Deductive Verification

The method of deducive verification enables proofs of temporal properties of systems with infinitely many states. It is based on the application of proof rules which have the general form

\[
\begin{array}{c}
\varphi_1 \\
\ldots \\
\varphi_n \\
\hline
\psi
\end{array}
\]

This rule implies that if the premises \( \varphi_1, \ldots, \varphi_n \) are valid, then so is the conclusion \( \psi \). Typically, the premises are assertions (non-temporal formulas) while the conclusion is temporal.

We distinguish several modes of validity:

- \( \models p \) General Validity — Formula \( p \) is valid over all models. If \( p \) is an assertions, this reduces to first-order validity.
- \( D \models p \) D-state validity — Assertion \( p \) is valid over all \( D \)-reachable states. The same as \( D \models \Box p \).
- \( D \models p \) D-validity — Formula \( p \) is valid over all \( D \)-computations.

Verification of Invariance Properties

We may use the following basic invariance rule to prove the invariance of assertion \( p \). That is, establish that the formula \( \Box p \), for an assertion \( p \) is \( D \)-valid.

\[
\text{Rule BINV} \quad \begin{array}{c}
\text{I1. } \Theta \rightarrow p \\
\text{I2. } p \land \rho \rightarrow p' \\
\hline
\rho
\end{array}
\]

An assertion \( p \) satisfying I1 and I2 is called inductive.

Claim 13. Rule BINV is sound.

Proof Let \( \sigma : s_0, s_1, \ldots \) be a computation of \( D \). By premise I1, \( s_0 \) satisfies \( p \). We show that, for every \( j = 0, 1, \ldots \), the validity of \( p \) propagates from \( s_j \) to \( s_{j+1} \). Assume that \( s_j \models p \). This implies that \( p(s_j[V]) = 1 \). Since \( s_{j+1} \) is a \( D \)-successor of \( s_j \), it follows that \( p(s_j[V], s_{j+1}[V]) = 1 \). By premise I2, we infer that \( p(s_{j+1}[V]) = 1 \), i.e., \( s_{j+1} \models p \).

By induction on \( j = 0, 1, \ldots \), we conclude that every \( s_j \) satisfies \( p \), i.e., \( p \) is a \( D \)-invariant.

Example: Program MUX-SEM

Consider the following parameterized program coordinating mutual exclusion by semaphores.

\[
y : \text{integer where } y = 1 \\
\begin{array}{c}
\ell_0 : \text{loop forever do} \\
\ell_1 : \text{Non-critical} \\
\ell_2 : \text{request } y \\
\ell_3 : \text{Critical} \\
\ell_4 : \text{release } y \\
\end{array}
\]

The semaphore instructions request \( y \) and release \( y \) respectively stand for

\( \langle \text{when } y > 0 \text{ do } y := y - 1 \rangle \) and \( y := y + 1 \).

We use rule BINV to verify the invariance of the assertion

\( p_1 : \quad y \geq 0 \)

This assertion is inductive so the proof succeeds.
For example, one of the instances of premise I2 is
\[
y \geq 0 \land \exists i : \pi[i] = 2 \land y > 0 \land y' = y - 1 \land \pi' = (\pi \text{ with } (i) := 3)
\]

Next, let us try to verify the property of mutual exclusion which can be specified as the invariance of the assertion
\[
p_2 : \neg (at_{\ell_3}[1] \land at_{\ell_3}[2])
\]
This attempt fails.

\[\text{Rule INV}\]

The above considerations lead to the more general INV rule.

\[
\begin{align*}
\text{Rule INV} \\
\text{For an assertion } \varphi, \\
\text{I1. } & \Theta \rightarrow \varphi \\
\text{I2. } & \varphi \land p \rightarrow \varphi' \\
\text{I3. } & \varphi \rightarrow p \\
\hline
\end{align*}
\]

By premises I1 and I2, \( \varphi \) is an invariant of the system. That is, all reachable states satisfy \( \varphi \). Since, by premise I3, \( \varphi \) implies \( p \), it follows that \( p \) is also a \( D \)-invariant.

For example, we can establish the invariance of
\[
p_2 : \neg (at_{\ell_3}[1] \land at_{\ell_3}[2])
\]
using rule INV with the strengthening
\[
\varphi : (y \geq 0) \land (at_{\ell_3,[1]} + at_{\ell_3,[2]} + \cdots + at_{\ell_3,[N]} + y = 1)
\]

Not Every Invariant Assertion is Inductive

As is already explained when one learns mathematical induction, there are valid assertions \( p \) which cannot be proven by induction, where the induction hypothesis is taken to be \( p \) itself.

For example, the claim

\[
The sum 1 + 3 + 5 + \cdots + (2k - 1) \text{ is a perfect square}
\]
or more mathematically

\[
p : \exists u : 1 + 3 + 5 + \cdots + (2k - 1) = u^2
\]
cannot be proven by induction, using \( p \) as the induction hypothesis.

To overcome this difficulty, one often has to come up with a strengthening of \( p \), being an assertion \( \varphi \) which implies \( p \) and is inductive. For the above example, this can be

\[
\varphi : 1 + 3 + 5 + \cdots + (2k - 1) = k^2
\]

Using TLV for Incremental Strengthening

The TLV tool, developed by Elad Shahar, is a programmable symbolic calculator over finite-state systems, based on the CMU symbolic model checker SMV.

It can be used to model check LTL formulas over finite-state systems. As we will show, it can also be used for incremental development of inductive assertions.

To do so, we define a finite-state restriction of the original program, explicitly calculate the candidate assertion, and apply rule BINV.

- If the rule application produces a counter-example, the assertion is not inductive. We should strengthen it, and repeat the procedure.
- If the rule application succeeds, there are good chances (but no guarantee) that the assertion is inductive. This it the time to shift to PVS in order to get the final confirmation.
The Input File \textit{mux3.smv}

```plaintext
MODULE main
DEFINE N:= 3;
VAR y : boolean ;
P : array 1..N of process MP(y);
Id: process Idle;
ASSIGN
  init(y) := 1;
MODULE Idle
MODULE MP(y)
VAR loc: 0..4;
ASSIGN
  init(loc) := 0;
  next(loc) := case
    loc in {0,1,3,4} : (loc + 1) mod 5;
    loc = 2 & y : 3;
    1 : loc;
  esac;
  next(y) := case
    loc = 2 & next(loc) = 3 : 0;
    loc = 4 & next(loc) = 0 : 1;
    esac;
JUSTICE loc != 0, loc != 3, loc != 4
COMPASSION (loc = 2 & y, loc = 3)
```

Model Checking Mutual Exclusion

In file \textit{scr1.pf}, we place the text

```
Print \texttt{
  Model Check mutual exclusion between P[1] and P[2]
};
```

```
mc ltl([]!(P[1].loc=3 & P[2].loc=3));
```

We then run

```
tlv mux3.smv
TLV version 3.1
... ...
Loaded rules file /home/amir/Tlv/Rules.tlv.
Your wish is my command ...
>> Load \texttt{"scr1.pf"};
```

Model Check mutual exclusion between P[1] and P[2]
Model checking...
Try deductive verification of mutual exclusion

Checking Premise I1

Premise I1 is valid. Checking Premise I2.

Premise I2 is not valid. Counter-example =

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1,0</td>
<td>0,0</td>
<td>2,3</td>
<td>3,3</td>
</tr>
</tbody>
</table>

**Strengthening the Assertion**

The offending transition captures a situation in which $P[3]$ is already at location $\ell_3$ and $P[2]$ has just joined it. Is such a situation possible in a real computation?

No! because in a real computation, if any process is at $\ell_3$ then $y$ must equal 0.

Consequently, we strengthen $\varphi_2$ into

$$\varphi_3 : \varphi_2 \land \forall i : at_\ell_3[i] \rightarrow y = 0$$
### Trying Second Approximation:

\[ \varphi_3 : \forall i : (\text{at} \_ \ell_3[i] \rightarrow y = 0) \land \forall j \neq i : \neg(\text{at} \_ \ell_3[i] \land \text{at} \_ \ell_3[j]) \]

In file `scr3.pf`, we place

```plaintext
... 
While (i)
    Let ass := ass & ((P[i].loc = 3) -> y=0);
    Let j := N;
    While (j)
        Let ass := ass & (i=j | P[i].loc != 3 | P[j].loc != 3);
        Let j := j - 1;
    End -- While (j)
End -- While(i) ... 
```

Running this script file, we obtain:

```
> Load "scr3.pf";
Try deductive verification of mutual exclusion
Checking Premise I1
```

### Strengthening \( \varphi_3 \)

The offending transition originates at a state in which \( P[2] \) is at location \( \ell_4 \) while \( P[3] \) is at location \( \ell_3 \). Such a state is unreachable, because the range for which mutual exclusion is ensured includes \( \ell_4 \) together with \( \ell_3 \).

Consequently, we strengthen \( \varphi_3 \) into

\[ \varphi_4 : \forall i : \text{at} \_ \ell_3[i] \rightarrow y = 0 \land \forall j \neq i : \neg(\text{at} \_ \ell_3[i] \land \text{at} \_ \ell_3[j]) \]

### Trying next Approximation:

\[ \varphi_4 : \forall i : \text{at} \_ \ell_3[i] \rightarrow y = 0 \land \forall j \neq i : \neg(\text{at} \_ \ell_4[i] \land \text{at} \_ \ell_3[j]) \]

In file `scr4.pf`, we replace

```plaintext
Let ass := ass & (i=j | P[i].loc != 3 | P[j].loc != 3);
```

as it appeared in `scr3.pf`, by:

```plaintext
Let ass := ass & (i=j | P[i].loc < 3 | P[j].loc < 3);
```

Running this version, we obtain

```
> Load "scr4.pf";
Try deductive verification of mutual exclusion
Checking Premise I1
```

Premise I1 is valid. Checking Premise I2.

Premise I2 is not valid. Counter-example =

\[ y = 0,1 \quad P[1].loc = 0,0 \quad P[2].loc = 4,0 \quad P[3].loc = 3,3 \]

The pre-state of this counter-example is unreachable because it has \( P[2] \) at location \( \ell_4 \) while \( y = 1 \). It is thus necessary to extend the range for which \( y = 0 \) to include also \( \ell_4 \). Consequently, we strengthen \( \varphi_4 \) into

\[ \varphi_5 : \forall i : \text{at} \_ \ell_4[i] \rightarrow y = 0 \land \forall j \neq i : \neg(\text{at} \_ \ell_4[i] \land \text{at} \_ \ell_3[j]) \]
Once More: Try

$$\varphi_5 : \forall i: \text{at}_{3,4}[i] \rightarrow y = 0 \land \forall j \neq i: \neg (\text{at}_{3,4}[i] \land \text{at}_{3,4}[j])$$

In file \texttt{scr5.pf}, we replace

Let \texttt{ass := ass & ((P[i].loc = 3) -> y=0)};

as it appeared in \texttt{scr3.pf}, by:

Let \texttt{ass := ass & ((P[i].loc > 2) -> y=0)};

Running this version, we obtain

... Try deductive verification of mutual exclusion
Checking Premise I1
Premise I1 is valid. Checking Premise I2.
Premise I2 is valid.
* * * Assertion \texttt{p} is invariant.