Semantics and Applications to Verification

Xavier Rival, Sylvain Conchon and Jérôme Feret
Notes by Antoine Groudiev

8th March 2024

Contents

Introduction

This document is Antoine Groudiev's class notes while following the class *Sémantique et applications à la vérification de programmes* (Semantics and Applications to Verification) at the Computer Science Department of ENS Ulm. It is freely inspired by the class notes of Xavier Rival and Sylvain Conchon.

1 Introduction to Semantics

1.1 Case studies

We will study some examples of software errors: what are the causes of these, what kind of properties do we want to verify in order to prevent such failures?

1.1.1 Ariane 5 – Flight 501

Ariane 5 was a satellite launcher, aimed at replacing Ariane 4. Its first flight, June, 4th, 1996, was a failure, with more than $370 000 000 of damages. 37 seconds after the launch, the rocket exploded.

The system contained sensors, two calculators (SRI, OBC), actuators, and redundant systems (failure tolerant system). The failure was due to an unhandled arithmetic error. Each register of the SRI has a size of 16, 32, or 64 bits. The error was due to a conversion of a 64-bit float to a 16-bit integer. The value was too large to be represented in 16 bits, and the conversion failed. The software was not able to handle this error, and the system crashed.

Several solutions would have prevented this mishappening:

- Desactivate interruptions on overflows
- Fix the SRI code, so that no overflow can happen. All conversions must be *guarded against overflows*:

```
double x = /* ... */ ;
short i = /* ... */ ;
if ( -32768. <= x && x <= 32767. )
```

```
        i = ( short ) x ;
else
        i = /* default value */ ;
```

This may be costly, but redundant tests can be removed.

- Handle converson errors (not trivial): identify the problem and fix it at run time.

The piece of code that generated the error was used to do a useless task, the re-calibration process is not usefull after take-off. Furthermore, the code was already used in Ariane 4; initially protected by a safety guard, many conversions and tests were removed for the sake of performance after being tested on Ariane 4.

The crash was not prevented by redundant systems: the two calculators were running the same code, and the same error was made on both. Redundancy can prevent hardware errors, but is not enough to prevent software errors.

1.1.2 Lufthansa Flight 2904

On November 22, 2003, a Lufthansa Airbus A320-200 crashed at the airport of Warsaw, Poland. The plane was landing, and the weather was bad. The plane was not able to stop before the end of the runway, and crashed into a building. The cause of the crash was a software error in the plane's computer. The plane was not able to compute the correct deceleration, and the pilot was not able to stop the plane in time.

1.1.3 Patriot missile (anti-missile system), Dahran

The purporse of the Patriot system was to destroy foe missiles before they reach their target, and was used in the first Gulf War with a success rate around 50%. The system was used to destroy Scud missiles, and the system was not able to destroy one of them, which hit a barrack, killing 28 people. The cause of the failure was a software error in the system's clock. The system was not able to compute the time correctly due to fixed precision arithmetic error, and the missile was not destroyed.

1.1.4 General remarks

The examples given so far are not isolated cases. The typical causes of software errors are:

- Improper specification
- Incorrect implementation of a specification (the code should be free of runtime errors, and should produce a result that meets some property)
- Incorrect understanding of the execution model (generation of too imprecise results)

This creates new challenges to ensure embedded systems do not fail. The main techniques to ensure software safety are software development techniques:

- software engineering
- programming rules
- make software cleaner

In this class, we will instead dive into formal methods:

- should have sounds mathematical foundations
- should allow guaranteeing software meet some complex properties
- should be trustable
- increasingly used in real life applications

This course will contain two main parts. The first part will be about Semantics, which allow describing precisely the behavior of programs, express the properties to verify, and to transform and compile programs. The second part, Verification, aims at proving semantic properties of programs. A very strong limitation of verification is indecidability; several approaches make various compromises around indecidability.

1.2 Approaches to verification

1.2.1 The termination problem

Definition (Termination)**.** A program P terminates on input X if and only if any execution of P with input X eventually reaches a final state. A final state is a final point in the program (i.e., not an error).

Definition (Termination problem)**.** Can we find a program Pt that takes as argument a program P and data X and that returns `True` if P terminates on X, and `False` otherwise?

Property 1. *The termination problem is not computable.*

Proof. We assum there exists a program Pa such that Pa always terminates, and returns 1 if and only if P terminates on input X. We consider the following program:

```
void P0 ( P ) {
    if ( Pa ( P , P ) == 1 ) {
        while ( 1 ) {
            // loop forever
        }
    } else {
        return ; // do nothing
    }
}
```

and we consider the return value of $Pa(P0, P0)$. If $Pa(P0, P0) == 1$, then $P0$ loops forever, and if $Pa(P0, P0) == 0$, then $P0$ terminates. This is a contradiction, and the termination problem is not computable. □

Property 2. *The absence of runtime errors is not computable. We cannot find a program Pc that takes a program P and input X as arguments, always terminates, and returns 1 if and only if P runs on input X without a runtime error.*

Theorem (Rice theorem)**.** *Considering a Turing complete language, any non-trivial semantic specification is not computable. Therefore, all interesting properties are not computable (termination, absence of runtime/arithmetic errors, etc.).*

The initial verification problem is not computable. Several compromises can be made: simulation, testing, assisted theorem proving, model checking, bug-finding, static analysis with abstraction.

Definition (Safety verification problem)**.** The Semantics $[\![P]\!]$ of a program P is the set of behaviors of P (e.g. states). A property to verify $\mathcal{S}$ is the set of admissible behaviors (e.g. safe states). Our goal is to establish $[\![P]\!] \subseteq \mathcal{S}$

$[\![P]\!]$ can be sound (identify any wrong program), complete (accept all correct programs), and automated, but not all three at the same time.

Testing by simulation The principle of testing by simulation is to run the program on finitely many finite inputs, to maximize coverage and inspect erroneous traces to fix bugs. It is very widely used, through unit testing, integration testing, etc. It is both automated and complete, but is unsound and costly.

Machine assisted proof The principle of machine assisted proof is to have a machine check proof that is partly human written: tactics or solvers may help in the inference, and the hardest invariants have to be user-supplied. It is sound and quasi-complete, but not fully automated and costly.

Model checking We consider finite systems only, using algorithms for exhaustive exploration, symmetry reduction, ... It is automated, sound, and complete *with respect to the model*.

Bug finding The principle of bug finding is to identify "likely" issues, patterns known to often indicate an error: it uses bounded symbolic execution, model exploration, and rank "defect" reports using heuristics. It is neither sound nor complete, but is fully automated.

Static analysis with abstraction The principle of static analysis with abstraction is to use some approximation, but always in a conservative manner. We can use under-approximation of the property to verify:

$$\mathcal{S}_{\text{under}} \subseteq \mathcal{S}$$

and over-approximation of the semantics:

$$[\![P]\!] \subseteq [\![P]\!]_{\text{upper}}$$

We let an automatic static analyser attempt to prove that:

$$[\![P]\!]_{\text{upper}} \subseteq \mathcal{S}_{\text{under}}$$

If it succedds, then we have proven that $[\![P]\!] \subseteq \mathcal{S}$. It is automated, sound, but incomplete.

1.2.2 A summary of common verification techniques

	Automatic	Sound	Complete	Source level
Simulation	Yes	No	Yes	Yes
Assisted Proving	No	Yes	Almost	Partially
Model-checking	Yes	Yes	Partially	No
Bug-finding	Yes	No	No	Yes
Static analysis	Yes	Yes	No	Yes

1.3 Orderings, lattices and fixpoints

1.3.1 Basic definitions on orderings

Definition (Partially ordered set (poset))**.** Let a set $\mathcal{S}$ and a binary relation $(\sqsubseteq) \subseteq \mathcal{S} \times \mathcal{S}$ over $\mathcal{S}$. Then, $\sqsubseteq$ is an order relation if and only if it is reflexive, transitive, antisymettric. Furthermore, we define $x \sqsubset y ::= (x \sqsubseteq y \wedge x \neq y)$. Most orders in this class won't be total.

We often use Hasse diagrams to represent posets.

In the following, we illustrate order relations and their usefulness in semantics using the standard definition of *word automata*. The semantics of an automaton is the set of words recognized by it.

We can already define a few semantic properties:

- $\mathcal{P}_0$: no recognized word contains two consecutive b

$$\mathcal{L}[\mathcal{A}] \subseteq L^\star \setminus L^\star bbL^\star$$

- $\mathcal{P}_1$: all recognized words contain at least one occurrence of a

$$\mathcal{L}[\mathcal{A}] \subseteq L^\star aL^\star$$

- $\mathcal{P}_2$: recognized words do not contain b

$$\mathcal{L}[\mathcal{A}] \subseteq (L \setminus \{b\})^\star$$

Definition ($\bot$, $\top$)**.** When they exist, we denote by infimum $\bot$ and supremum $\top$ the smallest and largest elements of the poset.

Definition ($\sqcup$, $\sqcap$)**.** We denote $\sqcap\mathcal{S}$ the glb (greatest lower bound) of $\mathcal{S}$, and $\sqcup$ the lub (lowest upper bound) of $\mathcal{S}$.

1.3.2 Complete lattice

Definition (Complet lattice)**.** A complete lattice is a tuple $(\mathcal{S}, \sqsubseteq, \bot, \top, \sqcup, \sqcap)$ where $(S, \sqsubseteq)$ is a poset, and any subset $\mathcal{S}'$ of $\mathcal{S}$ has a glb $\sqcap\mathcal{S}'$ and a lub $\sqcup\mathcal{S}$.

Definition (Lattice)**.** A lattice is a tuple $(\mathcal{S}, \sqsubseteq, \bot, \top, \sqcup, \sqcap)$ where $(S, \sqsubseteq)$, and any pair $\{x, y\}$ of $\mathcal{S}$ has a glb $x \sqcap y$ and a lub $x \sqcup y$.

Example. $\mathbb{Q} \cap [0, 1]$ *is a lattice but not a complete lattice, since*

$$\left\{ q \in \mathbb{Q} \cap [0, 1] \,\middle|\, q \leqslant \frac{\sqrt{2}}{2} \right\} \subseteq \mathbb{Q} \cap [0, 1]$$

has no lowest upper bound in $\mathbb{Q} \cap [0, 1]$.

Property 3. *A finite lattice is also a complete lattice.*

Definition (Increasing chain)**.** Let $(\mathcal{S}, \sqsubseteq)$ be a poset and $\mathcal{C} \subseteq \mathcal{S}$. It is an increasing chain if and only if it is not empty, and $(\mathcal{C}, \sqsubseteq)$ is total.

Example. *In the powerset* $(\mathcal{P}(\mathbb{N}), \subseteq)$*,*

$$\mathcal{C} := \left\{ \{2^0, \ldots, 2^i\} \,\middle|\, i \in \mathbb{N} \right\}$$

is an increasing chain.

Definition (Increasing chain condition)**.** The poset $(\mathcal{S}, \sqsubseteq)$ satisfies the increasing chain condition if and only if any increasing chain $\mathcal{C} \subseteq \mathcal{S}$ is finite.

Definition (Complete partial order)**.** A complete partial order (cpo) is a poset $(\mathcal{S}, \sqsubseteq)$ such that any increasing chain $\mathcal{C}$ of $\mathcal{S}$ has at least an upper bound. A pointed cpo is a cpo with a bottom element $\bot$.

1.3.3 How to prove semantic properties

1.3.4 Operators over a poset

Definition (Operators and orderings)**.** Let $(\mathcal{S}, \sqsubseteq)$ be a poset and $\varphi : \mathcal{S} \to \mathcal{S}$ be an operator over $\mathcal{S}$. Then, φ is:

- *monotone* if and only if $x \sqsubseteq y \Rightarrow \varphi(x) \sqsubseteq \varphi(y)$
- *continuous* if and only if, for any **chain** $\mathcal{S}' \subseteq \mathcal{S}$, then:

$$\begin{cases} \text{if } \sqcup\mathcal{S}' \text{ exists, so does } \sqcup\varphi(\mathcal{S}') \\ \text{and } \sqcup\varphi(\mathcal{S}') = \varphi(\sqcup\mathcal{S}') \end{cases}$$

- $\sqcup$-preserving if and only if:

$$\forall \mathcal{S}' \subseteq \mathcal{S}, \begin{cases} \text{if } \sqcup\mathcal{S}' \text{ exists, so does } \sqcup\varphi(\mathcal{S}') \\ \text{and } \sqcup\varphi(\mathcal{S}') = \varphi(\sqcup\mathcal{S}') \end{cases}$$

Property 4 (Continuity $\implies$ monotonicity)**.** *If φ is continuous, then it is monotone.*

Property 5 ($\sqcup$-preserving $\implies$ monotonicity)**.** *If φ preserves $\sqcup$, then it is monotone.*

1.3.5 Fixpoints theorems

Definition (Fixpoints)**.** Let $(\mathcal{S}, \sqsubseteq)$ be a poset and $\varphi : \mathcal{S} \to \mathcal{S}$ be an operator over $\mathcal{S}$.

- A fixpoint of φ is an element x such that $\varphi(x) = x$.
- A pre-fixpoint of φ is an element x such that $x \sqsubseteq \varphi(x)$.
- A post-fixpoint of φ is an element x such that $\varphi(x) \sqsubseteq x$.
- The least fixpoint of φ is a fixpoint x such that, for any fixpoint y, $x \sqsubseteq y$.
- The greatest fixpoint of φ is a fixpoint x such that, for any fixpoint y, $y \sqsubseteq x$.

Theorem (Tarski's)**.** *Let $(\mathcal{S}, \sqsubseteq, \bot, \top, \sqcup, \sqcap)$ be a complete lattice and $\varphi : \mathcal{S} \to \mathcal{S}$ be a monotone operator. Then:*

- *φ has a least fixpoint* $\operatorname{lfp}\varphi$ *and* $\operatorname{lfp}\varphi = \sqcap \{ x \in \mathcal{S} \mid \varphi(x) \sqsubseteq x \}$
- *φ has a greatest fixpoint* $\operatorname{gfp}\varphi$ *and* $\operatorname{gfp}\varphi = \sqcup \{ x \in \mathcal{S} \mid x \sqsubseteq \varphi(x) \}$
- *the set of fixpoints of φ is a complete lattice*

Example. *We consider a set $\mathcal{E}$, and a subset $\mathcal{A} \subseteq \mathcal{E}$. We let:*

$$\begin{aligned} f : \mathcal{P}(\mathcal{E}) &\to \mathcal{P}(\mathcal{E}) \\ X &\mapsto X \cup \mathcal{A} \end{aligned}$$

According to Tarski's theorem, the smallest fixpoint of f is $\mathcal{A}$, and the greastest is $\mathcal{E}$.

Theorem (Kleene's)**.** *Let $(\mathcal{S}, \sqsubseteq, \bot)$ be a pointed cpo and $\varphi : \mathcal{S} \to \mathcal{S}$ be a continuous operator over $\mathcal{S}$. The φ has a least fixpoint, and*

$$\operatorname{lfp}\varphi = \bigsqcup_{n \in \mathbb{N}} \varphi^n(\bot)$$

2 Operational Semantics

Operational semantics are mathematical descriptions of the executions of a program. It is based on a model of programs, called transition systems.

2.1 Definition and properties

Definition (Transition systems (TS))**.** A transition system is a tuple $(\mathbb{S}, \rightarrow)$ where $\mathbb{S}$ is the *set of states of the system*, and $\rightarrow \subseteq \mathbb{S} \times \mathbb{S}$ is the transition relation of the system.

Note that the set of states may be infinite. The majority of interesting examples come from the cases where $\mathbb{S}$ is infinite.

Definition (Deterministic system)**.** A deterministic system is such that a state fully determines the next state.

$$\forall s_0, s_1, s_1' \in \mathbb{S}, (s_0 \rightarrow s_1 \wedge s_0 \rightarrow s_1') \implies s_1 = s_1'$$

Otherwise, a transition system is non-deterministic.

The transition relation $\rightarrow$ defines atomic execution steps. It is often called *small-step semantics* or *structured operational semantics*. Steps are *discrete*, and we do not consider non-deterministic systems with probability on transitions (probabilistic transition systems).

Definition (Initial and final states)**.** We often consider transition systems with a set of initial and final states:

1. a set of initial states $\mathbb{S}_I \subseteq \mathbb{S}$ denots states where the execution should start
2. a set of final states $\mathbb{S}_F \subseteq \mathbb{S}$ denotes states where the execution should reach the end of the program

When needed, we add these to the definition of the transition systems $(S, \rightarrow, \mathbb{S}_I, \mathbb{S}_F)$.

Definition (Blocking state $\neq$ final state)**.** A state $s_0 \in \mathbb{S}$ is blocking when it is the origin of no transition:

$$\forall s_1 \in \mathbb{S}, \neg(s_0 \rightarrow s_1)$$

As an example, we often introduce an error state (usually noted Ω)

2.2 Examples

2.2.1 Word recognition

We can formalize the *word recognition* by a finite automaton using a transition system. We consider an automaton $\mathcal{A} = (Q, q_i, q_f, \rightarrow)$. A state is defined by the remaining of the word to recognize, and the automaton state that has been reached so far. Thus,

$$\mathbb{S} = Q \times \Sigma^\star$$

We define the transition relation $\rightarrow$ to be:

$$(q_O, aw) \rightarrow (q_1, w) \iff q_0 \rightarrow^a q_1$$

The initial and final states are defined by:

$$\begin{cases} \mathbb{S}_I = \{ (q_i, w) \mid w \in \Sigma^\star \} \\ \mathbb{S}_F = \{(q_f, \varepsilon)\} \end{cases}$$

2.2.2 Pure λ-calculus

A bare-bones model of functional programming:

Definition (λ-terms)**.** The set of λ-terms is defined by:

$$\begin{array}{rr} t, u, \cdots ::= x & \textit{variable} \\ |\lambda x \cdot t & \textit{abstraction} \\ |tu & \textit{application} \end{array}$$

Definition (β-reduction)**.**

The λ-calculus defines a transition system. $\mathbb{S}$ is the set of λ-terms and $\rightarrow_\beta$ the transition relation. $\rightarrow_\beta$ is non-deterministic, since multiple β-reduction are sometimes possible. Given a lambda term t_0, we may consider $(\mathbb{S}, \rightarrow_\beta, \mathbb{S}_I)$ where $\mathbb{S}_I = \{t_0\}$. Blocking states are terms with no redex $(\lambda x \cdot u)v$.

2.3 A MIPS like assembly language

We now consider a very simplified assembly language, containing machine integers $\mathbb{B}^{32}$. Instruction are encoded over 32 bits and stored in the same space as data, $\mathbb{B}^{32}$. We assume a fixed set of addresses A.

The memory configuration contains the program counter `pc`, the general purpose register $r_0, \ldots, r_{31}$, and the main memory (RAM):

$$\mathbf{mem} : A \subseteq \mathbb{B}^{32} \rightarrow \mathbb{B}^{32}$$

Definition (State)**.** A state is a tuple (π, ρ, μ) which comprises a program counter value $\pi \in \mathbb{B}^{32}$, a function mapping each general purpose register to its value $\rho : \{0, \ldots, 32\} \rightarrow \mathbb{B}^{32}$, and a function mapping each memory cell to its value, $\mu : A \rightarrow \mathbb{B}^{32}$.

We can define transition relations.

We now look at a more classical imperative language (a bare-bone subset of C), containing variables X, labels L, and values V. A syntax can be defined.

2.4 Traces semantics

Definition (Traces)**.** A *finite trace* is a finite sequence of states $s_0, \ldots, s_n$ noted $\langle s_0, \ldots, s_n \rangle$. An infinite trace is an infinite sequence of states $\langle s_0, \ldots \rangle$. Besides, we write $\mathbb{S}^\star$ for the set of finite traces, $\mathbb{S}^\omega$ for the set of infinite traces, and $S^\infty = \mathbb{S}^\star \cup \mathbb{S}^\omega$.

Definition (Concatenation operator $\cdot$)**.**

3 Traces Properties

3.1 A high level overview

3.1.1 Introduction

The goal of verification is to prove that $[\![P]\!] \subseteq \mathcal{S}$ (i.e. that all behaviors of P satisfy specifications $\mathcal{S}$) where $[\![P]\!]$ is the program semantics and $\mathcal{S}$ the desired specification. Today, we will mostly focus on program's properties, $\mathcal{S}$. We will see different families of properties, proof techniques, and specification of properties (are there languages to describe properties?).

A property is a set of traces, defining the admissible executions. There are *safety properties*: something will never happen, which is often proven by invariance; *liveness properties*: something will eventually happen, proven by variance; and beyond safety and liveness, there are *hyperproperties*.

As usual, we consider $\mathcal{S} = (\mathbb{S}, \rightarrow, \mathbb{S}_\mathcal{I})$.

Definition (Properties as sets of states)**.** A property $\mathcal{P} \subseteq \mathbb{S}$ is a set of states $\mathcal{P} \subseteq \mathbb{S}$. $\mathcal{P}$ is satisfied if and only if all reacheable states belong to $\mathcal{P}$.

This is the case of the absence of runtime errors, and non-termination.

Definition (Properties as sets of traces)**.** A property $\mathcal{T}$ is a set of traces $\mathcal{T} \subseteq \mathbb{S}^\infty$. $\mathcal{T}$ is satisfied if and only if all traces belong to $\mathcal{T}$, i.e. $[\![\mathbb{S}^\infty]\!] \subseteq \mathcal{T}$

State properties are trace properties. Functional properties (such as "program P takes one integer input x and returns its absolute value") and termination ($\mathcal{T} = \mathbb{S}^\star$) are trace properties.

3.1.2 A few monotonicity properties

Property 6. *Let $\mathcal{P}_0, \mathcal{P}_1 \subseteq \mathbb{S}$ be two state properties. When $\mathcal{P}_0 \subseteq \mathcal{P}_1$, we say that $\mathcal{P}_0$ is* stronger *than $\mathcal{P}_1$, i.e. if a program $\mathcal{S}$ satisfies $\mathcal{P}_0$ then it also satisfies $\mathcal{P}_1$.*

Property 7. *Let $\mathcal{T}_0, \mathcal{T}_1 \subseteq \mathbb{S}$ be two trace properties. When $\mathcal{T}_0 \subseteq \mathcal{T}_1$, we say that $\mathcal{T}_0$ is* stronger *than $\mathcal{T}_1$, i.e. if a program $\mathcal{S}$ satisfies $\mathcal{T}_0$ then it also satisfies $\mathcal{T}_1$.*

Property 8. *Let $\mathcal{S}_0, \mathcal{S}_1$ be two transition systems. When $[\![\mathcal{S}_0]\!] \subseteq [\![\mathcal{P}_1]\!]$, we say that $\mathcal{S}_1$* has more behaviors than *$\mathcal{S}_0$, i.e. if a property $\mathcal{P}$ is satisfied by $\mathcal{S}_1$ then it is also satisfied by $\mathcal{S}_0$.*

3.2 Safety properties

3.2.1 Informal and formal definitions

Definition (Informal definition of safety properties)**.** A safety property is a property which specifies that some (bad) behavior defined by a finite, irrecoverable observation will never occur, at any time.

Example. *The following properties are safety properties:*

- *Absence of runtime errors*
- *State properties (the "bad thing" is reaching $\mathbb{S} \setminus \mathcal{T}$)*
- *Non-termination*
- *"Not reaching state b after visiting state a"*

*Termination is **not** a safety property, since no finite execution is a counter-example of its termination.*

We now intend to provide a formal definition of safety. How to refute a safety property? We assume $\mathcal{S}$ does not satisfy safety property $\mathcal{P}$. Thus, there exists a counter-example trace $\sigma = \langle s_0, \ldots, s_n, \ldots \rangle \in [\![S]\!] \setminus \mathcal{P}$. At this point of our study, the trace may be finite or infinite. The intuitive definition says this trace *eventually exhibits some bad behavior*, that is *irrecoverable* at some *observed at some given time*, thus the observation corresponds to some index i. Therefore, trace $\sigma' = \langle s_0, \ldots, s_i \rangle$ violates $\mathcal{P}$, i.e. $\sigma' \notin \mathcal{P}$. Due to the irrecoverability of the observation, the same goes for any trace with the same prefix. We remark that σ' is finite.

A safety property that does not hold can always be refuted with a finite, irrecoverable counter-example.

3.2.2 A few operators on traces

Definition (Prefix)**.** We write $\sigma_{\lceil i}$ for the prefix of length i of trace σ.

Definition (Suffix or tail)**.** We write $\sigma_{i\rceil}$ for the suffix of length i of trace σ.

Definition (Upper closure operators (PCl))**.** In a preorder $(\mathcal{S}, \sqsubseteq)$, a function $\varphi : \mathcal{S} \to \mathcal{S}$ is an upper closure operatof if and only if it is monotone, extensive ($\forall x \in \mathcal{S}, x \sqsubseteq \varphi(x)$) and idempotent.

Definition (Prefix closure)**.** The prefix closure operator is defined by:

$$\begin{aligned} \mathrm{PCl} : \mathcal{P}(\mathbb{S}^\infty) &\to \mathcal{P}(\mathbb{S}^\star) \\ X &\mapsto \left\{ \sigma_{\lceil i} \;\middle|\; \sigma \in X, i \in \mathbb{N} \right\} \end{aligned}$$

PCl is monotone, idempotent, but not extensive on $\mathcal{P}(\mathbb{S}^\infty)$ (infinite traces do not appear anymore). Its restriction to $\mathcal{P}(\mathbb{S}^\star)$ is extensive.

Definition (The Lim operator)**.** The limit operator is defined by:

$$\begin{aligned} \mathrm{Lim} : \mathcal{P}(\mathbb{S}^\infty) &\to \mathcal{P}(\mathbb{S}^\infty) \\ X &\mapsto X \cup \left\{ \sigma \in \mathbb{S}^\infty \;\middle|\; \forall i \in \mathbb{N}, \sigma_{\lceil i} \in X \right\} \end{aligned}$$

Note that the operator Lim is an upper-closure operator.

Proof. Left as an exercise! □

Example. *Assume that:*

$$\mathcal{S} = \{\varepsilon, \langle a\rangle, \langle a, b\rangle, \langle a, b, a\rangle, \langle a, b, a, b\rangle, \langle a, b, a, b, a\rangle, \dots\}$$

then,

$$\mathrm{Lim}(\mathcal{S}) = \mathcal{S} \uplus \{\langle a, b, a, b, a, b, \dots\rangle\}$$

3.2.3 Formal definition of safety

Definition (The Safe operator)**.** Operator Safe is defined by

$$\mathrm{Safe} = \mathrm{Lim} \circ \mathrm{PCl}$$

Note that Safe is an upper closure operator over $\mathcal{P}(\mathbb{S}^\infty)$.

Definition (Safety property)**.** A trace property $\mathcal{T}$ is a safety property if and only if it is a fixpoint of the Safe operator, that is

$$\mathrm{Safe}(\mathcal{T}) = \mathcal{T}$$

Furthermore, if $\mathcal{T}$ is a trace property, then $\mathrm{Safe}(\mathcal{T})$ is a safety property, since Safe is idempotent.

Theorem. *Any state property is also a safety property.*

Proof. Consider a state property $\mathcal{P}$. It is equivalent to trace property $\mathcal{T} = \mathbb{P}^\infty$:

$$\begin{aligned} \mathrm{Safe}(\mathcal{T}) &= \mathrm{Lim} \circ \mathrm{PCl}(\mathcal{P}^\infty) \\ &= \mathrm{Lim}(\mathcal{P}^\star) \\ &= \mathcal{P}^\star \cup \mathcal{P}^\omega \\ &= \mathcal{P}^\infty \\ &= \mathcal{T} \end{aligned}$$

Therefore, $\mathcal{T}$ is indeed a safety property. □

3.2.4 Intuition of the formal definition

Operator Safe saturates a set of traces S with prefixes and infinite traces of all finite prefixes of which can be observed in S. Thus, if $\text{Safe}(S) = S$ and σ is a trace, to establish that $\sigma \notin S$, it is sufficient to discover a *finite prefix* of σ that cannot be observed in S.

Alternatively, if all finite prefixes of σ belong to S or can be observed as a prefix of another trace in S, by definition of the limit operator, σ belongs to S, *even if it is infinite*. Therefore, the definition captures properties that *can be disproved with a finite counter-example*.

3.2.5 Proof method for safety properties

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

Definition (Invariance proofs)**.** Let $\mathbb{I}$ be a set of finite traces; it is said to be an *invariant* if and only if:

$$\begin{cases} \forall s \in \mathbb{S}_\mathcal{I}, \langle s \rangle \in \mathbb{I} \\ F_*(\mathbb{I}) \subseteq \mathbb{I} \end{cases}$$

Where F_* is the semantic function, defined previously, that computes the traces of length $i+1$ from the traces of length i, and adds the traces of length 1.

$\mathbb{I}$ is *stronger* than $\mathcal{T}$ if and only if $\mathbb{I} \subseteq \mathcal{T}$. The proof methode *by invariance* is based on finding an invariant that is stronger than $\mathcal{T}$.

Theorem (Soundness of the invariance proof method)**.** *The invariance proof methode is* sound*: if we can find an invariant for $\mathcal{S}$, that is stronger than safety property $\mathcal{T}$, then $\mathcal{S}$ satisfies $\mathcal{T}$.*

Proof. Assume that $\mathbb{I}$ is an invariant of $\mathcal{S}$ and that it is stronger than $\mathcal{T}$. Let's show that $\mathcal{S}$ satisfies $\mathcal{T}$.

By induction over n, we can prove that $F_*^n(\{ \langle s \rangle \mid s \in \mathbb{S}_\mathcal{I} \}) \subseteq F_*^n(\mathbb{I}) \subseteq \mathbb{I}$. Therefore, $[\![\mathcal{S}]\!]^\star \subseteq \mathbb{I}$ and thus, $\text{Safe}([\![\mathcal{S}]\!]^\star) \subseteq \text{Safe}(\mathbb{I}) \subseteq \text{Safe}(\mathcal{T})$ since Safe is monotone.

We remark that $[\![\mathcal{S}]\!]^\propto = \text{Safe}([\![\mathcal{S}]\!]^\star)$, hence $[\![\mathcal{S}]\!]^\propto = \text{Safe}([\![\mathcal{S}]\!]^\star) \subseteq \text{Safe}\,\mathcal{T} = \mathcal{T}$. We conclude $[\![\mathcal{S}]\!]^\propto \subseteq \mathcal{T}$, i.e. $\mathcal{S}$ statisfies property $\mathcal{T}$. □

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

Proof. We choose $\mathbb{I} = [\![\mathcal{S}]\!]^\star$, which is both an invariant of $\mathcal{S}$ and is stronger than $\mathcal{T}$ since $\mathcal{S}$ satisfies $\mathcal{T}$. □

Note that $[\![\mathbb{S}]\!]^\propto$ is most likely not a very easy to express invariant, but it is just a convenient completeness argument. Therefore, completeness does not mean that the proof is easy.

Example. *Consider the following program which computed the sum of the elements of an array: We want to proove that when the exit is reached, $\mathtt{s} = \sum_{k=0}^{n-1} \mathtt{t}[k]$. For each program point l, we have a local invariant $\mathbb{I}_l$, denoted below by a logical formula instead of a set of states:*

The global invariant $\mathbb{I}$ is defined by:

$$\mathbb{I} := \{ \langle (l_0, m_0), \ldots, (l_n, m_n) \rangle \mid \forall n, m_n \in \mathbb{I}_{l_n} \}$$

which is the set of traces satisfying all the local invariants.

```
s = 0;
i = 0;
while (i < n) {
    s = s + t[i];
    i = i + 1;
}
```

Figure 1: A simple program which computes the sum of the elements of `t`

```
1 s = 0;
2 i = 0;
3 while (i < n) {
4     s = s + t[i];
5     i = i + 1;
6 }
```

$l_0 : (\!|\texttt{true}|\!)$

$l_1 : (\!|s = 0|\!)$

$l_2 : (\!|i = 1 \wedge s = 0|\!)$

$l_3 : (\!|0 \leqslant i < n \wedge s = \sum_{k=0}^{i-1} t[k]|\!)$

$l_4 : (\!|0 \leqslant i < n \wedge s = \sum_{k=0}^{i} t[k]|\!)$

$l_5 : (\!|0 \leqslant i \leqslant n \wedge s = \sum_{k=0}^{i-1} t[k]|\!)$

$l_6 : (\!|i = n \wedge s = \sum_{k=0}^{n-1} t[k]|\!)$

Figure 2: The program and the local invariants

3.3 Liveness properties

Similarly, we will start by sketching an informal definition of a *liveness property*, before introducing an operator-based formal definition and a proof mechanism.

3.3.1 Informal definition

Definition (Informal definition of liveness properties)**.** A liveness property is a property which specifies that some (good) behavior *will eventually occur*, and that this behavior *may still occur after any finite observation*.

- *Termination* is a liveness property, in which the "good behavior" is reaching a blocking state (no more transition are available).
- "State a will enventually be reached by any execution" is also a liveness property, in which the "good behavior" is reaching the state a.
- The absence of runtime errors is **not** a liveness property.

We can also ask ourselves how to refute a lineveness property. Considering a liveness property $\mathcal{T}$ (i.e. $\mathcal{T}$ is termination), we assume that $\mathcal{S}$ does **not** satisfy liveness property $\mathcal{T}$. Thus, there exists a counter-example trace $\sigma \in [\![\mathcal{S}]\!] \setminus \mathcal{T}$. The informal definition says "... may still occur after any finite observation", thus each finite trace σ' can be extended into a good trace. Therefore, σ is necessarily *infinite*.

To prove that a liveness property does not hold, we need to look for an infinite counter-example: no finite trace is a counter-example.

For example, when $\mathcal{T}$ is the termination property, no finite execution can guarantee us that the program will not eventually stop; to prove that a program will not halt, we need to exhibit an infinite execution.

3.3.2 Formal definition

Definition (Operator Live)**.** The operator Live is defined by:

$$\mathrm{Live}(\mathcal{T}) = \mathcal{T} \cup (\mathbb{S}^\infty \setminus \mathrm{Safe}(\mathcal{T}))$$

Definition (Liveness property)**.** $\mathcal{T}$ is a liveness property if and only if it satisfies one of the three following equivalend statements:

(i) $\mathrm{Live}(\mathcal{T}) = \mathcal{T}$

(ii) $\mathrm{PCl}(\mathcal{T}) = \mathbb{S}^\star$ (any finite trace is the prefix of a trace in $\mathcal{T}$)

(iii) $\mathrm{Lim} \circ \mathrm{PCl}(\mathcal{T}) = \mathbb{S}^\infty$

Proof: $(i) \implies (ii)$. We assume that $\mathrm{Live}(\mathcal{T}) = \mathcal{T}$, i.e. that $\mathcal{T} \cup (\mathbb{S}^\infty \setminus \mathrm{Safe}(\mathcal{T})) = \mathcal{T}$. Therefore, $\mathbb{S}^\infty \setminus \mathrm{Safe}(\mathcal{T}) \subseteq \mathcal{T}$.

Let $\sigma \in \mathbb{S}^\star$ and let us show that $\sigma \in \mathrm{PCl}(\mathcal{T})$. Since we must have $\sigma \in \mathbb{S}^\infty$:

- either $\sigma \in \mathrm{Safe}(\mathcal{T}) = \mathrm{Lim}(\mathrm{PCl}(\mathcal{T}))$, so all its prefixes are in $\mathrm{PCl}(\mathcal{T})$ and $\sigma \in \mathrm{PCl}(\mathcal{T})$;
- or $\sigma \in \mathcal{T}$ which implies that $\sigma \in \mathrm{PCl}(\mathcal{T})$.

□

Proof: $(ii) \implies (iii)$. If $\mathrm{PCl}\,\mathcal{T} = \mathbb{S}^\star$, then $\mathrm{Lim} \circ \mathrm{PCl}(\mathcal{T}) = \mathbb{S}^\infty$. □

Proof: $(iii) \implies (i)$. If $\mathrm{Lim} \circ \mathrm{PCl}(\mathcal{T}) = \mathbb{S}^\infty$, then

$$\mathrm{Live}(\mathcal{T}) = \mathcal{T} \cup (\mathbb{S}^\infty \setminus (\mathrm{Lim} \circ \mathrm{PCl}(\mathcal{T}))) = \mathcal{T} \cup (\mathbb{S}^\infty \setminus \mathbb{S}^\infty) = \mathcal{T}$$

□

Example (Termination)**.** *The property $\mathcal{T} = \mathbb{S}^\star$ corresponds to termination: there should be no infinite execution. $\mathcal{T}$ clearly satisfies*

$$\mathrm{PCl}(\mathcal{T}) = \mathbb{S}^\star$$

thus termination is indeed a liveness property according to the formal definition.

Example. *Assume that $\mathbb{S} = \{a, b, c\}$ and that $\mathcal{T}$ is the set of states in which b has been visited after a has been visited. $\mathcal{T}$ can be described by $\mathcal{T} = \mathrm{PCl}(\mathbb{S}^\star \cdot a \cdot \mathbb{S}^\star \cdot b \cdot \mathbb{S}^\star)$.*

Then, $\mathcal{T}$ is a liveness property. Let $\sigma \in \mathbb{S}^\star$; then $\sigma \cdot a \cdot b \in \mathcal{T}$, so $\sigma \in \mathrm{PCl}\,T$. Thus, $\mathrm{PCl}\,\mathcal{T} = \mathbb{S}^\star$, which corresponds to property (ii).

Theorem (Live is idempotent)**.** *If $\mathcal{T}$ is a trace property, then* $\mathrm{Live}(\mathcal{T})$ *is a liveness property.*

Proof. Let's show that $\mathrm{PCl} \circ \mathrm{Live}(\mathcal{T}) = \mathbb{S}^\star$. Consider $\sigma \in \mathbb{S}^\star$, we will prove that $\sigma \in \mathrm{PCl} \circ \mathrm{Live}(\mathcal{T})$. Note that:

$$\begin{aligned}\mathrm{PCl} \circ \mathrm{Live}(\mathcal{T}) &= \mathrm{PCl}(\mathcal{T}) \cup \mathrm{PCl}(\mathbb{S}^\infty \setminus \mathrm{Safe}(\mathcal{T})) \\ &= \mathrm{PCl}(\mathcal{T}) \cup \mathrm{PCl}(\mathbb{S}^\infty \setminus \mathrm{Lim} \circ \mathrm{PCl}(\mathcal{T}))\end{aligned}$$

If $\sigma \in \mathrm{PCl}(\mathcal{T})$, the result is obvious. If $\sigma \notin \mathrm{PCl}(\mathcal{T})$, then $\sigma \notin \mathrm{Lim} \circ \mathrm{PCl}(\mathcal{T})$ by definition of the limit, thus $\sigma \in \mathbb{S}^\infty \setminus \mathrm{Lim} \circ \mathrm{PCl}(\mathcal{T})$. Therefore, $\sigma \in \mathrm{PCl}(\mathbb{S}^\infty \setminus \mathrm{Lim} \circ \mathrm{PCl}(\mathcal{T}))$ as PCl is extensive when applied to sets of finite traces, which proves the above result. □

3.3.3 Termination proof with ranking function

In this section, we will only consider termination. Given a transition system $\mathcal{S} = (\mathbb{S}, \to, \mathbb{S}_\mathbb{I})$, and a liveness property $\mathcal{T}$, we look for a way of verifying that $\mathcal{S}$ satisfies termination, i.e. that $[\![\mathcal{S}]\!]^\infty \subseteq \mathbb{S}^\star$. To do so, we introduce the notion of ranking function.

Definition (Ranking function)**.** A *ranking function* is a function $\phi : \mathbb{S} \to E$ where $(E, \sqsubseteq)$ is a *well-founded* ordering[1], and

$$\forall s_1, s_2 \in \mathbb{S}, s_0 \to s_1 \implies \phi(s_1) \sqsubset \phi(s_0)$$

Intuitively, following a transition can only make $\phi(s)$ striclty decrease, and $\phi(s)$ cannot keep decreasing forever, since the order is well-founded.

Theorem. *If $\mathcal{S}$ has a ranking function ϕ, it satisfies termination.*

Example. *We consider the termination of the array sum program. As seen on Figure 3.3.3, ϕ is a ranking function, hence the transition system of our program satistifes termination.*

```
1  s = 0;
2  i = 0;
3  while (i < n) {
4      s = s + t[i];
5      i = i + 1;
6  }
```

$$\begin{aligned}
\phi : \ \mathbb{S} &\longrightarrow \mathbb{N} \\
(l_1, m) &\longmapsto 3n + 6 \\
(l_2, m) &\longmapsto 3n + 5 \\
(l_3, m) &\longmapsto 3n + 4 \\
(l_4, m) &\longmapsto 3(n - m(i)) + 3 \\
(l_5, m) &\longmapsto 3(n - m(i)) + 2 \\
(l_6, m) &\longmapsto 3(n - m(i)) + 4 \\
(l_7, m) &\longmapsto 0
\end{aligned}$$

Figure 3: The program and the ranking function

3.3.4 Proof by variance

We consider a transition system $\mathcal{S} = (\mathbb{S}, \to, \mathbb{S}_\mathcal{I})$, and liveness property $\mathcal{T}$. Infinite traces semantics is the greatest fixpoint of F_ω. We seek a way of verifying that $\mathcal{S}$ satisfies $\mathcal{T}$, i.e. that $[\![\mathcal{S}]\!]^\infty \subseteq \mathcal{T}$.

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

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

Theorem. *The variance proof method is both sound and complete.*

3.4 Decomposition of trace properties

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

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

```
s = 0;
i = 0;
while (i < n) {
    s = s + t[i];
    i = i + 1;
}
```

Figure 4: A simple program which computes the sum of the elements of `t`

Example. *Considering the same program: we try to prove its total correctness. We need to show that the program terminates, does not crash, and computes the sum of the elements in the array. We can apply the decomposition principle and express total correctness as the* conjunction of two proofs*: termination can be proved with a ranking function, and the absence of errors and result correctness can be proved with local invariants.*

Example. *Likewise, we consider a very simple greatest common divider code function:*

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

Figure 5: A simple program which computes the sum of the elements of `t`

The specification of this function – that we want to prove – is: "When applied to positive integers, function f should always return their GCD". The safety part is the conjunction of two properties: no runtime errors, and the final value of b is the GCD. The liveness part is termination on all traces starting with positive inputs.

3.5 A specification language: temporal logic

3.5.1 Notion of specification language

Ultimately, we would like to *verify* or *compute* properties, but so far, we simply describe properties with sets of executions or worse, natural language statements. Ideally, we would prefer to use a mathematical language for that, to gain in concision and avoid ambiguity.

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

$$[\![\cdot]\!] : \mathbb{L} \to \mathcal{P}(\mathbb{S}^\infty)$$

[1] An ordering is well-founded when every non-empty subset of E has a minimal element for $\sqsubseteq$.

3.5.2 A State specification language

3.5.3 Propositional temporal logic

3.6 Beyond safety and liveness

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

For example, one can think about security properties. We can consider just one: "An unauthorized observer should not be able to guess anything about private information by looking at public information".

4 Satisfiability Modulo Theories (SMT)

Since 2000, SAT solvers have become extremly performant. SMT is the extension of SAT with clauses instead of simply variables.

4.1 Modern SAT solvers

4.1.1 The SAT problem

Given a boolean formula without quantifiers, e.g.

$$(p \vee q \vee \neg r) \wedge (r \vee \neg p)$$

we try to find if this formula is satisfiable. Techniques to do so include truth tables, resolution-based procedures, backtracking-based procedure. Since the 80s, there's been a focus on variable selection heuristics, associated with search-pruning techniques (non-chronological backtracking, learning clauses): these are the CDCL algorithms (Conflicts-Driven Clause Learning). New techniques such as indexing and scoring have since emerged.

4.1.2 Resolution

Resolution uses a proof-finder procedure, which tries to reduce the equation.

The state of the procedure is represented by a variable F, to which we can apply some rules (Resolve, Empty, Tauto, Subsume, Fail). This allows to construct a proof tree.

4.1.3 DPLL

DPLL is a model-finder procedure that builds incrementally a model M for a CNF formula F by

- deducing the truth value of a literal l from M and F by Boolean Constraint Propagations (BCP):

$$\text{If } C \vee l \in F \text{ and } M \vDash \neg C \text{ then } l \text{ must be true}$$

- guessing the truth value of an unassigned literal:

$$\begin{gathered}\text{If } M \cup \{l\} \text{ leads to a model for which } F \text{ is unsatisfiable} \\ \text{then } \textit{backtrack} \text{ and try } M \cup \{\neg l\}\end{gathered}$$

Similarly, DPLL is a set of rules (Success, Unit, Decide, Backtrack, Fail).

4.1.4 Clause conflict, backjump

DPLL remains slow: instead of backtracking, we would prefer to *backjump* multiple steps before, to the node where the conflit has indeed been decided. Conflicts are reflected by backjump clauses. Given a backjump clause $C \vee l$, backjumping can undo several decisions at once: it goes back to the assignment M where $M \vDash \neg C$.

4.1.5 CDCL algorithm

Conflict-Driven Clause Learning SAT solvers (CDCL) add backjump clauses to M as learned clauses (or lemmas) to prevent future similar conflicts.

Lemmas can also be removed from M.

The algorithm now has two modes: search and resolution.

4.1.6 Heuristics, Two-watched literals

The Variable State Independent Decaying Sum (VSIDS) heuristic associates a score to each literal in order to select the literal with the hightest score when Decide is used.

CDCL performances are tightly related to their learning clause management. Keeping too many clauses decrease the BCP efficency, but cleaning out too many clauses break the overall learning benefit.

Quality measures for learning clauses are based on scores associated with learned clauses.

BCP represents 80% of SAT-solver runtime. The two watched literals technique assigns two non-false watched literals per clause. Only if one of the two watched literal becomes false, the clause is inspected. If the other watched literal is assigned to true, the do nothing. Otherwise, try to find another (unassigned) watched literal. If no such literal exists, then apply Backjump. If the only possible literal is the oother watched literal of the clause, then apply Unit.

Its main advantages are that clauses are inspected only when watched literal are assigned, and that no updating is required when backjumping.