Flexible Compilation and Refinement of Asynchronous Circuits

Ebelechukwu Esimai and Marly Roncken

Asynchronous Research Center and Department of Computer Science
Portland State University, Portland, Oregon, USA

0

Takeaway

Goal: "Make it easy to insert asynchrony appropriate for each design part"

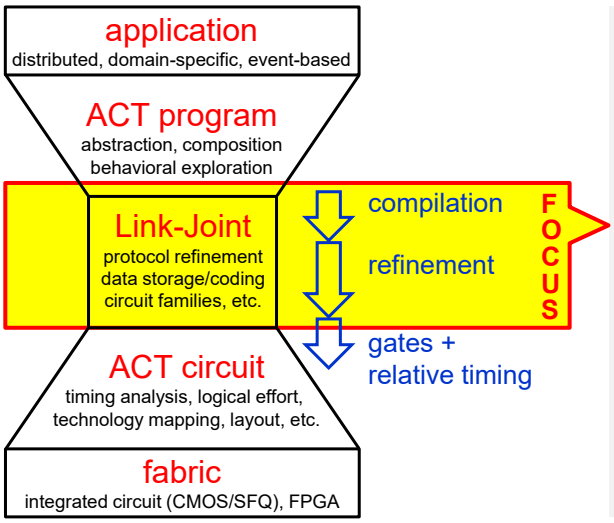
- Flexibility
  - Bind decisions as late as possible — to serve design and test
- This talk shows
  - per Link: **freedom of circuit family**
  - per Joint: **freedom of 2- or 4-phase protocol**
- Current support and w.i.p. includes:
  - 2- and 4-phase protocol, level- and pulse- and transition-logic, bundled and dual-rail data
  - Click, GasP, Set-Reset, Mousetrap, Micropipelines, Superconducting families

1

Outline

- Introduction
- Links and Joints — **flexible design and test**
- Flexible Compilation — **late binding**
- Flexible Refinement — **choose bindings**
- Mixed Protocols and Families — **easy mix and match**
- Conclusion

Introduction

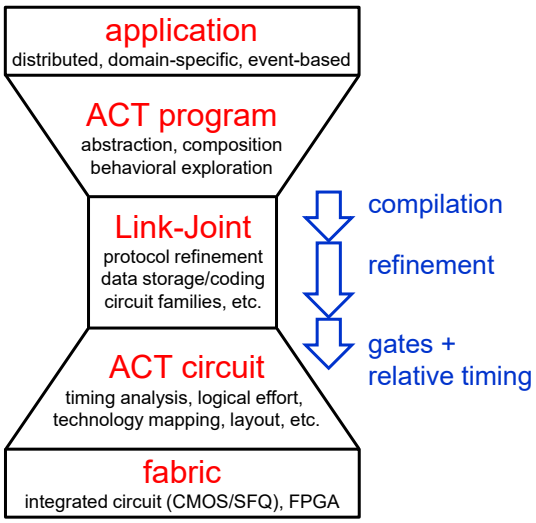


- Continuation of Link-Joint research
- Embedding into a design flow
  - Yale ACT (Asynchronous Circuit Toolkit)
  - shallow embedding — initially
- Link-Joint middle layer in the flow

  Compilation:
  - from algorithmic ACT programs
  - to circuit-neutral Link-Joint networks

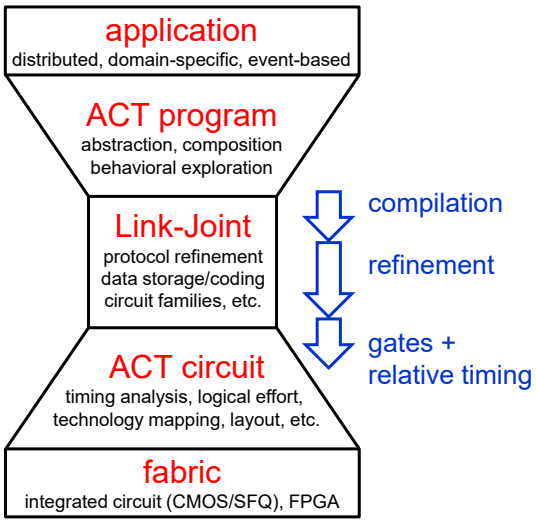
  Refinement (stepwise):
  - from Link-Joint networks
  - to ACT circuits

Introduction: why ACT?



- Asynchronous Circuit Toolkit
- Open source
- In active use
- Supports data and control flow
- Built with asynchronous expertise
- Incorporates proven ideas from
  - Philips and Handshake Solutions
  - University of Manchester, UK
  - Caltech

Introduction: Why a Link-Joint middle layer?



Benefit:

- circuit-neutral model
- embraces and combines
  - multiple protocols, data encodings
  - multiple circuit families and fabrics

Challenge:

- design-by-hand limits use and users

Solution:

- increase access by automation
- embed into middle of existing flow
  - re-use Link-Joint unrelated front and back parts
- maintain flexibility by using
  - circuit-neutral compilation
  - targeted refinements

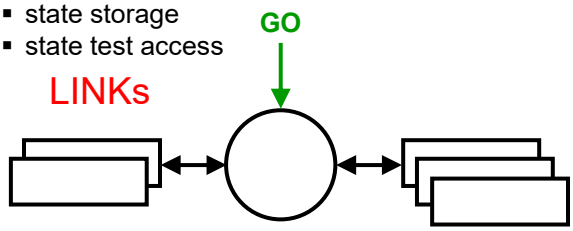
Outline

- Introduction
- Links and Joints — **flexible design and test**
- Flexible Compilation
- Flexible Refinement
- Mixed Protocols and Families
- Conclusion

Links and Joints

- communication
- state storage
- state test access

LINKs



JOINT

- computation
- flow control
- go-nogo test control

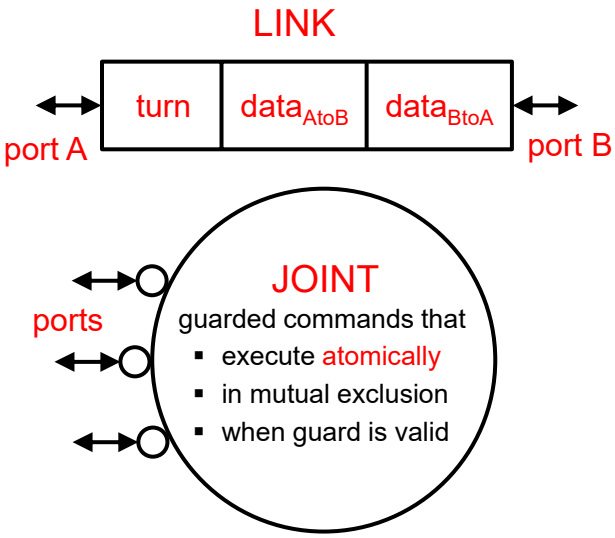
Link-Joint network:
- alternates Links and Joints

Link:
- shares and stores state
- connects two Joints

Joint:
- acts based on Link states
- changes states in (one or more) Links

Built-in initialization and test via:
- external access to Link states
- external *go*-control of Joint actions

Links and Joints: protocol and model



Protocol:
- follows good conversation practice
  - Joints take turns updating the Link state
  - Link tracks whose turn it is

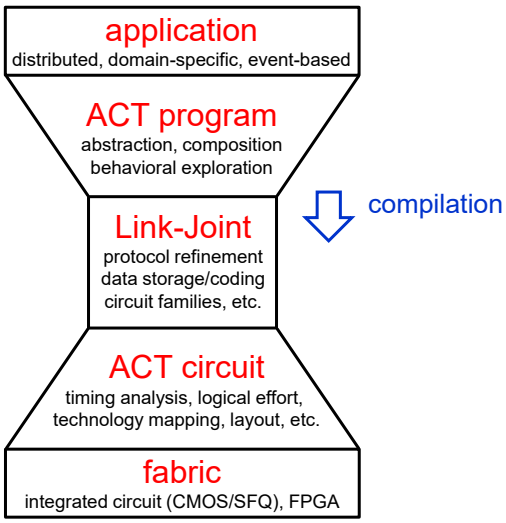
Link:
- has two ports to attach Joints: **A, B**
- has three state variables
  - **turn** points to A if A has the turn, else to B
  - **data$_{AtoB}$** stores ≥0 data bits from A to B
  - **data$_{BtoA}$** stores ≥0 data bits from B to A

Joint:
- Joint port connects to Link port A or B
- port must have turn to change Link state

Outline

- Introduction
- Links and Joints
- Flexible Compilation — **late binding**
- Flexible Refinement
- Mixed Protocols and Families
- Conclusion

Flexible Compilation



Strategy: syntax-directed translation

Source: ACT programs
- data-flow parts in ACT sub-language:
  - **dataflow**
- control-flow parts in ACT sub-language:
  - **C**ommunicating **H**ardware **P**rocesses

Target: circuit-neutral Link-Joint networks

Challenge:
- ***not*** **compiler**
  - like Philips, Manchester, Caltech, Yale
- ***but*** **library elements** used by the compiler
  - Link-Joint versions of channels + modules

Outline

- Introduction and Motivation
- Links and Joints
- Flexible Compilation: dataflow
- Flexible Refinement
- Mixed Protocols and Families
- Conclusion

Flexible Compilation: dataflow

ACT program:

```
defproc FIFO2_dataflow
(chan?(int) L ; chan!(int) R)
{
  chan(int) M ;
  dataflow { L → M ; M → R }
}
```

compiled Link-Joint network:

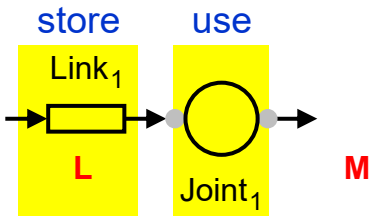
Compilation strategy
- syntax-directed
- store state before using state

Flexible Compilation: dataflow

ACT program:

```
defproc FIFO2_dataflow
(chan?(int) L ; chan!(int) R)
{
  chan(int) M ;
  dataflow { L → M ; M → R }
}
```

compiled Link-Joint network:



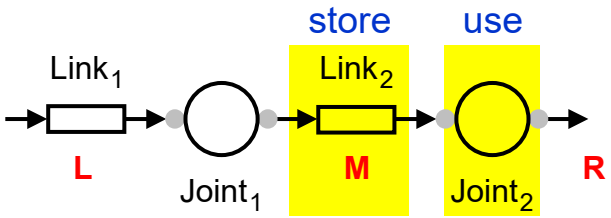
Compilation strategy
- syntax-directed
- store state before using state

Flexible Compilation: dataflow

ACT program:

```
defproc FIFO2_dataflow
(chan?(int) L ; chan!(int) R)
{
  chan(int) M ;
  dataflow { L → M ; M → R }
}
```

compiled Link-Joint network:



Compilation strategy

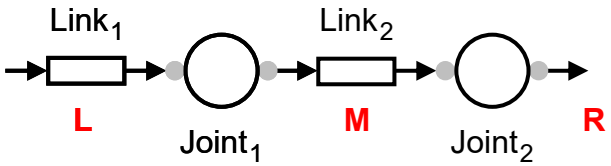
- syntax-directed
- store state before using state

Flexible Compilation: dataflow

ACT program:

```
defproc FIFO2_dataflow
(chan?(int) L ; chan!(int) R)
{
  chan(int) M ;
  dataflow { L → M ; M → R }
}
```

compiled Link-Joint network:



Compilation strategy

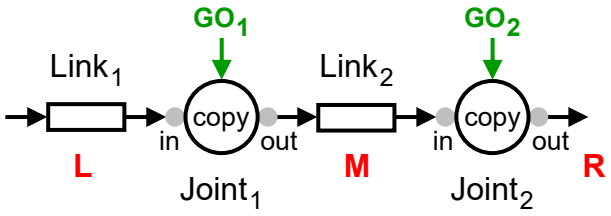
- syntax-directed
- store state before using state

Flexible Compilation: dataflow

ACT program:

defproc FIFO2_dataflow
(chan?(int) L ; chan!(int) R)
{
  chan(int) M ;
  dataflow { **L** → **M** ; **M** → **R** }
}

compiled Link-Joint network:



Compilation strategy

- syntax-directed
- store state before using state

Joint COPY

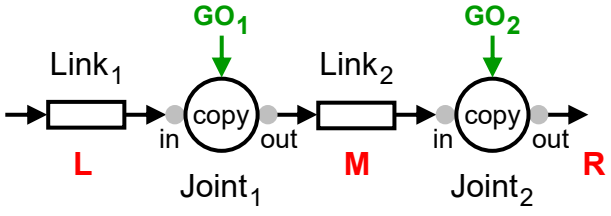
- circuit-neutral library element:
  - two ports: **in, out**
  - copies data from **in** to **out**
  - external *go*-control: **GO**

Flexible Compilation: dataflow

ACT program:

defproc FIFO2_dataflow
(chan?(int) L ; chan!(int) R)
{
  chan(int) M ;
  dataflow { **L** → **M** ; **M** → **R** }
}

compiled Link-Joint network:



Compilation strategy

- syntax-directed
- store state before using state

Joint COPY

- circuit-neutral library element:
  - two ports: **in, out**
  - copies data from **in** to **out**
  - external *go*-control: **GO**

- guarded command specification:
  myturn(in) ∧ *myturn(out)* ∧ ***GO***
  →
  myW(out) := myR(in)
  yourturn(in)
  yourturn(out)

Outline

- Introduction
- Links and Joints
- Flexible Compilation: CHP
- Flexible Refinement
- Mixed Protocols and Families
- Conclusion

Flexible Compilation: CHP

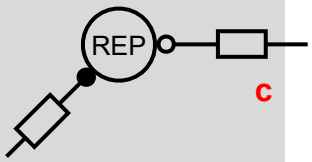
```
defproc onebuf
(chan?(int) L; chan!(int) R)
{
  int x ;
  chp { *[ L?x ; R!x ] }
}
```

onebuf

Flexible Compilation: CHP

```
defproc onebuf
(chan?(int) L; chan!(int) R)
{
  int x ;
  chp { *[ L?x ; R!x ] }
}
```

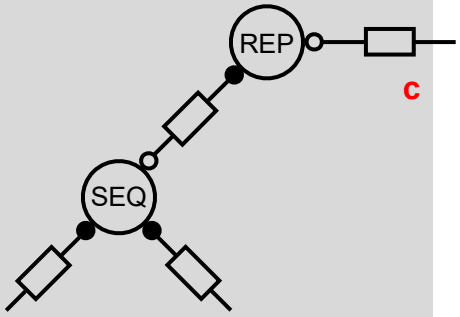
onebuf



Flexible Compilation: CHP

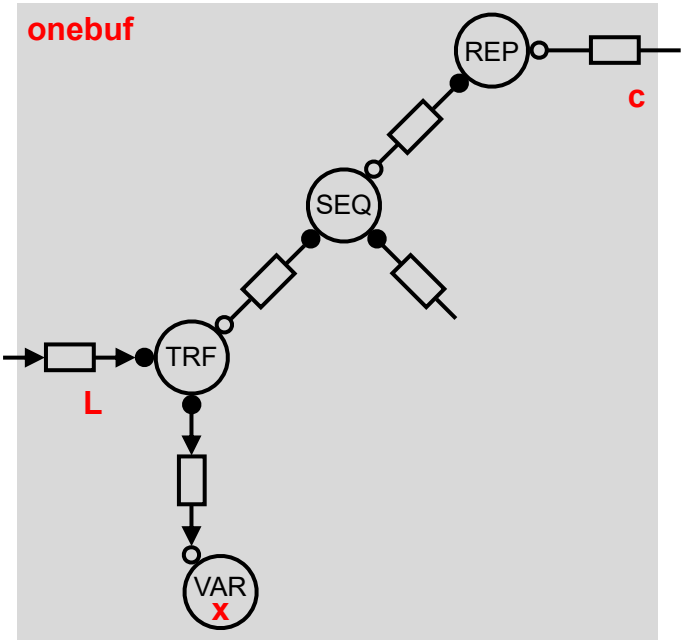
```
defproc onebuf
(chan?(int) L; chan!(int) R)
{
  int x ;
  chp { *[ L?x ; R!x ] }
}
```

onebuf



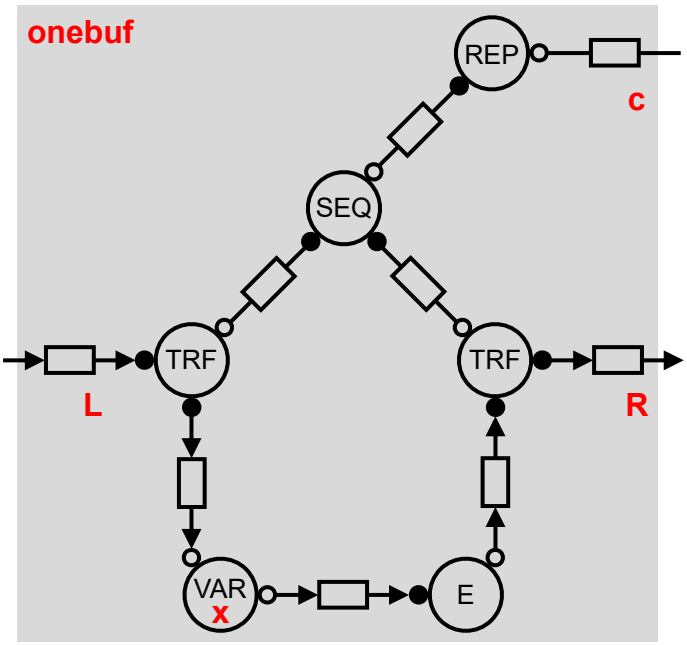
Flexible Compilation: CHP

```
defproc onebuf
(chan?(int) L; chan!(int) R)
{
  int x ;
  chp { *[ L?x ; R!x ] }
}
```



Flexible Compilation: CHP

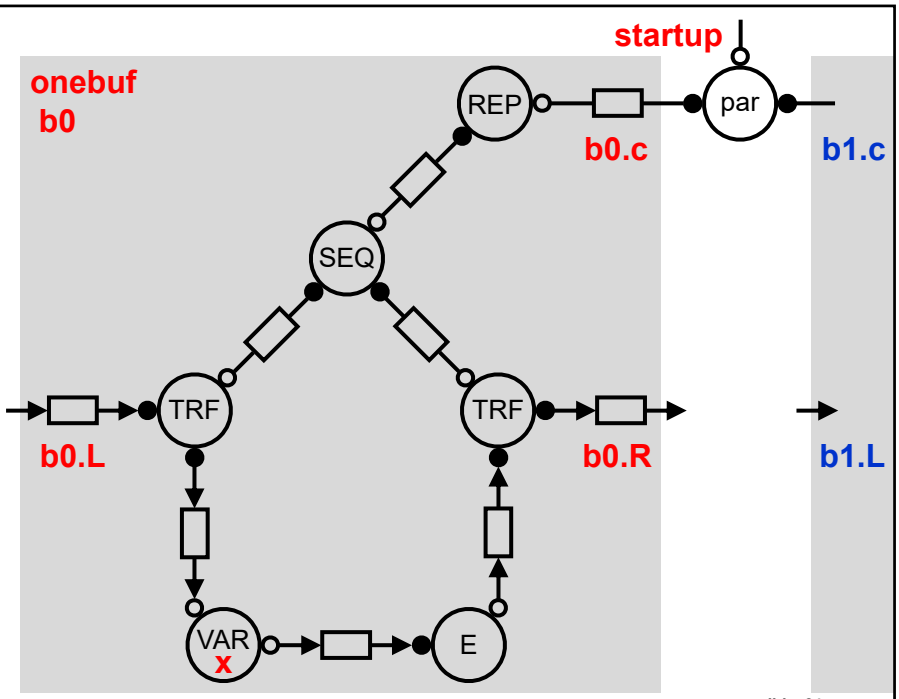
```
defproc onebuf
(chan?(int) L; chan!(int) R)
{
  int x ;
  chp { *[ L?x ; R!x ] }
}
```



Flexible Compilation: CHP

```
defproc onebuf
(chan?(int) L; chan!(int) R)
{
  int x ;
  chp { *[ L?x ; R!x ] }
}

defproc FIFO2_controlflow
(chan?(int) L; chan!(int) R)
{
  onebuf b0, b1;
  b0.L=L ; b0.R=b1.L ;
  b1.R=R
}
```

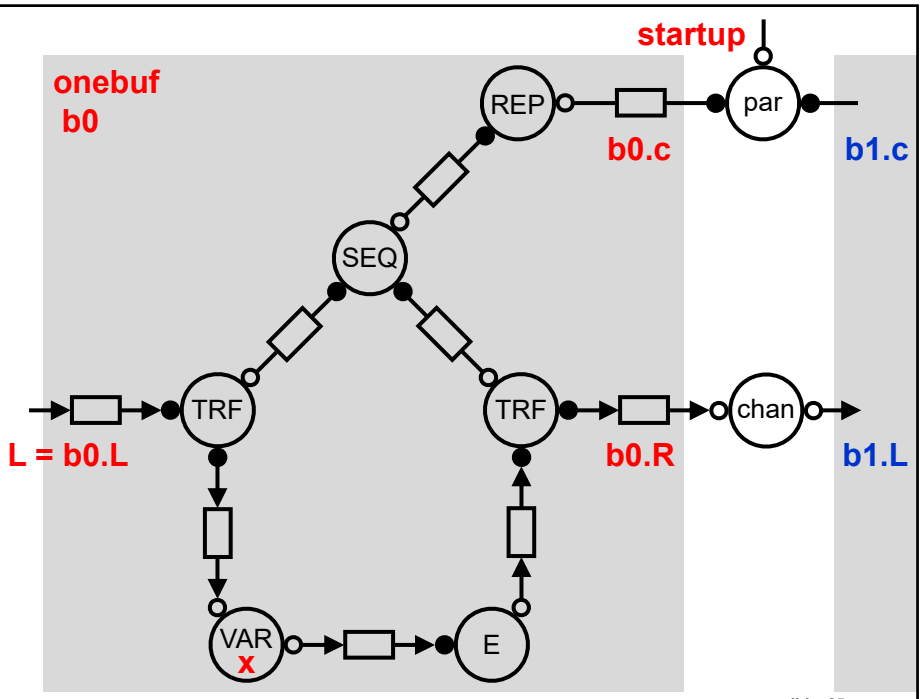


24

Flexible Compilation: CHP

```
defproc onebuf
(chan?(int) L; chan!(int) R)
{
  int x ;
  chp { *[ L?x ; R!x ] }
}

defproc FIFO2_controlflow
(chan?(int) L; chan!(int) R)
{
  onebuf b0, b1;
  b0.L=L ; b0.R=b1.L ;
  b1.R=R
}
```



25

Outline

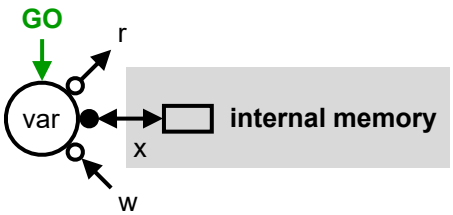
- Introduction
- Links and Joints
- Flexible Compilation: library elements
- Flexible Refinement
- Mixed Protocols and Families
- Conclusion

Flexible Compilation: library elements

icon:

flow:

Link-Joint network:

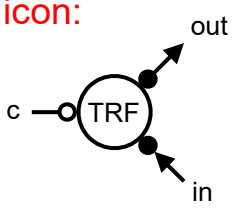


guarded command specification:

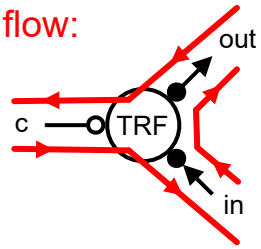
- *myturn(r) ∧ myturn(x) ∧ **GO** ⟶*
  myW(r) := myR(x) ; yourturn(r)
- *myturn(w) ∧ myturn(x) ∧ **GO** ⟶*
  myW(x) := myR(w) ; yourturn(w)

Flexible Compilation: library elements

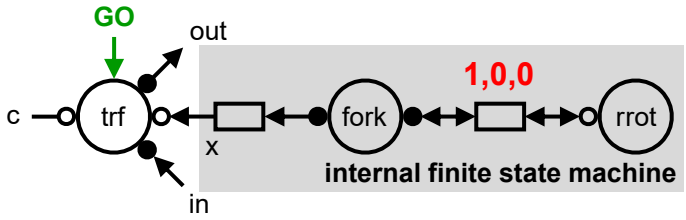
icon:



flow:



Link-Joint network:



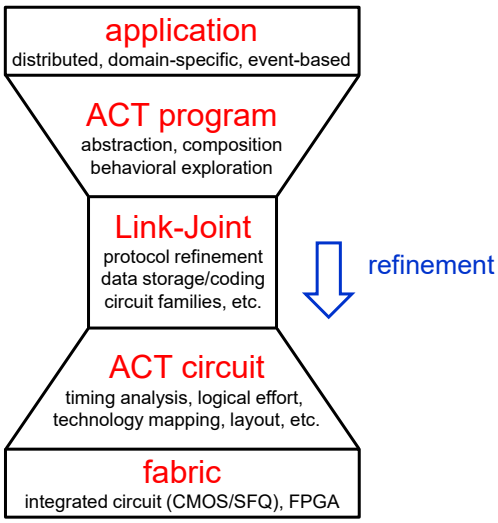
guarded command specification:

- *myturn(c,in,out,x)* ∧ ***GO*** ∧ *myR(x)[0]* ⟶ *yourturn(in, x)*
- *myturn(c,in,out,x)* ∧ ***GO*** ∧ *myR(x)[1]* ⟶ *myW(out) := myR(in) ; yourturn(out, x)*
- *myturn(c,in,out,x)* ∧ ***GO*** ∧ *myR(x)[2]* ⟶ *yourturn(c, x)*

Outline

- Introduction and Motivation
- Links and Joints
- Flexible Compilation
- Flexible Refinement — **choose bindings**
- Mixed Protocols and Families
- Conclusion

Flexible Refinement



Strategy:
- stepwise decisions for design and test

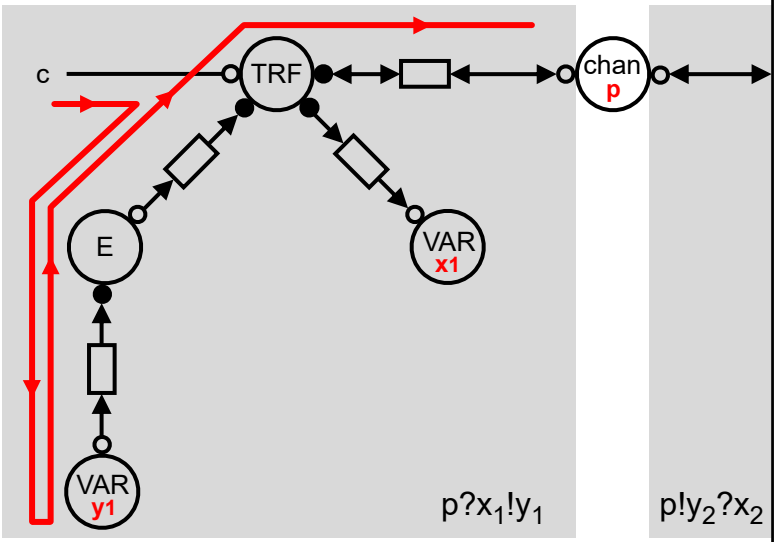
Source:
- circuit-neutral Link-Joint networks

Target:
- Link-Joint networks
- circuits

Challenge:
- preserve relation to program

Flexible Refinement: to store data — or not

Flexible Refinement: to store data — or not

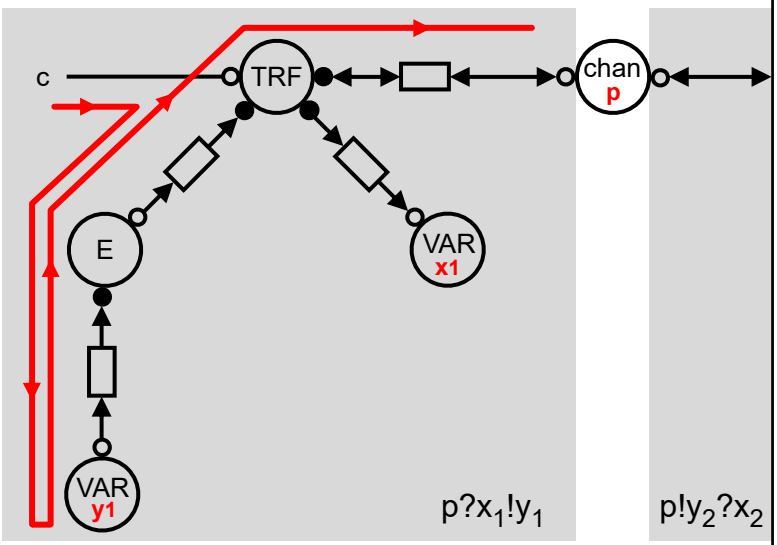


Flexible Refinement: to store data — or not

Path behavior:

- earlier Link stores data for later Link
- typical for CHP compilation
  - VAR **y1** stores data for **p**

Avoid data storage in the middle:



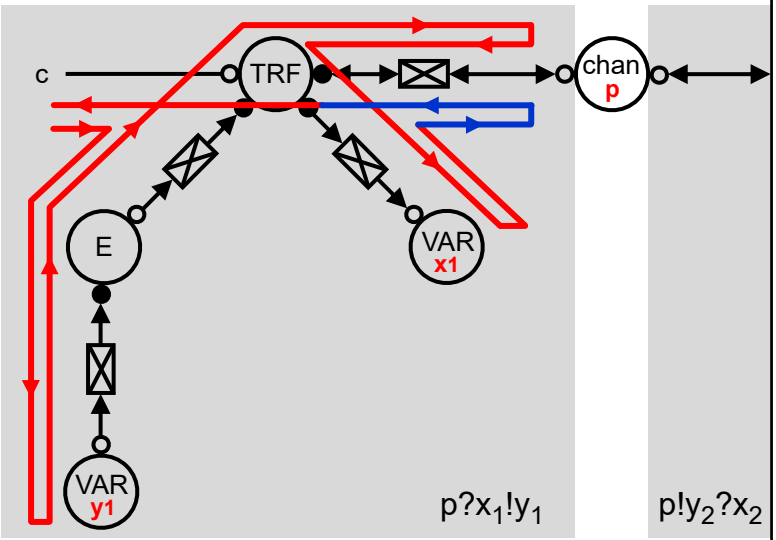
Flexible Refinement: to store data — or not

Path behavior:

- earlier Link stores data for later Link
- typical for CHP compilation
  - VAR **y1** stores data for **p**

Avoid data storage in the middle:

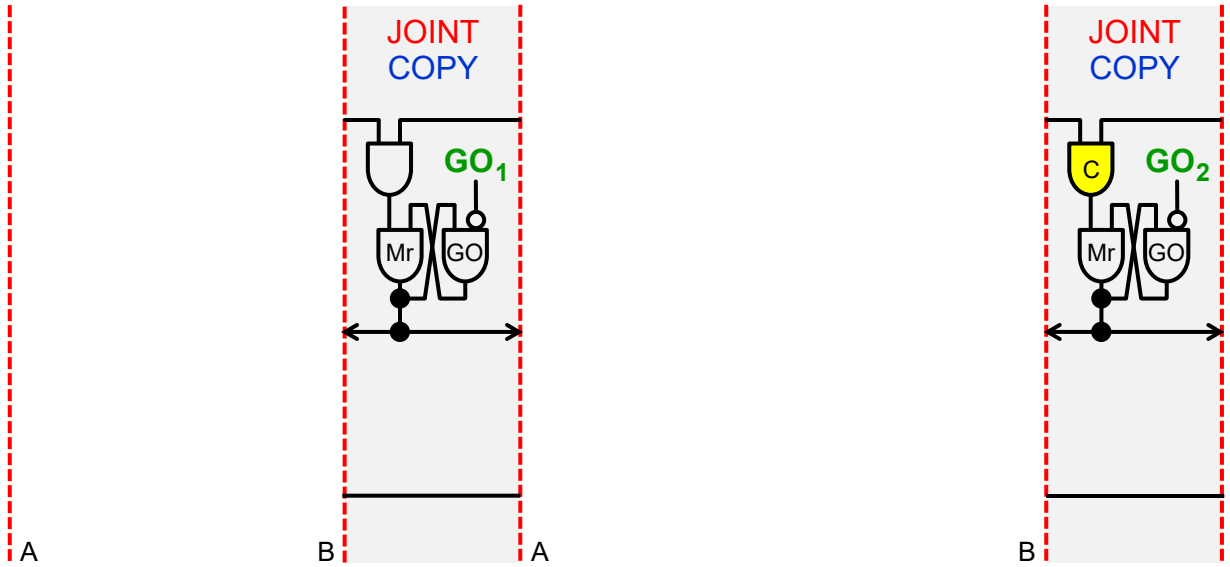
- Solution (race-free):
  - keep internal data storage (VAR)
  - ***no*** data storage otherwise: ⊠
  - use 4-phase **p** protocol
    ≈ 2x 2-phase protocol



Outline

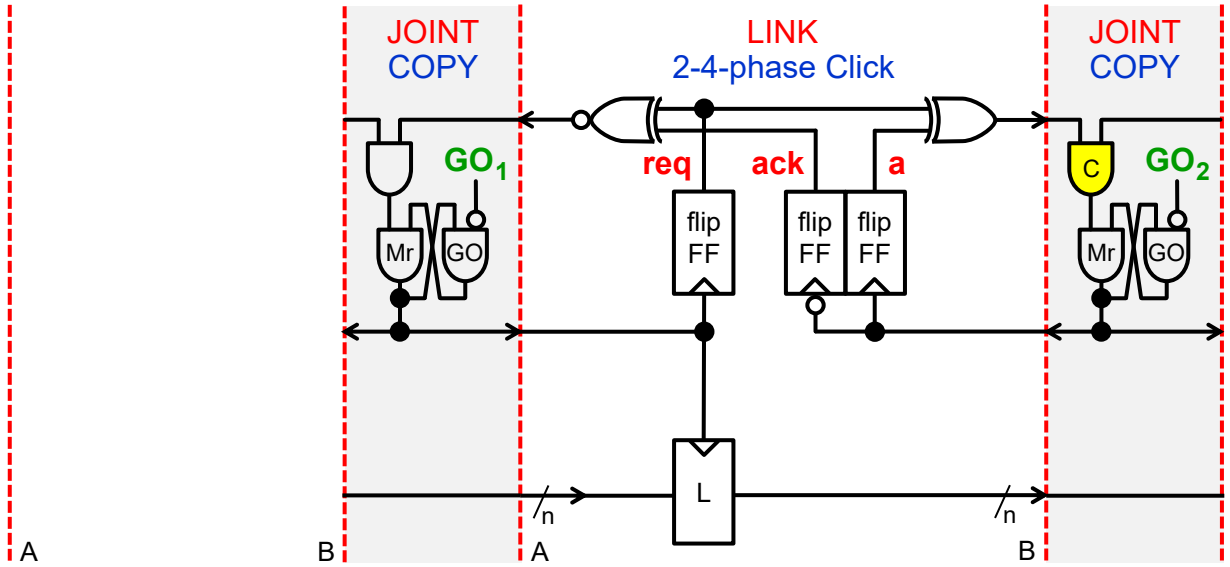
- Introduction and Motivation
- Links and Joints
- Flexible Compilation
- Flexible Refinement
- Mixed Protocols and Families — **easy mix and match**
- Conclusion

Mixed Protocols and Families



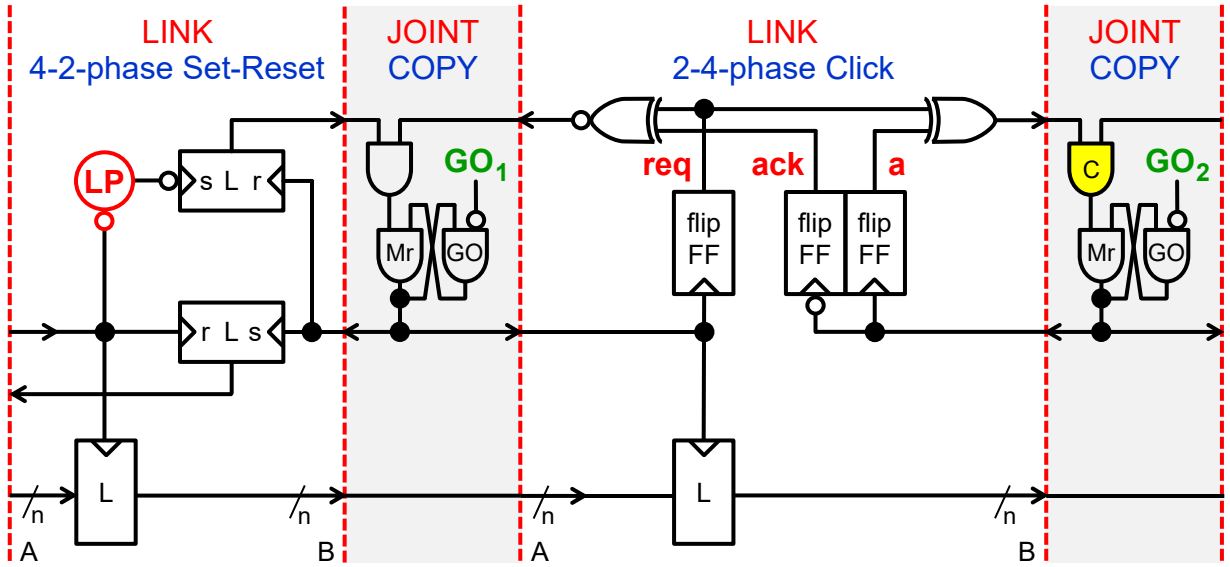
36

Mixed Protocols and Families



37

Mixed Protocols and Families



38

Outline

- Introduction and Motivation
- Links and Joints
- Flexible Compilation
- Flexible Refinement
- Mixed Protocols and Families
- Conclusion

39

Conclusion

- Link-Joint embedding into a Design Flow
  - enables more users and larger designs
  - combines design automation with Link-Joint flexibility
  - re-uses asynchronous ecosystem — Yale ACT
- Compilation and Refinement
  - from ACT programs with data- and control-flow
  - via circuit-neutral Link-Joint networks
  - to circuits

Conclusion

- Flexibility

  Bind decisions as late as possible — to serve design and test
  - initialize at run time (for design) or even throughout run time (for test)
  - per Link: **freedom of circuit family**
  - per Joint: **freedom of 2- or 4-phase protocol**
  - current support and w.i.p. includes:
    - 2- and 4-phase protocol, level- and pulse- and transition-logic, bundled and dual-rail data
    - Click, GasP, Set-Reset, Mousetrap, Micropipelines, Superconducting families

Goal: "Make it easy to insert asynchrony appropriate for each design part"

THANK YOU!