◇ Traces ◇

A *trace* of a process is a finite sequence of events, representing the behaviour of the process up to a certain point in time. Traces are written as comma-separated sequences of events, enclosed in angle brackets: for example, $\langle coin, choc, coin\rangle$. This is a trace of the recursive version of VM.

Example: $\langle open, close\rangle$ and $\langle open, close, open\rangle$ are traces of $DOOR$.

$(DOOR = open \to close \to DOOR)$

Example: $\langle staines, pound\rangle$ and $\langle ashford, pound\rangle$ are traces of $TICKET$, and also of $TICKETS$.

We will only consider *finite* traces.

The empty trace, containing no events, is written $\langle\rangle$ and pronounced "empty" or "nil". It is a trace of every process, corresponding to an observation when no event has yet happened.

If a process is defined without recursion, then it has a bound on the length of its traces. For example, if

$$PHONE = ring \to answer \to STOP$$

then the only traces of $PHONE$ are $\langle\rangle$, $\langle ring\rangle$ and $\langle ring, answer\rangle$.

A recursive process, which can keep performing events forever, can have an infinite set of traces. For example, if

$$CLOCK = tick \to CLOCK$$

then the traces of $CLOCK$ are

$$\langle\rangle, \langle tick\rangle, \langle tick, tick\rangle, \langle tick, tick, tick\rangle, \ldots$$

It is important to be clear about the fact that we are interested in potentially *infinite* sets of *finite* traces.

◇ Operations on Traces ◇

We will use various operations on traces, and a number of facts or laws about them. Most of the laws are rather obvious.

◇ Concatenation ◇

The first operation is *concatenation*, also called *catenation*. It joins traces together into longer traces:

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

Example: $\langle coin, choc\rangle^\frown\langle choc\rangle = \langle coin, choc, choc\rangle$. Concatenation is associative, and the empty trace is a unit, i.e.

$$tr_0 {}^\frown (tr_1 {}^\frown tr_2) = (tr_0 {}^\frown tr_1) {}^\frown tr_2$$
$$\langle\rangle {}^\frown tr = tr = tr {}^\frown \langle\rangle$$

The following laws are useful:

$$tr_0 {}^\frown tr_1 = tr_0 {}^\frown tr_2 \text{ if and only if } tr_1 = tr_2$$
$$tr_0 {}^\frown tr_2 = tr_1 {}^\frown tr_2 \text{ if and only if } tr_0 = tr_1$$
$$tr_0 {}^\frown tr_1 = \langle\rangle \text{ if and only if } tr_0 = \langle\rangle \text{ and } tr_1 = \langle\rangle$$

If n is a positive integer, then tr^n is defined to be n copies of the trace tr concatenated together. tr^n can be defined recursively by

$$\begin{aligned} tr^0 &= \langle\rangle \\ tr^{n+1} &= tr {}^\frown tr^n. \end{aligned}$$

◇ Functions on Traces ◇

Suppose f is a function which maps traces to traces. f is said to be *strict* if $f(\langle\rangle) = \langle\rangle$, and *distributive* if $f(tr_0 {}^\frown tr_1) = f(tr_0) {}^\frown f(tr_1)$.

In fact, any distributive function is strict: if f is distributive then

$$\begin{aligned} f(tr) {}^\frown \langle\rangle = f(tr) &= f(tr {}^\frown \langle\rangle) \\ &= f(tr) {}^\frown f(\langle\rangle) \end{aligned}$$

and so $f(\langle\rangle) = \langle\rangle$.

If f is distributive then its action on traces can be put together from its action on single-event traces:

$$\begin{aligned} f(\langle a_1, \ldots, a_n\rangle) &= f(\langle a_1\rangle {}^\frown \ldots {}^\frown \langle a_n\rangle) \\ &= f(\langle a_1\rangle) {}^\frown \ldots {}^\frown f(\langle a_n\rangle). \end{aligned}$$

◇ Restriction ◇

The expression $tr \upharpoonright A$ denotes the trace tr when *restricted* to events in the set A. $tr \upharpoonright A$ consists of tr with all events outside A omitted.

Example:

$\langle start, exercise, exercise, end\rangle \upharpoonright \{start, end\}$
$= \langle start, end\rangle$.

$\langle start, exercise, exercise, end\rangle \upharpoonright \{start, exercise\}$
$= \langle start, exercise, exercise\rangle$.

Restriction is distributive and therefore strict:

$$\begin{aligned} \langle\rangle \upharpoonright A &= \langle\rangle \\ (tr_0 {}^\frown tr_1) \upharpoonright A &= (tr_0 \upharpoonright A) {}^\frown (tr_1 \upharpoonright A). \end{aligned}$$

The effect of restriction on single-event traces is clear:

$$\begin{aligned} \langle x\rangle \upharpoonright A &= \langle x\rangle \text{ if } x \in A \\ \langle x\rangle \upharpoonright A &= \langle\rangle \text{ if } x \notin A \end{aligned}$$

Two other facts:

$$\begin{aligned} tr \upharpoonright \{\} &= \langle\rangle \\ (tr \upharpoonright A) \upharpoonright B &= tr \upharpoonright (A \cap B) \end{aligned}$$

◇ Head and Tail ◇

If tr is a non-empty trace, its first event is denoted tr_0 and the trace consisting of all events after the first is denoted tr'.

Neither $\langle\rangle_0$ nor $\langle\rangle'$ is defined.

Example: $\langle coin, choc\rangle_0 = coin$.

$\langle coin, choc\rangle' = \langle choc\rangle$.

A few facts:

$$\begin{aligned}(\langle x\rangle \frown tr)_0 &= x\\ (\langle x\rangle \frown tr)' &= tr\\ tr &= \langle tr_0\rangle \frown tr'\end{aligned}$$

◇ Star ◇

If A is a set of events, the set A^* is the set of all finite traces, including $\langle\rangle$, containing events from A.

Example:

$\{a,b\}^* = \{\langle\rangle, \langle a\rangle, \langle b\rangle, \langle a,a\rangle, \langle a,b\rangle, \langle b,a\rangle, \langle b,b\rangle, \ldots\}$

◇ Ordering ◇

A trace tr_0 is a *prefix* of a trace tr_1 if there is some extension tr_2 of tr_0 such that $tr_0 \frown tr_2 = tr_1$. We then write $tr_0 \leqslant tr_1$.

Example:

$$\begin{aligned}\langle a,b,c\rangle &\leqslant \langle a,b,c,d\rangle\\ \langle\rangle &\leqslant \langle a,b\rangle\end{aligned}$$

◇ Length ◇

The length of the trace tr is denoted $\#tr$.

Example: $\#\langle a,b\rangle = 2$, $\#\langle\rangle = 0$.

◇ Traces of a Process ◇

In general a process has many different possible behaviours, and we do not know in advance which traces will be generated by a particular execution. However, we can determine in advance the set of all possible traces of a process P. This set is written $traces(P)$.

Examples: $traces(STOP) = \{\langle\rangle\}$.

$traces(coin \rightarrow STOP) = \{\langle\rangle, \langle coin\rangle\}$.

$$\begin{aligned}traces(CLOCK) &= \{\langle\rangle, \langle tick\rangle, \langle tick, tick\rangle, \ldots\}\\ &= \{tick\}^*\end{aligned}$$

We can now systematically write down definitions of $traces(P)$ for processes P constructed from the operators we have seen so far. We already know the definition for $STOP$:

$$traces(STOP) = \{\langle\rangle\}.$$

$traces(a \rightarrow P)$ is constructed from $traces(P)$ by the addition of a as an initial event:

$traces(a \rightarrow P) = \{\langle\rangle\} \cup \{\langle a\rangle \frown tr \mid tr \in traces(P)\}$.

Notice the addition of the trace $\langle\rangle$, which must always be a trace of any process.

The definition of $traces(a \rightarrow P \mid b \rightarrow Q)$ is similar, taking account of the two possible first events:

$$\begin{aligned}traces(a \rightarrow P \mid b \rightarrow Q) &= \{\langle\rangle\}\\ &\cup\ \{\langle a\rangle \frown tr \mid tr \in traces(P)\}\\ &\cup\ \{\langle b\rangle \frown tr \mid tr \in traces(Q)\}.\end{aligned}$$

Also similarly, we can give a general definition of $traces(x : A \rightarrow P(x))$.

$$\begin{aligned}&traces(x : A \rightarrow P(x))\\ &\quad = \{\langle\rangle\}\\ &\quad \cup\ \{\langle a\rangle \frown tr \mid a \in A, tr \in traces(P(a))\}.\end{aligned}$$

A few facts about $traces$:

$\langle\rangle \in traces(P)$, for any P.

If $tr_0 \frown tr_1 \in traces(P)$ then $tr_0 \in traces(P)$.

$traces(P) \subseteq (\alpha(P))^*$.

Describing the set of traces of a recursive process is more complicated. Suppose we have the definition

$$X = F(X)$$

where $F(X)$ is a guarded expression. Guardedness means that we know at least the possible first events of $F(X)$. In fact, they are the same as the possible first events of $F(STOP)$, whatever X is.

Example: If $X = a \rightarrow X$ then we know that X can do a first, and this is the same first event as in $a \rightarrow STOP$.

Depending on the form of $F(X)$, we may know more than just the first event.

Example: If $X = a \rightarrow b \rightarrow X \mid c \rightarrow X$ we know that X can either do a then b, or c, so we know that $\langle a,b\rangle$ and $\langle c\rangle$ are traces of X. They are also traces of $a \rightarrow b \rightarrow STOP \mid c \rightarrow STOP$.

We can discover some traces of X by looking at $F(STOP)$. For the traces corresponding to running through F twice, we need to look at $F(F(STOP))$.

Example: If $X = a \to X$ we also have

$$X = a \to a \to X$$

so $\langle a, a\rangle$ is a trace of X.

If $X = a \to b \to X \mid c \to X$ we also have

$$\begin{aligned} X &= a \to b \to (a \to b \to X \mid c \to X) \\ &\mid c \to (a \to b \to X \mid c \to X) \end{aligned}$$

So $\langle a, b, a\rangle$, $\langle a, b, c\rangle$, $\langle c, a, b\rangle$ etc. are traces of X.

In general we can define iteration of F:

$$\begin{aligned} F^0(X) &= X \\ F^{n+1}(X) &= F(F^n(X)) \end{aligned}$$

and then, for $X = F(X)$, we have

$$\begin{aligned} \mathit{traces}(X) &= \bigcup_{n \geqslant 0} \mathit{traces}(F^n(\mathit{STOP})) \\ &= \mathit{traces}(\mathit{STOP}) \cup \mathit{traces}(F(\mathit{STOP})) \\ &\cup\ \mathit{traces}(F(F(\mathit{STOP}))) \cup \ldots \end{aligned}$$

Writing down the set of traces of a recursive process in a compact form is a little challenging. For example, if $X = a \to b \to X$, then $\mathit{traces}(X)$ contains not only $\langle a, b\rangle$, $\langle a, b, a, b\rangle$, $\langle a, b\rangle^3$ and so on, but also the intermediate traces ending in a. One way to describe $\mathit{traces}(X)$ is:

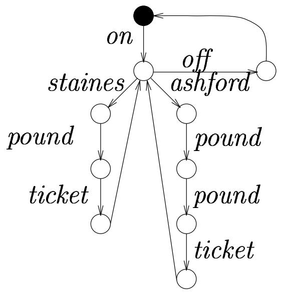
$$\mathit{traces}(X) = \{\mathit{tr} \mid \text{for some } n,\ \mathit{tr} \leqslant \langle a, b\rangle^n\}$$

◇ Traces and Diagrams ◇

There is a connection between the transition diagram of a process, and its traces. For example, recall the process $\mathit{TICKETS}$ defined by

$$\begin{aligned} \mathit{MACHINE} &= \mathit{on} \to \mathit{TICKETS} \\ \mathit{TICKETS} &= \mathit{staines} \to \mathit{pound} \to \mathit{ticket} \\ &\qquad \to \mathit{TICKETS} \\ &\mid \mathit{ashford} \to \mathit{pound} \to \mathit{pound} \to \mathit{ticket} \\ &\qquad \to \mathit{TICKETS} \\ &\mid \mathit{off} \to \mathit{MACHINE} \end{aligned}$$

and its transition diagram:



For any path through the diagram, starting from the black state, there is a trace consisting of the sequence of labels on the path. $\mathit{traces}(\mathit{TICKETS})$ is the set of traces corresponding to all these paths.

◇ Traces and Transitions ◇

The operational semantics of CSP allows us to unwind the behaviour of a process, one event at a time. Looking at the traces of a process gives us an overall view. Since the traces can be extracted from a transition diagram, and labelled transitions are supposed to capture the same information as the diagrams, we should also be able to write down a relationship between a process' traces and its labelled transitions. Here it is:

$$\begin{aligned} &\mathit{traces}(P) = \{\langle\rangle\} \\ &\qquad \cup\ \{\langle a\rangle \frown \mathit{tr} \mid P \xrightarrow{a} Q, \mathit{tr} \in \mathit{traces}(Q)\}. \end{aligned}$$

Later we will be defining new CSP operators, by means of labelled transition rules. We will use this relationship between transitions and traces to calculate the traces of processes defined in terms of the new operators.

◇ Exercises ◇

△ Write down $\mathit{traces}(\mathit{TICKET})$, where TICKET is defined as before by

$$\begin{aligned} \mathit{TICKET} &= \mathit{staines} \to \mathit{pound} \to \mathit{ticket} \to \mathit{STOP} \\ &\mid \mathit{ashford} \to \mathit{pound} \to \mathit{pound} \to \mathit{ticket} \to \mathit{STOP} \end{aligned}$$

◇ Exercises ◇

△ Define a process P such that

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

△ Define a process P such that $\langle a, b, c\rangle$ and $\langle a, b, a\rangle$ are both traces of P.

△ Is there a process P such that

$$\mathit{traces}(P) = \{\langle\rangle, \langle a\rangle, \langle a, b\rangle, \langle c, d\rangle\}?$$

◇ Traces for Concurrency ◇

$$\begin{aligned} traces(P\ {}_A\|_B\ Q) \ =\ \{tr \mid\ & tr \in (A \cup B)^* \\ & \text{and } tr\lceil A \in traces(P) \\ & \text{and } tr\lceil B \in traces(Q)\} \end{aligned}$$

If $A = B$, this definition reduces to

$$\begin{aligned} traces(P\ {}_A\|_A\ Q) \ =\ \{tr \mid\ & tr \in A^* \\ & \text{and } tr\lceil A \in traces(P) \\ & \text{and } tr\lceil A \in traces(Q)\} \end{aligned}$$

i.e. $traces(P\ {}_A\|_A\ Q) = traces(P) \cap traces(Q)$, because if $tr \in A^*$ then $tr\lceil A = tr$. This fits in with the earlier discussion of concurrency with the same alphabet.

If $A \cap B = \{\}$ then every event in a possible trace of $P\ {}_A\|_B\ Q$ is either an event from A or an event from B. In a trace tr of $P\ {}_A\|_B\ Q$, the events from A (i.e. $tr\lceil A$) must form a trace of P, and similarly the events from B must form a trace of Q. Any pair of traces, one from P and one from Q, can be *interleaved* to form a trace of $P\ {}_A\|_B\ Q$.

Example: $\langle left, right, right\rangle$ is a trace of LR and $\langle up, down\rangle$ is a trace of UD. So

$$\langle left, up, down, right, right\rangle$$

is a trace of $LR \parallel UD$.

In general, a trace of P and a trace of Q can be used to form a trace of $P\ {}_A\|_B\ Q$ as long as the events in $A \cap B$ appear in the same order in both traces.

Example: $\langle coin, beep, choc\rangle$ is a trace of VM and $\langle coin, shout, choc\rangle$ is a trace of $CUST$. The events common to both alphabets (i.e. $coin$ and $choc$) appear in the same order in both traces.

$\langle coin, beep, shout, choc\rangle$ and $\langle coin, shout, beep, choc\rangle$ are both traces of $VM \parallel CUST$.

◇ Trace Equivalence ◇

We have spoken vaguely of processes being equivalent to each other — for example, a process which can do no events is equivalent to $STOP$. In CSP there are in fact several notions of process equivalence, each of which is useful in different situations. The first is *trace equivalence*, denoted by $=_T$, and defined by

$$P =_T Q$$
if and only if
$$traces(P) = traces(Q)$$

Two processes are trace equivalent if they have the same observable behaviour, as measured by $traces$.

Example: Consider the process

$$a \to STOP\ {}_{\{a,b\}}\|_{\{a,b\}}\ b \to STOP.$$

The definition of $traces$ for a parallel combination of processes gives
$traces(a \to STOP\ {}_{\{a,b\}}\|_{\{a,b\}}\ b \to STOP)$
$= \{tr \in \{a,b\}^* \mid tr\lceil\{a,b\} \in traces(a \to STOP)$
and $tr\lceil\{a,b\} \in traces(b \to STOP)\}$.

i.e. $traces(a \to STOP\ {}_{\{a,b\}}\|_{\{a,b\}}\ b \to STOP)$
$= traces(a \to STOP) \cap traces(b \to STOP)$.

Because

$$traces(a \to STOP) = \{\langle\rangle, \langle a\rangle\}$$

and

$$traces(b \to STOP) = \{\langle\rangle, \langle b\rangle\}$$

we get

$traces(a \to STOP\ {}_{\{a,b\}}\|_{\{a,b\}}\ b \to STOP) = \{\langle\rangle\}$.

Therefore

$$a \to STOP\ {}_{\{a,b\}}\|_{\{a,b\}}\ b \to STOP =_T STOP.$$

◇ Refinement and Specification ◇

The *refinement* relation $\sqsubseteq_T$ on processes is defined by

$$P \sqsubseteq_T Q$$
if and only if
$$traces(Q) \subseteq traces(P)$$

$P \sqsubseteq_T Q$ is pronounced "P is refined by Q". The subscript T indicates that we are working with traces — later we will see other forms of refinement.

P is refined by Q if Q exhibits at most the behaviour exhibited by P — possibly less.

Example:

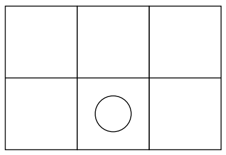
$$a \to b \to STOP \sqsubseteq_T a \to STOP$$

Example: For any process P, $P \sqsubseteq_T STOP$.

The main use of refinement is in specification. If we think of P as defining a range of permissible behaviour, then the statement $P \sqsubseteq_T Q$ can be read as the specification that Q's behaviour must stay within this range.

◇ Example ◇

Recall the example of a counter moving on a board.



$$\begin{aligned} LR &= left \to right \to LR \,\Box\, right \to left \to LR \\ UD &= up \to down \to UD \end{aligned}$$
$$SPEC = LR\ {}_{\{left,right\}}\|_{\{up,down\}}\ UD$$

We can now interpret $SPEC$ as a specification for processes which might describe movements of the counter. Because $SPEC$ describes exactly the behaviours which correspond to staying on the board, the specification

$$SPEC \sqsubseteq_T P$$

specifies that P must describe movement within the board — possibly restricted movement.

For example,

$$SPEC \sqsubseteq_T left \to up \to STOP$$

which we can check by writing down all the traces of the process on the right and showing that they are all traces of $SPEC$.

The specification

$$SPEC \sqsubseteq_T P$$

limits what P can do, but does not require it to do anything. For example,

$$SPEC \sqsubseteq_T STOP.$$

Specifications which simply restrict behaviour without requiring any particular behaviour are known as *safety specifications*. They specify that nothing bad can happen, without specifying that anything good must happen. $STOP$ satisfies any safety specification — doing nothing is always safe.

All specifications which can be expressed using trace refinement are safety specifications.

Specifications which require something positive to happen are called *liveness specifications*. We will see later how they can be expressed in CSP.

Example: If we define P by

$$P = left \to left \to STOP$$

then we do not have $SPEC \sqsubseteq_T P$ because

$$\begin{aligned} \langle left, left\rangle &\in traces(P) \\ \langle left, left\rangle &\notin traces(SPEC). \end{aligned}$$

◇ The Level Crossing ◇

As an example of writing a specification in CSP, we will look at a railway level crossing. One road and one railway line cross each other, and as usual there is a gate which can be lowered to prevent cars crossing the railway. If the gate is raised, then cars can freely cross the track. Trains can cross the road regardless of whether the gate is up or down.

We will consider the obvious safety property for the level crossing, which is:

There should never be a train and a car on the crossing at the same time.

Of course there are many other properties which we might like to specify, for example a liveness property:

Whenever a car approaches the crossing, it should eventually be able to cross.

but for the moment we will stick to safety.

We will use the following events to represent the interesting aspects of the behaviour of the system.

$car.approach$, $car.enter$, $car.leave$, $train.approach$, $train.enter$, $train.leave$, $gate.lower$, $gate.raise$ $crash$

The processes $CARS$ and $TRAINS$ supply streams of cars and trains.

$$\begin{aligned} CARS &= car.approach \rightarrow car.enter \rightarrow \\ &\quad car.leave \rightarrow CARS \\ TRAINS &= train.approach \rightarrow train.enter \rightarrow \\ &\quad train.leave \rightarrow TRAINS \end{aligned}$$

The following processes model the behaviour of the crossing. This is a complete description of all possibilities, including a car and a train simultaneously using the crossing. Later we will add a control process which uses the gate to restrict access by cars.

CR models the crossing with cars and trains. The processes C, T, CT model the crossing when there is a car, train or both present, respectively:

$$\begin{aligned} CR &= car.approach \rightarrow car.enter \rightarrow C \\ &\quad \Box\ train.approach \rightarrow train.enter \rightarrow T \\ C &= car.leave \rightarrow CR \\ &\quad \Box\ train.approach \rightarrow train.enter \rightarrow CT \\ T &= train.leave \rightarrow CR \\ &\quad \Box\ car.approach \rightarrow car.enter \rightarrow CT \\ CT &= crash \rightarrow STOP \end{aligned}$$

Cars can only enter the crossing when the gate is up:

$$\begin{aligned} GATE &= gate.lower \rightarrow gate.raise \rightarrow GATE \\ &\quad \Box\ car.enter \rightarrow GATE \end{aligned}$$

Defining some sets of events:

$$\begin{aligned} E_T &= \{train.approach, train.enter, train.leave\} \\ E_C &= \{car.approach, car.enter, car.leave\} \\ E_{GC} &= \{gate.raise, gate.lower, car.enter\} \\ E_X &= \{crash\} \\ E_S &= E_T \cup E_C \cup E_{GC} \cup E_X \end{aligned}$$

allows us to define the whole system as

$$\begin{aligned} SYSTEM &= ((CR\ {}_{E_T \cup E_C \cup E_X}\|_{E_{GC}}\ GATE) \\ &\qquad {}_{E_S}\|_{E_C}\ CARS)\ {}_{E_S}\|_{E_T}\ TRAINS. \end{aligned}$$

To specify that no crashes occur, we need a process $SPEC$ which can do any event except for $crash$.

$$\begin{aligned} SPEC &= train?x : \{approach, enter, leave\} \rightarrow SPEC \\ &\quad \Box\ car?x : \{approach, enter, leave\} \rightarrow SPEC \\ &\quad \Box\ gate?x : \{raise, lower\} \rightarrow SPEC \end{aligned}$$

In general, RUN_A is the process which can repeatedly do events from the set A:

$$RUN_A = x : A \rightarrow Run_A$$

so $SPEC = RUN_{E_C \cup E_T \cup E_{GC}}$.

The requirement that the crossing satisfies this specification is expressed by

$$SPEC \sqsubseteq_T SYSTEM.$$

It is possible to use the FDR tool to check trace refinement, and this is the easiest way to show that the specification is not satisfied (not surprisingly, as we haven't imposed any restrictions on when the gate can be raised or lowered).

Now we will define a process $CONTROL$ which, when placed in parallel with $SYSTEM$, will constrain the behaviour so that whenever a train approaches the gate must be lowered. This will be achieved by making $CONTROL$ and $SYSTEM$ synchronise on certain events. We hope that the result will be a system which satisfies the safety specification.

$$\begin{aligned} CONTROL &= (train.approach \rightarrow gate.lower \rightarrow \\ &\quad train.enter \rightarrow train.leave \rightarrow \\ &\quad gate.raise \rightarrow CONTROL) \\ &\quad \Box\ (car.approach \rightarrow car.enter \rightarrow \\ &\quad car.leave \rightarrow CONTROL) \end{aligned}$$

$$\begin{aligned} &SAFE\_SYSTEM = \\ &\quad SYSTEM\ {}_{E_S}\|_{E_T \cup E_C \cup E_{GC}}\ CONTROL \end{aligned}$$

Again, FDR can be used to test whether

$$SPEC \sqsubseteq_T SAFE\_SYSTEM$$

and this time we will find that it does.

Here is an alternative way of checking $SYSTEM$. Notice that when a car and a train use the crossing at the same time, the event $crash$ occurs, and the system stops. This is the only point at which we have deliberately introduced $STOP$ into the system, and we hope that there are no other deadlocks.

If we use FDR to check $SYSTEM$ for deadlock-freedom, then every time a deadlock is found we will see a trace leading to $STOP$. If the trace ends in $crash$, then we have identified a violation of safety. If the trace ends with some other event, then there is another deadlock in the system, which presumably represents a mistake in our model.

In general there are many different ways of modelling a system, and many different ways of writing a specification. The challenge is to model the system in such a way that the bad property appears as a kind of behaviour (in this example, occurrence of $crash$) which can be ruled out by a suitable specification.

◇ Another Level Crossing ◇

Here is another way of modelling the level crossing. Remove the *crash* event, and change the definition of CT to

$$\begin{aligned} CT \; &= \; car.leave \rightarrow T \\ &\;\;\;\; \Box \; train.leave \rightarrow C. \end{aligned}$$

Also introduce

$$E_G \; = \; \{gate.raise, gate.lower\}$$

and change the definition of E_S to

$$E_S = E_C \cup E_T \cup E_G.$$

Similarly to before,

$$\begin{aligned} SYSTEM \; &= \; ((CR \; {}_{E_T \cup E_C}\|_{E_{GC}} \; GATE) \\ &\qquad {}_{E_S}\|_{E_C} \; CARS) \; {}_{E_S}\|_{E_T} \; TRAINS. \end{aligned}$$

The specification now consists of two parts.

$$\begin{aligned} SPEC1 \; &= \; RUN_{E_G} \\ SPEC2 \; &= \; train.approach \rightarrow train.enter \rightarrow \\ &\qquad train.leave \rightarrow SPEC2 \\ &\quad \Box \; car.approach \rightarrow car.enter \rightarrow \\ &\qquad car.leave \rightarrow SPEC2 \\ SPEC \; &= \; SPEC1 \; {}_{E_G}\|_{E_T \cup E_C} \; SPEC2 \end{aligned}$$

$SPEC1$ allows the gate to be raised and lowered freely. $SPEC2$ only allows trains and cars to enter the crossing separately.

Again we can check

$$SPEC \sqsubseteq_T SYSTEM$$

which is not true, and define

$$SAFE\_SYSTEM = SYSTEM \; {}_{E_S}\|_{E_S} \; CONTROL$$

and check

$$SPEC \sqsubseteq_T SAFE\_SYSTEM$$

which is true.