Sverres lecture notes in TTK3 An introduction to formal verification

Sverre Hendseth

September 2, 2019

Contents

1 Day 1: Intro, FSP modelling and the LTSA tool

1.1 Course Intro

- Modelling of concurrent processes
  - (Program is too detailed to be useful)
- Focus on process interaction
- Using CSP, developed by Hoare from 1978.
- Purpose ? Deduce properties (like correctness :-))

Check background of students: Semaphores ? Monitors ? Race Condisions

Mature field ? Bordering, can still only be done by PhDs, but necessary!. For more and more sw, arguing correctness becomes necessary.

Motivating examples:

- Autronica: Fire alarm network.
- SIEMENS: Distributing routing tables in unreliable underwater network.

Slide: Learning Outcomes

.Terms:

- CSP, Communicating Sequential Processes: "Branch of mathematics" by Hoare
- FSP, Finite State Process: Subset of CSP
- LTSA, Labeled Transition System Analyzer: Java Program that reads and analyses FSP-modeller.

Slide: Curriculum

1.2 Plan for today: Practical FSP, LTSA.

By Today: Excersice: Model dining philosophers problem.

1.3 Slide: Deadlock.

```
T1 = (t1wa -> t1wb -> t1sb -> t1sa -> T1).
T2 = (t2wb -> t2wa -> t2sa -> t2sb -> T2).
SA = (t1wa -> t1sa -> SA
   |  t2wa -> t2sa -> SA).
SB = (t1wb -> t1sb -> SB
   |  t2wb -> t2sb -> SB).

||SYSTEM = (T1 || T2 || SA || SB).
```

FSP/CSP syntaks:

- **Processes** that participants in **events** (Large and small initial letters)
- events are global! (This is not an imperative language)
- prefixing "->"
- recursion
- Choice "|"
- Parallel processes "||"

Draw/deduce transition diagram...
Where is the deadlock ?
Note: No difference between (passive) semaphore and active threads!

1.4 And btw. Lets cover Livelock (Assignment):

FSP/CSP syntaks:

- Partial processes ","
- Progress: "progress pName = {set of events}": At least one in the set can always be reached.

```
P = (a->P2),
P2 = (b -> c -> P2).
```

and add to fix it:

```
progress N1 = {b}
progress N2 = {c}
```

Terms:

- Deadlock: A state we cannot leave (STOP)
- Livelock: A subset of states we cannot leave (DIV)
- progression: the absense of livelocks
- liveness: what should happen happens sooner or later.
- safety: Something bad never happens
- ... (all kinds of logic: "is it true that for all states... after this event have happened...")
- Model checking: does the model of the implementation correspond to the model of the specification?

1.5 Slide: Bounded buffer.

```
SEM(N=1,MAXN=1) = SEM[N],
SEM[n:0..MAXN]
      = (when (n<MAXN) signal->SEM[n+1]
        |when (n>0) wait->SEM[n-1]
        ).

PUTTER = (nFree.wait -> putter.mutex.wait ->
          putter.mutex.signal -> nInBuffer.signal -> PUTTER).
GETTER = (nInBuffer.wait -> getter.mutex.wait ->
          getter.mutex.signal -> nFree.signal -> GETTER).

||SYSTEM(S=3) = (PUTTER || GETTER || nFree:SEM(S,S) ||
                 nInBuffer:SEM(0,S) ||{getter,putter}::mutex:SEM(1,1)).
```

1.6 The modelchecking example:

```
SEM(N=1,MAXN=1) = SEM[N],
SEM[n:0..MAXN]
      = (when (n<MAXN) signal->SEM[n+1]
        |when (n>0) wait->SEM[n-1]
        ).

PUTTER = (nFree.wait -> putter.mutex.wait -> put ->
          putter.mutex.signal -> nInBuffer.signal -> PUTTER).
GETTER = (nInBuffer.wait -> getter.mutex.wait -> get ->
          getter.mutex.signal -> nFree.signal -> GETTER).

||SYSTEM(S=3) = (PUTTER || GETTER || nFree:SEM(S,S) ||
                nInBuffer:SEM(0,S) ||{getter,putter}::mutex:SEM(1,1)).

property
 BUF(N=5) = BUF[0],
 BUF[n:0..N] = (when n < N put -> BUF[n+1]
              | when n > 0 get -> BUF[n-1]).

||SAFETY = (BUF(4) ||SYSTEM(3)).
```

FSP/CSP syntaks:

- Parameters "P(N=5)": Just a constant
- indexes "[]": Part of the name
- types and ranges: n:0..5
- if & when
- Labeling: a:SEM: Exchanges all event names (wait -> a.wait)
- Sharing: {a,b}::SEM : Both a.wait and b.wait becomes possible
-
- Relabeling "/" - change name of one event
- constants, sets
- properties: a partial model that the system is checked against
- Trace: A possible sequence of events
- Alphabet: The set of events a process participates in. (can be manipulated by @ and \ )

1.7 Prepare exercise: Dining Philosophers.

Model and find the deadlock.

Implement one strategy for solving the deadlock, prove that it works.

2 Day 2

2.1 From last time.

- Some FSP syntax, modelling the deadlock, and the bounded buffer.
- Deadlock, livelock, progression, liveness, safety…
- modelchecking, "property".
- The bounded buffer of 3 satisfies the property of 5!
- Dining philosophers?

Ref. point 4: What are we really checking ?

2.2 Plan for the day

- Sverre the prophet talks. (More context)
- A hard to detext bug.
- Starting the book.

2.3 Huge sidestep: Why do we have too many states in the bounded buffer example ?

(Demo again of bounded buffer, changing size and noting states) Why is there too many states in the semaphore solution of the bounded buffer compared to the n+1 we could have expected ? (brainstorm)

- Caused by the way we model ?
- Have we modeled on a too low level of abstraction ?
- Something is wrong with our implementation ?
- Semaphores not suited for making bounded buffer ? (What is, then?)
- ???

Slide: Sverres Hypothesis: This hurts maintainability. (We can barely see wether a 15 line program is correct)

- Can we implement this system so that the number of states gets to be N+1, 3 or at least lower than 8+n*15 ?
- Could we imagine higherlevel language primitives that yielded simpler models ?
- Could we imagine implementational patterns that had deserved simpler modelling ? (Yes, "bounded buffer", of course, but more? :-))

2.4 How do we program so that the models get simple ?

We whish for **events** that more threads participates in: (Brainstorm)

- Ada entries (synch tow-way comm++)
- Synchroneous communication !
- barriers
- twophase commit protocol
- ...

Lets for now settle for the simples of these: Synchroneous communication
A new way of defining a process follows:

- Process: Participates in events

- event: synchroneous communication (asynch communication can be made with bounded buffers)
- Composition: More processes constitutes a process (Draw figure to illustrate this)
  - Internal states disappear
  - Internal communication disappears
- => Very nice scalability
- The primitive processes are tiny
  - => non preemptive scheduling becomes ok
  - (and by non-preemptive scheduling a semaphore is just a flag)
  - => small overhead and few demands on scheduler.
- No race condisions

(What? No shared variables ?)

(NB: Does not solve deadlock problem, but it gets smuch simpler since we have fewer states - and we have good patterns.)

The module, demonstrate scalablilty by enclosing more communicating processes in one larger in a communication diagram.

OCCAM/INMOS history...

- Choosing the simplest event: Synchronous communication
- OCCAM died with transputer
- People: Focused on other things than sw maintenance and scalability.

How would we implement a two-way event with semaphores ?

```
T1(){                T1(){
  c = 10;               wait(t1Ready)
  signal(T1Ready)       var = c;
  wait(t2Ready)         signal(T2Ready)
}                    }
```

(Enter the cs afterwards.)

(Giving OCCAM here since it corresponds to the books syntax)

OCCAM syntax

- ch ! c - writes the value of c to the channel ch
- ch ? c - reads c from the channel c
- The semaphore and the variable: **A channel**

2.5 Slide A Messagebased implementation of bounded buffer

(Should make the slide use valid Go syntax.)

How many states ? N+1

The module can be maintained without knowing the rest of the system! It decides for itself what it takes part in (as different from a classical C module or C++ object.):

There is no doubt that rest of the program cannot make this module inconsistent.

Modelling again:

- A process is described by the alternative sequences of events it may take part in.

3 Day 3

3.1 Plan for the day

- Chapter 1: Introducing refinement

3.2 Summary and Slide on RT Properties of messagebased systems

- Remember implicit vs explicit, communication vs. synchronization/shared variables, small processes (not necessarily preemptive). Excelent scalability. You can convince yourself that a process is correct without knowing the rest of the system.
- Why do we not use messagebased systems more ?
  - HW is not built that way
  - Culture, lack of education.
  - "messages" usually used with "networks"
  - Some infrastructure missing; (unless Occam or Go)
  - Bad RT properties (ref ADA Ravenscar)

... But to use multicore/multiprocessor systems, and handle the increasing complexity of sw, we have to...

Advice for a programmer:

- Step 1: Think about building program with server threads and messages
  - "While(true) select()..."
  - Use lots of such processes.
  - prosesses is a better way than objects for dividing the system into modules.

Step 2: Skip buffered communication (Block on write!) Step 3: Choose fine-granular processes (& preemptive scheduling...)

Slide: RT properties of messagebased systems

- Not too good; Assumption is that what can run, runs. Little control over which (of the 100000 ready processes) gets to run.
- Priorities ? One thing is that priorities over 100000 tiny processes are difficult, but also breaks connection to the models.
- Schedulability proofs ?

But we **do** have:
Tempting to set this up as an interesting research field.
But then... The HW is made for shared resource systems...

3.3 What you havent learned.

Slide: Burns & wellings bug
How could this be detected ?

- Find the bug:
- How would the property look ?
- How could we check, with ltsa, this program in a way where we detect the error ? (Very confusing.)

Summary before we go to the book:

- Remember how the FSP bounded buffer property worked
  - What did we really check ?
  - and a buffer with 3 elements satisified a property buffer with 5 elements (?)
- And what we need for the hard bug ?
- Equality between processes ? Is $(P = a \rightarrow a \rightarrow P) equalto (P = a \rightarrow P)$?
  - How would we argue this ?

How **exactly** are we comparing two processes with a FSP property ?
Appreciating the buffer example (Chapter 5; 5.1) is the goal of the rest of the course.

3.4 The book

Slide: You know:

- Events; ($\in \Sigma$), global, subject to choice when modelling, Simplest SW pattern modeled is synchronous communication, but also other interactions like semaphore interaction, barriers, ...
- Processes, recursion, prefixing, Partial processes/mutual recursion
- Guarded alternatives: |
- indexes, parameters, subscripts
- $STOP$: (Does not accept more inputs - deadlock)
- Finite vs. infinite models.

3.5 More on events

New:

- "Namless recursive processes": P = F(P): $\mu p.a \rightarrow p$
- Fancy if syntax: $P \nless test \ngtr Q$
- Prefix choice: $?x : A \rightarrow P(x)$

  Ex: $REPEAT = ?x : \Sigma \rightarrow x \rightarrow REPEAT$

- Special event syntax for channels: $channel.TypeSet- > (A = [channel.t|tinTypeset])$
- For convenience: Channel?x:TypeSet -> P(x)
- And: Channel!x == Channel.x ...
- ref OCCAM syntax!
- Also "multiple data transfers" in one event...

  Slide: Example: Buffer page 19.

- Scope of x in $c?x \rightarrow P$ implicitly $P(x)$

3.6 Nonderterminism (two types of choice)

External choice $\square$: $P\square Q$: The environment decides whether the process behaves as P or Q on which event is offered first

$(a \rightarrow P\square b \rightarrow Q) \equiv (a \rightarrow P|b \rightarrow Q)$

Internal (nondeterministic) choice $\sqcap$: $P \sqcap Q$. The process may behave either as P or Q.

Typically this enters our models when abstracting, hiding events (Chapter 3): Ex. The minibank, where the comminucation with bank database is hidden, leaving us with a model that either swallows the card or not.

Btw. $(a \rightarrow P\square a \rightarrow Q) \equiv (a \rightarrow (P \sqcap Q))$

Plus set versions of these... -> Unbounded nondeterminism.

3.7 Refinement

If $R = R \sqcap P$ then P *is more deterministic* than R: P *refines* R.

$P \sqsupseteq R$

- We have a way of comparing processes.!
- But we do not know how to check this yet.
- Is the Buffer3 a refinment of Buffer5 ?

3.8 A few important processes (STOP,RUN,Chaos,div,SKIP)

- $STOP$
- $RUN_A = ?x : A \to RUN_A$ — Accepts anyting in A in any order.
- \$Chaos$_{\mathrm{A}}$ $=$ STOP $\sqcap(?x : A \to$ Chaos$_{\mathrm{A}})$
- div: Chapter 3
- $SKIP$: Chapter 6

3.9 Algebra

- Commutative
- symmetry
- associative
- distributive
- unit
- zero
- idempotence
- step

Slides: p30, Basic Rules I and II
Recursion is not distributive.
Compare $\mu\ p.((a \to p) \sqcap (b- > p))$ with $(\mu\ p.a \to p) \sqcap (\mu\ p.b \to p)$
1.14: As training, reading a step-law :-)
Unit law for external choice: stop.
And then: Be aware of Variable Bindings: Example p. 34
and Mu-unwind: $\mu\ p.P = P[\mu\ p.P/p]$

4 Day 4

4.1 Slide Refinement.

4.2 Traces

Slide: Traces Basic Rules.

- traces(P): Complete set of all sequences of events that a process might do
- concatenation operator ˆ
- "prefixed closed"
- All traces have at least the empty element! (No "infinite execution" assumption like in FSP progress test.)

- Traces of internal and external choice is indistinguishable. (Rules p37) (trace equality is weaker than equality)
- $traces(P) = traces(P \sqcap STOP)$

Recursion often yield infinite trace sets... How to compare?

- Handling recursion through X=F(X) and "least fixed points". (P = F(P) -> traces(P) = G(traces(P))) <- for guarded recursions the least fix point is unique; Proof consist basicly of comapring structure of G functions (P38)

4.3 Trace refinments

- Useful part of specifying sw: Put demands on Traces set.
- (Basicly any sorts of specification): R(tr) - a requirement
- #s: Length of trace.
- s up A: The trace restricted to A. (Just skipping all other elements)
- s down c: The number of occurrences of c in s. c may be a channel.

Ex. tr down signal >= tr down wait
(Some proof rules are given p. 44...)

- *STOP* satisfies all!
- One process P is "the largest" of all processes satisfying a trace specification (p45)
- $P =_T \sqcap\{Q|QsatR(tr)\}$
- Q sat R(tr) -> traces(Q) subset of traces(P)
- Q is a traces-refinment of P!

And now, what about Buf3 vs. Buf5 ?
And what is lacking ?

- Refinment is transitive. We can do gradual refinement
- After(P/s)
- initials(P)

4.4 Parallel operators: Synchronous parallel

Synchronous parallel ||. All events are shared:
Example: $REPEAT||(a \to REPEAT)$

-step law: $?x:A → P   ?x:B -> Q = ?x : A∩ B -> (P   Q)$

It this like || in FSP? (Yes or no ?)
Write out REPEAT: Example of LFP rule.
$traces(P||Q) = (traces(P) \cap traces(Q))$ Easy.
But: P ≠ (P||P)

4.5 Alphabetized parallel

$P\ _X =_Y Q$ must agree on $X \cap Y$
All alphabet is in
$P = ?x : A \to P'$
$Q = ?x : B \to Q'$
$C = (A \cap (X \setminus Y)) \cup (B \cap (Y \setminus X)) \cup (A)$
Then
$_X||_Y - step$: (p57)
(Big version exists)
$traces(P_X||_Y Q)$ p60

4.6 Interleaving (completely independent processes. . .)

Operator: |||
Slide: |||-step
$P = up \to down \to P$
How do $P|||P$ behave ?
What about: $C = up \to (C|||down \to STOP)$
(Counting semaphore one-liner :-))
Defines also ||| for traces. P67 (Possible syntax training.)

4.7 Generalized parallel

Covers all the others (++): $P \underset{<interface>}{||} Q$

- $(P|||Q) = P \underset{<>}{||} Q$
- $(P\ _X||_Y Q) = P \underset{<X \cap Y>}{||} Q$
- $(P||Q) = P \underset{<X \cup Y>}{||} Q$

But also:
How does $P \underset{<up>}{||} P$ behave if $P = up \to down \to P$?

5 Day 5

5.1 Repeat progress so far:

- The FSP tool. We can prove things. (but what, exactly ?)
- A bit of math, refinement as a way of comparing processes.
- traces, and trace refinments (BUF_3 trace-refines BUF_5)
- TracesRequirements and Paralell operators as a way of **specifying** processes.
-

5.2 Parallel composition as a way to limit a specification

How do we make a bounded buffer from COUNT$_0$? Want to add a max count.

- $LIMIT(a,b)_0 = a \rightarrow LIMIT(a,b)_1$
- $LIMIT(a,b)_r = a \rightarrow LIMIT(a,b)_{r+1} \Box b \rightarrow LIMIT(a,b)_{r-1}$
- $LIMIT(put,get)_0 \underset{<get,put>}{||} LIMIT(get,put)_5$

Other example - compose with stop....

5.3 Hiding events

- $P \setminus X$: Events in X is replaced with unobservable "τ"
- Abstraction
- Scope

What about $(a \rightarrow P \Box b \rightarrow Q) \setminus \{a,b\}$?
(P\ \a,b\ $\sqcap Q$\ \a,b\)

- We have introduced nondeterminism!

But then, $(a \rightarrow P \Box b \rightarrow Q) \setminus \{b\}$?
$((a \rightarrow (P \setminus b)) \sqcap STOP) \Box (Q \setminus b)$

- Timeout, defining timeout operator for $(P \sqcap STOP \Box Q) = P \triangleright Q$
- Of course strange in an untimed framework :-)
- Handling nondeterminism is necessary...
- Also introduce livelock! - divergence! div $((\mu p.a \rightarrow p) \setminus a = \mu p.p)$
- The fixedpoint rule demanded guardedness, but hiding destroys guards...
- (discussing $P = a \rightarrow (P \setminus a)$)

5.4 Renaming

- Assume $f : \Sigma \rightarrow \Sigma$: P' = f[P]
- if f is one-to-one (injective), then f[P] {*behaves similar} to P*
- *We have seen this from FSP*
- *Also - the LIMIT over could be much simplified.*
- *Process Renaming: a.P, short for f[P] where $f : x \rightarrow a.x$*
- *"All laws still valid for processes... "*
- *Noninjective functions? (many to one)*
- *(We can do much harm here :-))*

- *A useful example: The split p88*
- *Relational renaming (one to many)*

What kinds did we have in FSP ?

- Process sharing from FSP / renaming / prefixing
- $P[[R]]$
- $P[[a/b]]$: b is replaced by a
- Use . semaphores that can be used by many (book gives worse example. . .)
- There is potential expressive power here!
- Example of combination sharing + composition.

5.5 Closing in on refinement

- Trace refinement told what a process {*could} do*
- *Weakness: STOP was a tracerefinement of all!*
- *We need failures; (. . . to participate in events) the implementation should not have failures that the spec has not.*
- *refusals(P): The set of events that P can refuse.*
- *failure: (s,X) where s is a trace and X is refusals(P/s)*

Relevant for the BUF5/BUF3 example from FSP example.
The four examples 94-95:
$P_1 = (a \to b \to STOP) \Box (b \to a \to STOP)$
$traces(P_1) = <>, <a>, <b>, <a,b>, <b,a>$
$failures(P_1) = (<>, c), (<a>, a, c), (<b>, b, c), (<a,b>, a, b, c), (<b,a>, a, b, c)$
$P_2 = (x \to a \to STOP) \Box (b \to x \to STOP)\ x$
. . . discussing {*stable} nodes here. . .*
$traces(P_2) = <>, <a>, <b>$
$failures(P_2) = (<>, b, c), (<a>, a, b, c), (<b>, a, b, c)$
$P_3 = (a \to STOP) \sqcap (b \to STOP)$
$traces(P_3) = <>, <a>, <b>$
$failures(P_3) = (<>, a, b, c), (<a>, a, b, c), (<b>, a, b, c)$

5.6 Failure-refinement

A process Q failure-refines P iff

$traces(Q) \subseteq traces(P)$ *and* $failures(Q) \subseteq failures(P)$

Q can neither accept an event, or refuse one unless P does.

$P_2 \sqsupseteq_F P_3$ *P_3 {***may} refuse a at the beginning (while it also may accept). $\mathbf{P_2}$ can not refuse a at the beginning. We see that the "$\mathbf{P_2}$ is more deterministic than $\mathbf{P_3}$" holds.**

We now differ between external and internal choice!

$P_4 = (c \to a \to STOP) \Box (c \to b \to STOP)$
$traces(P_4) = <>, <c>, <c,a>, <c,b>$
$failures(P_4) = (<>, a, b), (<c>, a, b, c), (<c,a>, a, b, c), (<c,b>, a, b, c)$
(Defining failure-equivalence $\mathbf{P =_F Q}$)

5.7 Divergences

We are close now, what more is needed ?

What happens at "unstable nodes" ? No trace or failures are describing this. . .

If P refines Q: P should only be allowed to diverge when Q diverges.

(For completeness ? A bit contrieved, since who makes specifications that diverges ?)

Cite two sentences at page 96.

div allows all traces, denies everything, and diverges anywhere.

Everything refines div.

(Futile to analyze anything inside of a divergence; may get out of it, but may always not.)

P refines Q == P failure/divergence-refines Q

Also final definition of process equality.

A deterministic process: divergences(P) = {} and if s ˆ <a> $\in$ traces(P) then (s,{a}) $\notin$ failures(P)

. . . A deterministic process cannot be refined (further)

. . . our FSP example again. We have asked ourselved a equality question.

6 Day 6

6.1 Summary:

- **P refines Q: P (traces/)failures/divergence-refines Q**
- **Process equality: Processes are equal if they refine each other.**

How do we *specify*?

- **Demands on traces: Leads to characteristic process**
- **Put restricting process in paralell**
- **internal choice with stop: The implementation does not need to offer this.**
- **div: From now on the process may do whatever.**

We are at page 100, starting chapter 4 "Piping and enslavement". Goal is; The example in 5.1.

6.2 piping operator. >> Easy variant.

$P >> Q = (P[[mid/right]] \underset{mid}{||} Q[[mid/left]]) \setminus \{mid\}$

- **Convention: The difference between input and output**
- **($P >> Q >> R$ is more difficult than it seems)**
- **Notice the implicit buffering & timing decoupling**

- **Two basic applications of the operator: Pipelining (Data and even type may change) and communication (data and order should not change. Should at least trace-refine $B^{\infty}_{<>}$).**
- **If P and Q are buffers, then $P >> Q$ are also!**
- **Syntax training: Cool application of operator, no mutual recursion $B^{+} = left?x \rightarrow (B^{+} >> right!x \rightarrow COPY)$ (Proof difficult because of the hiding removing guardedness...)**
- $>> -step$: **p103. Write 1 and 3 for use later.**

6.3 (Enslavement - skip this)

- $\mathbf{P/_{Y Q} = (P\ \_\sigma\ ||_Y\ Q)}$

(Not used)

6.4 Ch 5: The specification of a buffer

- **Our problem: Show that $P >> E >> Q$ is a buffer, where E is *not* a buffer (e.g. unreliable).**
- **Example 5.1.1 Our E may switch at most one of 3 bits.**

Classroom task: model E: Make a model of E
Classroom task: Specify a general, unlimited buffer. (What are the demands? Implement it)

- **Trace specification too short:** $s \in traces(B) \Rightarrow \Sigma = left.T \cup right.T$ **and** $s \downarrow right \leq s \downarrow left$.
- **Problem is that this is satisfied by** $STOP$, $\mu p.left?x \rightarrow p$, $\mu p.left.0 \rightarrow right.0 \rightarrow p$, $STOP \sqcap COPY$
- **Possibly; Write ii and iii on whiteboard from page 116.**
- **But anyway: The characteristic process p117. Notice the stop that allows the buffer to be of limited size.**
- **So: Any process that refines $BUFF$ is a Buffer!**
- **But $BUFF$ is infinite... Not suited for automatic checking.**
- **Strategy: Try to refine $BUFF^N$. If that works, then ok.**
- **But if not we have two possibilities: No buffer or N not big enough...**
- **Show the weak buffer p118 as example of div used for specification!**
- **Whether we refine $WBUF$ tells us if size of N is the problem or not.**

6.5 The buffer laws

- **BL1: If P and Q is buffers then so is $P >> Q$**
- **BL2: If $P >> Q$ and P is buffers, so is Q**
- **BL3: If $P >> Q$ and Q is buffers, so is P**
- **BL4: If $P >> Q$ is a buffer, then $left?x \to (P >> right!x \to Q)$ is also.**
- **BL5: If $P >> Q \sqsupseteq left?x \to (P >> right!x \to Q)$ then $P >> Q$ is a buffer**

BL5 is useful if we need to prove that two non-buffers together is a buffer.

6.6 The simple example. P and Q page 120.

. . . is a buffer by buffer law 5.

6.7 Example 5.1.1

The faulty communication process: Can fail at most every 3rd time.
$E_0 = left?x \to (right!x- > E_0 \sqcap right!(1-x) \to E_2)$
$E_n = left?x \to right!x \to E_{n-1}$

- **(Notice btw: $E_0 \sqsubseteq E_1 \sqsubseteq E_2$)**

To make a buffer $S >> E_0 >> R$ we can use:
$S = left?x \to right!x \to right!x \to right!x \to S$
$R = left?x \to left?y \to left?z \to right!(x \not< x = y \not> z) \to R$
Using the same $>> -step$ functions as in the small example we arrive at 3rd last equation on page 122.

- **See that all R^{xxx}'s are $right!x > R$**
- **See that $E_0 \sqcap E_1 \sqcap E_2 \sqcap E_0 = E_0$ $**
- **We are ready to apply (even a weak version of) Buffer Law 5.**

One more exercise: Do this proof in LTSA! Why does it work ? Is this proof equally good ?

7 Skipped.

7.1 Dividing communication protocols into layers

Task:

- **T»M»R: where M is unrelable: design T and R to achieve reliability.**
- **T=T_1»T_2 and R=R_2 » R_1: B = T_2»M»R_2 could solve part of the problem, where T_1»B»R_1 is simpler. Layerd communication protocols is what we usually have.**

8 Latex symbols

"p24 i The comprehensive Latex symbol list"
mathaqbx $\ngtr\nless$