Keys in Formal Verification: Abstraction for Progress

Amir Pnueli

Weizmann Institute of Sciences and New York University

Marktoberdorf, August, 2004

Joint work with:

Yonit Kesten and Lenore Zuck

Lectures Outline

- Safety and progress. Fair discrete systems and operations on them.
- A general framework for abstraction for verification. Is the method complete?
- Counter Abstraction. Will skip – interested students can read it in the notes.
- Methods of invisible auxiliary constructs for safety and progress. Small model theorems.
- Small models for predicate abstraction – application to shape analysis.

Safety and Progress

Program properties are often partitioned into safety and progress (liveness) properties.

- Lamport suggested the following informal characterization of the two classes:

  A safety property states that something bad does not happen.

  A progress property states that something good will eventually happen.

These two classes complement one another. A specification that contains only one of the classes is usually incomplete.

Illustrate a Specification

The following program MUX-SEM, implements mutual exclusion by semaphores.

$$y: \textbf{natural initially } y = 1$$

$$P_1 :: \left[\begin{array}{l} \ell_0 : \quad \textbf{loop forever do} \\ \left[\begin{array}{ll} \ell_1 : & \textbf{Non-critical} \\ \ell_2 : & \textbf{request } y \\ \ell_3 : & \textbf{Critical} \\ \ell_4 : & \textbf{release } y \end{array}\right] \end{array}\right] \quad \| \quad P_2 :: \left[\begin{array}{l} m_0 : \quad \textbf{loop forever do} \\ \left[\begin{array}{ll} m_1 : & \textbf{Non-critical} \\ m_2 : & \textbf{request } y \\ m_3 : & \textbf{Critical} \\ m_4 : & \textbf{release } y \end{array}\right] \end{array}\right]$$

The semaphore instructions **request** y and **release** y respectively stand for

$$\langle \textbf{await } y > 0;\ y := 0 \rangle \quad \text{and} \quad y := 1.$$

The safety specification

$$\square\, \neg(at\_\ell_3 \ \wedge\ at\_m_3)$$

requires that the two processes never simultaneously execute their critical sections.

Is this an adequate (complete) specification?

A Faulty Implementation Satisfying mutual exclusion

The following program satisfies the safety requirement of mutual exclusion $\square\, \neg(at\_\ell_3 \;\wedge\; at\_m_3)$:

$$y: \textbf{ natural initially } y = 1$$

$$P_1 :: \left[\begin{array}{l} \ell_0: \ \textbf{loop forever do} \\ \left[\begin{array}{ll} \ell_1: & \textbf{Non-critical} \\ \ell_2: & \textbf{go to } \ell_1 \\ \ell_3: & \textbf{Critical} \end{array} \right] \end{array} \right] \quad \| \quad P_2 :: \left[\begin{array}{l} m_0: \ \textbf{loop forever do} \\ \left[\begin{array}{ll} m_1: & \textbf{Non-critical} \\ m_2: & \textbf{go to } m_1 \\ m_3: & \textbf{Critical} \end{array} \right] \end{array} \right]$$

In this implementation neither process ever visits its critical section.

A Faulty Implementation Satisfying mutual exclusion

The following program satisfies the safety requirement of mutual exclusion $\square\, \neg(at\_\ell_3 \;\wedge\; at\_m_3)$:

$$y: \textbf{natural initially } y = 1$$

$$P_1 :: \left[\begin{array}{ll} \ell_0: & \textbf{loop forever do} \\ & \left[\begin{array}{ll} \ell_1: & \textbf{Non-critical} \\ \ell_2: & \textbf{go to } \ell_1 \\ \ell_3: & \textbf{Critical} \end{array}\right] \end{array}\right] \quad \| \quad P_2 :: \left[\begin{array}{ll} m_0: & \textbf{loop forever do} \\ & \left[\begin{array}{ll} m_1: & \textbf{Non-critical} \\ m_2: & \textbf{go to } m_1 \\ m_3: & \textbf{Critical} \end{array}\right] \end{array}\right]$$

In this implementation neither process ever visits its critical section.

To prevent such unacceptable implementations, we need the following complete specification, which contains the additional progress requirements of accessibility:

$$\square\, \neg(at\_\ell_3 \;\wedge\; at\_m_3)$$
$$at\_\ell_2 \Rightarrow \diamond\, at\_\ell_3 \qquad at\_m_2 \Rightarrow \diamond\, at\_m_3$$

The property $at\_\ell_2 \Rightarrow \diamond\, at\_\ell_3$ requires that, after any state at which process P_1 is at location ℓ_2, there follows a state at which P_1 is critical.

Verifying Safety Properties

In order to verify the safety property $\square\, p$ over program P, we may use the following rule:

Rule INV
Find an assertion φ satisfying,

I1. All P-initial states satisfy φ
I2. Every P-successor of a φ-state satisfies φ
I3. $\varphi \rightarrow p$

$\square\, p$

An assertion φ satisfying premises I1 and I2 is called inductive. Every inductive assertion is invariant, i.e. holds in any reachable state of the system. Not every invariant assertion is inductive.

Example: Program MUX-SEM

Reconsider program MUX-SEM. We wish to prove

MUX-SEM $\models \square \neg(at\_\ell_3 \ \wedge \ at\_m_3)$.

y: **natural initially** $y = 1$

```
P_1 :: [ ℓ_0 : loop forever do
          [ ℓ_1 : Non-critical
            ℓ_2 : request y
            ℓ_3 : Critical
            ℓ_4 : release y        ] ]
   ||
P_2 :: [ m_0 : loop forever do
          [ m_1 : Non-critical
            m_2 : request y
            m_3 : Critical
            m_4 : release y        ] ]
```

We choose:

$\varphi : \quad at\_\ell_{3,4} + at\_m_{3,4} + y = 1$

Check premises:

I1. Initially, $at\_\ell_{3,4} = at\_m_{3,4} = 0$, $y = 1$, $0 + 0 + 1 = 1$.

I2. Induction step – Transitions ℓ_2, ℓ_4 preserve $at\_\ell_{3,4} + y$, transitions m_2, m_4 preserve $at\_m_{3,4} + y$. All other transitions preserve the full expression.

I3. Implication. $\varphi \rightarrow p$. Since $at\_\ell_3 = at\_m_3 = 1$ implies $y = -1$ violating y's type.

Verifying Progress Properties

Progress properties are verified using a combination of invariants and variants. These are ranking functions measuring distance from the goal.

For sequential programs, we can use a ranking function ranging over a well founded domain (such as the naturals), which decreases on every step of the program.

For example,

```
$y$: natural
$\ell_0$ :   while $y > 0$ do
               [ $\ell_1$ :   $y := y - 1$ ]
               [ $\ell_2$ :   skip         ]
$\ell_3$ :
```

We can take:

$$\varphi \quad : \quad \textit{at\_}\ell_1 \rightarrow y > 0$$

$$\delta \quad : \quad 2 \cdot \textit{at\_}\ell_2 + \textit{at\_}\ell_0 + 3 \cdot y$$

program	δ
ℓ_0 : while $y > 0$ do	$3 \cdot y + 1$
[ℓ_1 : $y := y - 1$]	$3 \cdot y$
[ℓ_2 : skip]	$3 \cdot y + 2$
ℓ_3 :	0

Concurrent Programs

When we consider concurrent programs, we can no longer rely on a ranking function which decreases at every step. Consider, for example, program UP-DOWN.

$x,\ y:$ **natural initially** $x = y = 0$

```
P1 :: [ ℓ0 : while x = 0 do
              [ℓ1 : y := y + 1]
        ℓ2 : while y > 0 do
              [ℓ3 : y := y − 1]
        ℓ4 :                      ]
   ||
P2 :: [ m0 : x := 1
        m1 :                      ]
```

When process P_2 is at location m_0, no number of steps of process P_1 will get us closer to termination.

Verifying Progress of Concurrent Programs

The approach to verification of progress for concurrent programs uses, in addition to a ranking functions, also an identification of helpful steps. We thus require:

- The ranking function decreases on every helpful step
- It never increases, even on unhelpful steps
- Until the goal is reached, a helpful step is always available, and some helpful step must eventually be taken

Illustrate on Program UP-DOWN

$x,\ y$: **natural initially** $x = y = 0$

$$P_1 :: \left[\begin{array}{l} \ell_0 : \quad \textbf{while } x = 0 \textbf{ do} \\ \qquad [\ell_1 :\ y := y + 1] \\ \ell_2 : \quad \textbf{while } y > 0 \textbf{ do} \\ \qquad [\ell_3 :\ y := y - 1] \\ \ell_4 : \end{array}\right] \quad \| \quad P_2 :: \left[\begin{array}{ll} m_0 : & x := 1 \\ m_1 : & \end{array}\right]$$

The following table specifies the conditions under which a step is helpful:

Step	When helpful
m_0	at_m_0
ℓ_0	$at\_\ell_0 \ \wedge\ x = 1$
ℓ_1	$at\_\ell_1 \ \wedge\ x = 1$
ℓ_2	$at\_\ell_2 \ \wedge\ x = 1$
ℓ_3	$at\_\ell_3 \ \wedge\ x = 1$

The ranking function ranges over lexicographic pairs, and is given by:

$$\delta : \qquad (at\_m_0, \quad at\_m_1 \cdot (2 \cdot y + 2 \cdot at\_\ell_0 + 5 \cdot at\_\ell_1 + at\_\ell_2))$$

Fair Discrete Systems

As our computational model, we take fair discrete systems. An FDS $\mathcal{D} = \langle V, \Theta, \rho, \mathcal{J}, \mathcal{C} \rangle$ consists of:

- V – A finite set of typed state variables. A V-state s is an interpretation of V. Denote by Σ_V – the set of all V-states.
- Θ – An initial condition. A satisfiable assertion that characterizes the initial states.
- ρ – A transition relation. An assertion $\rho(V, V')$, referring to both unprimed (current) and primed (next) versions of the state variables. For example, $x' = x + 1$ corresponds to the assignment $x := x + 1$.
- $\mathcal{J} = \{J_1, \ldots, J_k\}$ A set of justice (weak fairness) requirements. Ensure that a computation has infinitely many J_i-states for each J_i, $i = 1, \ldots, k$.
- $\mathcal{C} = \{\langle p_1, q_1 \rangle, \ldots \langle p_n, q_n \rangle\}$ A set of compassion (strong fairness) requirements. Infinitely many p_i-states imply infinitely many q_i-states.

A Simple Programming Language: SPL

A language allowing composition of parallel processes communicating by shared variables as well as message passing.

Example: Program ANY-Y

Consider the program

$$x,\ y: \textbf{natural initially } x = y = 0$$

$$\left[\begin{array}{ll} \ell_0: & \textbf{while } x = 0 \textbf{ do} \\ & [\ell_1: \ y := y + 1] \\ \ell_2: & \end{array}\right] \quad \| \quad \left[\begin{array}{ll} m_0: & x := 1 \\ m_1: & \end{array}\right]$$

$$-\ P_1\ - \qquad\qquad -\ P_2\ -$$

The Corresponding FDS

- State Variables $V: \left(\begin{array}{lcl} x, y & : & \text{natural} \\ \pi_1 & : & \{\ell_0, \ell_1, \ell_2\} \\ \pi_2 & : & \{m_0, m_1\} \end{array}\right)$.

- Initial condition: $\Theta: \ \pi_1 = \ell_0 \ \wedge \ \pi_2 = m_0 \ \wedge \ x = y = 0$.

- Transition Relation: ρ: $\rho_I \vee \rho_{\ell_0} \vee \rho_{\ell_1} \vee \rho_{m_0}$, with appropriate disjunct (transition) for each statement. For example, the disjuncts ρ_I and ρ_{ℓ_0} are

$$\rho_I: \quad \pi_1' = \pi_1 \ \wedge \ \pi_2' = \pi_2 \ \wedge \ x' = x \ \wedge \ y' = y$$

$$\rho_{\ell_0}: \quad \pi_1 = \ell_0 \ \wedge \left(\begin{array}{c} x = 0 \ \wedge \ \pi_1' = \ell_1 \\ \vee \\ x \neq 0 \ \wedge \ \pi_1' = \ell_2 \end{array}\right) \wedge \ \pi_2' = \pi_2 \ \wedge \ x' = x \ \wedge \ y' = y$$

- Justice set: $\mathcal{J}$: $\{\neg \textit{at\_}\ell_0, \neg \textit{at\_}\ell_1, \neg \textit{at\_}m_0\}$. Usually, we have a justice transition expressing the disableness of each just transition.

- Compassion set: $\mathcal{C}$: $\emptyset$.

Computations

Let $\mathcal{D}$ be an FDS for which the above components have been identified. The state s' is defined to be a $\mathcal{D}$-successor of state s if

$$\langle s, s' \rangle \models \rho_{\mathcal{D}}(V, V').$$

We define a computation of $\mathcal{D}$ to be an infinite sequence of states

$$\sigma : s_0, s_1, s_2, ...,$$

satisfying the following requirements:

- Initiality: s_0 is initial, i.e., $s_0 \models \Theta$.
- Consecution: For each $j \geq 0$, the state s_{j+1} is a $\mathcal{D}$-successor of the state s_j.
- Justice: For each $J \in \mathcal{J}$, σ contains infinitely many J-positions. This guarantees that every just transition is disabled infinitely many times.
- Compassion: For each $\langle p, q \rangle \in \mathcal{C}$, if σ contains infinitely many p-positions, it must also contain infinitely many q-positions. This guarantees that every compassionate transition which is enabled infinitely many times is also taken infinitely many times.

Justice is not Enough. You also Need Compassion

The following program MUX-SEM, implements mutual exclusion by semaphores.

$$y: \textbf{ natural initially } y = 1$$

```
P_1 :: [ ℓ_0 : loop forever do              ]   ||   P_2 :: [ m_0 : loop forever do              ]
       [      [ ℓ_1 : Non-critical   ]      ]               [      [ m_1 : Non-critical   ]      ]
       [      [ ℓ_2 : request y      ]      ]               [      [ m_2 : request y      ]      ]
       [      [ ℓ_3 : Critical       ]      ]               [      [ m_3 : Critical       ]      ]
       [      [ ℓ_4 : release y      ]      ]               [      [ m_4 : release y      ]      ]
```

The compassion set of this program consists of

$$\mathcal{C}: \quad \{(at\_\ell_2 \wedge y > 0, at\_\ell_3), \quad (at\_m_2 \wedge y > 0, at\_m_3)\}.$$

Usually, with a compassionate transition τ, we associate the compassion requirement

$$(En(\tau), \quad taken(\tau))$$

FDS Operations: Asynchronous Parallel Composition

The asynchronous parallel composition of systems $\mathcal{D}_1$ and $\mathcal{D}_2$, denoted by $\mathcal{D}_1 \| \mathcal{D}_2$, is given by $\mathcal{D} = \langle V, \Theta, \rho, \mathcal{J}, \mathcal{C} \rangle$, where

$$\begin{array}{lclcl}
V & = & V_1 & \cup & V_2 \\
\Theta & = & \Theta_1 & \wedge & \Theta_2 \\
\rho & = & (\rho_1 \wedge \mathit{pres}(V_2 - V_1)) & \vee & (\rho_2 \wedge \mathit{pres}(V_1 - V_2)) \\
\mathcal{J} & = & \mathcal{J}_1 & \cup & \mathcal{J}_2 \\
\mathcal{C} & = & \mathcal{C}_1 & \cup & \mathcal{C}_2
\end{array}$$

The predicate $\mathit{pres}(U)$ stands for the assertion $U' = U$, implying that all the variables in U are preserved by the transition.

Asynchronous parallel composition represents the interleaving-based concurrency which is assumed in shared-variables models.

Claim 1. $\mathcal{D}(P_1 \| P_2) \quad \sim \quad \mathcal{D}(P_1) \| \mathcal{D}(P_2)$

Synchronous Parallel Composition

The synchronous parallel composition of systems $\mathcal{D}_1$ and $\mathcal{D}_2$, denoted by $\mathcal{D}_1 \parallel\!\mid \mathcal{D}_2$, is given by the FDS $\mathcal{D} = \langle V, \Theta, \rho, \mathcal{J}, \mathcal{C} \rangle$, where

$$\begin{array}{lllll}
V & = & V_1 & \cup & V_2 \\
\Theta & = & \Theta_1 & \wedge & \Theta_2 \\
\rho & = & \rho_1 & \wedge & \rho_2 \\
\mathcal{J} & = & \mathcal{J}_1 & \cup & \mathcal{J}_2 \\
\mathcal{C} & = & \mathcal{C}_1 & \cup & \mathcal{C}_2
\end{array}$$

Synchronous parallel composition is used for the construction of an observer: a system O which observes and evaluates the behavior of an observed system $\mathcal{D}$. Running $\mathcal{D} \parallel\!\mid O$, we let $\mathcal{D}$ behave as usual, while O observes its behavior.

Verification-Driven Abstraction

We will consider a general approach to abstraction of systems in order to simplify their verification.

Lessons to be learned from today's lecture:

- It is possible to do abstraction without lattices, Galois connection, Moore families, fixpoints, and 3-valued Logic.

  Abstract interpretation for dummies (Model Checkers)

- When abstracting the verification problem $P \models \varphi$, it is necessary to over-abstract P and under-abstract φ.

- For verification of progress, state abstraction is inadequate. You also need to abstract transitions.

Approaches to Formal Verification

The Common Wisdom:

To verify a reactive system S,

- If it is finite state, model check it.
- Otherwise, prove it by temporal deduction, using a temporal deductive system such as [MP] or TLA, supported by theorem provers, such as STeP, TLP, or PVS.

Often, both approaches to verification can be simplified by using abstraction.

AAV: Abstraction Aided Verification

An Obvious idea:

- Abstract system S into S_A – a simpler system, but admitting more behaviors.
- Verify property for the abstracted system S_A.
- Conclude that property holds for the concrete system.

Approach is particularly impressive when abstracting an infinite-state system into a finite-state one.

Technically, Define the methodology of Verification by Finitary Abstraction (VFA) as follows:

To prove $\mathcal{D} \models \psi$,

- Abstract $\mathcal{D}$ into a finite-state system $\mathcal{D}^\alpha$ and the specification ψ into a propositional LTL formula ψ^α.
- Model check $\mathcal{D}^\alpha \models \psi^\alpha$.

The question considered here is whether we can find instantiations of this general methodology which are sound and (relatively) complete.

Finitary Abstraction

Based on the notion of abstract interpretation [CC77].

Let Σ denote the set of states of an FDS $\mathcal{D}$ – the concrete states. Let $\alpha : \Sigma \mapsto \Sigma_A$ be a mapping of concrete into abstract states. α is finitary if Σ_A is finite.

We consider abstraction mappings which are presented by a set of equations $\alpha : (u_1 = E_1(V), \ldots, u_n = E_n(V))$ (or more compactly, $V_A = \mathcal{E}_\alpha(V)$), where $V_A = \{u_1, \ldots, u_n\}$ are the abstract state variables and V are the concrete variables.

Example: Program ANY-Y

Consider the program

$$x,\ y : \textbf{integer initially } x = y = 0$$

$$P_1 :: \left[\begin{array}{l} \ell_0 : \quad \textbf{while } x = 0 \textbf{ do} \\ \qquad [\ell_1 : \ y := y + 1] \\ \ell_2 : \end{array}\right] \quad \| \quad P_2 :: \left[\begin{array}{ll} m_0 : & x := 1 \\ m_1 : & \end{array}\right]$$

Assume we wish to verify the property $\square\ (y \geq 0)$ for system ANY-Y.

We introduce two abstract variables:

$$X : \textbf{boolean}, \quad Y : \{-1, 0, +1\}$$

The abstraction mapping α is specified by the following list of defining expressions:

$$\alpha : \quad [X = (x \neq 0), \quad Y = \mathit{sign}(y)]$$

where $\mathit{sign}(y)$ is defined to be -1, 0, or 1, according to whether y is negative, zero, or positive, respectively.

The Abstracted Version

With the mapping α, we can obtain the abstract version of ANY-Y, called ANY-Y$^\alpha$:

$$\begin{array}{lll} X: & \textbf{boolean} & \textbf{initially } X = 0 \\ Y: & \{\mathit{neg}, \mathit{zero}, \mathit{pos}\} & \textbf{initially } Y = \mathit{zero} \end{array}$$

$$P_1 :: \left[\begin{array}{l} \ell_0 : \quad \textbf{while } X = 0 \textbf{ do} \\ \qquad [\ell_1 : \; Y := \left(\begin{array}{ll} \textbf{if} & Y = \mathit{neg} \\ \textbf{then} & \{\mathit{neg}, \mathit{zero}\} \\ \textbf{else} & \mathit{pos} \end{array}\right)] \\ \ell_2 : \end{array}\right] \quad \| \quad P_2 :: \left[\begin{array}{ll} m_0 : & X := 1 \\ m_1 : & \end{array}\right]$$

The original invariance property ψ: $\square\,(y \geq 0)$, is abstracted into:

$$\psi^\alpha: \quad \square\,(Y \in \{\mathit{zero}, \mathit{pos}\}),$$

which can be model-checked over ANY-Y$^\alpha$.

When is Such an Abstraction Sound?

Reconsider program ANY-Y, but this time with the property

$$\square\ (0 \leq y \leq 10)$$

Abstracting this property in a way consistent with the abstraction function $Y = \mathit{sign}(y)$, we obtain the same abstraction as before, namely,

$$\square\ (Y \in \{\mathit{zero}, \mathit{pos}\})$$

which we know to be valid over the abstracted version of ANY-Y.

Can we conclude

$$\text{ANY-Y} \quad \models \quad \square\ (0 \leq y \leq 10)?$$

Obviously not!!!

What went wrong?

Lifting a State Abstraction to Assertions

For an abstraction mapping $\alpha : V_A = \mathcal{E}_\alpha(V)$ and an assertion $p(V)$, there are two ways we can abstract p:

- The expanding α-abstraction (over approximation) of p is given by

$$\overline{\alpha}(p): \quad \exists V\ (V_A = \mathcal{E}_\alpha(V)\ \wedge\ p(V)) \qquad \|\overline{\alpha}(p)\| = \{\alpha(s) \mid s \in \|p\|\}$$

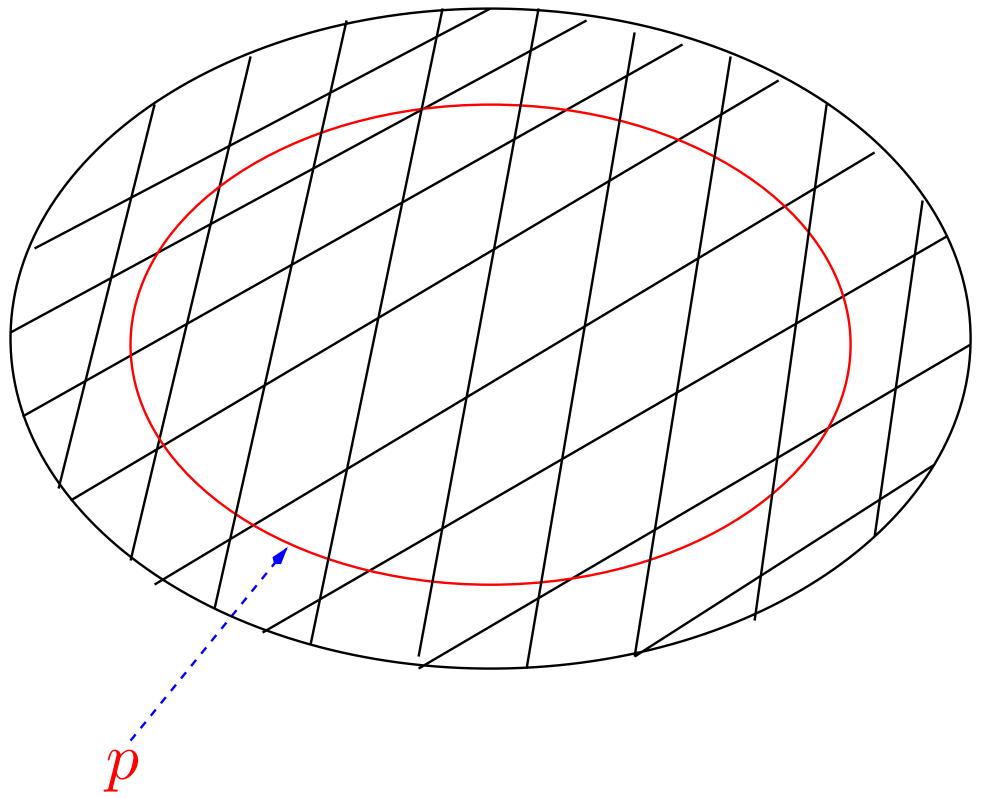
  An abstract state S belongs to $\|\overline{\alpha}(p)\|$ iff there exists some concrete state $s \in \alpha^{-1}(S)$ such that $s \in \|p\|$.

- The contracting abstraction (under approximation) is given by

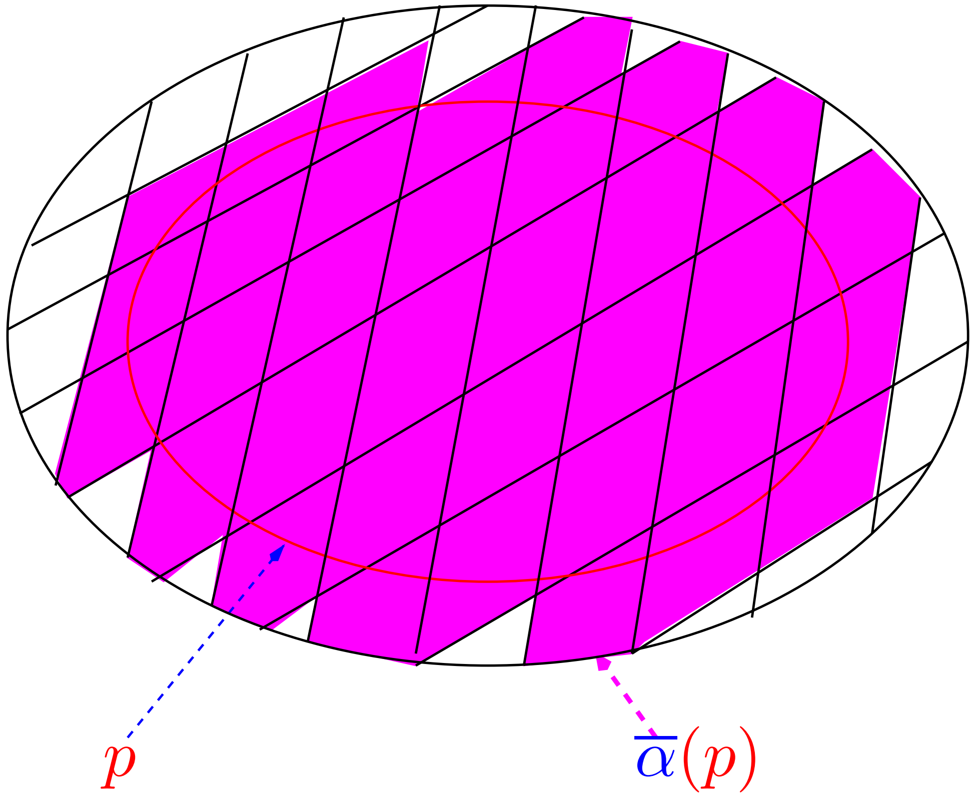
$$\begin{array}{lcl} \underline{\alpha}(p): & \exists V\ (V_A = \mathcal{E}_\alpha(V)) & \wedge \quad \forall V\ (V_A = \mathcal{E}_\alpha(V)\ \rightarrow\ p(V)) \\ \|\underline{\alpha}(p)\| & = & \{S \mid \alpha^{-1}(S) \subseteq \|p\|\} \end{array}$$

  An abstract state S belongs to $\|\underline{\alpha}(p)\|$ iff all concrete states $s \in \alpha^{-1}(S)$ satisfy $s \in \|p\|$.

Visual Illustration of the Two Abstraction Transformers

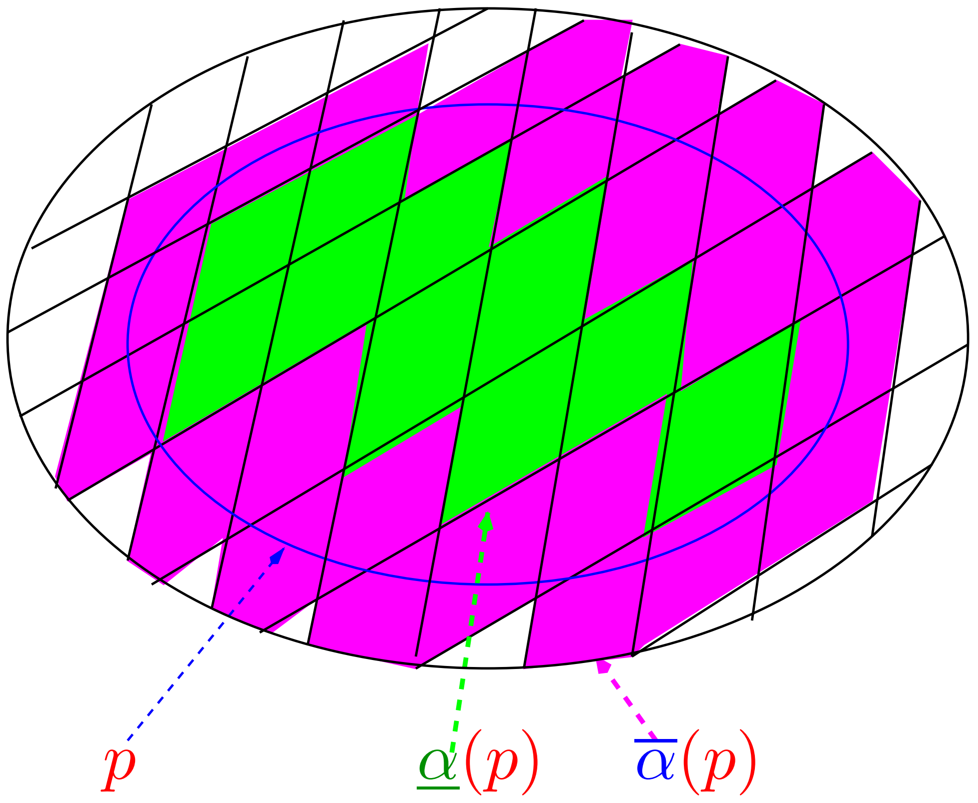


The Existential (expanding) Abstraction



Abstract state S belongs to $\overline{\alpha}(p)$ if some concrete state α-mapped into S satisfies p.

The Universal (contracting) Abstraction



Abstract state S belongs to $\underline{\alpha}(p)$ if all concrete states α-mapped into S satisfy p.

In many cases, the abstraction α is precise with respect to the assertion p. This is when p does not distinguish between two concrete states which are mapped by α to the same abstract state. In such cases

$$\overline{\alpha}(p) \quad = \quad \underline{\alpha}(p)$$

Sound Abstraction of Set Inclusions

Claim 2. *Let T_1 and T_2 be two sets of concrete states. The correct sound abstraction of the inclusion problem $T_1 \subseteq T_2$ is*

$$\overline{\alpha}(T_1) \quad \subseteq \quad \underline{\alpha}(T_2)$$

That is, the claim states that $\overline{\alpha}(T_1) \subseteq \underline{\alpha}(T_2)$ is a sufficient condition for the inclusion $T_1 \subseteq T_2$.

Proof: Assume that $\overline{\alpha}(T_1) \subseteq \underline{\alpha}(T_2)$ and let $s \in T_1$. It follows that the abstract state $S = \alpha(s)$ is a member of $\overline{\alpha}(T_1)$. Since $\overline{\alpha}(T_1) \subseteq \underline{\alpha}(T_2)$, it follows that $S \in \underline{\alpha}(T_2)$. According to the definition of $\underline{\alpha}(T_2)$, $S \in \underline{\alpha}(T_2)$ implies that all α-sources of S, in particular s, belong to T_2. We conclude that $T_1 \subseteq T_2$. ▟

Sound Joint Abstraction

For a temporal formula ψ, we define ψ^α to be the formula obtained by replacing every (maximal) state sub-formula $p \in \psi$ by $\underline{\alpha}(p)$.

For an FDS $\mathcal{D} = \langle V, \Theta, \rho, \mathcal{J}, \mathcal{C} \rangle$, we define the α-abstracted version $\mathcal{D}^\alpha = \langle V_A, \Theta^\alpha, \rho^\alpha, \mathcal{J}^\alpha, \mathcal{C}^\alpha \rangle$, where

$$\begin{array}{lcl}
\Theta^\alpha & = & \overline{\alpha}(\Theta) \\
\rho^\alpha & = & \overline{\overline{\alpha}}(\rho) \\
\mathcal{J}^\alpha & = & \{\overline{\alpha}(J) \mid J \in \mathcal{J}\} \\
\mathcal{C}^\alpha & = & \{(\underline{\alpha}(p), \overline{\alpha}(q)) \mid (p,q) \in \mathcal{C}\}
\end{array}$$

Soundness:

If α is an abstraction mapping and $\mathcal{D}$ and ψ are abstracted according to the recipes presented above, then

$$\mathcal{D}^\alpha \models \psi^\alpha \qquad \text{implies} \qquad \mathcal{D} \models \psi.$$

Computing the Abstract ρ

Technically,

$$\overline{\overline{\alpha}}(\rho) : \qquad \exists V, V' : (V_A = \mathcal{E}_\alpha(V) \ \wedge \ V'_A = \mathcal{E}_\alpha(V') \ \wedge \ \rho(V, V'))$$

For example,

$$\overline{\overline{\alpha}}(y' = y+1)(Y, Y') = \exists y, y' : (Y = \textit{sign}(y) \ \wedge \ Y' = \textit{sign}(y') \ \wedge \ y' = y+1)$$

In many cases, it is possible to break the computation of $\overline{\overline{\alpha}}(\rho)$ into a set of decision problems, such as:

$$\begin{array}{lcl}
\overline{\overline{\alpha}}(\rho)(-1,-1) = 1 & \Longleftrightarrow & y < 0 \ \wedge \ y' < 0 \ \wedge \ y' = y+1 \text{ is satisfiable} \\
\overline{\overline{\alpha}}(\rho)(-1,\ \ 0) = 1 & \Longleftrightarrow & y < 0 \ \wedge \ y' = 0 \ \wedge \ y' = y+1 \text{ is satisfiable} \\
\overline{\overline{\alpha}}(\rho)(-1,+1) = 1 & \Longleftrightarrow & y < 0 \ \wedge \ y' > 0 \ \wedge \ y' = y+1 \text{ is satisfiable} \\
 & \cdots &
\end{array}$$

This enumeration yields the following abstraction:

$$\overline{\overline{\alpha}}(y' = y+1) : \qquad Y = -1 \ \wedge \ Y' \in \{-1, 0\} \quad \vee \quad Y \in \{0, 1\} \ \wedge \ Y' = 1$$

Rectifying the False Counter-Example

Previously, we considered the problem

$$\text{ANY-Y} \quad \models \quad \square\,(0 \leq y \leq 10)$$

The correct abstraction of this problem is

$$\text{ANY-Y}^{\alpha} \quad \models \quad \square\,(Y = 0)$$

The assertion $Y = 0$ is not an invariant of the abstract program ANY-Y$^{\alpha}$. Therefore we cannot (falsely) conclude that $0 \leq y \leq 10$ is an invariants of ANY-Y.

Proving Soundness of the Method

Let α be an abstraction mapping over the variables of an FDS $\mathcal{D}$. For $\sigma : s_0, s_1, \ldots$ a computation of $\mathcal{D}$, we denote by σ^α the sequence of abstract states $\sigma^\alpha : \alpha(s_0), \alpha(s_1), \ldots$. The proof of soundness if based on the following two claims:

Claim 3. *If σ is a computation of $\mathcal{D}$, then σ^α is a computation of $\mathcal{D}^\alpha$.*

and

Claim 4. *Let σ be a state sequence and ψ a positive form temporal formula. If $\sigma^\alpha \models \psi^\alpha$ then $\sigma \models \psi$.*

Predicate Abstraction

The mapping α is called a predicate abstraction if it contains the boolean equation $B_p = p$ for each atomic state formula p occurring in ψ, Θ, $\mathcal{J}$, and C.

Let $p_1, p_2, \ldots, p_k$ be the set of all atomic formulas referring to the data (non-control) variables appearing within conditions in the program P and within the temporal formula ψ.

Following [GS97], define abstract boolean variables $B_{p_1}, B_{p_2}, \ldots, B_{p_k}$, one for each atomic data formula. The abstraction mapping α is defined by

$$\alpha: \quad \{B_{p_1} = p_1, B_{p_2} = p_2, \ldots, B_{p_k} = p_k\}$$

Example: Program BAKERY-2

```
local  $y_1, y_2$ : natural initially $y_1 = y_2 = 0$

$P_1$ ::  $\ell_0$ :loop forever do
              $\ell_1$ :  Non-Critical
              $\ell_2$ :  $y_1 := y_2 + 1$
              $\ell_3$ :  await $y_2 = 0 \lor y_1 < y_2$
              $\ell_4$ :  Critical
              $\ell_5$ :  $y_1 := 0$

    $\|$

$P_2$ ::  $m_0$ :loop forever do
              $m_1$ :  Non-Critical
              $m_2$ :  $y_2 := y_1 + 1$
              $m_3$ :  await $y_1 = 0 \lor y_2 \leq y_1$
              $m_4$ :  Critical
              $m_5$ :  $y_2 := 0$
```

The temporal properties for program BAKERY-2 are

ψexc : $\square \neg(at\_\ell_4 \land at\_m_4)$

ψacc : $\square \; (at\_\ell_2 \rightarrow \diamond \; at\_\ell_4)$,

Abstracting Program BAKERY-2

Define abstract variables $B_{y_1=0}$, $B_{y_2=0}$, and $B_{y_1<y_2}$.

```
local $B_{y_1=0}, B_{y_2=0}, B_{y_1<y_2}$: boolean
      where $B_{y_1=0} = B_{y_2=0} = 1, B_{y_1<y_2} = 0$

$P_1$ ::  $\ell_0$ :loop forever do
              $\ell_1$ :  Non-Critical
              $\ell_2$ :  $(B_{y_1=0}, B_{y_1<y_2}) := (0,0)$
              $\ell_3$ :  await $B_{y_2=0} \lor B_{y_1<y_2}$
              $\ell_4$ :  Critical
              $\ell_5$ :  $(B_{y_1=0}, B_{y_1<y_2}) := (1, \neg B_{y_2=0})$

$\|$

$P_2$ ::  $m_0$ :loop forever do
              $m_1$ :  Non-Critical
              $m_2$ :  $(B_{y_2=0}, B_{y_1<y_2}) := (0,1)$
              $m_3$ :  await $B_{y_1=0} \lor \neg B_{y_1<y_2}$
              $m_4$ :  Critical
              $m_5$ :  $(B_{y_2=0}, B_{y_1<y_2}) := (1,0)$
```

The abstracted properties can now be model-checked.

The Question of Completeness

We have shown above that the AAV method is sound. How about completeness?

Completeness means that for every FDS $\mathcal{D}$ and temporal property ψ such that $\mathcal{D} \models \psi$, there exists a finitary abstraction mapping α such that $\mathcal{D}^\alpha \models \psi^\alpha$.

At this point we can only claim completeness for the special case that ψ is an invariance property.

Claim 5. [Completeness for Invariance Properties]
Let $\mathcal{D}$ be an FDS and $\psi : \square\, p$ be an invariance property such that $\mathcal{D} \models \square\, p$. Then there exists a finitary abstraction mapping α such that $\mathcal{D}^\alpha \models \square\, \underline{\alpha}(p)$.

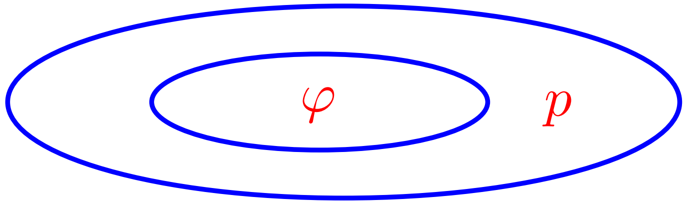
Sketch of the Proof

Like many completeness proofs in logic, the proof of this theorem is simple but not very useful.

Let $\mathcal{D} = \langle V, \Theta, \rho, \ldots \rangle$ be an FDS and p be an assertion such that $\mathcal{D} \models \Box\, p$. We will show that there exists a finitary abstraction α which transforms the verification problem $\mathcal{D} \models \Box\, p$ into a simple finite-state problem.

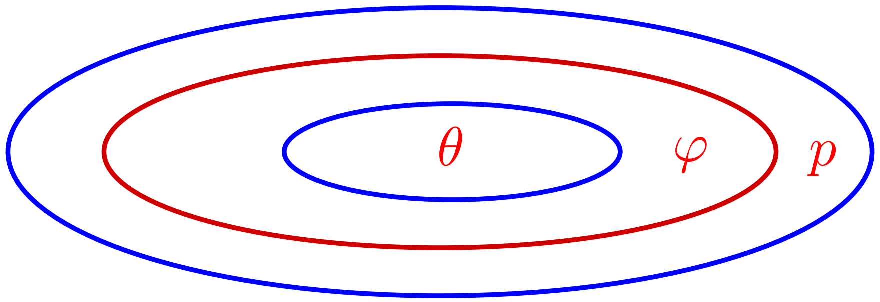
By the theory of temporal verification, $\mathcal{D} \models \Box\, p$ implies the existence of an assertion φ satisfying the following 3 premises:

$$\begin{array}{ccc} \Theta & \rightarrow & \varphi \\ \varphi \wedge \rho & \rightarrow & \varphi' \\ \varphi & \rightarrow & p \end{array}$$

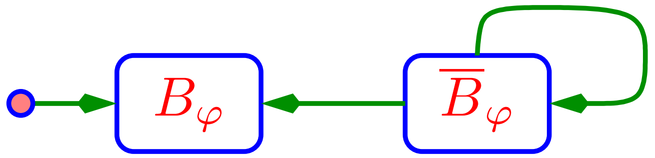


As the abstraction mapping, we take $\alpha : B_\varphi = \varphi$ using a single abstract boolean variable B_φ which is true whenever the corresponding concrete state satisfies φ. This leads to the following abstractions:

Proof Continued



The abstractions:

System $\mathcal{D}^\alpha$	Property ψ^α
$V: \quad B_\varphi$: **boolean** $\overline{\alpha}(\Theta): \quad B_\varphi$ $\overline{\overline{\alpha}}(\rho): \quad B_\varphi \rightarrow B'_\varphi \ \wedge \ \cdots$ 	$\square \ \underline{\alpha}(p) = \square \ B_\varphi$

The only computation of $\mathcal{D}^\alpha$ is $\sigma^\alpha : B_\varphi, B_\varphi, \ldots$. It follows that $\mathcal{D}^\alpha \models \psi^\alpha$.

Abstracting for Progress

Today's Lessons:

- It is not enough to abstract states. To verify progress, one must also abstract transitions.
- Compassion provides a useful abstraction to well-foundedness.
- Predicate abstraction should be augmented by ranking function abstraction.

Inadequacy of State Abstraction

Not all properties can be proven by pure finitary state abstraction.

Consider the program LOOP.

```
            $y$:  natural
$\ell_0$ :   while $y > 0$ do
             [ $\ell_1$ :   $y := y - 1$ ]
             [ $\ell_2$ :   skip        ]
$\ell_3$ :
```

Termination of this program cannot be proven by pure finitary abstraction. For example, the abstraction $\alpha : \mathbb{N} \mapsto \{zero, pos\}$ leads to the abstracted program

```
            $Y$:  $\{zero, pos\}$
$\ell_0$ :   while $Y = pos$ do
             [ $\ell_1$ :   $Y := sub1(Y)$ ]
             [ $\ell_2$ :   skip          ]
$\ell_3$ :
```

where

$$sub1(Y) \quad = \quad \textbf{if } Y = pos \textbf{ then } \{zero, pos\} \textbf{ else } zero$$

This abstracted program may diverge!

Solution: Augmentation with a Non-Constraining Progress Monitor

y: **natural**

$$\left[\begin{array}{l} \ell_0 : \textbf{while } y > 0 \textbf{ do} \\ \qquad \left[\begin{array}{l} \ell_1 : y := y - 1 \\ \ell_2 : \textbf{skip} \end{array}\right] \\ \ell_3 : \end{array}\right] \quad ||| \quad \left[\begin{array}{l} \quad \mathit{inc} \quad : \{-1, 0, 1\} \\ \quad \textbf{compassion} \\ \qquad (\mathit{inc} < 0, \mathit{inc} > 0) \\ \textbf{always do} \\ m_0 : \ \mathit{inc} := \mathit{sign}(y' - y) \end{array}\right]$$

— LOOP — — MONITOR M_y —

Forming the cross product, we obtain:

$$\boxed{\begin{array}{l} \qquad\qquad y \quad : \textbf{natural} \\ \qquad\qquad \mathit{inc} \ : \{-1, 0, 1\} \\ \qquad\qquad \textbf{compassion } (\mathit{inc} < 0, \mathit{inc} > 0) \\ \ell_0 : \ \textbf{while } y > 0 \textbf{ do} \\ \qquad \left[\begin{array}{llll} \ell_1 : & (y, \mathit{inc}) & := & (y - 1, \mathit{sign}(y' - y)) \\ \ell_2 : & \mathit{inc} & := & \mathit{sign}(y' - y) \end{array}\right] \\ \ell_3 : \end{array}}$$

Abstracting the Augmented System

We obtain the program

```
                 $Y$           : $\{zero, pos\}$
                 $inc$         : $\{-1, 0, 1\}$
                 compassion  $(inc < 0, inc > 0)$

$\ell_0$ :  while $Y = pos$ do
      [ $\ell_1$ : $(Y, inc)$ := ( if    $Y = pos$                  ) ]
      [                           then  $(\{pos, zero\}, -1)$      ) ]
      [                           else  $(zero, 0)$                ) ]
      [                                                              ]
      [ $\ell_2$ :       $inc$ := $0$                              ]
$\ell_3$ :
```

Which always terminates.

A More Complicated Case

Sometimes we need a more complex progress measure:

```
              $y$: natural

$\ell_0$ :   while $y > 1$ do
                 [ $\ell_1$ :   $y := y - 2$          ]
                 [ $\ell_2$ :   $y := \{y + 1, y\}$   ]
                 [ $\ell_3$ :   skip                  ]
$\ell_4$ :
```

To prove termination of this program we augment it by the monitor:

```
     define          $\delta = y + at\_\ell_2$
     inc             $: \{-1, 0, 1\}$
     compassion      $(inc < 0, inc > 0)$

$m_0$ :   always do
               $inc := sign(\delta' - \delta)$
```

Complicated Case Continued

Augmenting and abstracting, we get:

$$\begin{array}{ll} Y & : \{\textit{zero}, \textit{one}, \textit{large}\} \\ \textit{inc} & : \{-1, 0, 1\} \\ \textbf{compassion} & (\textit{inc} < 0, inc > 0) \end{array}$$

$$\begin{array}{l} \ell_0 : \quad \textbf{while } Y = \textit{large} \textbf{ do} \\ \qquad \left[\begin{array}{lrcl} \ell_1 : & (Y, \textit{inc}) & := & (\textit{sub2}(Y),\ -1) \\ \ell_2 : & (Y, \textit{inc}) & := & \{(\textit{add1}(Y), 0),\ (Y, -1)\} \\ \ell_3 : & \textit{inc} & := & 0 \end{array}\right] \\ \ell_4 : \end{array}$$

where,

$$\begin{array}{lcl} \textit{sub2}(Y) & = & \textbf{if } Y \in \{\textit{zero}, \textit{one}\} \textbf{ then } \textit{zero} \textbf{ else } \{\textit{zero}, \textit{one}, \textit{large}\} \\ \textit{add1}(Y) & = & \textbf{if } Y = \textit{zero} \textbf{ then } \textit{one} \textbf{ else } \textit{large} \end{array}$$

This program always terminates

Verification by Augmented Finitary Abstraction - The AFA Method

To verify that ψ is $\mathcal{D}$-valid,

- Optionally choose a non-constraining progress monitor FDS M and let $\mathcal{A} = \mathcal{D} \ ||| \ M$. In case this step is skipped, we let $\mathcal{A} = \mathcal{D}$.
- Choose a finitary state abstraction mapping α and calculate $\mathcal{A}^\alpha$ and ψ^α according to the sound recipes.
- Model check $\mathcal{A}^\alpha \models \psi^a$.
- Infer $\mathcal{D} \models \psi$.

Claim 6. *The* AFA *method is complete, relative to deductive verification.*

That is, whenever there exists a deductive proof of $\mathcal{D} \models \psi$, we can find a finitary abstraction mapping α and a non-constraining progress monitor M, such that $\mathcal{A}^\alpha \models \psi^a$.

Some Comments

One possible interpretation of our results is

It is not sufficient to abstract states. One should also abstract transitions.

Our completeness results can also be related to the work of Abadi and Lamport [AL91] who claim that abstraction can be made complete by the addition of history and prophecy variables. In our case, the history is represented by the reference to both y and y'. The prophecy is represented by the compassion requirement $(\mathit{inc} < 0, \mathit{inc} > 0)$.

How Practical is This Approach?

The advantage of this approach over conventional deductive verification is that one may throw in as many elementary ranking functions as she wishes, and let the model checker sort out their interaction and relevance.

Consider the following program NESTED-LOOPS:

```
                  $x, y$: natural
$\ell_0$ :   $x :=?$
$\ell_1$ :   while $x > 0$ do
          [ $\ell_2$ :   $y :=?$
            $\ell_3$ :   while $y > 0$ do
                      [ $\ell_4$ :   $y := y - 1$ ]
                      [ $\ell_5$ :   skip         ]
            $\ell_6$ :   $x := x - 1$
            $\ell_7$ :   skip ]
$\ell_8$ :
```

Note: Due to the presence of unbounded non-determinism, a deductive termination proof of this program needs to use a ranking function ranging over lexicographic triplets, whose core is $(\textit{at}\_\ell_0, x, y)$.

The Augmented-Abstraction Version

We augment the system with monitors for the ranking functions x, y, and abstract the domain of x, y into $\{\mathit{zero}, \mathit{pos}\}$. This yields:

```
            $X, Y: :$                 $\{\mathit{zero}, \mathit{pos}\}$
            $\mathit{incx}, \mathit{incy}:$     $\{-1, 0, 1\}$
            compassion            $(\mathit{incx} < 0, \mathit{incx} > 0)$,   $(\mathit{incy} < 0, \mathit{incy} > 0)$
$\ell_0$:  $(X, Y, \mathit{incx}, \mathit{incy}) := (?, Y, ?, 0)$
$\ell_1$:  while $X = \mathit{pos}$ do
    $\ell_2$:  $(X, Y, \mathit{incx}, \mathit{incy}) := (X, ?, 0, ?)$
    $\ell_3$:  while $Y = \mathit{pos}$ do
        $\ell_4$:  $(X, Y, \mathit{incx}, \mathit{incy}) := \left(\begin{array}{l}\text{if } Y = \mathit{zero} \text{ then } (X, \mathit{zero}, 0, 0) \text{ else} \\ \{(X, \mathit{pos}, 0, -1), (X, \mathit{zero}, 0, -1)\}\end{array}\right)$
        $\ell_5$:  $\mathit{incy} := 0$
    $\ell_6$:  $(X, Y, \mathit{incx}, \mathit{incy}) := \left(\begin{array}{l}\text{if } X = \mathit{zero} \text{ then } (\mathit{zero}, Y, 0, 0) \text{ else} \\ \{(\mathit{pos}, Y, -1, 0), (\mathit{zero}, Y, -1, 0)\}\end{array}\right)$
    $\ell_7$:  $\mathit{incx} := 0;$
$\ell_8$:
```

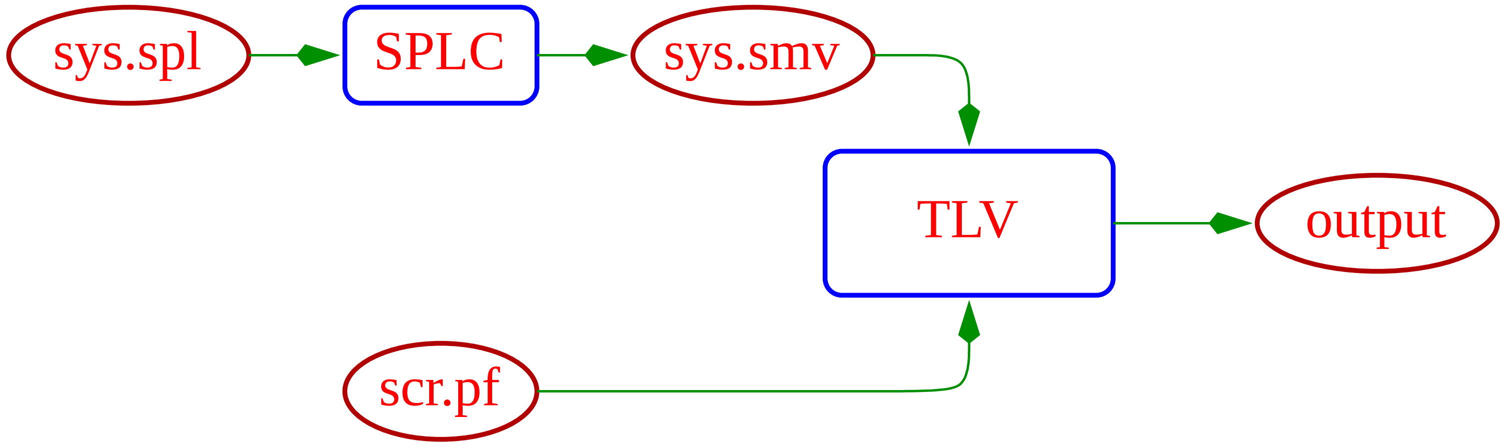
Model checking this program, we find that it always terminates.

Doing it in TLV

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

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 computations of abstractions.



File `nested-aug.smv`

```
MODULE main
VAR
  inpb: {zero,vpos};  inpc: -1..1;
  pi: 0..8;
  X,Y : {zero,vpos};
  incx,incy : -1..1;
ASSIGN
  init(pi) := 0;
  next(pi) := case
                pi in {0,2,4,6} : pi+1;
                pi=1 & X=vpos   : 2;
                pi=1            : 8;
                pi=3 & Y=vpos   : 4;
                pi=3            : 6;
                pi=5            : 3;
                pi=7            : 1;
                1               : pi;
              esac;
  next(X)   := case   pi=0               : inpb;
                      pi=6 & X=zero : zero;
                      pi=6               : inpb;
                      1                  : X;       esac;

  next(Y)   := case   pi=2               : inpb;
                      pi=4 & Y=zero : zero;
                      pi=4               : inpb;
                      1                  : Y;       esac;

  next(incx)  := case pi=0               : inpc;
                      pi=6 & X=zero : 0;
                      pi=6               : -1;
                      1                  : 0;       esac;
  next(incy)  := case pi=2               : inpc;
                      pi=4 & Y=zero : 0;
                      pi=4               : -1;
                      1                  : 0;       esac;
JUSTICE  1
COMPASSION  (incx<0,incx>0),(incy<0,incy>0)
```

File `nested-aug.pf`

In file `nested-aug.pf`, we place the following script:

```
Print "\n Model check termination of augmented program\n";
Call Temp_Entail(1,pi=8);
```

Doing it All within TLV

Outline:

- Small model theorem, justifying truncation of infinie-state systems.
- Computing abstractions (predicates+ranking) within TLV.
- Shape Analysis via P+R abstraction.

Computing the Abstraction within the Model Checker

For some simple infinite domains, it is possible to decide satisfiability (validity) by boolean methods. For such domains, it is possible to compute and apply the abstraction, all within a single session of the model checker.

A Poor Man's Decision Procedure

Consider system variables:

N :	**natural**
$z_1, \ldots, z_n$:	$1..N$

An EA-assertion is a formula of the form $\varphi : \exists \vec{x} \forall \vec{y}. p(\vec{x}, \vec{y}, \vec{u})$, where p is a boolean combination of atomic formulas of the form $z_i < z_j$.

Small Model Theorem

Claim 7.
Assertion $\varphi = \exists \vec{x} \forall \vec{y}.p(\vec{x}, \vec{y}, \vec{u})$ *is satisfiable iff* φ *is satisfiable in a model of size* $\leq N_0 = |\vec{x}| + |\vec{u}|$.

Proof:
Assume φ is satisfiable in a model M_1 of size $N_1 > N_0$. We show how to construct a satisfying model M_2 of size $N_2 \leq N_0$.

Let $v_1 < v_2 < \cdots < v_k$ be all the distinct values assumed by $\vec{x}, \vec{u}$ in model M_1. Obviously $k \leq N_0$. We construct a model M_2 of size k.

For every $z \in \vec{x} \cup \vec{u}$, let

$$M_2[z] = j \qquad \text{iff} \qquad M_1[z] = v_j$$

We can show that $M_2 \models \varphi$ ▟

Truncation of Infinite-state Systems

Let P be a (potentially) infinite-state program. Denote by $\lfloor P \rfloor_N$ the N-truncated version of P obtained by restricting all integer variables to the subrange $-N..N$ (or $0..N$ for naturals) and all array bounds to N.

Let α be a finitary abstraction mapping and R a ranking augmentation which is a conjunction of expression of the form $\mathit{inc}' = \mathit{sign}(\delta' - \delta)$.

An abstraction problem (P, α, R) is called truncatable if there exists a natural $N > 0$ such that

$$\alpha(P \mathrel{|||} R) \quad \sim \quad \alpha(\lfloor P \rfloor_N \mathrel{|||} R)$$

Claim 8. *If all assertions occurring within α, R, and the* FDS *of P are EA-assertions, then (P, α, δ) is truncatable.*

The value of N is determined as he maximal $|\vec{x}| + |\vec{u}|$ over all abstraction formulas.

Example: Program NESTED-LOOPS

Reconsider program NESTED-LOOPS.

```
              x, y: natural
ℓ0 :  x :=?
ℓ1 :  while x > 0 do
        [ ℓ2 :  y :=?
          ℓ3 :  while y > 0 do
                  [ ℓ4 :  y := y − 1 ]
                  [ ℓ5 :  skip       ]
          ℓ6 :  x := x − 1
          ℓ7 :  skip                   ]
ℓ8 :
```

with the abstracion $\alpha : (X = (x > 0), Y = (y > 0))$ and ranking augmentation $R : \mathit{incx}' = \mathit{sign}(x' - x) \ \wedge \ \mathit{incy}' = \mathit{sign}(y' - y)$.

Note that $x' = x - 1$ is expressible as $x = 0 \wedge x' = 0 \ \vee \ x' < x \wedge \forall u.u \leq x' \vee u \geq x$, and $\mathit{incx}' = \mathit{sign}(x' - x)$ is expressible as

$$\mathit{incx}' = -1 \wedge x' < x \ \vee \ \mathit{incx}' = 0 \wedge x' = x \ \vee \ \mathit{incx}' = 1 \wedge x' > x$$

Shape Analysis by P+R-Abstracion

Consider the REVERSE program:

$$\ell_0 : y := \textit{null}; \quad \ell_1 : \textbf{while } x \neq \textit{null} \textbf{ do } \ell_2 : (x, y, x.n) := (x.n, x, y); \quad \ell_3 :$$

We define the predicate $\textit{reach}(u, v)$ which means that there is a chain of *next*-links leading from the node pointed to by u to the node to which v points.

One of the properties we would like to prove is

$$\textit{at}\_\ell_0 \wedge t \neq \textit{null} \wedge \textit{reach}(x, t) \rightarrow \square\, (\textit{at}\_\ell_3 \rightarrow \textit{reach}(y, t))$$

We therefore assume the initial condition $\Theta : t \neq \textit{null} \wedge \textit{reach}(x, t)$, and verify the invariance property

$$\square\, (\textit{at}\_\ell_3 \rightarrow \textit{reach}(y, t))$$

As a predicate base we take the following predicates:

$$x = \textit{null},\ t = \textit{null},\ \textit{reach}(x, t),\ \textit{reach}(y, t)$$

and use the following abstraction mapping:

$$\textit{x\_null} = (x = \textit{null}), \quad \textit{t\_null} = (t = \textit{null}), \quad \textit{r\_xt} = \textit{reach}(x, t), \quad \textit{r\_yt} = \textit{reach}(y, t)$$

The Abstracted Program

This leads to the following abstract program:

```
$\mathit{x\_null}, \mathit{t\_null}, \mathit{r\_xt}, \mathit{r\_yt}$ : boolean where $\mathit{x\_null} = \mathit{t\_null} = 0,\ \mathit{r\_xt} = 1$
$\ell_0$ : $\mathit{r\_yt} := \mathit{t\_null}$
$\ell_1$ : while $\neg \mathit{x\_null}$ do
      $\ell_2$ : [ $(\mathit{r\_xt}, \mathit{r\_yt}) :=$ case
                        $\neg \mathit{r\_xt} \wedge \neg \mathit{r\_yt}$ : $(0,0)$
                        $\neg \mathit{r\_xt} \wedge \mathit{r\_yt}$   : $\{(0,1),(1,1)\}$
                        $1$                 : $\{(0,1),(1,0),(1,1)\}$
                   esac
             $\mathit{x\_null} :=$ if $\mathit{r\_xt}$ then $0$ else $\{0,1\}$ ]
$3$
```

It is not difficult to verify (say by model checking) that

$$\Pi = 3 \;\Rightarrow\; \mathit{r\_yt}$$

Doing it in TLV

We will proceed to show how the analysis of this pointer manipulating program can be managed automatically within the TLV framework.

We start by considering a finite-state bounded instance of program REVERSE. In file `reverse.smv` (next slide), we present an SMV program in which the heap size has been set to 4. In this presentation, the array `Next` represents the *next*-links. The special index 0 represents *null*. We allocate the variables `x`, `y`, and `t`, all of them ranging over the domain $0..4$.

Note that 0 is not a legitimate index in the array `Next`, however, since we wanted the definition `Nextx:= Next[x]` to be valid uniformly, we added the extra definition `Next[0] := 0`.

The body of the program executes the simultaneous assignment $(x, y, x.n) := (x.n, x, y)$ whenever $x \neq$ *null*. The assignment to $x.n$ is performed by the loop constructor

```
for (i=1; i<= N; i=i+1)
     {next(Next[i]) := (go & i=x) ? y : Next[i];}
```

File `reverse.smv`

```
MODULE main
DEFINE N:=4;
       null := 0;
       go := (x != null);
       Next[0] := 0;
       Nextx := Next[x];
VAR    x : 0..N;    y : 0..N;    t : 0..N;
       Next : array 1..N of 0..N;
ASSIGN   next(x) := case
                      go : Nextx;
                      1  : x;
                    esac;
         next(y) := case
                      go : x;
                      1  : y;
                    esac;
        next(t) := t;
        for (i=1; i<= N; i=i+1)
           {next(Next[i]) := (go & i=x) ? y : Next[i];}
```

Preparation of the `pf` File

Note that file `reverse.smv` does not specify any initial condition. This is because the initial condition should specify that t is reachable from x, and this requirement is better specified in the `pf` file.

File `abs-reverse.pf` starts with several definition of functions:

Function `reach1(x,y)` returns an assertion which expresses the property that y is reachable from x in 0 or 1 steps. For the case of 1 step, it is required that $x \neq \mathit{null}$.

Function `reach2(x,y)` returns an assertion which expresses the property that y is reachable from x in 2 or less steps.

Function `reach4(x,y)` returns an assertion which expresses the property that y is reachable from x in 4 or less steps. For a heap of size 4 this also captures the fact that y is reachable from x by a path of any length.

Following these definitions, we set the initial condition of the system to require $y = \mathit{null} \;\wedge\; t \neq \mathit{null} \;\wedge\; \mathit{reach}(x,t)$. We then continue to model check that $x = \mathit{null} \rightarrow \mathit{reach}(y,t)$ is an invariant of the system.

File abs-reverse.pf

```
Func reach1(x,y);
  Return x!=nil & (Next[x]=y) | (x=y);
End -- Func reach1(x,y)
Func reach2(x,y);
  Local r := 0;
  For (i in 1...N)
    Let r := r | reach1(x,i) & reach1(i,y);
  End -- For (i in 1...N)
Return r;  End -- Func reach2(x,y)
Func reach4(x,y);
  Local r := 0;
  For (i in 1...N)
    Let r := r | reach2(x,i) & reach2(i,y);
  End -- For (i in 1...N)
Return r;   End -- Func reach4(x,y)
Let nil := 0;
Print "Model Check Integrity of Terms\n";
Let _s[1].i := (y=nil) & (t != nil) & reach4(x,t);
Call Invariance(x=nil -> reach4(y,t));
```

Moving to Abstraction

In the next step we move to computing and model checking abstraction of program `abs-reverse.smv`, using the abstraction

$$\textit{x\_null} = (x = \textit{null}), \quad \textit{t\_null} = (t = \textit{null}), \quad \textit{r\_xt} = \textit{reach}(x, t), \quad \textit{r\_yt} = \textit{reach}(y, t)$$

We prepare file `abs-reverse.smv` in which we embed two systems: `conc-reverse` which is a copy of `reverse.smv` and `abs-reverse` which would hold the abstract version. The `abs-reverse` system only defines the abstract variables `xnull`, `tnull`, `rxt`, and `ryt`.

The definitions of `x`, `y`, and `t` at the top level is intended to make this variables (which are local to system `CS`) visible at the top level. Similarly for the abstract variables `xnull`, `tnull`, `rxt`, and `ryt`.

File abs-reverse.smv

```
MODULE main
DEFINE  N:= 4;  x:= CS.x;  y:= CS.y;  t:= CS.t;
  xnull:= AS.xnull;  tnull:= AS.tnull;
  rxt  := AS.rxt;     ryt  := AS.ryt;
VAR    CS: system conc-reverse(N);
       AS: system abs-reverse;
MODULE conc-reverse(N)
DEFINE null   := 0;  go   := (x != null);
       Next[0]:= 0;  Nextx:= Next[x];
VAR  x : 0..N;   y : 0..N;   t : 0..N;
     Next : array 1..N of 0..N;
ASSIGN   next(x) := go ? Nextx: x;
         next(y) := go ? x : y;
         next(t) := t;
  for (i=1; i<= N; i=i+1)
    {next(Next[i]) := (go & i=x) ? y : Next[i];}
MODULE abs-reverse
VAR  xnull: boolean;  tnull: boolean;
     rxt  : boolean;  ryt  : boolean;
```

File `abs-reverse.pf`

This file consists of the following parts:

1. Functions `reach1`, `reach2`, and `reach`=`reach4`.

2. Abstractions of an assertion, a relation and a Transition System, assuming that the abstraction mapping has been pre-computed.

3. A new procedure `check_counter` which prints a concretized version of a counter example, in case the abstract model checking produced an abstract counter-example. It should be invoked following every call to `Invariance`.

4. The main part of the program, which defines the abstraction mapping, performs the abstraction of `CS` into `AS` and invokes model checking on the abstract system.

We will present each part in a separate slide.

The reach Functions

```
Func reach1(x,y);
  Local result := (CS.Next[x]=y) & x!=nil | (x=y);
  Return result;
End -- Func reach1(x,y)
Func reach2(x,y);
  Local r := 0;
  For (i in 1...N)
    Let r := r | reach1(x,i) & reach1(i,y);
  End -- For (i in 1...N)
  Return r;
End -- Func reach2(x,y)
Func reach(x,y);
  Local r := 0;
  For (i in 1...N)
    Let r := r | reach2(x,i) & reach2(i,y);
  End -- For (i in 1...N)
  Return r;
End -- Func reach(x,y)
```

The Abstraction Functions

```
Func abs-assert(phi);
  Return (phi & abst) forsome vars1;  End -- abs-assert
Func concs(st);  -- Concretize State ‘‘st’’
  Return (st & abst) forsome vars2;   End -- concs;
Func abs-trans(rho);
  Return (rho & abst & abstp) forsome all_vars1;
End -- Func abs-assert(phi);
To abs-sys;
  Let _s[2].i := abs-assert(_s[1].i);
  Let _s[2].tn := _s[1].tn;
  For (i in 1..._s[1].tn)
    Let _s[2].t[i] := abs-trans(_s[1].t[i]);
  End -- For (i in 1..._s[1].tn)
  Let _s[2].jn := _s[1].jn;
  For (i in 1..._s[1].jn)
    Let _s[2].j[i] := abs-assert(_s[1].j[i]);
  End -- For (i in 1..._s[1].jn)
End -- To abs-sys;
```

Print Concrete Counter Example

```
To check_counter; -- Print concrete counter-example if exists
  If(exist(ce[1]))
    Print "\n Concrete counter-example follows:\n";
    Local L := length(ce);   Local ic := 1;
    Let cce[1] := concs(ce[1]) & _s[1].i;
    While (ic<L)
      Local nxst := succ1(cce[ic],1) & concs(ce[ic+1]);
      If(nxst) Let ic := ic+1;  Let cce[ic] := nxst;
      Else     Break;     End -- If(nxst)
    End -- While (ic<L)
    Let cce[ic] := fsat(cce[ic],vars1);  Let jc := ic - 1;
    While (jc>0)
      Let cce[jc] := fsat(pred1(cce[jc+1],1) & cce[jc],vars1);
      Let jc := jc - 1;
    End -- While (jc>0)
    For (i in 1...ic)
      Print "\n---- State no. ", i," =\n", cce[i];  End -- For
  End -- If(exist(ce[1]))
End -- To check_counter;
```

Main Part of File `abs-reverse4.pf`

```
Let vars1 := _s[1].v;
Let vars2 := _s[2].v;
Let all_vars1 := set_union(vars1,prime(vars1));
Let nil := 1;
Let _s[1].i := (y=nil) & (t != nil) & reach(x,t);
Let abst := (xnull <-> (x=nil)) &
            (tnull <-> (t=nil)) &
            (rxt   <-> (reach(x,t))) &
            (ryt   <-> (reach(y,t));
Let abstp := prime(abst);
abs-sys;
Print "Check presence of t in abstract system\n";
Call Invariance(xnull -> ryt,2);
check_counter;
```

Proving Progress Properties

Reconsider the REVERSE program:

$$\ell_0 : y := \textit{null}; \quad \ell_1 : \textbf{while } x \neq \textit{null} \textbf{ do } \ell_2 : (x, y, x.n) := (x.n, x, y); \quad \ell_3 :$$

A relevant progress property of this program is that of termination, which can be specified as

$$1 \Rightarrow \Diamond (pi = 3)$$

A possible way of solving the problem is that of augmentation, composing the system with the following progress monitor:

```
[ inc : $\{-1, 0, 1\}$
  compassion $(\textit{inc} < 0, \textit{inc} > 0)$
  loop forever do
     $\textit{inc} := \textit{sign}(|\{i \mid \textit{reach}'(x, i)\}| - |\{j \mid \textit{reach}(x', j)\}|)$ ]
```

In the following slides we will show the additions to the relevant smv/pf files.

Additions to `abs-reverse.smv`

To module `conc-reverse`, we add the following paragraphs:

```
JUSTICE  1
```

To module `abs-reverse` we add the declaration

```
inc: -1..1;
```

Additions to File `abs-reverse.pf`

We add the following auxiliary function which computes the ranking:

```
Func del(x);
  Local sum := 0;
  For (i in 1...N)
    Let sum := sum + reach(x,i);
  End -- For (i in 1...N)
  Return sum;
End -- Func del(x);
```

Then, we add the following statements to the main part:

```
Let abst := abst & (AS.Inc=CS.inc);
Let diff := prime(del(x))-del(x);
Let _s[1].t[1] := _s[1].t[1] &
         (next(AS.inc) = case
                           diff>0 :  1;
                           diff<0 : -1;
                           1      :  0;   esac);
Print "Check Termination\n";  Call Temp_Entail(1,xnull,2);
```

First Attempt

```
*** Property is NOT VALID ***
Counter-Example Follows:
---- State no. 1 =
AS.Pi = 0,     AS.xnull = 0,  AS.rxt = 1,  AS.ryt = 0,
AS.tnull = 0, AS.inc = -1,
---- State no. 2 =
AS.Pi = 1,     AS.xnull = 0,  AS.rxt = 1,  AS.ryt = 0,
AS.tnull = 0, AS.inc = 0,
---- State no. 3 =
AS.Pi = 2,     AS.xnull = 0,  AS.rxt = 1,  AS.ryt = 0,
AS.tnull = 0, AS.inc = 0,
---- State no. 4 =
AS.Pi = 1,     AS.xnull = 0,  AS.rxt = 0,  AS.ryt = 1,
AS.tnull = 0, AS.inc = 1,
---- State no. 5 =
AS.Pi = 2,     AS.xnull = 0,  AS.rxt = 0,  AS.ryt = 1,
AS.tnull = 0, AS.inc = 0,
 Repeating Period
---- State no. 6 =
AS.Pi = 2,     AS.xnull = 0,  AS.rxt = 0,  AS.ryt = 1,
AS.tnull = 0, AS.inc = 0,
---- State no. 7 =
AS.Pi = 1,     AS.xnull = 0,  AS.rxt = 0,  AS.ryt = 1,
AS.tnull = 0, AS.inc = 1,
```

What Went Wrong?

According to the counter-example, t is possible o take a transition which causes the number of nodes reachable from x to increase. To see how this is possible, we performed the following:

```
>> Let t1:= _s[1].t[1];
>> Let st := x!=nil & t!=nil;
>> Print st & t1 & next(inc)=1;
pi = 2,1        x = 4,4         y = 2,4         t = 4,4
Next[1] = 0,0   Next[2] = 0,0   Next[3] = 0,0   Next[4] = 4,2
inc = ,1
```

This shows that the number of x-reachable nodes can increase if one of these nodes participate in a cycle.

To avoid this, we add the predicate

$$\textit{r\_xn} = \textit{reach}(x, \textit{null})$$

and added $\textit{reach}(x, \textit{null})$ to the initial condition.

Now it ran successfully!!!