

Integrating Concurrency Theory with State

Steve Schneider
University of Surrey, UK

Joint work with Helen Treharne

Challenge

- How can we model and reason about systems with complex and detailed state information and complex concurrent or control behaviour?
- How far can we get by integrating formal methods?

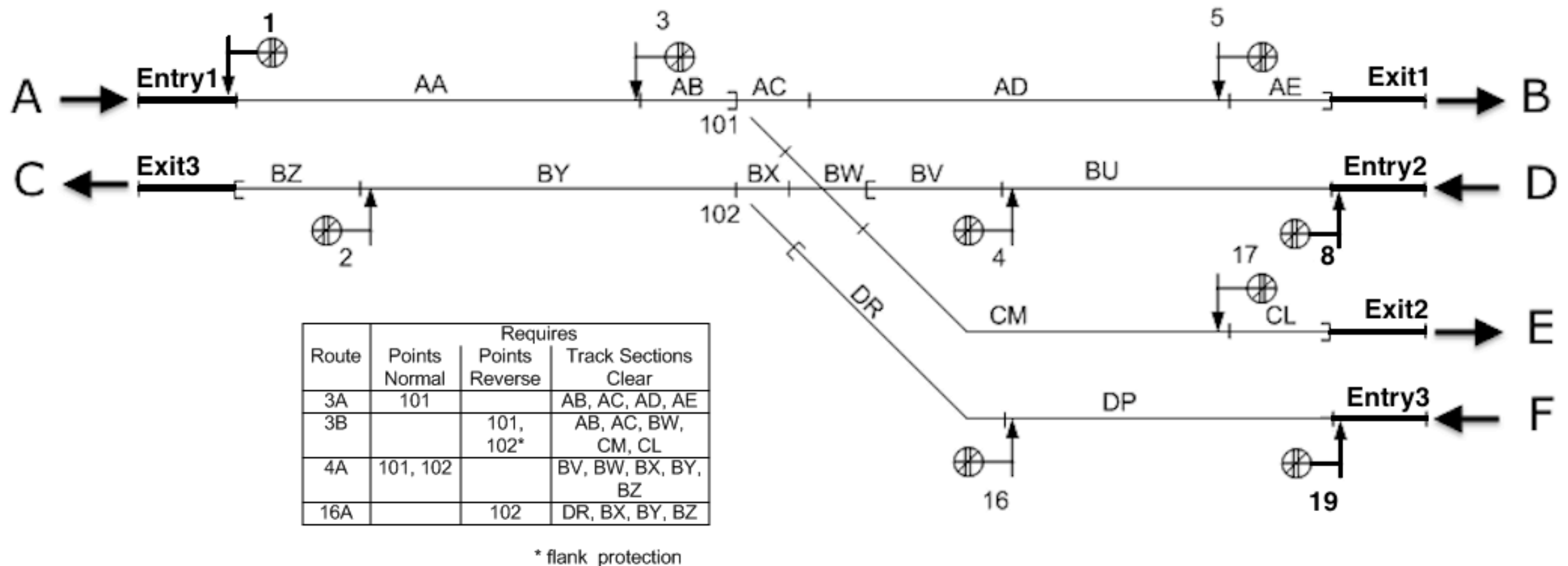
State- and event-based formalisms

- Process algebra/concurrency theory:
 - Good for complex interaction between processes
 - Focus on patterns of communication and control
 - Abstract state as much as possible
 - State tends to be mostly orthogonal
- **Challenge:** handling rich state information

State- and event-based formalisms

- State-based formalisms (Z, B, VDM,...)
 - Suited to data-rich systems and structured state
 - State transitions: specify how and when state is updated
 - Weak or cumbersome notions of flow of control and interaction
- **Challenge:** want to consider higher level behavioural properties (e.g. traces, LTL)

Motivating Example: railways



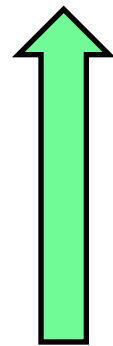
State:

train locations;
points positions;
signals; locks on routes;
control tables; topology

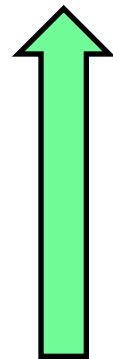
Trains:
driving rules;

Road map

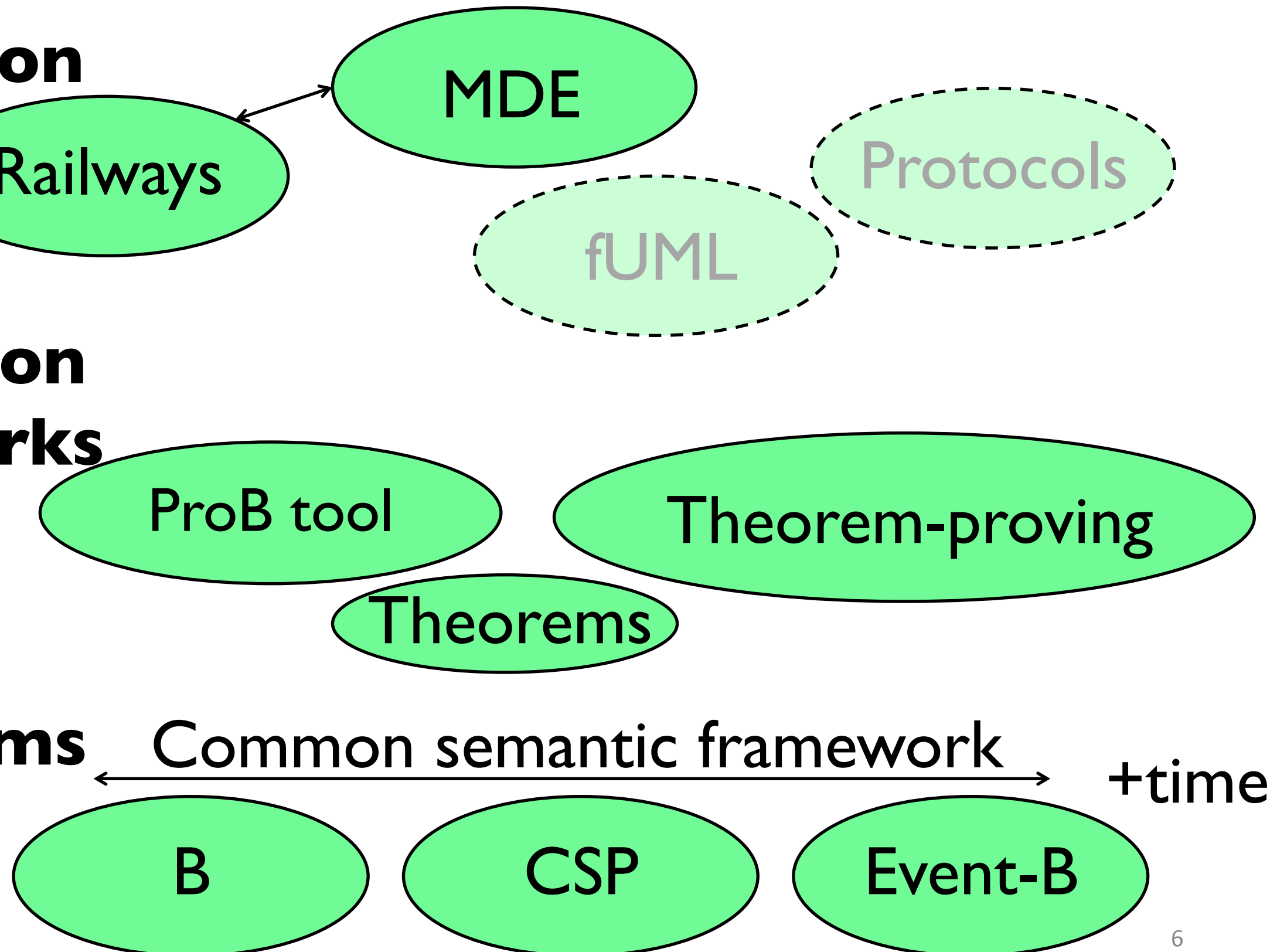
Application



Verification frameworks



Formalisms



Structure of Talk

- B and Event-B for state
- Combination with CSP
- Example: modelling for railway safety
- Incorporating time

Structure of Talk

- **B and Event-B for state**
- Combination with CSP
- Example: modelling for railway safety
- Incorporating time

Classical B machines and control

Machine M1 =
Variables n
Invariant $n \in \{0,1\}$
Initialisation $n:=0$
Operations
 up=
 pre $n=0$ then $n:=n+1$ end
down =
 pre $n = 1$ then $n:=n-1$ end

n: 0

* Operations always enabled, but the machine will break (diverge) if they are called outside their preconditions

Classical B machines and control

Machine M1 =

Variables n

Invariant $n \in \{0,1\}$

Initialisation $n:=0$

Operations

up=

pre $n=0$ then $n:=n+1$ end

down =

pre $n = 1$ then $n:=n-1$ end

n: 1

- * Operations always enabled, but the machine will break (diverge) if they are called outside their preconditions
- * up can occur as the first event, setting n to 1.

Classical B machines and control

Machine M1 =

Variables n

Invariant $n \in \{0,1\}$

Initialisation $n:=0$

Operations

up=

pre $n=0$ then $n:=n+1$ end

down=

pre $n = 1$ then $n:=n-1$ end

n: 0

- * Operations always enabled, but the machine will break (diverge) if they are called outside their preconditions
- * up can occur as the first event
- * down can occur as the second event

Classical B machines and control

Machine M1 =

Variables n

Invariant $n \in \{0,1\}$

Initialisation $n:=0$

Operations

up=

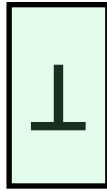
pre $n=0$ then $n:=n+1$ end

down

=

pre $n = 1$ then $n:=n-1$ end

n:



- * Operations always enabled, but the machine will break (diverge) if they are called outside their preconditions
- * up can occur as the first event
- * down can occur as the second event
- * a second occurrence of down will diverge the machine

Classical B machines and control

Machine M1 =
Variables n
Invariant $n \in \{0,1\}$
Initialisation $n:=0$
Operations
 up=
 pre $n=0$ then $n:=n+1$ end
 down =
 pre $n = 1$ then $n:=n-1$ end

n: 0

* Operations always enabled, but the machine will break (diverge) if they are called outside their preconditions

- preconditions do not determine control flow
- they constrain what control flow is necessary to ensure correct behaviour

Classical B machines and control

Machine M1 =
Variables n
Invariant $n \in \{0,1\}$
Initialisation $n:=0$
Operations
 up=
 pre $n=0$ then $n:=n+1$ end
down =
 pre $n = 1$ then $n:=n-1$ end

n: 0

* Operations always enabled,
but the machine will break
(diverge) if they are called
outside their preconditions

$$P = up \rightarrow down \rightarrow P$$



Variant: Event-B for modelling

Machine M1 =
Variables n
Invariant $n \in \{0,1\}$
Initialisation $n:=0$
Event up =
 when $n=0$ then $n:=n+1$ end
Event down =
 when $n = 1$ then $n:=n-1$ end

n: 0

- * Events are enabled when their guard is true, and **blocked** when their guard is false
- * up can occur as the first event
- * down cannot occur, and will be blocked

- * control variables in the guards used to manage control flow
- * very cumbersome in practice, and scales badly

B machine semantics

- **Proof obligations** for a consistent machine
 - Initialisation establishes the invariant
 - Operations preserve the invariant: they are guaranteed to establish it when called within their preconditions
 - Machine Invariant captures safety properties on states
- wp semantics used:
 - for Invariant I and operation **PRE P THEN S END**:

$$I \wedge P \Rightarrow wp(S, I)$$

Structure of Talk

- B and Event-B for state
- **Combination with CSP**
- Example: modelling for railway safety
- Incorporating time

CSP(-lite)

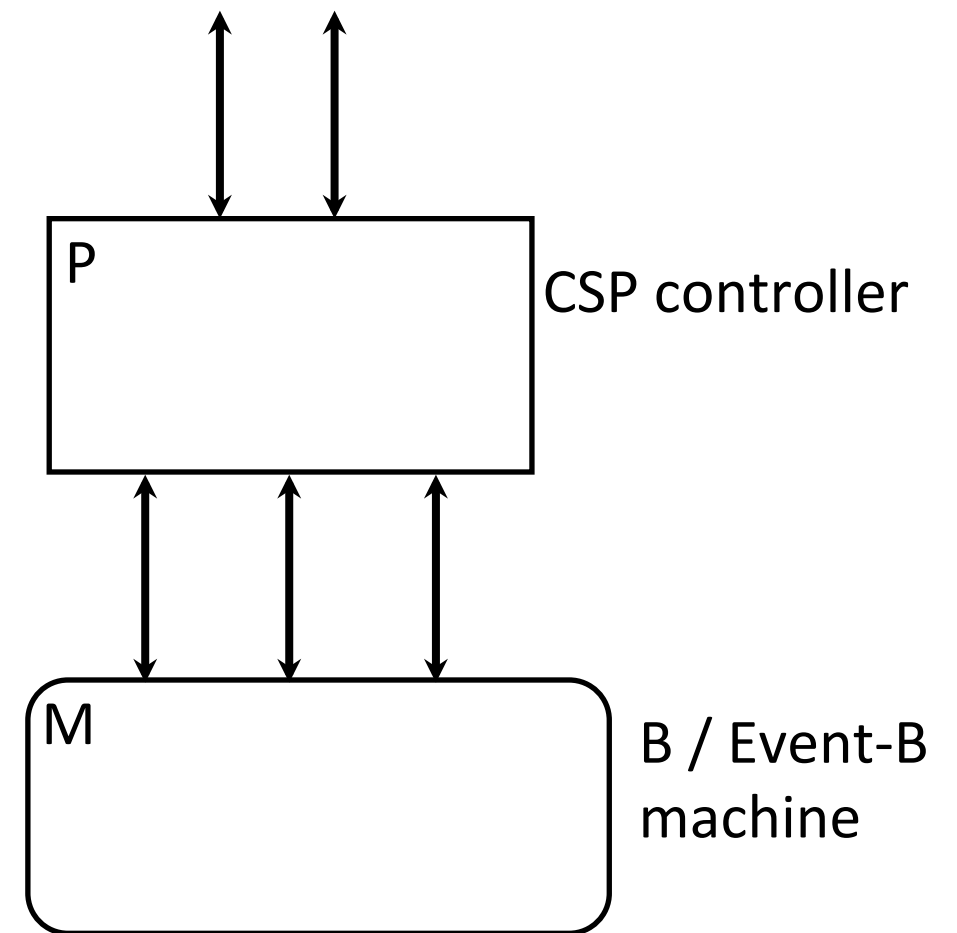
$$\begin{aligned} P \quad ::= & \quad STOP \mid SKIP \mid P; Q \mid \\ & a \rightarrow P \mid c?x \rightarrow P \mid c!v \rightarrow P \mid \\ & P \sqcap Q \mid P \sqcup Q \mid \\ & P \parallel Q \mid P ||| Q \mid P \setminus A \\ & X \mid \mu X . P \end{aligned}$$

CSP semantics: sets of observations

- **Traces:** finite sequences of (visible) events:
 $\text{traces}(P) = \{ \text{tr} \mid \text{tr can be observed of } P \}$
- **Failures:** traces together with refusal sets:
 $\text{failures}(P) = \{ (\text{tr}, X) \mid (\text{tr}, X) \text{ can be observed of } P \}$
- **Divergences:** traces during which the process may diverge:
 $\text{divergences}(P) = \{ \text{tr} \mid P \text{ may diverge when performing } \text{tr} \}$
- **Infinite traces:** infinite sequences of (visible) events:
 $\text{infinities}(P) = \{ u \mid u \text{ can be observed of } P \}$

Combining CSP and B

- Some operations in the machine correspond to events in the controller
- The interface is the alphabet of M
- Divergence can occur if an operation of M is called outside its precondition (Classical B)
- Consistency: when P ensures that M does not diverge.



Semantic approach

- The B machine is given a CSP semantics.
- Morgan's wp semantics (1990) for action systems is applicable here, to give CSP semantics for B machines

$$\overline{wp}(S, P) = \neg wp(S, \neg P)$$

$$traces(M) = \{ \langle a_1, a_2, \dots, a_n \rangle \mid \overline{wp}(init; a_1; a_2; \dots; a_n, true) \}$$

$$failures(M) = \{ \langle (a_1, a_2, \dots, a_n, X) \rangle \mid \overline{wp}(init; a_1; a_2; \dots; a_n, \bigwedge_{a \in X} \neg g_a) \}$$

$$divergences(M) = \{ \langle a_1, a_2, \dots, a_n \rangle \mid \overline{wp}(init; a_1; a_2; \dots; a_n, false) \}$$

CSP | B semantic foundation

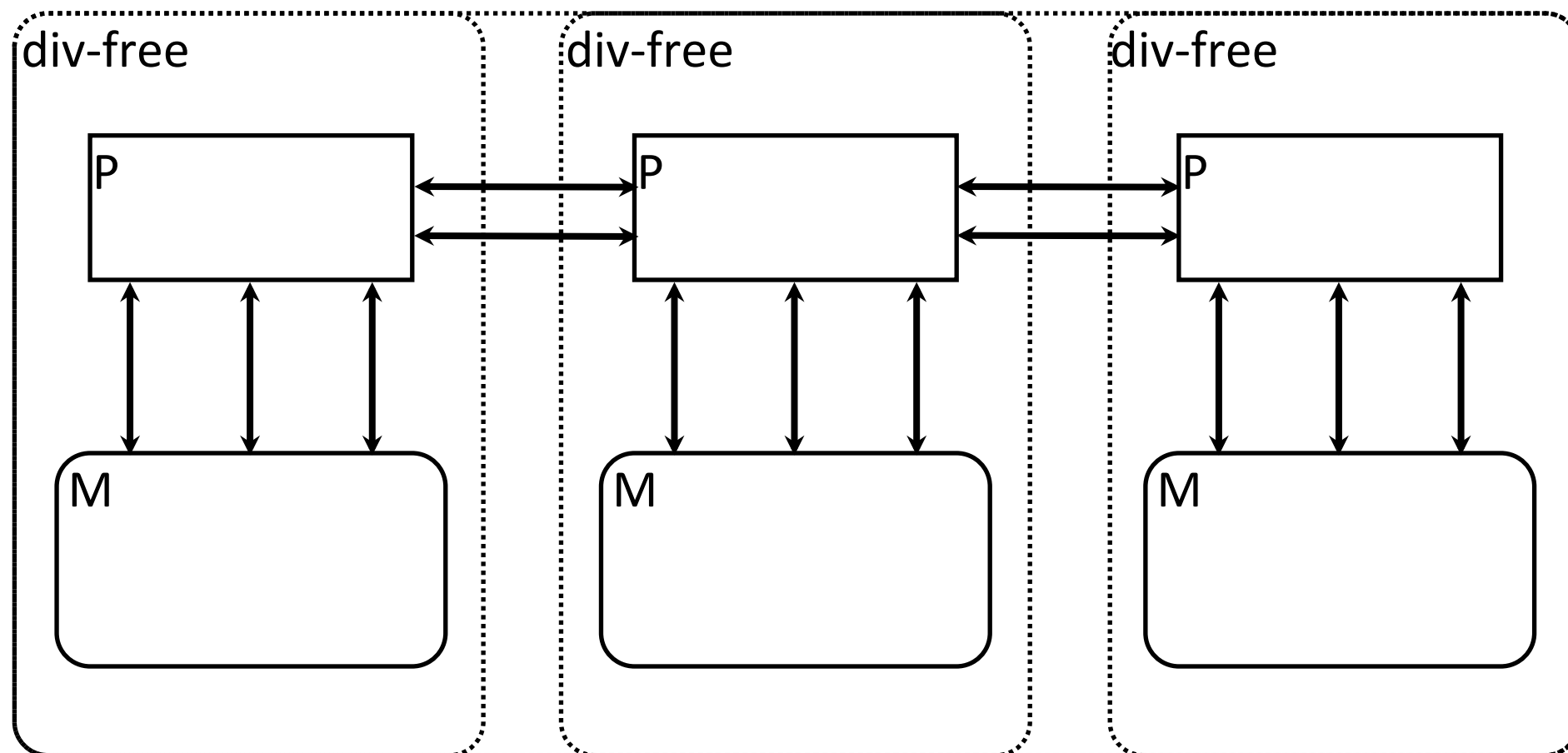
- Once M has a CSP semantics then it can be treated as a CSP process, and combined using the CSP operators.
- Thus a controlled component $P \mid M$ can be given a CSP semantics.
- Want access to tool support available for B and for CSP, and lift the results to the combination.

CSP || B and CSP || Event-B

- Various results, theorems and proof rules have been obtained for (combinations of) controlled components:
 - deadlock-freedom (conditions for just checking CSP)
 - divergence-freedom (control loop invariants)
 - refinement is compositional (particular proof rules available to maintain liveness of machines)
 - model-checking with ProB (invariant checking, LTL, CTL ...)

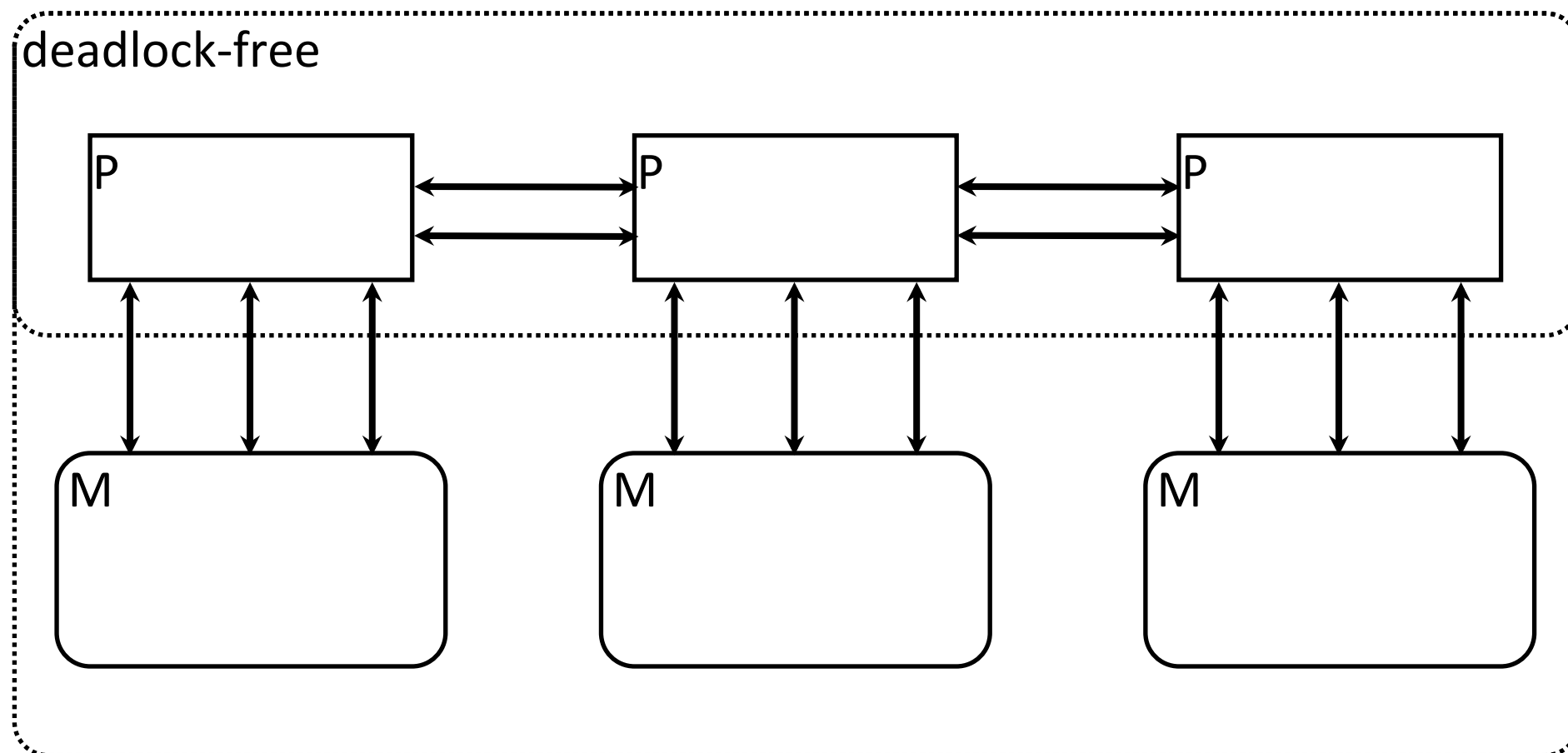
Composition results

- If each $CSP \mid B$ controlled component is divergence-free... (*B theorems available*)
...then so is their combination



Composition results

- If the combination of CSP controllers is deadlock-free... (...and the combination is divergence-free...)
...then so is their CSP || B combination



Structure of Talk

- B and Event-B for state
- Combination with CSP
- **Example: modelling for railway safety**
- Incorporating time

Application to Railway modelling

Steve Schneider
Helen Treharne
Matthew Trumble

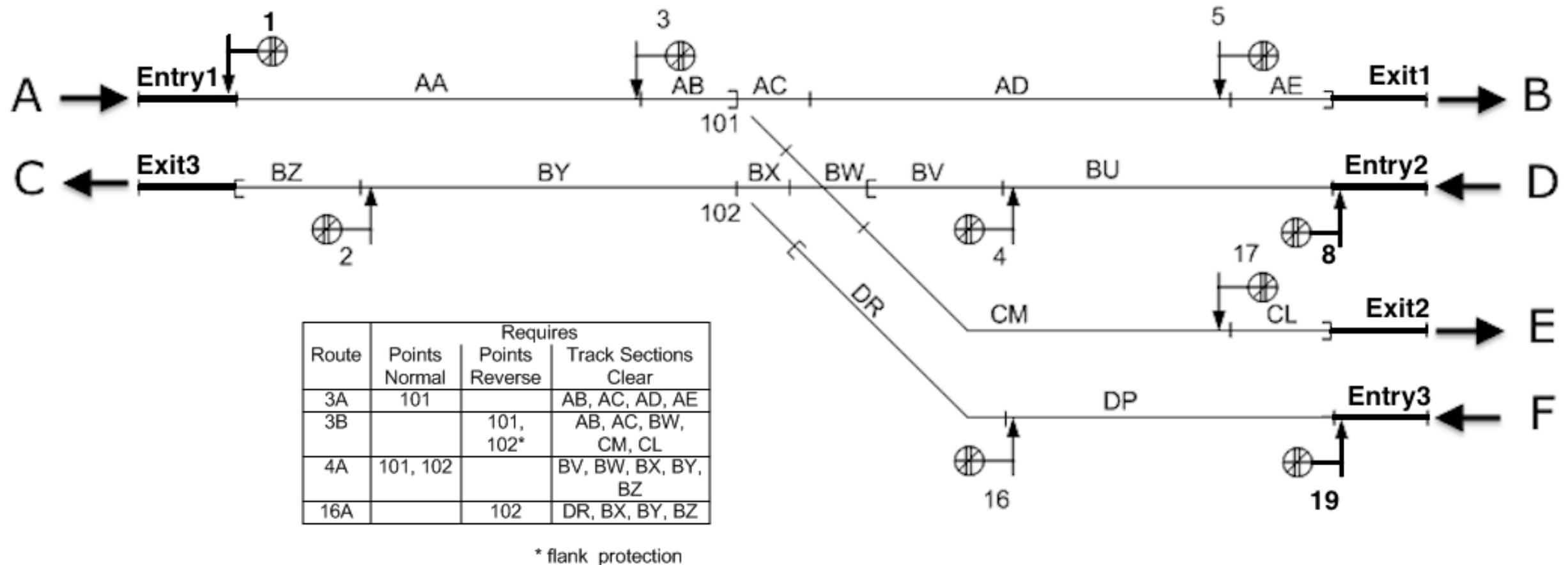


Markus Roggenbach
Faron Moller
Nga Hoang Nguyen
Phil James



Swansea University
Prifysgol Abertawe

Railway example track plan



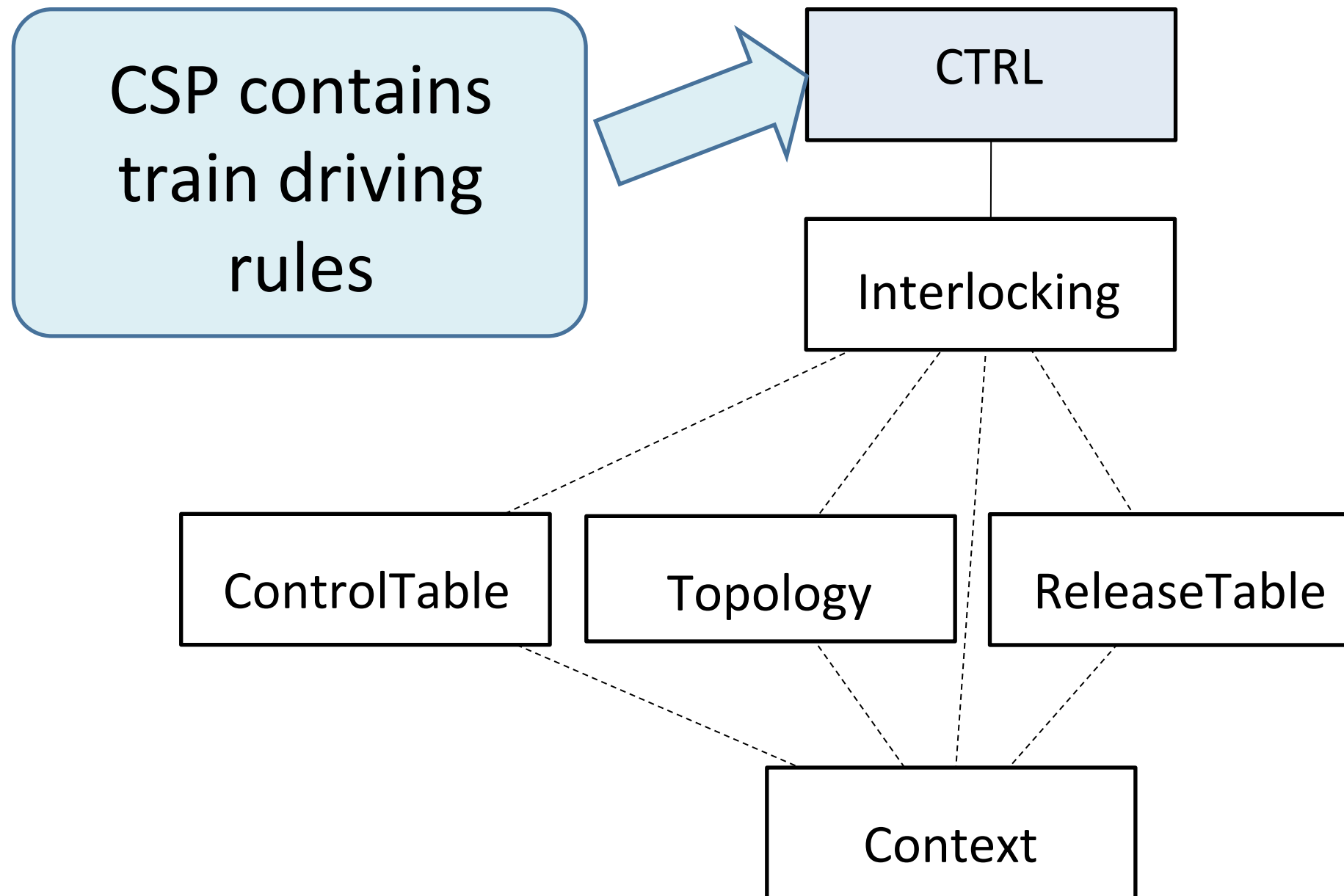
State:

train locations;
points positions;
signals; locks on routes

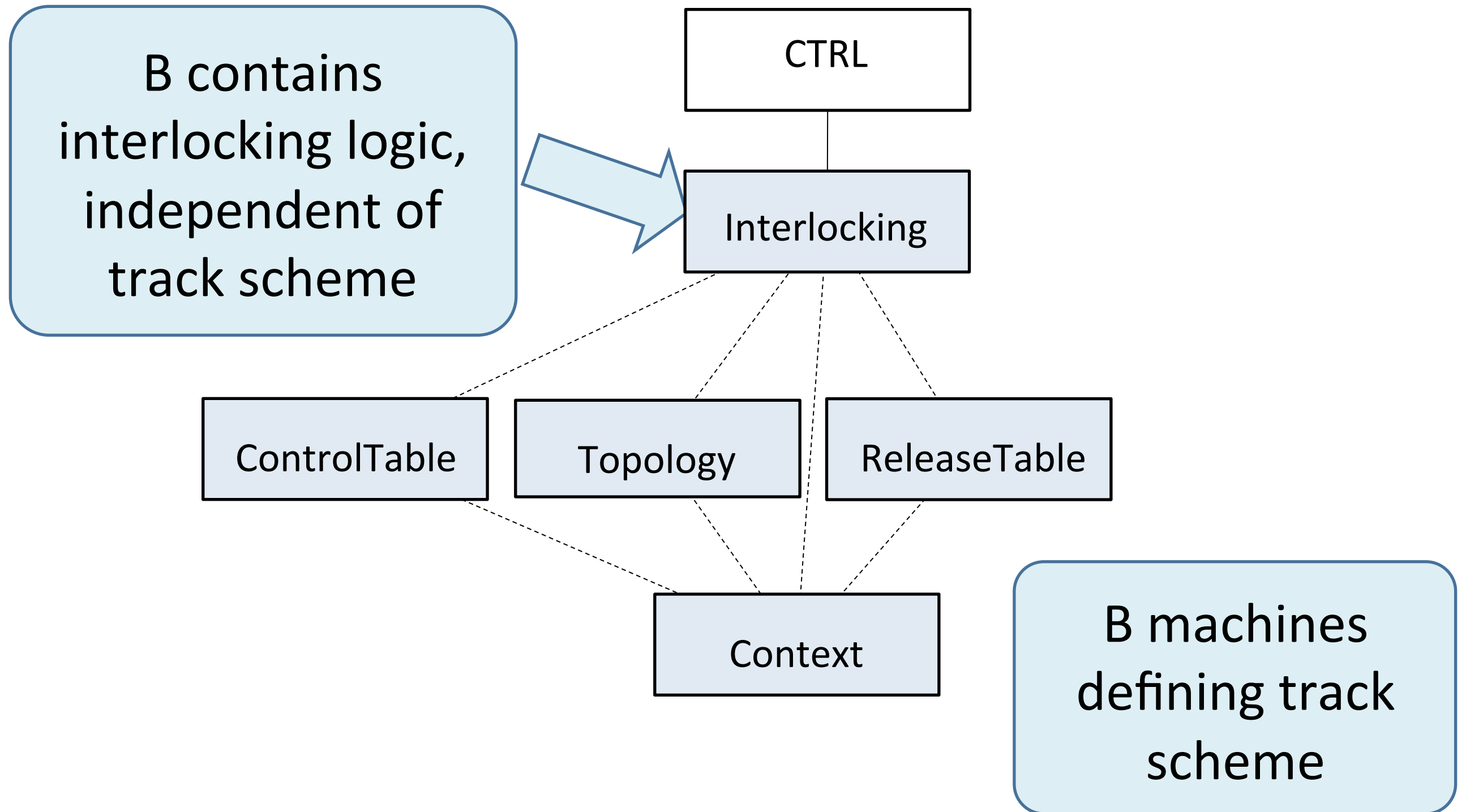
Context:

track topology;
control table;
lock table

CSP || B Architecture



CSP || B Architecture



Example of CSP model

$RW_CTRL =$

$\sqcap_{r \in ROUTE} (request!r?b \rightarrow RW_CTRL)$
 $\sqcap$
 $\sqcap_{r \in ROUTE} (release!r?b \rightarrow RW_CTRL)$

$TRAIN_OFF(t) = enter!t?newp \rightarrow TRAIN_CTRL(t, newp)$

$TRAIN_CTRL(t, pos) =$

$pos \notin EXIT \wedge pos \in SIGNALHOMES \ \& \ nextSignal!t?aspect \rightarrow$

if aspect == green
then

$move!t.pos?newp \rightarrow TRAIN_CTRL(t, newp)$

$\sqcap$

$stay!t.pos \rightarrow TRAIN_CTRL(t, pos)$

else

$stay!t.pos \rightarrow TRAIN_CTRL(t, pos)$

$\square$

$pos \notin EXIT \wedge pos \notin SIGNALHOMES \ \&$

$move!t.pos?newp \rightarrow TRAIN_CTRL(t, newp)$

$\sqcap$

$stay!t.pos \rightarrow TRAIN_CTRL(t, pos)$

$\square \dots$

$move!t.pos?newp \rightarrow \dots$

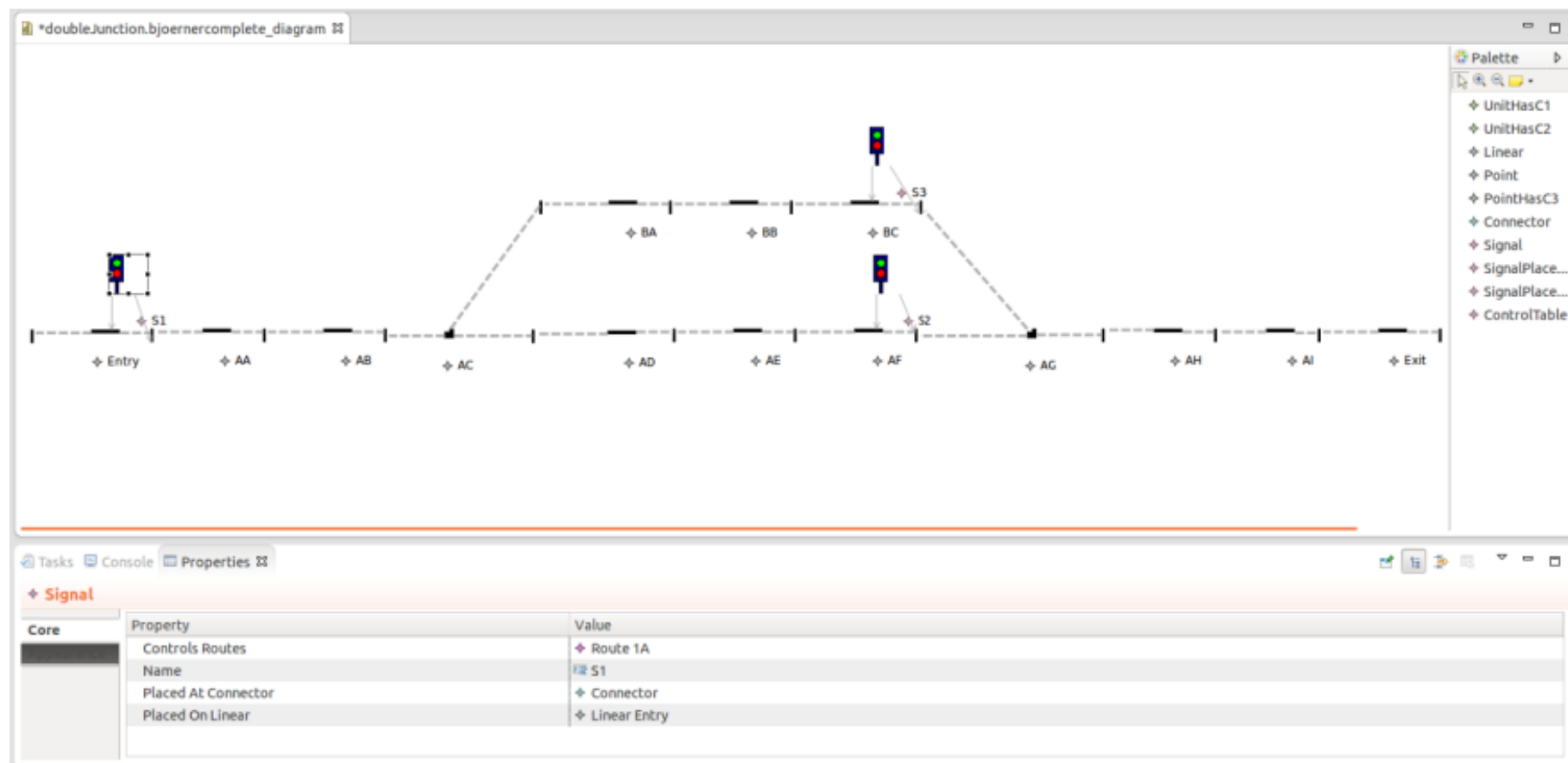
$ALL_TRAINS = |||_{t \in TRAIN} TRAIN_OFF(t)$

Example of B Interlocking operation

```
currp, newp <-- move(t) =  
PRE t : TRAIN & t : dom(pos)  
THEN  
    IF (pos(t) /: dom(nextd)) THEN  
        LET track BE  
            track = nullTrack  
        IN  
        currp := pos(t) ||  
            pos(t) := track ||  
            newp := track  
        END  
    ELSE  
        LET track BE  
            track = nextd(pos(t))  
        IN  
        currp := pos(t) ||  
        pos(t) := track ||  
        newp := track ||  
        currentLocks := currentLocks - releaseTable[{track}] ||  
        IF (pos(t) : ran(homeSignal))  
        THEN  
            signalStatus( homeSignal~(pos(t))) := red  
        END  
    END  
END  
END;  
END;
```


OnTrack: capturing track plans

- Model-Driven Engineering: translating railway domain concepts into CSP | Event-B models



ProB model checker for B (Leuschel) can check CSP || B combinations

The screenshot displays the ProB 1.3.5-beta12 model checker interface. The main window shows a B model titled "Interlocking.mch". The model includes a header with metadata, a SEES section for context, a SETS section for Bool_TYPE, and a VARIABLES section for various state variables. The bottom panel is divided into three sections: State Properties, Enabled Operations, and History. The State Properties section lists various cardinalities and constants for the model.

```
MACHINE Interlocking

/* Date: 12 03 2012
 * Notes: supporting fm2012 submission
 * Authors: Moller, Nguyen, Roggenbach, Schneider, Treharne
 * Corresponding Authors: S.Schneider@surrey.ac.uk, H.Treharne@surrey.ac.uk
 *
 * Version: Safety-Scenario2
 * Use CTRL.csp to guide Interlocking machine
 */

SEES ClosedContext,
      ClosedTopology,
      ControlTable,
      ReleaseTable

SETS
  Bool_TYPE = {true,false}

VARIABLES
  emptyTracks,
  occupiedTracks,
  pos,
  nextd,

  signalStatus,
```

Ln 16, Col 20

State Properties	Enabled Operations	History
invariant_ok MAXINT = 3 MININT = -1 card(Bool_TYPE) = 2 card(TRACKSTATUS) = 2 card(ASPECT) = 3 card(ALLTRACK) = 7 card(SIGNAL) = 3 card(TRAIN) = 2 card(POINTS) = 2 card(POINTPOSITION) = 2 card(POINTSTATUS) = 2 card(ROUTE) = 4	SETUP_CONSTANTS({red,green},{TAZ,TAB,T	

The Model panel

The screenshot displays the ProB 1.3.5-beta12 Model panel for the file `[*Interlocking.mch]`. The main text area shows the source code for the `MACHINE Interlocking`. The code includes a header with metadata (Date: 12 03 2012, Authors: Moller, Nguyen, Roggenbach, Schneider, Treharne, Corresponding Authors: S.Schneider@surrey.ac.uk, H.Treharne@surrey.ac.uk, Version: Safety-Scenario2, and a note to use `CTRL.csp` to guide the machine). The code defines the `SEES` (ClosedContext, ClosedTopology, ControlTable, ReleaseTable), `SETS` (Bool_TYPE = {true,false}), and `VARIABLES` (emptyTracks, occupiedTracks, pos, nextd, signalStatus).

Below the main text area, the status bar indicates `Ln 4, Col 1`. At the bottom, there are three panels: **State Properties**, **Enabled Operations**, and **History**. The **State Properties** panel lists various properties and their cardinalities: `invariant_ok`, `MAXINT = 3`, `MININT = -1`, `card(Bool_TYPE) = 2`, `card(TRACKSTATUS) = 2`, `card(ASPECT) = 3`, `card(ALLTRACK) = 7`, `card(SIGNAL) = 3`, `card(TRAIN) = 2`, `card(POINTS) = 2`, `card(POINTPOSITION) = 2`, `card(POINTSTATUS) = 2`, and `card(ROUTE) = 4`. The **Enabled Operations** panel shows the operation `SETUP_CONSTANTS({red,green},{TAZ,TAB,T`. The **History** panel is currently empty.

```
MACHINE Interlocking

/* Date: 12 03 2012
 * Authors: Moller, Nguyen, Roggenbach, Schneider, Treharne
 * Corresponding Authors: S.Schneider@surrey.ac.uk, H.Treharne@surrey.ac.uk
 *
 * Version: Safety-Scenario2
 * Use CTRL.csp to guide Interlocking machine
 */

SEES ClosedContext,
      ClosedTopology,
      ControlTable,
      ReleaseTable

SETS
  Bool_TYPE = {true,false}

VARIABLES
  emptyTracks,
  occupiedTracks,
  pos,
  nextd,

  signalStatus,
```

Ln 4, Col 1

State Properties	Enabled Operations	History
<code>invariant_ok</code> <code>MAXINT = 3</code> <code>MININT = -1</code> <code>card(Bool_TYPE) = 2</code> <code>card(TRACKSTATUS) = 2</code> <code>card(ASPECT) = 3</code> <code>card(ALLTRACK) = 7</code> <code>card(SIGNAL) = 3</code> <code>card(TRAIN) = 2</code> <code>card(POINTS) = 2</code> <code>card(POINTPOSITION) = 2</code> <code>card(POINTSTATUS) = 2</code> <code>card(ROUTE) = 4</code>	<code>SETUP_CONSTANTS({red,green},{TAZ,TAB,T</code>	

State and invariant panel

The screenshot shows the ProB 1.3.5-beta12: [Interlocking.mch] window. The main editor displays a BPN model with the following code:

```
reversePoints := {} || ''  
currentLocks := {}  
  
END  
  
OPERATIONS  
  
/*  
  Description: Position a train on the track for simulation  
  Inputs: a train and its position which is not occupied  
  Outputs: no outputs  
*/  
enter(t,p) =  
PRE t : TRAIN & t /: dom(pos) & p : TRACK & p /: ran(pos)  
THEN  
  pos(t) := p ||  
  occupiedTracks := occupiedTracks \ {p} ||  
  emptyTracks := emptyTracks - {p}  
END;  
  
/*  
  Description: Determine the signal status available to the train t making the request  
  Inputs: a train  
  Outputs: aspect of a signal  
*/
```

The status bar at the bottom of the editor indicates "Ln 21, Col 14".

Below the editor, there are three panels:

- State Properties**:
 - invariant_ok
 - SIGNALSTATUS = {red,green}
 - TRACK = {TAZ,TAB,TAC,TAD,TAE,TBA}
 - signal(A8) = S8
 - signal(B8) = S8
 - signal(A12) = S12
 - signal(A14) = S14
 - homeSignal(S8) = TAE
 - homeSignal(S12) = TAC
 - homeSignal(S14) = TBA
 - homePoints(P201) = TAB
 - homePoints(P202) = TAD
 - next(TAZ) = TAB
- Enabled Operations**:
 - nextSignal(albert)-->none
 - nextSignal(bertie)-->red
 - move(albert)-->TAZ,TAB
 - move(bertie)-->TBA,nullTrack
 - request(A8)-->>false
 - request(B8)-->>false
 - request(A12)-->>true
 - request(A14)-->>true
- History**:
 - enter(bertie,TBA)
 - enter(albert,TAZ)
 - INITIALISATION({},{TAZ,TAB,TAC,TAD,TAE,
 - SETUP_CONSTANTS({red,green},{TAZ,TAB,

Enabled operations

ProB 1.3.5-beta12: [Interlocking.mch]

```
reversePoints := {} || ''  
currentLocks := {}  
  
END  
  
OPERATIONS  
  
/*  
  Description: Position a train on the track for simulation  
  Inputs: a train and its position which is not occupied  
  Outputs: no outputs  
*/  
enter(t,p) =  
PRE t : TRAIN & t /: dom(pos) & p : TRACK & p /: ran(pos)  
THEN  
  pos(t) := p ||  
  occupiedTracks := occupiedTracks \ {p} ||  
  emptyTracks := emptyTracks - {p}  
END;  
  
/*  
  Description: Determine the signal status available to the train t making the request  
  Inputs: a train  
  Outputs: aspect of a signal  
*/
```

Ln 21, Col 14

State Properties	Enabled Operations	History
<pre>invariant_ok SIGNALSTATUS = {red,green} TRACK = {TAZ,TAB,TAC,TAD,TAE,TBA} signal(A8) = S8 signal(B8) = S8 signal(A12) = S12 signal(A14) = S14 homeSignal(S8) = TAE homeSignal(S12) = TAC homeSignal(S14) = TBA homePoints(P201) = TAB homePoints(P202) = TAD next(TAZ) = TAB</pre>	<pre>nextSignal(albert)-->none nextSignal(bertie)-->red move(albert)-->TAZ,TAB move(bertie)-->TBA,nullTrack request(A8)-->>false request(B8)-->>false request(A12)-->>true request(A14)-->>true</pre>	<pre>enter(bertie,TBA) enter(albert,TAZ) INITIALISATION({},{TAZ,TAB,TAC,TAD,TAE, SETUP_CONSTANTS({red,green},{TAZ,TAB,</pre>

If train moves through a red light...

invariant
violation

ProB 1.3.5-beta12: [Interlocking.mch]

```
SEES ClosedContext,  
      ClosedTopology,  
      ControlTable,  
      ReleaseTable  
  
SETS  
      Bool_TYPE = {true,false}  
  
VARIABLES  
      emptyTracks,  
      occupiedTracks,  
      pos,  
      nextd,  
  
      signalStatus,  
  
      normalPoints,  
      reversePoints,  
  
      currentLocks  
  
/* Collision-freedom:  
   We want to allow behaviour of model to allow more than one train on the same track  
   to show that combined CTRL/Interlocking system can never evolve to this behaviour.
```

Ln 21, Col 14

KO State Properties

- releaseTable(TAE) = (A12|->P202)
- releaseTable(TAE) = (A14|->P202)
- releaseTable(TBA) = (A8|->P201)
- occupiedTracks = {TAZ,nullTrack}
- emptyTracks = {TAB,TAC,TAD,TAE,TBA}
- pos(albert) = TAZ
- pos(bertie) = nullTrack
- nextd(TAZ) = TAB
- nextd(TAB) = TAC
- nextd(TAC) = TAD
- nextd(TAD) = TAE
- nextd(TAE) = TAZ
- signalStatus(S8) = red

Enabled Operations

- nextSignal(albert)-->none
- nextSignal(bertie)-->none
- move(albert)-->TAZ,TAB
- move(bertie)-->>nullTrack,nullTrack
- request(A8)-->>false
- request(B8)-->>false
- request(A12)-->>true
- request(A14)-->>true

History

- move(bertie)-->TBA,nullTrack
- enter(bertie,TBA)
- enter(albert,TAZ)
- INITIALISATION({},{TAZ,TAB,TAC,TAD,TAE,
- SETUP_CONSTANTS({red,green},{TAZ,TAB,

Analysing the invariant

invariant
violation

ProB 1.3.5-beta12: [Interlocking.mch]

```
SEES ClosedContext,  
      ClosedTopology,  
      ControlTable,  
      ReleaseTable  
  
SETS  
      Bool_TYPE = {true,false}  
  
VARIABLES  
      emptyTracks,  
      occupiedTracks,  
      pos,  
      nextd,  
  
      signalStatus,  
  
      normalPoints,  
      reversePoints,  
  
      currentLocks  
  
/* Collision-freedom:  
   We want to allow behaviour of model to allow  
   to show that combined CTRL||Interlocking sv
```

State Properties

- releaseTable(TAE) = (A12|->P202)
- releaseTable(TAE) = (A14|->P202)
- releaseTable(TBA) = (A8|->P201)
- occupiedTracks = {TAE,nullTrack}
- emptyTracks = {TAB,TAC,TAD,TAE,TBA}
- pos(albert) = TAZ
- pos(bertie) = nullTrack
- nextd(TAZ) = TAB
- nextd(TAB) = TAC
- nextd(TAC) = TAD
- nextd(TAD) = TAE
- nextd(TAE) = TAZ
- signalStatus(S8) = red

Analysing INVARIANT

(Non-Typing) Conjuncts of INVARIANT:

```
** CONJUNCTS OVER IDENTIFIERS: ['TRACK',emptyTracks,nextd,occupiedTracks,pos] *  
*  
emptyTracks <: TRACK  
== TRUE  
  
occupiedTracks <: TRACK  
== false  
  
emptyTracks /\ occupiedTracks = {}  
== TRUE  
  
pos : TRAIN >+> TRACK  
== false  
  
nextd : TRACK >+> TRACK  
== TRUE  
  
** CONJUNCTS OVER IDENTIFIERS: ['SIGNALSTATUS',signalStatus] **
```

Save... Done

History panel: operations performed so far

The screenshot displays the ProB 1.3.5-beta12 interface for the file [Interlocking.mch]. The main editor window contains the following code:

```
SEES ClosedContext,  
      ClosedTopology,  
      ControlTable,  
      ReleaseTable  
  
SETS  
  Bool_TYPE = {true,false}  
  
VARIABLES  
  emptyTracks,  
  occupiedTracks,  
  pos,  
  nextd,  
  
  signalStatus,  
  
  normalPoints,  
  reversePoints,  
  
  currentLocks  
  
/* Collision-freedom:  
We want to allow behaviour of model to allow more than one train on the same track  
to show that combined CTRL-Interlocking system can never evolve to this behaviour.
```

The status bar at the bottom indicates "Ln 21, Col 14". Below the editor, there are three panels:

- State Properties:** Contains a list of current state values, including track releases, occupied/empty track sets, positions, next destinations, and signal status.
- Enabled Operations:** Lists operations that are currently enabled, such as `nextSignal`, `move`, and `request` for various trains and signals.
- History:** This panel is highlighted with a thick black border and shows a sequence of operations performed so far, including `move(bertie)`, `enter(bertie)`, `enter(albert)`, and initialization/setup constants.

Checking safety conditions

The screenshot displays a software interface for model checking. The main window has a menu bar with options: Apple icon, prob, File, Edit, Animate, **Verify**, Analyse, Plugins, Preferences, Debug, Files, Help. The 'Verify' menu is open, showing options: Model Check... (⌘M), Check LTL Formula... (⌘L), Check LTL Assertions, Check CSP-M Assertions, Check CTL Formula..., Trace Refinement Check..., Save State for Later Refinement Check, Constraint Based Checking, External Tools, and Trace Checking.

A 'Model Check:' dialog box is open in the foreground. It contains a 'Model Check' button, a 'Cancel' button, and a 'Search Strategy' dropdown menu. Below these are several checkboxes: ☒ Find Deadlocks, ☒ Find Invariant Violations, ☐ Find B ASSERTION Violations, ☐ Find event on goal CSP channel, ☐ Stop when all Operations covered, and ☒ Search for new Errors. At the bottom, it says 'Searching... 1400'.

Below the dialog box, there are two panels: 'State Properties' and 'Enabled Operations'. The 'State Properties' panel shows a list of properties: invariant_ok, CSP: RW_CTRL@_1[{|enter,request|}]{|enter,|}, SIGNALSTATUS = {red,green}, TRACK = {TAZ,TAB,TAC,TAD,TAE,TBA}, signal(A8) = S8, signal(B8) = S8, signal(A12) = S12, signal(A14) = S14, homeSignal(S8) = TAE, homeSignal(S12) = TAC, homeSignal(S14) = TBA, homePoints(P201) = TAB, and homePoints(P202) = TAD. The 'Enabled Operations' panel shows a list of operations: request(A12)-->true, request(A14)-->true, request(A8)-->false, request(B8)-->false, nextSignal(albert)-->none, and nextSignal(bertie)-->red.

To the right of these panels, a message box is displayed with the text 'No error state found, all NEW nodes visited.' and an 'OK' button. A large green checkmark is positioned at the bottom right of the image.

Scenario - Faulty Tracks in “ClearTable”

ProB 1.3.5-beta14: [DJ]Interlocking.mch



Formula FALSE.
Counterexample found for AG(not e[collision]).

OK

```
/*exit(t,p) =
PRE t : TRAIN & p : EXIT & pos
THEN
  pos(t) := p
END;
*/

/* Collision operator to detect collision
to check for collision, use CTL formula AG(not e[collision])
*/
collision =
SELECT #(t1,t2).(t1 : TRAIN & t2 : TRAIN &
  t1:dom(pos) & t2:dom(pos) & t1 /= t2 &
  pos(t1) /= EXIT & pos(t2) /= EXIT &
  pos(t1) = pos(t2))
THEN skip
END;

/* Derailment operator to detect derailment
to check for collision, use CTL formula AG(not e[derailment])
*/
derailment =
SELECT #(t1).(t1 : TRAIN & t1:dom(pos) & pos(t1) = nullTrack)
THEN skip
END;
```

Ln 91, Col 3

State Properties	Enabled Operations	History
invariant_ok MAXINT = 3 MININT = -1 card(ANSWERS) = 2 card(TRACKSTATUS) = 2 card(ASPECT) = 2 card(ALLTRACK) = 21 card(SIGNAL) = 9 card(TRAIN) = 2 card(POINTS) = 4 card(POINTPOSITION) = 2 card(POINTSTATUS) = 2 card(ROUTE) = 10	start_cspm_MAIN	

352,367 states checked

request(A1)-->yes
enter(albert,Entry1)
nextSignal(albert)-->green
move(albert)-->Entry1,AA
request(A3)-->yes
nextSignal(albert)-->green
move(albert)-->AA,AB
move(albert)-->AB,AC
request(A1)-->yes
enter(bertie,Entry1)
request(A3)-->yes
nextSignal(bertie)-->green
move(bertie)-->Entry1,AA
nextSignal(bertie)-->green
move(bertie)-->AA,AB
move(bertie)-->AB,AC
request(A16)-->yes
request(A5)-->yes
request(A2)-->yes
move(albert)-->AC,AD
move(bertie)-->AC,AD
nextSignal(albert)-->green

Where we're at

- Engineers understand/agree with the models (train driving rules; track interlocking rules)
- The combined semantics also provides a foundation for compositional results on complex track plans:
 - Finitisation: limiting the number of trains to include in a safety analysis
 - Coverings: breaking down a complex track plan into manageable fragments that can be independently checked

Structure of Talk

- B and Event-B for state
- Combination with CSP
- Example: modelling for railway safety
- **Incorporating time**

Introducing time

railway designers are also interested in network capacity

Challenge: Introducing time

- We want to introduce time to CSP | | Event-B
- Natural and obvious approach: use Timed CSP as the control language and manage all the time in there
 - leave the B models as untimed, embedded in the timed CSP semantic framework
- ... but not quite so simple...

Timed CSP

- Timed language:

$$\begin{aligned} P ::= & \text{STOP} \mid \text{SKIP} \mid \text{WAIT } t \mid P; Q \mid \\ & a \rightarrow P \mid c?x \rightarrow P \mid c!v \rightarrow P \mid \\ & P \sqcap Q \mid P \sqbox Q \mid P \overset{t}{\triangleright} Q \mid \\ & P \parallel Q \mid P \parallel\!\!\parallel Q \mid P \setminus A \\ & X \mid \mu X . P \end{aligned}$$

- (recursions time-guarded)

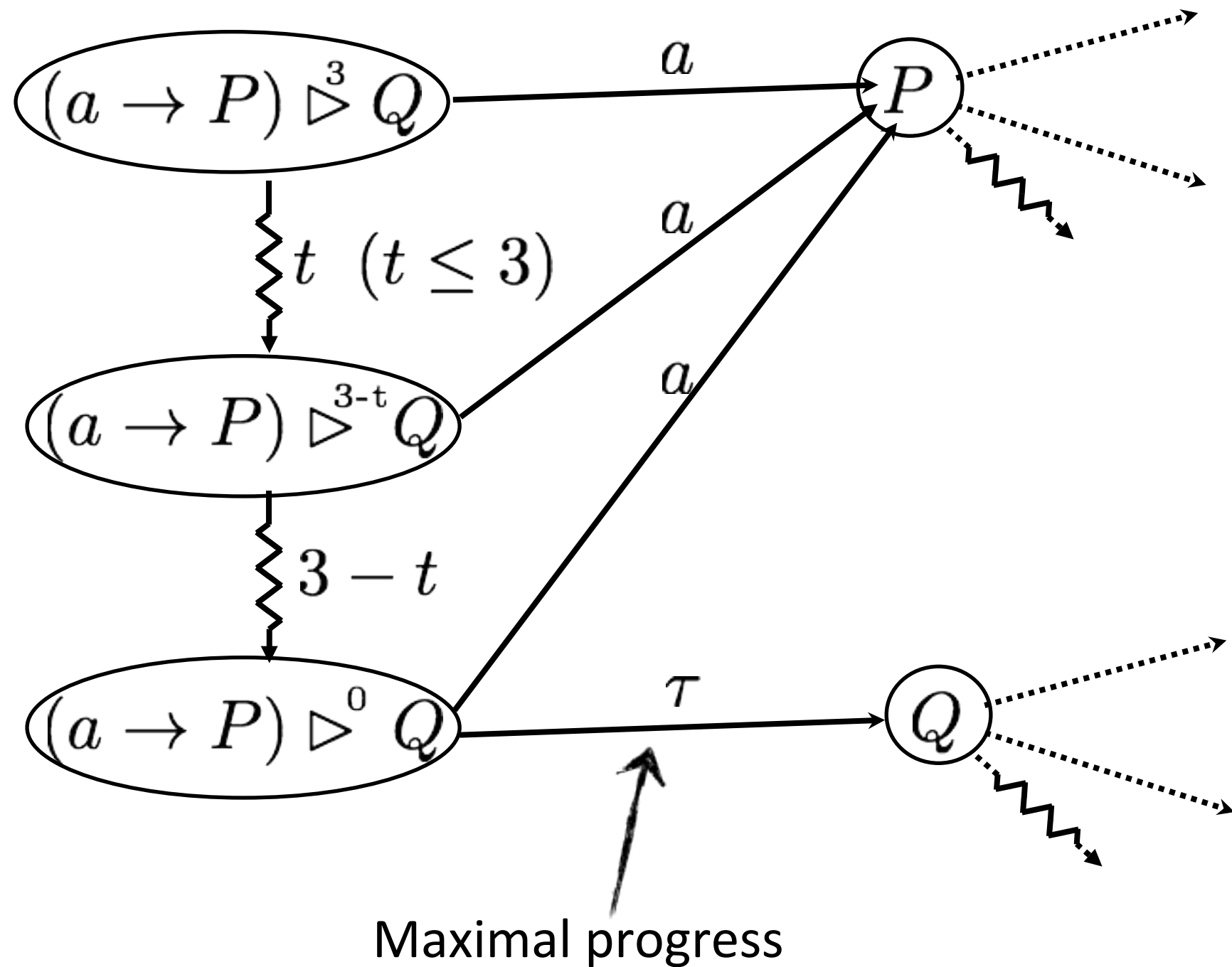
Timed CSP transitions

$$(a \rightarrow P) \xrightarrow{a} P$$

$$(a \rightarrow P) \xrightarrow{2} (a \rightarrow P)$$

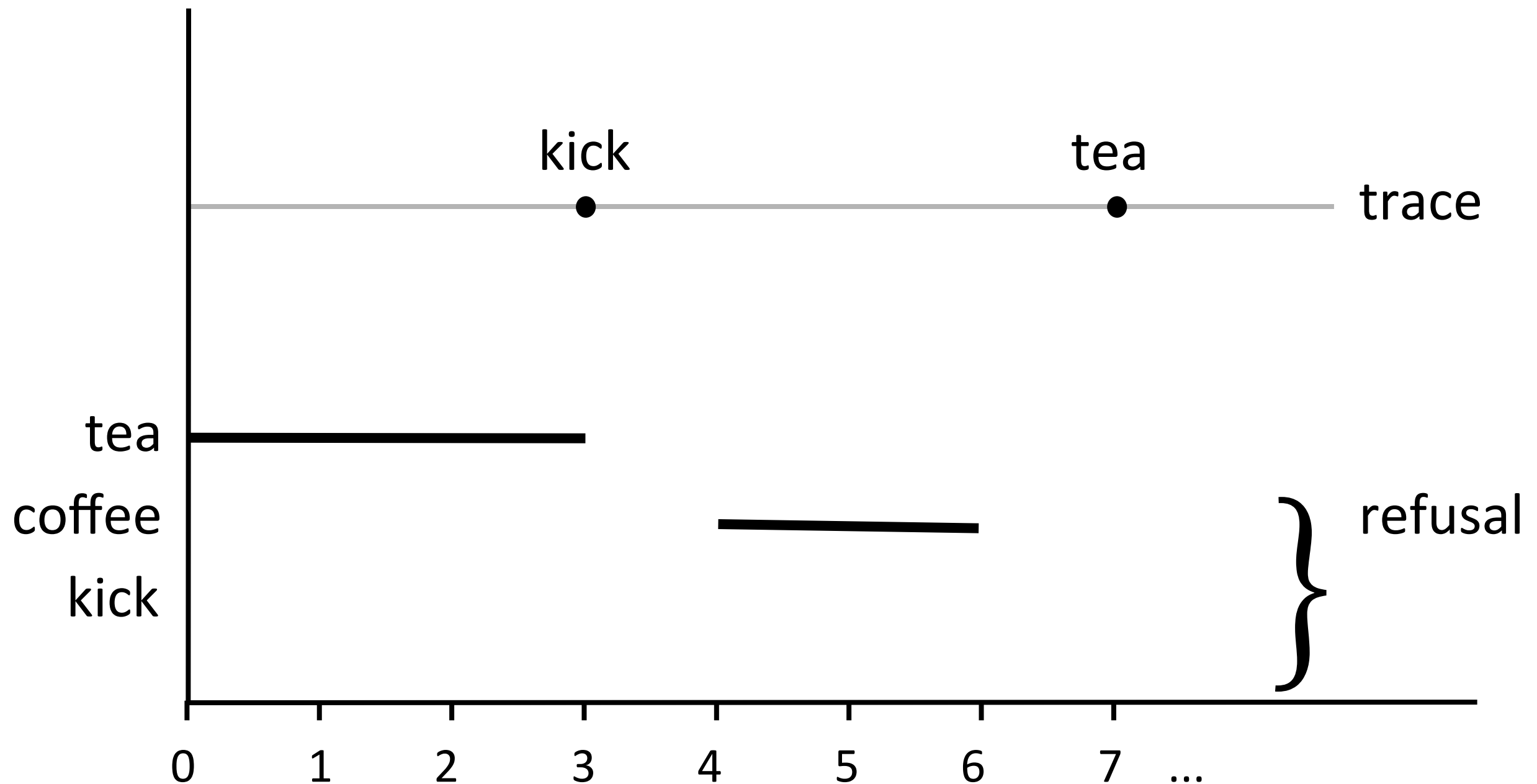
$$(P \triangleright^7 Q) \xrightarrow{2} (P' \triangleright^5 Q)$$

Example fragment of a timed transition system



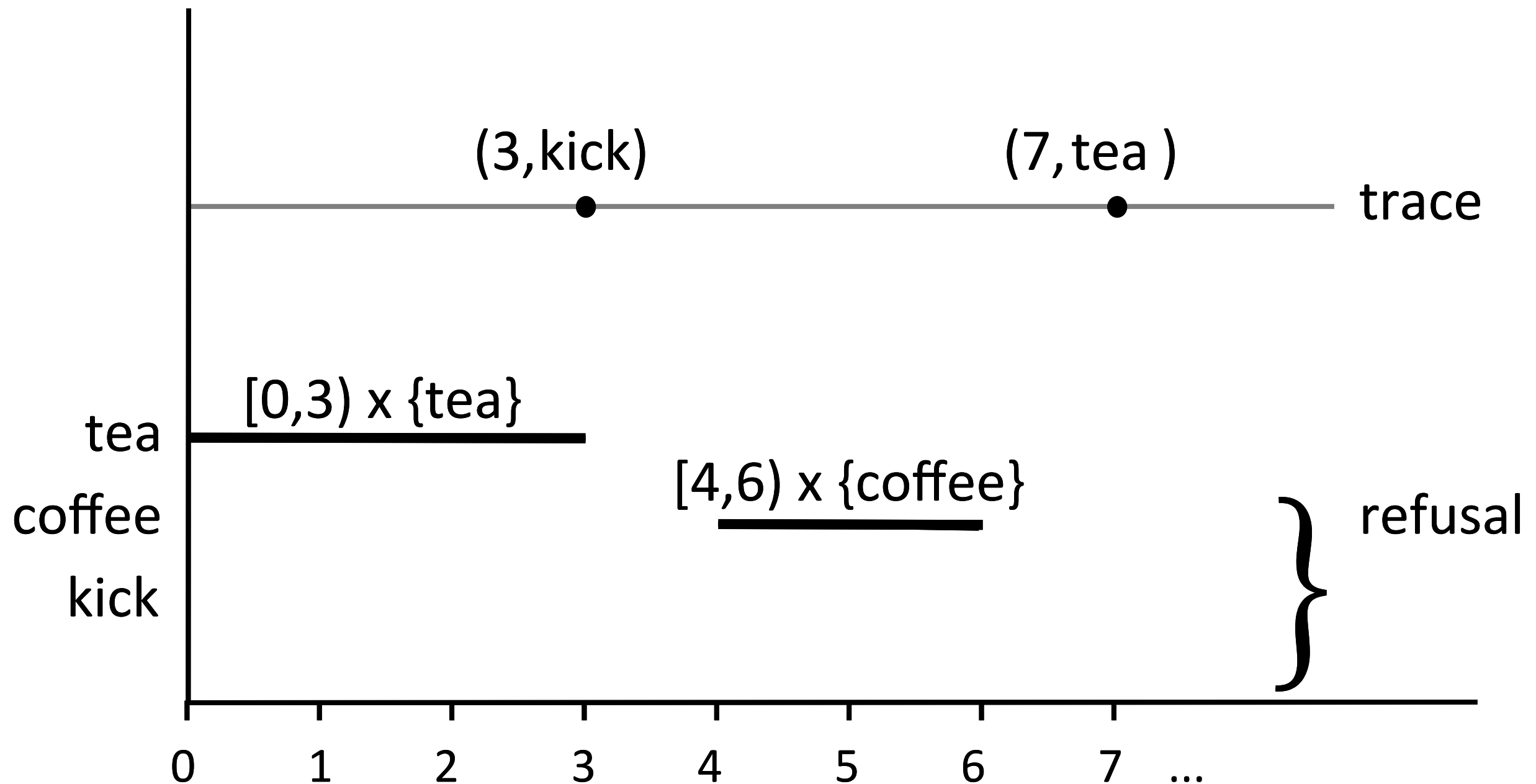
Observations for denotational semantics: Timed Failures

$(coffee \rightarrow STOP) \sqcap (kick \rightarrow tea \rightarrow STOP)$



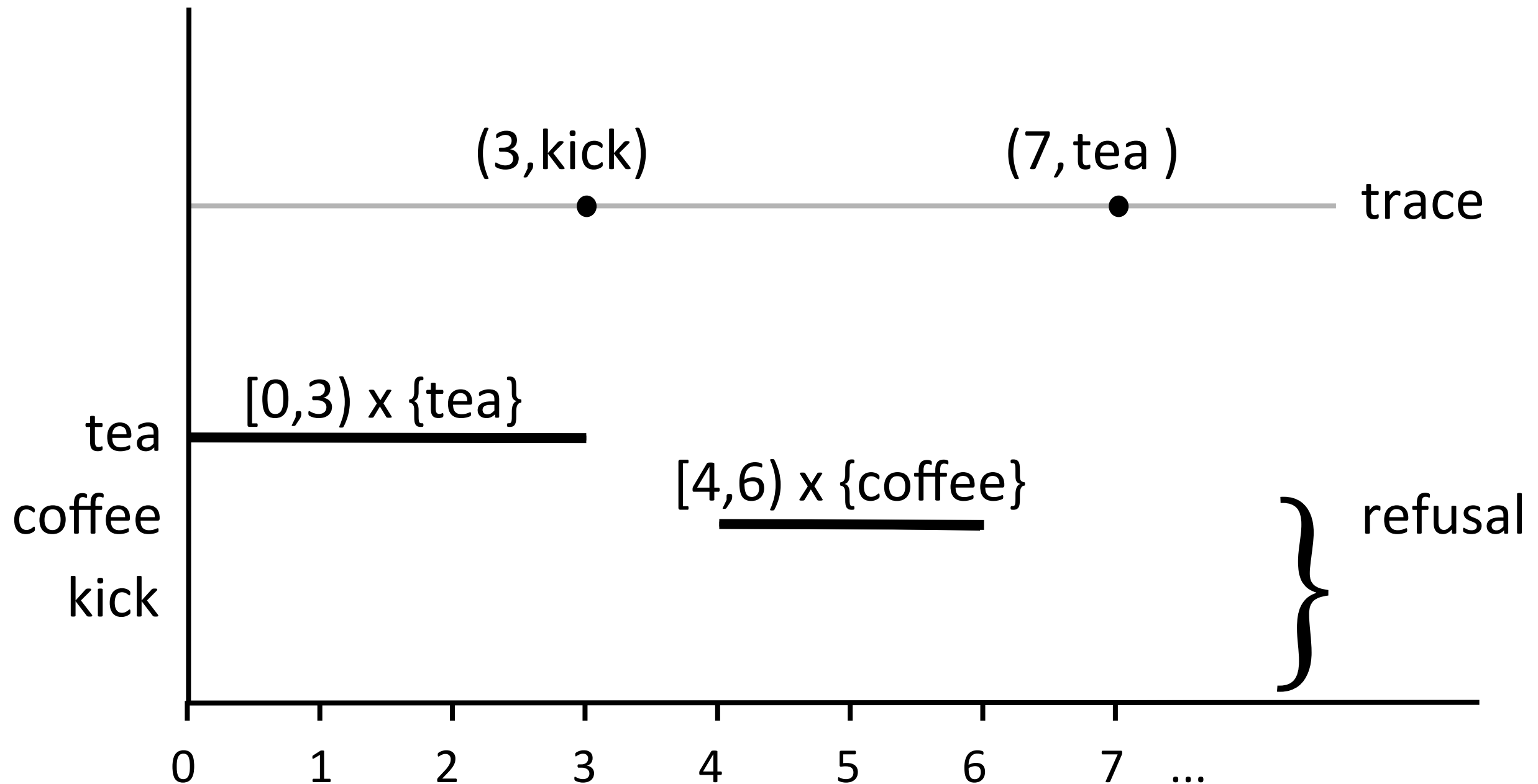
Observations for denotational semantics: Timed Failures

$(coffee \rightarrow STOP) \sqsubseteq (kick \rightarrow tea \rightarrow STOP)$



Observations for denotational semantics: Timed Failures

$\langle [0, 3) \times \{tea\}, (3, kick), [4, 6) \times \{coffee\}, (7, tea), \emptyset \rangle$



Timed CSP and B/Event-B: issues

- “leave the B models as untimed, embedded in the timed CSP semantic framework”
- 1. wp semantics doesn't give timed CSP semantics (even for “untimed” machines)
 - no refusals during the trace
- 2. machines react and change state instantly
 - they are not “time guarded”
- 3. need to include divergence (abort) in machines (for B, but not for Event-B)

1. Timed refusal testing

$$\begin{aligned} P1 &= (coffee \rightarrow STOP) \sqcap (kick \rightarrow tea \rightarrow STOP) \\ &\quad \sqcap (tea \rightarrow STOP) \sqcap (kick \rightarrow coffee \rightarrow STOP) \\ P2 &= (coffee \rightarrow STOP) \sqcap (kick \rightarrow coffee \rightarrow STOP) \\ &\quad \sqcap (tea \rightarrow STOP) \sqcap (kick \rightarrow tea \rightarrow STOP) \end{aligned}$$

1. Timed refusal testing

$$\begin{aligned} P1 &= (coffee \rightarrow STOP) \sqcap (kick \rightarrow tea \rightarrow STOP) \\ &\quad \sqcap (tea \rightarrow STOP) \sqcap (kick \rightarrow coffee \rightarrow STOP) \end{aligned}$$

$$\begin{aligned} P2 &= (coffee \rightarrow STOP) \sqcap (kick \rightarrow coffee \rightarrow STOP) \\ &\quad \sqcap (tea \rightarrow STOP) \sqcap (kick \rightarrow tea \rightarrow STOP) \end{aligned}$$

$$\langle [0, 3) \times \{tea\}, (3, kick), [4, 6) \times \{coffee\}, (7, tea), \emptyset \rangle$$

Timed CSP semantics?

Now need to distinguish these two:

Machine VM1 =

Variables n

Invariants $n \in \{0,1,2\}$

Initialisation $n:=0 \sqcap n:=1$

Operations

tea =

when $n=0$ then $n:=2$ end

coffee =

when $n = 1$ then $n:=2$ end

kick =

when $n<2$ then $n:=1-n$ end

Machine VM2 =

Variables n

Invariants $n \in \{0,1,2\}$

Initialisation $n:=0 \sqcap n:=1$

Operations

tea =

when $n=0$ then $n:=2$ end

coffee =

when $n = 1$ then $n:=2$ end

kick =

when $n<2$ then skip end

$$P = (tea \rightarrow STOP) \triangleright^3 (kick \rightarrow tea \rightarrow STOP)$$

$P \parallel VM1$ can't refuse tea for ever, $P \parallel VM2$ can

Untimed refusal traces and wp

$$\langle X_0, a_1, X_1, a_2, X_2, \dots \rangle \in \text{refusal traces}(M) \iff$$

$$\begin{array}{ll} \exists P_0, P_1, \dots & \begin{array}{l} 1. \quad \overline{wp}(\text{init}, P_0) \\ P_0 \Rightarrow \overline{wp}(a_1, P_1) \\ \vdots \\ P_i \Rightarrow \overline{wp}(a_{i+1}, P_{i+1}) \\ \\ 2. \quad \text{for each } i: P_i \Rightarrow \bigwedge_{a \in X_i} \neg g_a \end{array} \end{array}$$

Example: $\langle \{tea\}, kick, \emptyset, tea, \emptyset \rangle$

$\overline{wp}(init, P_0)$

$P_0 : n = 1$

$\neg g_{tea} : n \neq 0$

$P_0 \Rightarrow \overline{wp}(kick, P_1)$

$P_1 : n = 0$

$\bigwedge_{a \in X_1} \neg g_a : true$

$P_1 \Rightarrow \overline{wp}(tea, P_2)$

$P_2 : true$

$\bigwedge_{a \in X_2} \neg g_a : true$

Defining timed refusals via wp: relating timed and untimed observations

$$\langle \mathcal{N}_0, (t_1, a_1), \mathcal{N}_1, (t_2, a_2), \mathcal{N}_2 \dots \rangle \in \text{timed failures}(M) \iff$$

$$\langle \sigma(\mathcal{N}_0), a_1, \sigma(\mathcal{N}_1), a_2, \sigma(\mathcal{N}_2) \dots \rangle \in \text{refusal traces}(M)$$

Just the events;
times removed

NB: True for “untimed” B machines, because
state changes are instant and states are stable
between transitions.

Not true for arbitrary timed CSP processes

$$\langle [0, 3) \times \{tea\}, (3, kick), [4, 6) \times \{coffee\}, (7, tea), \emptyset \rangle$$

$$\langle \{tea\}, kick, \{coffee\}, tea, \emptyset \rangle$$

Timed CSP: Fixed Point theory

- Reed&Roscoe 1988 (+ see Schneider 1999):
 - Semantic model is a complete metric space
 - » $d(P,Q)$ is 2^{-t} , where P and Q first differ at time t
 - Recursions are time guarded
 - Recursions are contraction mappings, giving unique fixed points
 - No divergences (processes don't go infinitely fast). Infinite internal transitions (e.g. from internal loops) take infinite time.

$$\mu X . (send!v \rightarrow (rec?ack \rightarrow STOP) \triangleright^3 X)$$

$$\mu X . (rec1?x \rightarrow Q) \triangleright^3 ((rec2?x \rightarrow R) \triangleright^3 X)$$

Timed CSP semantics?

Instant response

Machine VM1 =

Variables n

Invariants $n \in \{0,1,2\}$

Initialisation $n:=0 \sqcap n:=1$

Operations

tea =

 when n=0 then n:=2 end

coffee =

 when n = 1 then n:=2 end

kick =

 when n<2 then n:=1-n end

$M0 = tea \rightarrow STOP \sqcap kick \rightarrow M1$

$M1 = coffee \rightarrow STOP \sqcap kick \rightarrow M0$

$VM1 = M0 \sqcap M1$

$VM1 \setminus \{kick\}$

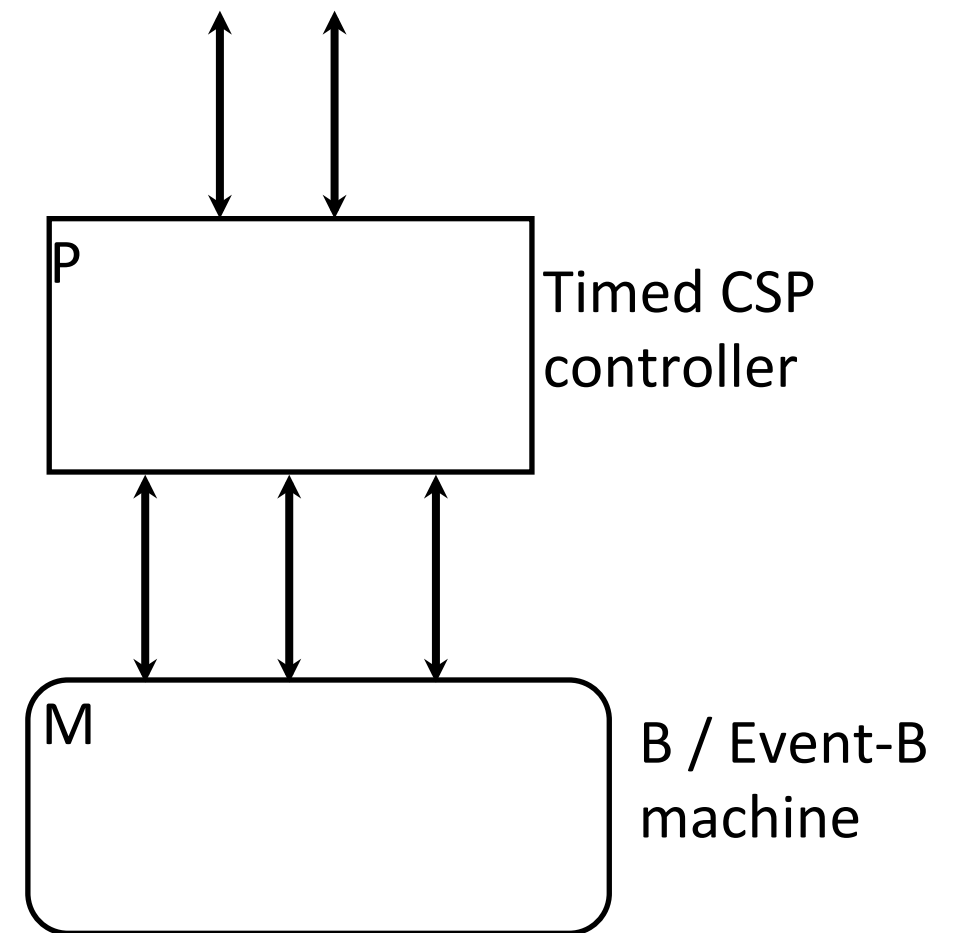
A more recent semantics for timed CSP...

...does what we need

- Ouaknine&Worrell, “Timed CSP = Closed Timed ϵ -Automata”, Nordic Journal of Computing 10 (2), 2003.
- Gives a more elaborate semantics for Timed CSP, including the following features in the language:
 - Arbitrarily fast processes (non-time-guarded recursions permitted)
 - Divergences (infinite tau loops in zero time), zeno processes?
 - Timestops and urgent events
- Semantics
 - cpo rather than cms
 - operational semantics (claimed to match denotational)

Combining Timed CSP and B / Event-B

- P and M both have a Ouaknine/Worrell timed failures semantics.
- So $P \parallel M$ also has a semantics.
- The B machine can go arbitrarily fast...
- ...but in practice is driven by the Timed CSP controller, which we can make sure (and verify) is well-behaved: no divergences, no timestops, time-guarded loops.



Further Challenges and Questions

- What are the best ways to manage (large) state with concurrency?
- Time? Probability? Mobility ??
- Hierarchical use
- Pragmatics: more integrated tool support

Exploring the detailed trace

enter(bertie,Entry2)

request(R118A)-->yes

nextSignal(bertie)-->green

move(bertie)-->Entry2,BM

request(R116A)-->yes

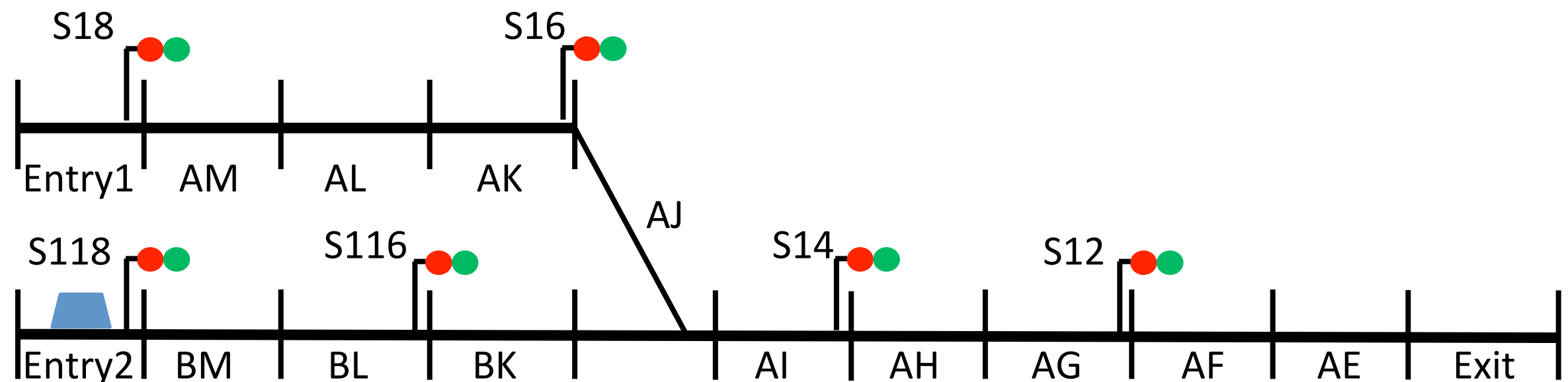
move(bertie)-->BM,BL

nextSignal(bertie)-->green

move(bertie)-->BL,BK

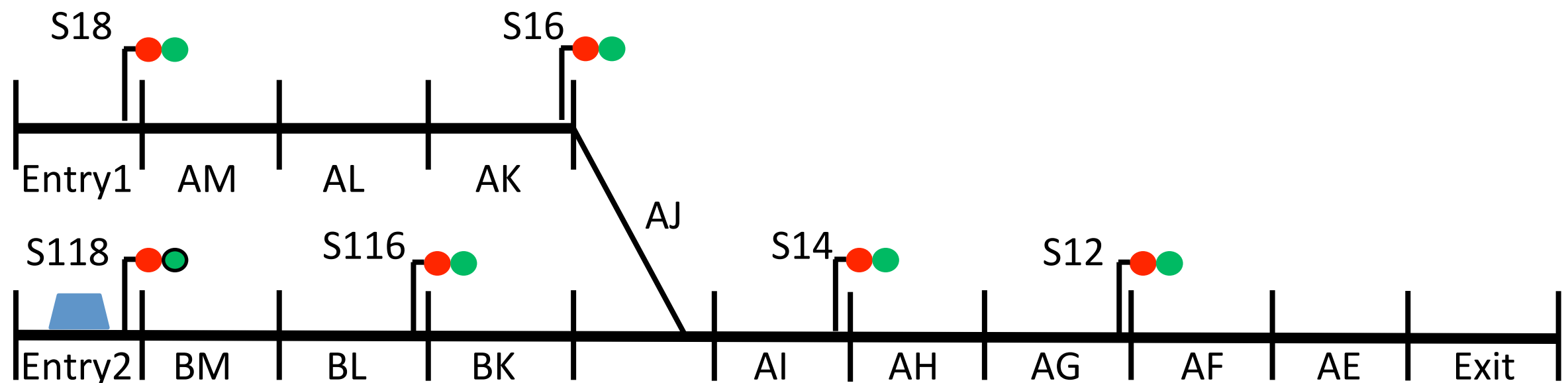
move(bertie)-->BK,AJ

move(bertie)-->AJ,AI



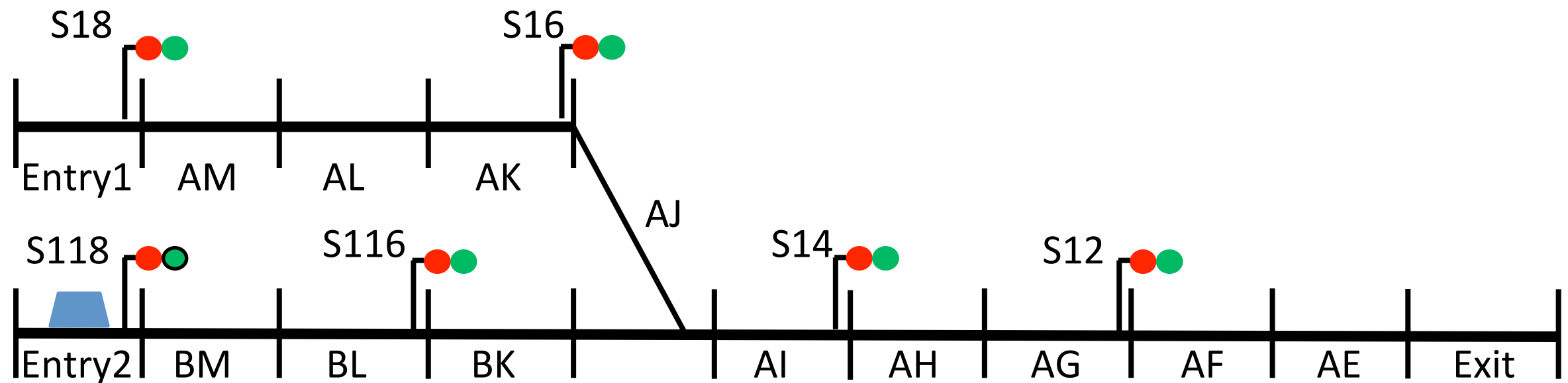
Exploring the detailed trace

```
enter(bertie,Entry2)
request(R118A)-->yes
nextSignal(bertie)-->green
move(bertie)-->Entry2,BM
request(R116A)-->yes
move(bertie)-->BM,BL
nextSignal(bertie)-->green
move(bertie)-->BL,BK
move(bertie)-->BK,AJ
move(bertie)-->AJ,AI
```



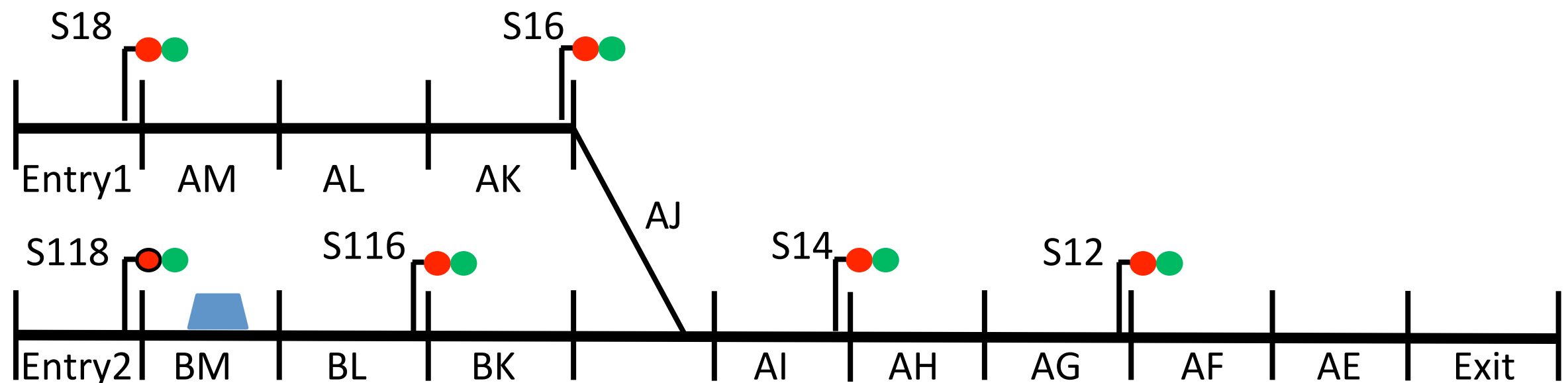
Exploring the detailed trace

```
enter(bertie,Entry2)
request(R118A)-->yes
nextSignal(bertie)-->green
move(bertie)-->Entry2,BM
request(R116A)-->yes
move(bertie)-->BM,BL
nextSignal(bertie)-->green
move(bertie)-->BL,BK
move(bertie)-->BK,AJ
move(bertie)-->AJ,AI
```



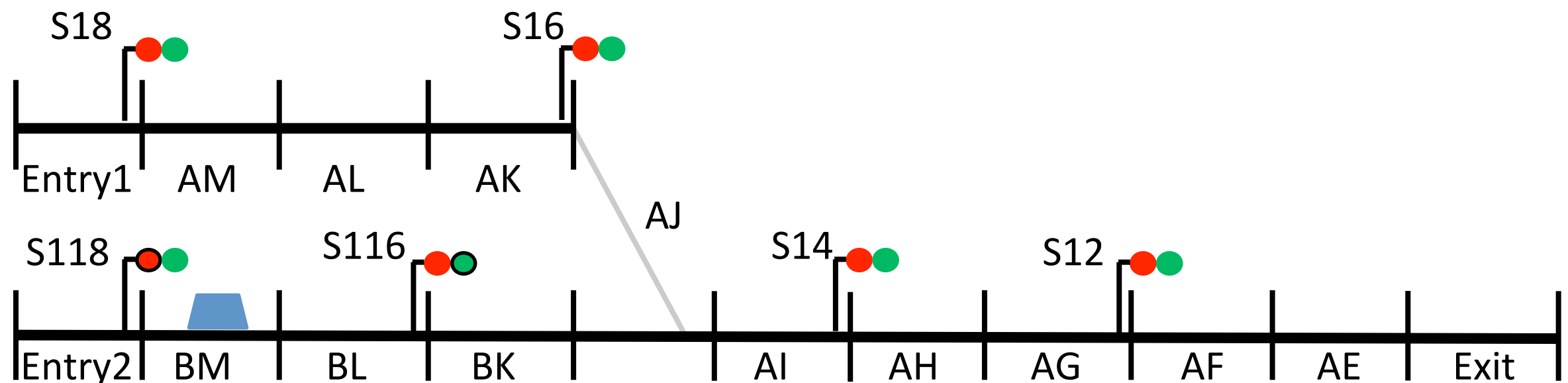
Exploring the detailed trace

```
enter(bertie,Entry2)
request(R118A)-->yes
nextSignal(bertie)-->green
move(bertie)-->Entry2,BM
request(R116A)-->yes
move(bertie)-->BM,BL
nextSignal(bertie)-->green
move(bertie)-->BL,BK
move(bertie)-->BK,AJ
move(bertie)-->AJ,AI
```



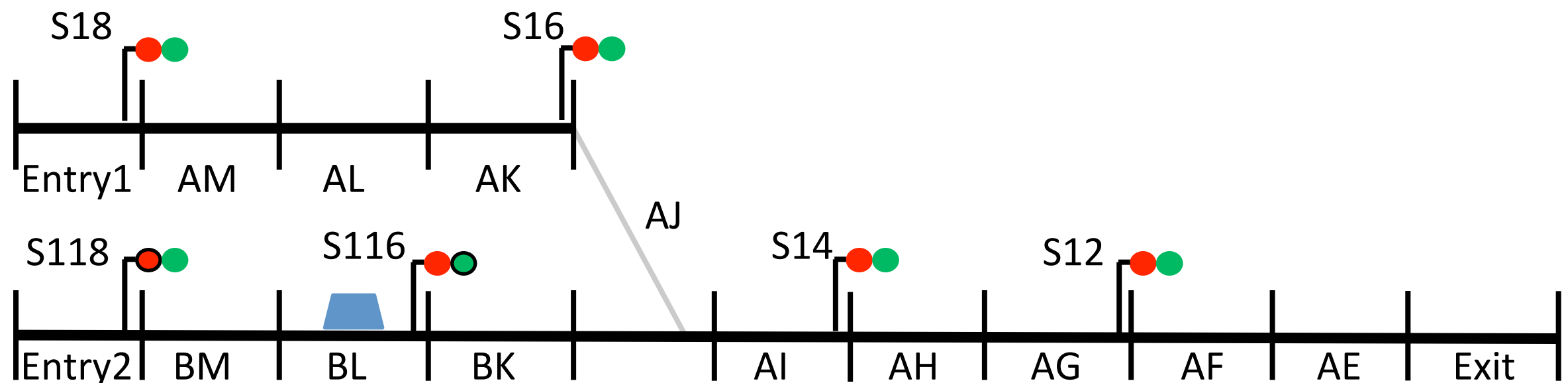
Exploring the detailed trace

```
enter(bertie,Entry2)
request(R118A)-->yes
nextSignal(bertie)-->green
move(bertie)-->Entry2,BM
request(R116A)-->yes
move(bertie)-->BM,BL
nextSignal(bertie)-->green
move(bertie)-->BL,BK
move(bertie)-->BK,AJ
move(bertie)-->AJ,AI
```



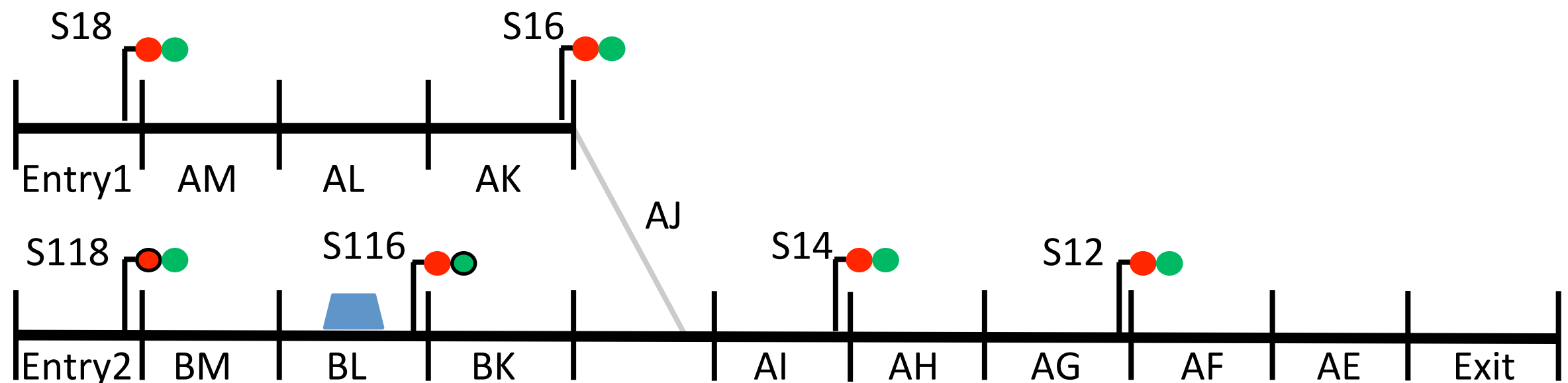
Exploring the detailed trace

```
enter(bertie,Entry2)
request(R118A)-->yes
nextSignal(bertie)-->green
move(bertie)-->Entry2,BM
request(R116A)-->yes
move(bertie)-->BM,BL
nextSignal(bertie)-->green
move(bertie)-->BL,BK
move(bertie)-->BK,AJ
move(bertie)-->AJ,AI
```



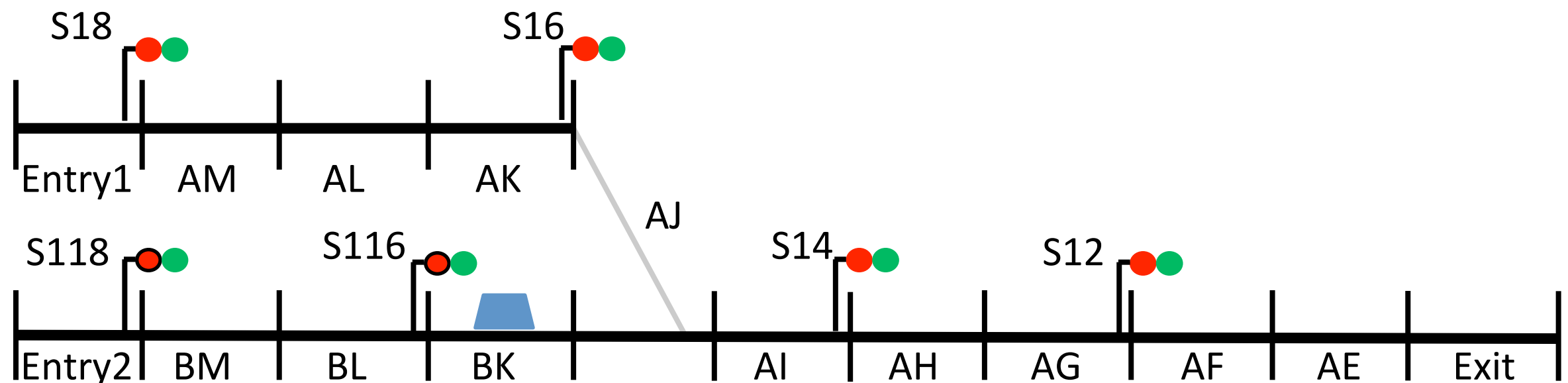
Exploring the detailed trace

```
enter(bertie,Entry2)
request(R118A)-->yes
nextSignal(bertie)-->green
move(bertie)-->Entry2,BM
request(R116A)-->yes
move(bertie)-->BM,BL
nextSignal(bertie)-->green
move(bertie)-->BL,BK
move(bertie)-->BK,AJ
move(bertie)-->AJ,AI
```



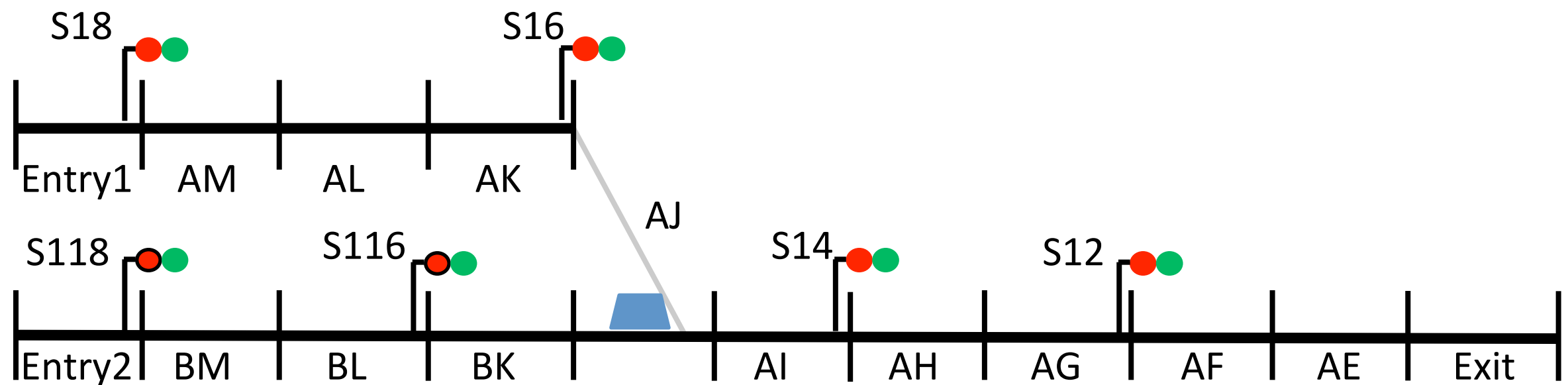
Exploring the detailed trace

enter(bertie,Entry2)
request(R118A)-->yes
nextSignal(bertie)-->green
move(bertie)-->Entry2,BM
request(R116A)-->yes
move(bertie)-->BM,BL
nextSignal(bertie)-->green
move(bertie)-->BL,BK
move(bertie)-->BK,AJ
move(bertie)-->AJ,AI



Exploring the detailed trace

enter(bertie,Entry2)
request(R118A)-->yes
nextSignal(bertie)-->green
move(bertie)-->Entry2,BM
request(R116A)-->yes
move(bertie)-->BM,BL
nextSignal(bertie)-->green
move(bertie)-->BL,BK
move(bertie)-->BK,AJ
move(bertie)-->AJ,AI



Exploring the detailed trace

enter(bertie,Entry2)
request(R118A)-->yes
nextSignal(bertie)-->green
move(bertie)-->Entry2,BM
request(R116A)-->yes
move(bertie)-->BM,BL
nextSignal(bertie)-->green
move(bertie)-->BL,BK
move(bertie)-->BK,AJ
move(bertie)-->AJ,AI

