Verification of Progress Properties

Amir Pnueli

New York University and Weizmann Institute of Sciences (Emeritus)

Onassis Lectures in Embedded Systems,
Crete, 24 July 2008

Joint work with

Ittai Balaban, Yonit Kesten, Lenore Zuck

Contradicting our Guest of Honor

Joseph claimed:

> We do not know how to construct correct software, in a systematic way similar to the ones used in other Engineering disciplines.

Contradicting our Guest of Honor

Joseph claimed:

> We do not know how to construct correct software, in a systematic way similar to the ones used in other Engineering disciplines.

My version:

- He is too modest!
- We do know how to construct correct software systematically, partly due to Joseph's important contribution.
- We are not ready to pay the price.
- Software is the discipline in which it is cheapest to construct badly designed artifacts. That art is very widely practiced.

Perhaps summarize as:

> We do not know how to construct correct software economically.

Taming Software Complexity

There are two ways to overcome software complexity:

- Composition — Infer properties of systems from the properties of their components. Constructively: Build correct systems from correct components.
- Abstraction — Verify the desired property over an abstracted (simpler) version of the program. Constructively: Start with a correct abstract version of the program and derive an effective implementation by a sequence of stepwise refinement transformations.

The methods are not disjoint. In compositional verification one uses abstraction in order to represent the environment of a component.

The Verification Problem

Given a system (program, HW design) S and a property φ, ascertain that

$$S \models \varphi$$

That is, all behaviors (executions) of S satisfy φ.

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 φ 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 these lectures are 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 ψ into a propositional LTL formula ψ^α.
- 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 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.

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. 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.

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

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

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
- 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.

A state is called feasible if it appears in some computation. There exists a (symbolic) algorithm which computes all the feasible states in a given FDS.

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 α to abstract p:

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

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

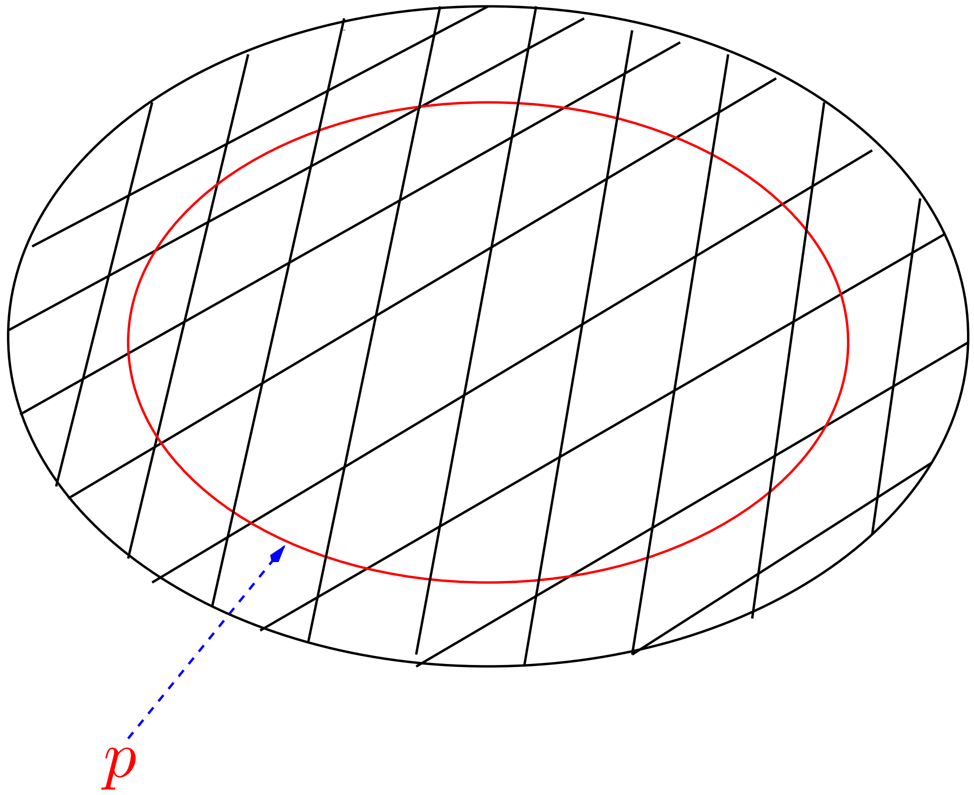
  An abstract state S belongs to $\|\overline{\alpha}(p)\|$ iff there exists some concrete state $s \in \alpha^{-1}(S)$ such that $s \in \|p\|$.

- The contracting abstraction (under approximation) is given by

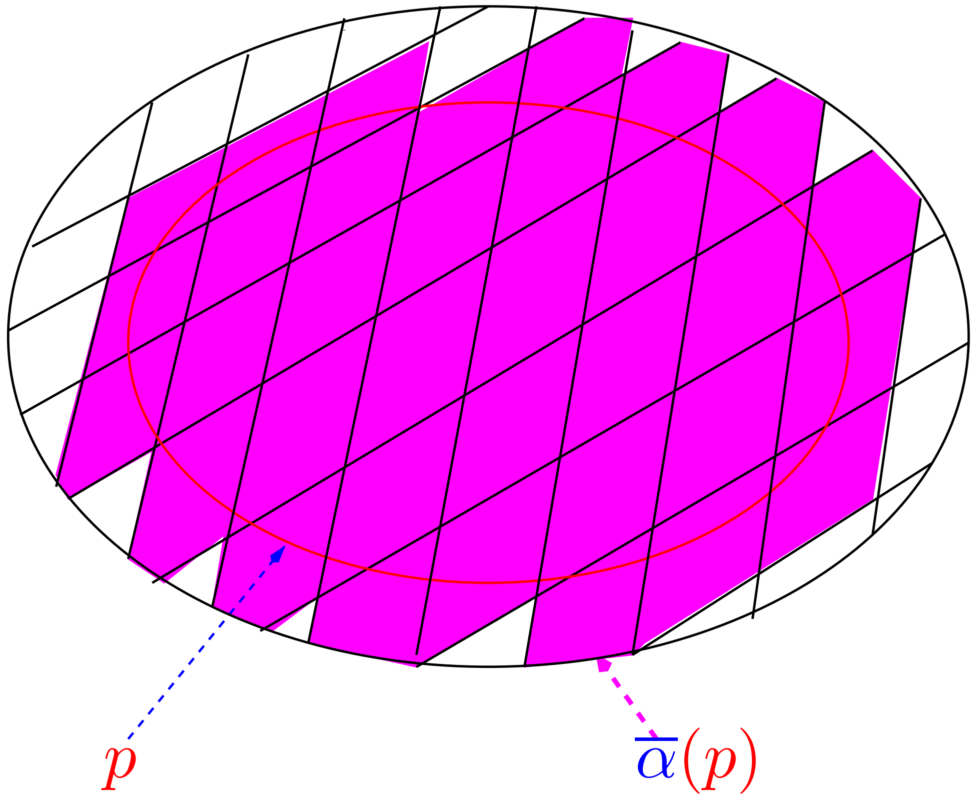
$$\underline{\alpha}(p): \quad \forall V : (V_A = \mathcal{E}_\alpha(V)) \;\rightarrow\; p(V) \qquad\qquad \|\underline{\alpha}(p)\| = \{S \mid \alpha^{-1}(S) \subseteq \|p\|\}$$

  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

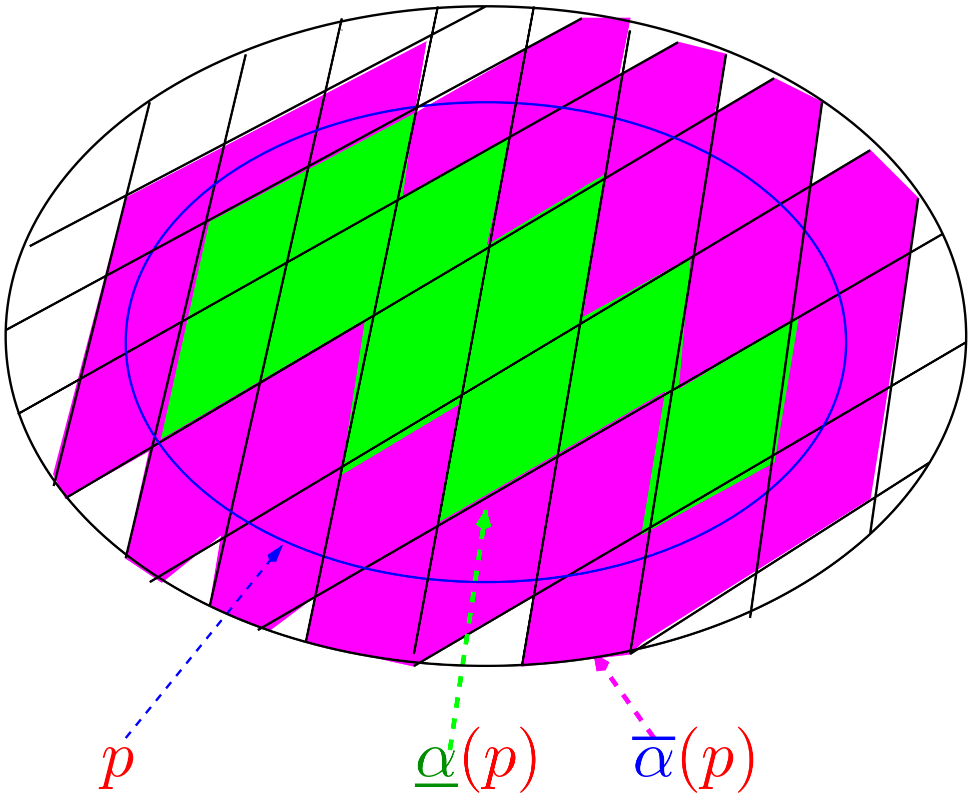


The Existential (expanding) Abstraction



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 Joint Abstraction

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

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

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

Rationale for Using Opposite Abstractions

In order to verify

$$\|\mathcal{D}\| \subseteq \|\psi\|$$

by abstraction, we actually prove

$$\|\mathcal{D}\| \subseteq \|\overline{\alpha}(\mathcal{D})\| \subseteq \|\underline{\alpha}(\psi)\| \subseteq \|\psi\|$$

Example: Program INCREASE

Consider the program

$$
\begin{array}{c}
y : \textbf{integer initially } y = 0 \\
\left[\begin{array}{ll}
\ell_0 : & \textbf{while } y \geq 0 \textbf{ do } \quad [\ell_1 : \ y := y + 1] \\
\ell_2 : &
\end{array}\right]
\end{array}
$$

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

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

The abstraction mapping α is specified by the defining expression:

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

where $\textit{sign}(y)$ 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 obtain the abstract version of INCREASE, called INCREASE$^\alpha$:

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

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

$$\psi^\alpha\colon \quad \diamondsuit\,\square\,(Y=+1),$$

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

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

A Case with No Conclusions

Reconsider program INCREASE, but this time with the property

$$\psi_2 : \square\,(0 \leq y \leq 10)$$

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

$$\psi_2^\alpha = \underline{\alpha}(\psi_2) : \quad \square\,(Y = 0)$$

Since INCREASE$^\alpha$ $\not\models$ $\square\,(Y = 0)$, we can draw no conclusions about program INCREASE and property ψ_2.

- Note that if, instead of $\underline{\alpha}(\psi_2)$, we would have taken

$$\overline{\alpha}(\psi_2) : \square\,(Y \in \{0, 1\}),$$

we would be led to the false conclusion

$$\text{INCREASE} \quad \models \quad \square\,(0 \leq y \leq 10)$$

Thus, it is essential to under-approximate the property.

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 $p_1, p_2, \ldots, p_k$ 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 ψ.

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

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

where $B_{p_1}, B_{p_2}, \ldots, B_{p_k}$ 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** $y_1 = y_2 = 0$

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

$$\|$$

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

The temporal properties for program BAKERY-2 are

ψ_{exc} : $\square\, \neg(at\_\ell_4 \ \wedge \ at\_m_4)$ $--$ Mutual Exclusion (safety)

ψ_{acc} : $\square\, (at\_\ell_2 \ \rightarrow \ \diamondsuit\, at\_\ell_4)$ $--$ Accessibility for P_1 (liveness)

Abstracting Program BAKERY-2

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

$$
\begin{array}{l}
\textbf{local } B_{y_1=0}, B_{y_2=0}, B_{y_1<y_2}\textbf{: boolean} \\
\qquad\qquad \textbf{where } B_{y_1=0} = B_{y_2=0} = 1, B_{y_1<y_2} = 0
\end{array}
$$

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

$$\|$$

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

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 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 1. [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 $\mathcal{D}^\alpha \models \square\, \underline{\alpha}(p)$.

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

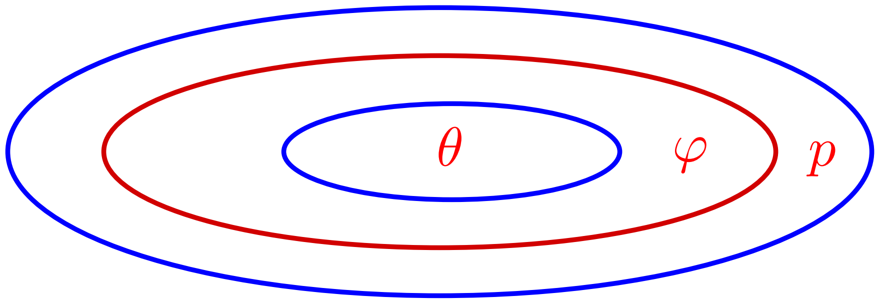
Sketch of the Proof

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

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

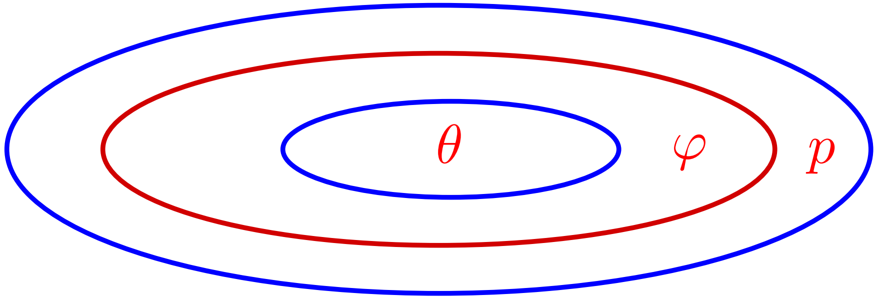
By the deductive theory of temporal verification, $\mathcal{D} \models \square\, p$ implies the existence of an assertion φ satisfying the following 3 premises:

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

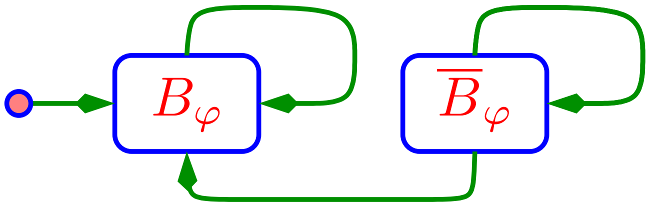


As the abstraction mapping, we take $\alpha : B_\varphi = \varphi$ using a single abstract boolean variable B_φ which is true whenever the corresponding concrete state satisfies φ. This leads to the following abstractions:

Proof Continued



The abstractions:

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

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

Inadequacy of State Abstraction for Proving Liveness

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

Consider the program LOOP.

$$
\begin{array}{l}
\qquad\qquad y\text{: } \textbf{natural} \\
\ell_0: \quad \textbf{while } y > 0 \textbf{ do} \\
\qquad\qquad \left[\begin{array}{ll} \ell_1: & y := y - 1 \\ \ell_2: & \textbf{skip} \end{array}\right] \\
\ell_3:
\end{array}
$$

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

$$
\begin{array}{l}
\qquad\qquad Y\text{: } \{0, +1\} \\
\ell_0: \quad \textbf{while } Y = +1 \textbf{ do} \\
\qquad\qquad \left[\begin{array}{ll} \ell_1: & Y := \textbf{if } Y = +1 \textbf{ then } \{0, +1\} \textbf{ else } 0 \\ \ell_2: & \textbf{skip} \end{array}\right] \\
\ell_3:
\end{array}
$$

This abstracted program may diverge!

No Finitary Abstracion Can Lead to Termination

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

Why?

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

Solution: Augment with a Non-Constraining Progress Monitor

$$y\text{: } \textbf{natural}$$

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

— LOOP — — MONITOR M_y —

Forming the cross product, we obtain:

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

Abstracting the Augmented System

We obtain the program

$$
\begin{array}{ll}
 & \begin{array}{lll} Y & : & \{0, +1\} \\ \textit{dec} & : & \{-1, 0, 1\} \\ \textbf{compassion} & & (\textit{dec} > 0, \textit{dec} < 0) \end{array} \\[2ex]
\ell_0: & \textbf{while } Y = +1 \textbf{ do} \\
 & \left[\begin{array}{llll} \ell_1: & (Y, \textit{dec}) & := & \left(\begin{array}{ll} \textbf{if} & Y = +1 \\ \textbf{then} & (\{+1, 0\}, 1) \\ \textbf{else} & (0, 0) \end{array}\right) \\[2ex] \ell_2: & \textit{dec} & := & 0 \end{array}\right] \\
\ell_3: &
\end{array}
$$

Which always terminates, due to the compassion requirement $(\textit{dec} > 0, \textit{dec} < 0)$.

A More Complicated Case

Sometimes we need a more complex progress measure:

$$
\begin{array}{l}
\quad\quad\quad\quad y\text{: } \textbf{natural} \\[6pt]
\ell_0 : \quad \textbf{while } y > 1 \textbf{ do} \\
\quad\quad\quad \left[\begin{array}{ll} \ell_1 : & y := y - 2 \\ \ell_2 : & y := \{y+1, y\} \\ \ell_3 : & \textbf{skip} \end{array}\right] \\
\ell_4 :
\end{array}
$$

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

$$
\begin{array}{ll}
\textbf{define} & \delta = y + \mathit{at\_}\ell_2 \\
\mathit{dec} & : \{-1, 0, 1\} \\
\textbf{compassion} & (\mathit{dec} > 0, \mathit{dec} < 0) \\[6pt]
m_0 : \quad \textbf{always do} & \\
\quad\quad \mathit{dec} := \mathit{sign}(\delta - \delta') &
\end{array}
$$

Complicated Case Continued

Augmenting and abstracting, we get:

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

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

where,

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

$$\mathit{add1}(Y) \quad = \quad \textbf{if } Y = 0 \textbf{ then } \mathit{one} \textbf{ else } \mathit{large}$$

This program always terminates

Verification by Ranking Abstraction

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

- Optionally choose one or more non-constraining progress monitors $M_1, \ldots, M_r$ and let $\mathcal{A} = \mathcal{D} \ ||| \ M_1 \ ||| \ \cdots \ ||| \ M_r$. In case this step is skipped, 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 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 α and a non-constraining progress monitor M, such that $\mathcal{A}^\alpha \models \psi^a$. Constructs α 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:

$$x, y\text{: } \textbf{natural}$$

$$\begin{array}{ll}
\ell_0: & x := ? \\
\ell_1: & \textbf{while } x > 0 \textbf{ do} \\
& \left[\begin{array}{ll}
\ell_2: & y := ? \\
\ell_3: & \textbf{while } y > 0 \textbf{ do} \\
& \left[\begin{array}{ll}
\ell_4: & y := y - 1 \\
\ell_5: & \textbf{skip}
\end{array}\right] \\
\ell_6: & x := x - 1 \\
\ell_7: & \textbf{skip}
\end{array}\right] \\
\ell_8: &
\end{array}$$

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

$$(\textit{at\_}\ell_0, \quad 5 \cdot x + 4 \cdot \textit{at\_}\ell_7 + 3 \cdot \textit{at\_}\ell_1 + 2 \cdot \textit{at\_}\ell_2 + \textit{at\_}\ell_{3..5}, \quad 3 \cdot y + 2 \cdot \textit{at\_}\ell_5 + \textit{at\_}\ell_3)$$

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:

$$\begin{array}{ll}
X, Y: : & \{0, +1\} \\
\mathit{decx}, \mathit{decy}: & \{-1, 0, 1\} \\
\textbf{compassion} & (\mathit{decx} > 0, \mathit{decx} < 0), \quad (\mathit{decy} > 0, \mathit{decy} < 0)
\end{array}$$

$$\begin{array}{l}
\ell_0: \ (X, Y, \mathit{decx}, \mathit{decy}) := (?, Y, ?, 0) \\
\ell_1: \ \textbf{while } X = +1 \textbf{ do} \\
\quad \left[\begin{array}{l}
\ell_2: \ (X, Y, \mathit{decx}, \mathit{decy}) := (X, ?, 0, ?) \\
\ell_3: \ \textbf{while } Y = +1 \textbf{ do} \\
\quad \left[\begin{array}{l}
\ell_4: \ (X, Y, \mathit{decx}, \mathit{decy}) := \left(\begin{array}{l} \textbf{if } Y = 0 \textbf{ then } (X, 0, 0, 0) \textbf{ else} \\ \{(X, +1, 0, 1), (X, 0, 0, 1)\} \end{array}\right) \\
\ell_5: \ \mathit{decy} := 0
\end{array}\right] \\
\ell_6: \ (X, Y, \mathit{decx}, \mathit{decy}) := \left(\begin{array}{l} \textbf{if } X = 0 \textbf{ then } (0, Y, 0, 0) \textbf{ else} \\ \{(+1, Y, 1, 0), (0, Y, 1, 0)\} \end{array}\right) \\
\ell_7: \ \mathit{decx} := 0;
\end{array}\right] \\
\ell_8:
\end{array}$$

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:

$$\begin{tikzcd} S_0 \arrow[r] & \bullet\bullet\bullet \arrow[r] & S_k \arrow[r] & \bullet\bullet\bullet \arrow[r] & S_n \arrow[ll, bend right] \end{tikzcd}$$

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

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

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.

$$\begin{tikzcd} S_0 \arrow[r] & \bullet\bullet\bullet \arrow[r] & S_k \arrow[r] & \bullet\bullet\bullet \arrow[r] & S_n \arrow[ll, bend right] \end{tikzcd}$$

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

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

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

Also form the assertion $\varphi_k = S_k[(p_1, \ldots, p_r)/(B_1, \ldots, B_r)]$ obtained by viewing abstract state S_k as a boolean expression over the abstract variables $B_1, \ldots, B_r$ 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 φ_k-states. Hence we search for a variable or an expression δ, such that

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

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

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

Example

Reconsider a version of program NESTED-LOOPS:

$$
\begin{array}{l}
\quad x, y\text{: } \textbf{natural initially } x = y = 0 \\
\ell_0 : \quad x :=? \\
\qquad \textbf{while } x > 0 \textbf{ do} \\
\qquad \left[\begin{array}{l}
\ell_1 : \quad y :=? \\
\qquad \textbf{while } y > 0 \textbf{ do} \\
\qquad [\ \ell_2 : \quad y := y - 1\ ] \\
\ell_3 : \quad x := x - 1
\end{array} \right] \\
\ell_4 :
\end{array}
$$

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 $\{X = \textit{sign}(x), Y = \textit{sign}(y), \textit{decy} = \textit{sign}(y - y')\}$, we obtain:

$$
\begin{array}{ll}
 & \begin{array}{ll}
X, Y: : & \{0, +1\} \textbf{ initially } X = Y = 0 \\
\textit{decy} : & \{-1, 0, 1\} \textbf{ initially } \textit{decy} = 0 \\
\textbf{compassion} & (\textit{decy} > 0, \textit{decy} < 0)
\end{array} \\
\ell_0 : & X := \{0, 1\} \\
 & \textbf{while } X = 1 \textbf{ do} \\
 & \left[\begin{array}{ll}
\ell_1 : & (Y, \textit{decy}) := (?, ?) \\
 & \textbf{while } Y = 1 \textbf{ do} \\
 & [\ell_2 : \ (Y, \textit{decy}) := \textbf{if } Y = 0 \textbf{ then } (0, 0) \textbf{ else } \{(0, 1), (1, 1)\}] \\
\ell_3 : & X := \textbf{if } X = 0 \textbf{ then } 0 \textbf{ else } \{0, 1\}
\end{array}\right] \\
\ell_4 : &
\end{array}
$$

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

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

Concretizing and Refining

Concretizing the abstract trace

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

we obtain:

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

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

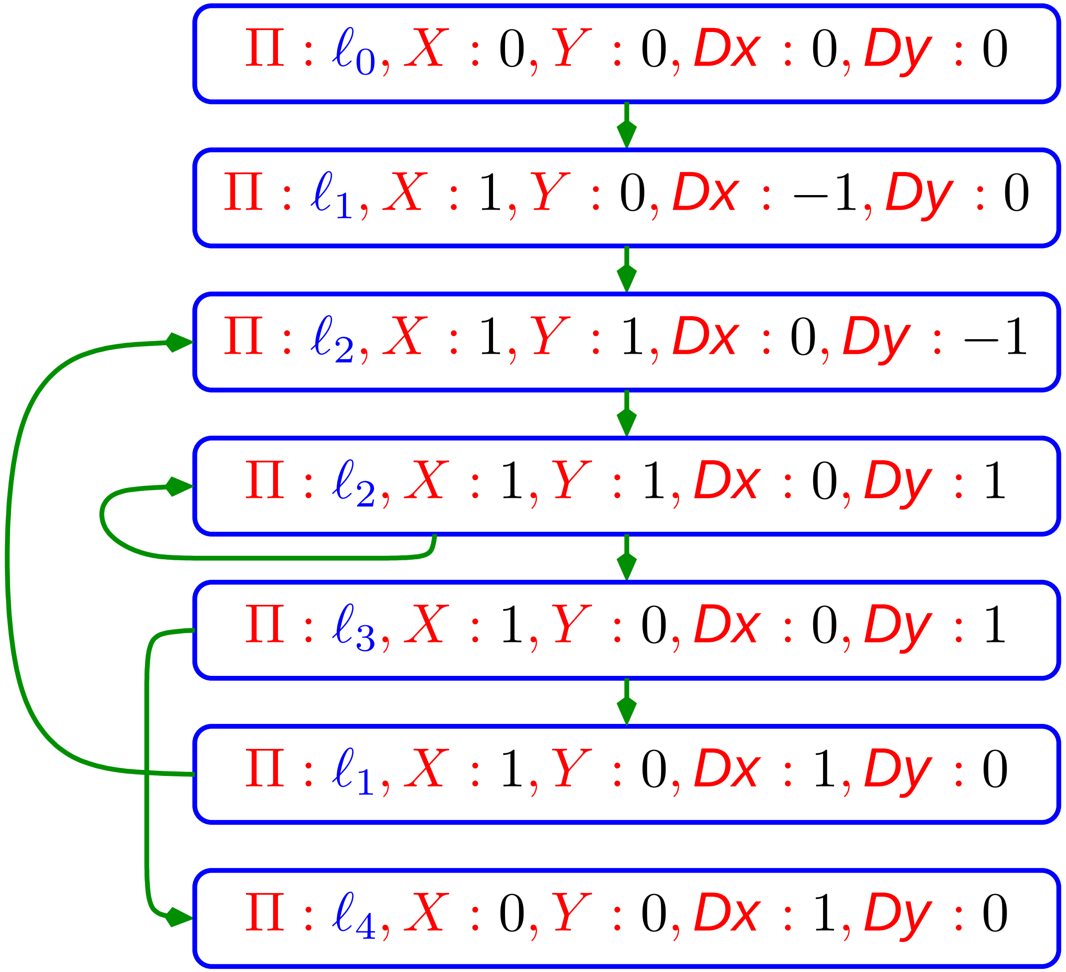
We will show how to extract a global ranking function from an abstract terminating program. Assume that we constructed a state-transition graph containing all the reachable states of the abstracted program.

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. $(\mathit{decx}_i > 0, \mathit{decx}_i < 0)$ violated by the component.
  - Add x_i to the ranking tuple.
  - Remove all edges entering $(\mathit{decx}_i > 0)$-nodes.
  - Return to top for recursive processing of remaining subgraph.

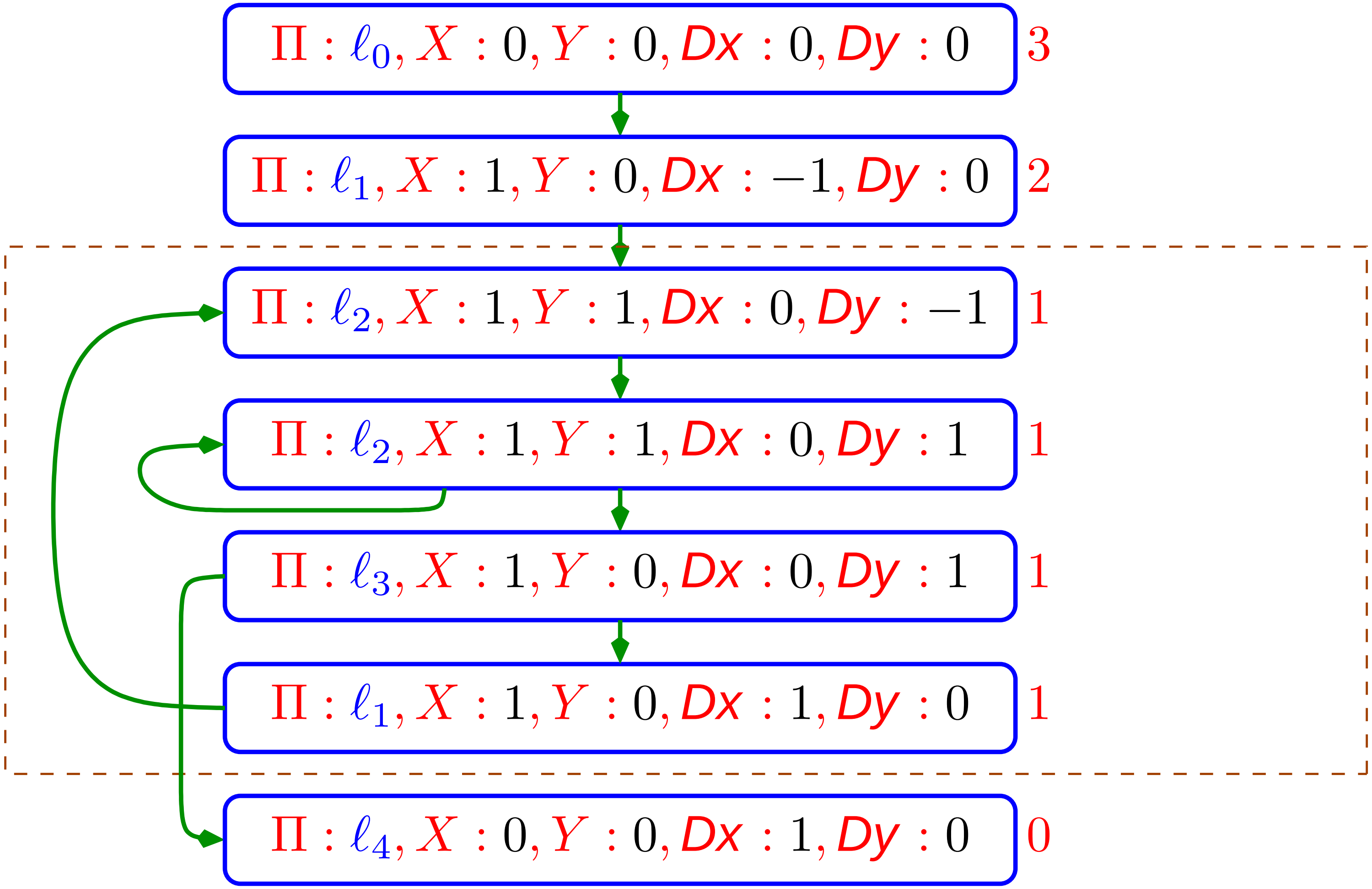
Example

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

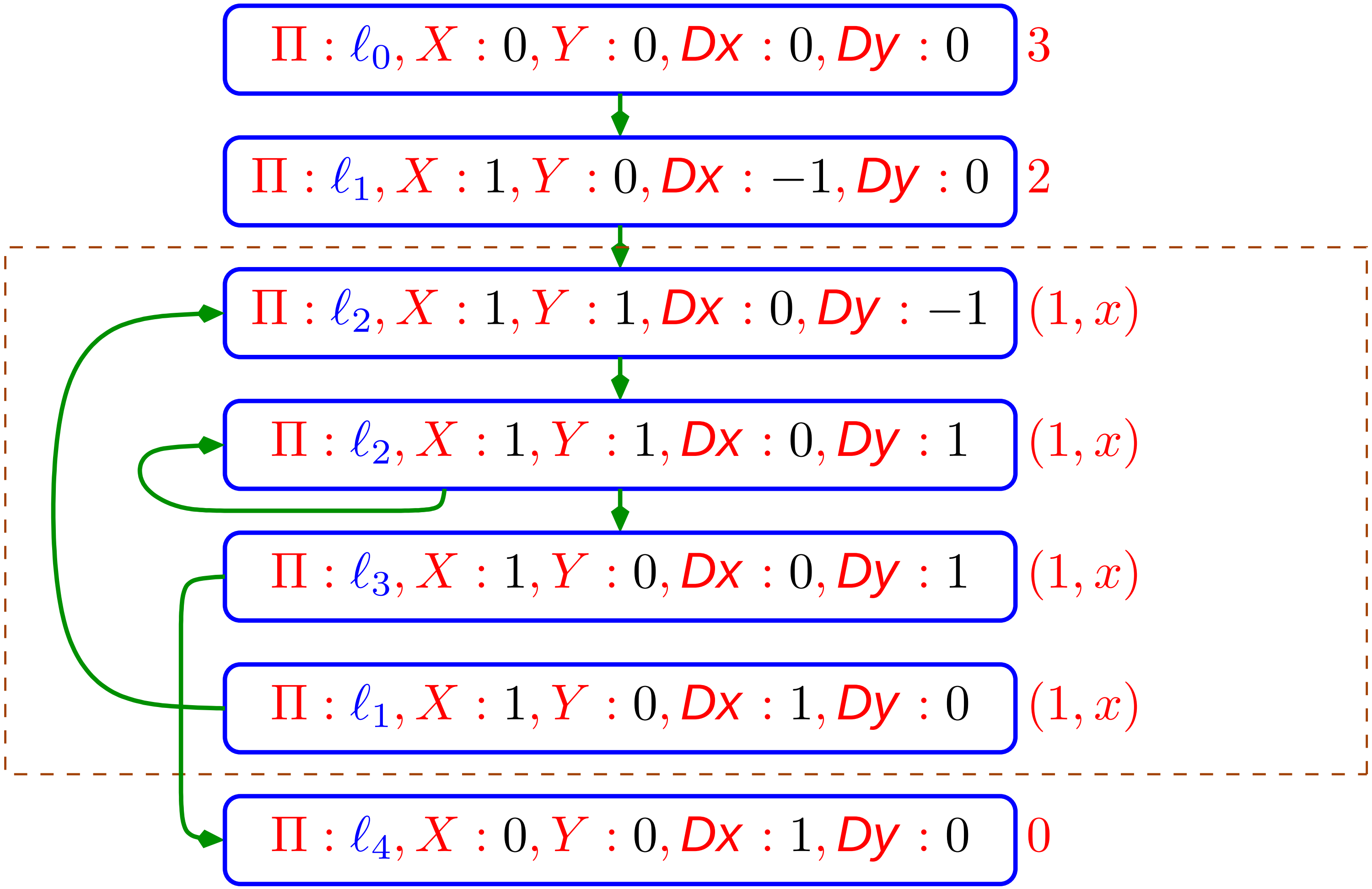
MSCC's decomposition, topologically sorting, and sequentially ranking, yields:



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

Add x to Ranking

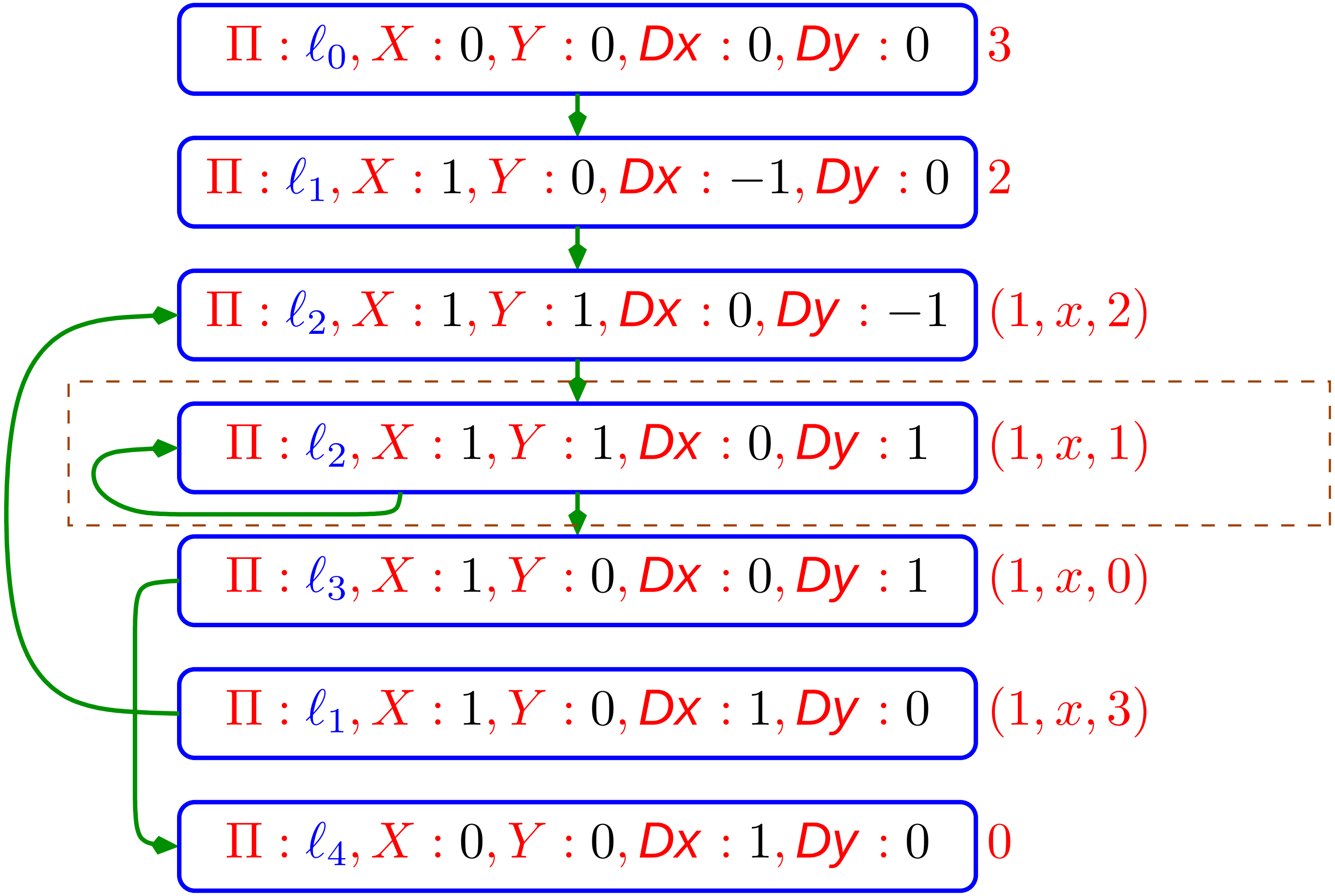
Add x to ranking, and remove edges entering $(Dx > 0)$-nodes.



Note that component is no longer strongly connected.

Decompose, Sort, and Rank Subgraph

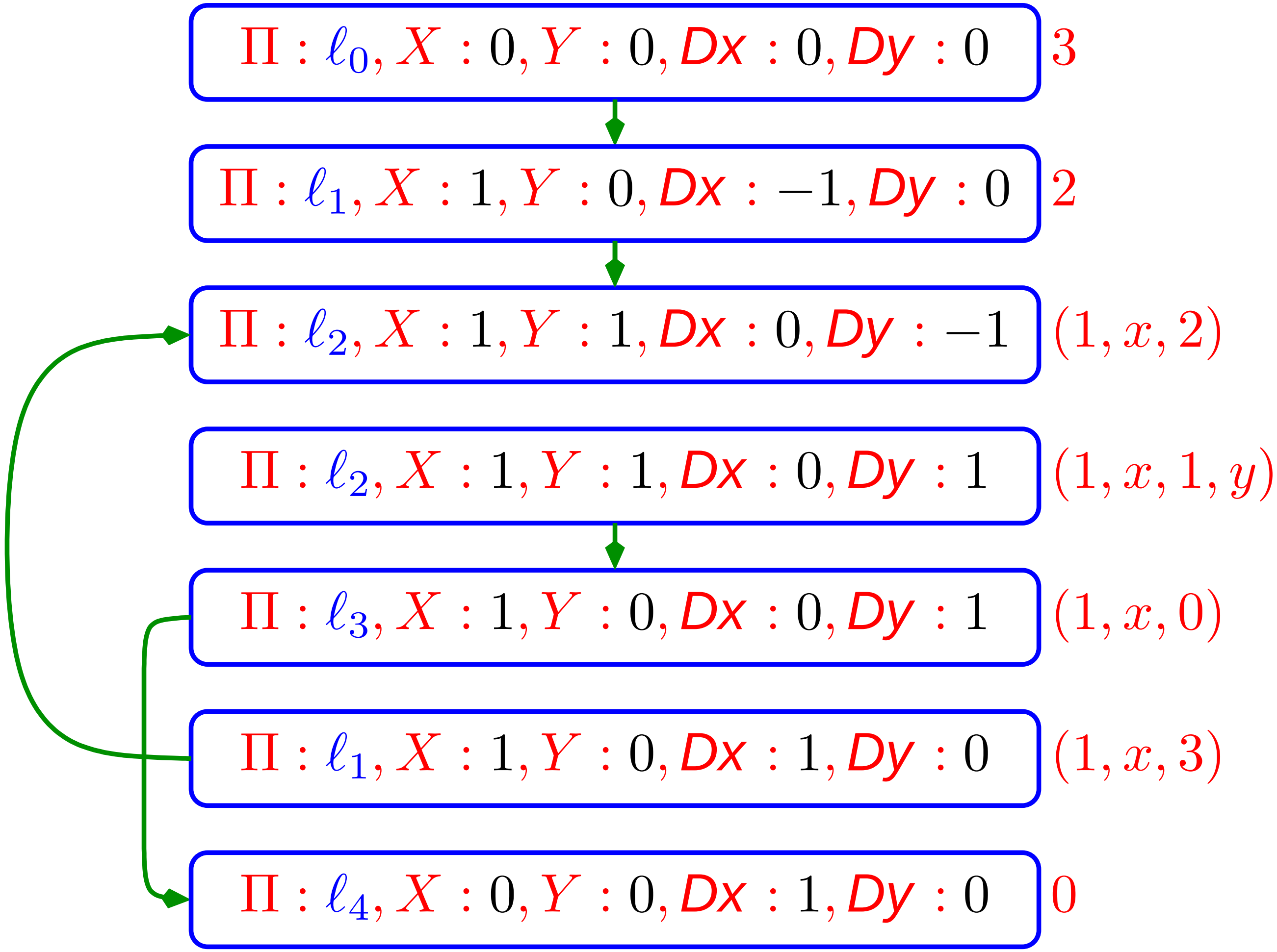
Applying the decomposition+ranking to the unraveled subgraph yields:



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:

$\Pi : \ell_0, X : 0, Y : 0, Dx : 0, Dy : 0$	3
$\Pi : \ell_1, X : 1, Y : 0, Dx : -1, Dy : 0$	2
$\Pi : \ell_2, X : 1, Y : 1, Dx : 0, Dy : -1$	$(1, x, 2)$
$\Pi : \ell_2, X : 1, Y : 1, Dx : 0, Dy : 1$	$(1, x, 1, y)$
$\Pi : \ell_3, X : 1, Y : 0, Dx : 0, Dy : 1$	$(1, x, 0)$
$\Pi : \ell_1, X : 1, Y : 0, Dx : 1, Dy : 0$	$(1, x, 3)$
$\Pi : \ell_4, X : 0, Y : 0, Dx : 1, Dy : 0$	0

Padding to the Right

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

$\Pi : \ell_0, X : 0, Y : 0, Dx : 0, Dy : 0 \quad (3, 0, 0, 0)$

$\Pi : \ell_1, X : 1, Y : 0, Dx : -1, Dy : 0 \quad (2, 0, 0, 0)$

$\Pi : \ell_2, X : 1, Y : 1, Dx : 0, Dy : -1 \quad (1, x, 2, 0)$

$\Pi : \ell_2, X : 1, Y : 1, Dx : 0, Dy : 1 \quad (1, x, 1, y)$

$\Pi : \ell_3, X : 1, Y : 0, Dx : 0, Dy : 1 \quad (1, x, 0, 0)$

$\Pi : \ell_1, X : 1, Y : 0, Dx : 1, Dy : 0 \quad (1, x, 3, 0)$

$\Pi : \ell_4, X : 0, Y : 0, Dx : 1, Dy : 0 \quad (0, 0, 0, 0)$

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.
- Never pay too much attention to completeness theorems. They may provide a misleading view of the usefulness of a method.