Software Model Checking

Predicate Abstraction with Automatic Precision Refinement

Prof. Nian-Ze Lee
nzlee@ntu.edu.tw

ForMACE Lab, National Taiwan University

2025-08-18

(Part of the slides are courtesy of SoSy-Lab, LMU Munich, Germany.)

Who Am I?

- A new faculty member at NTUEE/GIEE (starting from February, 2025)
- A PostDoc at LMU Munich, Germany, from 2021 to 2024
- A PhD graduate from GIEE in 2021
- A Bachelor's graduate from NTUEE in 2014

Problems with This Code

```
1   int binarySearch(int arr[], int left, int right, int target) {
2     while (left <= right) {
3         int mid = (left + right) / 2;
4         if (arr[mid] = target)
5             return mid;
6         if (arr[mid] < target)
7             left = mid;
8         else
9             right = mid;
10    }
11  }
```

Problems with This Code

```
1   int binarySearch(int arr[], int left, int right, int target) {
2     while (left <= right) {
3         // Bug 1: Potential integer overflow when computing mid
4         // Found in "java.util.Arrays" in 2006
5         int mid = (left + right) / 2;
6         // Bug 2: Incorrect comparison (assignment instead of comparison)
7         if (arr[mid] = target)
8             return mid;
9         // Bug 3: Incorrect updates to left and right may cause infinite loop
10        if (arr[mid] < target)
11            left = mid;
12        else
13            right = mid;
14    }
15    // Bug 4: Forgetting to return a value may cause undefined behavior
16  }
```

Problems with This Code

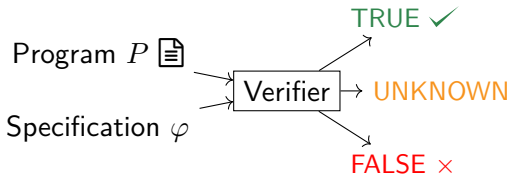
```
1   int binarySearch(int arr[], int left, int right, int target) {
2     while (left <= right) {
3         // Fix 1: Prevent overflow
4         int mid = left + (right - left) / 2;
5         // Fix 2: Use comparison
6         if (arr[mid] == target)
7             return mid;
8         // Fix 3: Update left and right correctly
9         if (arr[mid] < target)
10            left = mid + 1;
11        else
12            right = mid - 1;
13    }
14    // Fix 4: Return -1 to indicate target not found
15    return -1;
16  }
```

Demo: CPAchecker

- Automatic program analyzer based on **Configurable Program Analysis**
- Open-source tool: GitLab repository
- Found more than 100 bugs (confirmed and fixed) in Linux kernel modules
- Top contender at annual competitions for software verifiers (SV-COMP)

Software Model Checking

Formally proves if a program P satisfies a specification φ



Disprove (×) Find a program execution that violates φ (**counterexample**)

Prove (✓) Show that **every** program execution satisfies φ

Inconclusive Due to timeout, memory-out, etc.

Model Checking vs. Testing

- Static vs. Dynamic
- Syntax vs. Semantics: $M \vdash \varphi \iff M \models \varphi$
- Mathematical rigor: aim to cover all possibilities (hard!)

Practical Examples

- Software
  - Facebook: static analyzer Infer
  - Amazon: storage services, low-level C code
- Hardware
  - Intel: Pentium FDIV bug, CPU design
  - Cadence: commercial tools for IC verification
- Aerospace and avionics
  - NASA: robot controller
  - Airbus: flight system (*unit proof*)

History and Foundations

- *Design and synthesis of synchronization skeletons using branching time temporal logic*, Edmund M. Clarke and E. Allen Emerson, 1981
- *Specification and verification of concurrent systems in CESAR*, Jean-Pierre Queille and Joseph Sifakis, 1982
- Turing Award 2007

History and Foundations

- *Design and synthesis of synchronization skeletons using branching time temporal logic*, Edmund M. Clarke and E. Allen Emerson, 1981
- *Specification and verification of concurrent systems in CESAR*, Jean-Pierre Queille and Joseph Sifakis, 1982
- Turing Award 2007
- Foundations
  - Automata theory (for representing models)
  - Temporal logic (for encoding specifications)
  - Symbolic representation (e.g., binary decision diagrams)
  - Constraint solving (e.g., satisfiability)
  - Abstraction (e.g., predicate abstraction)
  - ...

Today's Lecture

- Morning
  - Lattices
  - Program representation, semantics, and analysis
  - Configurable program analysis (CPA)
- Afternoon
  - Cartesian predicate abstraction
  - Craig interpolation
  - Counterexample-guided abstraction refinement (CEGAR)
  - Automatic precision refinement using CEGAR

Pioneering and Related Papers in the Literature

- *Construction of Abstract State Graphs with PVS*, CAV 1997
- *Counterexample-Guided Abstraction Refinement*, CAV 2000
- *Interpolation and SAT-Based Model Checking*, CAV 2003
- *Abstractions from Proofs*, POPL 2004

- *A Unifying View on SMT-Based Software Verification*, JAR 2018
- *Interpolation and SAT-Based Model Checking Revisited: Adoption to Software Verification*, JAR 2025

Lattices

Partial and Total Orders

Definition (Partial Order)

Let E be a set and $\sqsubseteq \subseteq E \times E$ be a binary relation on E.
The tuple $(E, \sqsubseteq)$ is a *partial order* if $\sqsubseteq$ is:

- **Reflexive**
  $\forall e \in E, e \sqsubseteq e$
- **Antisymmetric**
  $\forall e_1, e_2 \in E, (e_1 \sqsubseteq e_2 \land e_2 \sqsubseteq e_1) \Rightarrow e_1 = e_2$
- **Transitive**
  $\forall e_1, e_2, e_3 \in E, (e_1 \sqsubseteq e_2 \land e_2 \sqsubseteq e_3) \Rightarrow e_1 \sqsubseteq e_3$

Definition (Total Order)

A partial order $(E, \sqsubseteq)$ is a *total order* if $\forall e, e' \in E. e \sqsubseteq e' \lor e' \sqsubseteq e$.

Examples of Partial Orders

- $(\mathbb{Z}, \leq)$
- $(2^S, \subseteq)$
- $(\Sigma^*, \text{lexicographic order})$
- $(\Sigma^*, \text{suffix})$

Chains

Let $(E, \sqsubseteq)$ be a partial order.

Definition (Chain)

A subset $E_{\text{sub}} \subseteq E$ is a *chain* if $(E_{\text{sub}}, \sqsubseteq)$ is a total order.
(A chain E_{sub} is finite if $|E_{\text{sub}}| \in \mathbb{N}$.)

Definition (Ascending Chain)

A sequence $(e_1, e_2, e_3, \ldots)$ is an *ascending chain* if $\forall n, n' \in \mathbb{N}.\, n \leq n' \Rightarrow e_n \sqsubseteq e_{n'}$.
(This condition implies $\{e_1, e_2, e_3, \ldots\}$ is a chain.)

Definition (Stabilization)

An ascending chain $(e_1, e_2, e_3, \ldots)$ is *stabilizing* if
$\exists n_0 \in \mathbb{N}, \forall n \in \mathbb{N}.\, n \geq n_0 \Rightarrow e_n = e_{n_0}$.

Examples of Chains

Consider partial order $(\mathbb{Z}, =)$.

- Set {1,2} not a chain.
- $(1, 1, 1, 1, \dots)$ is ascending and stabilizing.

Consider partial order $(\mathbb{Z}, \leq)$.

- Every subset of $\mathbb{Z}$ is a chain.
- $(e_1, e_2, \dots)$ with $e_i = \begin{cases} 0 & \text{if } i \text{ even} \\ 1 & \text{else} \end{cases}$ not ascending.
- $(e_1, e_2, \dots)$ with $e_i = i$ is ascending but not stabilizing.
- $(e_1, e_2, \dots)$ with $e_i = \min(i, 10)$ is ascending and stabilizing.
- No infinite subset of $\mathbb{Z}$ is a stabilizing ascending chain.

Height of Partial Orders

Let $(E, \sqsubseteq)$ be a partial order.

- $(E, \sqsubseteq)$ has a finite height if all chains are finite.
- $(E, \sqsubseteq)$ has height h if all chains contain at most $h + 1$ elements and one chain contains exactly $h + 1$ elements.
  (The longest ascending chain with distinct elements has $h + 1$ elements.)

Remark: If $|E| \in \mathbb{N}$, $(E, \sqsubseteq)$ finite height (but not vice versa).

Example?

Upper Bounds

Let $(E, \sqsubseteq)$ be a partial order.

Definition (Upper Bound)

An element $e \in E$ is an upper bound of a subset $E_{\text{sub}} \subseteq E$ if

$$\forall e' \in E_{\text{sub}}.\, e' \sqsubseteq e.$$

Definition (Least Upper Bound (LUB))

An element $e \in E$ is a least upper bound of $E_{\text{sub}} \subseteq E$ if

- e is an upper bound of E_{sub} and
- for all upper bounds e' of E_{sub}, $e \sqsubseteq e'$.

Lower Bounds

Let $(E, \sqsubseteq)$ be a partial order.

Definition (Lower Bound)

An element $e \in E$ is a lower bound of a subset $E_{\text{sub}} \subseteq E$ if

$$\forall e' \in E_{\text{sub}}.\, e \sqsubseteq e'.$$

Definition (Greatest Lower Bound (GLB))

An element $e \in E$ is a greatest lower bound of $E_{\text{sub}} \subseteq E$ if

- e is a lower bound of E_{sub} and
- for all lower bounds e' of E_{sub}, $e' \sqsubseteq e$.

Facts about Upper and Lower Bounds

1. LUBs and GLBs do not always exist. Examples?
2. The least upper bound and the greatest lower bound are unique if they exists. Why?

(Bounded) Lattices

Definition (Lattice)

A tuple $\mathcal{E} = (E, \sqsubseteq, \sqcup, \sqcap, \top, \bot)$ is a *lattice* if

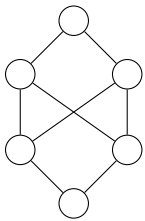
- $(E, \sqsubseteq)$ is a partial order,
- $\sqcup : 2^E \mapsto E$ is a *join operator* that computes the LUB of a subset $E_{\text{sub}} \subseteq E$ (LUB exists for all subsets),
- $\sqcap : 2^E \mapsto E$ is a *meet operator* that computes the GLB of a subset $E_{\text{sub}} \subseteq E$ (GLB exists for all subsets),
- $\top = \sqcup E = \sqcap \emptyset$ (top element), and
- $\bot = \sqcap E = \sqcup \emptyset$ (bottom element).

Example:
For any set S, the tuple $(2^S, \subseteq, \cup, \cap, S, \emptyset)$ is a lattice.

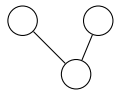
Which Partial Orders Are Lattices?

(a) (b)

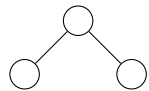
(c)



(d)



(e)



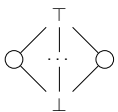
(f)

Flat Lattices

Definition (Flat Lattice)

A *flat lattice* of set E consists of

- an extended set $E_{\bot}^{\top} = E \cup \{\top, \bot\}$ and
- a flat ordering $\sqsubseteq$: $\forall e \in E. \bot \sqsubseteq e \sqsubseteq \top$ and $\bot \sqsubseteq \top$.
- $\sqcup = \begin{cases} \bot & E_{\text{sub}} = \emptyset \vee E_{\text{sub}} = \{\bot\} \\ e & E_{\text{sub}} = \{e\} \vee E_{\text{sub}} = \{\bot, e\} \\ \top & \text{else} \end{cases}$
- $\sqcap = \begin{cases} \top & E_{\text{sub}} = \emptyset \vee E_{\text{sub}} = \{\top\} \\ e & E_{\text{sub}} = \{e\} \vee E_{\text{sub}} = \{\top, e\} \\ \bot & \text{else} \end{cases}$



E (for example, program locations)

Product Lattices

Let $\mathcal{E}_1 = (E_1, \sqsubseteq_1, \sqcup_1, \sqcap_1, \top_1, \bot_1)$ and $\mathcal{E}_2 = (E_2, \sqsubseteq_2, \sqcup_2, \sqcap_2, \top_2, \bot_2)$ be lattices.

The *product lattice* $\mathcal{E}_\times = (E_1 \times E_2, \sqsubseteq_\times, \sqcup_\times, \sqcap_\times, \top_\times, \bot_\times)$ with

- $(e_1, e_2) \sqsubseteq_\times (e_1', e_2')$ if $e_1 \sqsubseteq_1 e_1' \wedge e_2 \sqsubseteq_2 e_2'$,
- $\sqcup_\times E_{\text{sub}} = (\sqcup_1\{e_1 \mid (e_1, \cdot) \in E_{\text{sub}}\}, \sqcup_2\{e_2 \mid (\cdot, e_2) \in E_{\text{sub}}\})$,
- $\sqcap_\times E_{\text{sub}} = (\sqcap_1\{e_1 \mid (e_1, \cdot) \in E_{\text{sub}}\}, \sqcap_2\{e_2 \mid (\cdot, e_2) \in E_{\text{sub}}\})$,
- $\top_\times = (\top_1, \top_2)$ and $\bot_\times = (\bot_1, \bot_2)$.

is a lattice.

Note: Program states

Join Semi-Lattices

Complete lattice always not required
$\Rightarrow$ remove unused elements

Definition (Join Semi-Lattice)

A tuple $\mathcal{E} = (E, \sqsubseteq, \sqcup, \top)$ is a *join semi-lattice* if

- $(E, \sqsubseteq)$ is a partial order,
- $\sqcup : 2^E \mapsto E$ is a *join operator* that computes the LUB of a subset $E_{\text{sub}} \subseteq E$ (LUB exists for all subsets), and
- $\top = \sqcup E = \sqcap \emptyset$ (top element).

Note: Program abstractions

Program Representation

Programs

Theory: simple while-programs

- **Integer** constants and variables
- Minimal set of statements (assignment, if-else, while)

Practice: C programs

- Widely-used language
- Tool support by many software verifiers

While-Programs

- Arithmetic expressions
  aexpr := $\mathbb{Z}$ | `var` | -aexpr | aexpr op_a aexpr
  - op_a arithmetic operator: $+, -, *, /, \%, \ldots$
- Boolean expressions
  bexpr := aexpr | aexpr op_c aexpr | !bexpr | bexpr op_b bexpr
  - Value 0 represents false; other values represent true.
  - op_c comparison operator: $<, <=, >=, >, ==, !=$
  - op_b Boolean connective: $\&\&(\wedge), ||\ (\vee),$ ^ (xor), $\ldots$
- Program
  S:= `var=aexpr;` | `while` bexpr S | `if` bexpr S `else` S |
  `if` bexpr S | S;S
  - Assume: `bexpr` (true or false)
  - Assignment: `var=aexpr;`

How to Represent a Program?

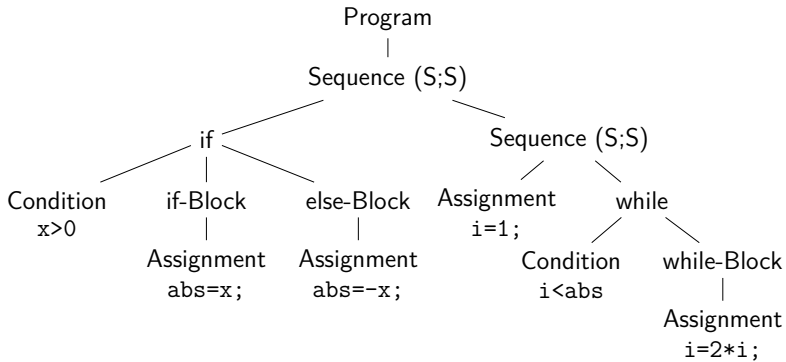
1. Source code

```
1  if (x>0)
2      abs = x;
3  else
4      abs = −x;
5  i = 1;
6  while(i<abs)
7      i = 2∗i;
```

- A sequence of characters
- No explicit information about the structure or paths of programs

How to Represent a Program?

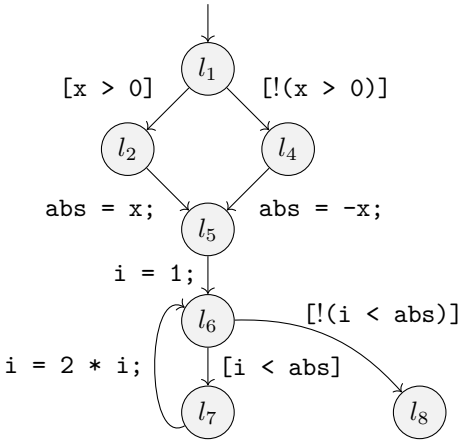
2. Abstract-syntax tree (AST)



- Hierarchical representation w.r.t syntax rules
- Hard to detect program paths

How to Represent a Program?

3. Control-flow automaton (CFA)




```
1  if (x>0)
2      abs = x;
3  else
4      abs = −x;
5  i = 1;
6  while(i<abs)
7      i = 2∗i;
```

Remark: indices of l_i do not need to match line numbers.

Control-Flow Automata

Definition (CFA)

A *control-flow automaton* (CFA) is a three-tuple $P = (L, l_0, G)$ consisting of

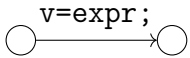
- a set L of program locations (domain of program counter),
- an initial program location $l_0 \in L$, and
- a set $G \subseteq L \times Ops \times L$ of control-flow edges.

Operations Ops

- Assume: `bexpr` (Boolean expressions, true or false)
- Assignment: `var=aexpr;`
  (variable assignment, update variable values)

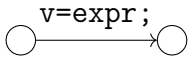
From Source Code to Control-Flow Automaton

Assignment `var=expr;`

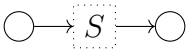


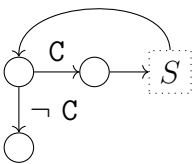
From Source Code to Control-Flow Automaton

Assignment `var=expr;`



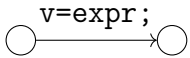
While-Statement `while` (C) S



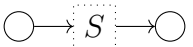


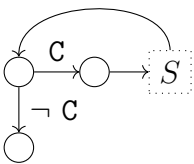
From Source Code to Control-Flow Automaton

Assignment `var=expr;`

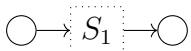


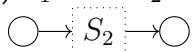
While-Statement `while` (C) S

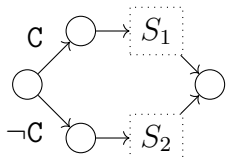




If-Statement `if` (C) S_1 `else` S_2

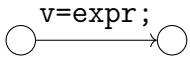




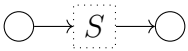


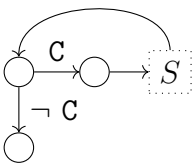
From Source Code to Control-Flow Automaton

Assignment `var=expr;`

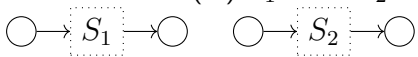


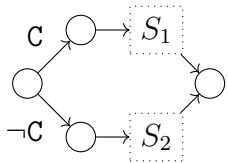
While-Statement `while` (C) S



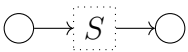


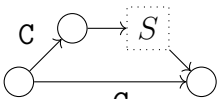
If-Statement `if` (C) S_1 `else` S_2





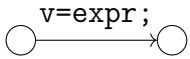
If-Statement `if` (C) S



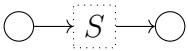


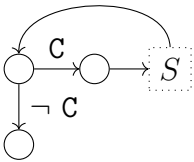
From Source Code to Control-Flow Automaton

Assignment `var=expr;`

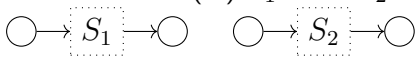


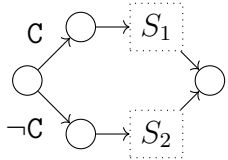
While-Statement `while` (C) S



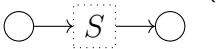


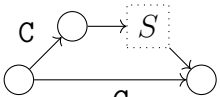
If-Statement `if` (C) S_1 `else` S_2



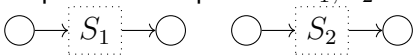


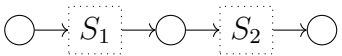
If-Statement `if` (C) S





Sequential Composition $S_1; S_2$





Program Semantics

Semantics of a Program: Concrete States

Given a CFA $P = (L, l_0, G)$, a *concrete state* $c = (pc, \sigma)$ is a pair of program counter pc and data state σ.

- Program counter pc
  - Location in CFA
- Data state $\sigma : \mathrm{Vars}(P) \to \mathbb{Z} \cup \{\top\}$
  - Values of variables (unknown values represented by $\top$)

We use $C = L \times \Sigma$ to represent the set of concrete states, where L is the set of program counters, Σ is the set of data states.

- We will use $c(pc)$ and $c(\sigma)$ to denote the program counter and the data state of c, respectively.

State Update

Given a concrete state $c = (pc = l, \sigma)$ and a control-flow edge $g = (l, op, l')$, the concrete state c' after executing op:

- **Assignment**: $g = (l, v = aexpr, l')$
  - Update program counter from l to l'
  - Update the value of variable v in the data state (evaluate $aexpr$ over the current data state)
  - Ex: next state of $(l_0, \{x \mapsto 1\})$ after executing $(l_0, \texttt{x=x+2;}, l_1)$?
- **Assume**: $g = (l, bexpr, l')$
  - Update program counter from l to l' if $bexpr$ evaluates to true over the current data state
  - Data state remains unchanged.
  - Ex: next state of $(l_0, \{x \mapsto 1\})$ after executing $(l_0, \texttt{[x<2]}, l_2)$?

State Update

Given a concrete state $c = (pc = l, \sigma)$ and a control-flow edge $g = (l, op, l')$, the concrete state c' after executing op:

- **Assignment**: $g = (l, v = aexpr, l')$
  - Update program counter from l to l'
  - Update the value of variable v in the data state (evaluate $aexpr$ over the current data state)
  - Ex: next state of $(l_0, \{x \mapsto 1\})$ after executing $(l_0, \texttt{x=x+2;}, l_1)$?
- **Assume**: $g = (l, bexpr, l')$
  - Update program counter from l to l' if $bexpr$ evaluates to true over the current data state
  - Data state remains unchanged.
  - Ex: next state of $(l_0, \{x \mapsto 1\})$ after executing $(l_0, \texttt{[x<2]}, l_2)$?

(c, g, c') is a valid transition if the above conditions hold.

Program Paths

Defined inductively

- Every concrete state c with $c(pc) = l_0$ is a program path.
- If $c_0 \xrightarrow{g_1} c_1 \cdots \xrightarrow{g_n} c_n$ is a program path and (c_n, g_{n+1}, c_{n+1}) is valid, $c_0 \xrightarrow{g_1} c_1 \cdots \xrightarrow{g_n} c_n \xrightarrow{g_{n+1}} c_{n+1}$ is a program path.

Specifications

- Linear temporal logic (LTL) formulas
- Interpreted over program paths

Specifications

- Linear temporal logic (LTL) formulas
- Interpreted over program paths

Definition (Reachability-Safety Verification Task)

Given a CFA $P = (L, l_0, G)$ and an error location $l_{ERR} \in L$, a *reachability-safety verification task* is to check if there exists a program path from l_0 to l_{ERR}.
In other words, the specification is $\varphi = \mathtt{G}(l \neq l_{ERR})$.

Challenges for Program Analysis

- Infinitely many program states
- Infinitely many program paths (in general, with loops)

How to Deal with Infinity?

Idea: analyze (infinitely many) states and paths together!

- Merge concrete states into abstract states
  (e.g., when program paths converge)
- Define (abstract) semantics for abstract states

$\Rightarrow$ Abstract domain

Abstract Domains

Definition (Abstract Domain)

An abstract domain $D = (C, \mathcal{E}, [\![\cdot]\!])$ consists of:

- A set C of concrete states
- A join semi-lattice $\mathcal{E} = (E, \sqsubseteq, \sqcup, \top)$
- A concretization function $[\![\cdot]\!] : E \to 2^C$
  (give meanings to abstract states)
  - $[\![\top]\!] = C$
  - $\forall E_{\text{sub}} \subseteq E.\ \bigcup_{e \in E_{\text{sub}}} [\![e]\!] \subseteq [\![\sqcup E_{\text{sub}}]\!]$
    (overapproximating join operator)
  - $\forall e_1, e_2 \in E.\ e_1 \sqsubseteq e_2 \Rightarrow [\![e_1]\!] \subseteq [\![e_2]\!]$ (implied)

Example: Concretization

Consider $D = (\mathbb{Z}, \mathcal{E}, [\![\cdot]\!])$, where $\mathcal{E} = (E, \sqsubseteq, \sqcup, \top)$ is a lattice of intervals. Given an interval $e \in E$, $[\![e]\!] = \{i | i \in e\}$. For example, $[\![[2, 4]]\!] = \{2, 3, 4\}$. Is $[\![\cdot]\!]$ a concretization function?

Abstract Transfer Relations

Used to describe how abstract states evolve: $\rightsquigarrow \subseteq E \times G \times E$

- $\forall e \in E, g \in G.$
  $\bigcup_{c \in [\![e]\!]} \{c' \mid (c, g, c') \text{ is valid}\} \subseteq \bigcup_{(e,g,e') \in \rightsquigarrow} [\![e']\!]$

Abstract Transfer Relations

Used to describe how abstract states evolve: $\rightsquigarrow \subseteq E \times G \times E$

- $\forall e \in E, g \in G.$
  $\bigcup_{c \in [\![e]\!]} \{c' \mid (c, g, c') \text{ is valid}\} \subseteq \bigcup_{(e,g,e') \in \rightsquigarrow} [\![e']\!]$

Safe overapproximation: All concrete next states are covered by the LUB of the abstract next states.

Abstract Transfer Relations

Used to describe how abstract states evolve: $\rightsquigarrow \subseteq E \times G \times E$

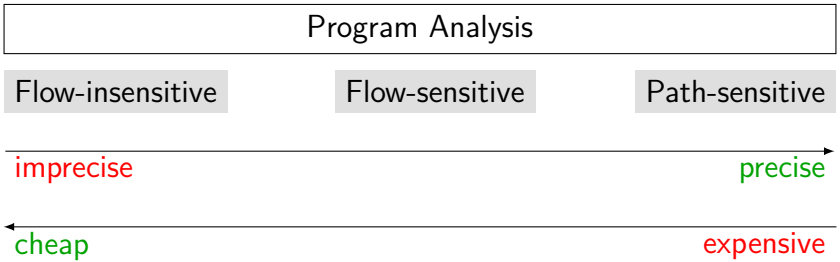
- $\forall e \in E, g \in G.$
  $\bigcup_{c \in [\![e]\!]} \{c' \mid (c, g, c') \text{ is valid}\} \subseteq \bigcup_{(e,g,e') \in \rightsquigarrow} [\![e']\!]$

Safe overapproximation: All concrete next states are covered by the LUB of the abstract next states.

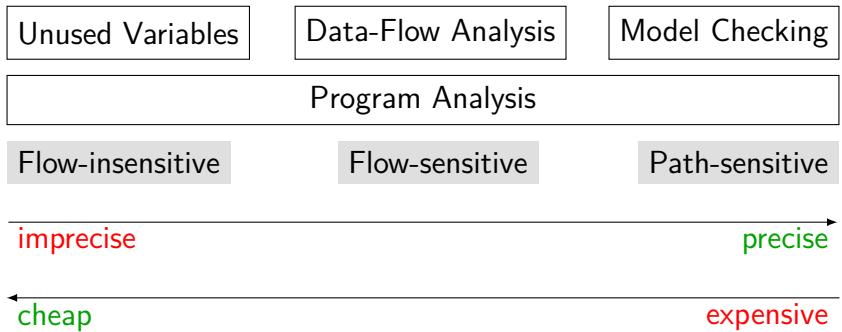
- How to define an abstract transfer relation for intervals?

Common Program Analyses

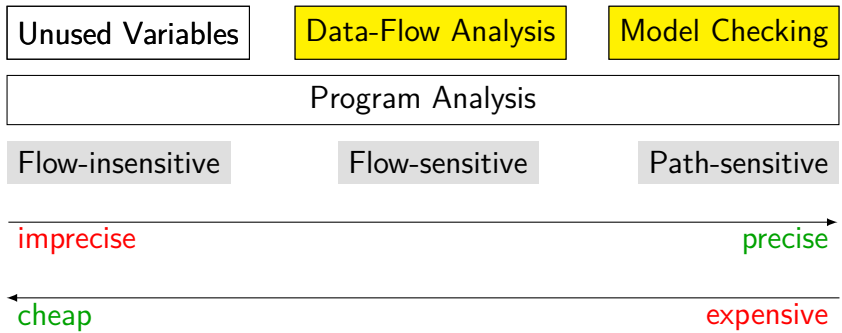
Styles of Program Analysis: Precision vs. Cost



Styles of Program Analysis: Precision vs. Cost



Styles of Program Analysis: Precision vs. Cost



Data-Flow Analysis: Ideas

- Compute one fact (abstract state) per program location
- Merge information from different program paths when they meet $\Rightarrow$ efficient but often imprecise
  - Use *least upper bound* to overapproximate several concrete states

Data-Flow Analysis: Algorithm

Input: a set L of locations, an abstract domain D,
an abstract transfer relation $\rightsquigarrow$,
an initial abstract state (l_0, e_0) with $l_0 \in L$, $e_0 \in E$ of D (the set of abstract states of the domain)

Output: the set of reachable abstract states (**ARG**)

1: waitlist := $\{(l_0, e_0)\}$;
2: reached := $\{(l_0, e_0)\}$;
3: **while** (waitlist $\neq \emptyset$) **do**
4: choose (l, e) from waitlist;
5: waitlist := waitlist $\setminus \{(l, e)\}$;
6: **for** each (l', e') with $(l, e) \rightsquigarrow (l', e')$ **do**
7: $e'' := (l', \widehat{e}) \in$ reached ? $\widehat{e}$: $\bot$;
8: **if** $(e' \not\sqsubseteq e'')$ **then**
9: reached := reached $\cup \{(l', e' \sqcup e'')\} \setminus \{(l', e'')\}$;
10: waitlist := waitlist $\cup \{(l', e' \sqcup e'')\} \setminus \{(l', e'')\}$;
11: **return** reached

Data-Flow Analysis: Remarks

- Efficient: at most $|L|$ abstract states in the reached set
- Less precise: overapproximation by "join" abstract states
- Termination?

Model Checking: Ideas

- Higher precision (path-sensitive) than data-flow analysis
- Keep information separate (never "join" abstract states)

Model Checking: Algorithm

Input: a set L of locations, an abstract domain D,
an abstract transfer relation $\rightsquigarrow$,
an initial abstract state (l_0, e_0) with $l_0 \in L$, $e_0 \in E$ of D (the set of abstract states of the domain)

Output: the set of reachable abstract states (**ARG**)

1: waitlist := $\{(l_0, e_0)\}$;
2: reached := $\{(l_0, e_0)\}$;
3: **while** (waitlist $\neq \emptyset$) **do**
4: choose (l, e) from waitlist;
5: waitlist := waitlist $\setminus \{(l, e)\}$;
6: **for** each (l', e') with $(l, e) \rightsquigarrow (l', e')$ **do**
7: **if** $(\nexists (l', e'') \in$ reached. $e' \sqsubseteq e'')$ **then**
8: reached := reached $\cup \{(l', e')\}$;
9: waitlist := waitlist $\cup \{(l', e')\}$;
10: **return** reached

Model Checking: Remarks

- Precise: no overapproximation by "join" abstract states
- Less efficient: How many abstract states in reached?
- Termination?

Data-Flow Analysis vs. Model Checking

Given a newly discovered abstract state (l', e'):

- DF: reached $:=$ reached $\cup \{(l', e' \sqcup e'')\} \setminus \{(l', e'')\}$;
- MC: reached $:=$ reached $\cup \{(l', e')\}$;

Data-Flow Analysis vs. Model Checking

Given a newly discovered abstract state (l', e'):

- DF: reached := reached $\cup \{(l', e' \sqcup e'')\} \setminus \{(l', e'')\}$;
- MC: reached := reached $\cup \{(l', e')\}$;

Can we have a universal framework to represent different program analyses?

Data-Flow Analysis vs. Model Checking

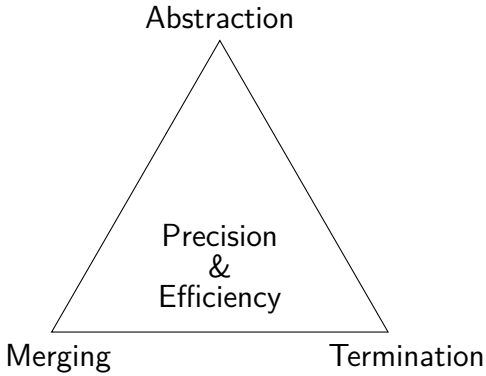
Given a newly discovered abstract state (l', e'):

- DF: reached $:=$ reached $\cup \{(l', e' \sqcup e'')\} \setminus \{(l', e'')\}$;
- MC: reached $:=$ reached $\cup \{(l', e')\}$;

Can we have a universal framework to represent different program analyses?
$\rightarrow$ **Configurable Program Analysis**

Configurable Program Analysis

Exploring the Configuration Space



- Abstraction: What aspects of a program to track?
- Merging: How to group abstract states?
- Termination: When to stop exploring a new abstract state?

Configurable Program Analysis (CPA)

A CPA $\mathbb{D}$ is a 4-tuple $(D, \rightsquigarrow, \text{merge}, \text{stop})$, consisting of:

- **Abstract domain**: $D = (C, (E, \sqsubseteq, \sqcup, \top), [\![\cdot]\!])$
- **Abstract transfer relation**: $\rightsquigarrow \subseteq E \times G \times E$
- **Merge operator**: $\text{merge} : E \times E \to E$

$$\forall e_1, e_2 \in E : e_2 \sqsubseteq \text{merge}(e_1, e_2)$$

- **Stop operator**: $\text{stop} : E \times 2^E \to \{true, false\}$

$$\forall e \in E, E_{\text{sub}} \subseteq E. \text{stop}(e, E_{\text{sub}}) \Rightarrow ([\![e]\!] \subseteq \bigcup_{e' \in E_{\text{sub}}} [\![e']\!])$$

A CPA (data structure) will be used by the CPA algorithm to construct ARG.

CPA Algorithm

Input: $P = (L, \ell_0, G)$, a CPA $\mathbb{D} = (D, \rightsquigarrow, \text{merge}, \text{stop})$, an initial abstract state $e_0 \in E$
Output: the set of reachable abstract states (**ARG**)

```
1: reached={e_0}; waitlist={e_0};
2: while (waitlist ≠ ∅) do
3:     pop e from waitlist;
4:     for each e ⇝ e' do
5:         for each e_r ∈ reached do
6:             e_m := merge(e', e_r)
7:             if (e_m ≠ e_r) then
8:                 reached := (reached \{e_r}) ∪ {e_m};
9:                 waitlist := (waitlist \{e_r}) ∪ {e_m};
10:        if (¬stop(e', reached)) then
11:            reached := reached ∪{e'};
12:            waitlist := waitlist ∪{e'};
13: return reached
```

Merge Operator

Defines when and how to combine abstract states

$$\text{merge} : E \times E \to E$$

Soundness criterion: The merged abstract state must subsume the second parameter (already explored state in the reached set).

$$\forall e_1, e_2 \in E.\, e_2 \sqsubseteq \text{merge}(e_1, e_2)$$

Example Merge Operators

- Joining abstract states: $\text{merge}^{join}(e_1, e_2) = \sqcup\{e_1, e_2\}$
- Separating abstract states: $\text{merge}^{sep}(e_1, e_2) = e_2$

Example Merge Operators

- Joining abstract states: $\text{merge}^{join}(e_1, e_2) = \sqcup\{e_1, e_2\}$
- Separating abstract states: $\text{merge}^{sep}(e_1, e_2) = e_2$

- Data-flow analysis:
  $$\text{merge}^{join}((l_1, e_1), (l_2, e_2)) = \begin{cases} (l_1, \sqcup\{e_1, e_2\}) & \text{if } l_1 = l_2 \\ (l_2, e_2) & \text{else} \end{cases}$$
- Model checking: $\text{merge}^{sep}((l_1, e_1), (l_2, e_2)) = (l_2, e_2)$

Stop Operator

Defines when to stop exploration (termination check)

$$\mathsf{stop} : E \times 2^E \to \{true, false\}$$

Soundness criterion: Exploration can be stopped only if the abstract state is covered by the second parameter (set of explored states).

$$\forall e \in E, E_{\text{sub}} \subseteq E.\, \mathsf{stop}(e, E_{\text{sub}}) \Rightarrow (\llbracket e \rrbracket \subseteq \bigcup_{e' \in E_{\text{sub}}} \llbracket e' \rrbracket)$$

Examples Stop Operators

- Never stop: $\text{stop}^{never}(e, E_{\text{sub}}) = false$
- When some explored abstract state subsumes e:
  $\text{stop}^{sep}(e, E_{\text{sub}}) = \exists e' \in E_{\text{sub}}.\, e \sqsubseteq e'$
- (If D is a power-set domain, where $\bigcup_{e' \in E_{\text{sub}}} [\![e']\!] = [\![\sqcup E_{\text{sub}}]\!]$)
  When the LUB of reached subsumes e: $\text{stop}^{join}(e, E_{\text{sub}}) = e \sqsubseteq \sqcup E_{\text{sub}}$

Modified Variable Analysis as CPA

- Abstract domain $D = (C, (E, \sqsubseteq, \sqcup, \top), [\![\cdot]\!])$
  - Join semi-lattice $(2^{Var}, \subseteq, \cup, Var)$
  - $[\![e]\!] = C$ (unimportant)
- Abstract transfer relation: $\rightsquigarrow (e, v = expr;) = e \cup \{v\}$ and $\rightsquigarrow (e, expr) = e$
- Merge operator: $\text{merge}^{join}(e_1, e_2) = \sqcup\{e_1, e_2\}$
- Stop operator $\text{stop}^{sep}(e, E_{\text{sub}}) = \exists e' \in E_{\text{sub}}.\, e \sqsubseteq e'$

Composition of CPAs

A composite CPA $\mathbb{D} = ((\mathbb{D}_1, \mathbb{D}_2, \ldots, \mathbb{D}_n), \rightsquigarrow_\times, \mathsf{merge}_\times, \mathsf{stop}_\times)$:

- Component CPAs: $(\mathbb{D}_1, \mathbb{D}_2, \ldots, \mathbb{D}_n)$
- Composite transfer relation: $\rightsquigarrow_\times \subseteq (E_1 \times \ldots \times E_n) \times G \times (E_1 \times \ldots \times E_n)$
- Composite merge operator: merge two abstract states when all component merge operators agree
- Composite stop operator: stop exploring an abstract state when all component abstract states are covered
- Information sharing between component CPAs via the strengthening operator (details in the Handbook Section 16.3.6)

Data-Flow Analysis with Value Abstraction as Composite CPA

Composition of LocationCPA and ValueAnalysisCPA:

- Abstract domain: $\mathbb{L} \times \mathbb{V}$
- Abstract transfer relation: $\rightsquigarrow_{\mathbb{L}\times\mathbb{V}}$
- Composite merge operator:

$$\text{merge}((l_1, v_1), (l_2, v_2)) = \begin{cases} (l_1, \sqcup\{v_1, v_2\}) & \text{if } l_1 = l_2 \\ (l_2, v_2) & \text{else} \end{cases}$$

- Composite stop operator: $\text{stop}((l, v), E_{\text{sub}}) = \exists (l, v') \in E_{\text{sub}}.\, (l, v) \sqsubseteq (l, v')$

Termination of CPA Algorithm

- Generally not guaranteed (inherited from model checking)
- Depends on configuration
  (may not terminate even for loop-free programs, e.g., with stop^{never})

Soundness

The returned reached set (ARG) overapproximates all reachable states of program P ($reach(P)$), if the initial abstract state e_0 covers all initial states:

$$\{c \mid c(pc) = l_0\} \subseteq [\![e_0]\!] \Rightarrow reach(P) \subseteq \bigcup_{e \in \text{reached}} [\![e]\!]$$

Reasons

- CPA algorithm explores all successors of states in reached (always adds states to waitlist if added to reached).
- Abstract transfer relation overapproximates concrete transfer relation.
- Merge operator replaces states by more abstract ones.
- Stop operator adds abstract successors if not covered.

Strongest Postconditions

– Capturing program semantics with first-order logical formulas –

Motivation

- At a program location, there may be multiple data states:
  - Ex: $(l, \{x \mapsto 0\}), (l, \{x \mapsto 2\}), (l, \{x \mapsto 4\}), \ldots$
- Represent all possible data states as a first-order logic (FOL) formula
  - Ex: $(l, x\%2 = 0)$
  - The represented data states are models of the formula.
- Operation on a program edge can also be represented as an FOL formula.
- Deduce how program evolves using *Satisfiability Modulo Theories* (SMT)
  - For software verification, we often use the theory of bit-vectors and arrays.

Strongest Postconditions

Goal: describe the most precise set of successor data states
$SP : Ops \times FOL \mapsto FOL$

Given an FOL f and an operation:

- Assume $expr$: $SP([expr], f) = f \wedge expr$
- Assignment $v = expr;$

$$SP(v = expr;, f) = \exists v'.\, f|_{v \to v'} \wedge v = expr|_{v \to v'}$$

Extension to a sequence of operations (i.e., a program path):

- $SP(\varepsilon, f) = f$
- $SP((op_1, \ldots, op_n), f) = SP((op_2, \ldots, op_n), SP(op_1, f))$

Examples for Strongest Postconditions

Consider the following sequences of operations G_i. Compute the strongest post condition $SP(G_i, true)$.

- $G_1 : ([x > 0], y = 1; , [x <= 0])$
- $G_2 : ([x > 0], y = 2; , [x > 0])$
- $G_3 : ([!(x > 0)], y = -1; , [y == 0])$
- $G_4 : (y = 0; , [x > 0], y = 5; , [x > 10])$

Which formulas are satisfiable?
What does a satisfiable postcondition mean?
What is the difference between $y = 1;$ and $y = 1$?

Static Single-Assignment (SSA) Form

Goal: eliminate quantifiers from FOLs for data states (resulting from assignments)

Idea borrowed from compiler design:

- Every variable is only assigned once.
- For each program variable, use a sequence of indexed variables, i.e., x_i for $\mathtt{x}$
- Start with index 0; increment by one when variable is assigned
- Evaluate expressions with the largest index of the variables
- When program paths converge, equalize SSA indices on both paths.

Cartesian Predicate Abstraction

– Deducing program correctness from small facts –

Predicate Abstraction for Programs

- Analyzing infinite-state systems by constructing finite abstractions
- Idea: tracking small, finitely many facts (analogous to human analysis)

Predicate Abstraction for Programs

- Analyzing infinite-state systems by constructing finite abstractions
- Idea: tracking small, finitely many facts (analogous to human analysis)
- Atomic predicate: a Boolean expression without Boolean connectives
  - Ex: $x > 0, y = 4, a < b, z\%3 \neq 0, \ldots$
- Precision in predicate abstraction: a set π of predicates to keep track of
  - Given arbitrarily (e.g., all Boolean expressions in a program)
  - Derived automatically (with CEGAR)

Predicate Abstraction for Programs

- Analyzing infinite-state systems by constructing finite abstractions
- Idea: tracking small, finitely many facts (analogous to human analysis)
- Atomic predicate: a Boolean expression without Boolean connectives
  - Ex: $x > 0, y = 4, a < b, z\%3 \neq 0, \ldots$
- Precision in predicate abstraction: a set π of predicates to keep track of
  - Given arbitrarily (e.g., all Boolean expressions in a program)
  - Derived automatically (with CEGAR)
- Cartesian predicate abstraction: strongest conjunction of predicates from π that is implied by given formula φ, i.e., $\{t \in \pi | \varphi \Rightarrow t\}$
  - We will define a Cartesian PredicateCPA $\mathbb{P}$.

Abstract Domain for Cartesian PredicateCPA $\mathbb{P}$

Given a precision $\pi = \{p_1, \ldots, p_n\}$:

- Cartesian predicate abstraction considers subsets of π as abstract states.
  - Ex: $e_1 = \{p_1, p_2\}, e_2 = \{p_1, p_2, p_5, p_8\}, e_3 = \pi, e_4 = \emptyset, \ldots$
  - 2^n abstract states
  - Meaning of an abstract state e: all predicates in e hold.

Abstract Domain for Cartesian PredicateCPA $\mathbb{P}$

Given a precision $\pi = \{p_1, \ldots, p_n\}$:

- Cartesian predicate abstraction considers subsets of π as abstract states.
  - Ex: $e_1 = \{p_1, p_2\}, e_2 = \{p_1, p_2, p_5, p_8\}, e_3 = \pi, e_4 = \emptyset, \ldots$
  - 2^n abstract states
  - Meaning of an abstract state e: all predicates in e hold.
- Abstract domain: $D_{\mathbb{P}} = (C, (2^\pi, \supseteq, \cap, \emptyset), [\![\cdot]\!])$
  - $[\![\top]\!] = [\![\emptyset]\!] = C$
  - For $e \in 2^\pi$, $[\![e]\!] = \{c \in C \mid \forall p \in e.\, c(\sigma) \Rightarrow p\}$
  - Ex: $e = \{x > 0, x < 10, x\%2 = 0\}$,
    $[\![e]\!] = \{(l, \{x \mapsto 2\}), (l, \{x \mapsto 4\}), (l, \{x \mapsto 6\}), (l, \{x \mapsto 8\})\}$
  - In other words, $[\![e]\!]$ is the models of $\bigwedge_{p \in e} p$.

Abstract Domain for Cartesian PredicateCPA $\mathbb{P}$

Given a precision $\pi = \{p_1, \ldots, p_n\}$:

- Cartesian predicate abstraction considers subsets of π as abstract states.
  - Ex: $e_1 = \{p_1, p_2\}, e_2 = \{p_1, p_2, p_5, p_8\}, e_3 = \pi, e_4 = \emptyset, \ldots$
  - 2^n abstract states
  - Meaning of an abstract state e: all predicates in e hold.
- Abstract domain: $D_{\mathbb{P}} = (C, (2^\pi, \supseteq, \cap, \emptyset), [\![\cdot]\!])$
  - $[\![\top]\!] = [\![\emptyset]\!] = C$
  - For $e \in 2^\pi$, $[\![e]\!] = \{c \in C \mid \forall p \in e.\, c(\sigma) \Rightarrow p\}$
  - Ex: $e = \{x > 0, x < 10, x\%2 = 0\}$,
    $[\![e]\!] = \{(l, \{x \mapsto 2\}), (l, \{x \mapsto 4\}), (l, \{x \mapsto 6\}), (l, \{x \mapsto 8\})\}$
  - In other words, $[\![e]\!]$ is the models of $\bigwedge_{p \in e} p$.
- Boolean predicate abstraction: all Boolean formulas over π as abstract states

Abstract Transfer Relation for Cartesian PredicateCPA $\mathbb{P}$

- How does a predicate abstract state e evolve into its successor e' given an edge g? (e and e' are sets of predicates.)

Abstract Transfer Relation for Cartesian PredicateCPA $\mathbb{P}$

- How does a predicate abstract state e evolve into its successor e' given an edge g? (e and e' are sets of predicates.)
- We define an abstract SP operator $SP^{\pi}_{\mathbb{C}} : Ops \times 2^{\pi} \mapsto 2^{\pi}$ for Cartesian predicate abstraction. (Recall $SP : Ops \times FOL \mapsto FOL$.)

Abstract Transfer Relation for Cartesian PredicateCPA $\mathbb{P}$

- How does a predicate abstract state e evolve into its successor e' given an edge g? (e and e' are sets of predicates.)
- We define an abstract SP operator $SP^{\pi}_{\mathbb{C}} : Ops \times 2^{\pi} \mapsto 2^{\pi}$ for Cartesian predicate abstraction. (Recall $SP : Ops \times FOL \mapsto FOL$.)
- Given $e \in 2^{\pi}, op \in Ops$, the Cartesian predicate abstraction of the strongest postcondition is defined to be the abstract successor e' of e.

$$SP^{\pi}_{\mathbb{C}}(op, e) = \{t \in \pi | SP(op, \bigwedge_{p \in e} p) \Rightarrow t\} \stackrel{\text{def}}{=} e'$$

- $(e, g, e') \in \rightsquigarrow_{\mathbb{P}}$ if op is the operation of g and $e' = SP^{\pi}_{\mathbb{C}}(op, e)$.

Abstract Transfer Relation for Cartesian PredicateCPA $\mathbb{P}$

- How does a predicate abstract state e evolve into its successor e' given an edge g? (e and e' are sets of predicates.)
- We define an abstract SP operator $SP^{\pi}_{\mathbb{C}} : Ops \times 2^{\pi} \mapsto 2^{\pi}$ for Cartesian predicate abstraction. (Recall $SP : Ops \times FOL \mapsto FOL$.)
- Given $e \in 2^{\pi}, op \in Ops$, the Cartesian predicate abstraction of the strongest postcondition is defined to be the abstract successor e' of e.

$$SP^{\pi}_{\mathbb{C}}(op, e) = \{t \in \pi | SP(op, \bigwedge_{p \in e} p) \Rightarrow t\} \stackrel{\text{def}}{=} e'$$

- $(e, g, e') \in \rightsquigarrow_{\mathbb{P}}$ if op is the operation of g and $e' = SP^{\pi}_{\mathbb{C}}(op, e)$.
- When $SP(op, \bigwedge_{p \in e} p)$ is UNSAT, $e' = \pi$
  - Alternative definition: no successor (need to check the satisfiability of SP)
  - Implementation: avoid satisfiability check for SP by tracking $false$ in π

Examples

Given $\pi = \{i > 0, x = 10\}$:

- $SP^{\pi}_{\mathbb{C}}(i = 1;, \{x = 10\})$
- $SP^{\pi}_{\mathbb{C}}(i = i * 2;, \{i > 0\})$
- $SP^{\pi}_{\mathbb{C}}([i < 4], \{i > 0\})$
- $SP^{\pi}_{\mathbb{C}}([x > 10], \{x = 10, i > 0\})$

Merge and Stop Operators for Cartesian PredicateCPA $\mathbb{P}$

- $\text{merge}_{\mathbb{P}}(e_1, e_2)$: merge^{sep} (never merge), merge^{join} (intersection of predicates)
- $\text{stop}_{\mathbb{P}}(e, R)$: stop^{sep}

Cartesian Predicate Abstraction Using Composite CPA $\mathbb{L} \times \mathbb{P}$

- Running CPA algorithm using $\mathbb{L} \times \mathbb{P}$, given a precision π

Cartesian Predicate Abstraction Using Composite CPA $\mathbb{L} \times \mathbb{P}$

- Running CPA algorithm using $\mathbb{L} \times \mathbb{P}$, given a precision π

What happens if we use $\pi = \emptyset$ for $\mathbb{P}$?

Craig Interpolation

– Learning from unsatisfiability –

Craig's Interpolation Theorem

Theorem (W. Craig, 1957)

Let ϕ and ψ be two FOL formulas such that $\phi \Rightarrow \psi$.
There exists an FOL formula θ such that:

1. $\phi \Rightarrow \theta$
2. $\theta \Rightarrow \psi$
3. $Vars(\theta) \subseteq Vars(\phi) \cap Vars(\psi)$

Craig's Interpolation Theorem

Theorem (W. Craig, 1957)

Let ϕ and ψ be two FOL formulas such that $\phi \Rightarrow \psi$.
There exists an FOL formula θ such that:

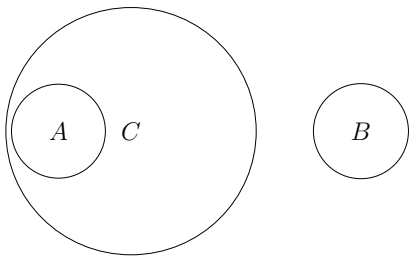
1. $\phi \Rightarrow \theta$
2. $\theta \Rightarrow \psi$
3. $Vars(\theta) \subseteq Vars(\phi) \cap Vars(\psi)$

- Intuition: θ is "in between" ϕ and ψ.
- First used in model checking by K. L. McMillan in 2003
  - Widely adopted in formal methods since then for various purposes, including CEGAR, overapproximation, trace abstraction, etc.

An Equivalent Form of Craig's Interpolation Theorem

Often used in model checking:

- If $A \wedge B$ is UNSAT, there exists an interpolant C such that:
  1. $A \Rightarrow C$,
  2. $C \wedge B$ is UNSAT, and
  3. $Vars(C) \subseteq Vars(A) \cap Vars(B)$.
- Denoted as $C = itp(A, B)$ (for convenience; in general, C is not unique!)



Example: Craig Interpolants

Consider the formulas

$$A = x_0 > 10 \wedge y_1 = 1$$

$$B = x_0 \leq 0 \wedge z_0 = 0$$

Which of the following formulas are Craig interpolants of A and B?

- A
- $x_0 > 0 \wedge (z_0 > 0 \vee z_0 \leq 0)$
- $\bot$
- $x_0 \neq 0$

Interpolation Sequence

Let $A_1, \ldots, A_n$ be a sequence of formulas such that $A_1 \wedge \cdots \wedge A_n$ is UNSAT.
There exists a sequence $\tau_0, \ldots, \tau_n$ of inductive interpolants,
called an *interpolation sequence*, such that:

1. $\tau_0 = \top$ (logical true) and $\tau_n = \bot$ (logical false),
2. Inductiveness $\tau_{i-1} \wedge A_i \Rightarrow \tau_i$ for $1 \leq i \leq n$, and
3. $Vars(\tau_i) \subseteq Vars(\bigwedge_{j=1}^{i} A_j) \cap Vars(\bigwedge_{j=i+1}^{n} A_j)$ for $1 \leq i < n$.

Interpolation Sequence

Let $A_1, \ldots, A_n$ be a sequence of formulas such that $A_1 \wedge \cdots \wedge A_n$ is UNSAT.
There exists a sequence $\tau_0, \ldots, \tau_n$ of inductive interpolants,
called an *interpolation sequence*, such that:

1. $\tau_0 = \top$ (logical true) and $\tau_n = \bot$ (logical false),
2. Inductiveness $\tau_{i-1} \wedge A_i \Rightarrow \tau_i$ for $1 \leq i \leq n$, and
3. $Vars(\tau_i) \subseteq Vars(\bigwedge_{j=1}^{i} A_j) \cap Vars(\bigwedge_{j=i+1}^{n} A_j)$ for $1 \leq i < n$.

Why does an interpolation sequence exist? (proof by induction)

Example: Interpolation Sequence

Consider the following program path:

$$l_0 \xrightarrow{x=\texttt{nondet}();} l_1 \xrightarrow{[x<10]} l_2 \xrightarrow{x=x+1;} l_3 \xrightarrow{[x==20]} l_{ERR}$$

The strongest postcondition at l_{ERR} using the SSA form is:

$$x_1 = r \wedge x_1 < 10 \wedge x_2 = x_1 + 1 \wedge x_2 = 20$$

(r is a free variable denoting the returned value of function `nondet()`.)

What is a valid interpolation sequence for this sequence of formulas?

Counterexample-Guided Abstraction Refinement (CEGAR)

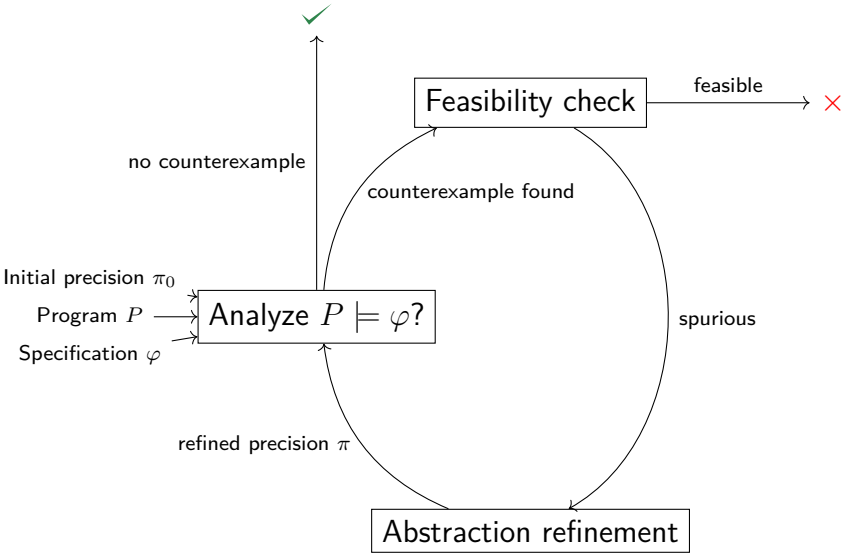
CEGAR Principle

Find a good trade-off between precision and efficiency automatically

- Start with a coarse abstraction
- Refine if it fails to (dis)prove specification, using the spurious counterexample
- Restart analysis with a refined, more precise abstraction
  - Guarantee progress: new precision excludes previous spurious counterexamples

CEGAR is a general principle, not a specific algorithm itself.

CEGAR Overview



Combining CEGAR and Configurable Program Analysis

- Analyze $P \models \varphi$
  - Start with a coarse initial precision, e.g., an empty set of predicates.
  - Build an ARG using the CPA algorithm under the current precision.
  - If the ARG has no abstract counterexamples, the specification is satisfied.

Combining CEGAR and Configurable Program Analysis

- Analyze $P \models \varphi$
  - Start with a coarse initial precision, e.g., an empty set of predicates.
  - Build an ARG using the CPA algorithm under the current precision.
  - If the ARG has no abstract counterexamples, the specification is satisfied.
- Feasibility check
  - If an abstract counterexample exists, compute its strongest postcondition.
  - If SP is SAT, the specification is violated.
  - If SP is UNSAT, the abstract counterexample is spurious.

Combining CEGAR and Configurable Program Analysis

- Analyze $P \models \varphi$
  - Start with a coarse initial precision, e.g., an empty set of predicates.
  - Build an ARG using the CPA algorithm under the current precision.
  - If the ARG has no abstract counterexamples, the specification is satisfied.
- Feasibility check
  - If an abstract counterexample exists, compute its strongest postcondition.
  - If SP is SAT, the specification is violated.
  - If SP is UNSAT, the abstract counterexample is spurious.
- Abstraction refinement
  - Given an UNSAT SP, compute an interpolation sequence for refinement.
  - Restart the ARG construction with the refined precision.

Precision Refinement for Predicate Abstraction

Location-Aware Precisions

- So far, predicates are tracked uniformly at every program location.
- More fine-grained alternative: $\pi : L \mapsto \Pi$ (Π: the set of precisions)
- An empty precision π_0 maps each location l to $\emptyset$ (initial precision).

Precision Refinement via Interpolation Sequence

Let $l_0 \xrightarrow{op_1} l_1 \cdots \xrightarrow{op_n} l_{ERR}$ be a spurious counterexample.

1. Compute the strongest postcondition $A_1 \wedge \ldots \wedge A_n$ of the counterexample.
2. Construct an interpolation sequence $\langle \tau_0, \ldots, \tau_n \rangle$ from SP.
   - In practice, SMT solvers compute interpolants from UNSAT proofs.
3. Extract atomic predicates in each τ_i into a sequence $\langle \rho_0, \ldots, \rho_n \rangle$ of sets.
   - SSA indices need to be removed!
4. Refine the precision as $\pi'(l) = \pi(l) \cup \bigcup\{\rho_i \mid l_i = l\}$.

Example: Deriving New Predicates

Consider the following program path:

$$l_0 \xrightarrow{x=\texttt{nondet}();} l_1 \xrightarrow{[x<10]} l_2 \xrightarrow{x=x+1;} l_3 \xrightarrow{[x==20]} l_{ERR}$$

The strongest postcondition at l_{ERR} using the SSA form is:

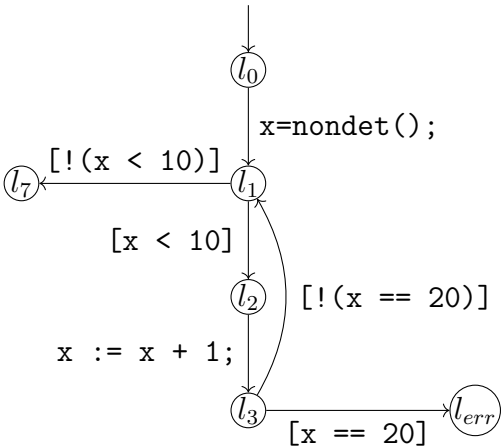
$$x_1 = r \wedge x_1 < 10 \wedge x_2 = x_1 + 1 \wedge x_2 = 20$$

A valid interpolant sequence is $\langle \top, \top, x_1 < 10, x_2 \neq 20, \bot \rangle$.

The extracted sets of predicates are $\langle \{\top\}, \{\top\}, \{x < 10\}, \{x \neq 20\}, \{\bot\} \rangle$.
The previous precision π is refined with the obtained predicates at l_2, l_3, and l_{ERR}.

Cartesian Predicate Abstraction with Precision Adjustment

Start with the empty precision π_0.



Conclusion: What Have We Learned Today?

- Program representation: control flow automata (CFA)
- Dealing with infinitely many program states: abstraction
  - Abstract state: an element in a lattice
- Configurable program analysis (CPA)
  - A flexible framework to represent various program analyses
- Cartesian predicate abstraction
  - Deducing program correctness from small facts
- Automatically learning facts to track: Craig interpolation and CEGAR
- CPAchecker: a state-of-the-art software model checker for C programs
  - https://cpachecker.sosy-lab.org/