Talk Outline

- Motivation (2 mn) ............................................. 6
- Applications of abstract interpretation (2 mn) ............. 9
- Elements of abstract interpretation (12 mn) ................ 12
- Examples:
  - Semantics (8 mn) ........................................... 30
  - Typing (9 mn) ............................................... 40
  - Termination Proofs (8 mn) ............................... 55
  - Hardware verification (4 mn) ............................ 66
- Static Analysis of Avionic Safety-Critical Software (5 mn) 71
- Projects (4 mn) ............................................... 79
- Conclusion (1 mn) ............................................. 87

Abstraction and Approximation

Two fundamental concepts in computer science (and engineering):

- Abstraction: to reason on complex systems;
- Approximation: to make undecidable reasoning computationally feasible.

Formalized by Abstract Interpretation.

References


Abstract Interpretation

- Born to formalize static program analysis;
- Viewed today as a general formalism to reason about semantics of computer systems at different levels of abstraction;
- Successfully applied to automatic analysis of complex computer systems.
A Few Applications of Abstract Interpretation (Cont’d)

- (Abstract) Model Checking [POPL ’00]
- Program Transformation (partial evaluation, monitoring, ...) [POPL ’02]
- Software Watermarking [POPL ’04]
- Bisimulations [RT-ESOP ’04]

All these techniques involve sound approximations that can be formalized by abstract interpretation

---

A Few Applications of Abstract Interpretation

- Static Program Analysis [POPL ’77], [POPL ’78], [POPL ’79] including Dataflow Analysis [POPL ’79], [POPL ’00], Set-based Analysis [FPCA ’95], Predicate Abstraction [Manna’s festschrift ’03], ...

- Syntax Analysis [TCS 290(1) 2002]
- Hierarchies of Semantics (including Proofs) [POPL ’92], [TCS 277(1–2) 2002]
- Typing & Type Inference [POPL ’97]
Program Semantics

Language Semantics

- A language $\mathcal{L}$ is a set of program texts $P \in \mathcal{L}$
- A semantic domain $\mathcal{D}$ is a set of program semantics
- A program semantics is a mathematical object formally describing program executions (i.e. the effect of running a program on a computer)
- A language semantics $S$ maps programs $P \in \mathcal{L}$ to their semantics $S[P] \in \mathcal{D}$

Example: Trace Semantics

Formal Definition of the Language Semantics

- A language semantics $S \in \mathcal{L} \mapsto \mathcal{D}$ is formally defined
  - denotationally: by induction on the syntax of programs $P \in \mathcal{L}$
  - compositionally: by composing elementary mathematical objects and structures (numbers, pairs, tuples, relations, orders, functions, functionals, fixpoints, etc)
Least Fixpoint Trace Semantics

Traces = \{ \bullet | \bullet \text{ is a final state} \}

\begin{align*}
\cup \{ & \bullet \rightarrow \bullet \rightarrow \cdots \rightarrow \bullet | \bullet \rightarrow \bullet \text{ is a transition step} & & \bullet \rightarrow \cdots \rightarrow \bullet \in \text{Traces}^+ \} \\
\cup \{ & \bullet \rightarrow \bullet \rightarrow \cdots \rightarrow \cdots | \bullet \rightarrow \bullet \text{ is a transition step} & & \bullet \rightarrow \cdots \rightarrow \cdots \in \text{Traces}^\infty \}
\end{align*}

- In general, the equation has multiple solutions;
- Choose the least one for the computational ordering:
  “more finite traces & less infinite traces”.

Iterative Fixpoint Calculation of the Trace Semantics

\begin{align*}
\text{Iterates} & \quad \text{Finite traces} & \quad \text{Infinite traces} \\
F^0 & \{ \} & \{ \bullet \rightarrow \bullet \rightarrow \cdots \rightarrow \bullet \} \\
F^1 & \{ \bullet \rightarrow \bullet \rightarrow \cdots \rightarrow \bullet \} & \{ \bullet \rightarrow \bullet \rightarrow \cdots \rightarrow \bullet \} \\
F^2 & \{ \bullet \rightarrow \bullet \rightarrow \cdots \rightarrow \bullet \} & \{ \bullet \rightarrow \bullet \rightarrow \cdots \rightarrow \bullet \} \\
F^3 & \{ \bullet \rightarrow \bullet \rightarrow \cdots \rightarrow \bullet \} & \{ \bullet \rightarrow \bullet \rightarrow \cdots \rightarrow \bullet \} \\
\vdots & \vdots & \vdots \\
F^n & \{ \bullet \rightarrow \bullet \rightarrow \cdots \rightarrow \bullet \} & \{ \bullet \rightarrow \bullet \rightarrow \cdots \rightarrow \bullet \} \\
\vdots & \vdots & \vdots \\
F^\infty & \{ \bullet \rightarrow \bullet \rightarrow \cdots \rightarrow \bullet \} & \{ \bullet \rightarrow \bullet \rightarrow \cdots \rightarrow \bullet \}
\end{align*}

Program Properties
Program Properties & Static Analysis

– A program property $P \in \wp(\mathcal{D})$ is a set of possible semantics for that program (hence a subset of the semantic domain $\mathcal{D}$)
– A property $P \in \wp(\mathcal{D})$ is stronger (or more precise) than a property $Q \in \wp(\mathcal{D})$ iff $P \subseteq Q$ (i.e. $P$ implies $Q$, $P \Rightarrow Q$)
– The strongest program property\(^1\) is $\{S[\!\!\!P]\!\!\!\} \in \wp(\mathcal{D})$
– A static analysis effectively approximates the strongest property of programs

\(^1\) also called the collecting semantics

Example Program Property

– Correct implementations: print 0, [print 1|loop], ...
– Excludes [print 0|print 1]
– Note for specialists: neither a safety nor a liveness property.

Abstraction of Program Properties

Abstraction

– Replace actual/concrete properties $P \in \wp(\mathcal{D})$ by an approximate abstract properties $\alpha(P)$
– Examples:
  - engineering:
    $\alpha$(property of an object) = property of a model of the object
  - partial correctness in computer science:
    $\alpha$(program property) = restriction of the property to finite executions
Commonly Required Properties of the Abstraction

- [In this talk,] we consider sound overapproximations:
  \[ \mathcal{P} \subseteq \alpha(\mathcal{P}) \]
  - If the abstract property \( \alpha(\mathcal{P}) \) does hold then so does the concrete properties \( \mathcal{P} \)
  - If the abstract property \( \alpha(\mathcal{P}) \) does not hold then the concrete properties \( \mathcal{P} \) may hold or not!\(^2\)

- All information is lost at once:
  \[ \alpha(\alpha(\mathcal{P})) = \alpha(\mathcal{P}) \]

- The abstraction of more precise properties is more precise:
  \[ \text{if } \mathcal{P} \subseteq \mathcal{Q} \text{ then } \alpha(\mathcal{P}) \subseteq \alpha(\mathcal{Q}) \]

\(^2\) In this case we speak of “false alarm”.

Abstraction 1: Functions

- Let \( \langle \mathcal{P}(\mathcal{D}), \subseteq \rangle \xleftarrow{\gamma} \langle \mathcal{D}^\# , \subseteq \rangle \)

- How to abstract a property transformer \( F \in \mathcal{P}(\mathcal{D}) \mapsto \mathcal{P}(\mathcal{D}) \)?

- The most precise sound overapproximation is
  \[ F^\# \in \mathcal{D}^\# \mapsto \mathcal{D}^\# \]
  \[ F^\# = \alpha \circ F \circ \gamma \]

- This is a Galois connection
  \[ \langle \mathcal{P}(\mathcal{D}) \mapsto \mathcal{P}(\mathcal{D}) , \subseteq \rangle \xleftarrow{\gamma} \langle \mathcal{D}^\# \mapsto \mathcal{D}^\# , \subseteq \rangle \]

Abstraction 2: Fixpoints

- Let \( \langle \mathcal{P}(\mathcal{D}), \subseteq \rangle \xleftarrow{\gamma} \langle \mathcal{D}^\# , \subseteq \rangle \)

- How to abstract a fixpoint property \( \text{lfp} \subseteq F \) where \( F \in \mathcal{P}(\mathcal{D}) \mapsto \mathcal{P}(\mathcal{D}) \)?

- Approximate Sound Abstraction:
  \[ \text{lfp} \subseteq F \subseteq \gamma(\text{lfp} \subseteq \alpha \circ F \circ \gamma) \]

- Complete Abstraction: if \( \alpha \circ F = F^\# \circ \alpha \) then
  \[ F^\# = \alpha \circ F \circ \gamma , \text{ and} \]
  \[ \alpha(\text{lfp} \subseteq F) = \text{lfp} \subseteq F^\# \]
Abstract Interpretation-Based Static Analysis

- an inductive compositional language semantics \( S \in \mathcal{L} \mapsto D \)
- program concrete properties \( \varphi(D) \)
- an abstract domain \( \langle \varphi(D), \subseteq \rangle \xrightarrow{\gamma} \langle D \#, \subseteq \rangle \) designed inductively and compositionally to approximate the property to be analyzed
- the A.I. Theory is used to systematically derive the sound abstract semantics \( S\#[P] \supseteq \alpha(\{S[P]\}) \)
- the static analysis algorithm is the computation of the abstract semantics and is correct by construction

Example 1: Trace Semantics Abstraction

Objective

- A unifying formalization of the classical semantics as abstract interpretations of the trace semantics
- (\ldots and of a few new ones)

Semantics Abstractions

1 — Relational Semantics Abstractions

\[ \langle \varphi(\Sigma^+ \cup \Sigma^\omega), \subseteq \rangle \xleftarrow{\gamma} \langle \varphi(\Sigma \times (\Sigma \cup \{\bot\})), \subseteq \rangle \]

Finite and infinite traces

Relation between initial and final states or \( \bot \)

Reference


3 \( \bot \) in Dana Scott’s traditional notation for non-termination.
1 — Relational Semantics Abstractions (Cont’d)

\( \alpha^h(X) = \{(s, s') \mid s \sigma s' \in X \cap \Sigma^+\} \)
\( \cup \{\langle s, \bot \rangle \mid s \sigma \in X \cap \Sigma^\omega\} \)
trace to natural relational semantics

\( \alpha^h(X) = \{(s, s') \mid s \sigma s' \in X \cap \Sigma^+\} \)
trace to angelic relational semantics

\( \alpha^h(X) = \{(s, s') \mid s \sigma s' \in X \cap \Sigma^+\} \)
\( \cup \{\langle s, s' \rangle \mid s \sigma \in X \cap \Sigma^\omega \land s' \in \Sigma \cup \{\bot\}\} \)
trace to demoniac relational semantics

\( \langle \Sigma \mapsto \wp(\Sigma \cup \{\bot\}), \subseteq \rangle \xrightarrow{\gamma^\sigma} \langle \wp(\Sigma) \mapsto \wp(\Sigma \cup \{\bot\}), \subseteq \rangle \)
Map of initial states
Map of sets of initial states or \( \bot \)

\( \alpha^\pi(\phi) = \lambda P.\{s' \in \Sigma \cup \{\bot\} \mid \exists s \in P : s' \in \phi(s)\} \)
denotational to predicate transformer semantics

2 — Denotational Semantics Abstractions

\( \langle \wp(\Sigma \times (\Sigma \cup \{\bot\})), \subseteq \rangle \xrightarrow{\gamma^\sigma} \langle \Sigma \mapsto \wp(\Sigma \cup \{\bot\}), \subseteq \rangle \)
Relation between initial and final states or \( \bot \)
Map of initial states to sets of final states or \( \bot \)

\( \alpha^\sigma(\Sigma) = \lambda s.\{s' \in \Sigma \cup \{\bot\} \mid (s, s') \in X\} \)
relational to denotational semantics

3 — Predicate Transformer Abstractions

\( \langle \wp(\Sigma) \mapsto \wp(\Sigma \cup \{\bot\}), \subseteq \rangle \xrightarrow{\gamma^\sigma} \langle \wp(\Sigma) \mapsto \wp(\Sigma \cup \{\bot\}), \subseteq \rangle \)

\( \alpha^\pi(\phi) = \lambda P.\neg(\phi(\neg P)) \)
conjugate

\( \alpha^\cap(\phi) = \lambda Q.\{s \in \Sigma \mid \phi(\{s\}) \cap Q \neq \emptyset\} \)
\( \cup \)-inversion

\( \alpha^{\cap}(\phi) = \lambda Q.\{s \in \Sigma \mid \phi(\neg\{s\}) \cup Q = \Sigma \cup \{\bot\}\} \)
\( \cap \)-inversion

4 — Predicate Transformer Abstractions (Cont’d)

\( \langle \wp(\Sigma) \mapsto \wp(\Sigma \cup \{\bot\}), \subseteq \rangle \xrightarrow{\gamma^\sigma} \langle \wp(\Sigma) \mapsto \wp(\Sigma \cup \{\bot\}), \subseteq \rangle \)

\( \alpha^\pi(\phi) = \lambda P.\neg(\phi(\neg P)) \)
conjugate

\( \alpha^\cap(\phi) = \lambda Q.\{s \in \Sigma \mid \phi(\{s\}) \cap Q \neq \emptyset\} \)
\( \cup \)-inversion

\( \alpha^{\cap}(\phi) = \lambda Q.\{s \in \Sigma \mid \phi(\neg\{s\}) \cup Q = \Sigma \cup \{\bot\}\} \)
\( \cap \)-inversion

---

4 States that must reach \( P \) by state transformer \( \phi \) or block

5 Non-blocking states that may reach \( Q \) by state transformer \( \phi \)

6 Non-blocking states that must reach \( Q \) by state transformer \( \phi \)
5 — Hoare Logic Abstractions

\[ \langle \wp(\Sigma) \sqcup \wp(\Sigma \cup \{\bot\}), \subseteq \rangle \overset{\alpha^H}{\longrightarrow} \langle \wp(\Sigma) \otimes \wp(\Sigma \cup \{\bot\}), \subseteq \rangle \]

\begin{itemize}
  \item Map of sets of initial states to sets of final states or \( \bot \)
  \item Set of all Hoare triples (generalized to non-termination)
\end{itemize}

\[ \alpha^H(\Phi) = \{ \langle P, Q \rangle \mid \Phi(P) \subseteq Q \} \]

Predicate transformer to Hoare logic semantics

---

6 — Safety Abstraction

- Disjunctive abstraction: \( \alpha_u(P) \overset{\Delta}{=} \bigcup P \)
  \[ \langle \wp(\Sigma^+ \cup \Sigma^\omega), \subseteq \rangle \overset{\gamma^H_u}{\longrightarrow} \langle \wp(\Sigma^+ \cup \Sigma^\omega), \subseteq \rangle \]

- Prefix abstraction (time invariance):
  \[ \alpha_p(P) \overset{\Delta}{=} \{ \sigma \in \Sigma^+ \mid \exists \sigma' \in \Sigma^+ \cup \Sigma^\omega : \sigma' \in P \} \]
  \[ \langle \wp(\Sigma^+ \cup \Sigma^\omega), \subseteq \rangle \overset{\gamma^p}{\longrightarrow} \langle \wp(\Sigma^+ \cup \Sigma^\omega), \subseteq \rangle \]

- Limit abstraction (infinite behaviors are not observable):
  \[ \alpha_{\ell}(P) \overset{\Delta}{=} \{ \sigma \in \Sigma^\omega \mid \alpha_p(\{\sigma\}) \subseteq P \} \]
  \[ \langle \wp(\Sigma^+ \cup \Sigma^\omega), \subseteq \rangle \overset{\gamma_{\ell}}{\longrightarrow} \langle \wp(\Sigma^+ \cup \Sigma^\omega), \subseteq \rangle \]

- Safety abstraction (can be monitored at runtime):
  \[ \langle \wp(\Sigma^+ \cup \Sigma^\omega), \subseteq \rangle \overset{\gamma_{\xi} \cdot \gamma^p \cdot \alpha_u}{\longrightarrow} \langle \wp(\Sigma^+ \cup \Sigma^\omega), \subseteq \rangle \]

---

Lattice of Semantics

**Hoare logics**

- weakest precondition semantics
- denotational semantics
- relational semantics
- trace semantics

\( \tau^H \)

\( \tau^P \)

\( \tau^D \)

\( \tau^T \)

\( \tau^M \)

\( \tau^\omega \)

\( \tau^\infty \)

\( \tau^\ast \)

\( \tau^\dagger \)

\( \tau^\natural \)

**Equivalence**

- abstractions
- restrictions

**Restriction**

- angelic
- natural
- demoniac
- determinist
- infinite

---

Example 2: Typing

---

Reference

Objective

– Show that static typing and type inference are abstract interpretations of a semantics with runtime type checking
– (… and consider nontermination in type soundness)

Semantic Domains

\[ \begin{align*}
\Omega & \quad \text{wrong/runtime error value} \\
\bot & \quad \text{non-termination} \\
W & \triangleq \{\Omega\} \\
z \in \mathbb{Z} & \quad \text{integers} \\
u, f, \varphi \in U & \cong W \oplus \mathbb{Z} \oplus [U \mapsto U] \quad \text{values} \\
R \in R & \triangleq X \mapsto U \\
\phi \in S & \triangleq R \mapsto U \\
\end{align*} \]

\[ \text{environments} \]

\[ \text{semantic domain} \]

\[ ^8 [U \mapsto U]: \text{continuous, } \bot\text{-strict, } \bot\text{-strict functions from values } U \text{ to values } U. \]

Syntax of the Eager Lambda Calculus

\[ \begin{align*}
x, f, \ldots & \in X & \quad \text{variables} \\
e & \in E & \quad \text{expressions} \\
e ::= x & \quad \text{variable} \\
| \lambda x \cdot e & \quad \text{abstraction} \\
| e_1(e_2) & \quad \text{application} \\
| \mu f \cdot \lambda x \cdot e & \quad \text{recursion} \\
| 1 & \quad \text{one} \\
| e_1 - e_2 & \quad \text{difference} \\
| (e_1 ? e_2 : e_3) & \quad \text{conditional} \\
\end{align*} \]

Denotational Semantics with Run-Time Type Checking

\[ \begin{align*}
S[1]R & \triangleq 1 \\
S[e_1 - e_2]R & \triangleq (S[e_1]R = \bot \lor S[e_2]R = \bot) \lor (S[e_1]R = z_1 \land S[e_2]R = z_2 \iff z_1 \neq 0 ? S[e_2]R \\
S[\mu f \cdot \lambda x \cdot e]R & \triangleq (S[e_1]R = \bot \lor S[e_1]R = z \neq 0 ? S[e_2]R \\
S[(e_1 ? e_2 : e_3)]R & \triangleq (S[e_1]R = \bot \lor S[e_1]R = z \neq 0 ? S[e_2]R \\
& \quad \lor S[e_1]R = 0 ? S[e_3]R \\
& \quad \lor \Omega) \\
\end{align*} \]
Abstracting with Church/Curry Monotypes

– Simple types are monomorphic:
\[ m \in \mathbb{M}^c, \quad m ::= \text{int} | m_1 \to m_2 \quad \text{monotype} \]

– A type environment associates a type to free program variables:
\[ H \in \mathbb{H}^c \triangleq X \mapsto \mathbb{M}^c \quad \text{type environment} \]

Standard Denotational and Collecting Semantics

– The denotational semantics is:
\[ S[\bullet] \in \mathbb{E} \mapsto S \]

– A concrete property \( P \) of a program is a set of possible program behaviors:
\[ P \in \wp(S) \]

– The standard collecting semantics is the strongest concrete property:
\[ C[\bullet] \in \mathbb{E} \mapsto \wp(S) \quad C[e] \triangleq \{ S[e] \} \]

Abstracting with Church/Curry Monotypes (Cont’d)

– A typing \( \langle H, m \rangle \) specifies a possible result type \( m \) in a given type environment \( H \) assigning types to free variables:
\[ \theta \in \Gamma^c \triangleq \mathbb{H}^c \times \mathbb{M}^c \quad \text{typing} \]

– An abstract property or program type is a set of typings:
\[ T \in \mathbb{T}^c \triangleq \wp(\Gamma^c) \quad \text{program type} \]
Concretization Function

The meaning of types is a program property, as defined by the concretization function $\gamma^c$: 9

- Monotypes $\gamma^c_i \in \mathcal{M}^c \mapsto \wp(\mathcal{U})$:

$$
\gamma^c_i(\text{int}) \triangleq \mathbb{Z} \cup \{\bot\}
$$

$$
\gamma^c_i(m_1 \to m_2) \triangleq \{\varphi \in \mathcal{U} \mapsto \mathcal{U} \mid \forall u \in \gamma^c_i(m_1) : \varphi(u) \in \gamma^c_i(m_2)\} \cup \{\bot\}
$$

- type environment $\gamma^c_e \in \mathcal{H}^c \mapsto \wp(\mathbb{R})$:

$$
\gamma^c_e(H) \triangleq \{R \in \mathbb{R} \mid \forall x \in X : R(x) \in \gamma^c_i(H(x))\}
$$

- typing $\gamma^c_e \in \mathcal{I}^c \mapsto \wp(\mathcal{S})$:

$$
\gamma^c_e((H, m)) \triangleq \{\phi \in \mathcal{S} \mid \forall R \in \gamma^c_e(H) : \phi(R) \in \gamma^c_i(m)\}
$$

- program type $\gamma^c \in \mathcal{T}^c \mapsto \wp(\mathcal{S})$:

$$
\gamma^c(T) \triangleq \bigcap_{\theta \in T} \gamma^c_i(\theta)
$$

$$
\gamma^c(0) \triangleq \mathcal{S}
$$

9 For short up/down lifting/injection are omitted.

Program Types

- Galois connection:

$$
\langle \wp(\mathcal{S}), \subseteq, \wp(\mathcal{S}), \cup, \cap \rangle \xrightarrow{\gamma^c} \langle \wp(\mathcal{S}), \supseteq, \wp(\mathcal{S}), \emptyset, \cap \rangle
$$

- Types $T[e]$ of an expression $e$:

$$
T[e] \subseteq \alpha^c(C[e]) = \alpha^c(\{S[e]\})
$$

Typable Programs Cannot Go Wrong

$$
\Omega \in \gamma^c(T[e]) \iff T[e] = 0
$$

Church/Curry Monotype Abstract Semantics

$$
T[x] \triangleq \{(H, H(x)) \mid H \in \mathbb{H}^c\} \quad \text{(VAR)}
$$

$$
T[\lambda x \cdot e] \triangleq \{(H, m_1 \to m_2) \mid \langle H[x \leftarrow m_1], m_2 \rangle \in T[e]\} \quad \text{(ABS)}
$$

$$
T[e_1(e_2)] \triangleq \{(H, m_2) \mid \langle H, m_1 \to m_2 \rangle \in T[e_1] \land \langle H, m_1 \rangle \in T[e_2]\} \quad \text{(APP)}
$$
\[
\begin{align*}
T[\mu f \cdot \lambda x \cdot e] & \triangleq \{ \langle H, m \rangle \mid (H[m \leftarrow f], m) \in T[\lambda x \cdot e] \} \quad \text{(REC)} \\
T[1] & \triangleq \{ \langle H, \text{int} \rangle \mid H \in \mathbb{H}^c \} \quad \text{(CST')} \\
T[e_1 - e_2] & \triangleq \{ \langle H, \text{int} \rangle \mid (H, \text{int}) \in T[e_1] \cap T[e_2] \} \quad \text{(DIF)} \\
T[(e_1 ? e_2 : e_3)] & \triangleq \{ \langle H, m \rangle \mid (H, \text{int}) \in T[e_1] \land (H, m) \in T[e_2] \cap T[e_3] \} \quad \text{(CND)}
\end{align*}
\]

Example 3: Termination Proofs

The Herbrand Abstraction to Get Hindley’s Type Inference Algorithm

\[\langle \varphi(\text{ground}(T)), \subseteq, \emptyset, \text{ground}(T), \sqcup, \sqcap, \rangle \]

where:
- \(T\): set of terms with variables \('a\, ,\ldots\)
- \(\text{l cg}\): least common generalization
- \(\text{ground}\): set of ground instances
- \(\subseteq\): instance preordering
- \(\text{g ci}\): greatest common instance

Objective

- Show that program termination proofs are abstract interpretations of a relational semantics
- (\ldots and automatize such proofs)

References

Termination Proof

- **Problem**: prove that all executions of a program loop terminate

- **Principle**\(^\text{10}\): Exhibit a ranking function of the program variables in a well-founded set that strictly decreases at each program step for reachable states.

\(^{10}\) Robert Floyd, 1967, note the similarity with Lyapunov, 1890, “an invariant set of a differential equation is stable in the sense that it attracts all solutions if one can find a function that is bounded from below and decreases along all solutions outside the invariant set”.

Termination Proof by Static Analysis

1. Perform an iterated forward/backward relational static analysis of the loop to determine a necessary termination precondition

2. Assuming the termination precondition, perform an forward relational static analysis of the loop to determine the loop invariant (overapproximating reachable states)

3. Assuming the loop invariant, perform an forward relational static analysis of the loop body to determine the loop abstract operational semantics

4. Assuming the loop semantics, use an abstraction of Floyd’s ranking function method to prove termination of the loop

Example (Arithmetic Mean)

\{x=y+2k,x\geq y\} \leftarrow \text{necessary termination precondition}

\text{while } (x \not= y) \text{ do}

\{x=y+2k,x\geq y+2\} \leftarrow \text{loop invariant}

\{(x=x0)\&(y=y0)\&(k=k0)\}

\begin{align*}
& k := k - 1; \\
& x := x - 1; \\
& y := y + 1
\end{align*}

\{x+2=y+2k,0,y=y0+1, \leftarrow \text{loop abstract}

x+1=x0,x=y+2k,x\geq y\}

\text{operational semantics}

\text{od}

\{k=0\}

\begin{align*}
\forall x0,x : \bigwedge_{i=1}^{N} \sigma_i(k0,x0,y0,k,x,y) \geq 0
\Rightarrow r(x0) \geq 0
\end{align*}

Find an \(\mathbb{R}/\mathbb{Q}/\mathbb{Z}\)-valued unknown rank function \(r\) and \(\eta > 0\) such that:

- The rank is nonnegative:

\[\forall x0,x : \bigwedge_{i=1}^{N} \sigma_i(x0,x) \geq 0 \Rightarrow r(x0) \geq 0\]

- The rank is strictly decreasing:

\[\forall x0,x : \bigwedge_{i=1}^{N} \sigma_i(x0,x) \geq 0 \Rightarrow r(x0) - r(x) - \eta \geq 0\]
Abstraction

1. Eliminate $\bigwedge$ and $\Rightarrow$ by Lagrangian relaxation\(^\mathrm{11}\)

2. Choose a parametric abstraction $r_a$ for the ranking function $r$ in term of unknown parameters $a$ e.g. $r_a(x) = a_x^T$ (linear), $r_a(x) = a(x 1)^T$ (affine) or $r_a(x) = (x 1).a.(x 1)^T$ (quadratic)

3. Eliminate the universal quantification $\forall$ using linear matrix inequalities (LMIs) in favor of positive semidefiniteness i.e. $M(\lambda) > 0 = \forall X \in \mathbb{R}^N : X^T M(\lambda) X \geq 0$ where $M(\lambda) = M_0 + \sum_{i=1}^{N} \lambda_i M_i$

\(^{11}\) [\(\forall x : (\lambda, f(x) \geq 0) \Rightarrow (\lambda(x) \geq 0) \iff \exists \lambda, \geq 0 : \forall x : g(x) - \sum \lambda f(x) \geq 0\), sound by Lagrange, complete by Farkas in linear case and Yakubovich's S-procedure with one quadratic constraint]

---

Abstract Floyd's Ranking Function Method

Find $\mathbb{R}/\mathbb{Q}/\mathbb{Z}$-valued unknown parameters $a$, such that:

- Nonnegative: $\exists \lambda \in [1, N] \mapsto \mathbb{R}^+ :$

  $\forall x_0, x : r_a(x_0) - \sum_{i=1}^{N} \lambda_i (x_0 x 1) M_i (x_0 x 1)^T \geq 0$

- Strictly decreasing: $\exists \eta > 0 : \exists \lambda' \in [1, N] \mapsto \mathbb{R}^+ :$

  $\forall x_0, x : (r_a(x_0) - r_a(x) - \eta) - \sum_{i=1}^{N} \lambda'_i (x_0 x 1) M_i (x_0 x 1)^T \geq 0$

Finally, solve these convex constraints by semidefinite programming to get the parameters $a$ (and $\lambda$)

---

Example (Arithmetic Mean)

\{x=y+2k, x>y\} $\leftarrow$ necessary termination precondition

while $(x <> y)$ do

  $k := k - 1$;
  $x := x - 1$;
  $y := y + 1$

od

\[ r(x, y, k) = +4.k -2 \]

Generalization: non-convex polynomial constraints can be approximated in semidefinite programming form as SOS.

---

Termination of a Fair Parallel Program

\[ \text{interleaving} \quad \text{+ scheduler} \]

\[[ \text{while } [(x>0) \lor (y>0) \text{ do } x := x - 1 \text{ od ]} ] \]

\[[ \text{while } [(x>0) \lor (y>0) \text{ do } y := y - 1 \text{ od ]} ] \]

\{m>=1\}

termination precondition determined by iterated forward/backward polyhedral analysis

\[ \begin{array}{l}
  t := 7; \\
  \text{assume} (0 <= t \land t <= 1); \\
  s := 7; \\
  \text{assume} ((1 <= s) \land (s <= m)); \\
  \text{while } ((x > 0) \lor (y > 0)) \text{ do } \\
  \text{if } (t = 1) \text{ then } x := x - 1; \\
  \text{else } y := y - 1; \\
  \text{fi}; \\
  \text{if } (s = 0) \text{ then } \\
  \text{else } \\
  \text{skip}; \\
  \text{fi}; \\
  \text{if } (s = 0) \text{ then } \\
  \text{else } \\
  \text{skip}; \\
  \text{fi}; \\
  s := s - 1; \\
  \text{od}; \\
\end{array} \]

\[ \text{penbmi: } r(x, y, m, s, t) = +1.000468e+00 \cdot x +1.000611e+00 \cdot y +2.855769e-02 \cdot m -3.929197e-07 \cdot s +6.588027e-06 \cdot t +9.998392e+03 \]
Example of Challenge in Embedded Software Verification

Given a control/command program, prove that requests have responses in bounded time:

– solved for synchronous programs by abstract interpretation-based worst-case execution time (WCET) static analysis; does scale up!12!
– Opened challenge to scale up for asynchronous control/command programs with real-time scheduling

---

Example 4: Hardware Verification

Objective

– Show that hardware verification is an abstract interpretation of a monitored operational semantics
– (... and automatize such a verification without state explosion)

Hardware Verification in VHDL (Code13)

```vhdl
loop
  clk <= not clk;
  wait for 1;
end;

Clock clk = 0 1 0

Action o := x \land \neg y on
1 0 1 0 1 ...

“clk = 1” events
```

12 See aiT WCET Analyzers of Abteil Angewandte Informatik GmbH

13 Very High Speed Integrated Circuit Hardware Description Language (VHDL) pseudo-code at the Behavioral Level.
Hardware Verification in VHDL (Specification)

```
loop
  clk <= not clk;
  wait for 1;
end;

Clock clk = 0 1 0 1 0 1 ...  
Action o := x \land \neg y on  
"clk = 1" events
```

Hardware Verification in VHDL (Monitoring)

```
loop
  clk <= not clk;
  wait for 1;
end;

Clock clk = 0 1 0 1 0 1 ...  
Action o := x \land \neg y on  
"clk = 1" events
```

Hardware Verification in VHDL (Proof)

```
loop
  clk <= not clk;
  wait for 1;
end;

Clock clk = 0 1 0 1 0 1 ...  
Action o := x \land \neg y on  
"clk = 1" events
```

Runtime monitor:
- Generates all possible entries
- Checks the property

Model checking/static analysis show the assertion to always hold

Hardware Verification (Reed-Solomon – Code)

```
x <= 0; y <= 1;
wait on clk;
loop
  x <= rnd;
  assert (o != 1);
  wait on clk;
end;

Runtime monitor:
- Generates all possible entries
- Checks the property
```

Model checking/static analysis show the assertion to always hold
Example of Challenge in Hardware/Software Verification

- Data transmission using USB/AFDX is now preferred to avionic ARINC 429 transmit and receive channels
- Challenge: prove communications correct on a USB port, given
  - a software driver in C;
  - a hardware controller in VHDL;
  - a formal specification of “correct communication”.

Example 5: Static Analysis of Avionic Safety-Critical Software

References

Objective

– Show that static analysis by abstract interpretation does scale up
– (… and report on an industrialization success story)

Example 1: CBMC

– CBMC is a Bounded Model Checker for ANSI-C programs (started at CMU in 1999).
– Allows verifying array bounds (buffer overflows), pointer safety, exceptions and user-specified assertions.
– Aimed for embedded software, also supports dynamic memory allocation using malloc.
– Done by unwinding the loops in the program and passing the resulting equation to a SAT solver.
– Problem (a.o.): does not scale up!

Example 2: ASTRÉE

– ASTRÉE is an abstract interpretation-based static analyzer for ANSI-C programs (started at ENS in 2001).
– Allows verifying array bounds (buffer overflows), pointer safety, exceptions and user-specified assertions.
– Aimed for embedded software, does not support dynamic memory allocation.
– Done by abstracting the reachability fixpoint equations for the program operational semantics.
– Advantage (a.o.): does scale up!

The Static Analysis Problem

– Given a C control/command program and a configuration file,
– effectively compute a computer representation of an overapproximation of the reachable program states from the initial states,
– in order to statically prove the absence of runtime and user-defined errors.
– Extremely difficult to scale up!

13 Physical range hypotheses for some sensor inputs
Ellipsoid Abstract Domain for Filters

- Computes $X_n = \alpha X_{n-1} + \beta X_{n-2} + Y_n$
- The concrete computation is bounded, which must be proved in the abstract.
- There is no stable interval or octagon.
- The simplest stable surface is an ellipsoid.

Filter Example [6]

```c
typedef enum {FALSE = 0, TRUE = 1} BOOLEAN;
BOOLEAN INIT; float P, X;

void filter () {
    static float E[2], S[2];
    if (INIT) { S[0] = X; P = X; E[0] = X; }
    else { P = ((((0.5 * X) - (E[0] * 0.7)) + (E[1] * 0.4))
                + (S[0] * 1.5)) - (S[1] * 0.7)); }
    E[1] = E[0]; E[0] = X; S[1] = S[0]; S[0] = P;
    /* S[0], S[1] in [-1327.02698354, 1327.02698354] */
}

void main () { X = 0.2 * X + 5; INIT = TRUE;
    while (1) {
        X = 0.9 * X + 35; /* simulated filter input */
        filter (); INIT = FALSE; }
}
```

Success Story

- A340 family (200/300/500/600): ASTRÉE is now part of the production line of the Primary Flight Control Software (130-250 000 lines)

- A380: ASTRÉE is still being tuned up to handle the Primary Flight Control Software (1000 000 lines) without false alarms
ASTRÉE Follow-on (I)

Space Software Validation by Abstract Interpretation

- ESA ITI Initiative, 2006–2008
- ENS + CEA + EADS SPACE Transportation
- Verification of the MSU software of the ATV docking the ISS

14 MSU: Monitoring and Safety Unit, ATV: Automated Transfer Véhicule, ISS: International Space Station.

THÉSÉE

- Verification of absence of runtime errors, data races and deadlocks in asynchronous safety-critical real-time embedded control/command software
- 2006–2009
- ENS + Airbus + EDF International (1600-megawatt EPR (Evolutionary Power Reactor) for the Finnish Olkiluoto 3 plant unit, to be operational in 2009)

ASTRÉE Follow-on (II)

- Aeronautics, space, automotive, railway, medical industries
- ENS + Airbus + Astrium + Barco + CS SI + Daimler-Chrysler AG + Siemens VDO / Transportation + Thales Avionics + . . .
- Static analysis verification tools for embedded software:

ASBAPROD

- Translation validation (Scade → C → ASM)
- Verification of functional properties of safety-critical real-time embedded synchronous electric flight control software, for example:
  - One and only one computer has control at any time,
  - If some input $i$ changes by $\Delta_i$ then some output $o$ changes by at most $\Delta_o$, etc
- 2006–2010
- ENS + Airbus
CONTROVERT

– CONTROl system VERificaTion
– 2006–2009
– ENS (computer scientists) + ONERA Toulouse (control theoreticians)

The Current Situation

(1) Model design
(2) Simulation
(3) Implementation
(4) Program analysis

16 greatly simplified, system dependability is simply ignored!

The Project

(1) Model design
(2) Model analysis
(3) Program analysis

Example (response analysis)

17 greatly simplified, system dependability is simply ignored!

Conclusion
Formal Methods

– Formal methods have made considerable academic progress these last 30 years
– Automatic formal methods still have to scale up for everyday industrial practice
– The high-technology industries have imperative needs in software design & verification
– Static program analysis is progressively becoming an advanced industrial practice
– Automatic verification from specification design down to program implementation is a challenge

Abstract Interpretation

– Theoretical foundations: deep unification of formal methods, semantics, modularity/incrementability, parallelism/distribution/mobility, object-orientation, complex hardware/software/communication systems, integration of continuous/discrete/probabilistic models of the physical world/user interaction, …
– Abstractions: abstract domains for safety, security, …, controlability, robustness, …
– Applications: beyond computer science, control/command, biology, …

References

[1] www.astree.ens.fr [2, 3, 4, 5, 6, 7, 10, 11, 12, 13]