APR Team

Mopsa – Static program verification by abstract interpretation

Antoine Miné

APR team
LIP6
Sorbonne Université
Paris, France

Journée Airbus–LIP6
20 février 2019
Overview

- **APR Team**: themes and activities
- **Static analysis by abstract interpretation**
  - introduction
  - example analyses developed in APR
  - **Mospa**: designing a Modular Open Platform for Static Analysis
APR Team
Principles: complementary approaches to safe and efficient software

- **Language**: design, implementation, automated analysis
- **Algorithmic**: modeling, complexity analysis, random generation

From theory to practice:

- **formal methods**: semantics (language) and combinatoric (algorithmic)
- **concrete objectives**: software prototype, industrial applications

Connections:

- teaching, from introductory programmatic courses to research-oriented master-level teaching
- strong link with IRILL (Center for Research and Innovation on Free Software)
- academic and industrial project partners (Airbus, …)
APR: a few projects

- **Analytic combinatorics for concurrency**
  - simulate and analyze large combinatorial models
  - application to generate interleavings to test concurrent systems
  - **Arbogen**: fast uniform random generator for trees

- **Multicore, concurrent, mobile programming**
  - parallel algorithms: **Graph-GPU**
  - session types for web programming (**π-calculus**)

- **Embedded system programming**
  - functional programming on micro-controllers (**Ocaml**)
  - virtual machine approach and optimisation: **OMicrob**
  - mixing synchronous and functional programming: **OCaLustre**

- **Static program analysis**
Static analysis
Static analysis

**Goal:** program verification by static analysis

```
int search(int* t, int n) {
    int i;
    for (i=0; i<n; i++) {
        if (t[i]) break;
    }
    return t[i];
}
```

```
int search(int* t, int n) {
    int i;
    for (i=0; i<n; i++) {
        // 0 ≤ i < n
        if (t[i]) break;
    }
    // 0 ≤ i ≤ n ∨ n < 0
    return t[i];
}
```

- work directly on the **source code**
- infer properties on **program executions**
- **automatically** (cost effective)
- by constructing dynamically a **semantic abstraction** of the program
- deduce program **correctness** or raise **alarms**
  - implicit specification: absence of RTE; or user-defined properties: contracts
- with **approximations** (efficient, but possible false alarms)
- soundly (no false positive)

Based on the **abstract interpretation framework**.
Abstract interpretation: theory of the approximation of (program) semantics

Principle: be tractable by reasoning at an abstract level

Concrete executions:
\{(0, 3), (5.5, 0), (12, 7), \ldots\} (not computable)

Box domain:
\[ X \in [0, 12] \land Y \in [0, 8] \] (linear cost)

Polyhedra domain:
\[ 6X + 11Y \geq 33 \land \cdots \] (exponential cost)
Abstract computations

Define an interpretation of atomic statements in the abstract domain.

Compose the interpretation to analyze the program.

**Example in the polyhedra domain**

**Assignment:**
- $X = X + 1$

translation

**Join after branches:**
- if ··· then ··· else ··· fi

convex hull

**Loop invariants:**
- while ··· do ··· done

iteration with widening
Goal: prove that a program $P$ satisfies its specification $S$

$\implies$ we consider only its abstraction $A$ and check that $A \subseteq S$
Soundness and precision

Goal: prove that a program $P$ satisfies its specification $S$

$\iff$ we consider only its abstraction $A$ and check that $A \subseteq S$

impossible unsound analysis where $A \subseteq S$ but $P \not\subseteq A$

undesirable imprecise analysis where $A \not\subseteq S$ but $P \subseteq A$
Example projects
Example projects

APR research in static analysis

Goal: analyze more complex properties, for more general programs, beyond safety analysis of embedded C programs

- analysis of Python programs (A. Fromherz, R. Monat)
- probabilistic analysis of communication protocols (A. Ouadjaout)
- analysis of program patches (D. Delmas)
- analysis of string manipulations in C (M. Journault)
- analysis of TinyOS device drivers (A. Ouadjaout)
- static analyses under weakly memory models (T. Suzanne)
- novel algorithms for numeric domains (A. Marechal)
- combining abstract interpretation and constraint programming (G. Ziat)
Analysis of Python programs

Highly **dynamic language**, lack of static information in the source

⇒ **advanced semantic static analyses can help !**

```python
def gen():
    i = 0
    while i < 10:
        i += 1
        yield i

def f():
    b = gen()
    j = 0
    while j < 5:
        j += 1
        a = b.__next__()
```

- **complex semantics** of operators
- **type-based** case analysis
- **complex control** (generators)
- **continuation-based** semantics

Python analyses:
- **formal** semantic of Python 3
- **value-analysis**, **uncaught exceptions** (A. Fromherz)
- **type-based modular analysis** (R. Monat)

Remaining challenges:
- **large library and built-ins**
- **meta-programming** introspection, dynamic classes
- **eval**
Example projects

Analysis of probabilistic properties (A. Ouadjaout)

Communication protocols in a stochastic environment

```java
int n = 0, a = 0;
while (true) {
    data = sense();
    sleep(uniform(1, B)); // Uniform backoff
    if (unicast(data)) a++; // Transmission with ack
    n++;
    sleep(S) // Save energy
}
```

- concrete Markov chain model for one parameter value
- inferred abstract Markov chain with symbolic parameters $B$, $S$
- infer automatically an abstract Markov chain from the program source
- compute quantitative properties: goodput
  safe symbolic bounds, as a function of $B$ and $S$
Modular Platform for Static Analysis
A classic analyzer (Astrée, Frama-C) has:

- a front-end
- a simplified target analysis language
- an iterator
- a tree-structure combination of abstractions with layered abstraction signatures

Limitations:

- static simplifications in the front-end cripple precision before the analysis
- restrictions to domain composition and domain reuse
Mopsa characteristics

A novel static analysis platform in construction:

- a single extensible common AST to represent
  - each source language faithfully (C, Python)
  - simplified intermediate languages (numeric fragment, structure-free, etc.)
- generalized view of abstract domains
  - obey a common signature
  - represent partial functions (gather environments over different variables)
  - iterators over AST fragments are abstract domains
  - domains can dynamically rewrite expressions (delegation to other domains)
  - abstract state can be shared by several parent domain (DAG-organisation instead of tree)
- domain composition with inter-domain communication (reduced product, disjoint union)

⇒ engineering and formalization effort

OCaml implementation, released as open-source soon (framework + C)

Goal: make analyzer design simpler and more effective.
Example: C value analyzer

Status: analysis of Juliet tests, analysis of coreutils in progress
Example: Python value analyzer

Status: analysis of a few regressions tests from Python distribution
Example: stub contract language

• inspired from ACSL
• target stub modeling (not functional verification)
• interpret formulas in abstract domains
  \[\implies\] domain dedicated to quantified formulas (strings, arrays)
• modeling of resources (memory, file descriptors, etc.)

```c
/*$
* requires: exists int i in [0, size(__file) - 1]: __file[i] == 0;
* 
* case "success":
*   local: void* fd = new FileDescriptor;
*   ensures: return == (int)fd;
* 
* case "failure":
*   assigns: _errno;
*   ensures: return == -1;
*/
int open (const char *__file, int __oflag, ...);
```