

# Using **Abstraction** to Verify Arbitrary **Temporal** Properties

Amir Pnueli

New York University and Weizmann Institute of Sciences (Emeritus)

Perspectives in Concurrency Theory,  
Chennai, 15 December 2008

Joint work with

Ittai Balaban, Yonit Kesten, Lenore Zuck

# Dedication

This lecture is dedicated to

**Prof. P.S. Thiagarajan**

a great scientist and a dear personal friend on his 60th birthday.

Dear Thiagu, may you have (infinitely) many more years of continued constructive scientific activity, a healthy and happy life in the bosom of your loving family!

# The Verification Problem

Given a system (program, HW design)  $S$  and a property  $\varphi$ , ascertain that

$$S \models \varphi$$

That is, all behaviors (executions) of  $S$  satisfy  $\varphi$ .

In practice, this is usually “established” by **testing**. Full coverage not guaranteed.

Obviously, solving the verification problems is essential for the construction of correct and reliable programs.

# Formal Verification

Establish  $S \models \varphi$  with **mathematical certainty**. Two (and a half) methods have been proposed:

## Algorithmic Verification (Model Checking)

For finite-state systems, systematically explore all possible behaviors.

- + Fully automatic.
- Restricted to (not too big) finite-state systems.

## Deductive Verification

Devise auxiliary assertions (invariants) and ranking functions (variants) and establish, using automated theorem provers, the validity of proof rules (e.g. induction).

- + Complete. Applicable to infinite-state systems (programs).
- Requires user expertise and ingenuity.

## Third Method – Abstraction

Compute (characterize) all behaviors of system  $S$  and then examine whether property  $\varphi$  is satisfied by all of them.

Usually intractable. So instead of analyzing  $S$  we analyze an **abstracted version** of  $S$ .

- + Requires less user ingenuity. Often the creative step is the selection of the abstract domain and abstraction mapping.
- Standard method can deal with only a subset of the properties one may wish to verify

Resolving the last deficiency is what this lecture is about.

## 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  $\psi$  into a propositional **LTL** formula  $\psi^\alpha$ .
- **Model check**  $\mathcal{D}^\alpha \models \psi^\alpha$ .

The question considered here is finding **instantiations** of this general methodology which are **sound** and **(relatively) complete** for both **safety** and **liveness** properties.

# State Abstraction

Based on the notion of **abstract interpretation** [CC77]. There are, however, several technical differences.

Let  $\Sigma$  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**.  $\alpha$  is **finitary** if  $\Sigma_A$  is finite.

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

## Fair Discrete Systems

As a computational model for reactive systems, we take **fair discrete system (FDS)**  $\mathcal{D} = \langle V, \Theta, \rho, \mathcal{J}, \mathcal{C} \rangle$  consisting of:

- $V$  – A finite set of typed **state variables**. A  $V$ -state  $s$  is an interpretation of  $V$ .  
 $\Sigma_V$  – the set of all  $V$ -states.
- $\Theta$  – An **initial condition**. A satisfiable assertion that characterizes the **initial states**.
- $\rho$  – 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  $x := x + 1$ .

- $\mathcal{J} = \{J_1, \dots, J_k\}$  A set of **justice (weak fairness)** requirements. Ensure that a computation has **infinitely many  $J_i$ -states** for each  $J_i$ ,  $i = 1, \dots, k$ .
- $\mathcal{C} = \{\langle p_1, q_1 \rangle, \dots, \langle p_n, q_n \rangle\}$  A set of **compassion (strong fairness)** requirements. **Infinitely many  $p_i$ -states** imply **infinitely many  $q_i$ -states**.



## Lifting a State Abstraction to Assertions

For an abstraction mapping  $\alpha : V_A = \mathcal{E}_\alpha(V)$  and an assertion  $p(V)$ , we can lift the state abstraction  $\alpha$  to abstract  $p$ :

- The **expanding  $\alpha$ -abstraction** (over approximation) of  $p$  is given by

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

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

## Sound Joint Abstraction

For a positive normal form temporal formula  $\psi$ , we define  $\psi^\alpha$  to be the formula obtained by replacing every (maximal) state sub-formula  $p \in \psi$  by  $\underline{\alpha}(p) = \neg\bar{\alpha}(\neg p)$ .

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

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

### Soundness:

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

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

## Example: Program INCREASE

Consider the program

$$\begin{array}{l}
 y : \text{integer initially } y = 0 \\
 \left[ \begin{array}{l}
 \ell_0 : \text{while } y \geq 0 \text{ do } [\ell_1 : y := y + 1] \\
 \ell_2 :
 \end{array} \right]
 \end{array}$$

Assume we wish to verify the property  $\diamond \square (y > 0)$  for program INCREASE. This property states that, eventually,  $y$  becomes positive and remains positive forever.

Introduce the abstract variable  $Y : \{-1, 0, +1\}$ .

The abstraction mapping  $\alpha$  is specified by the defining expression:

$$\alpha : [Y = \text{sign}(y)]$$

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

## The Abstracted Version

With the mapping  $\alpha$ , we obtain the abstract version of INCREASE, called INCREASE $^\alpha$ :

$$\begin{array}{c}
 Y: \{-1, 0, +1\} \text{ initially } Y = 0 \\
 \left[ \begin{array}{l}
 \ell_0: \text{ while } Y \in \{0, 1\} \text{ do} \\
 \ell_2:
 \end{array} \left[ \begin{array}{l}
 \ell_1: Y := \left( \begin{array}{l}
 \text{if } Y = -1 \\
 \text{then } \{-1, 0\} \\
 \text{else } +1
 \end{array} \right)
 \end{array} \right] \right]
 \end{array}$$

The original invariance property  $\psi: \diamond \square (y > 0)$ , is abstracted into:

$$\psi^\alpha: \diamond \square (Y = +1),$$

which can be model-checked over INCREASE $^\alpha$ , yielding INCREASE $^\alpha \models \diamond \square (Y = +1)$ , from which we infer

$$\text{INCREASE} \models \diamond \square (y > 0)$$

## Predicate Abstraction

The above style of abstraction abstracts each program variable separately. Such abstraction preserves the variable structure of the program. This is not the most general mode of abstraction, nor is it the most useful.

Let  $p_1, p_2, \dots, p_k$  be a set of assertions (state formulas) referring to the **data** (non-control) state variables. We refer to this set as the **predicate base**. Usually, we include in the base all the atomic formulas appearing within conditions in the program  $P$  and within the temporal formula  $\psi$ .

Following [GS97], define a **predicate abstraction** to be an abstraction mapping of the form

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

where  $B_{p_1}, B_{p_2}, \dots, B_{p_k}$  is a set of **abstract boolean variables**, one corresponding to each assertion appearing in the predicate base.

## Example: Program BAKERY-2

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

$$P_1 :: \left[ \begin{array}{l} \ell_0 : \text{loop forever do} \\ \left[ \begin{array}{l} \ell_1 : \text{Non-Critical} \\ \ell_2 : y_1 := y_2 + 1 \\ \ell_3 : \text{await } y_2 = 0 \vee y_1 < y_2 \\ \ell_4 : \text{Critical} \\ \ell_5 : y_1 := 0 \end{array} \right] \end{array} \right]$$

||

$$P_2 :: \left[ \begin{array}{l} m_0 : \text{loop forever do} \\ \left[ \begin{array}{l} m_1 : \text{Non-Critical} \\ m_2 : y_2 := y_1 + 1 \\ m_3 : \text{await } y_1 = 0 \vee y_2 \leq y_1 \\ m_4 : \text{Critical} \\ m_5 : y_2 := 0 \end{array} \right] \end{array} \right]$$

The temporal properties for program BAKERY-2 are

- $\psi_{exc} : \square \neg (at_{\ell_4} \wedge at_{m_4})$  --- Mutual Exclusion (safety)  
 $\psi_{acc} : \square (at_{\ell_2} \rightarrow \lozenge at_{\ell_4})$  --- Accessibility for  $P_1$  (liveness)

## 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**  $B_{y_1=0} = B_{y_2=0} = 1, B_{y_1 < y_2} = 0$

$$P_1 :: \left[ \begin{array}{l} \ell_0 : \text{loop forever do} \\ \left[ \begin{array}{l} \ell_1 : \text{Non-Critical} \\ \ell_2 : (B_{y_1=0}, B_{y_1 < y_2}) := (0, 0) \\ \ell_3 : \text{await } B_{y_2=0} \vee B_{y_1 < y_2} \\ \ell_4 : \text{Critical} \\ \ell_5 : (B_{y_1=0}, B_{y_1 < y_2}) := (1, \neg B_{y_2=0}) \end{array} \right] \end{array} \right]$$

||

$$P_2 :: \left[ \begin{array}{l} m_0 : \text{loop forever do} \\ \left[ \begin{array}{l} m_1 : \text{Non-Critical} \\ m_2 : (B_{y_2=0}, B_{y_1 < y_2}) := (0, 1) \\ m_3 : \text{await } B_{y_1=0} \vee \neg B_{y_1 < y_2} \\ m_4 : \text{Critical} \\ m_5 : (B_{y_2=0}, B_{y_1 < y_2}) := (1, 0) \end{array} \right] \end{array} \right]$$

The abstracted properties can now be **model-checked**.

# The Question of Completeness

We have claimed above that the VFA method is sound. How about completeness?

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

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

## Claim 1. [Completeness for Invariance Properties]

Let  $\mathcal{D}$  be an FDS and  $\psi : \Box p$  be an invariance property such that  $\mathcal{D} \models \Box p$ . Then there exists a finitary abstraction mapping  $\alpha$  such that  $\mathcal{D}^\alpha \models \Box \alpha(p)$ .

In fact, the proof shows that there always exists a predicate abstraction validating the invariance property.



# Inadequacy of State Abstraction for Proving Liveness

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

Consider the program **LOOP**.

```

      y: natural
l0 : while y > 0 do
      [ l1 : y := y - 1
        l2 : skip
      ]
l3 :

```

**Termination** of this program cannot be proven by pure finitary abstraction. For example, the abstraction  $\alpha : \mathbf{N} \mapsto \{0, +1\}$  leads to the abstracted program

```

      Y: {0, +1}
l0 : while Y = +1 do
      [ l1 : Y := if Y = +1 then {0, +1} else 0
        l2 : skip
      ]
l3 :

```

This abstracted program may **diverge**!

## No Finitary Abstraction Can Lead to Termination

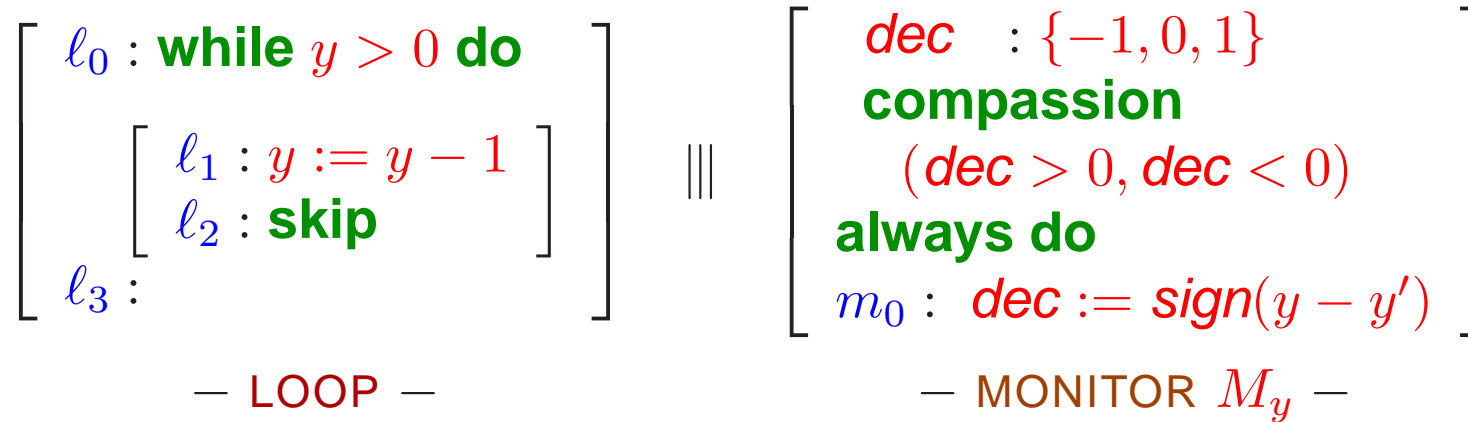
The fault with the above example is not with the particular abstraction chosen. In fact, no finitary abstraction can lead to a terminating program.

Why?

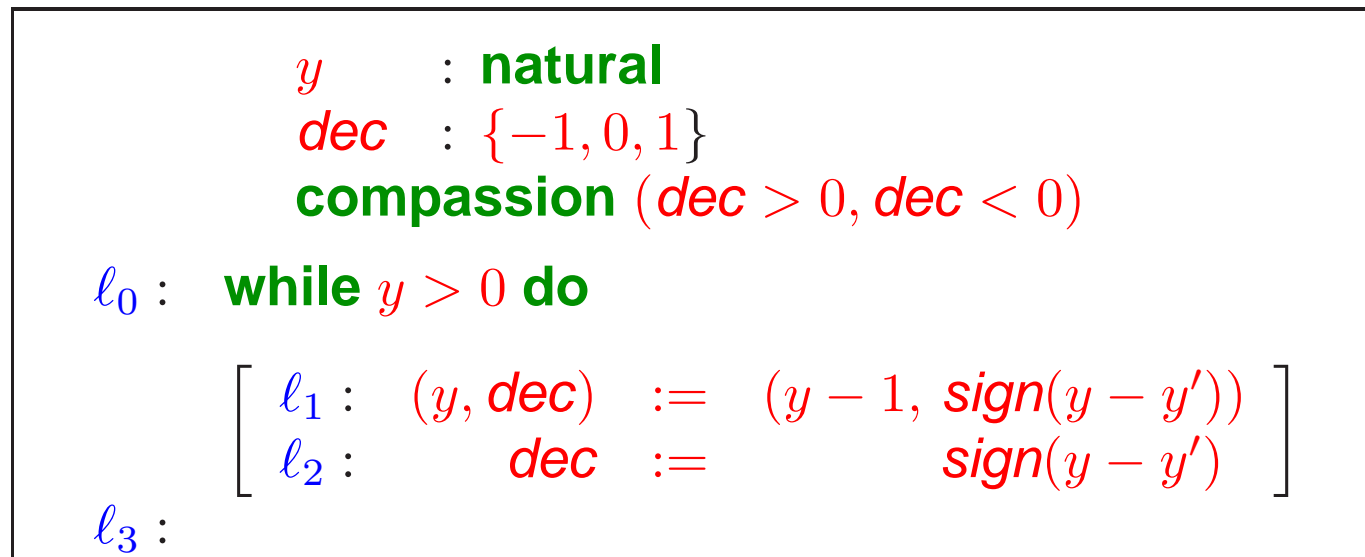
The above program has arbitrary long computations, but they all terminate. No finite-state program can behave in such a manner.

# Solution: Augment with a Non-Constraining Progress Monitor

$y$ : natural

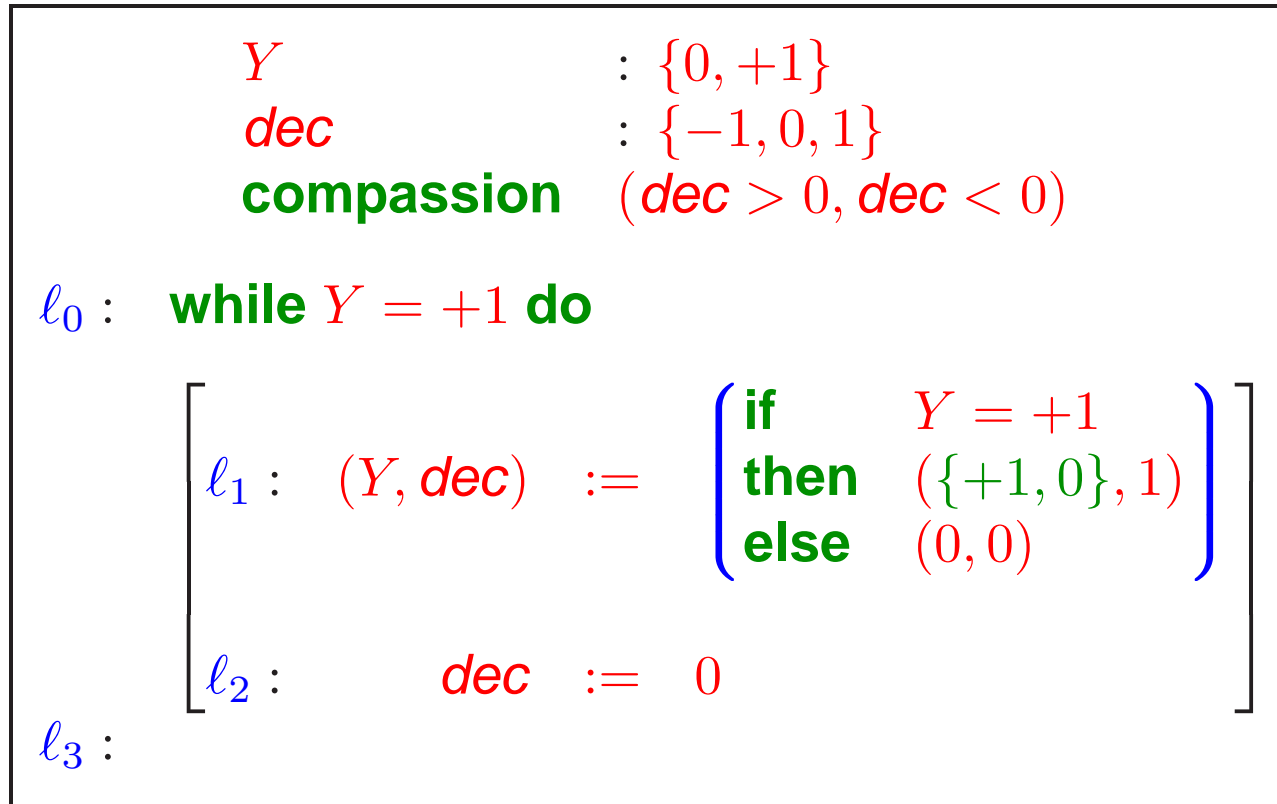


Forming the cross product, we obtain:



# Abstracting the Augmented System

We obtain the program



Which **always terminates**, due to the compassion requirement  $(dec > 0, dec < 0)$ .

## A More Complicated Case

Sometimes we need a more complex progress measure:

	$y$ : <b>natural</b>
$\ell_0$ :	<b>while</b> $y > 1$ <b>do</b>
	$\left[ \begin{array}{l} \ell_1 : y := y - 2 \\ \ell_2 : y := \{y + 1, y\} \\ \ell_3 : \mathbf{skip} \end{array} \right]$
$\ell_4$ :	

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

<b>define</b>	$\delta = y + \mathbf{at\_}\ell_2$
<b>dec</b>	$: \{-1, 0, 1\}$
<b>compassion</b>	$(\mathbf{dec} > 0, \mathbf{dec} < 0)$
$m_0$ :	<b>always do</b>
	$\mathbf{dec} := \mathbf{sign}(\delta - \delta')$

## Complicated Case Continued

Augmenting and abstracting, we get:

$Y$  :  $\{0, \text{one}, \text{large}\}$   
 $dec$  :  $\{-1, 0, 1\}$   
**compassion** ( $dec > 0, dec < 0$ )

$l_0$  : **while**  $Y = \text{large}$  **do**

$$\left[ \begin{array}{l} l_1 : (Y, dec) := (sub2(Y), 1) \\ l_2 : (Y, dec) := \{(add1(Y), 0), (Y, 1)\} \\ l_3 : \quad \quad dec := \quad \quad \quad 0 \end{array} \right]$$

$l_4$  :

where,

$sub2(Y) = \text{if } Y \in \{0, \text{one}\} \text{ then } 0 \text{ else } \{0, \text{one}, \text{large}\}$

$add1(Y) = \text{if } Y = 0 \text{ then } \text{one} \text{ else } \text{large}$

This program **always terminates**

## Verification by Ranking Abstraction

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

- Optionally choose one or more **non-constraining progress monitors**  $M_1, \dots, M_r$  and let  $\mathcal{A} = \mathcal{D} \parallel M_1 \parallel \dots \parallel M_r$ . In case this step is skipped, let  $\mathcal{A} = \mathcal{D}$ .
- Choose a finitary state abstraction mapping  $\alpha$  and calculate  $\mathcal{A}^\alpha$  and  $\psi^\alpha$  according to the sound recipes.
- Model check  $\mathcal{A}^\alpha \models \psi^\alpha$ .
- Infer  $\mathcal{D} \models \psi$ .

**Claim 2.** *The **Ranking Abstraction** method is **complete**, relative to **deductive verification** [KP00].*

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

## Is it Just Deductive Verification in Dressing?

Why is this method better than **deductive verification**?

It is often the case that the user can identify (or conjecture) a set of possible ranking elements, but does not know how to combine them into a single **global ranking function**, which is required by deductive verification.



## An Illustrative Example

Consider the following 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 :
  
```

A deductive termination proof of this program may be based on the ranking function

$$(\mathit{at}_{l_0}, \quad 5 \cdot x + 4 \cdot \mathit{at}_{l_7} + 3 \cdot \mathit{at}_{l_1} + 2 \cdot \mathit{at}_{l_2} + \mathit{at}_{l_{3..5}}, \quad 3 \cdot y + 2 \cdot \mathit{at}_{l_5} + \mathit{at}_{l_3})$$

whose core constituents are *x* and *y*.

## The Ranking Abstraction Approach

We augment the system with monitors for the ranking functions  $x$ ,  $y$ , and abstract the domain of  $x, y$  into  $\{0, +1\}$ . This yields:

```

X, Y : :      {0, +1}
decx, decy : {−1, 0, 1}
compassion (decx > 0, decx < 0),   (decy > 0, decy < 0)

l0 : (X, Y, decx, decy) := (?, Y, ?, 0)
l1 : while X = +1 do
  [
    l2 : (X, Y, decx, decy) := (X, ?, 0, ?)
    l3 : while Y = +1 do
      [
        l4 : (X, Y, decx, decy) := ( if Y = 0 then (X, 0, 0, 0) else
          {(X, +1, 0, 1), (X, 0, 0, 1)} )
        l5 : decy := 0
      ]
    l6 : (X, Y, decx, decy) := ( if X = 0 then (0, Y, 0, 0) else
      {(+1, Y, 1, 0), (0, Y, 1, 0)} )
    l7 : decx := 0;
  ]
l8 :

```

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

## Main Features of Predicate Abstraction

Can be used for the automatic verification of **some LTL** (**all** invariance) properties of infinite-state systems.

- Has a heuristic for an **initial selection** of a **predicate base**: Include all atomic formulas appearing in the program and property.
- Has a heuristic for **refining** the abstraction (expanding the predicate base), as a result of a **spurious counter example**.
- Does not require the specification of an **inductive invariant**. Sufficient to provide the constituents from which such an invariant can be constructed by a **boolean combination**.
- Can be used to **derive** the best inductive invariant expressible over the predicate base: **Abstract**, compute  $Reach(P_A)$ , and then **concretize**.

## In Comparison, Ranking Abstraction

Can be used, in conjunction with **predicate abstraction**, for the automatic verification of **all LTL** properties (in particular, termination) of infinite-state systems.

- Has a heuristic for an **initial selection** of a **ranking core**: Include all variables and expressions which consistently increase (decrease) within loops. Specifically, loop indices.
- Has a heuristic for **refining** the **predicate or ranking** abstraction (expanding the predicate base **or** ranking core), as a result of a **spurious counter example**.
- Does not require the specification of a global **ranking function**. Sufficient to provide the constituents from which such a function can be constructed by a **lexicographic tupling**.
- Can be used to **derive** the best global **ranking function** expressible over the ranking core: Use recursive **SCC**'s analysis.

## A Counter-Example Guided Refinement of a Joint Abstraction

An abstract counter example of a liveness property has the form of a lasso:



As a first step, we attempt to concretize this sequence into a program trace

$$\sigma : s_0, \dots, s_k, \dots, s_n, s_{n+1}$$

such that  $S_i = \alpha(s_i)$ , for  $i \leq n$ , and  $S_k = \alpha(s_{n+1})$ . There are three possible outcomes to this attempt:

1. We succeed to find a concretization such that  $s_{n+1} = s_k$ . In this case, there exists a concrete counter example and the property is **invalid** over the original system. In all other cases, the counter example is **spurious**.
2. The concretization is blocked at state  $s_i$ ,  $i \leq n$ , such that  $s_i$  has no concrete successor belonging to  $\alpha^{-1}(S_{i+1})$ . In this case, apply regular **predicate abstraction refinement** (e.g. [BPR'02]).
3. The concretization completes, but  $s_{n+1} \neq s_k$ . In this case, apply **ranking refinement**. A **loop** has been concretized into a **spiral**.

# Ranking Refinement

Recall the structure of the abstract counter example.



Assume that the labels of states  $S_k, \dots, S_n$  are  $\ell_k, \dots, \ell_n$ . Form the (concrete) transition relation  $\rho_{k..n,k}$  defined by

$$\rho_{k..n,k} : \rho(\ell_k, \ell_{k+1}) \circ \dots \circ \rho(\ell_{n-1}, \ell_n) \circ \rho(\ell_n, \ell_k)$$

This transition relation relates the values of variables in states  $S_k$  and  $S_{n+1}$  such that there exists a computation segment  $S_k, \dots, S_n, S_{n+1}$  passing through the sequence of labels  $\ell_k, \dots, \ell_n, \ell_k$ , respectively.

Also form the assertion  $\varphi_k = S_k[(p_1, \dots, p_r)/(B_1, \dots, B_r)]$  obtained by viewing abstract state  $S_k$  as a boolean expression over the abstract variables  $B_1, \dots, B_r$  and then substituting the predicate  $p_i$  for each occurrence of variable  $B_i$ . This assertion characterizes all the concrete states which are abstracted into  $S_k$ .

## Expanding the Ranking Core

A sufficient condition which guarantees that the obtained lasso cannot be concretized into an infinite computation is that the relation  $\rho_{k..n,k}$  be **well founded** over all  $\varphi_k$ -states. Hence we search for a variable or an expression  $\delta$ , such that

$$\varphi_k \wedge \rho_{k..n,k} \rightarrow \delta > \delta'$$

Heuristics such as the ones expounded in [PR'04] can be used in order to identify such expressions  $\delta$ .

Having found such a  $\delta$ , we add it to the ranking core. Abstract and try again.

## Example

Reconsider a version of program **NESTED-LOOPS**:

```

x, y: natural initially  $x = y = 0$ 
ℓ0 :  $x := ?$ 
      while  $x > 0$  do
        [
          ℓ1 :  $y := ?$ 
            while  $y > 0$  do
              [ ℓ2 :  $y := y - 1$  ]
          ℓ3 :  $x := x - 1$ 
        ]
ℓ4 :

```

Apply joint abstraction with  $\{X = \mathit{sign}(x), Y = \mathit{sign}(y), \mathit{decy} = \mathit{sign}(y - y')\}$ . Note that the ranking core is incomplete.



## The Abstracted program

With the abstraction  $\{X = \text{sign}(x), Y = \text{sign}(y), \text{decy} = \text{sign}(y - y')\}$ , we obtain:

```

      X, Y: :      {0, +1} initially X = Y = 0
      decy :      {-1, 0, 1} initially decy = 0
      compassion (decy > 0, decy < 0)

l0 :  X := {0, 1}
      while X = 1 do
      [
l1 :  (Y, decy) := (?, ?)
      while Y = 1 do
      [l2 :  (Y, decy) := if Y = 0 then (0, 0) else {(0, 1), (1, 1)}]
l3 :  X := if X = 0 then 0 else {0, 1}
      ]
l4 :

```

Model checking this program for termination, we obtain the following counter-example lasso:

$$\begin{aligned}
 S_0 &: \langle \Pi : l_0, X : 0, Y : 0, \text{Decy} : 0 \rangle, \\
 S_1 &: \langle \Pi : l_1, X : 1, Y : 0, \text{Decy} : 0 \rangle, \quad S_2 : \langle \Pi : l_2, X : 1, Y : 1, \text{Decy} : -1 \rangle, \\
 S_3 &: \langle \Pi : l_3, X : 1, Y : 0, \text{Decy} : 1 \rangle, \quad S_4 = S_1
 \end{aligned}$$

## Concretizing and Refining

Concretizing the abstract trace

$$\begin{aligned}
 S_0 &: \langle \Pi : \ell_0, X : 0, Y : 0, \mathit{Decy} : 0 \rangle, \\
 S_1 &: \langle \Pi : \ell_1, X : 1, Y : 0, \mathit{Decy} : 0 \rangle, \quad S_2 : \langle \Pi : \ell_2, X : 1, Y : 1, \mathit{Decy} : -1 \rangle, \\
 S_3 &: \langle \Pi : \ell_3, X : 1, Y : 0, \mathit{Decy} : 1 \rangle, \quad S_4 = S_1
 \end{aligned}$$

we obtain:

$$\begin{aligned}
 s_0 &: \langle \pi : \ell_0, x : 0, y : 0, \mathit{decy} : 0 \rangle, \\
 s_1 &: \langle \pi : \ell_1, x : 4, y : 0, \mathit{decy} : 0 \rangle, \quad s_2 : \langle \pi : \ell_2, x : 4, y : 1, \mathit{decy} : -1 \rangle, \\
 s_3 &: \langle \pi : \ell_3, x : 4, y : 0, \mathit{decy} : 1 \rangle, \quad s_4 : \langle \pi : \ell_1, x : 3, y : 0, \mathit{decy} : 0 \rangle
 \end{aligned}$$

We therefore compute  $\varphi_1 : x > 0 \wedge y = 0$  and  $\rho_{1..3,1} : x' = x - 1 \wedge x' > 0$ . A natural choice for additional rank is  $\delta = x$  whose descent is implied by  $\rho_{1..3,1}$ .

# A Global Ranking Function From a Terminating Program

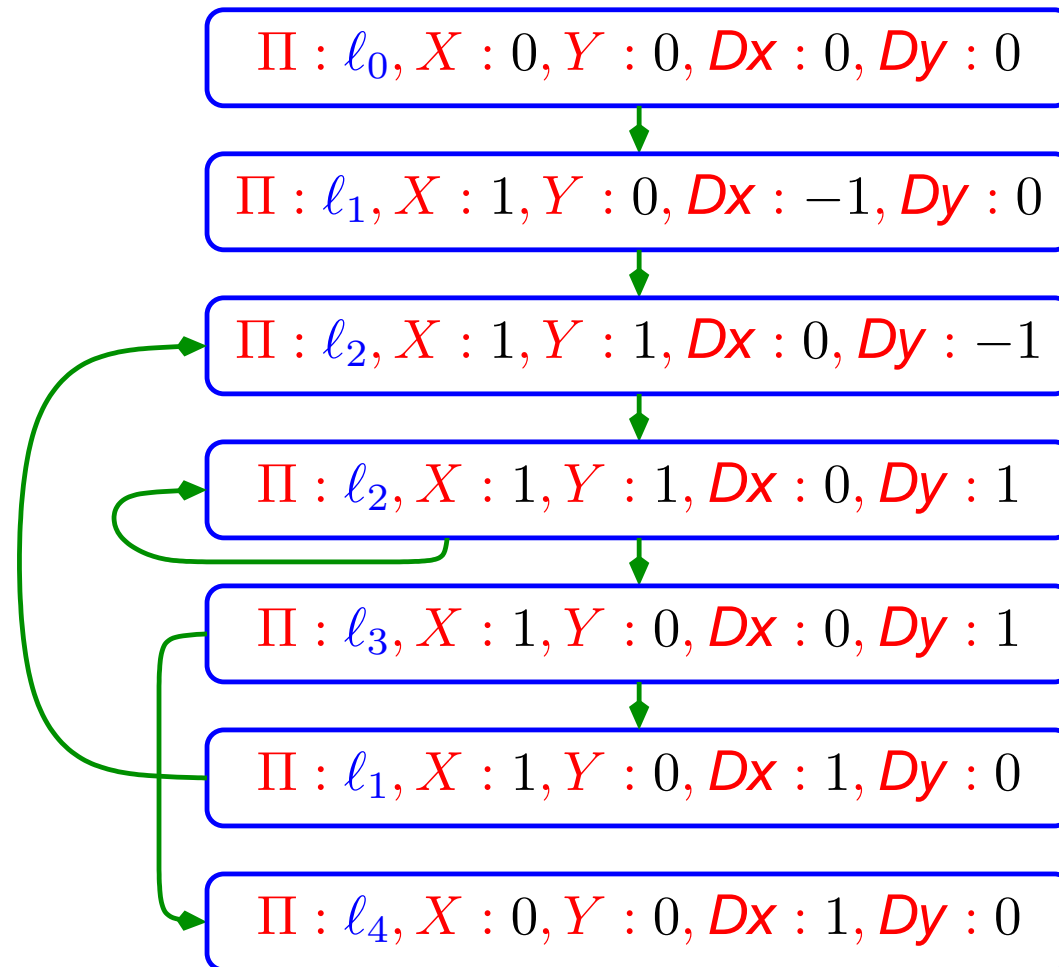
We will show how to extract a **global ranking function** from an **abstract terminating program**. Assume that we constructed a state-transition graph containing all the reachable states of the abstracted program.

The extraction algorithm can be described as follows:

- **Decompose** into **MSCC**'s, **Sort** topologically, and **Rank** sequentially.
- For each non-singular component:
  - **Identify** a compassion req.  $(decx_i > 0, decx_i < 0)$  violated by the component.
  - **Add**  $x_i$  to the ranking tuple.
  - **Remove** all edges entering  $(decx_i > 0)$ -nodes.
  - **Return** to top for recursive processing of remaining subgraph.

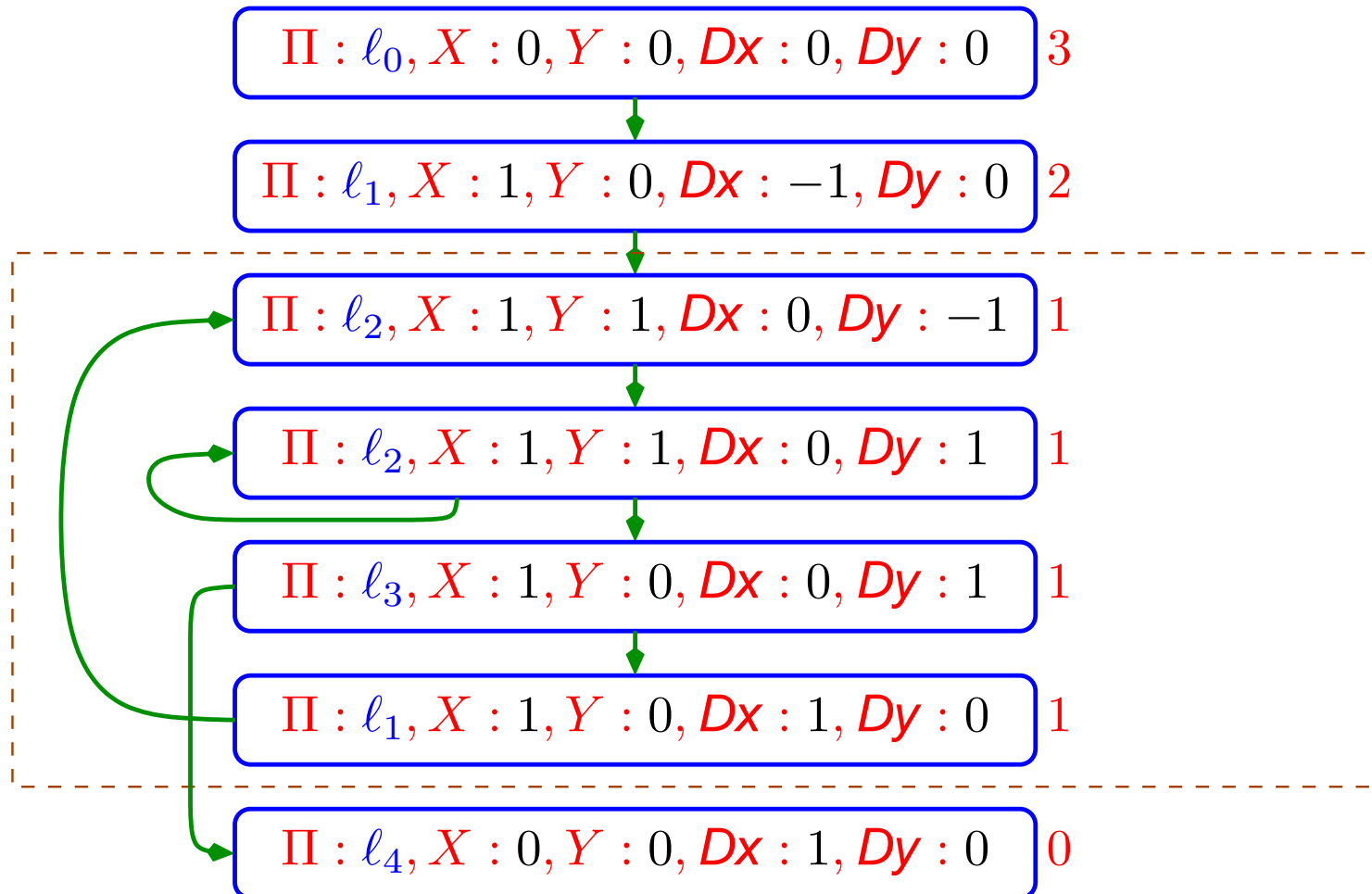
## Example

Analyzing abstracted program **NESTED-LOOPS** with ranking core consisting of  $\{x, y\}$ , the program always terminates. The resulting state transition graph is:



## Decompose, Sort, and Rank

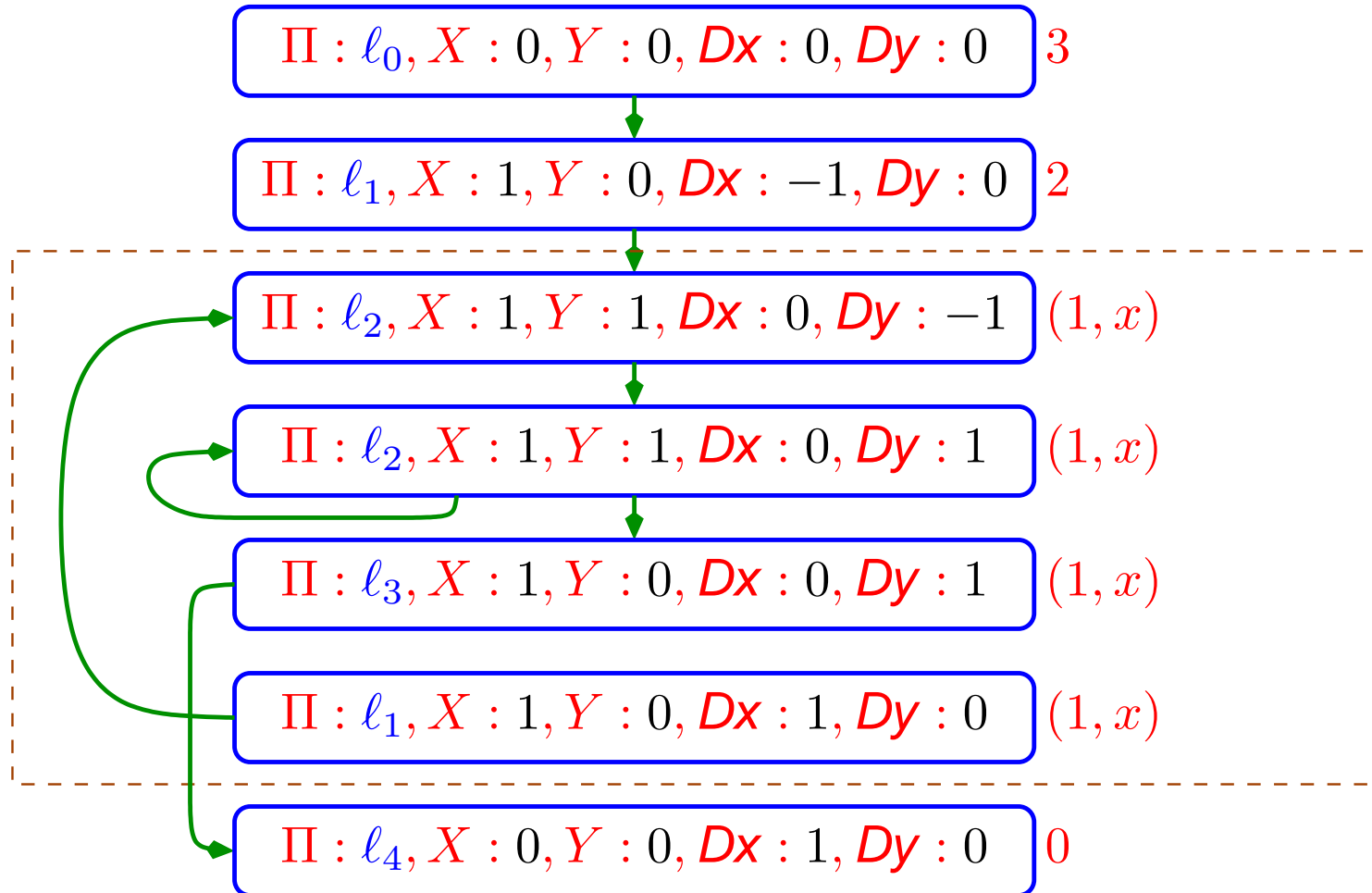
MSCC's decomposition, topologically sorting, and sequentially ranking, yields:



Non-singular component is unfair w.r.t ( $Dx > 0, Dx < 0$ ).

## Add $x$ to Ranking

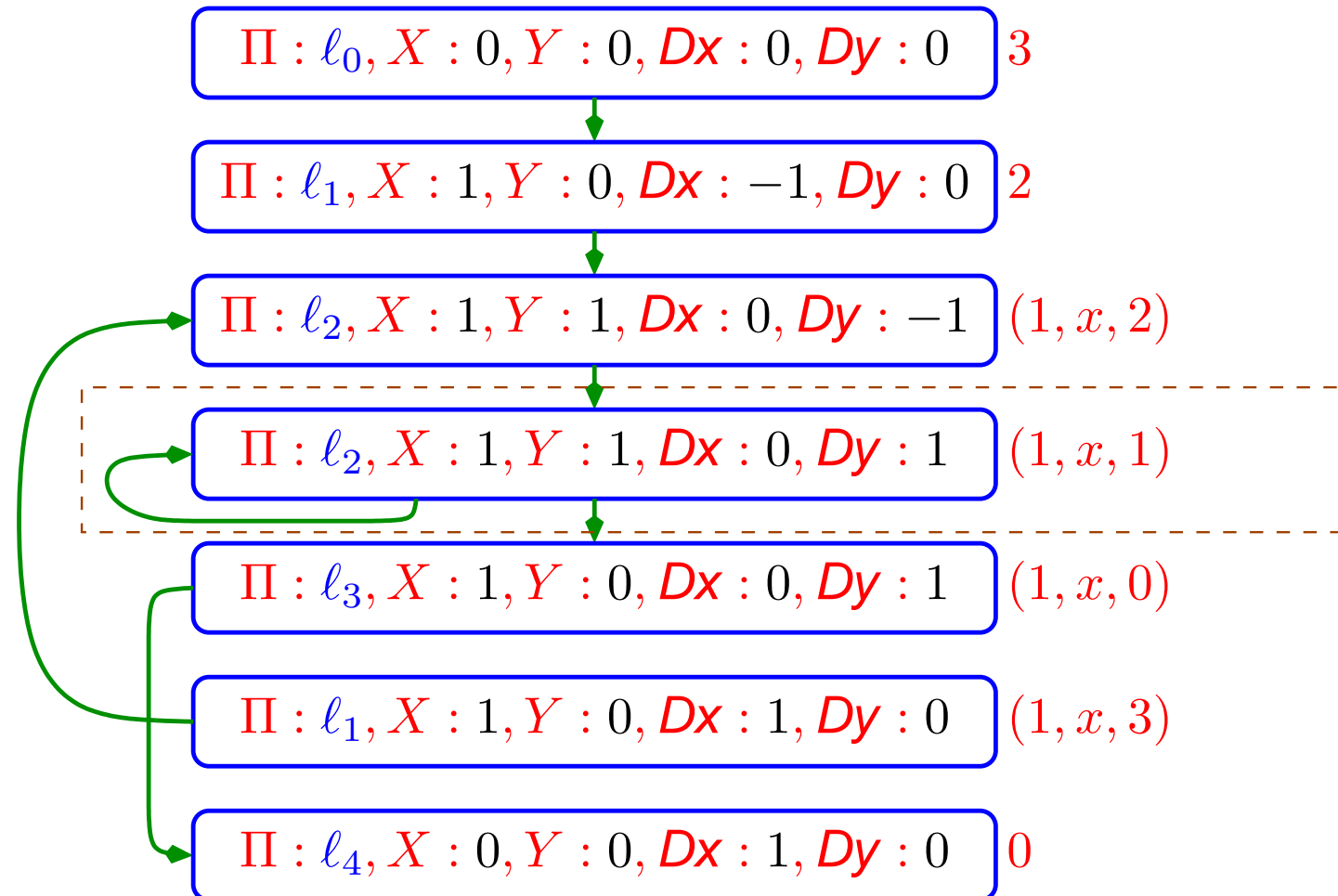
Add  $x$  to ranking, and remove edges entering  $(Dx > 0)$ -nodes.



Note that component is no longer strongly connected.

## Decompose, Sort, and Rank Subgraph

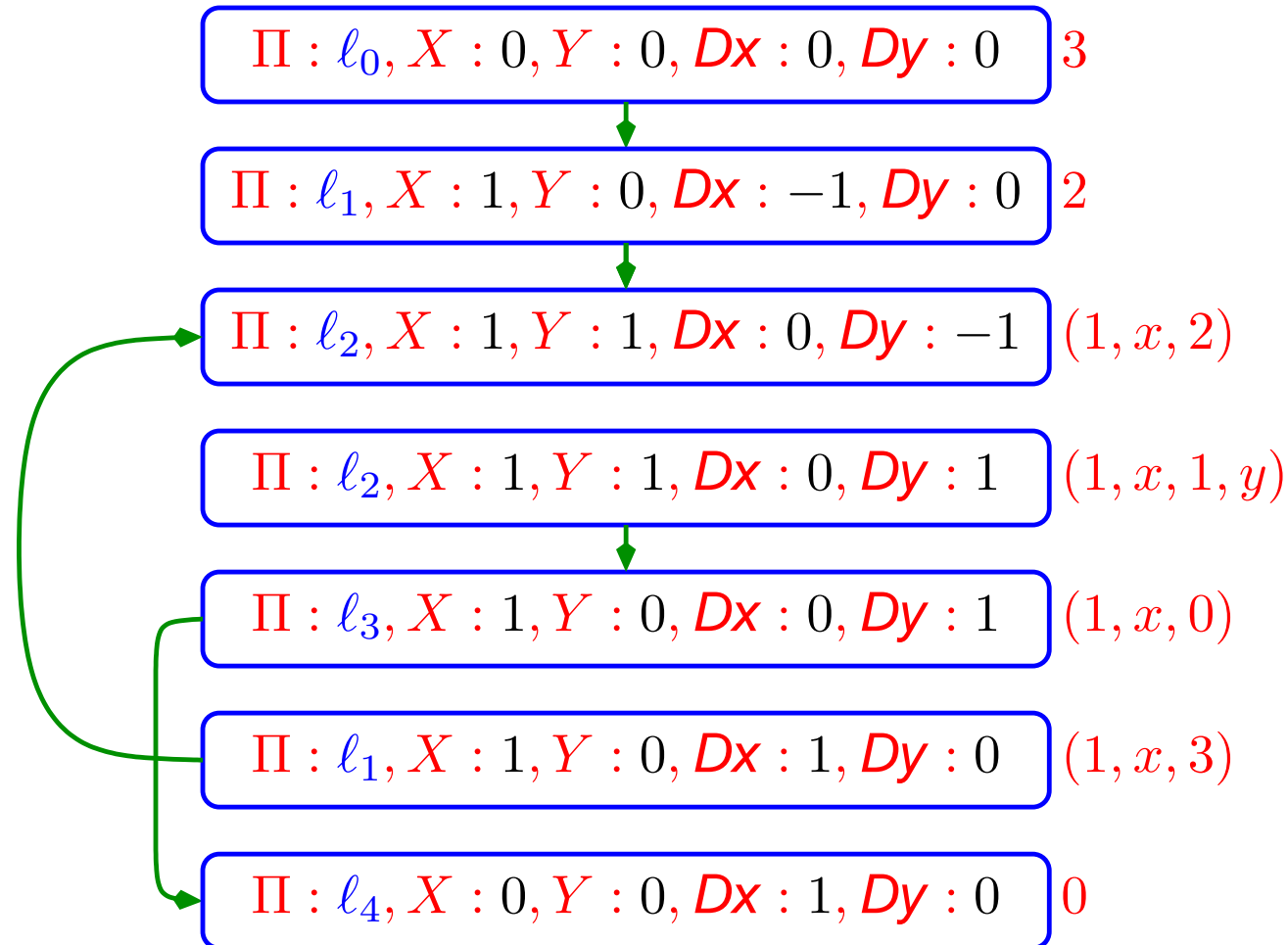
Applying the decomposition+ranking to the unraveled subgraph yields:



Note that the non-singular component is unfair w.r.t  $(Dy > 0, Dy < 0)$ .

## Add $y$ to the Ranking

Processing the  $\langle \Pi : \ell_2, X : 1, Y : 1, Dx : 0, Dy : 1 \rangle$  component, we add  $y$  to its ranking and remove all incoming edges. This yields:

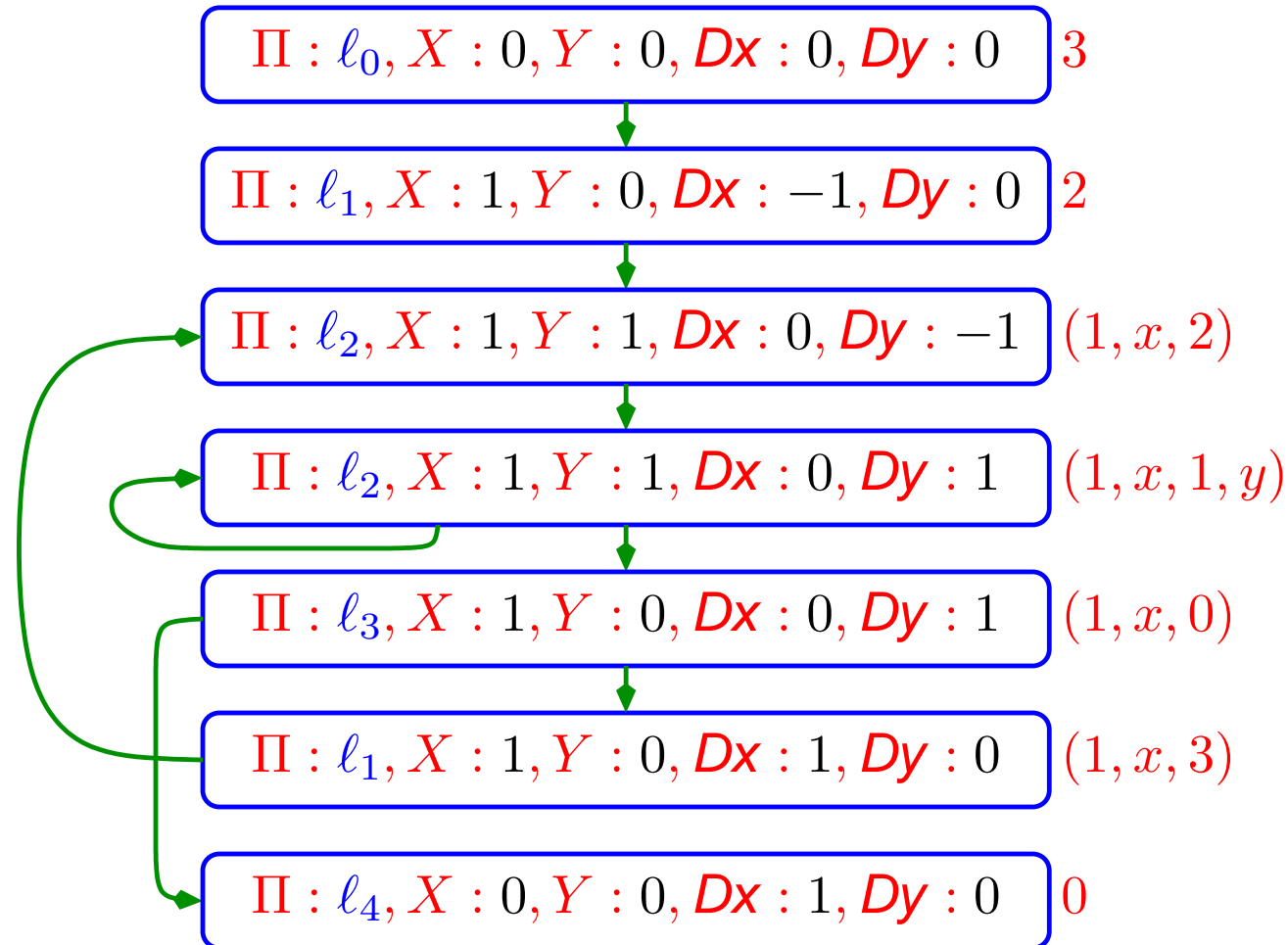


The resulting graph is acyclic, implying that the algorithm terminated.



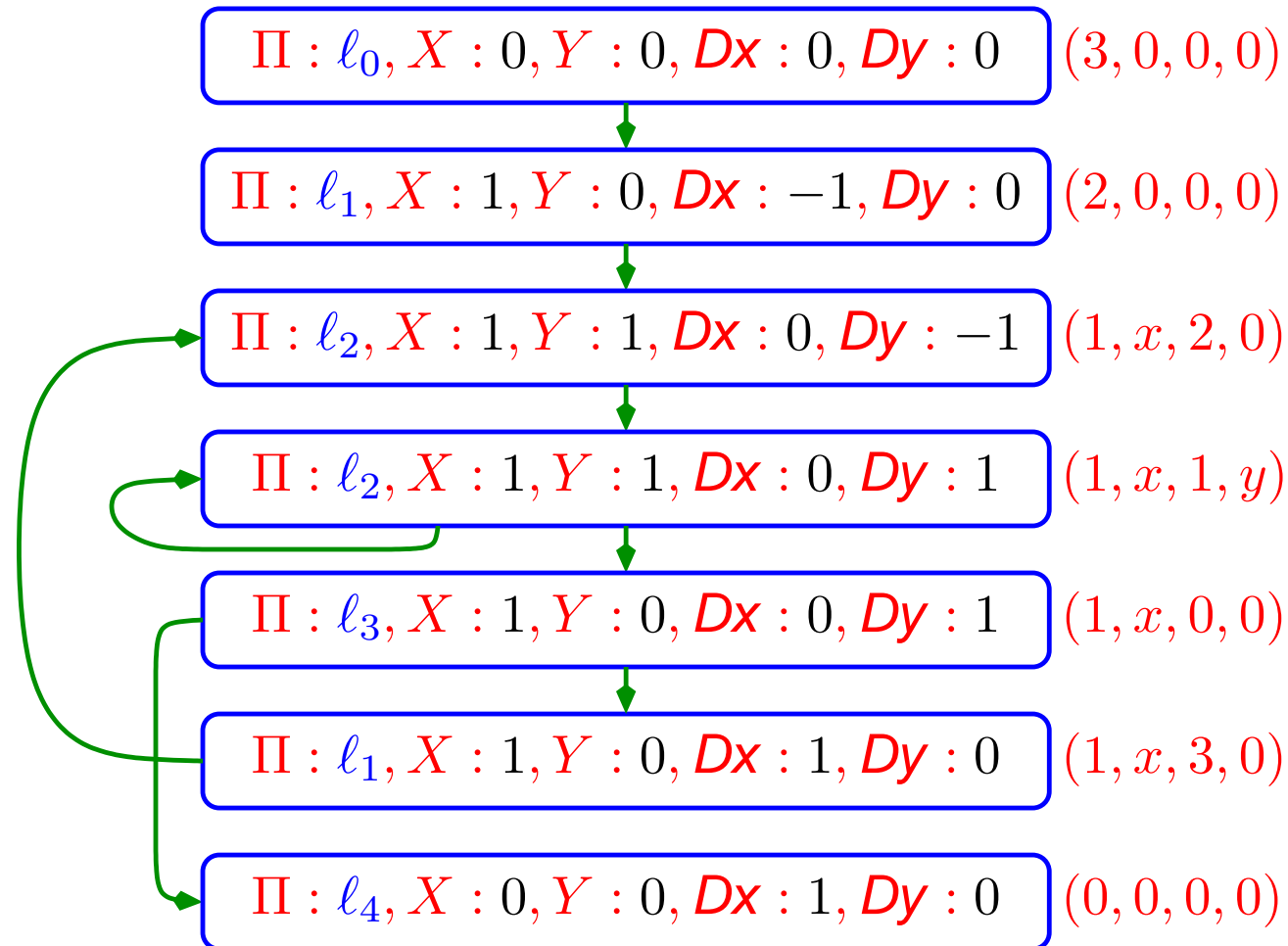
## The Final Global Ranking

Summarizing all that was accumulated, yields the following global ranking:



## Padding to the Right

If necessary, we can make all tuples to be of length 4, by adding zeros to the right.



## Conclusions

- **Ranking abstraction** should be considered as an inseparable companion to **predicate abstraction**. Only their combination can verify the full set of **LTL** properties.
- We call upon implementors of **abstraction-based** software verification systems, such as **SLAM** and **BLAST**, to enhance the proving power of their systems by adding the component of **ranking abstraction**.
- Like predicate abstraction, **ranking abstraction** is easier to apply than its deductive counterpart, because it is sufficient to provide only the **constituents** and let the model checker figure out their right combination.
- We should not consider abstraction as **replacing deduction**, but rather as **complementing** and **enhancing** deduction.