDDC Specification

- MutualExclusion
- NoLostCycle
- NoSpuriousAck
- Response

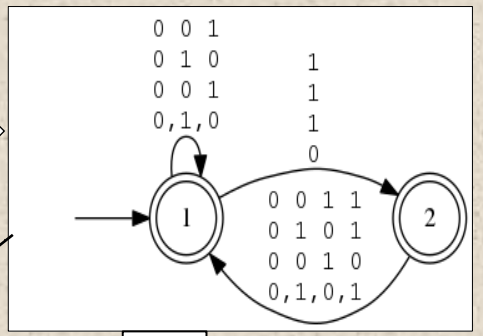
Monitor Automaton (DCValid)



MPNC (DCSynthG: Step 1)



LODC (DCSynthG: Step 2)



Soft Requirement = [r1 =>a1 , a2]

SCADE Encoding (DCSynthG: Step 3)

```
node Arb2Cell (r1,r2:bool)
returns (a1,a2:bool)
 var st: int;
let
 a1,a2,st =
 if pre st = 1 and not r1 and
not r2 then ( F,F,1)
 else if pre st = 1 and not r1
and r2 then ( F,T,1)
……..
……..
else if pre st = 2 and r1 and
r2 then ( F,T,1)
 else ( F,F,pre st) ;
tel
```

Example: AAS

- Addresses the indication status of actuators like lamps and hooters based on inputs signal status
- Specification (SRS of an I&C System): Can be provided as timing diagram

Interface:

  - Inputs: signal status (Normal/Alarm), ack & reset
  - Outputs: lamp(Off/steady/slow flash/fast flash), hooter

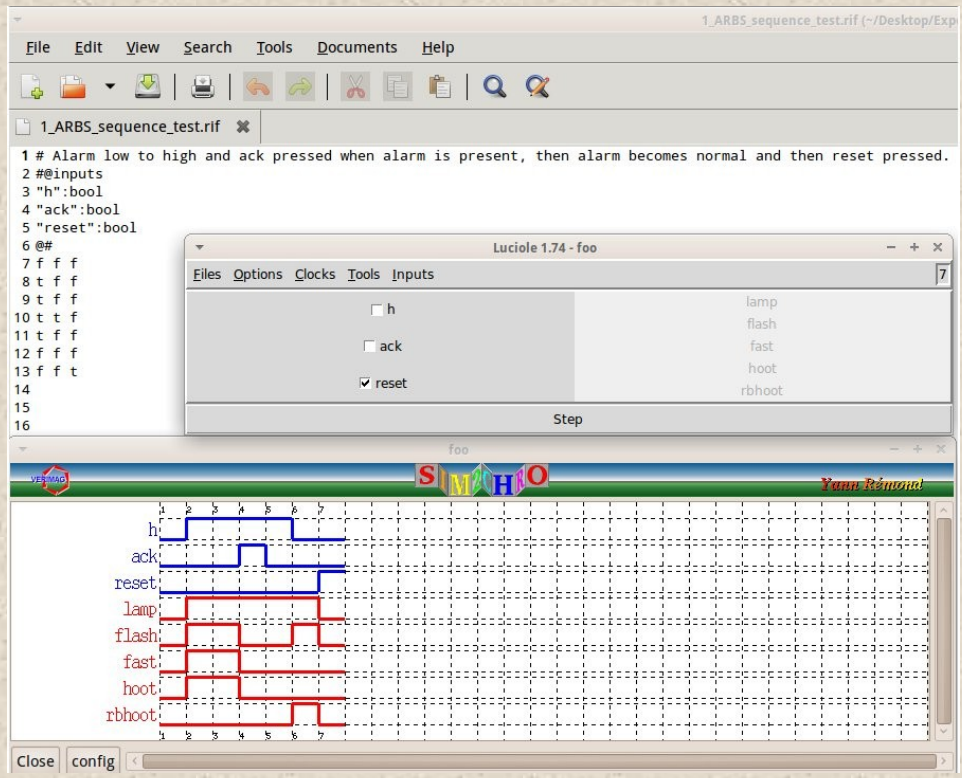
Functional:

  - signal goes to alarm, then fast flash and normal hooter ON
  - alarm acked, make lamp steady and normal hooter OFF
  - signal goes to normal, then slow flash and RB hooter ON
  - When reset, then lamp OFF and RB hooter is OFF

AAS

```
-- st1,st2;
-- lamp flash fast group has states {off, fastflash, steady, slowflash}
-- hoot,rbhoot has two states {off, hoot, rbhoot}
-- st1,st2  := 00  normal 10 abnormal 01 acknowledged 11 unreset
var h,ack,reset,lamp,flash,fast,hoot,rbhoot;

define unless[P11,Q11] as
   !(([[!Q11]])^<!P11>^true);

define persist[P111,Q111] as
[](<P111>^ext => slen=1^unless[P111,Q111]);

define transit[P,Q,R] as
[](<P >^slen=1^<Q> => true^<R>);

infer
ex st1. ex st2.
<!st1 && !st2>^true &&  -- initial state
persist[!st1 && !st2, h] &&
transit[!st1 && !st2, h, (st1 && !st2) ]  &&

persist[st1 && !st2,(ack) ] &&
transit[st1 && !st2, ack && h,!st1 && st2]  &&

transit[st1 && !st2, ack && !h,st1 && st2]  &&

persist[!st1 && st2, !h] &&
transit[!st1 && st2, !h, st1 && st2] &&

persist[st1 && st2, !h && reset || h] &&
transit[st1 && st2, !h&&reset, !st1 && !st2] &&

transit[st1 && st2, h, st1 && !st2] &&
[[ !st1 && !st2 => !lamp && !hoot && !rbhoot ]] &&
[[ st1 && !st2 => lamp && flash && fast && hoot && !rbhoot]] &&
[[ !st1 && st2 => lamp && !flash && !hoot && !rbhoot]] &&
[[ st1 && st2 => lamp && flash && !fast && !hoot && rbhoot]] && true.
```

Change of state	OUTPUT	
At INPUT	**Lamp**	**Audio**
Normal to Alarm	Fast Flash	Normal Alarm Hooter On
Acknowledged	**Steady**	**Normal Alarm Hooter Off**
Alarm to Normal	Slow Flash	Ring Back Hooter On
Reset	**Off**	**Ring Back Hooter Off**



Comparison With Other Tools

Problem	Lily		AcaciaPlus		DCSynthG	
	Time (sec)	States	Time (sec)	States	Time (sec)	States
Arb_Bounded_Resp_4Cell	161.9	108	0.4	55	0.09	50
Arb_Bounded_Resp_5Cell	TO	-	11.4	293	4.8	432
Arb_Bounded_Resp_6Cell	TO	-	TO	-	80	4802
Arbiter_token_8Cell	TO	-	46.4	73	1.9	8
Arbiter_token_10Cell	-	-	NC	-	137	10
Arbiter_token_12Cell	-	-	NC	-	10318	12
Arbiter_GR1_6Cell	TO	-	1153	1131	NE	-
Minepump_Latency	TO	-	NC	-	0.06	32
Minepump_Soft_PumpOff	NE	-	NE	-	0.06	87
Minepump_Soft_MethanSafe	NE	-	NE	-	0.13	43

TO = Timeout, MO = Memory Out, NE = Not expressible and NC = Inconclusive

Possible Extension

- Reactive synthesis:Extending the theory of soft requirement to add *globally optimal* synthesis (e.g for mean pay-off objective)
- Algorithmic extension to add robust synthesis with *Don’t be lazy* and *Never give up goals*.