Advanced Concurrency Tools

Modelling Discrete Time

Bill Roscoe

Oxford University

13–17 July 1998

Discrete vs Continuous

- The theory of Timed CSP uses a continuous model of time.
- This substantially complicates the problem of automating verification.
- Except in very restricted circumstances, model checking based on state enumeration is infeasible or impossible.
- A reasonable alternative is to model time discretely.
- This can be done in normal, un-timed CSP by using an explicit event for each tick of a clock or any other time marker.

‘Tock’ Time

- Conventionally, the time event is called “$tock$” (since “tick” refers to ✓).
- Simple requirement: all processes with timing constraints must synchronise on $tock$.
- Advantages of this approach:
  - (i) great flexibility in modelling and reasoning (since the $tock$s can have many interpretations);
  - (ii) no need to learn the complicated Timed CSP model; and
  - (iii) FDR (and ProBE) can be used on such systems.

Tock Time: Example

- A timed version of the one-place buffer, $COPY$, might be:

$$TCOPY \quad = \quad left?x \rightarrow tock \rightarrow right!x \rightarrow tock \rightarrow TCOPY$$

- This says that it takes one time unit for a data value to pass through the buffer, and one more between output and the next input.

- So it *insists* on performing an action between each time unit.

- This is too prescriptive: the original $COPY$ will wait as long as you like to either input or output.

Tock Time: Better Example

- A better version of a timed $COPY$ is:

$$
\begin{aligned}
TCOPY2 &= left?x \rightarrow tock \rightarrow TCOPY2'(x) \\
&\quad \Box\ tock \rightarrow TCOPY2 \\[1em]
TCOPY2'(x) &= right!x \rightarrow tock \rightarrow TCOPY2 \\
&\quad \Box\ tock \rightarrow TCOPY2'(x)
\end{aligned}
$$

- The difference is that this says there is *at least* one time unit between input and output events.

- There is no limit to the time that can pass between events.

Tock Time: Other Examples

- Another version of a timed $COPY$ is:

$$\begin{aligned} TCOPY3 \quad &= \quad left?x \to tock \to right!x \to tock \to TCOPY3 \\ & \qquad \Box\, tock \to TCOPY3 \end{aligned}$$

- This is maybe more realistic: it will wait indefinitely for an input, but will not allow its output to be delayed.

- Other variations might put a bound on the time between communications.

- More complex versions might nondeterministically select when (and for how long) a communication is available.

Timing Requirements

- Adding the notion of time clearly allows us to be more expressive about how a process behaves.
- But it also makes it easier to write broken specifications.
- If two processes placed in parallel have inconsistent timing requirements then the combination can deadlock.

Timing Requirements: Simple Example

- Suppose we had a buffer that took longer to reset and accept another input:

$$
\begin{aligned}
SLOW\_TCOPY \quad &= \quad left?x \rightarrow tock \rightarrow \\
&\qquad right!x \rightarrow tock \rightarrow tock \rightarrow SLOW\_TCOPY
\end{aligned}
$$

- If we add this in parallel to our first version of a timed buffer, $TCOPY$, we get:

$$
\begin{aligned}
SLOW\_TCOPY \parallel TCOPY \quad &= \quad left?x \rightarrow tock \rightarrow \\
&\qquad right!x \rightarrow tock \rightarrow STOP
\end{aligned}
$$

- The deadlock happens because $SLOW\_TCOPY$ is only willing to do another $tock$, whereas $TCOPY$ is only willing to do an input.

Timing Requirements: Time Stop

- Deadlock due to mismatched timing requirements is known as *time-stop*.
- Because the result is deadlock, it is easy to find using FDR.
- Notice that $SLOW\_TCOPY$ does not deadlock in combination with the other versions of the timed buffer ($TCOPY2$ and $TCOPY3$), since they allow any amount of time between output and input.
- $TCOPY$ is very inflexible in terms of timing.

Timing Requirements: Another Example

- A more reasonable and realistic constraint might be:

$$
\begin{array}{lcl}
TLimiter(n, m) & = & \text{if } n = 0 \text{ then } RUN \\
 & & \text{else } tock \rightarrow TLimiter(n-1, m) \\
 & & \quad \Box \\
 & & \quad (\text{if } m = 0 \text{ then} \\
 & & \quad\quad STOP \\
 & & \quad \text{else} \\
 & & \quad\quad \displaystyle\Box_{x \neq tock} \; x \rightarrow TLimiter(n, m-1))
\end{array}
$$

- The process $TLimiter(N, M)$ permits at most M communications in the first N time units.

Timing Requirements: Time Limiter

- It's not immediately obvious whether adding $TLimiter$ as a constraint can introduce time-stop.
- Recall that $TCOPY$ insists on doing one communication in each time unit.
- So, combining it with $TLimiter(N, M)$ will cause time-stop if $M < N$.
- Again, $TCOPY2$ is much more relaxed about timing, so combining with $TLimiter$ will not introduce time-stop.
- In this case, $TLimiter$ will act as intended and limit the communications of the buffer.
- What about $TCOPY3$? Is that OK?

Timing Requirements: Two-Place Buffer

- Another version of timed $COPY$, parameterised by the delay between input and output:

$$\begin{array}{rcl} TCOPY4(D) & = & left?x \to TCOPY4'(D, D, x) \\ & & \Box\ tock \to TCOPY4(D) \\ TCOPY4'(D, n, x) & = & \text{if } n > 0 \text{ then} \\ & & \quad tock \to TCOPY4'(D, n-1, x) \\ & & \text{else} \\ & & \quad right!x \to tock \to TCOPY4(D) \end{array}$$

- This can be chained to create a two-place (timed) buffer (with extra synchronisation on $tock$):

$$TWO\_TCOPY(D_1, D_2) \quad = \quad TCOPY4(D_1) \gg TCOPY4(D_2)$$

Timing Requirements: Producer/Consumer

- The chaining is actually a parallel combination (plus renaming and hiding):

$$\begin{array}{rcl} TWO\_TCOPY(D_1, D_2) & = & (TCOPY4(D_1)[\![{}^{mid}/_{right}]\!] \\ & & \underset{\{\mid mid, tock \mid\}}{\parallel} \\ & & TCOPY4(D_2)[\![{}^{mid}/_{left}]\!]) \quad \setminus \{\mid mid \mid\} \end{array}$$

- From the view of the internal communication in $P \gg Q$, P can be thought of as a producer and Q a consumer.

- By convention, P produces on $right$ for Q, and Q consumes these outputs as inputs on $left$.

- Because of the parallel combination, there is potential for time-stop.

Timing Requirements: Producer/Consumer

- TWO_TCOPY is parameterised by the output delays for each side of the chaining.
- If the producer side produces too quickly for the consumer, the system can deadlock (i.e., time-stop).
- This manifests itself when $D_1 < D_2$.
- The problem is when the producer has waited long enough and is now only willing to produce its new output, whereas the consumer is still waiting to get rid of its last input.
- Of course, this can be avoided if the environment is constrained to be sufficiently slow in creating inputs to TWO_TCOPY.

Timing Requirements: Infinite Actions

- In most systems it is not sensible to permit an infinite number of actions between two $tock$s.
- This would mean an infinite amount of activity could occur in a finite time interval.
- To test for this we can check whether the following process diverges:

$$P \setminus (\Sigma - \{tock\})$$

Timing Requirements: Sanity Check

- The new, finite actions (livelock) test and the old, time-stop (deadlock) tests can be combined into one refinement check.

- If $P \setminus (\Sigma - \{tock\})$ refines (in the failures-divergences model):

$$TOCKS \quad = \quad tock \rightarrow TOCKS$$

  then it must be livelock and deadlock free.

- $TCOPY$, $TCOPY2$ and $TCOPY3$ all satisfy this check.

Timing Requirements: Non-Instant Communication

- It may be that we need a process to be tolerant of not receiving some or all communications instantly.
- The most obvious distinction is between *inputs* and *outputs*.
- It may be reasonable to assume the environment is always willing to accept an output, but unreasonable to assume that it is always waiting to send an input.
- *[This has a lot in common with concepts (like signal events) that we'll meet later when dealing with abstraction.]*

Timing Requirements: Non-Instant Communication

- If there is a set of events D which the environment is assumed to be allowed to delay indefinitely, but may still allow, we can modify the sanity check to test whether $TOCKS$ is refined by:

$$(P \underset{D}{\|} Chaos_D) \setminus (\Sigma - \{tock\})$$

- The $Chaos_D$ component may either allow or prevent actions from D, so P can neither discount the occurrence of these events nor rely on them happening.

- If D were $\Sigma - \{tock\}$, this checks whether *all* normal events could be arbitrarily delayed by the environment. (*The conventional CSP view of communication.*)

- With $D = \{| \ left \ |\}$ we have that $TCOPY3$ satisfies this requirement, but $TCOPY$ does not.

Case Study: Level Crossing

- Model of a level crossing, where a road crosses a railway track.
- This uses a gate to stop cars getting onto the railway line when a train is passing.
- We'll use timing to:
  (i) model untimed safety properties (e.g., when a train is near or on the gate, the gate should be down); and

  (ii) specify timing properties of the system (e.g., the gate goes up within K time units when there is no train near or approaching).

Level Crossing: Modelling Approach

- We'll divide the system into a number of process representing key parts:
  - the track (in segments);
  - trains;
  - timing constraints;
  - the gate; and
  - controllers, monitoring sensors and activating the gate.
- For simplicity, we'll consider the case of a single line along which trains move in only one direction.

Level Crossing: More Modelling

- We will not rely on timing for anything other than the properties of the gate.
- Thus, we will not consider safety requirements such as never having trains trains on the same track segment.
- We'll model track segments and trains as untimed processes, and later add timed behaviour through parallel combination.

Level Crossing: Segments

- We'll assume that the track is broken into segments (which are at least as long as any train).
- We'll treat segment 0 as a 'notional' segment, representing the whole of the outside world.
- Segment 1 (the first real one a train will enter) will contain the sensor which detects new arrivals.
- Segment $GateSeg$ (the segment containing the gate) contains a sensor to detect departures.

Level Crossing: Trains

- The untimed description of a train t on track segment j:

$$Train(t, j) \quad = \quad enter.t.j \oplus 1 \rightarrow leave.t.j \rightarrow Train(t, j \oplus 1)$$

  (where $\oplus$ is addition modulo SC, the segment count).

- Notice that the train is seen to enter the next segment before it is seen to leave its current one.

- Leaving the last segment puts the train back in the outside world (segment 0).

Level Crossing: Gate Segment

- $GateSeg$ is somewhere between 1 and $SC - 1$.
- If it were the first, it wouldn't be possible to get the gate down in time (not enough notification).
- For our model we also need at least one segment after the gate, to ensure trains leave this important segment.
- So, $GateSeg$ is probably best as $SC - 2$ (the penultimate segment) and SC needs to be at least 4.

Level Crossing: Tracks

- A track segment can be occupied by one train at a time.
- Each time a train enters segment 1 or leaves segment $GateSeg$, a sensor fires.
- The untimed description of a segment:

$$
\begin{aligned}
Track(j) \quad &= \quad \text{if } j = 1 \\
&\quad\quad \text{then } enter?t!j \rightarrow sensor.in \rightarrow TrackF(t, j) \\
&\quad\quad \text{else } enter?t!j \rightarrow TrackF(t, j) \\
TrackF(t, j) \quad &= \quad \text{if } j = GateSeg \\
&\quad\quad \text{then } leave.t.j \rightarrow sensor.out \rightarrow Track(j) \\
&\quad\quad \text{else } leave.t.j \rightarrow Track(j)
\end{aligned}
$$

Level Crossing: Trains and Tracks

- The trains do not communicate with each other and neither do the track segments:

$$Trains \;=\; \mathop{|||}_{t \in TRAINS} Train(t, 0)$$

$$Tracks \;=\; \mathop{|||}_{i \in \{1..SC-1\}} Track(i)$$

where $TRAINS$ is a set of train identifiers, such as names (or numbers).

Level Crossing: Network

- The trains and tracks do need to cooperate on $enter$ and $leave$ events, except those involving segment 0.

$$Network \quad = \quad Trains \underset{IFace}{\parallel} Tracks$$

Where:

$$IFace = \{enter.t.j, leave.t.j \mid t \in TRAINS \wedge j \in \{1..SC-1\}\}$$

- We could prove some safety properties about this system, such as there are never two trains on the same track segment, but we want to examine timing.

Level Crossing: Timing Assumptions

- Suppose there are constants $MinTocksPerSegment$ and $MaxTocksPerSegment$ which represent the minimum and maximum number of time units a train can spend in a (real) track segment.
- We could add this timing constraint to the trains or the tracks part of the network.
- For the former (trains), this could be a constraint on the time between $enter$ events (plus a special case for behaviour on the notional segment).
- For the latter (tracks), it could be a constraint on the time between a train entering and leaving a segment ($enter/leave$), or between entering one segment and the next ($enter/enter$).

Level Crossing: Track Timing

- The $enter/leave$ approach has the disadvantage that the length of the train affects its speed.
- The $enter$ event is triggered by the front of the train, and the $leave$ event by the back.
- The longer the train, the greater the time between $enter$ and $leave$, for the same train speed.
- So, we'll regulate the speed using the $enter/enter$ approach on track segments.

Level Crossing: Speed Regulator

- The speed regulator on a particular track segment:

$$
\begin{aligned}
SpeedReg(j) &= enter?t!j \rightarrow SpeedReg'(j, 0) \\
SpeedReg'(j, n) &= (\text{if } n < MaxTocksPerSeg \\
&\quad \text{then } tock \rightarrow SpeedReg'(j, n+1) \text{ else } STOP) \\
&\quad \Box \\
&\quad (\text{if } n >= MinTocksPerSeg \\
&\quad \text{then } enter?t!j \oplus 1 \rightarrow SpeedReg(j) \text{ else } STOP)
\end{aligned}
$$

- Slight modifications would allow speed limits to vary from segment to segment, or a change from this ‘track-centred’ constraint to a ‘train-centred’ one.

Level Crossing: Sensor Firing

- The only other timing constraints we need are on the sensors, which must fire within one time unit:

$$
\begin{aligned}
\mathit{InSensorTiming} \quad &= \quad \mathit{enter}?t!1 \rightarrow \mathit{sensor.in} \rightarrow \mathit{InSensorTiming} \\
&\qquad \Box\ \mathit{tock} \rightarrow \mathit{InSensorTiming}
\end{aligned}
$$

$$
\begin{aligned}
\mathit{OutSensorTiming} \quad &= \quad \mathit{leave}?t!\mathit{GateSig} \rightarrow \mathit{sensor.out} \rightarrow \mathit{OutSensorTiming} \\
&\qquad \Box\ \mathit{tock} \rightarrow \mathit{OutSensorTiming}
\end{aligned}
$$

- We now have timing constraints on only the track segments.

- To add these to the network we could add each one to the relevant component processes, or add all of them together *en masse*. . .

Level Crossing: Combined Timing Constraints

- The parallel combinations of the timing constraints:

$$SpeedRegs \quad = \quad \Vert_{j=1}^{SC-1} (SpeedReg(j), \alpha SR(j))$$

$$\text{where} \quad \alpha SR(j) \quad = \quad \{tock, enter.t.j, enter.t.j \oplus 1 \mid t \in TRAINS\}$$

$$SensorTiming \quad = \quad InSensorTiming \underset{\{tock\}}{\Vert} OutSensorTiming$$

$$NetworkTiming \quad = \quad SpeedRegs \underset{S}{\Vert} SensorTiming$$

$$\text{where} \quad S \quad = \quad \{tock\} \cup \{enter.t.1 \mid t \in TRAINS\}$$

Level Crossing: Timed Network

- The timing constraints can now be added to the network:

$$
\begin{aligned}
TimedNetwork &= Network \underset{T}{\parallel} NetworkTiming \\
\text{where} \quad T &= \{leave.t.GateSeg \mid t \in TRAINS\} \\
&\quad \cup \{\mid enter, sensor \mid\}
\end{aligned}
$$

Level Crossing: The Gate

- The gate is operated by a controller process.
- We can either build the timing behaviour of the gate into the controller process or a separate gate process.
- The latter approach is better if there is any nondeterminism about the gate timing.
- So, we'll have separate controller and gate processes.
- There are four basic states of the controller: *up*, *down*, *going up* and *going down*.

Level Crossing: Controller *Up*

- In this state, the controller lets time pass or receives a signal from the 'incoming train' sensor (in segment 1).
- The reaction to the sensor is to signal the gate to go down.
- This must happen immediately (i.e., before the next $tock$).
- The process description:

$$\begin{aligned} ControllerUp \quad = \quad & sensor.in \rightarrow gate!godown \rightarrow ControllerGoingDown(1) \\ & \Box\ tock \rightarrow ControllerUp \end{aligned}$$

Level Crossing: Controller *Up*

- The first attempt at $ControllerUp$ seems reasonable, but what happens if $sensor.out$ happens in this state?
- As it stands, $ControllerUp$ would simply refuse this event.
- Refusing to look at a problem does not mean it isn't there!
- So, we could either:
  (i) prove later that this can never happen (the gate should be down!);
  (ii) note that $OutSensorTiming$ would cause a time-stop; or
  (iii) be 'up front' about the possibility, and add a clause which accepts the event and handles it in some way.

Level Crossing: Controller *Up*

- The last choice (be 'up front') is also useful for debugging, so the revised process description is:

$$\begin{aligned} ControllerUp \quad = \quad & sensor.in \rightarrow gate!godown \rightarrow ControllerGoingDown(1) \\ & \Box\ sensor.out \rightarrow ERROR \\ & \Box\ tock \rightarrow ControllerUp \end{aligned}$$

- In general, if a process is one that simply observes an event, rather than being able to refuse it, it is good practice to specify the process so that this is obviously adhered to!

- Suitable candidates for $ERROR$ are those processes that have an effect on the whole network and are easy to check for (e.g., **div** or $error \rightarrow STOP$).

Level Crossing: Controller *Down* and *Going Down*

- The *down* and *going down* states need to record the number of trains that must pass before the gate can go up.
- Each time the sensor detects a new train arriving this count is increased.
- But this would naturally make an infinite state process.
- However, there is a suitable limit for the count: the number of segments before the gate.
- This is because each track segment allows at most one train to be on it at a time.

Level Crossing: Controller *Going Down*

$$
\begin{aligned}
ControllerGoingDown(n) \quad = \quad & tock \rightarrow ControllerGoingDown(n) \\
& \Box\ gate.down \rightarrow ControllerDown(n) \\
& \Box\ sensor.out \rightarrow ERROR \\
& \Box\ (\text{if } n > GateSeg \text{ then} \\
& \qquad ERROR \\
& \quad\ \text{else} \\
& \qquad sensor.in \rightarrow ControllerGoingDown(n+1))
\end{aligned}
$$

Level Crossing: Controller *Down*

$$
\begin{array}{lll}
ControllerDown(n) & = & tock \rightarrow ControllerDown(n) \\
& & \Box\ sensor.out \rightarrow \\
& & \qquad (\text{if } n = 1 \text{ then} \\
& & \qquad\quad gate!goup \rightarrow ControllerGoingUp \\
& & \qquad \text{else} \\
& & \qquad\quad ControllerDown(n-1)) \\
& & \Box\ (\text{if } n > GateSeg \text{ then} \\
& & \quad ERROR \\
& & \quad \text{else} \\
& & \quad sensor.in \rightarrow ControllerDown(n+1))
\end{array}
$$

Level Crossing: Controller *Going Up*

- Just like for $ControllerUp$, it is an error if the other sensor fires.

$$
\begin{aligned}
ControllerGoingUp \quad = \quad & tock \rightarrow ControllerGoingUp \\
& \Box\ gate.up \rightarrow ControllerUp \\
& \Box\ sensor.in \rightarrow \\
& \qquad gate!godown \rightarrow ControllerGoingDown(1) \\
& \Box\ sensor.out \rightarrow ERROR
\end{aligned}
$$

Level Crossing: Gate *Up* and *Down*

- The gate process simply reacts to $gate$ commands.
- The movement from one position to another takes a certain amount of time (the elapsed time is recorded as a parameter to the $Going$ states).

$$
\begin{aligned}
GateUp \quad &= \quad gate.goup \rightarrow GateUp \\
&\qquad \Box\; gate.godown \rightarrow GateGoingDown(0) \\
&\qquad \Box\; tock \rightarrow GateUp
\end{aligned}
$$

$$
\begin{aligned}
GateDown \quad &= \quad gate.goup \rightarrow GateGoingUp(0) \\
&\qquad \Box\; gate.godown \rightarrow GateDown \\
&\qquad \Box\; tock \rightarrow GateDown
\end{aligned}
$$

Level Crossing: Gate *Going Down*

- The *going down* state takes a nondeterministic amount of time to get to *down*.
- $DownTime$ is the assumed bound on this delay.

$$
\begin{array}{rcl}
GateGoingDown(n) & = & gate.goup \rightarrow GateGoingUp(0) \\
 & & \Box\ gate.godown \rightarrow GateGoingDown(n) \\
 & & \Box\ (\text{if } n < DownTime \text{ then} \\
 & & \qquad gate.down \rightarrow GateDown \\
 & & \qquad\ \sqcap\ tock \rightarrow GateGoingDown(n+1) \\
 & & \quad \text{else} \\
 & & \qquad gate.down \rightarrow GateDown
\end{array}
$$

Level Crossing: Gate *Going Up*

- Similarly, $UpTime$ is the assumed bound on delay in getting the gate up.

$$
\begin{array}{rcl}
GateGoingUp(n) & = & gate.goup \rightarrow GateGoingUp(n) \\
 & & \Box\ gate.godown \rightarrow GateGoingDown(0) \\
 & & \Box\ (\text{if } n < UpTime \text{ then} \\
 & & \qquad gate.up \rightarrow GateUp \\
 & & \qquad \sqcap\ tock \rightarrow GateGoingUp(n+1) \\
 & & \quad \text{else} \\
 & & \qquad gate.up \rightarrow GateUp
\end{array}
$$

Level Crossing: The System

- The gate and the controller can be combined, synchronising on $tock$ (since they both use timing) and all communications on $gate$:

$$GateAndController \quad = \quad ControllerUp \underset{\{|tock, gate|\}}{\parallel} GateUp$$

- The complete system is the trains and tracks with the gate:

$$SYSTEM \quad = \quad TimedNetwork \underset{\{|tock, sensor|\}}{\parallel} GateAndController$$

- The initial state has the gate up and no trains in the real track segments.

Level Crossing: Sanity Check

- For our complete check we need to identify any events that could be considered delayable by the environment.
- The only events that the system does not rely on happening within some bound are those that represent the trains' entries into segment 1.
- So $D = \{enter.t.1 \mid t \in TRAINS\}$ and the check is to test the following process against $TOCKS$:

$$(SYSTEM \underset{D}{\parallel} Chaos_D) \setminus (\Sigma - \{tock\})$$

Level Crossing: Safety Requirement

- The safety requirement is that the gate is always down when there is a train on the gate segment.
- This can be divided into:
  (i) When a train enters $GateSeg$, the gate is down.
  (ii) The gate never goes up when there is a train on $GateSeg$.
- These correspond to:
  (i) When a train enters $GateSeg$, the most recent, non-$tock$ event communicated by the gate process is $gate.down$.
  (ii) No $gate$ event occurs between a train entering $GateSeg$ and leaving it.

Level Crossing: Safety Specifications

- Both requirements are elementary safety (i.e., trace) specifications.
- We can define a generic process $BetweenSets(A, B, C)$ that allow events in C to happen only between an element in A and B.
- That is, A events can be thought of as enabling C events, and B events as disabling C events.
- We can define $OutsideSets(A, B, C)$ to be the same except initially able to do C events (as if an A had happened).

Level Crossing: Safety Specifications

- $BetweenSets'(A, B, C)$ and $OutsideSets'(A, B, C)$ that consider only events in $A \cup B \cup C$ can be defined in terms of each other:

$$\begin{aligned}
BetweenSets'(A, B, C) \quad = \quad & \mathop{\Box}_{x \in A} x \rightarrow OutsideSets'(A, B, C) \\
& \Box \mathop{\Box}_{x \in B} x \rightarrow BetweenSets'(A, B, C)
\end{aligned}$$

$$\begin{aligned}
OutsideSets'(A, B, C) \quad = \quad & \mathop{\Box}_{x \in C} x \rightarrow OutsideSets'(A, B, C) \\
& \Box \mathop{\Box}_{x \in A} x \rightarrow OutsideSets'(A, B, C) \\
& \Box \mathop{\Box}_{x \in B} x \rightarrow BetweenSets'(A, B, C)
\end{aligned}$$

Level Crossing: Safety Specifications

- $BetweenSets$ and $OutsideSets$ are the promotions to the complete alphabet.

- They don't care what happens outside of $A \cup B \cup C$, so the most nondeterministic process ($Chaos$) is added in:

$$\begin{aligned} BetweenSets(A, B, C) &= BetweenSets'(A, B, C) \underset{A \cup B \cup C}{\parallel} Chaos_{\Sigma} \\ OutsideSets(A, B, C) &= OutsideSets'(A, B, C) \underset{A \cup B \cup C}{\parallel} Chaos_{\Sigma} \end{aligned}$$

Level Crossing: Safety Requirements

- The two safety requirements can now be expressed as:

$$
\begin{aligned}
EnterWhenDown &= BetweenSets(\{gate.down\}, \\
&\qquad \{gate.up, gate.goup, gate.godown\}, \\
&\qquad \{enter.t.GateSeg \mid t \in TRAINS\})
\end{aligned}
$$

$$
\begin{aligned}
GateStillWhenTrain &= OutsideSets(\{leave.t.GateSeg \mid t \in TRAINS\}, \\
&\qquad \{enter.t.GateSeg \mid t \in TRAINS\}, \\
&\qquad \{\mid gate \mid\})
\end{aligned}
$$

Level Crossing: Safety Requirement

- The requirements can be combined as:

$$Safety \quad = \quad EnterWhenDown \parallel GateStillWhenTrain$$

- Whether the $SYSTEM$ satisfies this specification depends on the parameters:
  (i) The number of track segments (SC).

  (ii) The speed limits ($MinTocksPerSegment$ and $MaxTocksPerSegment$).

  (iii) The speed of the gate movement ($UpTime$ and $DownTime$).

- The safety requirement is not really affected by the number of trains.

- Time is not mentioned in the safety specification, but timing is critical to its satisfaction!

Level Crossing: Other Specifications

- Another safety specification: the gate does not go down unless there is a train on a (real) segment of the track.
- Liveness: the gate must be up within a certain time of the last train leaving, and it must remain up until another train enters.
- Because of the $TOCKS$ sanity check, we know the system is live with respect to $tock$, so the above liveness condition can be simplified to a safety/trace specification!