Traces Properties
Semantics and applications to verification

Xavier Rival
École Normale Supérieure

February 22, 2019
Program of this lecture

Goal of verification

Prove that $\lbrack P \rbrack \subseteq S$
(i.e., all behaviors of $P$ satisfy specification $S$)
where $\lbrack P \rbrack$ is the program semantics and $S$ the desired specification

Last week, we studied a form of $\lbrack P \rbrack$...

Today’s lecture: we look back at program’s properties

- **families of properties:**
  what properties can be considered “similar” ? in what sense ?

- **proof techniques:**
  how can those kinds of properties be established ?

- **specification of properties:**
  are there languages to describe properties ?
In this lecture we look at **trace properties**

A property is **a set of traces**, defining the **admissible** executions

**Safety properties:**
- something (e.g., bad) will never happen
- proof by invariance

**Liveness properties:**
- something (e.g., good) will eventually happen
- proof by variance

**Beyond safety and liveness:** **hyperproperties** (e.g., security...)**
State properties

As usual, we consider $S = (\Sigma, \rightarrow, S_I)$

First approach: properties as sets of states

- A property $P$ is a set of states $P \subseteq S$
- $P$ is satisfied if and only if all reachable states belong to $P$, i.e.,
  
  $$ [S]_R \subseteq P \text{ where } [S]_R = \{ s_n \in S \mid \exists \langle s_0, \ldots, s_n \rangle \in [S]^*, s_0 \in S_I \} $$

Examples:

- Absence of runtime errors:
  
  $$ P = \Sigma \setminus \{ \Omega \} \text{ where } \Omega \text{ is the error state} $$

- Non termination (e.g., for an operating system):
  
  $$ P = \{ s \in \Sigma \mid \exists s' \in \Sigma, s \rightarrow s' \} $$
Second approach: properties as sets of traces

- A property $\mathcal{T}$ is a set of traces $\mathcal{T} \subseteq \mathcal{S}^\infty$
- $\mathcal{T}$ is satisfied if and only if all traces belong to $\mathcal{T}$, i.e., $[\mathcal{S}]^\infty \subseteq \mathcal{T}$

Examples:

- Obviously, state properties are trace properties
- Functional properties:
  e.g., “program $P$ takes one integer input $x$ and returns its absolute value”
- Termination: $\mathcal{T} = \mathcal{S}^*$ (i.e., the system should have no infinite execution)
Monotonicity

**Property 1**

Let $\mathcal{P}_0, \mathcal{P}_1 \subseteq \mathcal{S}$ be two state properties, such that $\mathcal{P}_0 \subseteq \mathcal{P}_1$. Then $\mathcal{P}_0$ is stronger than $\mathcal{P}_1$, i.e. if program $S$ satisfies $\mathcal{P}_0$, then it also satisfies $\mathcal{P}_1$.

**Property 2**

Let $\mathcal{T}_0, \mathcal{T}_1 \subseteq \mathcal{S}$ be two trace properties, such that $\mathcal{T}_0 \subseteq \mathcal{T}_1$. Then $\mathcal{T}_0$ is stronger than $\mathcal{T}_1$, i.e. if program $S$ satisfies $\mathcal{T}_0$, then it also satisfies $\mathcal{T}_1$.

**Property 3**

Let $S_0, S_1$ two transition systems, such that $S_1$ has more behaviors than $S_0$ (i.e., $[S_0] \subseteq [S_1]$), and $\mathcal{P}$ be a (trace or state) property. Then, if $S_1$ satisfies $\mathcal{P}$, so does $S_0$.

**Proofs:** straightforward application of the definitions
Outline

1. Safety properties
   - Informal and formal definitions
   - Proof method

2. Liveness properties

3. Decomposition of trace properties

4. A Specification Language: Temporal logic

5. Beyond safety and liveness

6. Conclusion
Safety properties

Informal definition: safety properties

A safety property is a property which specifies that some (bad) behavior will never occur, at any time

- **Absence of runtime errors** is a safety property ("bad thing": error)
- **State properties** is a safety property ("bad thing": reaching $S \setminus \mathcal{P}$)
- **Non termination** is a safety property ("bad thing": reaching a blocking state)
- "**Not reaching state $b$ after visiting state $a$**" is a safety property (and **not** a state property)
- **Termination** is **not** a safety property

We now intend to provide a **formal definition** of safety.
Towards a formal definition

How to refute a safety property?

- We assume $S$ does **not** satisfy safety property $P$
- Thus, there exists a **counter-example trace**
  $\sigma = \langle s_0, \ldots, s_n, \ldots \rangle \in \llbracket S \rrbracket \setminus P$;
  it may be finite or infinite...
- The intuitive definition says this trace **eventually exhibits some bad behavior**, at some given time, corresponding to some index $i$
- Therefore, trace $\sigma' = \langle s_0, \ldots, s_i \rangle$ violates $P$, i.e. $\sigma' \notin P$
- We remark $\sigma'$ is finite

---

A safety property that does not hold can always be refuted with a finite counter-example
A Few Operators on Traces

**Prefix:** We write $\sigma_i$ for the prefix of length $i$ of trace $\sigma$:

\[
\begin{align*}
\langle s_0, \ldots, s_n \rangle_i & = \epsilon \\
\langle s_0, \ldots, s_n \rangle_{i+1} & = \begin{cases} 
\langle s_0, \ldots, s_i \rangle & \text{if } i < n \\
\langle s_0, \ldots, s_n \rangle & \text{otherwise}
\end{cases} \\
\langle s_0, \ldots \rangle_{i+1} & = \langle s_0, \ldots, s_i \rangle
\end{align*}
\]

**Suffix** (or tail):

\[
\begin{align*}
\sigma_i & = \epsilon & \text{if } |\sigma| < i \\
(\langle s_0, \ldots, s_i \rangle \cdot \sigma)_{i+1} & ::= \sigma & \text{otherwise}
\end{align*}
\]
Upper closure operators

Definition: upper closure operator (uco)

We consider a preorder \((S, \sqsubseteq)\). Function \(\phi : S \rightarrow S\) is an upper closure operator iff:

- **monotone**
- **extensive**: \(\forall x \in S, \ x \sqsubseteq \phi(x)\)
- **idempotent**: \(\forall x \in S, \ \phi(\phi(x)) = \phi(x)\)

**Dual**: lower closure operator, monotone, reductive, idempotent

**Examples**:
- on real/decimal numbers, or on fraction: the **ceiling** operator, that returns the next integer is an upper-closure operator
Prefix closure

Definition: prefix closure

The prefix closure operator is defined by:

\[\text{PCl} : \mathcal{P}(S^\infty) \rightarrow \mathcal{P}(S^*)\]

\[X \mapsto \{\sigma_i \mid \sigma \in X, i \in \mathbb{N}\}\]

Example: assuming

\[S = \{\langle a, b, c \rangle, \langle a, c \rangle\}\]

then,

\[\text{PCl}(S) = \{\epsilon, \langle a \rangle, \langle a, b \rangle, \langle a, b, c \rangle, \langle a, c \rangle\}\]

Properties:

- \(\text{PCI}\) is monotone
- \(\text{PCI}\) is idempotent, i.e., \(\text{PCI} \circ \text{PCI}(X) = \text{PCI}(X)\)
- \(\text{PCI}\) is not extensive (infinite traces do not appear anymore)
Limit

**Definition: limit**

The *limit operator* is defined by:

\[
\text{Lim} : \mathcal{P}(S^\infty) \rightarrow \mathcal{P}(S^\infty)
\]

\[
X \mapsto X \cup \{\sigma \in S^\infty \mid \forall i \in \mathbb{N}, \sigma_{\lceil i \rceil} \in X\}
\]

Operator \textbf{Lim} is an upper-closure operator

**Proof**: exercise!

**Example**: assuming

\[
S = \{ \varepsilon, \langle a \rangle, \langle a, b \rangle, \langle a, b, a \rangle, \langle a, b, a, b \rangle, \ldots \}
\]

then,

\[
\text{Lim}(S) = S \uplus \{\langle a, b, a, b, a, b, \ldots \rangle\}
\]
Towards a formal definition for safety

**Operator Safe**

Operator **Safe** is defined by $\text{Safe} = \lim \circ \text{PCI}$.

Operator **Safe saturates** a set of traces $S$ with
- prefixes
- infinite traces all finite prefixes of which can be observed in $S$

Thus, if $\text{Safe}(S) = S$ and $\sigma$ is a trace, to establish that $\sigma$ is not in $S$, it is sufficient to discover a **finite prefix of $\sigma$** that cannot be observed in $S$.
- if $\sigma$ is finite the result is clear (consider $\sigma$)
- otherwise, if all finite prefixes of $\sigma$ are in $S$, then $\sigma$ is in the limit, thus in $S$.

**Safety: definition**

A trace property $\mathcal{T}$ is a **safety** property if and only if $\text{Safe}(\mathcal{T}) = \mathcal{T}$
Safety properties: formal definition

An upper closure operator

Operator Safe is an upper closure operator over $\mathcal{P}(S^\infty)$

Proof:

**Safe is monotone** since Lim and PCI are monotone

**Safe is extensive:**
indeed if $X \subseteq S^\infty$ and $\sigma \in X$, we can show that $\sigma \in Safe(X)$:

- if $\sigma$ is a finite trace, it is one of its prefixes, so $\sigma \in PCI(X) \subseteq Lim(PCI(X))$
- if $\sigma$ is an infinite trace, all its prefixes belong to PCI($X$), so $\sigma \in Lim(PCI(X))$
Safety properties: formal definition

Proof (continued):

Safe is idempotent:

- as Safe is extensive and monotone Safe \(\subseteq\) Safe \(\circ\) Safe, so we simply need to show that Safe \(\circ\) Safe \(\subseteq\) Safe

- let \(X \subseteq S^\infty, \sigma \in \text{Safe}(\text{Safe}(X))\); then:

\[
\sigma \in \text{Safe}(\text{Safe}(X)) \\
\Rightarrow \forall i, \sigma_{\lceil i} \in \text{PCI} \circ \text{Safe}(X) \quad \text{by def. of Lim} \\
\Rightarrow \forall i, \exists \sigma', j, \sigma_{\lceil i} = \sigma'_{\lceil j} \land \sigma' \in \text{Safe}(X) \quad \text{by def. of PCI} \\
\Rightarrow \forall i, \exists \sigma', j, \sigma_{\lceil i} = \sigma'_{\lceil j} \land \forall k, \sigma'_{\lceil k} \in \text{PCI}(X) \quad \text{by def. of Lim and case analysis over finiteness of } \sigma' \\
\Rightarrow \forall i, \exists \sigma', j, \sigma_{\lceil i} = \sigma'_{\lceil j} \land \sigma'_{\lceil j} \in \text{PCI}(X) \quad \text{if we take } k = j \\
\Rightarrow \forall i, \sigma_{\lceil i} \in \text{PCI}(X) \quad \text{by simplification} \\
\Rightarrow \sigma \in \text{Lim} \circ \text{PCI}(X) \quad \text{by def. of Lim} \\
\Rightarrow \sigma \in \text{Safe}(X)
\]
### Safety properties: formal definition

#### Safety: definition

A trace property $\mathcal{T}$ is a **safety** property if and only if $\text{Safe}(\mathcal{T}) = \mathcal{T}$

#### Theorem

If $\mathcal{T}$ is a trace property, then $\text{Safe}(\mathcal{T})$ is a safety property

#### Proof:

Straightforward, by idempotence of $\text{Safe}$

#### Intuition:

- if $\mathcal{T}$ is a trace property (not necessarily a safety property), $\text{Safe}(\mathcal{T})$ is the strongest safety property, that is weaker than $\mathcal{T}$
- at this point, this observation is not so useful... but it will be soon!
Example

We assume that:

- $S = \{a, b\}$
- $T$ states that *a should not be visited after state b is visited*; elements of $T$ are of the general form
  \[\langle a, a, a, \ldots, a, b, b, b, b, \ldots \rangle \text{ or } \langle a, a, a, \ldots, a, a, a, \ldots \rangle\]

Then:

- $\text{PCI}(T)$ elements are all finite traces which are of the above form (i.e., made of $n$ occurrences of $a$ followed by $m$ occurrences of $b$, where $n, m$ are positive integers)
- $\text{Lim}(\text{PCI}(T))$ adds to this set the trace made made of infinitely many occurrences of $a$ and the infinite traces made of $n$ occurrences of $a$ followed by infinitely many occurrences of $b$
- thus, $\text{Safe}(T) = \text{Lim}(\text{PCI}(T)) = T$

Therefore $T$ is indeed formally a safety property.
State properties are safety properties

Theorem

Any state property is also a safety property.

Proof:

Let us consider state property $\mathcal{P}$.
It is equivalent to trace property $\mathcal{T} = \mathcal{P}^\infty$:

$$\text{Safe}(\mathcal{T}) = \lim(\text{PCI}(\mathcal{P}^\infty))$$
$$= \lim(\mathcal{P}^*)$$
$$= \mathcal{P}^* \cup \mathcal{P}^\omega$$
$$= \mathcal{P}^\infty$$
$$= \mathcal{T}$$

Therefore $\mathcal{T}$ is indeed a safety property.
Intuition of the formal definition

Operator **Safe saturates** a set of traces $S$ with

- prefixes
- infinite traces all finite prefixes of which can be observed in $S$

Thus, if $\text{Safe}(S) = S$ and $\sigma$ is a trace, to establish that $\sigma$ is not in $S$, it is sufficient to discover a **finite prefix of $\sigma$** that cannot be observed in $S$.

Alternatively, if all finite prefixes of $\sigma$ belong to $S$ or can observed as a prefix of another trace in $S$, by definition of the limit operator, $\sigma$ **belongs to $S$** (even if it is infinite).

Thus, our definition **indeed captures properties that can be disproved with a finite counter-example**.
Outline

1 Safety properties
   - Informal and formal definitions
   - Proof method

2 Liveness properties

3 Decomposition of trace properties

4 A Specification Language: Temporal logic

5 Beyond safety and liveness

6 Conclusion
Proof by invariance

- We consider transition system $\mathcal{S} = (\mathcal{S}, \rightarrow, \mathcal{S}_I)$, and safety property $\mathcal{T}$. Finite traces semantics is the least fixpoint of $F_\ast$.
- We seek a way of verifying that $\mathcal{S}$ satisfies $\mathcal{T}$, i.e., that $[\mathcal{S}]^\infty \subseteq \mathcal{T}$

Principle of invariance proofs

Let $\mathcal{I}$ be a set of finite traces; it is said to be an invariant if and only if:

- $\forall s \in \mathcal{S}_I, \langle s \rangle \in \mathcal{I}$
- $F_\ast(\mathcal{I}) \subseteq \mathcal{I}$

It is stronger than $\mathcal{T}$ if and only if $\mathcal{I} \subseteq \mathcal{T}$.

The "by invariance" proof method is based on finding an invariant that is stronger than $\mathcal{T}$. 
Soundness

Theorem: soundness

The invariance proof method is **sound**: if we can find an invariant for $S$, that is stronger than safety property $T$, then $S$ satisfies $T$.

**Proof:**
We assume that $\comprehend{s} \in S_I$ and that it is stronger than $T$, and we show that $S$ satisfies $T$:

- by induction over $n$, we can prove that $F^*_n(\comprehend{s} \mid s \in S_I) \subseteq F^*_n(\comprehend{\comprehend{s}}) \subseteq \comprehend{\comprehend{s}}$
- therefore $[S]^* \subseteq \comprehend{\comprehend{s}}$
- thus, $\safe([S]^*) \subseteq \safe(\comprehend{\comprehend{s}}) \subseteq \safe(T)$ since $\safe$ is monotone
- we remark that $[S]^{\propto} = \safe([S]^*)$
- $T$ is a safety property so $\safe(T) = T$
- we conclude $[S]^{\propto} \subseteq T$, i.e., $S$ satisfies property $T$
Completeness

**Theorem: completeness**

The invariance proof method is **complete**: if $S$ satisfies safety property $\mathcal{T}$, then we can find an invariant $I$ for $S$, that is stronger than $\mathcal{T}$.

**Proof:**

We assume that $[S]^*$ satisfies $\mathcal{T}$, and show that we can exhibit an invariant.

Then, $I = [S]^*$ is an invariant of $S$ by definition of $[.]^*$, and it is stronger than $\mathcal{T}$.

**Caveat:**

- $[S]^\infty$ is most likely *not* a very easy to express invariant
- it is just a convenient completeness argument
- so, completeness does not mean the proof is easy!
Example

We consider the proof that the program below computes the sum of the elements of an array, i.e., when the exit is reached, \( s = \sum_{k=0}^{n-1} t[k] \):

\[
\begin{align*}
i, s & \text{ integer variables} \\
t & \text{integer array of length } n
\end{align*}
\]

\( \ell_0 : \) (true)
\[
s = 0;
\]

\( \ell_1 : \) (s = 0)
\[
i = 0;
\]

\( \ell_2 : \) (i = 0 \land s = 0)
\[
\text{while}(i < n)\{
\ell_3 : \) (0 \leq i < n \land s = \sum_{k=0}^{i-1} t[k])
\[
\quad s = s + t[i];
\]

\( \ell_4 : \) (0 \leq i < n \land s = \sum_{k=0}^{i} t[k])
\[
\quad i = i + 1;
\]

\( \ell_5 : \) (1 \leq i \leq n \land s = \sum_{k=0}^{i-1} t[k])
\]

\( \ell_6 : \) (i = n \land s = \sum_{k=0}^{n-1} t[k])

Principle of the proof:

- for each program point \( \ell \), we have a local invariant \( \mathbb{I}_\ell \)
  (denoted by a logical formula instead of a set of states in the figure)
- the global invariant \( \mathbb{I} \) is defined by:
  \[
  \mathbb{I} = \{ \langle (\ell_0, m_0), \ldots, (\ell_n, m_n) \rangle \mid \\
  \forall n, \ m_n \in \mathbb{I}_\ell \} 
  \]
Outline

1. Safety properties

2. Liveness properties
   - Informal and formal definitions
   - Proof method

3. Decomposition of trace properties

4. A Specification Language: Temporal logic

5. Beyond safety and liveness

6. Conclusion
Liveness properties

Informal definition: liveness properties

A liveness property is a property which specifies that some (good) behavior will eventually occur.

- **Termination** is a liveness property
  “good behavior”: reaching a blocking state (no more transition available)

- “**State a will eventually be reached by all execution**” is a liveness property
  “good behavior”: reaching state a

- The **absence of runtime errors** is *not* a liveness property

As for safety properties, we intend to provide a **formal definition** of liveness.
Intuition towards a formal definition

**How to refute a liveness property?**

- We consider liveness property $\mathcal{T}$ (think $\mathcal{T}$ is termination)
- We assume $S$ does **not** satisfy liveness property $\mathcal{T}$
- Thus, there exists a **counter-example trace** $\sigma \in [S] \setminus \mathcal{T}$;
- Let us assume $\sigma$ is actually finite...
  the definition of liveness says some (good) behavior should eventually occur:
  - how do we know that $\sigma$ cannot be extended into a trace $\sigma \cdot \sigma'$ that will satisfy this behavior?
  - maybe that after a few more computation steps, $\sigma$ **will reach a blocking state**...
Intuition towards a formal definition

To refute a liveness property, we need to look at infinite traces.

Example: if we run a program, and do not see it return...
- should we do Ctrl+C and conclude it does not terminate?
- should we just wait a few more seconds minutes, hours, years?

Towards a formal definition:
we expect any finite trace be the prefix of a trace in $\mathcal{T}$

... since finite executions cannot be used to disprove $\mathcal{T}$

Formal definition (incomplete)

$$\text{PCI}(\mathcal{T}) = S^*$$
Definition

Formal definition

Operator \textbf{Live} is defined by \textbf{Live}(\mathcal{T}) = \mathcal{T} \cup (\mathcal{S}^\infty \setminus \textbf{Safe}(\mathcal{T})). Given property \mathcal{T}, the following three statements are equivalent:

(i) \textbf{Live}(\mathcal{T}) = \mathcal{T}

(ii) \text{PCI}(\mathcal{T}) = \mathcal{S}^*

(iii) \text{Lim} \circ \text{PCI}(\mathcal{T}) = \mathcal{S}^\infty

When they are satisfied, \mathcal{T} is said to be a \textbf{liveness property}

Example: termination

- The property is \mathcal{T} = \mathcal{S}^*
  (i.e., there should be no infinite execution)
- Clearly, it satisfies (ii): \text{PCI}(\mathcal{T}) = \mathcal{S}^*
  thus termination indeed satisfies this definition
Proof of equivalence

**Proof of equivalence:**

(i) implies (ii):

We assume that \( \text{Live}(\mathcal{T}) = \mathcal{T} \), i.e., \( \mathcal{T} \cup (\mathbb{S}^\infty \setminus \text{Safe}(\mathcal{T})) = \mathcal{T} \)

therefore, \( \mathbb{S}^\infty \setminus \text{Safe}(\mathcal{T}) \subseteq \mathcal{T} \).

Let \( \sigma \in \mathbb{S}^* \), and let us show that \( \sigma \in \text{PCI}(\mathcal{T}) \); clearly, \( \sigma \in \mathbb{S}^\infty \), thus:

- either \( \sigma \in \text{Safe}(\mathcal{T}) = \text{Lim}(\text{PCI}(\mathcal{T})) \), so all its prefixes are in \( \text{PCI}(\mathcal{T}) \) and \( \sigma \in \text{PCI}(\mathcal{T}) \)
- or \( \sigma \in \mathcal{T} \), which implies that \( \sigma \in \text{PCI}(\mathcal{T}) \)

(ii) implies (iii):

If \( \text{PCI}(\mathcal{T}) = \mathbb{S}^* \), then \( \text{Lim} \circ \text{PCI}(\mathcal{T}) = \mathbb{S}^\infty \)

(iii) implies (i):

If \( \text{Lim} \circ \text{PCI}(\mathcal{T}) = \mathbb{S}^\infty \), then

\( \text{Live}(\mathcal{T}) = \mathcal{T} \cup (\mathbb{S}^\infty \setminus (\text{Lim} \circ \text{PCI}(\mathcal{T}))) = \mathcal{T} \cup (\mathbb{S}^\infty \setminus \mathbb{S}^\infty) = \mathcal{T} \)
Example

We assume that:

- $S = \{a, b, c\}$
- $T$ states that $b$ should eventually be visited, after $a$ has been visited; elements of $T$ can be described by
  \[ T = S^* \cdot a \cdot S^* \cdot b \cdot S^\infty \]

Then $T$ is a liveness property:

- let $\sigma \in S^*$; then $\sigma \cdot a \cdot b \in T$, so $\sigma \in PCI(T)$
- thus, $PCI(T) = S^*$
A property of **Live**

**Theorem**

If $\mathcal{T}$ is a trace property, then $\text{Live}(\mathcal{T})$ is a liveness property (i.e., operator $\text{Live}$ is **idempotent**).

**Proof:** we show that $\text{PCI} \circ \text{Live}(\mathcal{T}) = S^*$, by considering $\sigma \in S^*$ and proving that $\sigma \in \text{PCI} \circ \text{Live}(\mathcal{T})$; we first note that:

\[
\text{PCI} \circ \text{Live}(\mathcal{T}) = \text{PCI}(\mathcal{T}) \cup \text{PCI}(S^\infty \setminus \text{Safe}(\mathcal{T})) = \text{PCI}(\mathcal{T}) \cup \text{PCI}(S^\infty \setminus \text{Lim} \circ \text{PCI}(\mathcal{T}))
\]

- if $\sigma \in \text{PCI}(\mathcal{T})$, this is obvious.
- if $\sigma \notin \text{PCI}(\mathcal{T})$, then:
  - $\sigma \notin \text{Lim} \circ \text{PCI}(\mathcal{T})$ by definition of the limit
  - thus, $\sigma \in S^\infty \setminus \text{Lim} \circ \text{PCI}(\mathcal{T})$
  - $\sigma \in \text{PCI}(S^\infty \setminus \text{Lim} \circ \text{PCI}(\mathcal{T}))$ as $\text{PCI}$ is extensive when applied to sets of finite traces, which proves the above result
Outline

1 Safety properties

2 Liveness properties
   - Informal and formal definitions
   - Proof method

3 Decomposition of trace properties

4 A Specification Language: Temporal logic

5 Beyond safety and liveness

6 Conclusion
Termination proof with ranking function

- We consider only termination
- We consider transition system $S = (\mathcal{S}, \rightarrow, \mathcal{S}_I)$, and liveness property $\mathcal{T}$
- We seek a way of verifying that $S$ satisfies termination, i.e., that $[S]_\infty \subseteq \mathcal{S}^*$

**Definition: ranking function**

A **ranking function** is a function $\phi : \mathcal{S} \rightarrow E$ where:

- $(E, \sqsubseteq)$ is a well-founded ordering
- $\forall s_0, s_1 \in \mathcal{S}, s_0 \rightarrow s_1 \implies \phi(s_1) \sqsubseteq \phi(s_0)$

**Theorem**

If $S$ has a ranking function $\phi$, it satisfies termination.
We consider the termination of the array sum program:

\begin{align*}
i, s & \text{ integer variables} \\
t & \text{ integer array of length } n \\
l_0 & : \quad s = 0; \\
l_1 & : \quad i = 0; \\
l_2 & : \quad \textbf{while}(i < n)\{ \\
l_3 & : \quad s = s + t[i]; \\
l_4 & : \quad i = i + 1; \\
l_5 & : \quad \} \\
l_6 & : \quad \ldots
\end{align*}

**Ranking function:**

\[
\phi : \mathbb{S} \longrightarrow \mathbb{N}
\]

\[
\begin{align*}
(\ell_0, m) & \mapsto 3 \cdot n + 6 \\
(\ell_1, m) & \mapsto 3 \cdot n + 5 \\
(\ell_2, m) & \mapsto 3 \cdot n + 4 \\
(\ell_3, m) & \mapsto 3 \cdot (n - m(i)) + 3 \\
(\ell_4, m) & \mapsto 3 \cdot (n - m(i)) + 2 \\
(\ell_5, m) & \mapsto 3 \cdot (n - m(i)) + 4 \\
(\ell_6, m) & \mapsto 0
\end{align*}
\]
Proof by variance

- We consider transition system $S = (S, \rightarrow, S_T)$, and liveness property $T$; infinite traces semantics is the greatest fixpoint of $F_\omega$.
- We seek a way of verifying that $S$ satisfies $T$, i.e., that $[[S]]^\infty \subseteq T$

Principle of variance proofs

Let $(\mathbb{I}_n)_{n \in \mathbb{N}}, \mathbb{I}_\omega$ be elements of $S^\infty$; these are said to form a variance proof of $T$ if and only if:

- $S^\omega \subseteq \mathbb{I}_0$
- for all $k \in \{1, 2, \ldots, \omega\}$, $\forall s \in S, \langle s \rangle \in \mathbb{I}_k$
- for all $k \in \{1, 2, \ldots, \omega\}$, there exists $l < k$ such that $F_\omega(\mathbb{I}_l) \subseteq \mathbb{I}_k$
- $\mathbb{I}_\omega \subseteq T$

Proofs of soundness and completeness: exercise, similar to the previous proof but using the definition of $[[S]]^\infty$ instead
Outline

1 Safety properties
2 Liveness properties
3 Decomposition of trace properties
4 A Specification Language: Temporal logic
5 Beyond safety and liveness
6 Conclusion
Decomposition of trace properties

The decomposition theorem

**Theorem**

Let $T \subseteq S^\infty$; it can be decomposed into the conjunction of safety property $\text{Safe}(T)$ and liveness property $\text{Live}(T)$:

$$T = \text{Safe}(T) \cap \text{Live}(T)$$


- **Consequence of this result**: the proof of any trace property can be decomposed into
  - a proof of safety
  - a proof of liveness
Decomposition of trace properties

Proof

- **Safety part:**
  Safe is idempotent, so $\text{Safe}(\mathcal{T})$ is a safety property.

- **Liveness part:**
  Live is idempotent, so $\text{Live}(\mathcal{T})$ is a liveness property.

- **Decomposition:**

  
  \[
  \text{Safe}(\mathcal{T}) \cap \text{Live}(\mathcal{T}) = \text{Safe}(\mathcal{T}) \cap (\mathcal{T} \cup S^\infty \setminus \text{Safe}(\mathcal{T})) \\
  = \text{Safe}(\mathcal{T}) \cap \mathcal{T} \\
  \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \ quad
Decomposition of trace properties

Example: verification of total correctness

i, s integer variables
\( t \) integer array of length \( n \)

\( l_0 : \quad s = 0; \)
\( l_1 : \quad i = 0; \)
\( l_2 : \quad \textbf{while}(i < n)\{ \)
\( l_3 : \quad s = s + t[i]; \)
\( l_4 : \quad i = i + 1; \)
\( l_5 : \quad \} \)
\( l_6 : \quad ... \)

Property to prove:

\[ \text{total correctness} \]

1. the program terminates
2. and it computes the sum of the elements in the array

Application of the decomposition principle

Conjunction of two proofs:

1. Proved with a \textbf{ranking function}
2. Proved with \textbf{local invariants}
Safety and Liveness Decomposition Example

We consider a very simple **greatest common divider** code function:

```c
int f(int a, int b)
{
    while(a > 0)
    {
        int d = b/a;
        int r = b - a * d;
        b = a;
        a = r;
    }
    return b;
}
```

**Specification**

When applied to positive integers, function \( f \) should always return their GCD.
Safety and Liveness Decomposition Example

We consider a very simple greatest common divider code function:

```c
int f(int a, int b){
    while(a > 0){
        int d = b/a;
        int r = b - a * d;
        b = a;
        a = r;
    }
    return b;
}
```

Specification

When applied to positive integers, function \( f \) should always return their GCD.

Safety part

For all trace starting with positive inputs, a conjunction of two properties:

- no runtime errors
- the value of \( b \) is the GCD

Liveness part

Termination, on all traces starting with positive inputs
Decomposition of trace properties

The Zoo of semantic properties: current status

Trace properties
total correctness

Safety properties
never reach $s_0$ before $s_1$

Liveness properties
termination

State properties
absence or runtime errors
partial correctness

Safety: if wrong, can be refuted with a finite trace
proof done by invariance

Liveness: if wrong, has to be refuted with an infinite trace
proof done by variance
Outline

1. Safety properties
2. Liveness properties
3. Decomposition of trace properties
4. A Specification Language: Temporal logic
5. Beyond safety and liveness
6. Conclusion
Notion of specification language

- Ultimately, we would like to **verify or compute** properties.
- So far, we simply describe properties with **sets of executions** or worse, with English / French / ... statements.
- Ideally, we would prefer to use a **mathematical language** for that:
  - to gain in concision, avoid ambiguity.
  - to define sets of properties to consider, fix the form of inputs for verification tools...

**Definition: specification language**

A **specification language** is a set of terms $\mathbb{L}$ with an **interpretation function** (or semantics):

$$[.] : \mathbb{L} \rightarrow \mathcal{P}(\mathcal{S}^\infty) \quad \text{(resp., } \mathcal{P}(\mathcal{S})\text{)}$$

- We are now going to consider specification languages **for states, for traces**...
A State specification language

A first example of a (simple) specification language:

A state specification language

- **Syntax:** we let terms of $L_S$ be defined by:

  $$ p \in L_S ::= @l \mid x < x' \mid x < n \mid \neg p' \mid p' \land p'' \mid \Omega $$

- **Semantics:** $[p]_S \subseteq S_\Omega$ is defined by

  $$
  \begin{align*}
  [\@l]_S & = \{l\} \times M \\
  [x \leq x']_S & = \{(l, m) \in S \mid m(x) \leq m(x')\} \\
  [x \leq n]_S & = \{(l, m) \in S \mid m(x) \leq n\} \\
  [\neg p]_S & = S_\Omega \setminus [p]_S \\
  [p \land p']_S & = [p]_S \cap [p']_S \\
  [\Omega]_S & = \{\Omega\}
  \end{align*}
  $$

**Exercise:** add $\equiv$, $\lor$, $\implies$...
State properties: examples

Unreachability of control state $l_0$:
- **specification**: $\Omega \lor \neg \Diamond l_0$
- **property**: $[\Omega \lor \neg \Diamond l_0]_s = S_\Omega \setminus \{(l_0, m) \mid m \in M\}$

Absence of runtime errors:
- **specification**: $\neg \Omega$
- **property**: $[\neg \Omega]_s = S_\Omega \setminus \{\Omega\} = S$

Intermittent invariant:
- **principle**: attach a local invariant to each control state
- **example**:

  $l_0 : \text{if}(x \geq 0)\{$
  $l_1 : \quad y = x;$  \quad $\Diamond l_1 \implies x \geq 0$
  $l_2 : \} \text{else}\{$
  $l_3 : \quad y = -x;$  \quad $\Diamond l_3 \implies x < 0$
  $l_4 : \}$
  $\quad \wedge \quad \Diamond l_4 \implies x < 0 \land y > 0$
  $l_5 : \quad \}$
  $\quad \wedge \quad \Diamond l_5 \implies y \geq 0$
Propositional temporal logic: syntax

We now consider the specification of trace properties

- **Temporal logic**: specification of properties in terms of events that occur at distinct times in the execution (hence, the name “temporal”)
- There are many instances of temporal logic
- We study a simple one: Pnueli’s Propositional Temporal Logic

**Definition: syntax of PTL (Propositional Temporal Logic)**

Properties over traces are defined as terms of the form

\[
 t(\in \mathbb{L}_{PTL}) ::= \begin{array}{l}
 p \quad \text{state property, i.e., } p \in \mathbb{L}_S \\
 t' \lor t'' \quad \text{disjunction} \\
 \neg t' \quad \text{negation} \\
 \bigcirc t' \quad "\text{next}" \\
 t' \mathcal{U} t'' \quad "\text{until}", \text{i.e., } t' \text{ until } t'' 
\end{array}
\]
Propositional temporal logic: semantics

The semantics of a temporal property is a set of traces, and it is defined by induction over the syntax:

**Semantics of Propositional Temporal Logic formulae**

\[
\begin{align*}
[p]_t &= \{ s \cdot \sigma \mid s \in [p]_s \wedge \sigma \in S^\infty \} \\
[t_0 \lor t_1]_t &= [t_0]_t \cup [t_1]_t \\
[\neg t_0]_t &= S^\infty \setminus [t_0]_t \\
[\Box t_0]_t &= \{ s \cdot \sigma \mid s \in S \wedge \sigma \in [t_0]_t \} \\
[t_0 \mathcal{U} t_1]_t &= \{ \sigma \in S^\infty \mid \exists n \in \mathbb{N}, \forall i < n, \sigma_i \in [t_0]_t \wedge \sigma_n \in [t_1]_t \} 
\end{align*}
\]
Temporal logic operators as syntactic sugar

Many useful operators can be added:

- **Boolean constants:**
  
  \[
  \text{true ::= } (x < 0) \lor \neg(x < 0) \\
  \text{false ::= } \neg\text{true}
  \]

- **Sometime:**
  
  \[
  \Diamond t ::= \text{true} \lor t
  \]
  
  **intuition:** there exists a rank \( n \) at which \( t \) holds

- **Always:**
  
  \[
  \square t ::= \neg(\Diamond(\neg t))
  \]
  
  **intuition:** there is no rank at which the negation of \( t \) holds

**Exercise:** what do \( \Diamond \square t \) and \( \square \Diamond t \) mean?
Propositional temporal logic: examples

We consider the program below:

\begin{align*}
l_0 : & \quad x = \text{input}(); \\
l_1 : & \quad \text{if}(x < 8)\
& \quad l_2 : \quad x = 0; \\
& \quad l_3 : \quad } \text{else} \\
& \quad l_4 : \quad x = 1; \\
& \quad l_5 : \quad } \\
l_6 : & \quad ... \\
\end{align*}

Examples of properties:

- “when $l_4$ is reached, $x$ is positive”
  \[ \square(\Diamond \quad \Diamond l_4 \implies x \geq 0) \]

- “if the value read at point $l_0$ is negative, and when $l_6$ is reached, $x$ is equal to 0”
  \[ \square((\Diamond l_1 \wedge x < 0) \implies \square(\Diamond l_6 \implies x = 0)) \]
Outline

1 Safety properties
2 Liveness properties
3 Decomposition of trace properties
4 A Specification Language: Temporal logic
5 Beyond safety and liveness
6 Conclusion
Security properties

We now consider other interesting properties of programs, and show that they do not all reduce to trace properties.

Security

- Collects many kinds of properties
- So we consider just one:

  an unauthorized observer should not be able to guess anything about private information by looking at public information

- **Example:** another user should not be able to guess the content of an email sent to you
- We need to **formalize this property**
A few definitions

Assumptions:
- We let $\mathcal{S} = (S, \rightarrow, S_I)$ be a transition system.
- States are of the form $(l, m) \in \mathbb{L} \times \mathbb{M}$.
- Memory states are of the form $X \rightarrow V$.
- We let $l, l' \in \mathbb{L}$ (program entry and exit).
  and $x, x' \in X$ (private and public variables).

Security property we are looking at

Observing the value of $x'$ at $l'$
gives no information on the value of $x$ at $l$. 

A few examples

**A secure program** *(no information flow, no way to guess x)*:

\[
\ell : \quad x' = 84;
\]
\[
\ell' : \quad \ldots
\]

**An insecure program** *(explicit information flow, x’ gives a lot of information about x, so that we can simply recompute it)*:

\[
\ell : \quad x' = x - 2;
\]
\[
\ell' : \quad \ldots
\]

**An insecure program** *(implicit information flow, through a test)*:

\[
\ell : \quad \text{if}(x < 0)\{x' = 0;\}
\]
\[
\ell' : \quad \ldots
\]

How to characterize information flow in the semantic level?
Non-interference

We consider the transformer $\Phi$ defined by:

$$
\Phi : \mathbb{M} \longrightarrow \mathcal{P}(\mathbb{M})
$$

$$
m \longmapsto \{ m' \in \mathbb{M} | \exists \sigma = \langle (\ell, m), \ldots, (\ell', m') \rangle \in [\mathcal{S}] \} 
$$

Definition: non-interference

There is no interference between $(\ell, x)$ and $(\ell', x')$ and we write $(\ell', x') \not\Rightarrow (\ell, x)$ if and only if the following property holds:

$$
\forall m \in \mathbb{M}, \forall v_0, v_1 \in \mathcal{V},

\{ m'(x') | m' \in \Phi(m[x \leftarrow v_0]) \} = \{ m'(x') | m' \in \Phi(m[x \leftarrow v_1]) \}
$$

Intuition:

- if two observations at point $\ell$ differ only in the value of $x$, there is no difference in observation of $x'$ at $\ell'$
- in other words, observing $x'$ at $\ell'$ (even on many executions) gives no information about the value of $x$ at point $\ell$...
Non-interference is not a trace property

- We assume \( V = \{0, 1\} \) and \( X = \{x, x'\} \) (store \( m \) is defined by the pair \((m(x), m(x'))\)), and denoted by it.

- We assume \( L = \{l, l'\} \) and consider two systems such that all transitions are of the form \((l, m) \rightarrow (l', m')\) (i.e., system \( S \) is isomorphic to its transformer \( \Phi[S] \))

\[
\begin{align*}
\Phi[S_0] & : (0, 0) \mapsto M \quad (0, 1) \mapsto M \quad (1, 0) \mapsto M \quad (1, 1) \mapsto M \\
\Phi[S_1] & : (0, 0) \mapsto M \quad (0, 1) \mapsto M \quad (1, 0) \mapsto \{(1, 1)\} \quad (1, 1) \mapsto \{(1, 1)\}
\end{align*}
\]

- \( S_1 \) has fewer behaviors than \( S_0 \): \([S_1]^* \subset [S_0]^*\)

- \( S_0 \) has the non-interference property, but \( S_1 \) does not

- If non interference was a trace property, \( S_1 \) should have it (monotony)

Thus, the non interference property is not a trace property
Dependence properties

Many notions of dependences
So we consider just one:

what inputs may have an impact on the observation of a given output

Applications:
- reverse engineering: understand how an input gets computed
- slicing: extract the fragment of a program that is relevant to a result

This corresponds to the negation of non-interference
Interference

**Definition: interference**

There is *interference* between \((\ell, x)\) and \((\ell', x')\) and we write \((\ell', x') \rightsquigarrow (\ell, x)\) if and only if the following property holds:

\[
\exists m \in M, \exists v_0, v_1 \in V,
\{ m'(x') | m' \in \Phi(m[x \leftarrow v_0]) \} \neq \{ m'(x') | m' \in \Phi(m[x \leftarrow v_1]) \}
\]

- This expresses that there is at least one case, where the value of \(x\) at \(\ell\) has an impact on that of \(x'\) at \(\ell'\)
- It may not hold even if the computation of \(x'\) reads \(x\):

\[
\ell : \quad x' = 0 \ast x;
\ell' : \quad \ldots
\]
Beyond safety and liveness

Interference is not a trace property

- We assume \( V = \{0, 1\} \) and \( X = \{x, x'\} \) (store \( m \) is defined by the pair \((m(x), m(x'))\), and denoted by it)
- We assume \( L = \{l, l'\} \) and consider two systems such that all transitions are of the form \((l, m) \rightarrow (l', m')\) (i.e., system \( S \) is isomorphic to its transformer \( \Phi[S] \))

\[
\begin{align*}
\Phi[S_0] : & (0, 0) \longmapsto M & \Phi[S_1] : & (0, 0) \longmapsto \{(1, 1)\} \\
& (0, 1) \longmapsto M & & (0, 1) \longmapsto \{(1, 1)\} \\
& (1, 0) \longmapsto \{(1, 1)\} & & (1, 0) \longmapsto \{(1, 1)\} \\
& (1, 1) \longmapsto \{(1, 1)\} & & (1, 1) \longmapsto \{(1, 1)\}
\end{align*}
\]

- \( S_1 \) has fewer behavior than \( S_0 \): \([S_1]^* \subset [S_0]^*\)
- \( S_0 \) has the interference property, but \( S_1 \) does not
- If interference was a trace property, \( S_1 \) should have it (monotony)

Thus, the interference property is not a trace property
Conclusion:

- The absence of interference between \((\ell, x)\) and \((\ell', x')\) is not a trace property: we cannot describe as the set of programs the semantics of which is included into a given set of traces.
- It can however be described by a set of sets of traces: we simply collect the set of program semantics that satisfy the property.

This is what we call a hyperproperty:

**Hyperproperties**

- **Trace hyperproperties** are described by sets of sets of executions.
- **Trace properties** are described by sets of executions.

**2-safety**: to disprove the absence of interference (i.e., to show there exists an interference), we simply need to exhibit **two finite traces**.
Outline

1. Safety properties
2. Liveness properties
3. Decomposition of trace properties
4. A Specification Language: Temporal logic
5. Beyond safety and liveness
6. Conclusion
The Zoo of semantic properties

Sets of sets of executions
non-interference, dependency

Trace properties
total correctness

Safety properties
never reach $s_0$ before $s_1$

State properties
absence or runtime errors
partial correctness

Liveness properties
termination
Summary

To sum-up:

- **Trace properties** allow to express a large range of program properties
- **Safety** = absence of bad behaviors
- **Liveness** = existence of good behaviors
- Trace properties can be **decomposed** as conjunctions of safety and liveness properties, with **dedicated proof methods**
- Some interesting properties are **not trace properties**
  - security properties are **sets of sets of executions**
- Notion of **specification languages** to describe program properties