Department of Computer Science
University of Bristol

COMS20001 - Concurrent Computing

www.cs.bris.ac.uk/Teaching/Resources/COMS20001

Lecture 14

CSP: Liveness, Deadlock, Livelock

Sion Hannuna | hannuna@cs.bris.ac.uk
Tilo Burghardt | tilo@cs.bris.ac.uk
Dan Page | daniel.page@bristol.ac.uk

18 Slides

Recap: Refusals and Failures

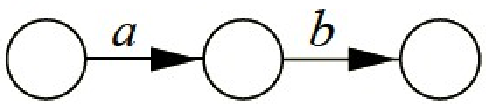
We write P/tr for the process whose behaviour is whatever P could do after the trace tr has been observed.

Failures of a process:

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

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

Transition Diagram of P:



$$
\begin{aligned}
traces(P) &= \{\langle\rangle, \langle a\rangle, \langle a, b\rangle\} \\[1em]
refusals(P/\langle\rangle) &= \{\{\}, \{b\}\} \\
refusals(P/\langle a\rangle) &= \{\{\}, \{a\}\} \\
refusals(P/\langle a, b\rangle) &= \{\{\}, \{a\}, \{b\}, \{a, b\}\} \\[1em]
failures(P) &= \{(\langle\rangle, \{\}), (\langle\rangle, \{b\}), \\
&\quad (\langle a\rangle, \{\}), (\langle a\rangle, \{a\}), \\
&\quad (\langle a, b\rangle, \{\}), (\langle a, b\rangle, \{a\}), (\langle a, b\rangle, \{b\}), (\langle a, b\rangle, \{a, b\})\}
\end{aligned}
$$

Failure Refinement I

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

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

(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 $SPEC \sqsubseteq_T P$.

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

- 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 $ALT = a \to b \to ALT$ as before.

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

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

All are the same as ALT when c is hidden!

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

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.

$$
\begin{aligned}
ELLA \;=\;& ella.get.box \to ella.get.easel \to ella.paint \to \\
& \quad ella.put.box \to ella.put.easel \to ELLA
\end{aligned}
$$

$$
\begin{aligned}
KATE \;=\;& kate.get.easel \to kate.get.box \to kate.paint \to \\
& \quad kate.put.easel \to kate.put.box \to KATE
\end{aligned}
$$

$$
\begin{aligned}
EASEL \;=\;& ella.get.easel \to ella.put.easel \to EASEL \\
\square\;& kate.get.easel \to kate.put.easel \to EASEL
\end{aligned}
$$

$$
\begin{aligned}
BOX \;=\;& ella.get.box \to ella.put.box \to BOX \\
\square\;& kate.get.box \to kate.put.box \to BOX
\end{aligned}
$$

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!

⇒ 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**!
  - ⇒ "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.
  - ⇒ **"No preemption"** condition **denied**.
- Imposition of a *linear* ordering of resource types.
  - ⇒ "Circular wait" condition denied.

Example: Breaking Deadlock with Semaphore

- Introduce a semaphore to break the deadlock!
- Ella and Kate share a paint box and an easel. (SIMPLIFIED)

$$
\begin{aligned}
ELLA &= ella.getsem \rightarrow ella.get.box \rightarrow ella.get.easel \rightarrow ella.paint \rightarrow \\
&\quad ella.put.box \rightarrow ella.put.easel \rightarrow ella.putsem \rightarrow ELLA \\
KATE &= kate.getsem \rightarrow kate.get.easel \rightarrow kate.get.box \rightarrow kate.paint \rightarrow \\
&\quad kate.put.easel \rightarrow kate.put.box \rightarrow kate.putsem \rightarrow KATE \\
BSEM' &= ella.getsem \rightarrow ella.putsem \rightarrow BSEM' \\
&\ \Box\ kate.getsem \rightarrow kate.putsem \rightarrow BSEM'
\end{aligned}
$$

- To achieve desired synchronisation with semaphore (in CSP), we need one channel for each process that uses the semaphore.

Now deadlock-free: $PAINTING = ELLA \| KATE \| BSEM \| EASEL \| BOX$

(Assume $\|$ abbreviates alphabetised parallel here, so synchronisation is on intersection of individual alphabets.)

Example: Breaking Deadlock via Preemption

- 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.

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

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

Livelock … liveness without progress

Difficult to distinguish *wanted* and *unwanted* repeated events!

- Hide "auxiliary" events, just focus on important events!
- In Ella/Kate example we are only interested in $paint$ events:

  $SYSTEM = PAINTING \backslash \{ella, kate\}.\{put, get\}.\{easel, box\}$

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

In CSP, possibility of an infinite sequence of τ events is called **livelock** or **divergence**.

- A divergent process cannot be guaranteed to make any progress.

- Ella can loop forever without achieving any painting!
- Busy waiting (e.g. in mutual exclusion algorithms) counts as divergence!

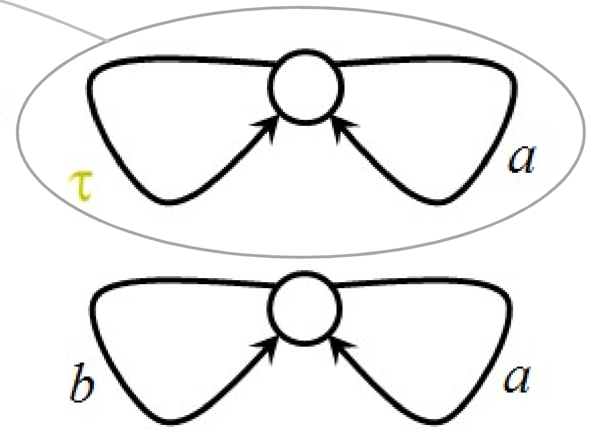
Failure Refinement **cannot** detect Livelock!

Trace model: $Spec \sqsubseteq_T Imp \text{ iff } traces(Spec) \supseteq traces(Imp)$
Failure model: $Spec \sqsubseteq_F Imp \text{ iff } failures(Spec) \supseteq failures(Imp)$

- Failures model does not allow to detect whether a process might livelock!
- $SPEC \sqsubseteq_T IMP \backslash \{b\}$ and $SPEC \sqsubseteq_F IMP \backslash \{b\}$ where

$$SPEC = a \to SPEC$$
$$IMP = a \to IMP \,\square\, b \to IMP$$

Both refinements hold. BUT IMP livelocks!



Remember:

- The process $P \backslash A$ undergoes same executions as P, but events from A occur as *internal events* τ, which are *not visible*.

Failure-Divergence Refinement

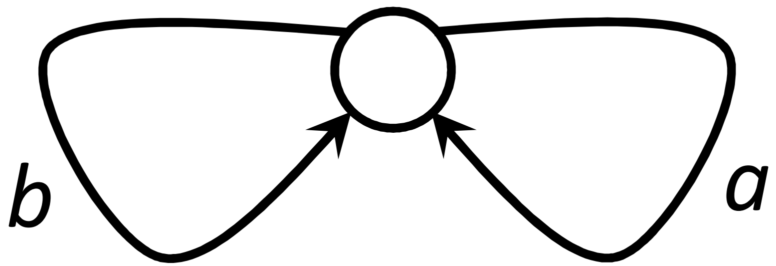
We need a model that is more thorough than stable failures model!

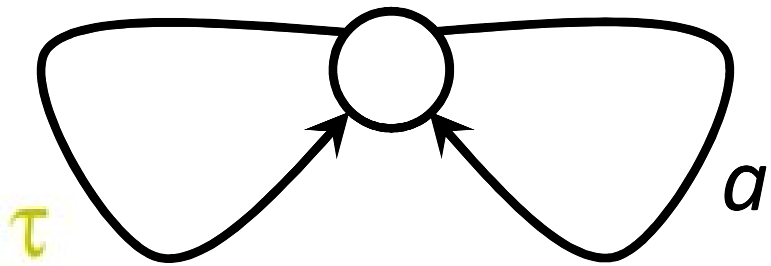
- **Failures Divergences model** represents process by its stable failures and its divergences.

A *divergence* is a finite trace during or after which the process can perform an infinite sequence of consecutive internal events.

$Spec \sqsubseteq_{FD} Imp$ *iff*

$\quad failures(Spec) \supseteq failures(Imp)$ *and* $divergences(Spec) \supseteq divergences(Imp)$





Summary of CSP…

Traces model:

- Safety properties (do no wrong).
- The sequences of traces that a process can perform.

Failures model:

- Liveness properties (do something right).
- Deadlock freedom.
- Pairs of traces and the refusals that may occur after them.

Failures-Divergence model:

- Livelock freedom.
- Failures plus the traces that lead to divergence.