Search-Guided Generation of Properties for Program Analyzers

[1,2]Daniela Ferreiro

[1]Universidad Politécnica de Madrid (UPM) [2]IMDEA Software Institute, Madrid, Spain

WHY VALIDATE STATIC ANALYZERS?

Static analysis tools are crucial in modern software development: verification, optimization, etc.

But building analyzers is **hard**:

- **Complex, large systems**
- **Prone to subtle bugs**
- **Used in critical tasks** needing trustworthy results

Validating analyzers is also difficult!

- **Formal methods are hard to apply** directly
- **Specifications are often missing or not complete**
- The ***who checks the checker?*** problem rises

In practice, extensive **testing** is the **most realistic** option.

- But, unit-tests miss integration bugs
- It's hard to define **testing oracles**
- Generation of **complex data structures** with hard-to-test conditions is challenging

HOW? *Checkification* Algorithm

1 Source program or set of benchmarks

tree.pl

```
:- module(tree, _, [assertions]).

:- pred insert(+nnegint, +tree, -).

insert(X,empty,tree(empty,X,empty)).
insert(X,tree(LC,X,RC),tree(LC,X,RC).
insert(X,tree(LC,Y,RC),tree(LC_p,Y,RC) :-
  X<Y,
  insert(X,tree(X,LC,LC_p).
insert(X,tree(LC,Y,RC),tree(LC,Y,RC_p) :-
  X>Y,
  insert(X,tree(X,RC,RC_p).

:- pred belongs(+nnegint, +tree).
...
:- pred root(+tree,-).
...
```

2 Program is analyzed with domain $\mathcal{D}$

tree_analysis.pl

```
:- module(tree_analysis, _, [assertions]).
:- true insert(N, T1, T2)
   : (nnegint(N), tree(T1), var(T2))
  => (nnegint(N), tree(T1), non_empty_tree(T2)).
insert(X,empty,tree(empty,X,empty)).
insert(X,tree(LC,X,RC),tree(LC,X,RC).
insert(X,tree(LC,Y,RC),tree(LC_p,Y,RC) :-
  true(nnegint(X),tree(LC),nnegint(Y),tree(RC)),
  X<Y,
  true(...),
  insert(X,tree(X,LC,LC_p),
  true(...).
insert(X,tree(LC,Y,RC),tree(LC,Y,RC_p) :-
  true(...),
  X>Y,
  true(...),
  insert(X,tree(X,RC,RC_p),
  true(...).
...
```

3 *true* status is replaced with *check*

tree_check.pl

```
:- module(tree_check, _, [assertions,rtchecks]).
:- check insert(N, T1, T2)
   : (nnegint(N), tree(T1), var(T2))
  => (nnegint(N), tree(T1), non_empty_tree(T2)).
insert(X,empty,tree(empty,X,empty)).
insert(X,tree(LC,X,RC),tree(LC,X,RC).
insert(X,tree(LC,Y,RC),tree(LC_p,Y,RC) :-
  check(nnegint(X),tree(LC),nnegint(Y),tree(RC)),
  X<Y,
  check(...),
  insert(X,tree(X,LC,LC_p),
  check(...).
insert(X,tree(LC,Y,RC),tree(LC,Y,RC_p) :-
  check(...),
  X>Y,
  check(...),
  insert(X,tree(X,RC,RC_p),
  check(...).
...
```

4 Generation of tests

Use **assertion preconditions** as **generators** of valid test inputs.

→ Prolog's declarative nature lets properties be expressed as predicates.

Traditional Prolog uses **depth-first** search: **efficient** but **incomplete**.

Classic Ciao Prolog **allows other search strategies**.

PROPOSED SOLUTION

A **new** mechanism to execute predicates under non-standard search rules (**breadth-first**, **id**, **random**, **guided**, ...) as well to explore **diverse inputs**.

✓ Not rewriting predicates, but running them with **alternative execution strategies**.

✓ More expressive specifications.

```
:- search_rule(tree/1,bf).

:- prop tree/1 + regtype.
tree(empty).
tree(tree(LC,N,RC)) :-
  tree(LC),
  gen([sr(df)],(nnegint(N)))
  tree(RC).

% Check if tree T is sorted
sorted_tree(T) :- ...

% Constrains the sum of all node values in
% tree T to equal N
tsum(T,N) :- ...

:- check insert(N, T1, T2)
   : (nnegint(N), tree(T1), var(T2))
  => (nnegint(N), tree(T1), non_empty_tree(T2)).
  + gen([sorted_tree(T1),tsum(T1,10)]).
```

✓ Facilitates **automatic + user-guided** test case generation.

✓ Pushes further towards general-purpose **flexible search** in (C)LP.

E.g., test cases `insert(N,T1,T2)` where `N` is instantiated to a non-negative integer, `T1` is a tree, and `T2` is a free variable.

But, in the `insert/3` specification, the generator is further guided by auxiliary properties: `sorted_tree/1` and `tsum/2`, which restrict the structure and content of the input trees.

- Test 1: `insert(2,tree(empty,10,empty),T)`
- Test 2: `insert(5,tree(empty,2,tree(empty,8,empty)),T)`
- ...
- Test n: `insert(8,tree(empty,1,tree(empty,4,tree(empty,5,empty))),T)`

If any of these **test cases** produce a **run-time error** then there is a **bug**!

I.e., if a run-time check reports a violation, then the analyzer must have inferred the assertion incorrectly, revealing a bug in the analyzer.

Current experimental outcomes

Many **applications**, depending on which parts of the system are **trusted**:

+ Debugging **Abstract Domains**.
+ Testing the **Abstract Interpretation Engine**.
  Testing less trusted fixpoints and options (e.g., incremental analysis).
+ Debugging **trust assertions** and custom transfer functions
+ Testing the **overall consistency** of the **framework**.
  E.g., when semantics is underspecified, check at least that runtime and static semantics agree.
+ Integration Testing of the Analyzer.
+ Testing **external** or **third party solvers** (e.g., PPL).

Analyzed programs with increasing levels of complexity:

→ Success in finding **known bugs** or unsupported features in old versions (e.g., rational terms, attributed variables).

→ Actual analysis bugs found, mainly in less mature domains.

→ Some **inconsistencies** found in the **framework** (e.g., in interpretation of native properties by analyzer and runtime-checks).

→ Some bugs in other components found and fixed (e.g., the analysis output).

→ **Reasonable overhead**, with test execution time < 60s.

Full paper?