Communicating Sequential Processes

Hoare’s CSP (Communicating Sequential Processes) an event based notation primarily aimed at describing the sequencing of behaviour within a process and the synchronisation of behaviour (or communication) between processes. Events represent a co-operative synchronisation between process and environment. Both process and environment may control the behaviour of the other by enabling or refusing certain events or sequences of events.

Specifying a Process

A process is determined (specified) by what it can do; i.e. a process is defined by its behaviour. The perceived behaviour of a process will depend upon the observer. We shall be mainly concerned with specifying the interaction between a system and its environment (i.e. external (visible) behaviour).
Events

A process engages in events; each event is an atomic action. e.g. the events for a vending machine are

\textit{coin}—insert a coin

\textit{choc}—extract a chocolate

The set of events that a process can possibly engage in is the alphabet of the process e.g. the alphabet of the vending machine is

\{\textit{coin}, \textit{choc}\}
Traces

A trace is a finite sequence of events.

A (deterministic) process is specified by the set of traces denoting its possible behaviour. e.g. the traces of the vending machine:

\[
\langle \ \rangle \\
\langle \text{coin} \rangle \\
\langle \text{coin, choc} \rangle \\
\langle \text{coin, choc, coin} \rangle \\
\ldots
\]

Any execution of the process will be one of these sequences. If \( s \sqsubseteq t \) is a trace of a process, then so also is \( s \); i.e. the set of traces is prefix closed.
Basic Process Notation

- lower case identifiers denote events;
  \( x, y, z \) are variables denoting events

- upper case identifiers denote processes;
  \( X, Y \) are variables denoting processes

- \( A, B, C \) denote sets of events

- if \( P \) is a process,
  \( \alpha P \) denotes the alphabet of \( P \)

- if \( P \) is a process,
  \( \text{traces}(P) \) denotes the set of traces of \( P \)
\section*{Trace Notation}

- if $A$ is a set of events, 
  seq $A$ denotes the set of all finite sequences of events from $A$.

- if $s, t : \text{seq } A$, 
  $s \bowtie t$ is the \textit{concatenation} of $s$ with $t$
  e.g. 
  \[ \langle b, a, b \rangle \bowtie \langle b, c, a \rangle = \langle b, a, b, b, c, a \rangle \]

- \[
\begin{array}{c}
\leq: \text{seq } A \leftrightarrow \text{seq } A \\
s \leq t \iff \\
\exists u : \text{seq } A \bullet s \bowtie u = t
\end{array}
\]

- $s^n = s \bowtie s \bowtie s \bowtie \cdots \bowtie s$
  i.e. $s$ concatenated with itself $n$ times
Examples

(1) $STOP_A$ is the process with alphabet $A$ that can do nothing.
\[ traces(STOP_A) = \{\langle \rangle\} \]

(2) $CLOCK$ is the process with $\alpha CLOCK = \{tick\}$ which can ‘tick’ at any time.
\[ traces(CLOCK) = tick^* \]

(3) $VM$ is the process with $\alpha VM = \{coin, choc\}$ which repeatedly supplies a chocolate after a coin is inserted.
\[ traces(VM) = \{s : seq\{coin, choc\} | \exists n : \mathbb{N} \cdot s \leq \langle coin, choc\rangle^n\} \]

(4) $WALK$ is a one-dimensional random walk process with $\alpha WALK = \{left, right\}$.
\[ traces(WALK) = (left \cup right)^* \]

(5) $LIFE$ is the process with $\alpha LIFE = \{beat\}$ which can stop (die) at any time.
\[ traces(LIFE) = beat^* \]
Prefix

A process which may participate in event \( a \) then act according to process description \( P \) is written

\[ a \rightarrow P. \]

The event \( a \) is initially enabled by the process and occurs as soon as it is requested by its environment, all other events are refused initially. The event \( a \) is sometimes referred to as the "guard" of the process. For examples:

\[ VMU = coin \rightarrow STOP \]

\[ SHORTLIFE = (beat \rightarrow (beat \rightarrow STOP)) = beat \rightarrow beat \rightarrow STOP \]

\[ VMS = coin \rightarrow choc \rightarrow STOP \]
Other CSP primitives:

- $P; Q$ (sequential composition)
- $P [[X]] Q$ (synchronous), $P ||| Q$ (asynchronous)
- $a \to P \boxdot b \to Q$ (external choice), $a \to P \sqcap b \to Q$ (internal choice)
- $P_1 \lor e \to P_2$ (interrupt process)
Sequential Composition

- The second form of sequencing is process sequencing. A distinguished event ✓ is used to represent and detect process termination.

- The sequential composition of $P$ and $Q$, written $P; Q$, acts as $P$ until $P$ terminates by communicating ✓ and then proceeds to act as $Q$.

- The termination signal is hidden from the process environment and therefore occurs as soon as enabled by $P$. The process which may only terminate is written SKIP.
**Parallel composition**

The parallel composition of processes $P$ and $Q$, synchronised on event set $X$, is written

$$P \parallel X \parallel Q.$$  

No event from $X$ may occur in $P \parallel X \parallel Q$ unless enabled jointly by both $P$ and $Q$. When events from $X$ do occur, they occur in both $P$ and $Q$ simultaneously and are referred to as *synchronisations*. Events not from $X$ may occur in either $P$ or $Q$ separately but not jointly. For example, in the process described by

$$(a \to P) \parallel a \parallel (c \to a \to Q)$$

all $a$ events must be synchronisations between the two processes.

In an asynchronous parallel combination

$$P \parallel\parallel Q$$

both components $P$ and $Q$ execute concurrently without any synchronisations.
Choice

Diversity of behaviour is introduced through two choice operators.

The external choice operator allows a process a choice of behaviour according to what events are requested by its environment. The process

\[(a \rightarrow P) \sqcup (b \rightarrow Q)\]

begins with both \(a\) and \(b\) enabled. The environment chooses which event actually occurs by requested one or the other first. Subsequent behaviour is determined by the event which actually occurred, \(P\) after \(a\) and \(Q\) after \(b\) respectively.

Internal choice represents variation in behaviour determined by the internal state of the process. The process

\[a \rightarrow P \sqcap b \rightarrow Q\]

may initially enable either \(a\), or \(b\), or both, as it wishes, but must act subsequently according to which event actually occurred. The environment cannot affect internal choice.
Channel

A channel is a collection of events of the form $c.n$: the prefix $c$ is called the channel name and the collection of suffixes is called the values of the channel.

When an event $c.n$ occurs it is said that the value $n$ is communicated on channel $c$. When the value of a communication on a channel is determined by the environment (external choice) it is called an input and when it is determined by the internal state of the process (internal choice) it is called an output.

It is convenient to write $c?n : N \to P(n)$ to describe behaviour over a range of allowed inputs instead of the longer $\ □ n : N \bullet c.n \to P(n)$. Similarly the notation $c!n : N \to P(n)$ is used instead of $\ □ n : N \bullet c.n \to P(n)$ to represent a range of outputs.

e.g.

$$COPYBIT = in.0 \to out.0 \to COPYBIT \ □ \ in.1 \to out.1 \to COPYBIT$$

$$\alpha COPYBIT = \{in.0, out.0, in.1, out.1\}$$
**Interrupt**

The interrupt process $P_1 \lor e \rightarrow P_2$ behaves as $P_1$ until the first occurrence of interrupt event $e$, then the control passes to $P_2$.

**Recursion**

Recursion is used to given finite representations of non-terminating processes. The process expression

$$\mu P \cdot a?n : \mathbb{N} \rightarrow b!f(n) \rightarrow P$$

describes a process which repeatedly inputs a natural on channel $a$, calculates some function $f$ of the input, and then outputs the result on channel $b$. 
State parameters

In general, the behaviour of a process at any point in time may be dependent on its internal state and this may conceivably take an infinite range of values.

It is often not possible to provide a finite representation of a process without introducing some notation for representing this internal process state.

The approach adopted by CSP is to allow a process definition to be parameterised by state variables. Thus a definition of the form

\[ P_{n:N} \equiv Q(n) \]

represents a (possibly infinite) family of definitions, one for each possible value of \( n \).

There is no inherent notion of process state in CSP, but rather these annotations are a convenient way to provide a finite representation of an infinite family of process descriptions.
Traces of Processes

\[ traces(a \rightarrow P) = \]
\[ \{ t : \text{seq } A \mid t = \langle \rangle \] 
\[ \vee \]
\[ \text{head } t = a \land \text{tail } t \in traces(P) \}\]

\[ traces(a \rightarrow P \mid b \rightarrow Q) = \]
\[ traces(a \rightarrow P) \cup traces(b \rightarrow Q) \]

\[ traces(x : B \rightarrow P(x)) = \]
\[ \{ t : \text{seq } A \mid t = \langle \rangle \] 
\[ \vee \]
\[ \text{head } t \in B \land \text{tail } t \in traces(P(\text{head } t)) \}\]
\[ = \bigcup \{ x : B \cdot traces(x \rightarrow P(x)) \} \]
Traces of Recursive Process

\[ traces(\mu X : A \bullet F(X)) = \]
\[ \bigcup n : \mathbb{N} \bullet traces(F^n(STOP_A)) \]

\[ VM = \mu X : \{coin, choc\} \bullet (coin \rightarrow choc \rightarrow X) \]

so in this case

\[ F(X) = coin \rightarrow choc \rightarrow X \]

and

\[ F^0(STOP_A) = STOP_A \]
\[ F(STOP_A) = coin \rightarrow choc \rightarrow STOP_A \]
\[ F^2(STOP_A) \]
\[ = F(F(STOP_A)) \]
\[ = coin \rightarrow choc \rightarrow (coin \rightarrow choc \rightarrow STOP_A) \]
\[ = coin \rightarrow choc \rightarrow coin \rightarrow choc \rightarrow STOP_A \]

\[ F^3(STOP_A) = \cdots \]

\[ \cdots \]
More on Concurrency

\[ P \parallel Q \]

\( P \) and \( Q \) must co-operate on all common actions; in general \( \alpha P \) is not the same as \( \alpha Q \).

Processes with the same alphabet (interaction) Example:

\[ VMC \vdash coin \rightarrow (choc \rightarrow VMC \]
\[ \quad \mid \quad bisc \rightarrow VMC) \]

\[ CHOCLOV = (choc \rightarrow CHOCLOV \]
\[ \quad \mid \quad coin \rightarrow choc \rightarrow CHOCLOV) \]

\[ VMC \parallel CHOCLOV = \mu X \bullet (coin \rightarrow choc \rightarrow X) \]

(assuming \( VMC \) and \( CHOCLOV \) have the same alphabet, otherwise?)
Processes with the similar alphabet (interaction) Example 1:

\[ VMH = on \rightarrow coin \rightarrow choc \rightarrow off \rightarrow VMH \]

\[ \alpha VMH = \{ on, coin, choc, off \} \]

\[ CUST = on \rightarrow (coin \rightarrow bisc \rightarrow CUST \]
\[ \mid curse \rightarrow coin \rightarrow choc \rightarrow CUST) \]

\[ \alpha CUST = \{ on, coin, bisc, curse, choc \} \]

\[ VMH \parallel CUST = \mu X \bullet \]
\[ (on \rightarrow (coin \rightarrow bisc \rightarrow STOP \]
\[ \mid curse \rightarrow coin \rightarrow choc \rightarrow off \rightarrow X)) \]
Processes with the similar alphabet (interaction) Example 3:

\[ SLOWALK = (left \rightarrow rest \rightarrow SLOWALK \]
\[ \quad \left| \quad right \rightarrow rest \rightarrow SLOWALK) \]

\[ SLOCLIMB = (up \rightarrow rest \rightarrow SLOCLIMB \]
\[ \quad \left| \quad down \rightarrow SLOCLIMB) \]

\[ SLOTREK = SLOWALK \parallel SLOCLIMB \]

Note:

\[ \langle up, rest \rangle \notin \text{traces}(SLOTREK) \]
Processes with disjoint alphabets (interleaving)

\[ WALK = (x : \{left, right\} \rightarrow WALK) \]

\[ \alpha WALK = \{left, right\} \]

\[ CLIMB = (y : \{up, down\} \rightarrow CLIMB) \]

\[ \alpha CLIMB = \{up, down\} \]

\[ TREK = WALK \parallel CLIMB \]

\[ \alpha TREK = \{left, right, up, down\} \]

\[ TREK = (z : \{left, right, up, down\} \rightarrow TREK) \]
Another example with disjoint alphabets

\[ VM = \text{coin} \rightarrow \text{choc} \rightarrow VM \]

\[ CLOCK = \text{tick} \rightarrow CLOCK \]

\[ VM \parallel CLOCK \]

is a vending machine that can ‘tick’ at any time.
Laws for Concurrency

L1 \[ P \parallel Q = Q \parallel P \]

L2 \[ P \parallel (Q \parallel R) = (P \parallel Q) \parallel R \]

L3 \[ P \parallel STOP_{\alpha P} = STOP_{\alpha P} \]
Let

\[ a \in (\alpha P - \alpha Q) \]
\[ b \in (\alpha Q - \alpha P) \]
\[ \{ c, d \} \subseteq (\alpha P \cap \alpha Q) \]

\begin{align*}
\textbf{L4A} & & (c \rightarrow P) \parallel (c \rightarrow Q) = c \rightarrow (P \parallel Q) \\
\textbf{L4B} & & (c \rightarrow P) \parallel (d \rightarrow Q) = \text{STOP} \quad \text{if } c \neq d \\
\textbf{L5A} & & (a \rightarrow P) \parallel (c \rightarrow Q) = a \rightarrow (P \parallel (c \rightarrow Q)) \\
\textbf{L5B} & & (c \rightarrow P) \parallel (b \rightarrow Q) = b \rightarrow ((c \rightarrow P) \parallel Q) \\
\textbf{L6} & & (a \rightarrow P) \parallel (b \rightarrow Q) = (a \rightarrow (P \parallel (b \rightarrow Q))) \parallel b \rightarrow ((a \rightarrow P) \parallel Q))
\end{align*}
Traces for Concurrency

If \( t \in \text{seq } A \) then
\[
   t \upharpoonright B
\]
denotes the trace of \( t \) restricted to events in \( B \).
e.g.
\[
   \langle c, a, d, b, a, c \rangle \upharpoonright \{b, c\} = \langle c, b, c \rangle
\]

If
\[
   A = \alpha P \cup \alpha Q
\]
then
\[
   \text{traces}(P \parallel Q) =
\]
\[
   \left\{ t : \text{seq } A \mid (t \upharpoonright \alpha P) \in \text{traces}(P) \land (t \upharpoonright \alpha Q) \in \text{traces}(Q) \right\}
\]
Interaction Deadlock

If

\[
GREEDY = (choc \rightarrow GREEDY \mid bisc \rightarrow GREEDY)
\]

\[
VMC = coin \rightarrow (choc \rightarrow VMC \mid bisc \rightarrow VMC)
\]

then

\[
GREEDY \parallel VMC = STOP \quad \text{(GREEDY and VMS have the same alphabet)}
\]

If

\[
BISCLOV = coin \rightarrow bisc \rightarrow BISCLOV
\]

\[
VM = coin \rightarrow choc \rightarrow VM
\]

then

\[
BISCLOV \parallel VM = coin \rightarrow STOP
\]

Such deadlock cannot be resolved by the environment.
Communication Deadlock

\[ P = \text{get.board} \rightarrow \text{play.chess} \rightarrow P \mid \text{get.racket} \rightarrow \text{play.tennis} \rightarrow P \]
\[ Q = \text{get.pieces} \rightarrow \text{play.chess} \rightarrow Q \mid \text{get.ball} \rightarrow \text{play.tennis} \rightarrow Q \]

A possible trace of \( P \parallel Q \) is
\[ \langle \text{get.racket}, \text{get.pieces} \rangle \]
which is deadlocked as no further events are possible.
This deadlock arises as \( P \) and \( Q \) are committed to conflicting activities; it can be resolved by the environment directing the processes.

\[ \text{GUIDE} = (\text{get.board} \rightarrow \text{get.pieces} \rightarrow \text{GUIDE} \mid \text{get.racket} \rightarrow \text{get.ball} \rightarrow \text{GUIDE}) \]

Now
\[ P \parallel Q \parallel \text{GUIDE} \]
cannot deadlock.
Resource Deadlock (Dining Philosophers)

There are two philosophers $P$ and $Q$ and two forks $G$ and $S$

$p.sits$ is the event ‘$P$ sits at the table’
$p.gold.up$ is the event ‘$P$ picks-up fork $G$’
$p.silver.up$ is the event ‘$P$ picks-up fork $S$’
$p.gold.down$ is the event ‘$P$ puts-down fork $G$’
$p.silver.down$ is the event ‘$P$ puts-down fork $S$’
$p.leaves$ is the event ‘$P$ leaves the table’

$q.sits$ is the event ‘$Q$ sits at the table’
$q.gold.up$ is the event ‘$Q$ picks-up fork $G$’
$q.silver.up$ is the event ‘$Q$ picks-up fork $S$’
$q.gold.down$ is the event ‘$Q$ puts-down fork $G$’
$q.silver.down$ is the event ‘$Q$ puts-down fork $S$’
$q.leaves$ is the event ‘$Q$ leaves the table’
\[ P = p.sits \rightarrow p.gold.up \rightarrow p.silver.up \rightarrow p.gold.down \rightarrow p.silver.down \rightarrow p.leaves \rightarrow P \]

\[ Q = q.sits \rightarrow q.silver.up \rightarrow q.gold.up \rightarrow q.silver.down \rightarrow q.gold.down \rightarrow q.leaves \rightarrow Q \]

\[ G = p.gold.up \rightarrow p.gold.down \rightarrow G \mid q.gold.up \rightarrow q.gold.down \rightarrow G \]

\[ S = p.silver.up \rightarrow p.silver.down \rightarrow S \mid q.silver.up \rightarrow q.silver.down \rightarrow S \]

\[ COLLEGE = P \parallel Q \parallel G \parallel S \]

has the deadlocked trace

\[ \langle p.sits, q.sits, p.gold.up, q.silver.up \rangle \]

This deadlock arises as \( P \) and \( Q \) are contending for common resources; it can be resolved by the environment allocating the resources.

\[ FOOT = p.sits \rightarrow p.leaves \rightarrow FOOT \mid q.sits \rightarrow q.leaves \rightarrow FOOT \]

Now it cannot deadlock:

\[ NEWCOLLEGE = COLLEGE \parallel FOOT \]
Starvation (Fairness)

The process NEWCOLLEGE does not ensure that the philosophers are treated fairly. The footman could perform
\[ \langle p.sits, p.leaves, p.sits, p.leaves, \cdots \rangle \]
leading to the exclusion of \( q \).

Replacing FOOT by the process
\[
FAIRFOOT = p.sits \rightarrow p.leaves \rightarrow q.sits \rightarrow q.leaves \rightarrow FAIRFOOT
\]
will ensure fairness but is too restrictive in practice.
Summary

CSP is an event-based formalism. The allowed sequences of events are clearly and concisely determined by the CSP model.

CSP still has no standard support for state modeling in the form of mathematical toolkits and libraries for constructing and reasoning about complex internal state. There is also no direct support for modeling/reasoning about Fairness.

At NUS, we have extended CSP with shared variables, array types, programming constructs … and developed a powerful model checker called PAT (Process Analysis Toolkit): http://www.comp.nus.edu.sg/~pat/

Next we will give you a brief demo through a case study on an embedded keyless system for the modern cars.
Keyless System

• One of the latest automotive technologies, push-button keyless system, allows you to start your car's engine without the hassle of key insertion and offers great convenience.

• Push-button keyless system allows owner with key-fob in her pocket to unlock the door when she is very near the car. The driver can slide behind the wheel, with the key-fob in her pocket (briefcase or purse or anywhere inside the car), she can push the start/stop button on the control panel. Shutting off the engine is just as hassle-free, and is accomplished by merely pressing the start/stop button.

• These systems are designed so it is impossible to start the engine without the owner's key-fob and it cannot lock your key-fob inside the car because the system will sense it and prevent the user from locking them in.

• However, the keyless system can also surprise you as it may allow you to drive the car without key-fob. E.g. you can drive without key!
Constant and variables

- `#define N 2;` // number of owners
- `#define far 0;` // owner is out and far away from the car
- `#define near 1;` // owner is close enough to open/lock the door if she has the keyfob
- `#define in 2;` // owner is in the car
- `#define off 0;` // engine is off
- `#define on 1;` // engine is on
- `#define unlock 0;` // door is unlocked but closed
- `#define lock 1;` // door is locked (must be closed)
- `#define open 2;` // door is open
- `#define incar -1;` // keyfob is put inside car
- `#define faralone -2;` // keyfob is put outside and far

- `var owner[N];` // owners' position. initially, all users are far away from the car
- `var engine = off;` // engine status, initially off
- `var door = lock;` // door status, initially locked
- `var key = 0;` // key fob position, initially, it is with first owner
- `var moving = 0;` // car moving status, 0 for stop and 1 for moving
- `var fuel = 10;` // energy costs, say 1 for a short drive and 5 for a long driving
Owner positions

car = (∥i:{0..N-1} @ (owner_pos(i) || motor(i) || door_op(i) || key_pos(i))),

owner_pos(i) =
    [owner[i] == far]towards.i{owner[i] = near} -> owner_pos(i)
    []
    [owner[i] == near]goaway.i{owner[i] = far} -> owner_pos(i)
    []
    [owner[i] == near && door == open && moving==0]getin.i{owner[i] = in}
        -> owner_pos(i)
    []
    [owner[i] == in && door == open && moving==0]goout.i{owner[i] = near}
        -> owner_pos(i);
Key-fob position

key_pos(i) =
    [key == i && owner[i] == in] putincar.i{key = incar} -> key_pos(i)
    []
    [key == i && owner[i] == far] putaway.i{key = faralone} -> key_pos(i)
    []
[(key == faralone && owner[i] == far) || (key == incar && owner[i] == in)] getkey.i{key = i} -> key_pos(i);
Door operation

door_op(i) =

[key == i && owner[i]==near && door == lock &&
  moving==0]unlockopen.i{door = open} -> door_op(i)

[owner[i]==near && door==unlock &&
  moving==0]justopen.i{door = open} -> door_op(i)

[door != open && owner[i] == in]insideopen.i{door = open}
  -> door_op(i)

[door == open]close.i{door = unlock} -> door_op(i)

[door==unlock&&owner[i]==in]insidelock.i{door=lock} -> door_op(i)

[door == unlock && owner[i]==near &&
key==i]outsidelock.i{door=lock} -> door_op(i);
Motor

\[
\text{motor}(i) = \\
\left[ \text{owner}[i]==\text{in} \&\& (\text{key}==i \| \text{key}==\text{incar}) \&\& \text{engine}==\text{off} \&\& \text{fuel}!=0 \right] \text{turnon}.i\{\text{engine} = \text{on}\} -> \text{motor}(i) \\
\left[ \text{engine}==\text{on} \&\& \text{owner}[i]==\text{in} \&\& \text{moving}==0 \right] \text{startdrive}.i\{\text{moving}=1\} -> \text{motor}(i) \\
\left[ \text{moving}==1 \&\& \text{fuel}!=0 \right] \text{shortdrive}.i\{\text{fuel}=\text{fuel}-1; \text{if (fuel}==0) \{\text{engine}=\text{off}; \text{moving} =0\}\} -> \text{motor}(i) \\
\left[ \text{moving}==1 \&\& \text{fuel} > 5 \right] \text{longdrive}.i\{\text{fuel}=\text{fuel}-5; \text{if (fuel}==0) \{\text{engine}=\text{off}; \text{moving} =0\}\} -> \text{motor}(i) \\
\left[ \text{engine}==\text{on} \&\& \text{moving}==1 \&\& \text{owner}[i]==\text{in} \right] \text{stop}.i\{\text{moving}=0\} -> \text{motor}(i) \\
\left[ \text{fuel}==0 \&\& \text{engine}==\text{off} \right] \text{refill}\{\text{fuel}=10\} -> \text{motor}(i) \\
\left[ \text{engine}==\text{on} \&\& \text{moving}==0 \&\& \text{owner}[i]==\text{in} \right] \text{turnoff}.i\{\text{engine} = \text{off}\} -> \text{motor}(i); \\
\]

\[
\text{car} = \left( \forall i:\{0..N-1\} \right) \left( \text{motor}(i) \| \text{door}_\text{op}(i) \| \text{key}_\text{pos}(i) \| \text{owner}_\text{pos}(i) \right));
\]
#define runwithoutowner (moving==1 && owner[0] == far && owner[1] == far);
#define ownerdrivetogether (moving==1 && owner[0] == in && owner[1] == in);
#define keylockinside (key == incar && door == lock && owner[0] != in && owner[1] != in);
#define drivewithoutengineon (moving==1 && engine==off);
#define drivewithoutfuel (moving==1&&fuel==0);
#define drivewithoutkeyholdbyother (moving ==1 && owner[1] == in && owner[0] == far && key == 0);

#define car deadlockfree;
#define car |= []<> longdrive.0;
#define car reaches keylockinside;
#define car reaches runwithoutowner;
#define car reaches ownerdrivetogether;
#define car reaches drivewithoutengineon;
#define car reaches drivewithoutfuel;
#define car reaches drivewithoutkeyholdbyother;