Concurrent Computing

Lecturers: Prof. Majid Mirmehdi (majid@cs.bris.ac.uk)
Dr. Tilo Burghardt (tilo@cs.bris.ac.uk)
Dr. Simon Hollis(simon@cs.bris.ac.uk)
Dr. Daniel Page (page@cs.bris.ac.uk)

Web: http://www.cs.bris.ac.uk/Teaching/Resources/COMS20001

Lecture 10 / 11

Liveness
&
Deadlock

[Many thanks to Kerstin Eder, major parts of these lecture slides are taken from or based on materials originally prepared by her.]

Recap: Labelled Transitions and Traces

Traces can be extracted from a transition diagram!

- Labelled transitions capture same information as transition diagrams!

Relationship between traces of a process and its labelled transitions:

$$traces(P) = \{\langle\rangle\} \cup \{\langle a\rangle \frown tr \mid P \xrightarrow{a} Q \text{ and } tr \in traces(Q)\}$$

where $\frown$ denotes concatenation, i.e.

$$\langle a_1, \ldots, a_m\rangle \frown \langle b_1, \ldots, b_n\rangle = \langle a_1, \ldots, a_m, b_1, \ldots, b_n\rangle$$

Behaviour of new CSP operators can be defined through labelled transition rules. Use above relationship to calculate the traces of processes defined in terms of *new* operators.

Recap: Refusals and Failures

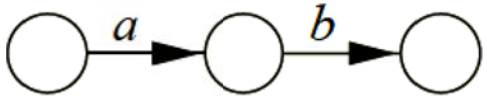
We write P/tr for the process whose behaviour is whatever P could do after the trace tr has been observed.

Failures of a process:

$$failures(P) = \{(tr, X) \mid tr \in traces(P) \text{ and } X \in refusals(P/tr)\}$$

■ $P = a \rightarrow b \rightarrow STOP$ with $\alpha(P) = \{a, b\}$

Transition Diagram of P:



$$traces(P) = \{\langle\rangle, \langle a\rangle, \langle a, b\rangle\}$$

$$refusals(P/\langle\rangle) = \{\{\}, \{b\}\}$$
$$refusals(P/\langle a\rangle) = \{\{\}, \{a\}\}$$
$$refusals(P/\langle a, b\rangle) = \{\{\}, \{a\}, \{b\}, \{a, b\}\}$$

$$failures(P) = \{(\langle\rangle, \{\}), (\langle\rangle, \{b\}), (\langle a\rangle, \{\}), (\langle a\rangle, \{a\}), (\langle a, b\rangle, \{\}), (\langle a, b\rangle, \{a\}), (\langle a, b\rangle, \{b\}), (\langle a, b\rangle, \{a, b\})\}$$

Failure Refinement I

Failure refinement is defined in a similar way to trace refinement:

$$P \sqsubseteq_F Q \text{ if and only if } failures(Q) \subseteq failures(P)$$

(Pronounce: "P is failure refined by Q")

Failure refinement in specifications:

- $SPEC = a \rightarrow b \rightarrow SPEC$
  - Use $SPEC$ with trace refinement, get a *safety* specification!
- Find some processes P which satisfy $SPEC \sqsubseteq_T P$.

$P = STOP$, $P = a \rightarrow STOP$, $P = a \rightarrow b \rightarrow STOP$, ...

- What effect has $SPEC \sqsubseteq_F P$? First, calculate $failures(SPEC)$!

Failure Refinement II

$$\begin{aligned} failures(SPEC) &= \{(\langle a,b\rangle^n \frown \langle a\rangle, \emptyset) \mid n \geq 0\} \\ &\cup \{(\langle a,b\rangle^n \frown \langle a\rangle, \{a\}) \mid n \geq 0\} \\ &\cup \{(\langle a,b\rangle^n, \emptyset) \mid n \geq 0\} \\ &\cup \{(\langle a,b\rangle^n, \{b\}) \mid n \geq 0\} \end{aligned}$$

To find out whether $SPEC \sqsubseteq_F STOP$, calculate:

$failures(STOP) = \{(\langle\rangle, \emptyset), (\langle\rangle, \{a\}), (\langle\rangle, \{b\}), (\langle\rangle, \{a,b\})\}$

Pairs $(\langle\rangle, \{a\})$ and $(\langle\rangle, \{a,b\})$ are failures of $STOP$, but not of $SPEC$. Hence, $SPEC \not\sqsubseteq_F STOP$.

Now, consider $P = a \rightarrow STOP$.

$failures(P) = \{(\langle\rangle, \emptyset), (\langle\rangle, \{b\}), (\langle a\rangle, \emptyset), (\langle a\rangle, \{a\}), (\langle a\rangle, \{b\}), (\langle a\rangle, \{a,b\})\}$

Failure pairs $(\langle a\rangle, \{b\})$ and $(\langle a\rangle, \{a,b\})$ are failures of P but not of $SPEC$; so again $SPEC \not\sqsubseteq_F P$.

Liveness (guaranteed execution of some behaviour)

$SPEC \sqsubseteq_F P$ is a *liveness* specification which requires P to do certain events.

- Which definitions of P satisfy $SPEC = a \rightarrow b \rightarrow SPEC$?
  Obviously $P = a \rightarrow b \rightarrow P$ does.

It is (in this case) the only process satisfying this specification!

(Specification is too tight; pins down implementation precisely.)

Liveness Specification Example & Hiding

Process P with alphabet $\{a, b, c\}$.

- Want to specify that P must be able to do an infinite sequence of alternating a and b events, starting with a.
- We do not care about c events.

■ Use process $ALT = a \rightarrow b \rightarrow ALT$ as before.

Allow c events to occur freely through hiding: $ALT \sqsubseteq_F (P \backslash \{c\})$

■ Definitions of P satisfying this specification include: $P = a \rightarrow b \rightarrow P$, $P = c \rightarrow a \rightarrow c \rightarrow c \rightarrow b \rightarrow P$, $P = a \rightarrow b \rightarrow c \rightarrow P$.

All are the same as ALT when c is hidden!

■ Definitions of P not satisfying this specification include: $P = STOP$, $P = a \rightarrow b \rightarrow (P \,\Box\, a \rightarrow c \rightarrow STOP)$

Safety Spec vs. Liveness Spec

Saying that $tr \in traces(P)$ is a *positive* statement.

- Describes something that P can do!

$SPEC \sqsubseteq_T P$ puts limit on traces that P can do; restricts behaviour.

- P may fail a *safety* (trace) specification by doing too much.

Saying that $(tr, X) \in failures(P)$ is a *negative* statement.

- Describes something that P cannot do!

$SPEC \sqsubseteq_F P$ puts limit on what P can fail to do.

$\Rightarrow$ Requires P to accept at least a certain range of behaviours.

- P may fail a *liveness* (failure) specification by refusing too much, i.e. by not doing enough.

Example: Moving Furniture

- Two furniture movers need to move a table and a piano. Each requires two people to lift it.

$$
\begin{aligned}
PETE &= lift.piano \rightarrow PETE \\
&\sqcap\ lift.table \rightarrow PETE
\end{aligned}
$$

$$
\begin{aligned}
DAVE &= lift.piano \rightarrow DAVE \\
&\sqcap\ lift.table \rightarrow DAVE
\end{aligned}
$$

$$
TEAM = PETE \parallel DAVE
$$

Both Pete and Dave make their decisions independently! ($\sqcap$)

- If both make same choice, they can cooperate in moving an object.

Deadlock Example: Moving Furniture

- If their choices are different, ...

$$PETE \xrightarrow{\tau} lift.piano \rightarrow PETE$$

$$DAVE \xrightarrow{\tau} lift.table \rightarrow DAVE$$

 $lift.piano \rightarrow PETE \parallel lift.table \rightarrow DAVE$ cannot do anything.

(It is equivalent to the process $STOP$!)

A state of a process is deadlocked if it can **refuse** to do every event.
$STOP$ is the simplest deadlocked process.

Deadlock Example: Children Painting

- Ella and Kate share a paint box and an easel.

$$ELLA = ella.get.box \rightarrow ella.get.easel \rightarrow ella.paint \rightarrow ella.put.box \rightarrow ella.put.easel \rightarrow ELLA$$

$$KATE = kate.get.easel \rightarrow kate.get.box \rightarrow kate.paint \rightarrow kate.put.easel \rightarrow kate.put.box \rightarrow KATE$$

$$EASEL = ella.get.easel \rightarrow ella.put.easel \rightarrow EASEL \;\square\; kate.get.easel \rightarrow kate.put.easel \rightarrow EASEL$$

$$BOX = ella.get.box \rightarrow ella.put.box \rightarrow BOX \;\square\; kate.get.box \rightarrow kate.put.box \rightarrow BOX$$

Combination of two children, box and easel:

$$PAINTING = ELLA||KATE||EASEL||BOX$$

(Assume synchronisation on (intersection of) individual alphabets.)

Conditions for Deadlock

Coffman, Elphick and Shoshani identified 4 *necessary and sufficient* conditions for deadlock [*System Deadlocks*. ACM Computing Surveys 3, 2 (June), p. 67-78, 1971.]

1. Agents claim exclusive control of the resources they require.

⇒ **"Mutual exclusion"** condition

2. Agents hold resources already allocated to them while waiting for additional resources.

⇒ **"Wait for"** condition

3. Resources cannot be forcibly removed from the agent holding them until the resources are used to completion.

⇒ **"No preemption"** condition

4. A circular chain of agents exists, s.t. each agent holds one or more resources that are being requested by the next task in the chain.

⇒ **"Circular wait"** condition

Breaking Deadlock

Aim: System in which possibility of deadlock is excluded a priori.

 Ensure that at least one of the conditions is not satisfied!

$\Rightarrow$ Constrain the way in which requests for resources are made.

- Usually **"Mutual exclusion"** condition **cannot be denied**.
- Each agent must request all its required resources at once and cannot proceed until all have been granted.  Make it **atomic**!
  $\Rightarrow$ "Wait for" condition denied.
- If an agent holding certain resources is denied a further request, that agent must release its original resources and, if necessary, request them again together with the original resources.
  $\Rightarrow$ **"No preemption"** condition **denied**.
- Imposition of a *linear* ordering of resource types.
  $\Rightarrow$ "Circular wait" condition denied.

Example: Breaking Deadlock

- If an agent holding certain resources is denied a further request, that agent must release its original resources and, if necessary, request them again together with the original resources.

$\Rightarrow$ **"No preemption"** condition **denied**.

- Let Ella return tools before they have been used, rather than wait indefinitely for them to be available.

$$
\begin{aligned}
ELLA' \;=\; & ella.get.box \rightarrow (ella.put.box \rightarrow ELLA' \\
& \;\Box\; ella.get.easel \rightarrow ella.paint \rightarrow \\
& \quad ella.put.box \rightarrow ella.put.easel \rightarrow ELLA') \\
& \Box\; ella.get.easel \rightarrow (ella.put.easel \rightarrow ELLA' \\
& \;\Box\; ella.get.box \rightarrow ella.paint \rightarrow \\
& \quad ella.put.easel \rightarrow ella.put.box \rightarrow ELLA')
\end{aligned}
$$

Will this get rid of the deadlock? Yes, but it introduces a "livelock"!

Summary

- **Deadlock**
  …failure model using trace-refusal pairs
  …conditions for deadlock
  …breaking deadlock

- **Trace Refinement:** safety specification – (do nothing wrong)
- **Failure Refinement:** liveness specification – (do something right)

Next Time: Livelock Freedom – (actually progress a task)