Checking for Feasibility

Before we discuss model checking response properties we discuss the problem of checking whether a given FDS is feasible.

A run of an FDS is an infinite sequence of states which satisfies the requirements of initiality and consecution but not necessarily any of the fairness requirements.

A state $s$ of an FDS $D$ is called reachable if it participates in some run of $D$.

A state $s$ is called feasible if it participates in some computation. The FDS is called feasible if it has at least one computation.

A set of states $S$ is defined to be an F-set if it satisfies the following requirements:

F1. All states in $S$ are reachable.

F2. Each state $s \in S$ has a $\rho$-successor in $S$.

F3. For every state $s \in S$ and every justice requirement $J \in J$, there exists an $S$-path leading from $s$ to some $J$-state.

F4. For every state $s \in S$ and every compassion requirement $(p, q) \in C$, either there exists an $S$-path leading from $s$ to some $q$-state, or $s$ satisfies $\neg p$. 
F-Sets Imply Feasibility

Claim 3. [F-sets]
A reachable state $s$ is feasible iff it has a path leading to some $F$-set.

Proof:
Assume that $s$ is a feasible state. Then it participates in some computation $\sigma$. Let $S$ be the (finite) set of all states that appear infinitely many times in $\sigma$. We will show that $S$ is an $F$-set. It is not difficult to see that there exists a cutoff position $t \geq 0$ such that $S$ contains all the states that appear at positions beyond $t$.

Obviously all states appearing in $\sigma$ are reachable. If $s \in S$ appears in $\sigma$ at position $i > t$ then it has a successor $s_{i+1} \in \sigma$ which is also a member of $S$.

Let $s = s_i \in \sigma$, $i > t$ be a member of $S$ and $J \in \mathcal{J}$ be some justice requirement. Since $\sigma$ is a computation it contains infinitely many $J$-positions. Let $k \geq i$ one of the $J$-positions appearing later than $i$. Then the path $s_i, \ldots, s_k$ is an $S$-path leading from $s$ to a $J$-state.

Let $s = s_i \in \sigma$, $i > t$ be a member of $S$ and $(p,q) \in \mathcal{C}$ be some compassion requirement. There are two possibilities by which $\sigma$ may satisfy $(p,q)$. Either $\sigma$ contains only finitely many $p$-positions, or $\sigma$ contains infinitely many $q$ positions. It follows that either $S$ contains no $p$-states, or it contains some $q$-states which appear infinitely many times in $\sigma$. In the first case, $s$ satisfies $\neg p$. In the second case, there exists a path leading from $s_i$ to $s_k$, a $q$-state such that $k \geq i$. 
Proof Continued

In the other direction, assume the existence of an $F$-set $S$ and a reachable state $s$ which has a path leading to some state $s_1 \in S$. We will show that there exists a computation $\sigma$ which contains $s$.

Since $s$ is reachable and has a path leading to state $s_1 \in S$, there exists a finite sequence of states $\pi$ leading from an initial state to $s_1$ and passing through $s$. We will show how $\pi$ can be extended to a computation by an infinite repetition of the following steps. At any point in the construction, we denote by $\text{end}(\pi)$ the state which currently appears last in $\pi$.

- We know that $\text{end}(\pi) \in S$ has a successor $s \in S$. Append $s$ to the end of $\pi$.
- Consider in turn each of the justice requirements $J \in J$. We append to $\pi$ the $S$-path $\pi_J$ connecting $\text{end}(\pi)$ to a $J$-state.
- Consider in turn each of the compassion requirements $(p, q) \in C$. If there exists an $S$-path $\pi_q$, connecting $\text{end}(\pi)$ to a $q$-state, we append $\pi_q$ to the end of $\pi$. Otherwise, we do not modify $\pi$. We observe that if there does not exist an $S$-path leading from $\text{end}(\pi)$ to a $q$-state, then $\text{end}(\pi)$ and all of its progeny within $S$ must satisfy $\neg p$.

It is not difficult to see that the infinite sequence constructed in this way is a computation. \[\blacksquare\]
Computing F-Sets

Assume an assertion $\varphi$ which characterizes an F-set. Translating the requirements 1–4 into formulas, we obtain the following requirements:

\[
\begin{align*}
\varphi & \rightarrow \text{reachable}_D \\
\varphi & \rightarrow \rho \lozenge \varphi & \text{Every } \varphi\text{-state has a } \varphi\text{-successor} \\
\varphi & \rightarrow (\varphi \land \rho)^* \lozenge (\varphi \land J) & \text{For every } J \in \mathcal{J} \\
\varphi & \rightarrow \neg p \lor (\varphi \land \rho)^* \lozenge (\varphi \land q) & \text{For every } (p, q) \in C
\end{align*}
\]

This can be summarized as

\[
\varphi \rightarrow \left( \text{reachable}_D \land \rho \lozenge \varphi \land \bigwedge_{J \in \mathcal{J}} (\varphi \land \rho)^* \lozenge (\varphi \land J) \land \bigwedge_{(p, q) \in C} \neg p \lor (\varphi \land \rho)^* \lozenge (\varphi \land q) \right)
\]

Since we are interested in a maximal F-set, the computation can be expressed as:

\[
\nu \varphi. \left( \text{reachable}_D \land \rho \lozenge \varphi \land \bigwedge_{J \in \mathcal{J}} (\varphi \land \rho)^* \lozenge (\varphi \land J) \land \bigwedge_{(p, q) \in C} \neg p \lor (\varphi \land \rho)^* \lozenge (\varphi \land q) \right)
\]
Algorithmic Interpretation

Computing the maximal fix-point as a sequence of iterations, we can describe the computational process as follows:

Start by letting $\varphi := \text{reachable}_D$. Then repeat the following steps:

- Remove from $\varphi$ all states which do not have a $\varphi$-successor.

- For each $J \in \mathcal{J}$, remove from $\varphi$ all states which do not have a $\varphi$-path leading to a $J$-state.

- For each $(p, q) \in \mathcal{C}$, remove from $\varphi$ all $p$-states which do not have a $\varphi$-path leading to a $q$-state.

until no further change.

To check whether an FDS $D$ is feasible, we compute for it the maximal $F$-set and check whether it is empty. $D$ is feasible iff the maximal $F$-set is not-empty.
Example

As an example, consider the following FDS:

\[
\begin{align*}
J_1 & : \ x \neq 1 \\
C_1 & : \ (x = 3, x = 5) \\
C_2 & : \ (x = 2, x = 1)
\end{align*}
\]

We set \( \varphi_0 : \{0..5\} \) and then proceed as follows:

- Removing from \( \varphi_0 \) all \((x = 2)\)-states which do not have a \( \varphi_0 \)-path leading to an \((x = 1)\)-state, we are left with \( \varphi_1 : \{0, 1, 3, 4, 5\} \).
- Successively removing from \( \varphi_1 \) all states without successors, leaves \( \varphi_2 : \{3, 4\} \).
- Removing from \( \varphi_2 \) all \((x = 3)\)-states which do not have a \( \varphi_2 \)-path leading to a \((x = 5)\)-state, we are left with \( \varphi_3 : \{4\} \).
- No reasons to remove any further states from \( \varphi_3 : \{4\} \), so this is our final set.

We conclude that the above FDS is feasible.
Verifying Response Properties Through Feasibility Checking

Let $\mathcal{D} : \langle V, \Theta, \rho, \mathcal{J}, \mathcal{C} \rangle$ be an FDS and $p \Rightarrow \Box q$ be a response property we wish to verify over $\mathcal{D}$. Let $\text{reachable}_\mathcal{D}$ be the assertion characterizing all the reachable states in $\mathcal{D}$.

We define an auxiliary FDS $\mathcal{D}_{p,q} : \langle V, \Theta_{p,q}, \rho_{p,q}, \mathcal{J}, \mathcal{C} \rangle$, where

$$\Theta_{p,q} : \text{reachable}_\mathcal{D} \land p \land \neg q$$

$$\rho_{p,q} : \rho \land \neg q'$$

Thus, $\Theta_{p,q}$ characterizes all the $\mathcal{D}$-reachable $p$-states which do not satisfy $q$, while $\rho_{p,q}$ allows any $\rho$-step as long as the successor does not satisfy $q$.

Claim 4. [Model Checking Response]
$\mathcal{D} \models p \Rightarrow \Box q$ iff $\mathcal{D}_{p,q}$ is unfeasible.

Proof: The claim is justified by the observation that every computation of $\mathcal{D}_{p,q}$ can be extendable to a computation of $\mathcal{D}$ which violates the response property $p \Rightarrow \Box q$. Indeed, let $\sigma : s_k, s_{k+1}, \ldots$ be a computation of $\mathcal{D}_{p,q}$. By the definition of $\Theta_{p,q}$, we know that $s_k$ is a $\mathcal{D}$-reachable $p$-state. Thus, there exists, a finite sequence $s_0, \ldots, s_k$, such that $s_0$ is $\mathcal{D}$-initial. The infinite sequence $s_0, \ldots, s_k, s_{k+1}, s_{k+1}, \ldots$ is a computation of $\mathcal{D}$ which contains a $p$-state at position $k$, and has no following $q$-state. This sequence violates $p \Rightarrow \Box q$. □
**Example: MUX-SEM**

Following is the set of all reachable states of program MUX-SEM.

Assume we wish to verify the property $T_2 \Rightarrow \Diamond C_2$. We start by forming $\text{MUX-SEM}_{T_2,C_2}$, whose set of reachable states is given by:

First, we eliminate all $(T_2 \land y = 1)$-states which do not have a path leading to a $C_2$-state. This leaves us with:

Next, we eliminate all states which do not have a path leading to a $\neg C_1$-state. This leaves us with nothing. We conclude that $\text{MUX-SEM} \models T_2 \Rightarrow \Diamond C_2$. 
Demonstrating what can be achieved by **Formal Verification**

We will illustrate how formal verification (when it works) can aid us in the development of **reliable programs**.

Consider the following program **TRY-1** which attempts to solve the mutual exclusion problem by shared variables:

```
local y_1, y_2 : boolean where y_1 = y_2 = 0

P_1 ::
[l_0 : loop forever do
  l_1 : Non-Critical
  l_2 : await \neg y_2
  l_3 : y_1 := 1
  l_4 : Critical
  l_5 : y_1 := 0
] ||

P_2 ::
[m_0 : loop forever do
  m_1 : Non-Critical
  m_2 : await \neg y_1
  m_3 : y_2 := 1
  m_4 : Critical
  m_5 : y_2 := 0
]
```

Variables $y_1$ and $y_2$ signify whether processes $P_1$ and $P_2$ are interested in entering their critical sections.
Program Properties: Invariance

For program TRY-1, the property of mutual exclusion can be specified by requiring that the assertion

\[ \varphi_{\text{exclusion}} : \ \neg (\text{at}_L \land \text{at}_M) \]

be an invariant of TRY-1. This implies that no execution of TRY-1 can ever get to a state in which both processes execute their critical sections at the same time.
Invoking TLV

To check whether assertion $\varphi_{exclusion}$ is an invariant of program $TRY\!-\!1$, we invoke the model checking tool TLV, a model checker based on the SMV tool developed in CMU by Ken McMillan and Ed Clarke.

We prepare two input files: $try1.spl$ which contains the SPL representation of $TRY\!-\!1$, and $try1.pf$, a proof script file. The proof script file contains some printing commands, definition of the assertion $\varphi_{exclusion}$ and a command to check its invariance over the program.

We will present each of these input files.
File try1.spl

local y1 : bool where y1 = F;
y2 : bool where y2 = F;

P1:: [l_0: loop forever do [
l_1: noncritical;
l_2: await !y2;
l_3: y1 := T;
l_4: critical;
l_5: y1 := F ] ]

||
P2:: [m_0: loop forever do [
m_1: noncritical;
m_2: await !y1;
m_3: y2 := T;
m_4: critical;
m_5: y2 := F ] ]
Print "Check for Mutual Exclusion\n";

Let exclusion := !(at_l_4 & at_m_4);
Call Invariance(exclusion);

The call to procedure Invariance invokes the process which checks whether any reachable state violates the assertion exclusion.
Results of Verifying TRY-1

The results of model-checking TRY-1 are

>> Load "try1.pf";
Check for Mutual Exclusion
Model checking Invariance Property
*** Property is NOT VALID ***
Counter-Example Follows:

----- State no. 1 =
pi1 = l_0, pi2 = m_0, y1 = 0, y2 = 0,

----- State no. 2 =
pi1 = l_1, pi2 = m_0, y1 = 0, y2 = 0,

----- State no. 3 =
pi1 = l_1, pi2 = m_1, y1 = 0, y2 = 0,

----- State no. 4 =
pi1 = l_1, pi2 = m_2, y1 = 0, y2 = 0,

----- State no. 5 =
pi1 = l_1, pi2 = m_3, y1 = 0, y2 = 0,

----- State no. 6 =
pi1 = l_2, pi2 = m_3, y1 = 0, y2 = 0,

----- State no. 7 =
pi1 = l_3, pi2 = m_3, y1 = 0, y2 = 0,

----- State no. 8 =
pi1 = l_3, pi2 = m_4, y1 = 0, y2 = 1,

----- State no. 9 =
pi1 = l_4, pi2 = m_4, y1 = 1, y2 = 1,
Expressed in a More Readable Form

\[
\text{local } y_1, y_2 : \text{boolean where } y_1 = y_2 = 0
\]

\[
P_1 :: \begin{cases}
\ell_0 : \text{loop forever do} \\
\ell_1 : \text{Non-Critical} \\
\ell_2 : \text{await } \neg y_2 \\
\ell_3 : y_1 := 1 \\
\ell_4 : \text{Critical} \\
\ell_5 : y_1 := 0
\end{cases}
\]

\|

\[
P_2 :: \begin{cases}
m_0 : \text{loop forever do} \\
m_1 : \text{Non-Critical} \\
m_2 : \text{await } \neg y_1 \\
m_3 : y_2 := 1 \\
m_4 : \text{Critical} \\
m_5 : y_2 := 0
\end{cases}
\]

The counter example is:

\[
\langle \ell_0, m_0, y_1 : 0, y_2 : 0 \rangle, \langle \ell_1, m_0, y_1 : 0, y_2 : 0 \rangle, \langle \ell_1, m_1, y_1 : 0, y_2 : 0 \rangle, \langle \ell_1, m_2, y_1 : 0, y_2 : 0 \rangle, \langle \ell_2, m_3, y_1 : 0, y_2 : 0 \rangle, \langle \ell_3, m_3, y_1 : 0, y_2 : 0 \rangle, \langle \ell_3, m_4, y_1 : 0, y_2 : 1 \rangle, \langle \ell_4, m_4, y_1 : 1, y_2 : 1 \rangle
\]

reaching the state \( \langle \ell_4, m_4, y_1 : 1, y_2 : 1 \rangle \) which violates mutual exclusion!

Obviously, the problem is that the processes test each other’s \( y \) value first and only later set their own \( y \).
Second Attempt: Set first and Test Later

The following program TRY-1 interchange the order of testing and setting:

\[
\text{local } y_1, y_2 : \text{boolean where } y_1 = y_2 = 0
\]

\[
P_1 ::
\begin{align*}
\ell_0 : \text{loop forever do} \\
\ell_1 : \text{Non-Critical} \\
\ell_2 : y_1 := 1 \\
\ell_3 : \text{await } \neg y_2 \\
\ell_4 : \text{Critical} \\
\ell_5 : y_1 := 0
\end{align*}
\]

\[
P_2 ::
\begin{align*}
m_0 : \text{loop forever do} \\
m_1 : \text{Non-Critical} \\
m_2 : y_2 := 1 \\
m_3 : \text{await } \neg y_1 \\
m_4 : \text{Critical} \\
m_5 : y_2 := 0
\end{align*}
\]

Let us see whether the program is now correct.
Program Properties: Absence of Deadlock

A state $s$ is said to be a deadlock state if no process can perform any action. In our FDS model, the idling transition is always enabled. Therefore, we define $s$ to be a deadlock state if it has no $D$-successor different from itself.

Mathematically, we can characterize all deadlock states by the assertion

$$\delta : \neg \exists V' \neq V : \rho(V, V')$$

and then check for the invariance of the assertion $\neg\delta$.

To check for the interesting properties of program TRY-2, we prepare the following script file:

```plaintext
Print "Check for Mutual Exclusion\n";
Let exclusion := !(at_l_4 & at_m_4);
Call Invariance(exclusion);
Run check_deadlock;
```
Model Checking TRY-2

We obtain the following results:

```plaintext
>> Load "try2.pf";
Check for Mutual Exclusion
Model checking Invariance Property
*** Property is VALID ***
Check for the absence of Deadlock.
Model checking Invariance Property
*** Property is NOT VALID ***
Counter-Example Follows:
---- State no. 1 =
pi1 = l_0, pi2 = m_0, y1 = 0, y2 = 0,
---- State no. 2 =
pi1 = l_1, pi2 = m_0, y1 = 0, y2 = 0,
---- State no. 3 =
pi1 = l_1, pi2 = m_1, y1 = 0, y2 = 0,
---- State no. 4 =
pi1 = l_1, pi2 = m_2, y1 = 0, y2 = 0,
---- State no. 5 =
pi1 = l_1, pi2 = m_3, y1 = 0, y2 = 1,
---- State no. 6 =
pi1 = l_2, pi2 = m_3, y1 = 0, y2 = 1,
---- State no. 7 =
pi1 = l_3, pi2 = m_3, y1 = 1, y2 = 1,
```
In a More Readable Form

\[ \text{local } y_1, y_2 : \text{boolean where } y_1 = y_2 = 0 \]

\[ P_1 :: \]
\[ \begin{cases} \ell_0 : \text{loop forever do} \\
\ell_1 : \text{Non-Critical} \\
\ell_2 : y_1 := 1 \\
\ell_3 : \text{await } \lnot y_2 \\
\ell_4 : \text{Critical} \\
\ell_5 : y_1 := 0 \end{cases} \]

\[ P_2 :: \]
\[ \begin{cases} m_0 : \text{loop forever do} \\
m_1 : \text{Non-Critical} \\
m_2 : y_2 := 1 \\
m_3 : \text{await } \lnot y_1 \\
m_4 : \text{Critical} \\
m_5 : y_2 := 0 \end{cases} \]

The counter example is:

\[ \langle \ell_0, m_0, y_1 : 0, y_2 : 0 \rangle, \langle \ell_1, m_0, y_1 : 0, y_2 : 0 \rangle, \langle \ell_1, m_1, y_1 : 0, y_2 : 0 \rangle, \]
\[ \langle \ell_1, m_2, y_1 : 0, y_2 : 0 \rangle, \langle \ell_1, m_3, y_1 : 0, y_2 : 1 \rangle, \langle \ell_2, m_3, y_1 : 0, y_2 : 1 \rangle, \]
\[ \langle \ell_3, m_3, y_1 : 1, y_2 : 1 \rangle \]

reaching the deadlock state \[ \langle \ell_3, m_3, y_1 : 1, y_2 : 1 \rangle \]


Try a Different Approach

The following program TRY-3 uses a variable $\text{turn}$ to indicate which process has the higher priority.

$$
\text{local } \text{turn} : [1..2] \text{ where } \text{turn} = 0
$$

$$
P_1 :: \begin{cases}
\ell_0 : \text{loop forever do} \\
\ell_1 : \text{Non-Critical} \\
\ell_2 : \text{await } \text{turn} = 1 \\
\ell_3 : \text{Critical} \\
\ell_4 : \text{turn} := 2
\end{cases}
\quad || 

P_2 :: \begin{cases}
m_0 : \text{loop forever do} \\
m_1 : \text{Non-Critical} \\
m_2 : \text{await } \text{turn} = 2 \\
m_3 : \text{Critical} \\
m_4 : \text{turn} := 1
\end{cases}
$$
Program Properties: Response

This property refers to two assertions $p$ and $q$. Written $p \Rightarrow \Box q$, it means

Every occurrence of a $p$-state must be followed by an occurrence of a $q$-state.

The response construct can be used to specify the property of accessibility. For example, the response property

$$at_{-l_2} \Rightarrow \Box at_{-l_3}$$

requires for program TRY-3 that every visit to $l_2$ must be followed by a visit to $l_3$.

To model check this property, we prepare the following file try3.pf:

```plaintext
Print "Check for Mutual Exclusion\n";
Let exclusion := !(at_l_3 & at_m_3);
Call Invariance(exclusion);
Run check_deadlock;
Print "\n Check Accessibility for P1\n";
Call Temp_Entail(at_l_2,at_l_3);
Print "\n Check Accessibility for P2\n";
Call Temp_Entail(at_m_2,at_m_3);
```
We obtain the following results:

```plaintext
>> Load "try3.pf";

Check for Mutual Exclusion
Model checking Invariance Property
*** Property is VALID ***

Check for the absence of Deadlock.
Model checking Invariance Property
*** Property is VALID ***

Check Accessibility for P1
Model checking...
*** Property is NOT VALID ***

Counter-Example Follows:

---- State no. 1 : pi1 = l_0, pi2 = m_0, turn = 1,
---- State no. 2 : pi1 = l_1, pi2 = m_0, turn = 1,
---- State no. 3 : pi1 = l_2, pi2 = m_0, turn = 1,
---- State no. 4 : pi1 = l_3, pi2 = m_0, turn = 1,
---- State no. 5 : pi1 = l_4, pi2 = m_0, turn = 1,
---- State no. 6 : pi1 = l_0, pi2 = m_0, turn = 2,
---- State no. 7 : pi1 = l_1, pi2 = m_0, turn = 2,
---- State no. 8 : pi1 = l_2, pi2 = m_0, turn = 2,

Loop back to state 8
```
In a More Readable Form

local turn : [1..2] where turn = 0

$P_1 :: \begin{bmatrix}
\ell_0 : \text{loop forever do} \\
\ell_1 : \text{Non-Critical} \\
\ell_2 : \text{await } turn = 1 \\
\ell_3 : \text{Critical} \\
\ell_4 : turn := 2
\end{bmatrix} \ || \ P_2 :: \begin{bmatrix}
m_0 : \text{loop forever do} \\
m_1 : \text{Non-Critical} \\
m_2 : \text{await } turn = 2 \\
m_3 : \text{Critical} \\
m_4 : turn := 1
\end{bmatrix}$

The counter example is:

$\langle \ell_0, m_0, turn : 1 \rangle, \langle \ell_1, m_0, turn : 1 \rangle, \langle \ell_2, m_0, turn : 1 \rangle$
$\langle \ell_3, m_0, turn : 1 \rangle, \langle \ell_4, m_0, turn : 1 \rangle, \langle \ell_0, m_0, turn : 2 \rangle$
$\langle \ell_1, m_0, turn : 2 \rangle, \langle \ell_2, m_0, turn : 2 \rangle$
Finally a good program for Mutual Exclusion

Following is a good shared variables solution to the mutual exclusion problem.

Peterson’s for 2 Processes:

```
local y1, y2 : boolean where y1 = y2 = 0
s : {1, 2} where s = 1
```

```
l0 : loop forever do
    l1 : Non-Critical
        l2 : (y1, s) := (1, 1)
        l3 : await y2 = 0 \lor s \neq 1
        l4 : Critical
        l5 : y1 := 0
    - P1 -

    m0 : loop forever do
        m1 : Non-Critical
        m2 : (y2, s) := (1, 2)
        m3 : await y1 = 0 \lor s \neq 2
        m4 : Critical
        m5 : y2 := 0
    - P2 -
```

Variables $y_1$ and $y_2$ signify whether processes $P_1$ and $P_2$ are interested in entering their critical sections. Variable $s$ serves as a tie-breaker. It always contains the signature of the last process to enter the waiting location ($l_3$, $m_3$). Model checking this program, we find that it satisfies the three properties of (invariance of) mutual exclusion, absence of deadlock, and accessibility.