Extending CSP with tests for availability

Gavin Lowe

Testing for availability

Many languages for message-passing concurrency allow programs to test whether a channel is ready for communication, without actually performing that communication.

How can we extend CSP to model such a construct, and so analyse systems that use it?

Overview

- New syntax;
- Examples;
- Operational semantics;
- Congruent denotational semantics: traces and failures;
- Simulating such availability tests using standard CSP; model checking.

New syntax

We add a single new construct to the language: the process

$\mathsf{if\ ready}\ a\ \mathsf{then}\ P\ \mathsf{else}\ Q$

tests whether the event a is ready for communication, and then acts like either P or Q, appropriately. More precisely, the process tests whether all other processes that have a in their alphabet (and so who must synchronise on a) are ready to perform a.

The test for the readiness of a is carried out only once: if the event becomes available or unavailable after the test is performed, that does not affect the branch that is selected.

We also define:

$$\begin{array}{rll} \mathsf{ready}\ a \mathbin{\&} P & \text{as shorthand for} & \mathsf{if\ ready}\ a\ \mathsf{then}\ P\ \mathsf{else}\ STOP, \\ \mathsf{notReady}\ a \mathbin{\&} P & \text{as shorthand for} & \mathsf{if\ ready}\ a\ \mathsf{then}\ STOP\ \mathsf{else}\ P. \end{array}$$

Capturing priority

The following construct

$$\begin{array}{l} a \rightarrow P \\ \Box\ \mathsf{notReady}\ a \ \&\ b \rightarrow Q \end{array}$$

gives priority to a over b: the event b can be performed only if the environment is not willing to perform a (at the point at which the test is made).

The readers and writers problem

A collection of readers and writers share a database. In order to maintain consistency, readers may not use the database at the same time as writers, and at most one writer may use the database at a time.

The following guard process supports this, maintaining the invariant $w \leq 1 \wedge (r > 0 \Rightarrow w = 0)$.

$$\begin{aligned}
&\mathit{Guard}(r, w) = \\
&\qquad w = 0 \mathbin{\&} \mathit{startRead} \rightarrow \mathit{Guard}(r + 1, w) \\
&\qquad \Box\ \mathit{endRead} \rightarrow \mathit{Guard}(r - 1, w) \\
&\qquad \Box\ r = 0 \wedge w = 0 \mathbin{\&} \mathit{startWrite} \rightarrow \mathit{Guard}(r, w + 1) \\
&\qquad \Box\ \mathit{endWrite} \rightarrow \mathit{Guard}(r, w - 1).
\end{aligned}$$

However, with this design writers may be permanently locked out of the database if there is always at least one reader using the database (even if no individual reader uses the database indefinitely).

The readers and writers problem

The following version gives priority to writers, by not allowing a new reader to start using the database if there is a writer waiting:

$$
\begin{array}{l}
Guard(r, w) = \\
\quad w = 0 \;\&\; \mathsf{notReady}\; startWrite \;\&\; startRead \rightarrow Guard(r+1, w) \\
\quad \Box\; endRead \rightarrow Guard(r-1, w) \\
\quad \Box\; r = 0 \wedge w = 0 \;\&\; startWrite \rightarrow Guard(r, w+1) \\
\quad \Box\; endWrite \rightarrow Guard(r, w-1).
\end{array}
$$

This idea can be extended, to make the system fair to both sides: see paper.

Urgency

Consider $P \parallel Q$ where

$$P = a \rightarrow STOP$$
$$Q = \textsf{if ready } a \textsf{ then } b \rightarrow STOP \textsf{ else } error \rightarrow STOP.$$

Clearly, it is possible for Q to detect that a is ready and so perform b.

Could Q detect that a is not ready, and so perform $error$?

- If P makes a available immediately then clearly the answer is no.
- If it takes P some time to make a available, then Q could test for the availability of a before P has made it available.

We believe that any implementation of prefixing will take some time to make a available. This is the intuition we follow.

Therefore the second of the above cases applies.

Operational semantics: standard events

As normal, we write $P \xrightarrow{a} P'$, for $a \in \Sigma \cup \{\tau\}$ to indicate that P performs the event a to become P'.

This is defined in the normal way, except we need to capture our intuition about the unurgency of prefixing: that $a \to P$ may not make the a available immediately.

We model this by a τ transition to a state where the a is indeed available. Within the operational semantic definitions, we will write this state as $\check{a} \to P$.

$$a \to P \xrightarrow{\tau} \check{a} \to P,$$
$$\check{a} \to P \xrightarrow{a} P.$$

(Note that $\check{a} \to P$ is not part of the syntax of the language.)

$$\begin{tikzcd}
a \to P \arrow[d, "\tau"] \\
\check{a} \to P \arrow[d, "a"] \\
P
\end{tikzcd}$$

Operational semantics: readiness and non-readiness

In addition, we include transitions to indicate successful readiness or non-readiness tests:

- We write $P \xrightarrow{ready\ a} P'$ to indicate that P detects that the event a is ready, and evolves into P';
- We write $P \xrightarrow{notReady\ a} P'$ to indicate that P detects that the event a is not ready, and evolves into P'.

The following rules show how the tests for readiness operate.

$$\textsf{if ready } a \textsf{ then } P \textsf{ else } Q \xrightarrow{ready\ a} P,$$
$$\textsf{if ready } a \textsf{ then } P \textsf{ else } Q \xrightarrow{notReady\ a} Q.$$

$$\begin{tikzcd}
 & \textsf{if ready } a \textsf{ then } P \textsf{ else } Q \arrow[dl, "ready\ a"'] \arrow[dr, "notReady\ a"] & \\
P & & Q
\end{tikzcd}$$

Operational semantics: rules for parallel composition

$$\frac{\begin{array}{l} P \xrightarrow{ready\ b} P' \\ Q \xrightarrow{b} \end{array}}{P \ {}_A\|_B\ Q \xrightarrow{ready\ b} P' \ {}_A\|_B\ Q}\ b \in B$$

$$\frac{\begin{array}{l} P \xrightarrow{notReady\ b} P' \\ Q \not\xrightarrow{b} \end{array}}{P \ {}_A\|_B\ Q \xrightarrow{\tau} P' \ {}_A\|_B\ Q}\ b \in B$$

$$\frac{\begin{array}{l} P \xrightarrow{notReady\ b} P' \\ Q \xrightarrow{b} \end{array}}{P \ {}_A\|_B\ Q \xrightarrow{notReady\ b} P' \ {}_A\|_B\ Q}\ b \in B$$

Denotational semantics

We want to build a denotational semantics that at least records the traces of visible events performed by processes. What else does the denotational model need to include?

It is useful to consider (informally) a form of testing: we will say that test T distinguishes processes P and Q if $P \parallel T$ and $Q \parallel T$ have different traces of visible events. In this case, the denotational model should also distinguish P and Q.

We want to record within traces the *ready* and *notReady* actions that are performed. For example, the processes $P = b \rightarrow STOP$ and $Q = \mathsf{ready}\ a \mathbin{\&} b \rightarrow STOP$ are distinguished by the test $T = b \rightarrow STOP$ (with alphabet $\{a, b\}$). We will distinguish them denotationally by including the *ready a* action in the latter's trace.

Denotational semantics

Further, we want to record the events that were available as alternatives to those events that were actually performed. For example, the processes $P = a \to STOP \,\Box\, b \to STOP$ and $Q = a \to STOP \sqcap b \to STOP$ can be distinguished by the test $T = \mathsf{ready}\ a \mathbin{\&} b \to STOP$. We will distinguish them denotationally by recording that the former offers a as an alternative to b.

We therefore add actions $\mathit{offer}\ a$ and $\mathit{notOffer}\ a$ to represent that a process is offering or not offering a, respectively. These actions will synchronise with $\mathit{ready}\ a$ and $\mathit{notReady}\ a$ actions.

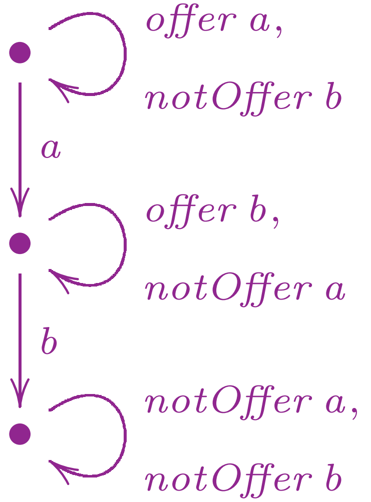
A trace of a process will, then, be a sequence of standard events and ready, $\mathit{notReady}$, offer and $\mathit{notOffer}$ actions.

Extracting traces from the operational semantics

We can formally define the denotational semantics by extracting the traces from the operational semantics.

It is convenient to define a derived operational semantics, where we augment the semantics with extra transitions as follows:

- We add $\mathit{offer}\ a$ loops on every state P such that $P \xrightarrow{a}$;
- We add $\mathit{notOffer}\ a$ loops on every state P such that $P \overset{a}{\not\longrightarrow}$.



The traces, then, are just the visible actions labelling paths through the derived operational semantics.

Compositional trace semantics

It is also possible to give compositional rules for calculating the traces of a process in terms of the traces of its subcomponents.

For example, the semantic equation for hiding of A captures that $\mathit{notReady}\ A$ and $\mathit{offer}\ A$ actions are blocked, A and $\mathit{ready}\ A$ actions are internalised, and arbitrary $\mathit{notOffer}\ A$ actions can occur.

$$\begin{aligned}&\mathit{traces}_R[\![P \setminus A]\!] = \\ &\quad \{tr \mid \exists\, tr_P \in \mathit{traces}_R[\![P]\!] \bullet tr_P \restriction (\mathit{notReady}\ A \cup \mathit{offer}\ A) = \langle\rangle \wedge \\ &\qquad tr_P \setminus (A \cup \mathit{ready}\ A) = tr \setminus \mathit{notOffer}\ A\}.\end{aligned}$$

See paper for the rest of the rules.

This compositional semantics is congruent to the operational semantics.

Stable failures

It is possible to extend the model to record stable failures. Each stable failure is a pair (tr, X), where

- tr is a trace of standard events and $ready$, $notReady$, $offer$ and $notOffer$ actions;
- X is a set of standard events.

This failure records that the process can perform the trace tr to reach a stable state (where no τ, $ready$ or $notReady$ actions are available), where none of the events from X is available.

Model checking

It is possible to simulate the extended version of CSP using standard CSP, and so use a model checker such as FDR to perform an analysis.

In particular, we simulate the *ready*, *notReady*, *offer* and *notOffer* actions by fresh CSP events on channels ready, notReady, offer and notOffer.

We add an offer.a or notOffer.a loop to each state, depending on whether an a is or is not offered. For example

$$P = a \rightarrow STOP$$

with alphabet $\{a, b\}$ is simulated by

```
PSim    = (notOffer.a → PSim □ notOffer.b → PSim) ▷ PSim'
PSim'   = a → STOPSim □ offer.a → PSim' □ notOffer.b → PSim'
STOPSim = notOffer.a → STOPSim □ notOffer.b → STOPSim
```

This can be done compositionally.

Simulating guards

ready a and notReady a guards can be simulated using ready.a and notReady.a events. For example

if ready a then P else Q

is simulated by

$$\begin{aligned}\mathsf{Sim} &= \mathsf{ready.a} \rightarrow \mathsf{PSim} \;\Box\; \mathsf{notReady.a} \rightarrow \mathsf{QSim} \\ &\quad \Box\; \mathsf{notOffer?x} \rightarrow \mathsf{Sim}\end{aligned}$$

where PSim and QSim simulate P and Q.

The ready and notReady events can be renamed to synchronise with the offer.a or notOffer.a events of the process's environment.

We intend to automate the translation that produces this simulation.

Summary

We have considered an extension of CSP that allows processes to test whether an event is available.

- Operational semantics;
- Denotational semantics;
- Simulation using standard CSP.