A. Pnueli

Fair Kripke Structures

An alternative computational model for finite-state reactive systems is that of Fair Kripke Structures (FKS). An FKS consists of:

\[ \langle \mathcal{L}, \mathcal{T}, \mathcal{H}, \mathcal{S} \rangle \]

where \( \mathcal{L} \) is a set of atomic propositions, \( \mathcal{T} \) a transition relation, \( \mathcal{H} \) a set of fairness sets, and \( \mathcal{S} \) a set of states. The relation \( \mathcal{H} \) is a subset of \( \mathcal{S} \times \mathcal{S} \), and the initial set of states is \( \mathcal{S}_0 \).

Given an FKS, a computation of \( \mathcal{M} \) is an infinite sequence of states \( s_0, s_1, \ldots \) satisfying:

- **Initiality** — \( s_0 \in \mathcal{S}_0 \).
- **Consecution** — For every \( j \geq 0 \), \( \mathcal{H}(s_j, s_{j+1}) = 1 \).
- **Fairness** — For every \( F \subseteq \mathcal{S} \), there exist infinitely many \( j \)'s.

The labels of each state are assumed to hold at that state.

\[ \begin{align*}
\forall p \in \mathcal{L} \left( s \in \mathcal{S} \right) & \quad s \models p \\
\forall s \in \mathcal{S} \left( s \times s \right) & \quad \mathcal{H}(s, s) \subseteq \mathcal{S}
\end{align*} \]
Example: A Mutual Exclusion Algorithm

Below, we present a simple algorithm for mutual exclusion MUX-SEM.

The semaphore instructions request and release respectively stand for

\begin{align*}
\text{request} & : y = 1 \\
\text{release} & : y = 0
\end{align*}

\textbf{Initial status:} boolean initially

\begin{align*}
\text{N} & = 1 \\
\text{C} & = 1
\end{align*}

\textbf{Example:} A Mutual Exclusion Algorithm

\textbf{When} 0 = y \text{ do } 1 = y
(N1 \lor \neg L \leftarrow \neg L, N2 \leftarrow \neg L, C1 \leftarrow \neg L, C2 \leftarrow \neg L)

The fairness sets can be characterized by the list

\{ \langle h, \neg L, N1, N2 \rangle \} = S0

The initial states consist of

S0 = \{ \langle h, N1, N2, L, L, C1, C2 \rangle \}

Following is an FKS for program MUX-SEM. The atomic propositions are

\( \mathcal{P} \)
Note that this translation can be applied only to FDS's with empty compass set.

\{ \mathcal{L} |= s \mid s \} \quad \text{the family contains the set, } \mathcal{L} \subseteq \mathcal{F}

For each \( s \subseteq \mathcal{F} \), that is, \( s \) is included by the propositions which are included in \( s \).

For each \( s \subseteq \mathcal{F} \), that is, \( s \) is a \( d \)-successor of \( s \).

That is we include in \( \mathcal{R} \) all pairs \( \langle s_1, s_2 \rangle \) such that \( s_2 \) is a \( d \)-successor of \( s_1 \).

\{ \Theta = s \mid S \subseteq s \} = \emptyset

\text{Interpretation of } \Lambda, \text{ where } \Lambda \cap \mathcal{A} = s \text{ is a subset of } S \subseteq \mathcal{F} \text{ is a state of } \Lambda. \text{ We can view } s \text{ as an } \Lambda \text{ state over the proposition set } \mathcal{A}, \text{ defined as follows:}

\langle \mathcal{F}, T' \mathcal{F}, \mathcal{R}, 0^\mathcal{F}, S \rangle = \mathcal{FDS}

Assume that all the variables in \( \Lambda \) are boolean. We construct an FKS as follows:

\langle \emptyset, \mathcal{L}, d, \emptyset, \Lambda \rangle = \emptyset \mathcal{FKS}

We show that the two computational models are essentially equivalent by showing

\( \text{FDS} \rightarrow \text{FKS} \)
\[\{ \mathcal{A} \in \mathcal{A} \mid (? = \nu) \land \} : \mathcal{L} \bullet \]

\[\left( (s)_{\mathcal{A}} \notin a \lor (s)_{\mathcal{A}} \in a \lor \_s = \_a \lor \_s = \_a \lor \_s = \nu \lor \_s = \nu \right) \land : d \bullet \]

\[\_0 \in \_s \land : \Theta \bullet \]

and an additional control variable \( ? \) ranging over \( \nu \cdot u \cdot v \cdot w \). Thus, the variables of consistent the propositions \( \mathcal{A} \). Therefore \( \{ u \cdot v \cdot w \} \cap dV = \Lambda \bullet \)

Construct a corresponding FDS as follows:

\[\langle \emptyset , \mathcal{L} , \mathcal{D} , \Theta , \Lambda \rangle = \mathcal{A} \]

\[\mathcal{S} = \langle S , T , S , S \rangle = W \]

FS ← FDS
Using TLV for Model Checking

We will introduce the programmable symbolic model checker TLV and illustrate its use for model checking finite-state systems.

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 temporal properties of finite-state systems.

A. Pnueli

Lecture 2: Other Models and Languages
Examples of Model Checking

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.

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

TRY-1

$$\begin{align*}
&\text{local } boolean \text{ where } 0 = \neg y_1 = \neg y_2 \\
&\text{loop forever do} \\
&\quad \text{critical } m_4 : y_1 = 0 \\
&\quad \text{critical } m_5 : y_1 = 1 \\
&\quad \text{wait } m_2 : y_2 = 1 \\
&\quad \text{critical } m_3 : y_2 = 0 \\
&\quad \text{non-critical } m_1 : \\
&\quad \text{critical } m_1 : \\
&\quad \text{non-critical } m_1 : \\
&\quad \text{loop forever do}
\end{align*}$$

TRY-2

$$\begin{align*}
&\text{local } boolean \text{ where } 0 = \neg y_1 = \neg y_2 \\
&\text{loop forever do} \\
&\quad \text{critical } j_4 : y_1 = 0 \\
&\quad \text{critical } j_5 : y_1 = 1 \\
&\quad \text{wait } j_2 : y_2 = 1 \\
&\quad \text{critical } j_3 : y_2 = 0 \\
&\quad \text{non-critical } j_1 : \\
&\quad \text{critical } j_1 : \\
&\quad \text{non-critical } j_1 : \\
&\quad \text{loop forever do}
\end{align*}$$

A. Pnueli

Examples of Other Models and Languages
Program Properties: Invariance

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

\[
\neg (at-l_4 \land at-m_4)
\]

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.

Invariance

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

\[
\neg (at-l_4 \land at-m_4)
\]
To check whether assertion exclusion is an invariant of program TRY-1, we invoke TLY. Invoking TLY

We will present each of these input files. It's invariance over the program.

Its invariance over the program. Invoking TLC to check whether assertion exclusion is a definition of the assertion exclusion command and a command to check printing commands' definition of the assertion exclusion

We prepare two input files: try1.smv, which contains the SMV representation of TRY-1, and try1.pf, a proof script file. The proof script file contains some printing commands, definition of the assertion exclusion and a command to check its invariance over the program.

Invoking TLY

A. Pnueli

Invoking TLY

In CMU by Ken McMillan and Ed Clarke.

In CMU by Ken McMillan and Ed Clarke.

The model checking tool TLY, a model checker based on the SMV tool developed by CMU, checks whether assertion exclusion is an invariant of program TRY-1.
File try1.smv

MODULE main
VAR y1: boolean; y2: boolean;

P[1]: process MP(y1, y2);
P[2]: process MP(y2, y1);

MODULE MP(mine, hers)
VAR loc: 0..5;

ASSIGN init(mine):=0; init(loc):=0;

next(loc) := case
            esac;

next (mine) := case
            esac;

next (loc) := case
            esac;

ASSIGN init (mine) := 0; init (loc) := 0;

VAR loc: 0..5;

MODULE MP (mine, hers)

P[1]: process MP(y1, y2);
P[2]: process MP(y2, y1);

VAR Y1: boolean!
Y2: boolean!

MODULE main

JUSTICE

loc != 0, !(loc = 2 & hers), loc != 3, loc != 4, loc != 5

Hardware Verification, NYU, Fall, 2004
A call to procedure \texttt{Invariance} invokes the process which checks whether any reachable state violates the assertion \texttt{exclusion}.

\begin{verbatim}
Call Invariance(exclusion); Let exclusion := !(p[1].loc=4 & p[2].loc=4);
Print "Check for Mutual Exclusion";
\end{verbatim}

Print "Check for Mutual Exclusion";

File try1.pf
A. Pnueli

Lecture 2: Other Models and Languages

TRY

The results of model-checking TRY-1 are

Results of Verifying TRY-1

Counter-example follows:

*** Property is NOT VALID

Model checking Invariance Property

>> Load "try1.pf"

>> Load "try1.pf"
Lecture 2: Other Models and Languages

Expressed in a More Readable Form

A.Pnueli

only later set their own.

Obviously, the problem is that the processes test each other's value first and

reaching the state which violates mutual exclusion!

The counter example is:

\[
\begin{align*}
\langle 0 : \text{wait} : I, \text{start} : I \rangle, \langle 0 : \text{wait} : I, \text{start} : I \rangle, \\
\langle 0 : \text{wait} : I, \text{start} : I \rangle, \langle 0 : \text{start} : I, \text{start} : I \rangle, \\
\langle 0 : \text{start} : I, \text{start} : I \rangle, \langle 0 : \text{wait} : I, \text{start} : I \rangle, \\
\langle 0 : \text{wait} : I, \text{start} : I \rangle
\end{align*}
\]
Let us see whether the program is now correct.

The following program, TRY-I, interchanges the order of testing and setting:

Second Attempt: Set first and Test Later.
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 successor different from itself. Mathematically, we can characterize all deadlock states by the assertion:

\[ (\not\exists \Lambda, \Lambda') : \Lambda \neq \Lambda' \]

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

```
Print "Check for Mutual Exclusion\n";
Let exclusion := !(P[1].loc=4 & P[2].loc=4);
Call Invariance(exclusion);
Run check_deadlock;
```

A state \( s \) is said to be a deadlock state if it has no successsor different from itself. Mathematically, we can characterize all deadlock states by the assertion:

\[ (\not\exists \Lambda, \Lambda') : \Lambda \neq \Lambda' \]

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

```
Print "Check for Mutual Exclusion\n";
Let exclusion := !(P[1].loc=4 & P[2].loc=4);
Call Invariance(exclusion);
Run check_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 successor different from itself. Mathematically, we can characterize all deadlock states by the assertion:

\[ (\not\exists \Lambda, \Lambda') : \Lambda \neq \Lambda' \]

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

```
Print "Check for Mutual Exclusion\n";
Let exclusion := !(P[1].loc=4 & P[2].loc=4);
Call Invariance(exclusion);
Run check_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 successor different from itself. Mathematically, we can characterize all deadlock states by the assertion:

\[ (\not\exists \Lambda, \Lambda') : \Lambda \neq \Lambda' \]
We obtain the following results:

```plaintext
>> Load "try2.pf"

Model Checking TRY-2

Counter-example follows:

Model checking Invariance Property
Check for the absence of deadlock.

*** Property is NOT VALID

State 7: y1 = 1, y2 = 1, P[1].loc = 1, P[2].loc = 1

State 6: y1 = 0, y2 = 0, P[1].loc = 0, P[2].loc = 0

State 5: y1 = 0, y2 = 0, P[1].loc = 0, P[2].loc = 0

State 4: y1 = 0, y2 = 0, P[1].loc = 0, P[2].loc = 0

State 3: y1 = 0, y2 = 0, P[1].loc = 0, P[2].loc = 0

State 2: y1 = 0, y2 = 0, P[1].loc = 0, P[2].loc = 0

State 1: y1 = 0, y2 = 0, P[1].loc = 0, P[2].loc = 0

P[1].loc = 0, P[2].loc = 0, y1 = 0, y2 = 0

P[1].loc = 1, P[2].loc = 1, y1 = 1, y2 = 1

P[1].loc = 2, P[2].loc = 2, y1 = 2, y2 = 2

P[1].loc = 3, P[2].loc = 3, y1 = 3, y2 = 3
```

```
A Pnueli
Lecture 2: Other Models and Languages
Hardware Verification, NYU, Fall 2004
```
reaching the deadlock state

The counter example is:

\[
\begin{align*}
\langle 1 : 0', 1 : 0', 1 : 0', 0 : 0', 0 : 0', 0 : 0', 0 : 0', 0 : 0', 0 : 0', 0 : 0' \rangle
\end{align*}
\]
Try a Different Approach

The following program uses a variable \( \text{turn} \) to indicate which process has the higher priority.

Hardware Verification, NYU, Fall, 2004
Program Properties: Response

This property refers to two assertions and \( \overline{d} \), it means

Every occurrence of a \( d \)-state must be followed by an occurrence of a \( \overline{d} \)-state

Example, the response property

For every occurrence of a \( b \)-state must be followed by an occurrence of a \( \overline{b} \)-state

To model check this property, we prepare the following file Try3.Pf:

```
Print"Check for Mutual Exclusion
"
Let exclusion := !(P[1].loc=3 & P[2].loc=3);
Call Invariance(exclusion);
Print "Check deadlock
"
Call Invariance(exclusion);
Run check deadlock;
```

Hardware Verification, NYU, Fall, 2004

A. Pnueli
We obtain the following results:

TRY

>> Load "try3.pf"

Check for Mutual Exclusion
*** Property is VALID

Check for the absence of deadlock.
*** Property is VALID

Counter-Example follows:
*** Property is NOT VALID

Check Accessibility for P1

Counter-Example follows:
*** Property is NOT VALID

Model Checking TRY-3
A. Pnueli

In a More Readable Form

The counterexample is:

\[
\begin{align*}
\langle 2 : \text{run} , \text{m0} , \text{run} \rangle & \quad \langle 2 : \text{run} , \text{m0} , \text{run} \rangle & \quad \langle 2 : \text{run} , \text{m0} , \text{run} \rangle \\
\langle 2 : \text{run} , \text{m0} , \text{run} \rangle & \quad \langle 1 : \text{run} , \text{m0} , \text{run} \rangle & \quad \langle 2 : \text{run} , \text{m0} , \text{run} \rangle \\
\langle 2 : \text{run} , \text{m0} , \text{run} \rangle & \quad \langle 2 : \text{run} , \text{m0} , \text{run} \rangle & \quad \langle 2 : \text{run} , \text{m0} , \text{run} \rangle \\
\langle 1 : \text{run} , \text{m0} , \text{run} \rangle & \quad \langle 1 : \text{run} , \text{m0} , \text{run} \rangle & \quad \langle 1 : \text{run} , \text{m0} , \text{run} \rangle \\
\end{align*}
\]

\[
\begin{align*}
\text{m4} & : \text{run} = 1 \\
\text{m3} & : \text{Critical} \\
\text{m2} & : \text{wait turn} = 2 \\
\text{m1} & : \text{Non-Critical} \\
\text{m0} & : \text{Loop forever do} \\
\end{align*}
\]
Finally a good program for mutual exclusion:

Peterson's for 2 Processes:

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

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. Model checking this program, we find that it satisfies the three properties of (invariance of) mutual exclusion, absence of deadlock, and accessibility.

Variables $s$, $s'$, $s''$ signify where processes are interested in entering their critical sections. Model checking this program, we find that it satisfies the three properties of (invariance of) mutual exclusion, absence of deadlock, and accessibility.

Hardware Verification, NYU, Fall, 2004