Introduction to CSP

CSP is a programming/modeling language based on synchronous message passing. It is intended to model complex reactive systems. Unlike SPL the main observables are communication events. Data is viewed as parameters of processes, and as components of messages.

The most basic process is $\text{Stop}$ which stops and does nothing.

Basic actions (messages) are sequenced by the prefixing operator $\rightarrow$. Thus, the process

$$in \rightarrow out \rightarrow Stop$$

performs the actions $in$ followed by $out$ and then stops.
Recursion

Process names can be introduced and used on the left-hand side as well as on the right-hand side of defining equations. This enables using recursion (in fact iteration) for modeling infinite behaviors. Thus, the following definitions are all equivalent and represent a process that alternately performs the actions to and fro ad infinitum.

\[
\begin{align*}
\text{Alt} & = \text{to} \rightarrow \text{fro} \rightarrow \text{Alt} \\
\text{Dalt} & = \text{to} \rightarrow \text{fro} \rightarrow \text{to} \rightarrow \text{fro} \rightarrow \text{Dalt} \\
\text{Malt1} & = \text{to} \rightarrow \text{Malt2}, \quad \text{Malt2} = \text{fro} \rightarrow \text{Malt1} \\
\text{Nalt} & = \text{to} \rightarrow \text{fro} \rightarrow \text{Dalt} \\
\mu P. \text{to} \rightarrow \text{fro} \rightarrow P
\end{align*}
\]

The operator $\square$ represents a selection between two possible behaviors. For example, let $B$ and $C$ be two sets of actions. The following process produces the action out.0 on each “input” of a $B$-action and produces out.1 on each $C$-action.

\[
P = ?x : B \rightarrow \text{out.0} \rightarrow P \square ?x : C \rightarrow \text{out.1} \rightarrow P
\]

The “wild card” $?x : B$ represents an arbitrary choice of any value out of the action set $B$. Once such an action is selected it is bounded to the variable $x$. For example, the following process performs pairs of consecutive actions selected out of a set $A$.

\[
P = ?x : A \rightarrow x \rightarrow P
\]
Other Types of Selection

Processes can be parameterized. This enables associating variable values with processes. For example, the following process generates a sequence of \textit{up} and \textit{down} actions, such that at every prefix of the sequence, the number of \textit{up}’s is greater than or equal to the number of \textit{down}’s.

\[
\begin{align*}
\text{Count}(0) & = \text{up} \rightarrow \text{Count}(1) \\
\text{Count}(n + 1) & = \begin{cases} 
\text{up} & \rightarrow \text{Count}(n + 2) \\
\square \text{down} & \rightarrow \text{Count}(n)
\end{cases}
\end{align*}
\]

Consider again the selection statement

\[P = \ ?x : B \rightarrow \text{out.}0 \rightarrow P \ \square \ ?x : C \rightarrow \text{out.}1 \rightarrow P\]

For the case that $B \cap C = \emptyset$ the selection is always associated with a visible action. On the other hand, if $B \cap C \neq \emptyset$, then the selection is not immediately visible, because it may be associated with an action $a \in B \cap C$. CSP has an explicit \textit{internal selection} operator, denoted $\sqcap$. 
Conditional Processes

The language allows also conditional selection depending on data values. For example, the following filter process inputs arbitrary integers and outputs only the non-negative among them:

\[ F = \text{in}\? x \rightarrow \text{if } x \geq 0 \text{ then } (\text{out}\! x \rightarrow F) \text{ else } F \]

The action \text{out}\! x is equivalent to \text{out}.x. For a boolean expression \( b \), the notation \( b \& P \) is an abbreviation for the process \text{if } b \text{ then } P \text{ else } \text{Stop}.

This combination is useful in order to guard communication actions by boolean guards. For example, the following process described a token moving around an \( N \times M \) board:

\[ \text{Counter}(i, j) = \begin{cases} (i > 0) \& \text{left} & \rightarrow & \text{Counter}(i-1, j) \\ (i < N-1) \& \text{right} & \rightarrow & \text{Counter}(i+1, j) \\ (j > 0) \& \text{down} & \rightarrow & \text{Counter}(i, j-1) \\ (j < M-1) \& \text{up} & \rightarrow & \text{Counter}(i, j+1) \end{cases} \]
Parallel Composition Operators

CSP has several parallel composition operators. They differ by the amount of synchronization on actions which each of them require.

On one extreme, we have the operator \( \parallel \) which requires synchronization on all visible actions. The following equivalence captures part of this full synchronization requirement:

\[
(?x : A \rightarrow P(x)) \parallel (?x : B \rightarrow Q(x)) \sim ?x : A \cap B \rightarrow (P(x) \parallel Q(x))
\]

As an additional example, consider the process \( \text{Mult}(N, 0) \) which is always ready to communicate \( a \) and communicate \( b \) only following a number of \( a \)'s which is evenly divisible by \( N \).

\[
\text{Mult}(N, m) = \left( \begin{array}{c}
\square & m = 0 \land b \quad \rightarrow \quad \text{Mult}(N, m + 1 \mod(N)) \\
\square & m = 0 \land a \quad \rightarrow \quad \text{Mult}(N, m)
\end{array} \right)
\]

The following equivalence shows the effect of forming the parallel composition of \( \text{Mult}(N, 0) \) and \( \text{Mult}(M, 0) \):

\[
\text{Mult}(N, 0) \parallel \text{Mult}(M, 0) \sim \text{Mult}(\text{lcm}(N, M), 0)
\]

where \( \text{lcm}(N, M) \) is the least common multiple of \( N \) and \( M \).

The most prevalent use of this parallel operator is for modeling hand shaking (synchronous) communication between two processes, as follows:

\[
(c!x \rightarrow P) \parallel (c?y \rightarrow Q(y)) \sim c!x \rightarrow (P \parallel Q(x))
\]
Parallelism with Selective Synchronization

The interfaced parallel operator $P \parallel_X Q$ forms parallel composition while requiring synchronization on all actions of $X$. Actions not belonging to $X$ are interleaved with the rest of the actions. Let $P = \{x : A \rightarrow P'(x)\}$ and $Q = \{x : B \rightarrow Q'(x)\}$, then

$$P \parallel_X Q \sim \begin{cases} \{x : X \cap A \cap B\} & \rightarrow P'(x) \parallel_X Q'(x) \\ \Box \{x : A - X\} & \rightarrow P'(x) \parallel_X Q \\ \Box \{x : B - X\} & \rightarrow P \parallel_X Q'(x) \end{cases}$$

As another example, consider the parallel composition $L \parallel_{\{mid\}} R$, where

$$L = \text{left?} x \rightarrow \text{mid!} x \rightarrow L \quad \text{and} \quad R = \text{mid?} x \rightarrow \text{right!} x \rightarrow R$$

This parallel composition represents a 1-place buffer which reads a value from $\text{left}$ and eventually outputs it to $\text{right}$. It is almost equivalent to the sequential process

$$B = \text{left?} x \rightarrow \text{right!} x \rightarrow B$$

except, that the parallel composition displays an additional communication event along channel $\text{mid}$.

At the other extreme of synchronization, we have the strict interleaving parallel operator $\parallel\parallel$ which can be defined as

$$P \parallel\parallel Q = P \parallel_Q 0$$

Here we form a parallel composition based on strict interleaving with no synchronization.
Two Representations of a Set

Let $A$ be a non-empty set, and let $i : A$. We present two representations of a class $Set$ which supports the methods $\text{add.}i$, $\text{delete.}i$, and $\text{member.}i$ for each $i \in A$, and global methods $\text{empty}$ and $\text{nonempty}$.

$$Set1(X) = \begin{cases} \text{add.?}i & \rightarrow Set1(X \cup \{i\}) \\ \text{delete.?}i & \rightarrow Set1(X - \{i\}) \\ \text{member.?}i : X & \rightarrow Set1(X) \\ X = \emptyset & \rightarrow Set1(X) \\ X \neq \emptyset & \rightarrow Set1(X) \end{cases}$$

Here answers to queries are expressed as readiness to synchronize, rather than returning a boolean value.

The other representation is given by $Set2 = \bigsqcup_{\text{empty}} \{S(i, 0) \mid i \in A\}$, where

$$S(i, b) = \begin{cases} \text{add.}i & \rightarrow S(i, 1) \\ \text{delete.}i & \rightarrow S(i, 0) \\ b & \text{member.}i & \rightarrow S(i, b) \\ b & \text{nonempty} & \rightarrow S(i, b) \\ \neg b & \text{empty} & \rightarrow S(i, b) \end{cases}$$

Note that all potential member must synchronize in order to respond to the $\text{empty}$ query.
Hiding and Renaming

The operation of hiding makes some of the events generated/received in a process invisible to the external world. For example, with the definitions

\[ L = \text{left} ? x \rightarrow \text{mid} ! x \rightarrow L \]
\[ R = \text{mid} ? x \rightarrow \text{right} ! x \rightarrow R \]
\[ B = \text{left} ? x \rightarrow \text{right} ! x \rightarrow B \]

we have the equivalence

\[ (L \parallel R) \setminus \text{mid} \overset{\sim}{\rightarrow} B \]

The operation of renaming allows us to rename some event/channel names into different names. For example:

\[ B \{\text{right} / \text{mid}\} = L \]
\[ B \{\text{left} / \text{mid}\} = R \]

An alternative way of achieving such renaming is based on an a priori parameterization followed by various instantiations:

\[ B(\text{left}, \text{right}) = \text{left} ? x \rightarrow \text{right} ! x \rightarrow B(\text{left}, \text{right}) \]
\[ L = B(\text{left}, \text{mid}) \]
\[ R = B(\text{mid}, \text{right}) \]
Termination and Sequential Concatenation

The process $\textit{Skip}$ represents \textit{successful termination}. This is interpreted as generating the special action of termination, denoted by $\checkmark$. The process $\textit{Skip}$ is different from process $\textit{Stop}$ which represents unconditional deadlock.

Using process $\textit{Skip}$, we can form general \textit{sequential composition}. The process $R = P; Q$ behaves like process $P$ until $P$ produces the event $\checkmark$, at which point $R$ proceeds to behave like process $Q$.

For example, we can create an unbounded stack via the definition:

\[
\begin{align*}
\text{Empty} &= \quad \text{in}\, ?\, x \rightarrow S(x) \; ; \; \text{Empty} \\
S(x) &= \quad \text{out}\, !\, x \rightarrow \text{Skip} \\
\Box \quad \text{in}\, ?\, y \rightarrow S(y) \; ; \; S(x)
\end{align*}
\]
**The Trace Semantics of Processes**

The simplest semantics of processes assign to each process $P$ a set of traces which are sequences of possible actions which the process can execute. For every process $P$ the set $\text{traces}(P)$ is nonempty and prefix-closed. Following is an inductive definition of $\text{traces}(P)$:

- $\text{traces}(\text{Stop}) = \{\langle \rangle \}$ – The empty trace.

- $\text{traces}(a \rightarrow P) = \{\langle \rangle \} \cup \{\langle a \rangle \circ t \mid t \in \text{traces}(P)\}$ – either the empty trace, or $a$ followed by a trace of $P$.

- $\text{traces}(?x : A \rightarrow P) = \{\langle \rangle \} \cup \{\langle a \rangle \circ t \mid a \in A, t \in \text{traces}(P[a/x])\}$ – similar to the above, except that $a$ is chosen out of the set $A$. $P[a/x]$ means the substitution of the value $a$ for all free occurrences of the identifier $x$.

- $\text{traces}(P \sqcap Q) = \text{traces}(P \cap Q) = \text{traces}(P) \cup \text{traces}(Q)$ – traces of $P$ or traces of $Q$.

- $\text{traces}(P \parallel Q) = \text{traces}(P) \cap \text{traces}(Q)$ – these are the traces shared by both $P$ and $Q$.

- $\text{traces}(P \parallel Q) = \{s \parallel t \mid s \in \text{traces}(P), t \in \text{traces}(Q)\}$ – where $s \parallel t$ is the set of traces obtained by identifying all events in $s$ and $t$ which belong to $X^\vee (= X \cup \{\square\})$, and interleaving all the rest. This set is non-empty only if $s[X^\vee] = t[X^\vee]$, where $s[Y]$ is the restriction of the sequence $s$ to the actions in $Y$ obtained by deleting all non-$Y$ actions from $s$.

- $\text{traces}(P || Q) = \text{traces}(P || Q)$.
Trace Semantics Continued

- \( \text{traces}(P \setminus X) = \{ t \setminus X \mid t \in \text{traces}(P) \} \) – where \( t \setminus X = t \restriction (\Sigma \vee - X) \).

- \( \text{traces}(P \{ R \}) = \{ t \mid \exists s \in \text{traces}(P) : sR^*t \} \) – where \( R \) is a renaming relation and \( R^* \) is the pointwise extension of \( R \cup (\sqrt{\_}, \sqrt{\_}) \).

- \( \text{traces} (\text{Skip}) = \{ \langle \_ \rangle, \langle \sqrt{\_} \rangle \} \) – either the empty trace or proper termination.

- \( \text{traces}(P ; Q) = (\text{traces}(P) \cap \Sigma^*) \cup \{ s \circ t \mid s \circ \sqrt{\_} \in \text{traces}(P), t \in \text{traces}(Q) \} \) – either a trace of \( P \) or a terminating trace of \( P \) followed by a trace of \( Q \).

It only remains to assign a trace semantics to a recursive definition such as \( P = F(P) \). As is often the case, this can be done by iterative unwinding of the recursive definition, starting with the minimal process \( \text{Stop} \). We define

\[
\begin{align*}
P^0 &= \text{Stop} \\
P^{n+1} &= F(P^n)
\end{align*}
\]

We then define

- \( \text{traces}(\mu P : F(P)) = \bigcup_{n \in \mathbb{N}} \text{traces}(P^n) \).
**Example: Computing the Semantics of a Recursive Process**

Consider the process defined by

\[ P = to \rightarrow fro \rightarrow P \]

The successive iterations for this definition yield:

\[
\begin{align*}
P^0 &= Stop \\
P^1 &= to \rightarrow fro \rightarrow Stop \\
P^2 &= to \rightarrow fro \rightarrow to \rightarrow fro \rightarrow Stop \\
\cdots
\end{align*}
\]

Computing the traces of these approximations and then taking their union yields

\[ traces(P) = (to \ fro)^* + (to \ fro)^* to \]
Trace Specifications

Viewing CSP processes as generators of traces, a specification of a process can be given as a predicate over traces. Given such a predicate \( \varphi \), a process \( P \) is said to satisfy the specification \( \varphi \), if all traces generated by \( P \) satisfy \( \varphi \).

Following are some typical properties we may wish to specify for processes:

- The event \( \text{error} \) never happens.
- Each occurrence of event \( \text{commit} \) is preceded by \( \text{start} \) then \( \text{running} \), both of which have occurred since the last \( \text{commit} \).
- The sequence of values appearing on channel \( \text{right} \) is always a prefix of the sequence of values appearing on channel \( \text{left} \), and the process performs no other actions. This is a specification for a buffer process.

There are different formalisms in which we can specify these properties. We will consider each in turn.
Temporal Logic LTL

The above properties can be specified by the following LTL formulas:

- Event \( error \) never happens:

  \[ \Box \neg error \]

- Every \( commit \) must be preceded by a \( start \) which is followed by a \( running \) event:

  \[ commit \Rightarrow \Diamond (((\neg commit) S (running \land \Diamond ((\neg commit) S start)))) \]

- \( B(left, right) \) behaves like a buffer. There is no simple and natural LTL specification for this.
An alternative style of specification uses a first-order theory of traces. In this language, the specification of the three properties would appear as follows:

- Event *error* never happens:

\[
tr \{ \text{error} \} = \langle \rangle
\]

- Every *commit* must be preceded by a *start* which is followed by a *running* event:

\[
tr = t \circ \langle \text{commit} \rangle \quad \rightarrow \quad \exists t_1, t_2 : \begin{cases}
t = t_1 \circ t_2 \\
\langle \text{start running} \rangle \leq t_2 \{ \text{start, running} \} \\
t_2 \{ \text{commit} \} = \langle \rangle
\end{cases}
\]

- \( B(\text{left}, \text{right}) \) behaves like a buffer.

\[
tr = tr \{ \text{left, right} \} \land tr \downarrow \text{left} \leq tr \downarrow \text{right}
\]

where \( tr \downarrow a \) is the sequence of values communicated along channel \( a \) in \( tr \).
Specification by Abstract Processes

The specification method which is most compatible with the CSP spirit, is when the specification is given in terms of (more) abstract processes.

For example,

- Event *error* never happens. This can be specified by \( \text{Run}(\Sigma - \{\text{error}\}) \), where

  \[
  \text{Run}(A) = ?x : A \rightarrow \text{Run}(A)
  \]

- Every *commit* must be preceded by *running*, which must be preceded by *start*. Can be specified by \( P_0 \), where

  \[
  P_0 = ?x : \Sigma - \{\text{start, commit}\} \rightarrow P_0 \quad \square \quad \text{start} \quad \rightarrow \quad P_1 \\
  P_1 = ?x : \Sigma - \{\text{running, commit}\} \rightarrow P_1 \quad \square \quad \text{running} \quad \rightarrow \quad P_2 \\
  P_2 = ?x : \Sigma - \{\text{commit}\} \rightarrow P_2 \quad \square \quad \text{commit} \quad \rightarrow \quad P_0
  \]

- \( B(\text{left, right}) \) can be specified by \( B^\infty(\langle\rangle) \), where

  \[
  B^\infty(\langle\rangle) = left?x \rightarrow B^\infty(\langle x\rangle) \\
  B^\infty(t \circ \langle y\rangle) = left?x \rightarrow B^\infty(\langle x\rangle \circ t \circ \langle y\rangle) \\
  \square \quad \text{right!y} \rightarrow B^\infty(t)
  \]
**Refinement between Processes**

We say that process $P$ is more deterministic than process $Q$, denoted $Q \subseteq P$, if

$$
\text{traces}(P) \subseteq \text{traces}(Q)
$$

This is also described by saying that $P$ refines $Q$. We can use $Q$ as a specification for process $P$. 

Beyond Traces

Consider the following two processes:

\[ P = a \rightarrow P \quad \text{and} \quad Q = (a \rightarrow Q) \Box (a \rightarrow \text{Stop}) \]

Both processes have the same set of traces \( a^* \). Yet, there is a feeling we should distinguish between them, since process \( Q \) can deadlock after any finite trace, while \( P \) can never deadlock.

Once we agree that we should distinguish between a process which can deadlock unconditionally and one that never deadlock, we should also distinguish between processes which deadlock under different conditions.

Consider the following processes:

\[ P = a \rightarrow ((b \rightarrow \text{Skip}) \Box (c \rightarrow \text{Skip})) \]
\[ Q = (a \rightarrow b \rightarrow \text{Skip}) \Box (a \rightarrow c \rightarrow \text{Skip}) \]

Both of them have the traces \( \{\langle\rangle, a, ab, ac, ab\sqrt{,}, ac\sqrt{,}\} \). A significant difference between them is that \( P \) decides between the two branches after performing the action \( a \), while \( Q \) decides before executing \( a \). As a result, we have the following different equivalences:

\[ (P \parallel P) \setminus \{b, c\}_{\{a, b, c\}} \sim a \rightarrow \text{Skip} \]
\[ (Q \parallel Q) \setminus \{b, c\}_{\{a, b, c\}} \sim (a \rightarrow \text{Skip}) \Box (a \rightarrow \text{Stop}) \]

The second part of the equivalence to \( Q \) can be caused by one copy of \( Q \) choosing the \( a \rightarrow b \rightarrow \text{Skip} \) branch while the other copy choosing the \( a \rightarrow c \rightarrow \text{Skip} \), and then they cannot synchronize.
The Broom Semantics

As a first approximation to a semantics that can capture the desired distinctions, we can introduce the broom semantics. This semantics lists for each process, in addition to its possible traces, also pairs of the form \((\sigma, X)\). The intended meaning of \((\sigma, X)\) is that the process can perform the trace \(\sigma\) and get into an internal state such that it can progress only be executing one of the actions in the set \(X\).

Reconsider the previous processes:

\[
\begin{align*}
P &= a \rightarrow ((\langle b \rightarrow \text{Skip} \rangle \ □ \ (c \rightarrow \text{Skip})) \\
Q &= (a \rightarrow b \rightarrow \text{Skip}) \ □ \ (a \rightarrow c \rightarrow \text{Skip})
\end{align*}
\]

Their broom semantics can be given as

\[
\begin{align*}
brooms(P) &= (\langle \rangle, \{a\}), (a, \{b, c\}), (ab, \{\checkmark\}), (ac, \{\checkmark\}) \\
brooms(Q) &= (\langle \rangle, \{a\}), (a, \{b\}), (a, \{c\}), (ab, \{\checkmark\}), (ac, \{\checkmark\})
\end{align*}
\]

Thus, \(brooms(P)\) contains the broom \(a \rightarrow ((b \rightarrow \text{Skip}) \ □ \ (c \rightarrow \text{Skip}))\) while \(brooms(P)\) contains the two separate brooms \(a \rightarrow b\) and \(a \rightarrow c\).
The Failures Semantics

Due to technical reasons, it proved more useful to list failures instead of their complementary ready sets. A failure is a pair \((\sigma, X)\) which expresses the capability of a process to execute the trace \(\sigma\) and get into an internal state at which it can refuse to execute any of the actions in the set \(X\).

Reconsider the processes:

\[
P = a \rightarrow ((b \rightarrow \text{Skip}) \square (c \rightarrow \text{Skip}))
\]
\[
Q = (a \rightarrow b \rightarrow \text{Skip}) \square (a \rightarrow c \rightarrow \text{Skip})
\]

Their failure semantics is given by

\[
\text{failures}(P) = (\emptyset, R_a), (a, R_{b,c}), (ab, R_{\sqrt{}}), (ac, R_{\sqrt{}})
\]
\[
\text{failures}(Q) = (\emptyset, R_a), (a, R_b), (a, R_c), (ab, R_{\sqrt{}}), (ac, R_{\sqrt{}})
\]

where \(R_a, R_b, R_c, R_{b,c}, R_{\sqrt{}}\) are any sets such that

\[
a \not\in R_a, \quad b \not\in R_b, \quad c \not\in R_c, \quad \sqrt{\not\in R_{\sqrt{}}}, \quad R_{b,c} \cap \{b, c\} = \emptyset
\]

The failure semantics also helps us to distinguish between the process

\[
(a \rightarrow \text{Stop}) \square (b \rightarrow \text{Stop})
\]

whose semantics contains the failures \((\emptyset, \{a\})\) and \((\emptyset, \{b\})\) and process

\[
(a \rightarrow \text{Stop}) \square (b \rightarrow \text{Stop})
\]

which cannot refuse to synchronize with either \(a\) or \(b\).
Specifying Progress

With the failure semantics, we can capture various notions of progress.

For example, a process \( P \) is deadlock free if \((\sigma, \Sigma^\prime) \notin \text{failures}(P)\) for all \( \sigma \in \Sigma^* \). That is, there exists no trace \( \sigma \) after which the process cannot perform any other action (including termination).

We can extend the buffer specification to guarantee that the buffer cannot refuse to accept an additional input if the buffer is currently empty, nor can it refuse to output a requested element if the buffer is non-empty. These two properties can be specified as follows:

\[
(s, X) \in \text{failures}(B) \land s \downarrow \text{right} = s \downarrow \text{left} \quad \rightarrow \quad X \cap \|\text{left}\| = \emptyset
\]

\[
(s, X) \in \text{failures}(B) \land s \downarrow \text{right} < s \downarrow \text{left} \quad \rightarrow \quad \|\text{right}\| \not\subseteq X
\]

We can extend the notion of process refinement to take failures into account. We say that a process \( Q \) constitutes a failure refinement of process \( P \), denoted \( P \sqsubseteq^f Q \) if

\[
\text{failures}(Q) \subseteq \text{failures}(P) \quad \text{and} \quad \text{traces}(Q) \subseteq \text{traces}(P)
\]
Divergences

In some cases, a process can engage in an infinite sequence of invisible actions. Such a behavior is called divergence. A third dimension of the full semantics of CSP processes is the identification of the capability of divergence.

For example, consider the following process:

\[
div = (\mu P : a \rightarrow P) \setminus \{a\}
\]

This process has only the empty trace and no failures at all. Consequently, this process forms a failures-refinement of any other process, which is completely unacceptable. Therefore, we add to the semantics of processes the component \(\text{divergences}(P)\). A trace \(\sigma\) belongs to \(\text{divergences}(P)\) if either \(\sigma\) or some prefix of it can lead to an internal state after which the process can engage in an infinite sequence of invisible actions. According to this definition, \(\text{divergences}(\text{div}) = \Sigma^*\).

We say that process \(Q\) is a failure-divergence refinement of process \(P\), denoted \(P \sqsubseteq_{FD} Q\), if the three following inclusions all hold:

\[
\begin{align*}
\text{traces}(Q) &\subseteq \text{traces}(P), \\
\text{failures}(Q) &\subseteq \text{failures}(P), \\
\text{divergences}(Q) &\subseteq \text{divergences}(P)
\end{align*}
\]

Usually, we add to a specification the requirement that \(\text{divergences}(P) = \emptyset\). For example, this would be the requirement for the buffer specification.