NTNU
Norwegian University of Science and Technology

Modeling of concurrent programs

Sverre Hendseth
Department of Engineering Cybernetics

Modelling of Concurrent Programs

Course Introduction

FSP/LTSA

Learning Outcomes

- Understanding how CSP is used for modelling concurrent programs.
- Understanding how CSP constitutes the theoretical basis for (tools like) LTSA.
- Ability to model simple programs in FSP for checking in LTSA.
- Ability to perform simple deductions on such models using CSP (on paper).
- Ability to model simple programs by drawing their transition diagrams.
- A good understanding of what CSP and corresponding tools can be used for, and not.
- Understanding how modelchecking can be used in a developement workflow.

Curriculum

— Roscoe: The theory and Practice of concurrency. Part 1, 150 pages.

— On LTSA: Reference and example programs.

Deadlock!

```
thread1(){
    while(1){
        Wait(A);
        Wait(B);
          ...
        Signal(B);
        Signal(A);
     }
  }
```

```
thread2()
    while(1){
        Wait(B);
        Wait(A);
        ...
        Signal(A);
        Signal(B);
    }
}
```

The Bounded Buffer

```
put(e){
  wait(NFree);
  wait(Mutex);
    // enter e into buffer
  signal(Mutex);
  signal(NInBuffer);
}
```

```
get(e){
   wait(NInBuffer);
   wait(Mutex);
       // get e from buffer
   signal(Mutex);
   signal(NFree);
}
```

NFree and NInbuffer are counting semaphores, Mutex is binary.

The FSP Bounded Buffer Model

```
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)).
```

What we can check

- **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?

Checking the Bounded Buffer: The Property

```
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)).
```

Sverres Hypothesis

— Systems that are simple to model - yields simple models - will be simple to maintain.

Modules

- You should be able to maintain a module without knowing the rest of the program
- You should be able to maintain the rest of the program without knowing the modules internals
- ... and we want *composition*; that supermodules can be made from submodules.

A Messagebased implementation of bounded buffer

```
process
boundedBuffer(channel put,get){
  while(1)
    select{
      n<N: put ? c; store c; n++;
      -- Assuming no selection on output...
      n>0: get ? dummy; retrieve c; get!c; n--;
    }
  }
}
```

Real-Time properties of messagepassing 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 ?

A hard-to-find bug

```
void allocate(int priority){
  Wait(M);
  if(busy){
    Signal(M);
    Wait(PS[priority]);
  }
  busy=true;
  Signal(M);
 }
```

```
void deallocate(){
  Wait(M);
  busy=false;
  waiting=GetValue(PS[1]);
  if(waiting>0) Signal(PS[1]);
  else{
    waiting=GetValue(PS[0]);
    if(waiting>0) Signal(PS[0]);
    else{
      Signal(M);
    }
  }
}
```

You know from FSP

- 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.

A buffer

$$
\begin{aligned}
B^{\infty}_{\langle\rangle} &= left?x : T \rightarrow B^{\infty}_{\langle x \rangle} \\
B^{\infty}_{s^\frown\langle y\rangle} &= (left?x : T \rightarrow B^{\infty}_{\langle x\rangle^\frown s^\frown\langle y\rangle} \\
&\quad \mid right!y \rightarrow B^{\infty}_{s})
\end{aligned}
$$

Some Laws

$$P \Box P = P \qquad \langle \Box\text{-idem} \rangle$$

$$P \sqcap P = P \qquad \langle \sqcap\text{-idem} \rangle$$

$$P \Box Q = Q \Box P \qquad \langle \Box\text{-sym} \rangle$$

$$P \sqcap Q = Q \sqcap P \qquad \langle \sqcap\text{-sym} \rangle$$

$$P \Box (Q \Box R) = (P \Box Q) \Box R \qquad \langle \Box\text{-assoc} \rangle$$

$$P \sqcap (Q \sqcap R) = (P \sqcap Q) \sqcap R \qquad \langle \sqcap\text{-assoc} \rangle$$

Some Laws II

$$P \,\Box\, (Q \sqcap R) = (P \,\Box\, Q) \sqcap (P \,\Box\, R) \qquad \langle \Box\text{-dist} \rangle$$

$$P \,\Box\, \bigsqcap S = \bigsqcap \{P \,\Box\, Q \mid Q \in S\} \qquad \langle \Box\text{-Dist} \rangle$$

$$a \rightarrow (P \sqcap Q) = (a \rightarrow P) \sqcap (a \rightarrow Q) \qquad \langle \text{prefix-dist} \rangle$$

$$a \rightarrow \bigsqcap S = \bigsqcap \{a \rightarrow Q \mid Q \in S\} \qquad \langle \text{prefix-Dist} \rangle$$

$$?x : A \rightarrow (P \sqcap Q) = (?x : A \rightarrow P) \sqcap (?x : A \rightarrow Q) \qquad \langle \text{input-dist} \rangle$$

$$?x : A \rightarrow \bigsqcap S = \bigsqcap \{?x : A \rightarrow Q \mid Q \in S\} \qquad \langle \text{input-Dist} \rangle$$

More Syntax Training

$$
\begin{aligned}
(?x : A \rightarrow P) \,\square\, (?x : B \rightarrow Q) \;=\; ?x : A \cup B \rightarrow \quad & ((P \sqcap Q) \\
& \lessdot x \in A \cap B \gtrdot \qquad \langle \square\text{-step} \rangle \\
& (P \lessdot x \in A \gtrdot Q))
\end{aligned}
$$

The problem of scoping

$$?x : \mathbb{N} \rightarrow ?x : \mathbb{N} \rightarrow (P \lessdot x \text{ is even} \gtrdot Q) \neq$$

$$?x : \mathbb{N} \rightarrow ((?x : \mathbb{N} \rightarrow P) \lessdot x \text{ is even} \gtrdot (?x : \mathbb{N} \rightarrow Q))$$

Refinement

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

Traces Basic Laws (p37)

1. $traces(STOP) = \{\langle\rangle\}$
2. $traces(a \rightarrow P) = \{\langle\rangle\} \cup \{\langle a \rangle\hat{\ }s \mid s \in traces(P)\}$ – this process has either done nothing, or its first event was a followed by a trace of P.
3. $traces(?x : A \rightarrow P) = \{\langle\rangle\} \cup \{\langle a \rangle\hat{\ }s \mid a \in A \wedge s \in traces(P[a/x])\}$ – this is similar except that the initial event is now chosen from the set A and the subsequent behaviour depends on which is picked: $P[a/x]$ means the substitution of the value a for all free occurrences of the identifier x.
4. $traces(c?x : A \rightarrow P) = \{\langle\rangle\} \cup \{\langle c.a \rangle\hat{\ }s \mid a \in A \wedge s \in traces(P[a/x])\}$ – the same except for the use of the channel name.
5. $traces(P \,\square\, Q) = traces(P) \cup traces(Q)$ – this process offers the traces of P and those of Q.
6. $traces(P \sqcap Q) = traces(P) \cup traces(Q)$ – since this process can behave like either P or Q, its traces are those of P and those of Q.
7. $traces(\bigsqcap S) = \bigcup\{traces(P) \mid P \in S\}$ for any non-empty set S of processes.
8. $traces(P \lessdot b \gtrdot Q) = traces(P)$ if b evaluates to $true$; and $traces(Q)$ if b evaluates to $false$.[7]

$|||$-step

$$
\begin{aligned}
P|||Q = {?x} : A \cup B \rightarrow & (P'|||Q) \sqcap (P|||Q') \\
& \nless x \in A \cap B \ngtr \\
& (P'|||Q) \nless x \in A \ngtr (P|||Q')
\end{aligned}
\tag{1}
$$