A half-baked approach to an embedded device driver framework in Esterel

Timothy Bourke and Leonid Ryzhyk

School of Computer Science and Engineering, UNSW

and National ICT Australia

Arcot Sowmya

UNSW Asia, Singapore

and UNSW Sydney

Ihor Kuz

National ICT Australia

THE UNIVERSITY OF NEW SOUTH WALES
SYDNEY · AUSTRALIA

Leonid's problem:

Reliable driver framework

My interest:

using synchronous languages?

- Evaluate feasibility and identify related work, so far:
  - More questions than answers
  - Ideas rather than implementations
- Today: criticisms, opinions, missing references

Reliable I/O Framework: Aims

Device drivers:

- 70% of systems code [Chou et al., 2001]
- 3 times more bugs per line of code

Reliable I/O Framework: Aims

Device drivers:

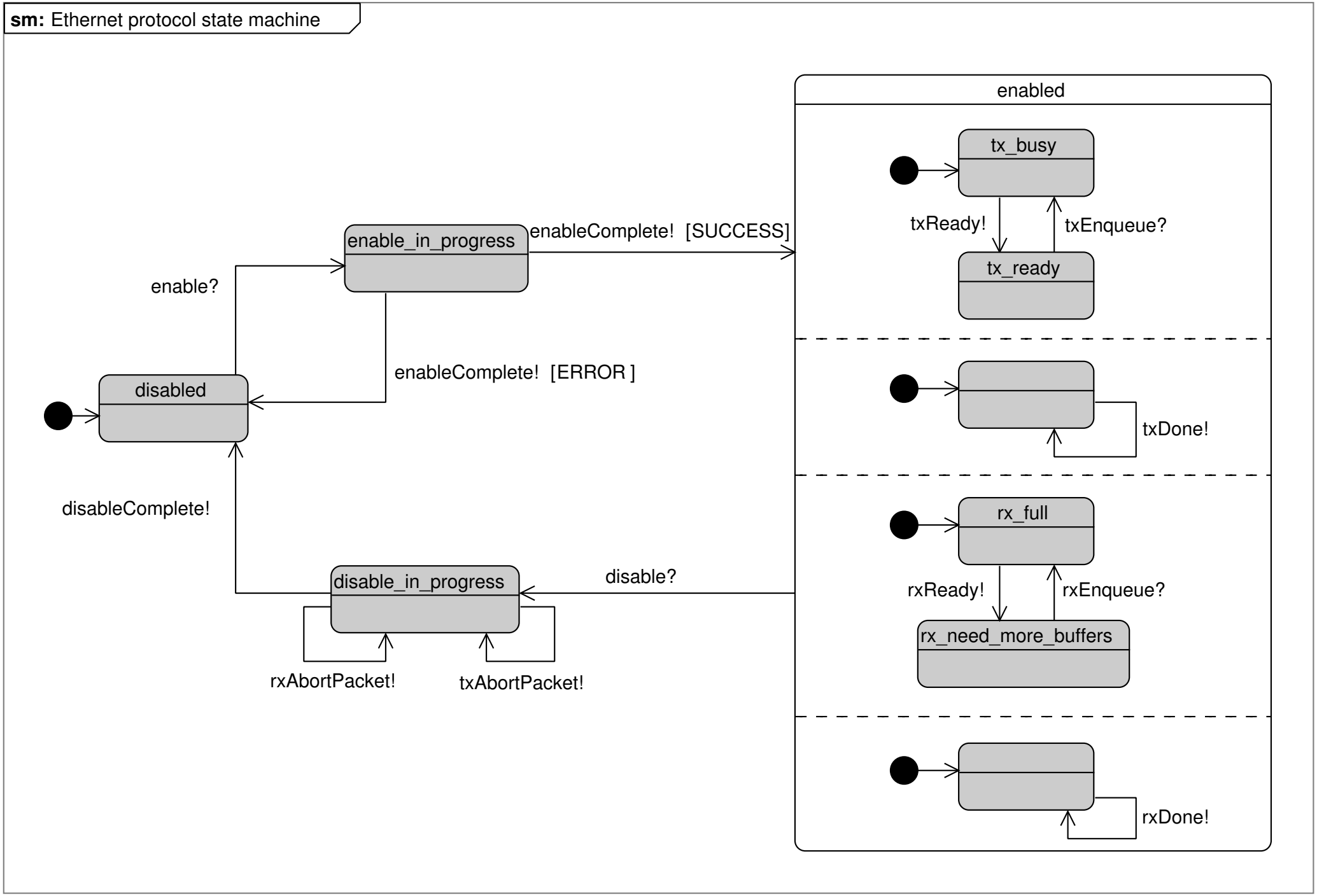
- 70% of systems code [Chou et al., 2001]
- 3 times more bugs per line of code

fault prevention
reduce number of faults

- **fault avoidance**
  - structure framework and API to simplify development
- **fault removal**
  - find and eliminate bugs in SW

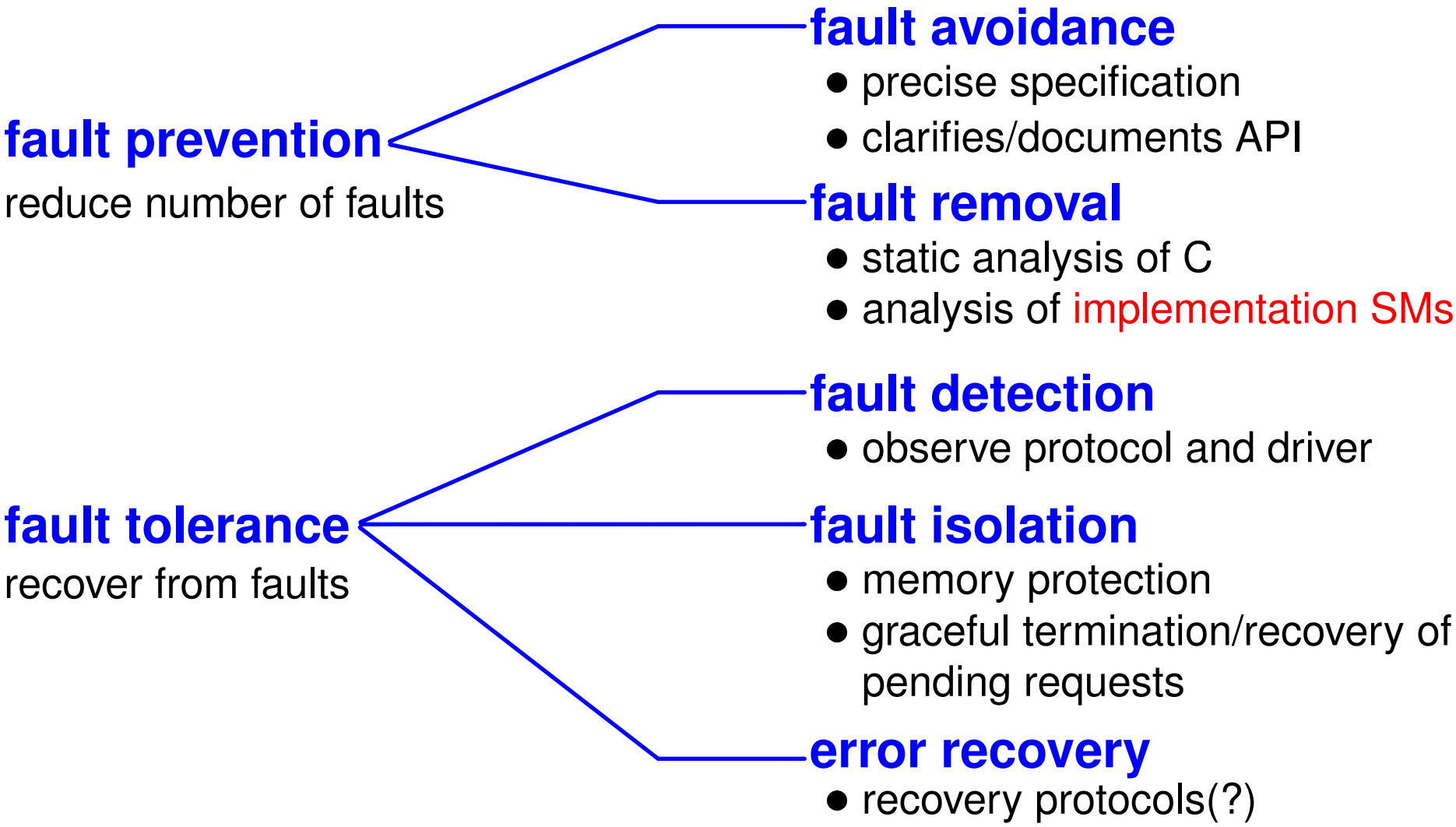
fault tolerance
recover from faults

- **fault detection**
  - recognise bad states & behaviours
- **fault isolation**
  - prevent faulty drivers from corrupting OS and applications
- **error recovery**
  - MINIX, NOOKS


sm: Ethernet protocol state machine
enabled
tx_busy
txReady!
txEnqueue?
tx_ready
enable_in_progress
enableComplete! [SUCCESS]
enable?
enableComplete! [ERROR]
disabled
txDone!
disableComplete!
rx_full
rxReady!
rxEnqueue?
rx_need_more_buffers
disable_in_progress
disable?
rxAbortPacket!
txAbortPacket!
rxDone!

Reliable I/O Framework: Protocol State Machines

Constraints on the ordering and content of messages exchanged through a port



Reliable I/O Framework: Extend ‘Iguana’ Framework

- L4 micro-kernel
- user level drivers
- shift complexity
  - buffering,
  - memory allocation,
  - concurrency

  into the framework

- single-threaded
  - non-blocking
- split events:
  - request
  - completion

L4 microkernel [Liedtke, 1995]

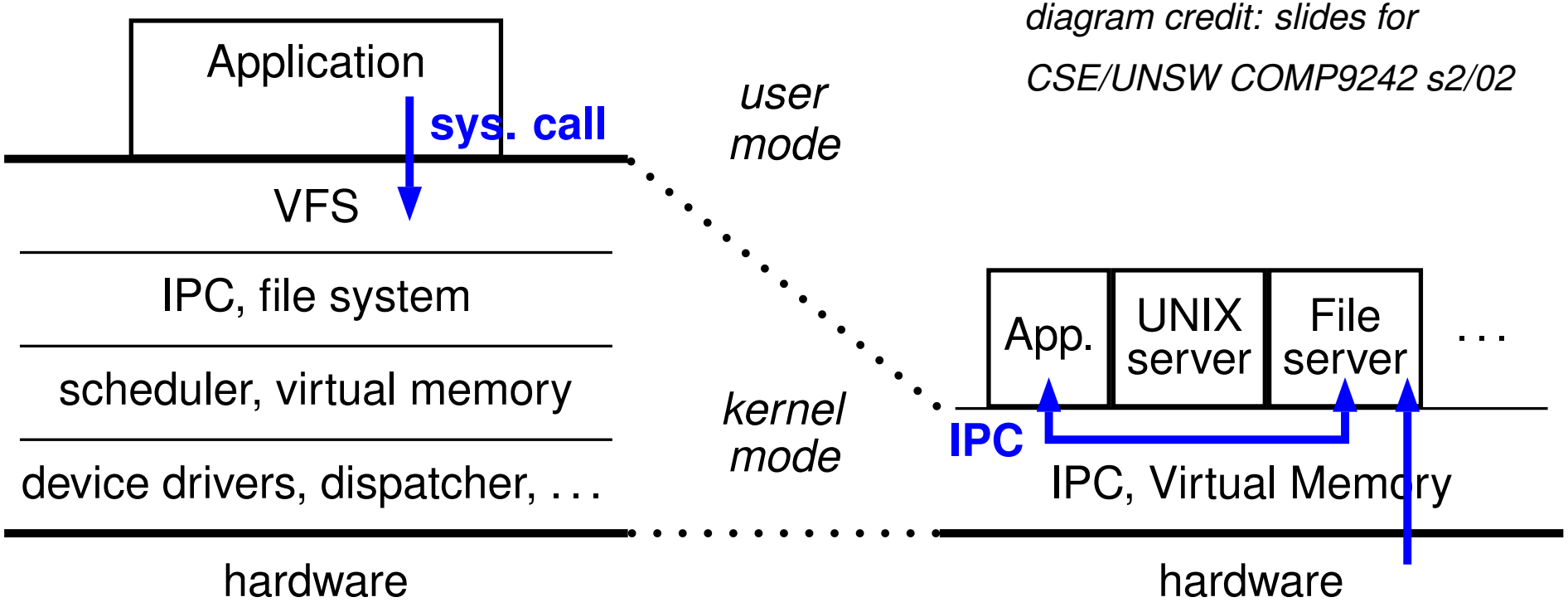


diagram credit: slides for CSE/UNSW COMP9242 s2/02

- Address spaces
- Threads
- Heavily optimised IPC

- thin layer on top of hardware
- Sydney version targeted at embedded systems
- Potential mismatch with IPC and architecture
- not real-time

Is Esterel suitable for such a subsystem?

Is Esterel suitable for such a subsystem?

(not answered!)

Is Esterel suitable for such a subsystem? *(not answered!)*

highly suitable? no research needed; *just* engineering

- answer: show efficient prototype using existing tools

Is Esterel suitable for such a subsystem? *(not answered!)*

highly suitable? no research needed; *just* engineering

- answer: show efficient prototype using existing tools

not suitable? give up!

- answer: reasons why, alternatives

Is Esterel suitable for such a subsystem? *(not answered!)*

highly suitable? no research needed; *just* engineering

- answer: show efficient prototype using existing tools

suitable. . .

- **if** *what must be done?*
- **but** *what are the limitations?*

not suitable? give up!

- answer: reasons why, alternatives

Is Esterel suitable for such a subsystem?

Advantageous properties:

- Notation for event-driven reactive systems
- Synchronous rounds may simplify recovery mechanisms
- Amenable to validation (interfaces, model-checking)

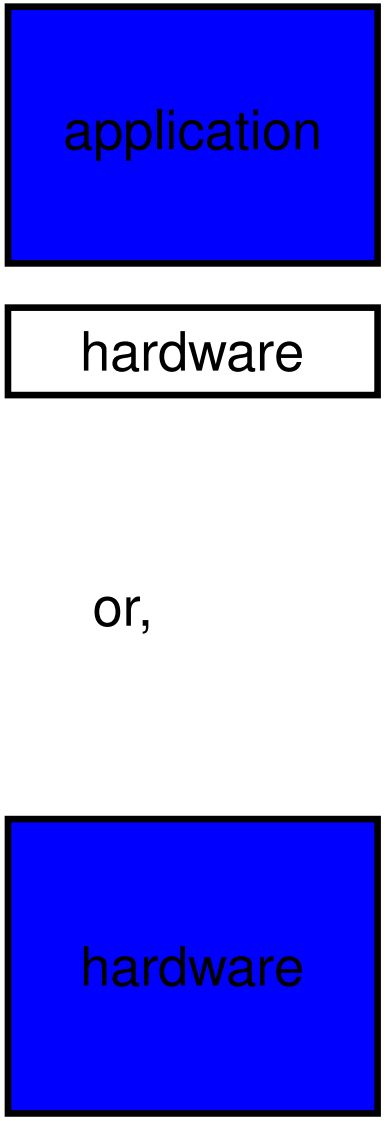
Desirable properties:

- Able to exploit hardware memory protection
- Separate compilation of drivers
- Operate in a multi-tasking environment
- Tractable reasoning about execution *within* the system context
- Limited addition and removal of hardware
- Track user request state

Presentation Outline

- √ Reliable I/O Framework
- ⇒ Possible Architectures
  - Strictly event-driven modules
  - Event-driven scheduling
  - Example: IP Stack and Ethernet driver
  - Other related work
  - Summary

Possible Architectures: monolithic



- Wristwatch [Berry, 1989]
- Most straight-forward application of Esterel
- Utilise standard tools
- Responsible for interface code

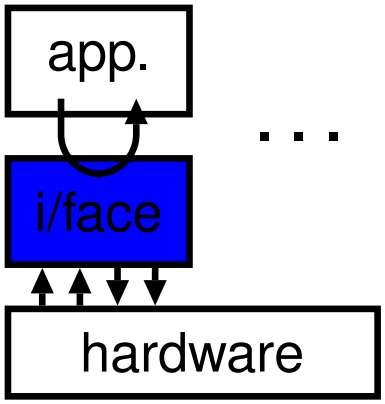
Reasoning about system:

- Behaviour of the underlying transition system
- Meaning of signals (depends on interface)

For driver framework:

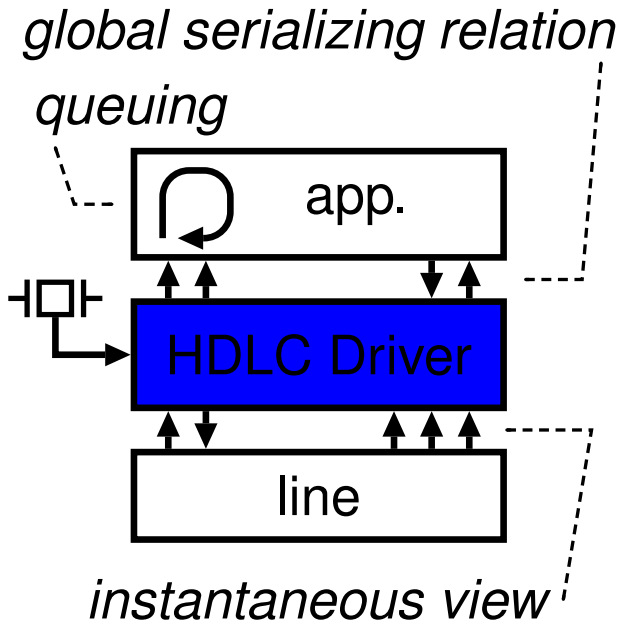
- Fast execution (almost no run-time overhead)
- No memory protection
- Relatively inflexible

Architectures: HW/SW Interface (single driver)



- HDLC Protocol [Berry and Gonthier, 1989]
- Utilize standard tools
- Application side: may need to avoid blocking

e.g.



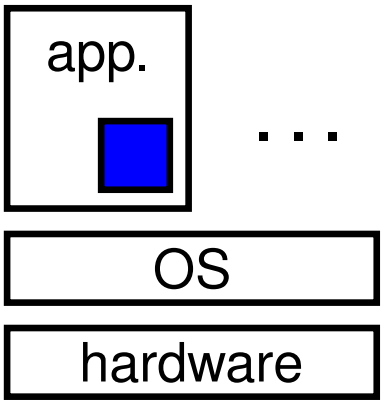
Reasoning about system:

- As per normal
- Queueing in application

For driver framework:

- Memory protection possible
- Communication between drivers?

Architectures: Application subsystem

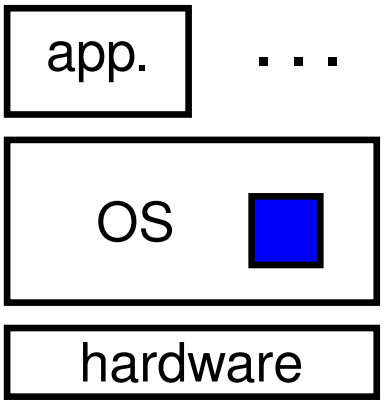


- AT&T 5ESS switch [Jagadeesan et al., 1995]
- Part of TCP/IP stack [Castelluccia et al., 1996]
- Utilize standard tools
- Ad hoc interfacing code

and/or,

Reasoning about system:

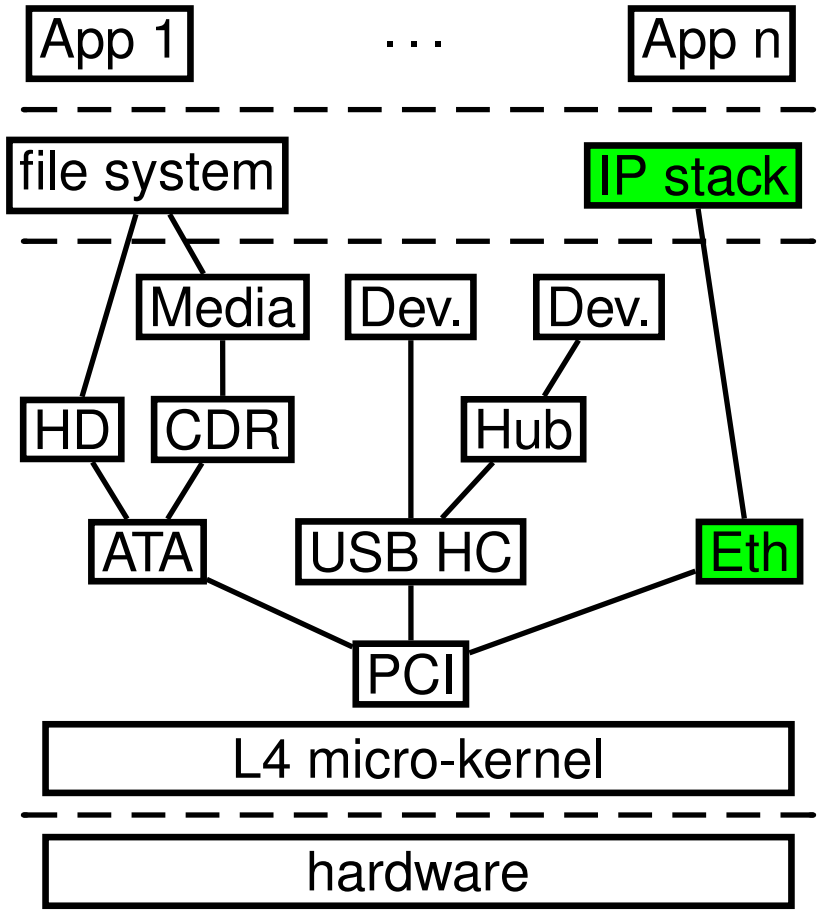
- Dependent on application environment



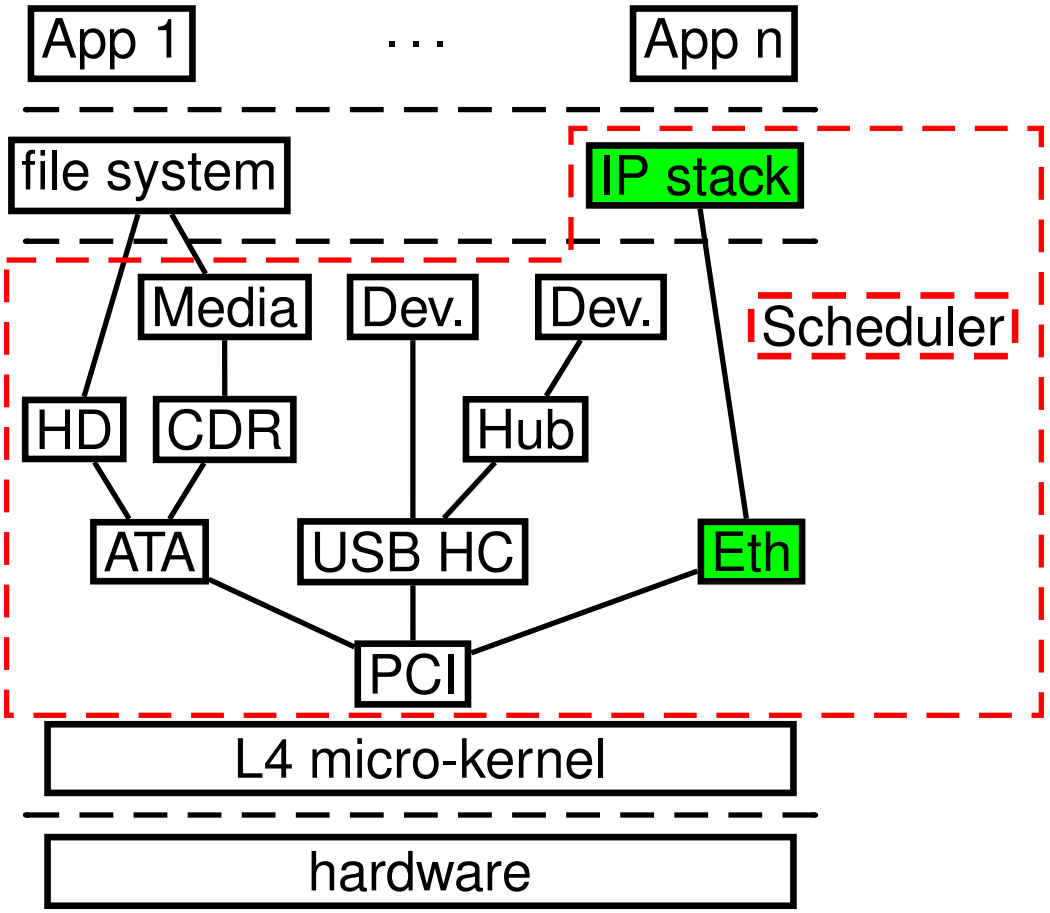
For driver framework:

- Less suitable for micro-kernel with user-level drivers

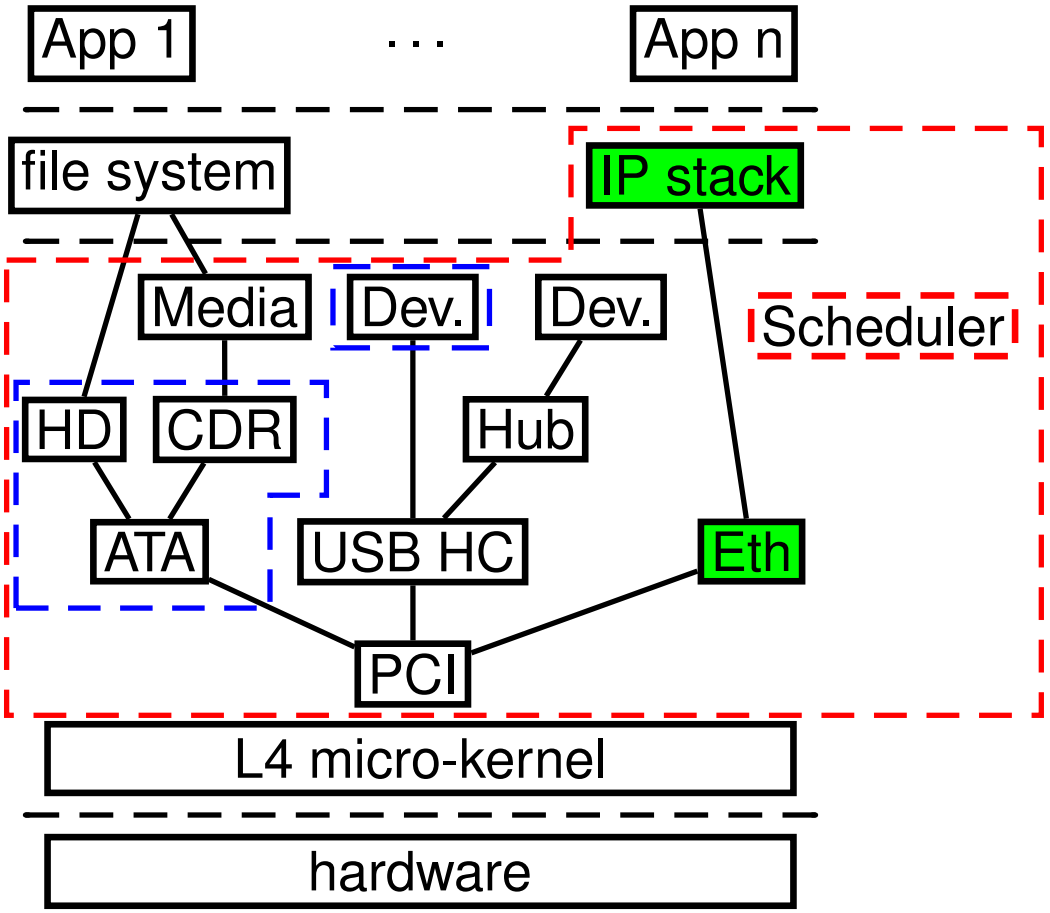
Architectures: Driver Framework



Architectures: Driver Framework



Architectures: Driver Framework



Two-level architecture

Dynamically scheduled level: flat calculus [Berry and Sentovich, 2001]

- **run** comp1[connections] || ... || **run** compN[connections]

- permit (initially):
  - **signal** S **in** ... **end**
  - **loop** ... **end**?
  - **weak abort** ... **when** S?

- Memory protection for top-level modules
- Trade-off cost of dynamic scheduling against safety/flexibility
- More constructs for recovery features?
- Feasiblity uncertain!

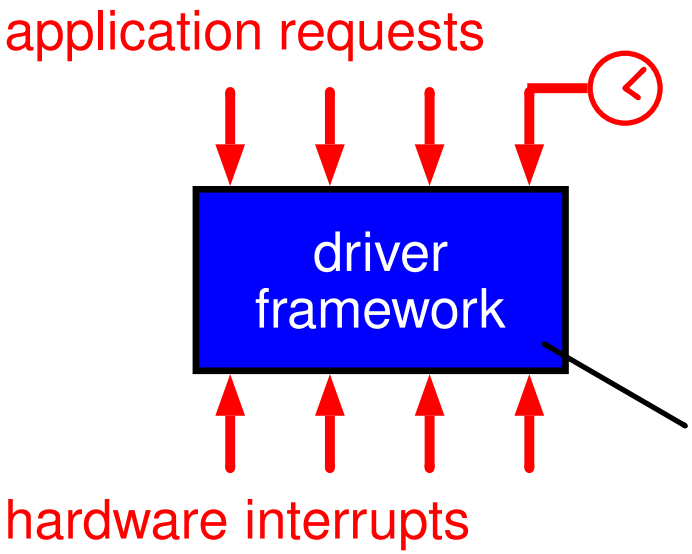
Component level: strictly event-driven modules

- Block waiting for events (as per usual)
- Minimise computation/context-switching

Presentation Outline

- √ Reliable I/O Framework
- √ Possible Architectures
- ⇒ Strictly event-driven modules
- Event-driven scheduling
- Example: IP Stack and Ethernet driver
- Other related work
- Summary

Triggering



Multiform time [Berry, 2000]

General form [Caspi, 2001]

Whatever the structure:

- monolithic
- multi-threaded
- . . .

event-driven

rather than

sample-driven

Strictly event-driven modules

Modules where observable behaviour and state changes occur strictly in response to events listed in the input/output interface and are otherwise independent of reaction triggering.

statement	**strictly event-driven?**
await A; **emit** O	
pause; **emit** O	
await A; **pause**; **emit** O	
loop pause end (**halt**)	

Strictly event-driven modules

Modules where observable behaviour and state changes occur strictly in response to events listed in the input/output interface and are otherwise independent of reaction triggering.

statement	**strictly event-driven?**
await A; **emit** O	**yes**
pause; **emit** O	
await A; **pause**; **emit** O	
loop pause end (**halt**)	

Strictly event-driven modules

Modules where observable behaviour and state changes occur strictly in response to events listed in the input/output interface and are otherwise independent of reaction triggering.

statement	**strictly event-driven?**
await A; **emit** O	**yes**
pause; **emit** O	**no**
await A; **pause**; **emit** O	
loop pause end (**halt**)	

Strictly event-driven modules

Modules where observable behaviour and state changes occur strictly in response to events listed in the input/output interface and are otherwise independent of reaction triggering.

statement	**strictly event-driven?**
await A; **emit** O	**yes**
pause; **emit** O	**no**
await A; **pause**; **emit** O	**no**
loop pause end (**halt**)	

Strictly event-driven modules

Modules where observable behaviour and state changes occur strictly in response to events listed in the input/output interface and are otherwise independent of reaction triggering.

statement	**strictly event-driven?**
await A; **emit** O	**yes**
pause; **emit** O	**no**
await A; **pause**; **emit** O	**no**
loop pause end (**halt**)	**yes**

Strictly event-driven modules

Modules where observable behaviour and state changes occur strictly in response to events listed in the input/output interface and are otherwise independent of reaction triggering.

statement	strictly event-driven?
await A; **emit** O	**yes**
pause; **emit** O	**no**
await A; **pause**; **emit** O	**no**
loop pause end (**halt**)	**yes**

$P \xhookrightarrow[O]{I} P'$ module: $(I_M, O_M).M$ $I_M \subseteq I$ and $O_M \subseteq O$

$$(\forall s \in I_M.\ s^- \in I) \Longrightarrow (\forall s \in O_M.\ s^- \in O)^\dagger \wedge (P = P')$$

(I, O).A notation: [Berry and Sentovich, 2001]

† *not quite*

Strictly event-driven modules: relative absence

. . . respond only to events with at least one of the module inputs present. . .

statement	strictly event-driven?
await A; **present** S **else emit** O **end**	
await [**not** A]	
await [A **and not** B]; **emit** O	
await [A **or not** B]; **emit** O	

Strictly event-driven modules: relative absence

. . . respond only to events with at least one of the module inputs present. . .

statement	strictly event-driven?
await A; **present** S **else emit** O **end**	**yes**
await [**not** A]	
await [A **and not** B]; **emit** O	
await [A **or not** B]; **emit** O	

Strictly event-driven modules: relative absence

. . . respond only to events with at least one of the module inputs present. . .

statement	strictly event-driven?
await A; **present** S **else emit** O **end**	**yes**
await [**not** A]	**no**
await [A **and not** B]; **emit** O	
await [A **or not** B]; **emit** O	

Strictly event-driven modules: relative absence

. . . respond only to events with at least one of the module inputs present. . .

statement	strictly event-driven?
await A; **present** S **else emit** O **end**	**yes**
await [**not** A]	**no**
await [A **and not** B]; **emit** O	**yes**
await [A **or not** B]; **emit** O	

Strictly event-driven modules: relative absence

. . . respond only to events with at least one of the module inputs present. . .

statement	strictly event-driven?
await A; **present** S **else emit** O **end**	**yes**
await [**not** A]	**no**
await [A **and not** B]; **emit** O	**yes**
await [A **or not** B]; **emit** O	**no**

Strictly event-driven modules: relative absence

. . . respond only to events with at least one of the module inputs present. . .

statement	strictly event-driven?
await A; **present** S **else emit** O **end**	**yes**
await [**not** A]	**no**
await [A **and not** B]; **emit** O	**yes**
await [A **or not** B]; **emit** O	**no**

```
module main:
input A;
output O;
await [not A];
emit O
end module
```

- no problem if sample-driven,
- but what should an event-driven system do?

Strictly event-driven modules: await

await A

```
trap T in
    loop
        pause;
        present A then exit T
                  else nothing
        end present
    end loop
end trap
```

await [**not** A]

```
trap T in
    loop
        pause;
        present A then nothing
                  else exit T
        end present
    end loop
end trap
```

any inputs with A absent imply $P \neq P'$

Strictly event-driven modules: other

strong abortion:

```
abort
      await A;
      emit O
when B
```

weak abortion:

```
weak abort
      await A;
      emit O
when B
```

suspension:

```
suspend
      await A;
      emit O
when B
```

sustain:

```
loop
      emit S;
      pause
end loop
```

Strictly event-driven modules: other

strong abortion:

```
abort
    await A;
    emit O
when B
```

restriction: **await** [A **and** (**not** B)]

triggering

weak abortion:

```
weak abort
    await A;
    emit O
when B
```

suspension:

```
suspend
    await A;
    emit O
when B
```

sustain:

```
loop
    emit S;
    pause
end loop
```

Strictly event-driven modules: other

strong abortion:

```
abort
      await A;
      emit O
when B
```

restriction: **await** [A **and** (**not** B)]

triggering

weak abortion:

```
weak abort
      await A;
      emit O
when B
```

triggering

suspension:

```
suspend
      await A;
      emit O
when B
```

sustain:

```
loop
      emit S;
      pause
end loop
```

Strictly event-driven modules: other

strong abortion:

```
abort
    await A;
    emit O
when B
```

→ restriction: **await** [A **and** (**not** B)]

→ triggering

weak abortion:

```
weak abort
    await A;
    emit O
when B
```

→ triggering

suspension:

```
suspend
    await A;
    emit O
when B
```

→ restriction: **await** [A **and** (**not** B)]

[Tardieu and de Simone, 2005]

sustain:

```
loop
    emit S;
    pause
end loop
```

Strictly event-driven modules: other

strong abortion:

```
abort
    await A;
    emit O
when B
```

→ restriction: **await** [A **and** (**not** B)]

→ triggering

weak abortion:

```
weak abort
    await A;
    emit O
when B
```

→ triggering

suspension:

```
suspend
    await A;
    emit O
when B
```

→ restriction: **await** [A **and** (**not** B)]

[Tardieu and de Simone, 2005]

sustain: † not strictly event-driven but I want it to be!

```
loop
    emit S;
    pause
end loop
```

more permanent, subject to restriction

Strictly event-driven modules: similarities

[Berry and Gonthier, 1992]:

pause $\stackrel{\text{def}}{=}$ **abort halt when** tick

now: **await** tick_i

[Berry and Sentovich, 2001]
[Halbwachs and Baghdadi, 2002]

[Berry, 1999]:

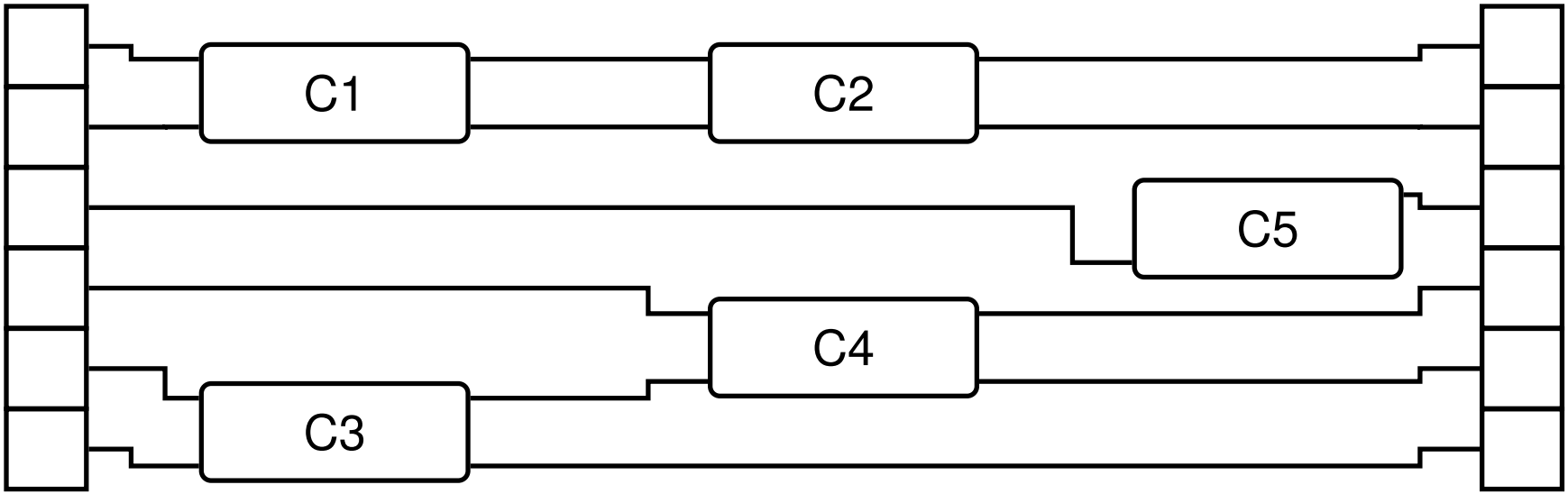
halt $\stackrel{\text{def}}{=}$ **loop pause end**

forbidden: **await** [**not** tick]

- *Not* a new semantics; rather a restriction
- Why?
  - conceptual issue of triggering
  - simplify reasoning about effects in system
  - try to exploit in implementation (scheduling)

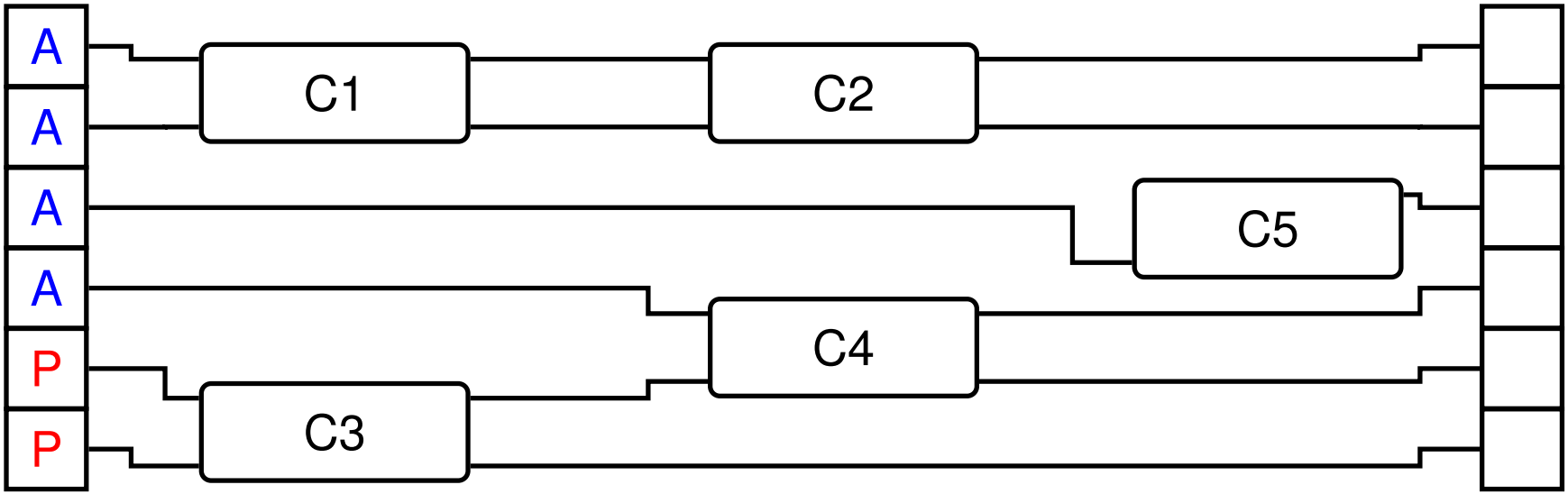
Event-driven scheduling

run C1 || **run** C2 || **run** C3 || **run** C4 || **run** C5



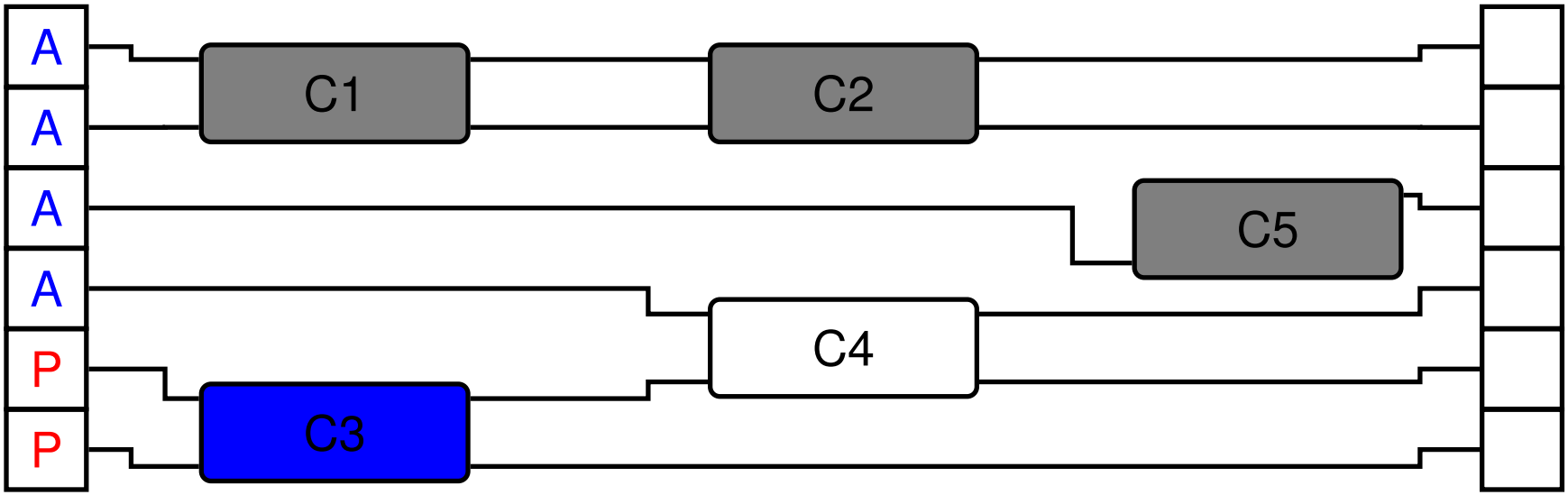
Event-driven scheduling

run C1 || **run** C2 || **run** C3 || **run** C4 || **run** C5



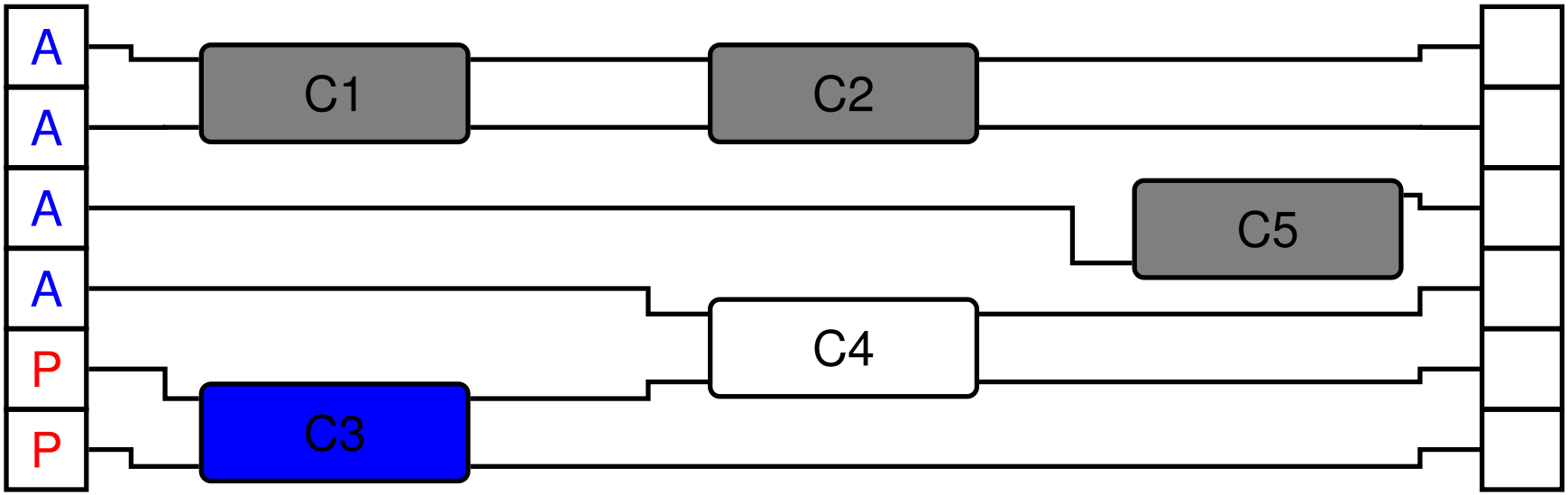
Event-driven scheduling

run C1 || **run** C2 || **run** C3 || **run** C4 || **run** C5



Event-driven scheduling

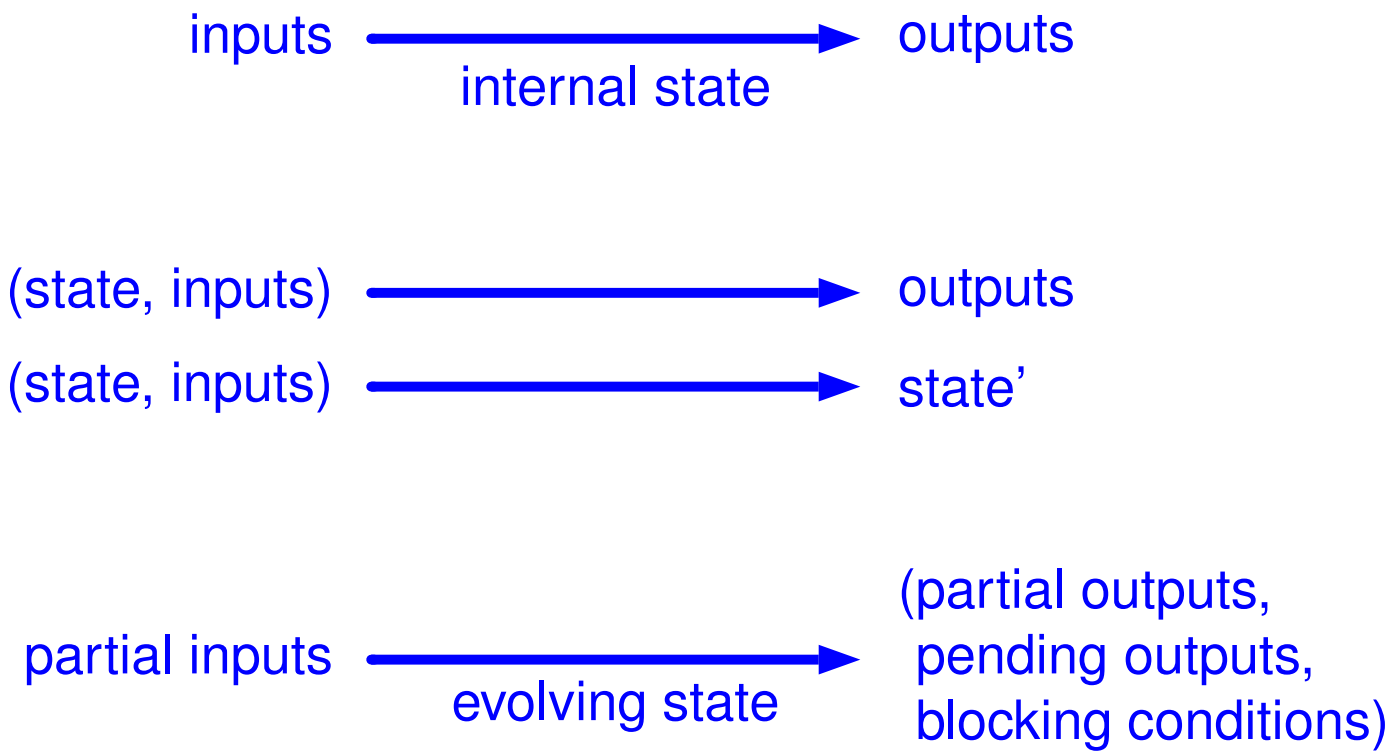
run C1 || **run** C2 || **run** C3 || **run** C4 || **run** C5



Why?

- Avoid unnecessary context switching
- Anticipate that many drivers are inactive at a given instant
- Make ‘stand-by’ drivers practicable

Module implementation interfaces



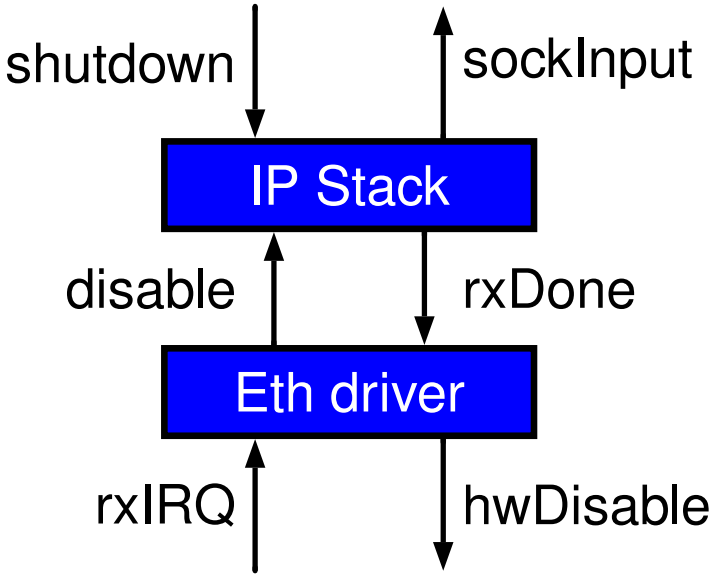
Two stage operation:

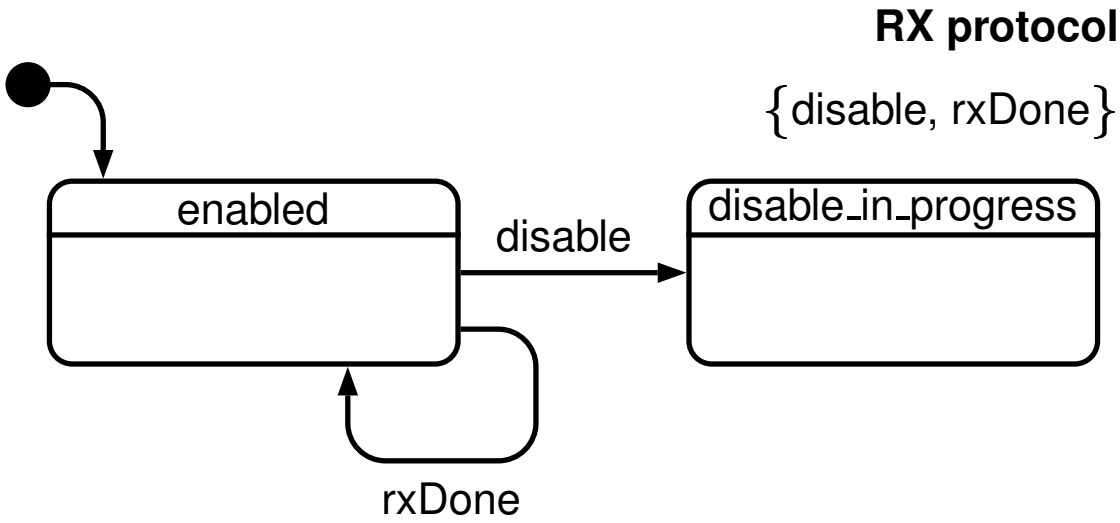
1. Paused: publish triggering events (sustained outputs)
2. Participating: between [Edwards and Lee, 2003] and [Potop-Butucaru, 2002]

Presentation Outline

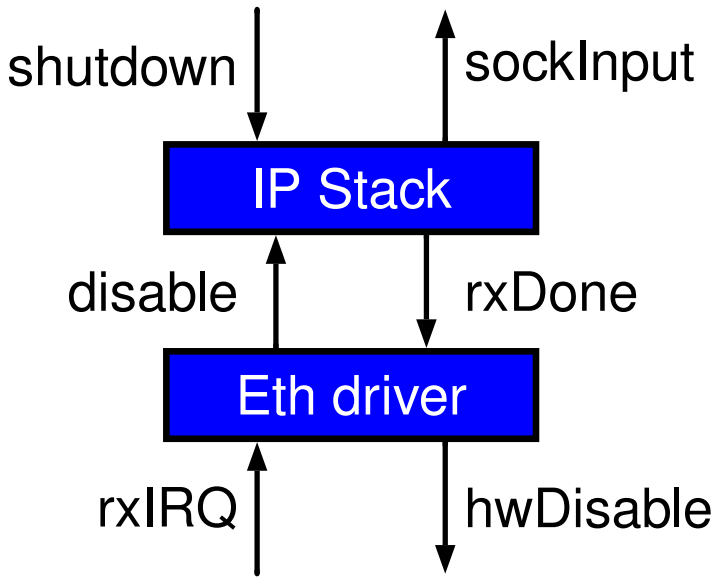
- √ Reliable I/O Framework
- √ Possible Architectures
- √ Strictly event-driven modules
- √ Event-driven scheduling
- ⇒ Example: IP Stack and Ethernet driver
- Other related work
- Summary

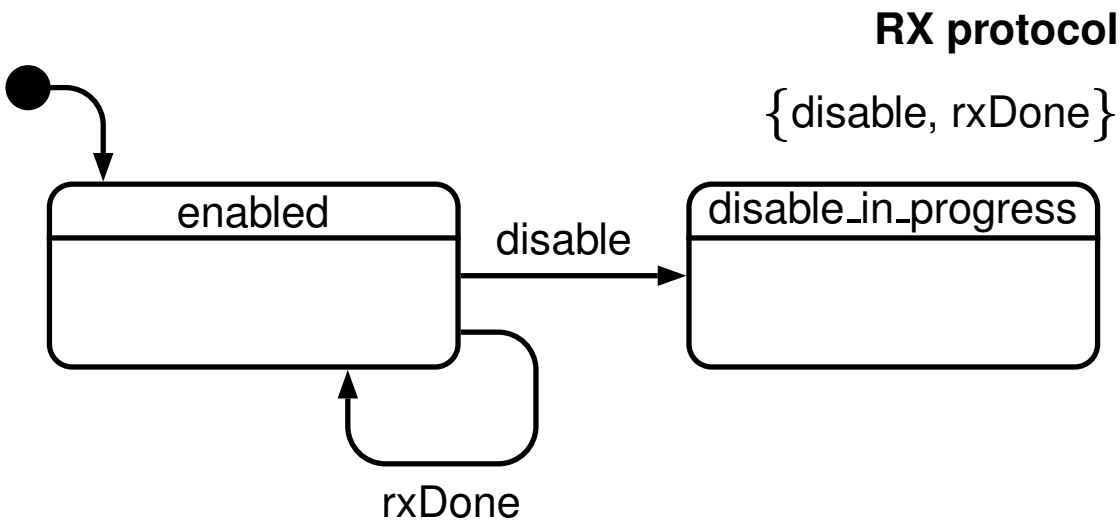
Example: IP Stack and Ethernet driver





Example: IP Stack and Ethernet driver

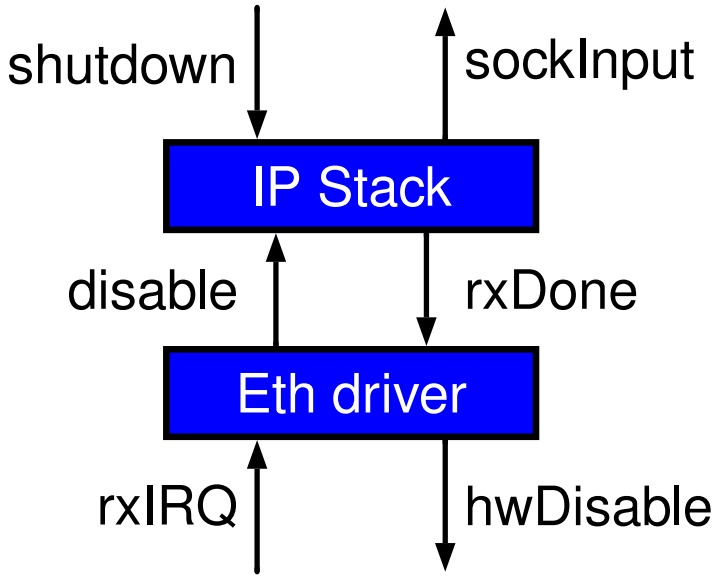


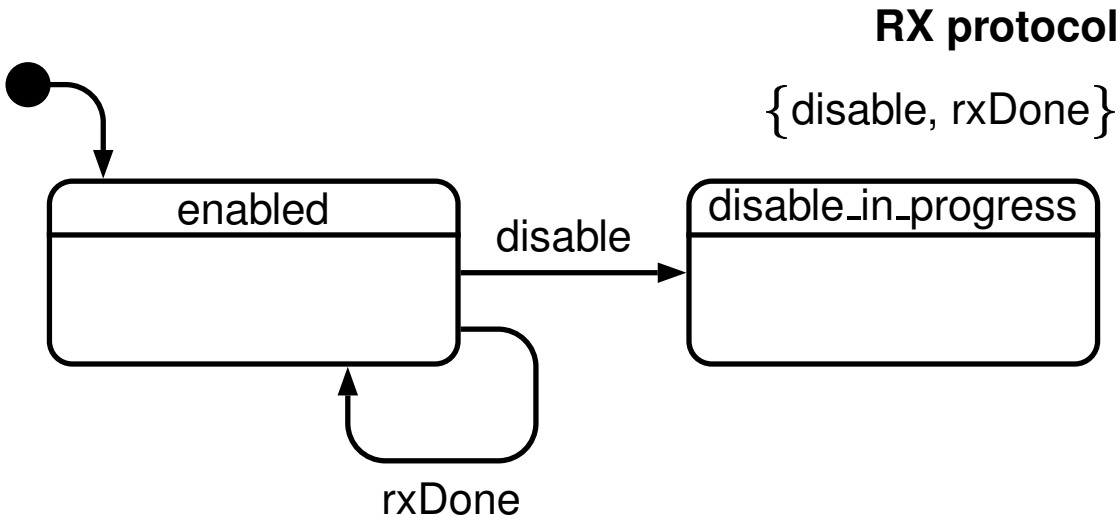



```
input rxIRQ, shutdown;
output hwDisable, sockInput;

signal disable, rxDone in
    run driver
    ||
    run ipstack
end signal
```

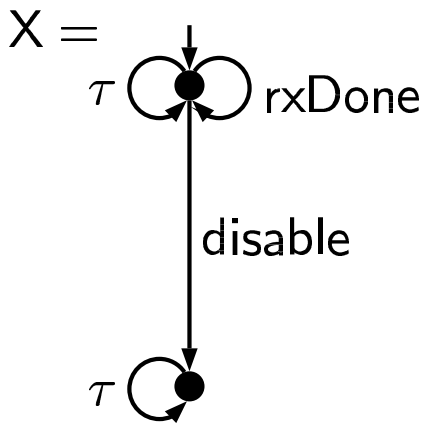
Example: IP Stack and Ethernet driver






```
input rxIRQ, shutdown;
output hwDisable, sockInput;

signal disable, rxDone in
    run driver
    ||
    run ipstack
end signal
```



$\tau_I(P)$ *simulates* X

$I = \mathrm{A} \backslash \{\mathrm{rxDone}, \mathrm{disable}\}$

```
module driver:
input disable, rxIRQ;
output rxDone, hwDisable;

abort
    every rxIRQ do
        emit rxDone
    end every
when disable;
emit hwDisable;
halt

end module
```

```
module ipstack:
input rxDone, shutdown;
output disable, sockInput;
procedure input () ();

abort
    every rxDone do
        call input () ();
        emit sockInput
    end every
when shutdown;
emit disable;
halt

end module
```

```
module driver:
input disable, rxIRQ;
output rxDone, hwDisable;

abort
    every rxIRQ do
        emit rxDone
    end every
when disable;
emit hwDisable;
halt

end module
```

external input: rxIRQ

rxDone

input()

sockInput

```
module ipstack:
input rxDone, shutdown;
output disable, sockInput;
procedure input () ();

abort
    every rxDone do
        call input () ();
        emit sockInput
    end every
when shutdown;
emit disable;
halt

end module
```

```
module driver:
input disable, rxIRQ;
output rxDone, hwDisable;

abort
    every rxIRQ do
        emit rxDone
    end every
when disable;
emit hwDisable;
halt

end module
```

external input: rxIRQ

rxDone

input()

sockInput

external input: shutdown

disable

hwDisable

```
module ipstack:
input rxDone, shutdown;
output disable, sockInput;
procedure input () ();

abort
    every rxDone do
        call input () ();
        emit sockInput
    end every
when shutdown;
emit disable;
halt

end module
```

Presentation Outline

- √ Reliable I/O Framework
- √ Possible Architectures
- √ Strictly event-driven modules
- √ Event-driven scheduling
- √ Example: IP Stack and Ethernet driver
- ⇒ Other related work
- Summary

Reactive C, SL, SugarCubes?

[Boussinot, 1991]
[Boussinot and de Simone, 1996]
[Amadio, 2005]
[Boussinot and Susini, 1997]

all (SL) programs are deterministic and coherent

- Simple and modular compilation, no expensive analysis
- Add and remove components at run-time without fear

Drawbacks:

- Strong preemption is forbidden
- Pausing in absence branches:
  - Programs react to triggering events over several instants
  - Could complicate reasoning about behaviour and timing
  - Could complicate recovery logic

Systems Programming [Montague and McDowell, 1997]

asked the question:

> Are synchronous/reactive techniques appropriate for high-performance generic system software?

“operating system kernels; file, database, and network systems; device drivers; and server architectures for I/O, transaction processing and multimedia”

Systems Programming [Montague and McDowell, 1997]

asked the question:

> Are synchronous/reactive techniques appropriate for ~~high-performance~~ embedded ~~generic-system~~ software?
> device-driver

"operating system kernels; file, database, and network systems; device drivers; and server architectures for I/O, transaction processing and multimedia"

Systems Programming [Montague and McDowell, 1997]

asked the question:

> Are synchronous/reactive techniques appropriate for ~~high-performance~~ embedded ~~generic-system~~ device-driver software?

"operating system kernels; file, database, and network systems; device drivers; and server architectures for I/O, transaction processing and multimedia"

Competitive concurrency

- illusion of time sharing
- increased overall throughput

Cooperative concurrency

- provide structure
- potential for parallelism

Systems Programming [Montague and McDowell, 1997]

Proposed: 'soft-instruction' architecture

Reject in favour of Esterel semantics:

- Synchronous broadcast communication
- Constructive causal ordering

Keep:

- non-blocking *request* and *completion* events
- context-structures?

Concerns over context-switching costs:

- Take advantage of micro-kernel optimizations
- Trade-off: safety (i.e. memory protection) against cost

Presentation Outline

- √ Reliable I/O Framework
- √ Possible Architectures
- √ Strictly event-driven modules
- √ Event-driven scheduling
- √ Example: IP Stack and Ethernet driver
- √ Other related work
- ⇒ Summary

Summary

Application: Device driver framework in Esterel

Single idea: Esterel modules that block waiting for the occurrence of events (like a normal OS)

Two Concepts:

1. Strictly event-driven modules
   (for reasoning and efficiency)
2. Two level structure
   (trade dynamic cost against memory safety)

Remaining problems:

- scheduling algorithm
- signal naming and mapping
- mismatch with L4 primitives
- more examples (requests with state)
- ...

Some of SYNCHRON so far...

Dr Caspi	Start simply (and the blackboard)
Dr de Roever	Reliability of components is (only) part of the problem
Dr von Hanxleden	Engineering of systems based on Esterel
Dr Maraninchi	Forget as much as possible (everything but events)
Dr Tardieu	Consider SHIM? (interactions with environment)

Some of SYNCHRON so far...

Dr Caspi	Start simply (and the blackboard)
Dr de Roever	Reliability of components is (only) part of the problem
Dr von Hanxleden	Engineering of systems based on Esterel
Dr Maraninchi	Forget as much as possible (everything but events)
Dr Tardieu	Consider SHIM? (interactions with environment)
Dr Halbwachs	Don't use Esterel!

References

[Amadio, 2005] Amadio, R. M. (2005). The SL synchronous language, revisited. *arXiv:cs:PL/0511092v1*, 0(0):0–0.

[Berry, 1989] Berry, G. (1989). Programming a digital watch in Esterel v3. Rapport de recherche 1032, Institut National de Recherche en Informatique en Automatique, Sophia Antipolis.

[Berry, 1999] Berry, G. (1999). *The Constructive Semantics of Pure Esterel*. `ftp://ftp-sop.inria.fr/meije/esterel/papers/constructiveness3.ps`, draft book, current version 3.0 edition.

[Berry, 2000] Berry, G. (2000). *The Esterel v5 Language Primer*. Ecole des Mines and INRIA, version 5.92 edition.

[Berry and Gonthier, 1989] Berry, G. and Gonthier, G. (1989). Incremental development of an HDLC protocol in Esterel. Rapport de recherche 1031, Institut National de Recherche en Informatique en Automatique, Sophia Antipolis.

[Berry and Gonthier, 1992] Berry, G. and Gonthier, G. (1992). The Esterel synchronous programming language: Design, semantics, implementation. *Science of Computer Programming*, 19(2):87–152.

[Berry and Sentovich, 2001] Berry, G. and Sentovich, E. (2001). Multiclock Esterel. In Margaria, T. and Melham, T. F., editors, *Proc. 11th Advanced Research Working Conference on Correct Hardware Design*

and Verification Methods, volume 2144 of *Lecture Notes in Computer Science*, pages 110–125, Livingston, Scotland. Springer-Verlag.

[Boussinot, 1991] Boussinot, F. (1991). Reactive c: An extension of c to program reactive systems. *Software Practice and Experience*, 21(4):401–428.

[Boussinot and de Simone, 1996] Boussinot, F. and de Simone, R. (1996). The SL synchronous language. *IEEE Trans. Software Engineering*, 22(4):256–266.

[Boussinot and Susini, 1997] Boussinot, F. and Susini, J.-F. (1997). The SugarCubes tool box. Rapport de recherche 3247, Institut National de Recherche en Informatique en Automatique.

[Caspi, 2001] Caspi, P. (2001). Embedded control: From asynchrony to synchrony and back. In Henzinger, T. A. and Kirsch, C. M., editors, *Proc. 1st International Conference on Embedded Software (EMSOFT'01)*, volume 2211 of *Lecture Notes in Computer Science*, pages 80–99, Tahoe City, USA. Springer-Verlag.

[Castelluccia et al., 1996] Castelluccia, C., Dabbous, W., and O'Malley, S. (1996). Generating efficient protocol code from an abstract specification. *ACM SIGCOMM Computer Communication Review*, 26(4):60–72.

[Chou et al., 2001] Chou, A., Yang, J.-F., Chelf, B., Hallem, S., and Engler, D. (2001). An empirical study of operating systems errors. In *Proc. 18th Symposium on Operating Systems Principles*, pages 73–88.

[Edwards and Lee, 2003] Edwards, S. A. and Lee, E. A. (2003). The semantics and execution of a

synchronous block-diagram language. *Science of Computer Programming*, 48:21–42.

[Halbwachs and Baghdadi, 2002] Halbwachs, N. and Baghdadi, S. (2002). Synchronous modeling of asynchronous systems. In Sangiovanni-Vincentelli, A. L. and Sifakis, J., editors, *Proc. 2nd International Conference on Embedded Software (EMSOFT'02)*, volume 2491 of *Lecture Notes in Computer Science*, pages 240–251, Grenoble, France. Springer-Verlag.

[Jagadeesan et al., 1995] Jagadeesan, L. J., Puchol, C., and Olnhausen, J. E. V. (1995). A formal approach to reactive systems software: A telecommunications application in Esterel. In *Proc. Workshop on Industrial-Strength Formal Specification Techniques*, pages 132–145, Florida, USA. IEEE.

[Liedtke, 1995] Liedtke, J. (1995). On μ-kernel construction. In *ACM Symposium on Operating System Principles (SOSP)*, pages ?–?, Colorado, USA.

[Montague and McDowell, 1997] Montague, B. R. and McDowell, C. E. (1997). Synchronous/reactive programming of concurrent system software. *Software Practice and Experience*, 27(3):207–243.

[Potop-Butucaru, 2002] Potop-Butucaru, D. (2002). *Optimizations for Faster Simulation of Esterel Programs*. PhD thesis, École des Mines de Paris.

[Tardieu and de Simone, 2005] Tardieu, O. and de Simone, R. (2005). Loops in Esterel. *ACM Trans. Embedded Computing Systems*, 4(4):708–750.