Guided Reactive Synthesis with Soft Requirements

Amol Wakankar[1],Paritosh Pandya[2], Raj Mohan Matteplackel[2]

[1]Bhabha Atomic Research Centre, Mumbai
Homi Bhabha National Institute, Mumbai
[2]Tata Institute of Fundamental Research Centre, Mumbai

Introduction

- DCSynth: A tool to synthesize a controller, that meets temporal specification of reactive system

Introduction

- DCSynth: A tool to synthesize a controller, that meets temporal specification of reactive system
- Problems:
  - Systems are Underspecified (Many possible controllers)
  - May have Conflicting requirements (No controllers)
- In this talk Synthesis under soft requirements:
  - To choose between controller
  - Resolve conflicts.

QDDC Formulae as Requirements

- Highly expressive and succinct Interval Temporal logic to specify bounded liveness properties of embedded system

  []([[req]] && slen = 10) => (scount ack < 3))

  All behaviours, where in any observation interval if request is true continuously for 10 cycles then there are atleast 3 acknowledgements.

- Focus on past time bounded liveness properties over D.

  s.t. for all $\sigma \in VAL+$,

  $\sigma,i \models D$ iff $\sigma[0:i] \in L(D)$.

QDDC Formulae as Requirements

- Highly expressive and succinct Interval Temporal logic to specify bounded liveness properties of embedded system

  []([[req]] && slen = 10) => (scount ack < 3))

  All behaviours, where in any observation interval if request is true continuously for 10 cycles then there are atleast 3 acknowledgements.

- Focus on past time bounded liveness properties over D.

  s.t. for all $\sigma \in VAL+$,

  $\sigma,i \models D$ iff $\sigma[0:i] \in L(D)$.

- Naturally and compositionally represents requirements (SEFM-2017)

Requirements in DCSynth

- A requirement is a past time bounded liveness property
- We Partition the Requirement(Hard+Soft)
  - Hard Requirements have to be met invariantly.
  - Soft Requirements hold transiently.

Soft specification is a prioritised list of requirements which have to be met as much as possible (robustness).

Requirements in DCSynth

- A requirement is a past time bounded liveness property
- We Partition the Requirement(Hard+Soft)
  - Hard Requirements have to be met invariantly.
  - Soft Requirements hold transiently.
    Soft specification is a prioritised list of requirements which have to be met as much as possible (robustness).
- Guarantees:
  - Analyze performance of a synthesized controller.
  - Based on user defined measure of performance and robustness.
  - Analysis using symbolic model checking techniques

N-Cell Synchronous Bus Arbiter

-- Specification for the arbiter with n requests and n acknowledgement lines.

req1, req2 .. req_n: INPUT; ack1, ack2 … ack_n: OUTPUT;

--**Spec1**: Exclusion. Atmost 1 acknowledgement can be given at a time.

-- **Spec2**: Noloss. If there is atleast one request, then one of them should be grated.

-- **Spec3**: NoSpuriousAck. Bus access should be granted, only if it was requested.

-- **Spec4:** Fairness. Access to the request should be granted withing bounded time

N-Cell Synchronous Bus Arbiter

-- Specification for the arbiter with n requests and n acknowledgement lines.

req1, req2 .. req_n: INPUT; ack1, ack2 … ack_n: OUTPUT;

--**Spec1**: Exclusion. Atmost 1 acknowledgement can be given at a time.

define Exclusion as

[[(ack1 => !(ack2.. ack_n)) && (ack2=> !(ack1.. ack_n)) .. (ack_n=> !(ack1.. ack_{n-1}))]];

-- **Spec2**: Noloss. If there is atleast one request, then one of them should be grated.

define Noloss as

[[(req1 || req2 .. req_n) => (ack1 || ack2 .. ack_n)]];

-- **Spec3**: NoSpuriousAck. Bus access should be granted, only if it was requested.

define NoSpuriousAck(req, ack) as

[[ack => req]];

-- **Spec4:** Fairness. Access to the request should be granted withing bounded time

define Response(req, ack, n) as

[] ([[req]] && slen=n-1 => <> <ack>);

N-Cell Synchronous Bus Arbiter

-- Specification for the arbiter with n requests and n acknowledgement lines.

req1, req2 .. req_n: INPUT; ack1, ack2 … ack_n: OUTPUT;

--**Spec1**: Exclusion. Atmost 1 acknowledgement can be given at a time.

define Exclusion as

[[(ack1 => !(ack2.. ack_n)) && (ack2=> !(ack1.. ack_n)) .. (ack_n=> !(ack1.. ack_{n-1}))]];

-- **Spec2**: Noloss. If there is atleast one request, then one of them should be grated.

define Noloss as

[[(req1 || req2 .. req_n) => (ack1 || ack2 .. ack_n)]];

-- **Spec3**: NoSpuriousAck. Bus access should be granted, only if it was requested.

define NoSpuriousAck(req, ack) as

[[ack => req]];

-- **Spec4:** Fairness. Access to the request should be granted withing bounded time

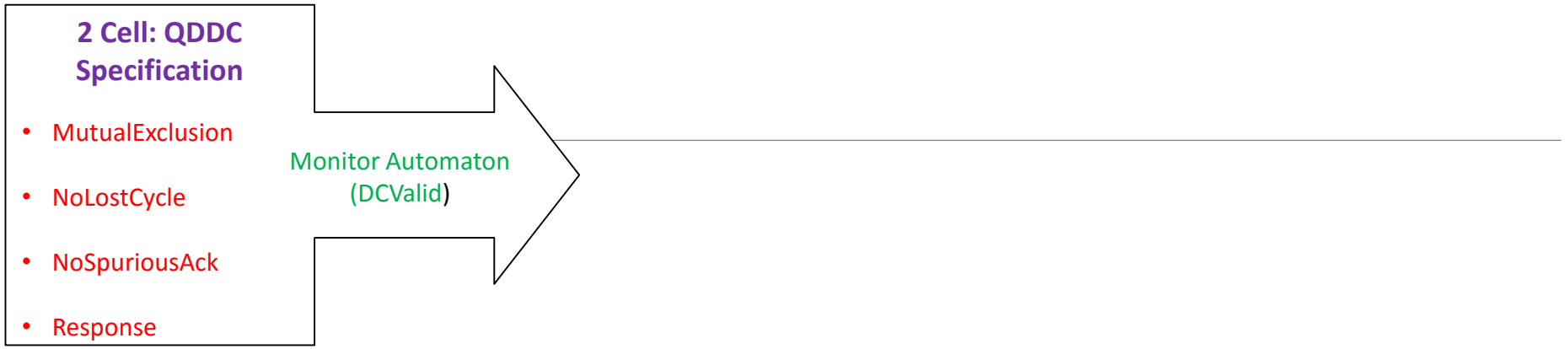
define Response(req, ack, n) as

[] ([[req]] && slen=n-1 => <> <ack>);

D^{hard}: Exclusion && Response[req_i,ack_i] && Noloss && NoSpuriousAck[req_i, ack_i]

D^{soft}: ack1>>ack2>>… >> ack_n

Synthesis Demonstration: 2 Cell

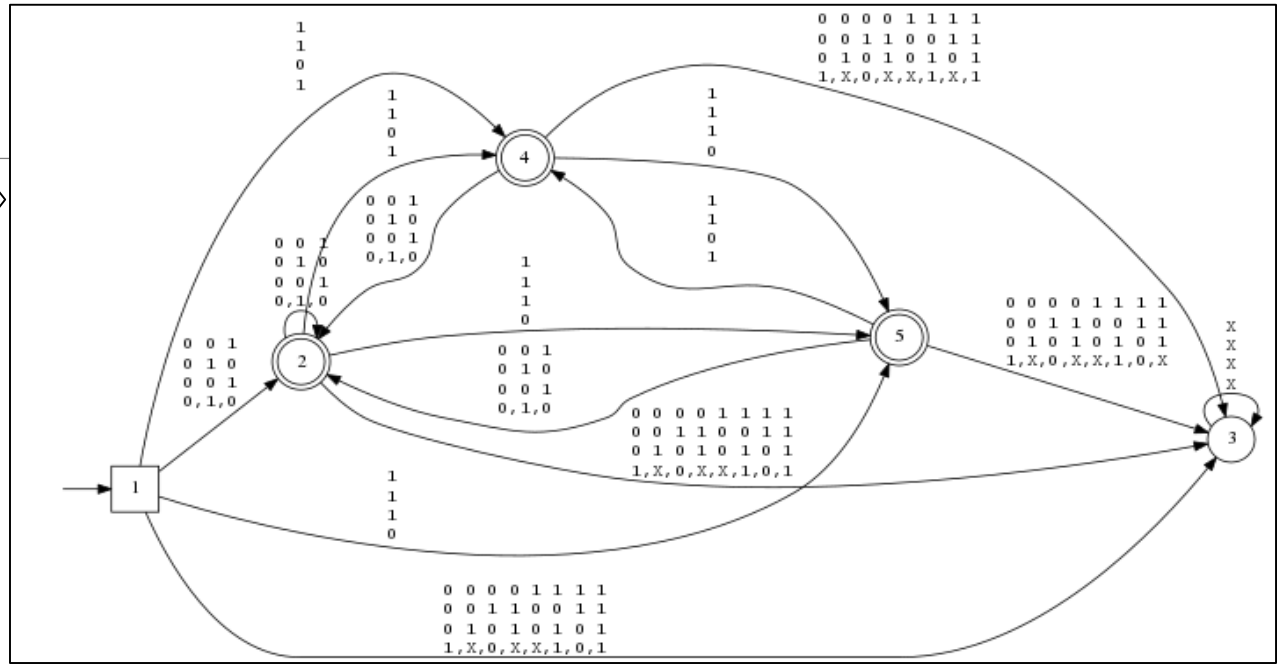


Synthesis Demonstration: 2 Cell

2 Cell: QDDC Specification

- MutualExclusion
- NoLostCycle
- NoSpuriousAck
- Response

Monitor Automaton (DCValid)



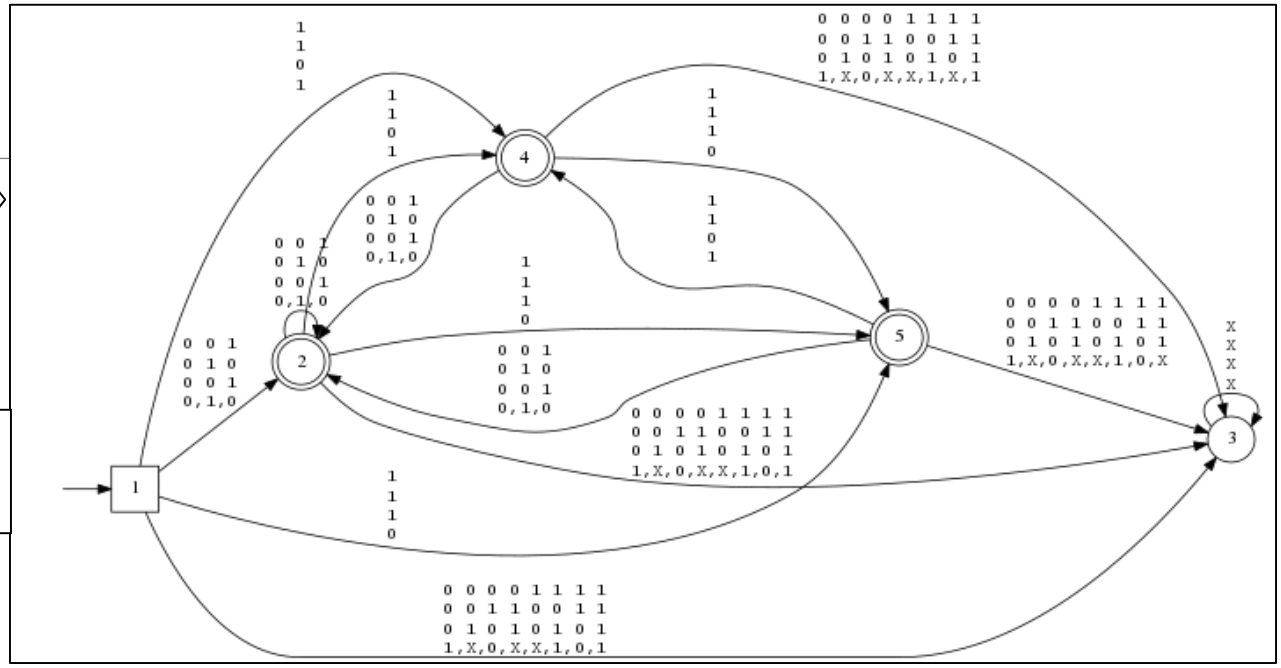
Synthesis Demonstration: 2 Cell

2 Cell: QDDC Specification

- MutualExclusion
- NoLostCycle
- NoSpuriousAck
- Response

Monitor Automaton (DCValid)

MPNC (DCSynth: Step 1)

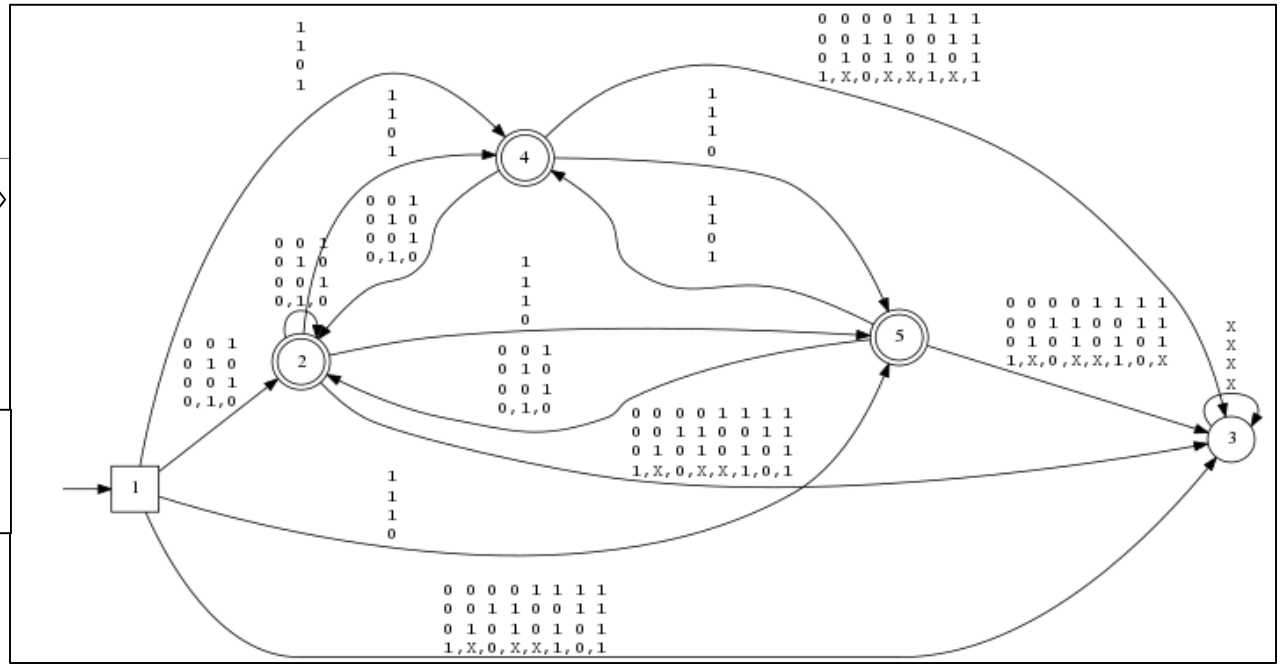


Synthesis Demonstration: 2 Cell

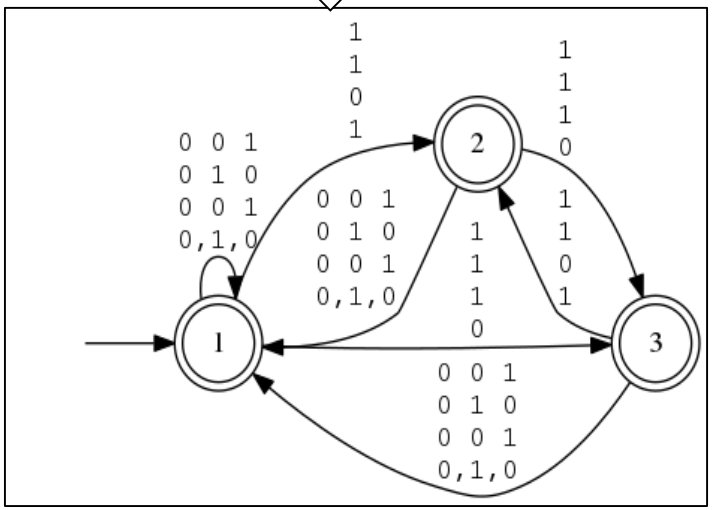
2 Cell: QDDC Specification

- MutualExclusion
- NoLostCycle
- NoSpuriousAck
- Response

Monitor Automaton (DCValid)



MPNC (DCSynth: Step 1)

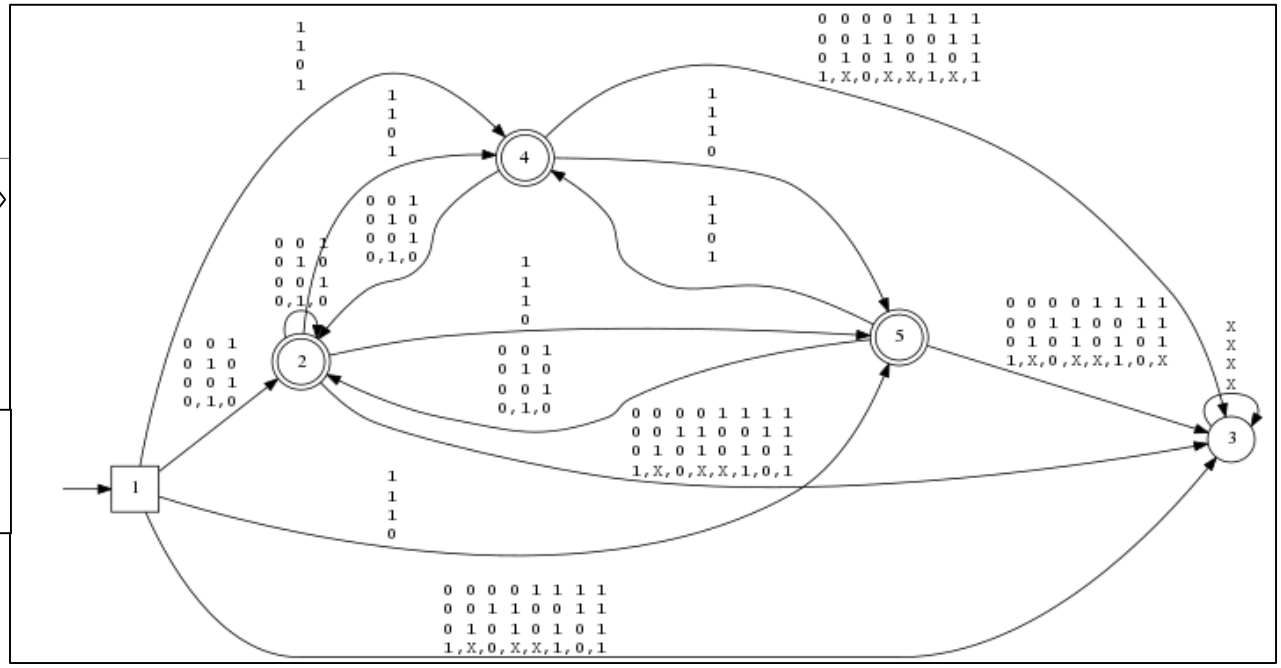


Synthesis Demonstration: 2 Cell

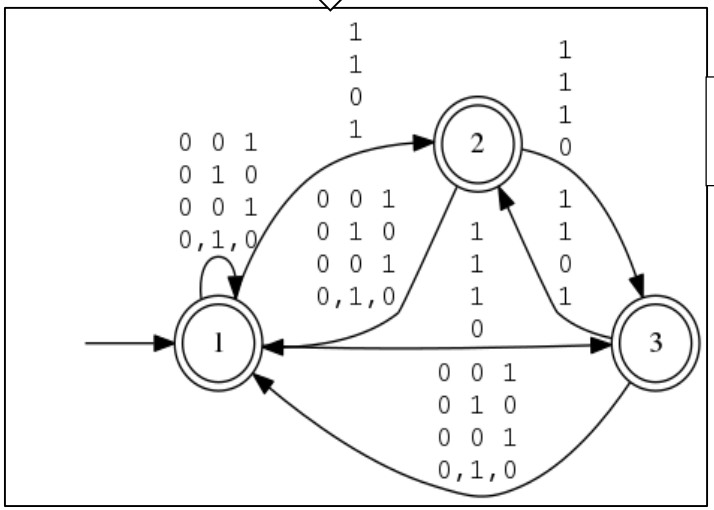
2 Cell: QDDC Specification

- MutualExclusion
- NoLostCycle
- NoSpuriousAck
- Response

Monitor Automaton (DCValid)



MPNC (DCSynth: Step 1)



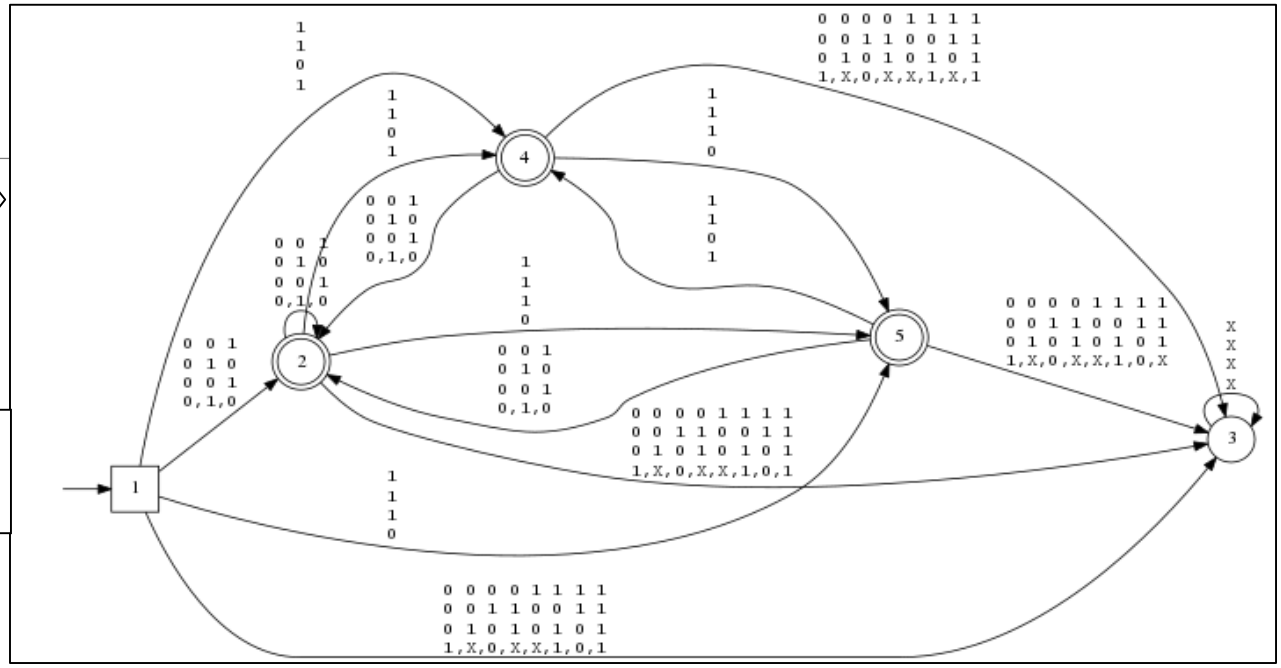
LODC (DCSynth: Step 2)

Synthesis Demonstration: 2 Cell

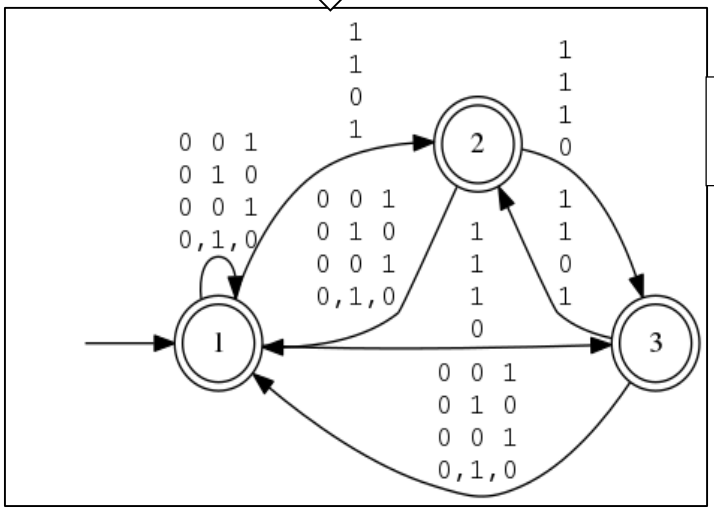
2 Cell: QDDC Specification

- MutualExclusion
- NoLostCycle
- NoSpuriousAck
- Response

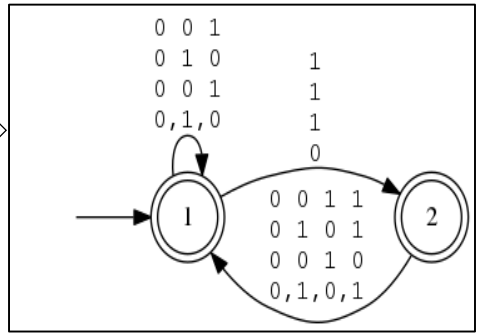
Monitor Automaton (DCValid)



MPNC (DCSynth: Step 1)



LODC (DCSynth: Step 2)

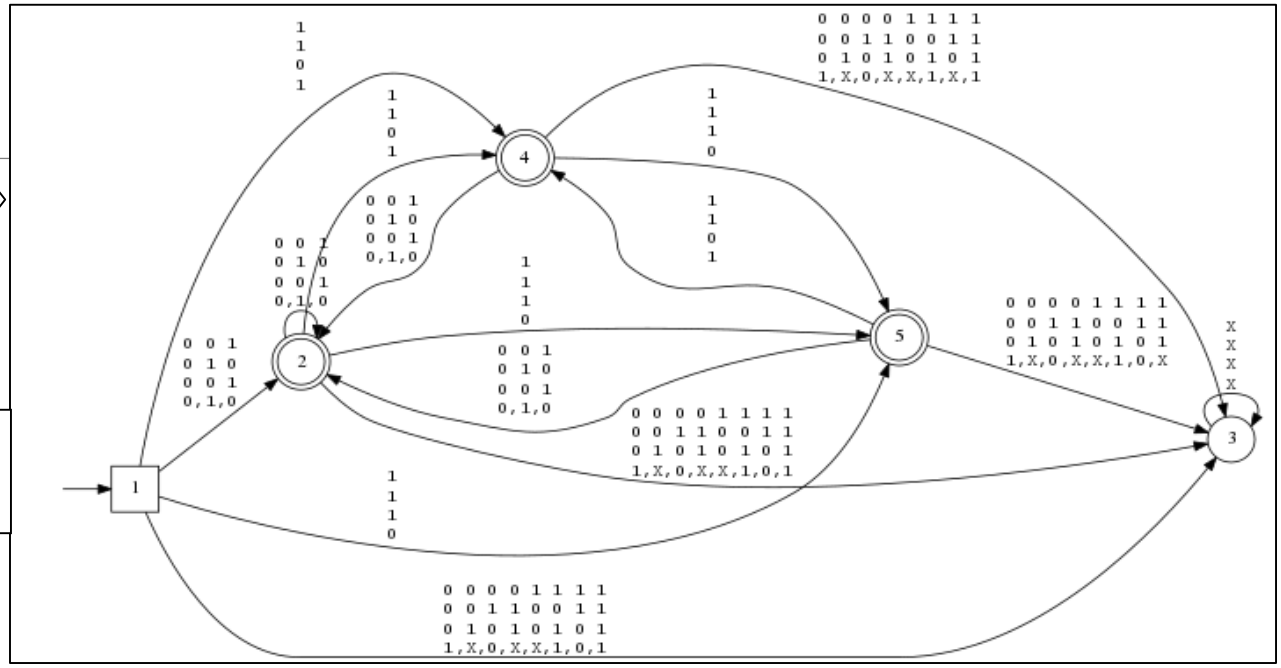


Synthesis Demonstration: 2 Cell

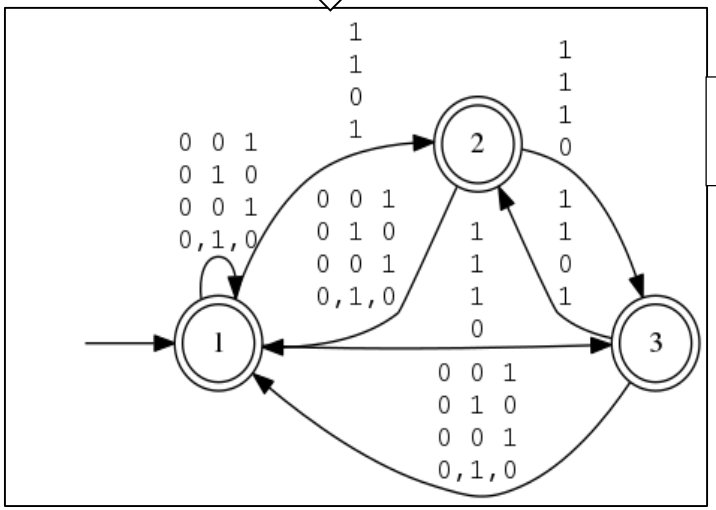
2 Cell: QDDC Specification

- MutualExclusion
- NoLostCycle
- NoSpuriousAck
- Response

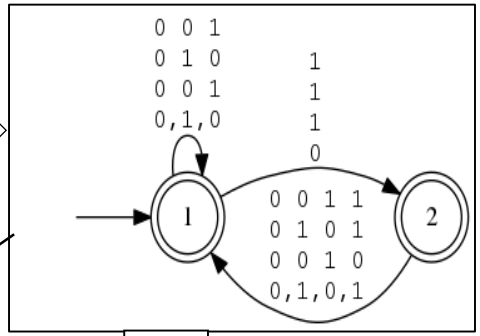
Monitor Automaton (DCValid)



MPNC (DCSynth: Step 1)



LODC (DCSynth: Step 2)



Soft Requirement

ack1 >> ack2

SCADE Encoding (DCSynth: 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
```

Controller Synthesis Steps

- Generate a DFA for a DCSynth specification S, including witness variables w_i
- Compute Maximally Permissive Non-deterministic Controller for D^{hard} with standard safety synthesis.
- Prune output non-determinism based on Soft Requirements, s.t. the controller satisfies maximal set of soft requirements at every point. It gives Locally Optimal Deterministic Controller.
- Encode LODC in required language (SCADE/SMV/Verilog)
- Symbolic Algorithm (*Without automaton spreading*)

Soft Requirements and Performance

- Requirement: Synthesize an 6 Cell Arbiter with 2 Cycle Response time for each cell (NOT REALIZABLE)

Soft Requirements and Performance

- Requirement: Synthesize an 6 Cell Arbiter with 2 Cycle Response time for each cell (NOT REALIZABLE)
- Use Soft Requirements:
  - Make Response Requirement for each cell as soft requirements.
  - Soft Reqs: (sr6)>>(sr5)>>(sr4)>>(sr3)>>(sr2)>>(sr1)

    Where, sr_i : (response(req_i, ack_i, 2)

Soft Requirements and Performance

- Requirement: Synthesize an 6 Cell Arbiter with 2 Cycle Response time for each cell (NOT REALIZABLE)
- Use Soft Requirements:
  - Make Response Requirement for each cell as soft requirements.
  - Soft Reqs: (sr6)>>(sr5)>>(sr4)>>(sr3)>>(sr2)>>(sr1)
    Where, sr_i : (response(req_i, ack_i, 2)
- Performance Measurement:
  - Maximum time Request 'i' should be kept on the get an acknowledgment. ([[req6]] && [[!ack6]])
  - Results (Guarantee):
    req6: 2 cycle,
    req5: 3 cycle,
    req1 to req4: Infinity

Case Studies

- Algorithm produces implementation as SCADE/SMV/C program which Realizes the given specification.
- Case Studies:

  Industrial

  Academic

Case Studies

- Algorithm produces implementation as SCADE/SMV/C program which Realizes the given specification.
- Case Studies:

  Industrial
  - Nuclear Reactor Controller for Pump/Valve (Tabular description)
  - Alarm Annunciation Logic for I&C system of Reactor
  - Discordance logic for a research reactor
  - Synchronous Bus Arbiter (With soft requirements)
  - AMBA bus Arbiter

  Academic

Case Studies

- Algorithm produces implementation as SCADE/SMV/C program which Realizes the given specification.
- Case Studies:

  Industrial
  - Nuclear Reactor Controller for Pump/Valve (Tabular description)
  - Alarm Annunciation Logic for I&C system of Reactor
  - Discordance logic for a research reactor
  - Synchronous Bus Arbiter (With soft requirements)
  - AMBA bus Arbiter

  Academic
  - Minepump (With soft requirements)
  - Pump On/Off example.
  - Traffic Light

Nuclear Reactor Controller for Pump/Valve

- **Control Elements and Modes of operation**

Valves

- Valve can have two positions: ***OPEN***, ***CLOSE***
- Control Logic generates commands: ***OPEN*** and **CLOSE** command

Pumps

- Any Pump can have two states: ***RUNNING***, ***STOPPED***
- Control Logic generates commands : ***START*** and ***STOP*** command

Modes of operation

- **Auto** : Equipment gets controlled through specified control logic
- **Manual** : Controlled through operator command (Priority over Auto).

Nuclear Reactor Controller for Pump/Valve

❑ Control Elements and Modes of operation

Valves

- ❑ Valve can have two positions: ***OPEN***, ***CLOSE***
- ❑ Control Logic generates commands: ***OPEN*** and **CLOSE** command

Pumps

- ❑ Any Pump can have two states: ***RUNNING***, ***STOPPED***
- ❑ Control Logic generates commands : ***START*** and ***STOP*** command

Modes of operation

- ❑ **Auto** : Equipment gets controlled through specified control logic
- ❑ **Manual** : Controlled through operator command (Priority over Auto).

❑ Elements to be controlled

- ❑ Identifier of valve being controlled- V1, V3
- ❑ Identifier of pump being controlled- P1
- ❑ This process system has 5 states – OFF, OFFH, ONH, START, NORMAL
- ❑ Control logic uses two digital signals – C16, C17

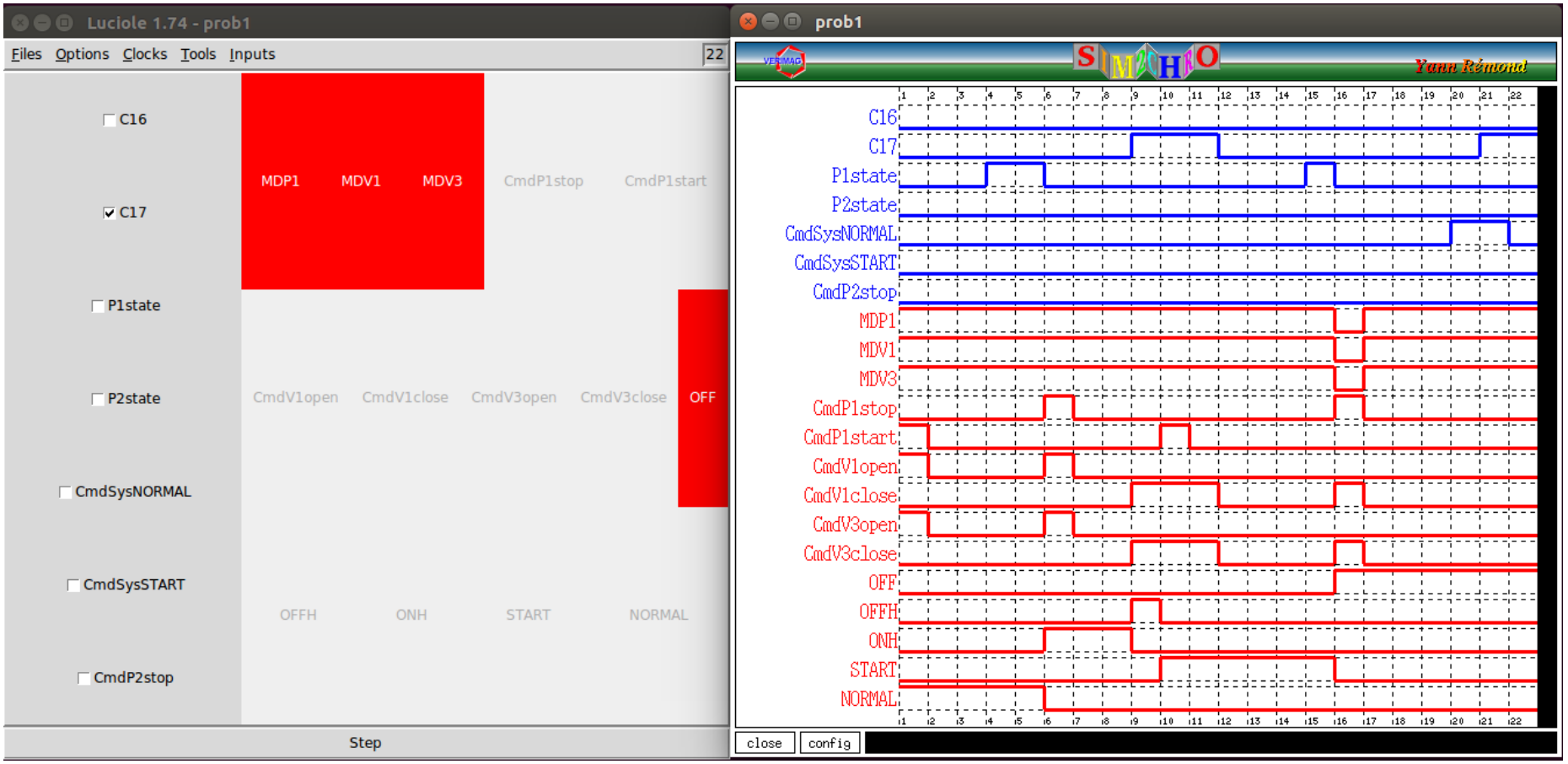
Nuclear Reactor Controller for Pump/Valve: Requirements

State	Control action on Entry	Control logic specified for the state
ONH	1. OPEN command for V1, V3 2. Valves V1, V3 are set to AUTO mode 3. STOP command for P1 4. Pump P1 is set to AUTO mode, If transfer to this state was not made on self stopping of Pump P1	1. Change state to NORMAL if Pump P2 self stops. (only if Pump P1 is set in AUTO mode) 2. Change state to OFFH if C17 is set to TRUE. This change of state shall be disabled, if C16 is also set to TRUE along with C17. <u>Constraints</u> Changing of state from ONH to START shall be disabled. When such command is issued, state shall be changed to NORMAL
NORMAL	1. START command for P1 2. OPEN command for V1, V3 3. Valves V1, V3 are set to AUTO mode (only when change to this state happens from OFF)	1. Change state to ONH if P1 self stops 2. Change state to OFFH, when signal C17 is set to TRUE. This change of state shall be disabled, if C16 is also set to TRUE along with C17.
OFF **OFFH** **START**	1. ……	1. ……

Ambiguities Found during Industrial Use case

- Good Requirement Specification: (IEC 60880)
  - A.2.3.2: SRS shall be free from contradiction and without duplication….
  - A.2.3.3: SRS shall be complete and consistent….
- Incomplete requirements and transition relation
- Conflicting Requirements
- The priority of transitions
- Strong vs Weak transitions
- The specification of default values
- Automatic Prototyping

Prototyping and Results



- No. of states in original aut Vs No. of states in MPNC aut = 20 /18
- MPNC construction time : 0.527963 sec and Memory: 7032 bytes
- No. of states in MPNC aut Vs No. of states in LODC aut = 18 /13
- LODC construction time : 5.540246 sec and Memory: 64040 bytes

Thank you

Backup Slides

QDDC Syntax and Semantics

- ***Syntax:***

D := <ϕ> | [ϕ] | [[ϕ]] | {{ϕ}} | D ^ D | ¬D | D ∨ D | D ∧ D | ∃p.D | ∀p.D | slen ⌂ c | scount ϕ ⌂ c.

where $\phi \in \Omega_\Sigma$, $p \in \Sigma$, $c \in N$ and $\text{⌂} \in \{<, \leq, =, \geq, >\}$

- ***Semantics:*** *Let σ be a word over Σ and let [b, e] ∈ Intv(N)*
- σ, [b, e] |= <ϕ> iff $\phi \in \Omega_\Sigma$, b = e, and σ, b |= ϕ,
- σ, [b, e] |= [ϕ] iff $\phi \in \Omega_\Sigma$ and ∀b ≤ i < e : σ, i |= ϕ,
- σ, [b, e] |= [[ϕ]] iff $\phi \in \Omega_\Sigma$ and ∀b ≤ i ≤ e : σ, i |= ϕ,
- σ, [b, e] |= {{ϕ}} iff $\phi \in \Omega_\Sigma$, e = b + 1 and σ, b |= ϕ,
- σ, [b, e] |= ¬D iff σ, [b, e] |≠ D,
- σ, [b, e] |= D1 ∨ D2 iff σ, [b, e] |= D1 or σ, [b, e] |= D2,
- σ, [b, e] |= D1 ∧ D2 iff σ, [b, e] |= D1 and σ, [b, e] |= D2,
- σ, [b, e] |= D1^D2 iff ∃b ≤ i ≤ e : σ, [b, i] |= D1 and σ, [i, e] |= D2

Algorithm for Safety Specification

- $\phi_{safe} = \square(\Lambda_{i \in I1} D_i)$ 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 $\exists i. \forall o : \delta(s, (i, o)) \in (S-G)$
- $V : Q \rightarrow \{1, 0\}$, 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):

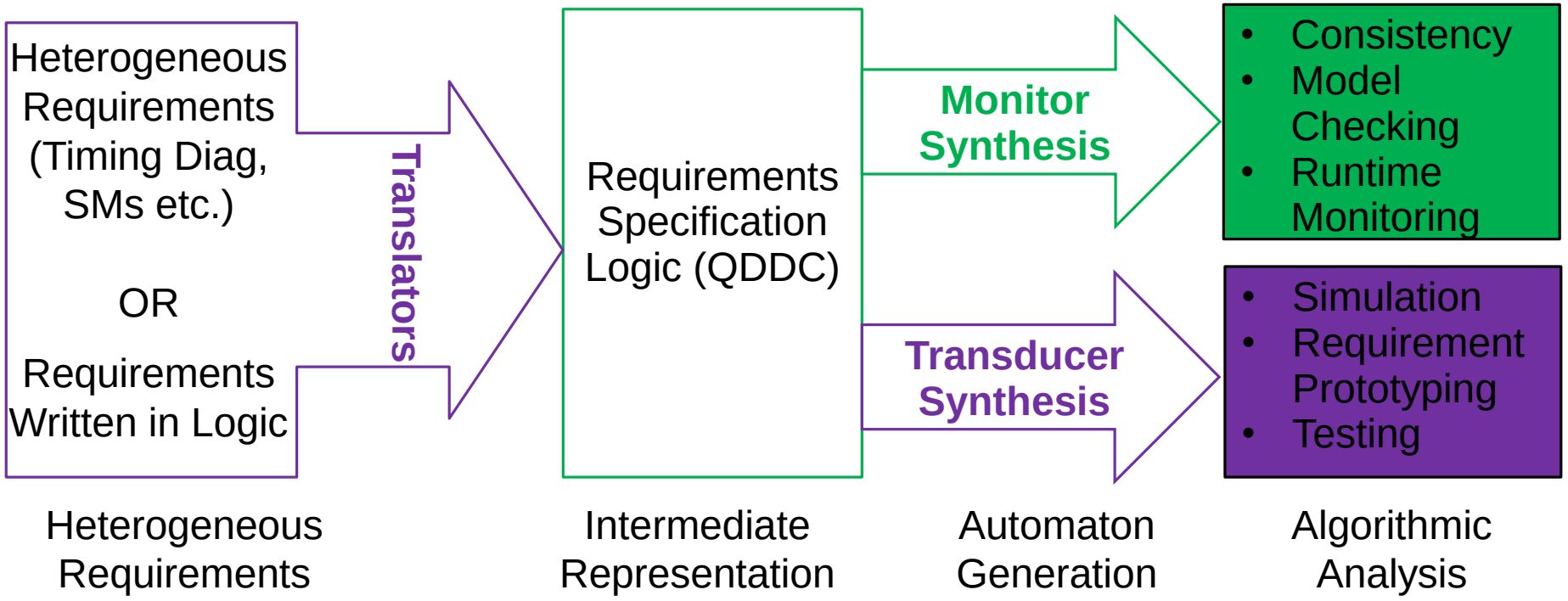
Input: A(D), G, Input-output partitioning
Output: Reach(A(D), G)
Set $V'(s) = 1, \forall s \in G$ and $V'(s) = 0, \forall s \in (S-G)$
Set $G_{reach} = G$
Do
    for each $s \in G_{reach}$ do
        If $Cstep(s, G_{reach}) = 0$ then
            $V'(s) = 0$
            $G_{reach} = G_{reach} - s$
While (*previous $V' \neq V'$ i.e. we reach a fix-point*)
Return (G_{reach})

Research Problems

- Requirement (Behavioural) Specification Language
  - High Level Language e.g. Timing diagram, SMs etc.
  - Formalization, Expressive Power, Synthesis Complexity….
- Complete Specification is difficult (Guided Synthesis)
  - May lead to Multiple correct implementation (IF REALIZABLE). Requires guiding the synthesis algorithm to produce best possible implementation.
  - Performance measurement to give guarantees
  - Explanation generation for unrealizable requirements.
- Architecture Centric analysis of non-performance attributes of Networked asynchronous control systems.
  - Dependability analysis of existing architecture.
- Monitor Synthesis (Error Correcting)
- Complexity of existing synthesis algorithms is very high
  - Compositional Synthesis
  - Domain specific optimization / Tools still impractical

Our Approach

- Use appropriate logic (QDDC) which is form of interval temporal logic, as formal representation for heterogeneous requirements.
- Translate heterogeneous requirements to QDDC formula Φ & analyze Φ for:
  - ❑ Satisfiability: Are the requirements consistent? If no, refine them.
  - ❑ Realizability: Are they implementable? If yes then synthesize it.



Guided Reactive Synthesis with Soft Requirements

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

- Requirements:
  - 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.

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.

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

Soft Requirements / Robust Synthesis

- 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. (85 States)
- Minepump_Soft_PumpOn [PumpOn >> !Alarm] : Implementation that switch on the pump as soon as possible. (28 States)
- Minepump_Soft_MethanSafe [(CH4_2Cyc => !PumpOn) >> PumpOn)] : Implementation that tries to keep pump off if there is a methane leak in last 2 cycles otherwise switch on the pump. (29 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.

Performance Measurement (Guarantees)

Sr.No	Example	Response Formula	Response
1	Arb_soft(6,2)	([[req6]] && ([[!ack6]]))	2
		[[req6]] && ((scount ack6 < 3))	6
2	Arb_soft(6,2)	([[req5]] && ([[!ack5]]))	3
		[[req5]] && ((scount ack5 < 3))	9
3	Arb_soft(6,2) for 1<=i<=4	([[req_i]] && ([[! ack_i]]))	Infinity
		[[req_i]] && ((scount ack_i < 3))	Infinity
4	Arb_soft(6,2)	([[req4 && !req6]] && ([[!ack4]]))	2
5	Arb_soft(5,3)	([[req5]] && ([[!ack5]]))	3
		[[req5]] && ((scount ack5 < 3))	14
6	Arb_soft(5,3)	([[req4]] && ([[!ack4]]))	4
		[[req4]] && ((scount ack4 < 3))	11
7	Arb_soft(5,3)	([[req3]] && ([[!ack3]]))	5
		[[req3]] && ((scount ack3 < 3))	8
8	MP_V1	[[AssumptionOk && HH2O]]	4
9	MP_V2	[[AssumptionOk && HH2O]]	6
10	MP_V3	[[AssumptionOk && HH2O]]	8

Algorithms for LODC Optimization

Problem	Soft Requirements	Without Optimization		With Optimization	
		time (Sec)	Memory (KB)	time (Sec)	Memory (KB)
$Arb^{hard}(4,4)$	ack4 >> ... >> ack1	0.023	3.4	0.014	3.3
$Arb^{hard}(5,5)$	ack5 >> ... >>ack1	0.57	24.7	0.33	22.4
$Arb^{hard}(5,5)$	ack1 >> ... >> ack5	0.54	25.0	0.30	22.4
$Arb^{hard}(6,6)$	ack6 >> ... >> ack1	24.4	488.8	14.8	334.5
MP_V1	PumpOn>> Alarm	0.0083	2.0	0.0017	2.1
MP_V2	(CH4_{Last2Cyc} =>!PumpOn) >> PumpOn	0.0041	2.1	0.0026	2.2
MP_V3	!PumpOn >> !Alarm	0.0053	2.1	0.0025	2.2

Effect of Soft Requirements

Current Application

- Allows Synthesis of better performing controller
- More concise specification by specification of default behaviour in industrial case studies
- One of the most important application of soft requirements is in the synthesis of robust controller

New Applications

- Application in the area of Shield Synthesis

Major Achievements

- Identified a logic (QDDC) that can be used to formalize the requirements given in heterogeneous formalisms.
  - ❑ Timing diagrams requirements are formalized in this logic. Paper accepted in International Conference on Software Engineering and Formal Methods (SEFM-2017)
- Algorithm design & implementation for automatic reactive synthesis from requirements in QDDC (safety subset)
  - ❑ Soft requirement based guiding of synthesis algorithm (LODC), with optimization strategy for LODC generation.
  - ❑ Robust synthesis with never give up strategy based on soft requirements.
  - ❑ Performance Measurement shows the effect of soft requirements
- Architecture centric analysis of non-performance properties(dependability) for Networked asynchronous control systems.

Plan of Work

- Extending the work on requirement formalization to *state-chart based requirements*.
- Reactive synthesis:
  - Extending the theory of soft requirement to add *Shield synthesis*
  - Algorithmic extension to add effective *counter example generation* for unrealizable requirements.
  - *Runtime monitoring* using aspect oriented programming
- Working on Following two paper submissions:
  - Guided Reactive Synthesis with Soft Requirements and Performance Measurement
  - Architecture Centric Dependability Analysis of Networked asynchronous control System

References

[1] Nina Amla, E. Allen Emerson, Robert P. Kurshan, and Kedar S. Namjoshi. Model checking synchronous timing diagrams. In Warren A. Hunt Jr. and Steven D. Johnson, editors, *Formal Methods in Computer-Aided Design, Third International Conference, FMCAD 2000, Austin, Texas, USA, November 1-3, 2000, Proceedings*, volume 1954 of *Lecture Notes in Computer Science*, pages 283–298. Springer, 2000.

[2] Bowen Alpern and Fred B. Schneider. Recognizing safety and liveness. *Distributed Computing*, 2(3):117–126, 1987.

[3] Aaron Bohy, Veronique Bruyere, Emmanuel Filiot, Naiyong Jin, and Jean-Fran¸cois Raskin. Acacia+, a tool for ltl synthesis. In *International Conference on Computer Aided Verification*, pages 652–657. Springer, 2012.

[4] Rastislav Bodik and Barbara Jobstmann. Algorithmic program synthesis: introduction. *International Journal on Software Tools for Technology Transfer*, 15(5-6):397–411, 2013.

[5] Hana Chockler and Kathi Fisler. Temporal modalities for concisely capturing timing diagrams. In Dominique Borrione and Wolfgang J. Paul, editors, *Correct Hardware Design and Verification Methods, 13th IFIP WG 10.5 Advanced Research Working Conference, CHARME 2005, Saarbr¨ucken, Germany, October 3-6, 2005, Proceedings*, volume 3725 of *Lecture Notes in Computer Science*,pages 176–190. Springer, 2005.

[6] Aliaksei Chapyzhenka and Jonah Probell. Wavedrom: Rendering beautiful waveforms from plain text. *Synopsys User Group*, 2016.

References

[6] Julien Delange and Peter Feiler. Architecture fault modeling with the aadl error-model annex. In 2014 40th EUROMICRO Conference on Software Engineering and Advanced Applications, pages 361–368. IEEE, 2014.
[7] Werner Damm and David Harel. Lscs: Breathing life into message sequence charts. Formal Methods in System Design, 19(1):45–80, 2001.
[8] Harel David Damm Werner. Lscs: Breathing life into message sequence charts. Formal Methods in System Design, 19(1):45–80, 2001.
[9] Cindy Eisner, Dana Fisman. Temporal logic made practical. Handbook of Model Checking. http://www.cis. upenn. edu/˜ fisman/documents/EFHBMC14. pdf, 2016
[10] E Allen Emerson and Joseph Y Halpern. “sometimes” & “not never” revisited: on branching vs linear time temporal logic. Journal of the ACM (JACM), 33(1):151–178, 1986
[11] Massimo Franceschet, Maarten de Rijke, and Bernd-Holger Schlingloff. Hybrid logics on linear structures: Expressivity and complexity. In 10th International Symposium on Temporal Representation and Reasoning / 4th International Conference on Temporal Logic (TIME-ICTL 2003), 8-10 July 2003, pages 166–173. IEEE Computer Society, 2003.
[12] Peter H Feiler, David P Gluch, and John J Hudak. The architecture analysis & design language (aadl): An introduction. Technical
report, DTIC Document, 2006.

References

[13] Kathi Fisler. Timing diagrams: Formalization and algorithmic verification. Journal of Logic, Language and Information, 8(3):323–361, 1999.
[14] Kathi Fisler. Two-dimensional regular expressions for compositional bus protocols. In Formal Methods in Computer-Aided Design, 7th International Conference, FMCAD 2007, Austin, Texas, USA, November 11-14, 2007, Proceedings, pages 154–157. IEEE Computer Society, 2007.
[15] Hans Hansson and Bengt Jonsson. A logic for reasoning about time and reliability. Formal aspects of computing, 6(5):512–535, 1994.
[16] David Harel and Michal Politi. Modeling Reactive Systems with Statecharts: The Statemate Approach. McGraw-Hill, Inc., New York, NY, USA, 1st edition, 1998.
[17] Barbara Jobstmann and Roderick Bloem. Optimizations for ltl synthesis. In 2006 Formal Methods in Computer Aided Design, pages 117–124. IEEE, 2006.
[18] Marta Kwiatkowska, Gethin Norman, and David Parker. Prism: Probabilistic symbolic model checker. In International Conference on Modelling Techniques and Tools for Computer Performance Evaluation, pages 200–204. Springer, 2002.
[19] Madhavan Mukund. Finite-state automata on infinite inputs. TCS,96:2, 1996.

References

[20] Paritosh K. Pandya. DCVALID user manual. Tata Institute of Fundamental Research, Mumbai, 1997.
[21] Paritosh K. Pandya. Specifying and deciding quantified discrete time duration calculus formulae using DCVALID. Technical report, Tata Institute of Fundamental Research, Mumbai, 2000.
[22] Paritosh K. Pandya. Model checking ctl*[dc]. In Tiziana Margaria and Wang Yi, editors, Tools and Algorithms for the Construction and Analysis of Systems, 7th International Conference, TACAS 2001 Proceedings, volume 2031 of Lecture Notes in Computer Science, pages 559–573. Springer, 2001.
[23] Paritosh K. Pandya. The saga of synchronous bus arbiter: On model checking quantitative timing properties of synchronous programs. Electr. Notes Theor. Comput. Sci., 65(5):110–124, 2002.
[24] WaveDrom. Wavedrom user manual. http://wavedrom.com/tutorial.html, 2016.
[25] Xiaomin Wei, Yunwei Dong, and Hong Ye. Qasten: Integrating quantitative verification with safety analysis for aadl model. In Theoretical Aspects of Software Engineering (TASE), 2015 International Symposium on, pages 103–110. IEEE, 2015.
[26] Wu, Meng and Zeng, Haibo and Wang, Chao. Synthesizing runtime enforcer of safety properties under burst error. NASA Formal Methods Symposium, 65–81, 2016. Springer.

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**