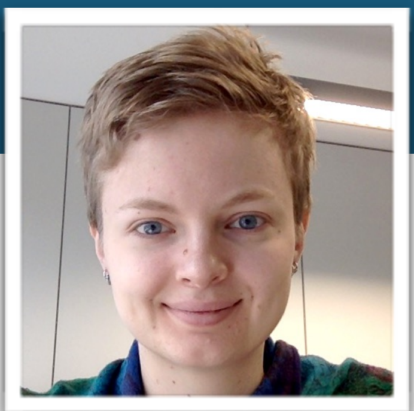
IMDEA Software Institute
Technical University of Madrid(UPM)

Runtime Verification Challenges
in the context of (Constraint) Logic Programming



Nataliia Stulova
Jose F. Morales
Manuel V. Hermenegildo

{nataliia.stulova,josef.morales,
manuel.hermenegildo}@imdea.org

Context

Runtime verification:
A technique that detects errors in programs, based on:
- Providing a specification for the program.
- Observing the behaviour of the running program.
- Detecting (and reacting to) any violations of the specified behaviour.

Some research challenges:
(a)Enhancing existing formalisms and specification languages.
(b)Reducing run-time overhead:
- Optimising program instrumentation.
- Combination with static analysis.

(Constraint) Logic Programming:
- Programs are expressed in terms of relations (facts, rules).
- Computation is initiated by running query on those relations.
- Higher-order programming is supported (patterns, templates, passing relations as arguments).

Experimental setting:
- *Prolog*-based multi-paradigm language.
- Rich *assertion language* for program specification.
- Availability of both compile time / runtime verification.

(a) More Expressive Specifications

Currently in most (C)LP systems not much can be specified about the predicate arguments of higher-order predicates, yet higher-order calls are commonly used:

```
% - - - - - - - - - a simple higher-order program - - - - - - - -
% 'old' specification: argument Cmp of predicate min/4 can be called

:- pred min(X,Y,Cmp,M) : callable(Cmp).

min(X,Y,Cmp,M) :- Cmp(R,X,Y), R <= 0, M = X.        % rule
min(X,Y,_  ,Y).                                      % fact
                        higher-order
less( 0,A,A).               call         lt('=',A,A).
less(-1,A,B) :- A < B.                   lt('<',A,B) :- A < B.
less( 1,_,_).                            lt('>',_,_).
```

We introduce the notion of *predicate properties* (like `comparator/1` below) that reuse Ciao assertion language to describe predicate arguments that are bound predicates themselves:

```
% 'new' specification: call and success patterns of a predicate
% that is bound to the Cmp variable
:- comparator(Cmp) {
:- pred Cmp(Res,X,Y) :  num(M) , num(N)      % precondition
                     => between(-1,1,Res).   % postcondition
}.
:- pred min(X,Y,Cmp,M) : comparator(Cmp).
```

This allows to capture precisely call and success patterns of higher-order calls and detect undesired and/or unexpected program behaviours.

PPDP'14
Assertion-based Debugging of Higher-Order (C)LP Programs

(b) Faster Checks of Program Specifications

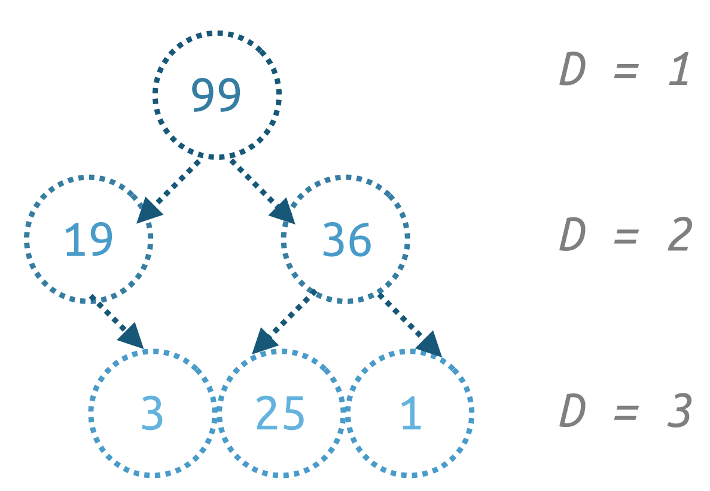
Runtime

Assertion-based runtime checking is costly: often pre- and postcondition checks are duplicated in recursive calls. Possible practical solution: cache already performed checks for properties of recursive data structures (lists, trees, etc.) up to some depth *D* and look up the results on demand.

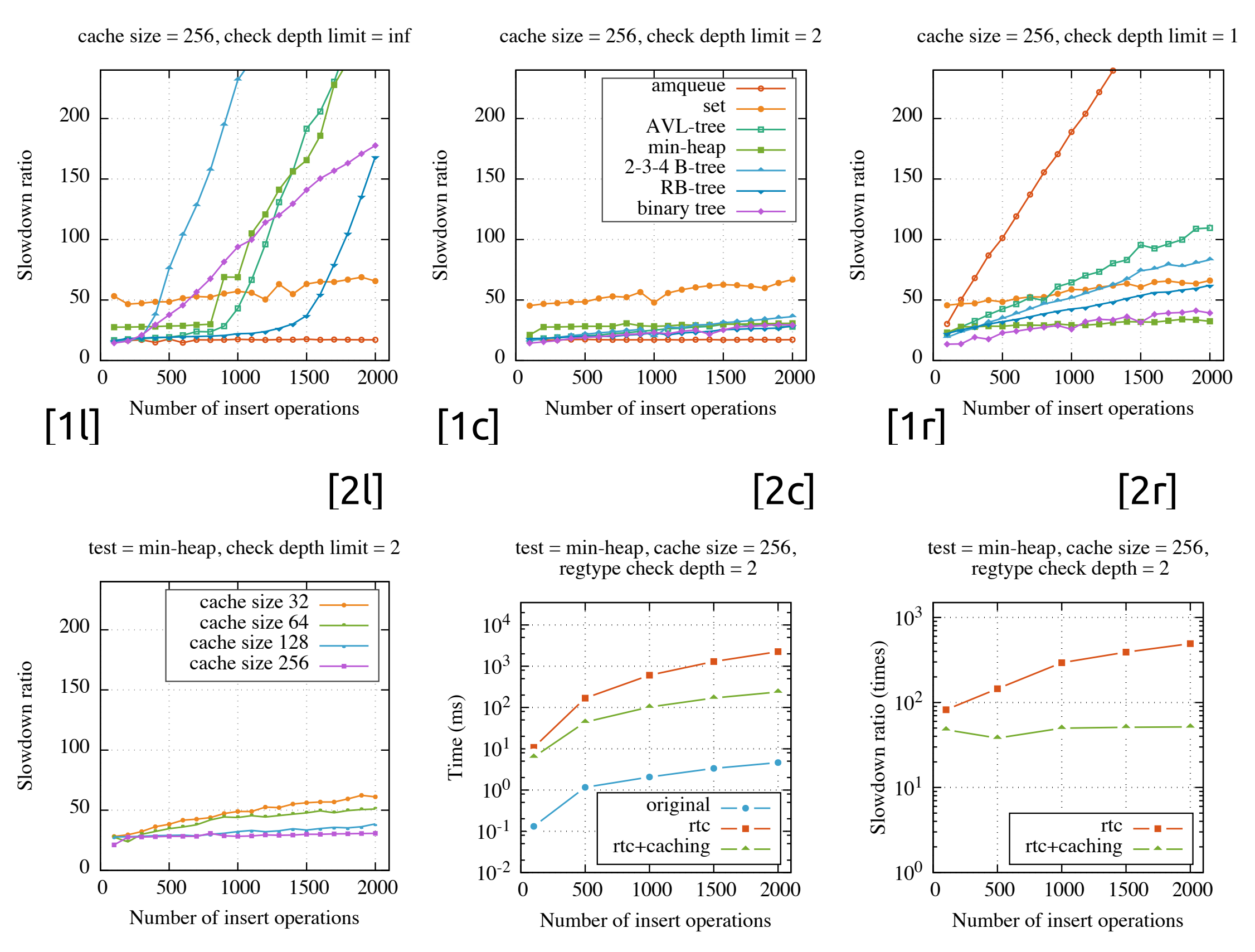
```
t(e).                     % empty binary tree
t(L,_,R) :- t(L), t(R).   % left and right subtrees

t(t(e,19,t(e,3,e)),99,t(t(e,25,e),36,t(e,1,e)))

t(                 ,99,                          )
  t( ,19,          )     t(           ,36,       )
   e     t( ,3, )          t( ,25, )     t( ,1, )
            e  e              e   e         e  e
```



Current experimental outcomes:
- relatively small cache size can be used (~250 elements) to gain overhead reduction;
- caching recursive data structures up to term depth 2 yields 1-2 orders of magnitude runtime overhead reduction (see Fig. 1c VS 1l and 1r);
- the slowdown ratio of programs with run-time checks using caching is almost constant, in contrast with the linear growth in the case where caching is not used (see Fig. 2l, 2c and 2r);

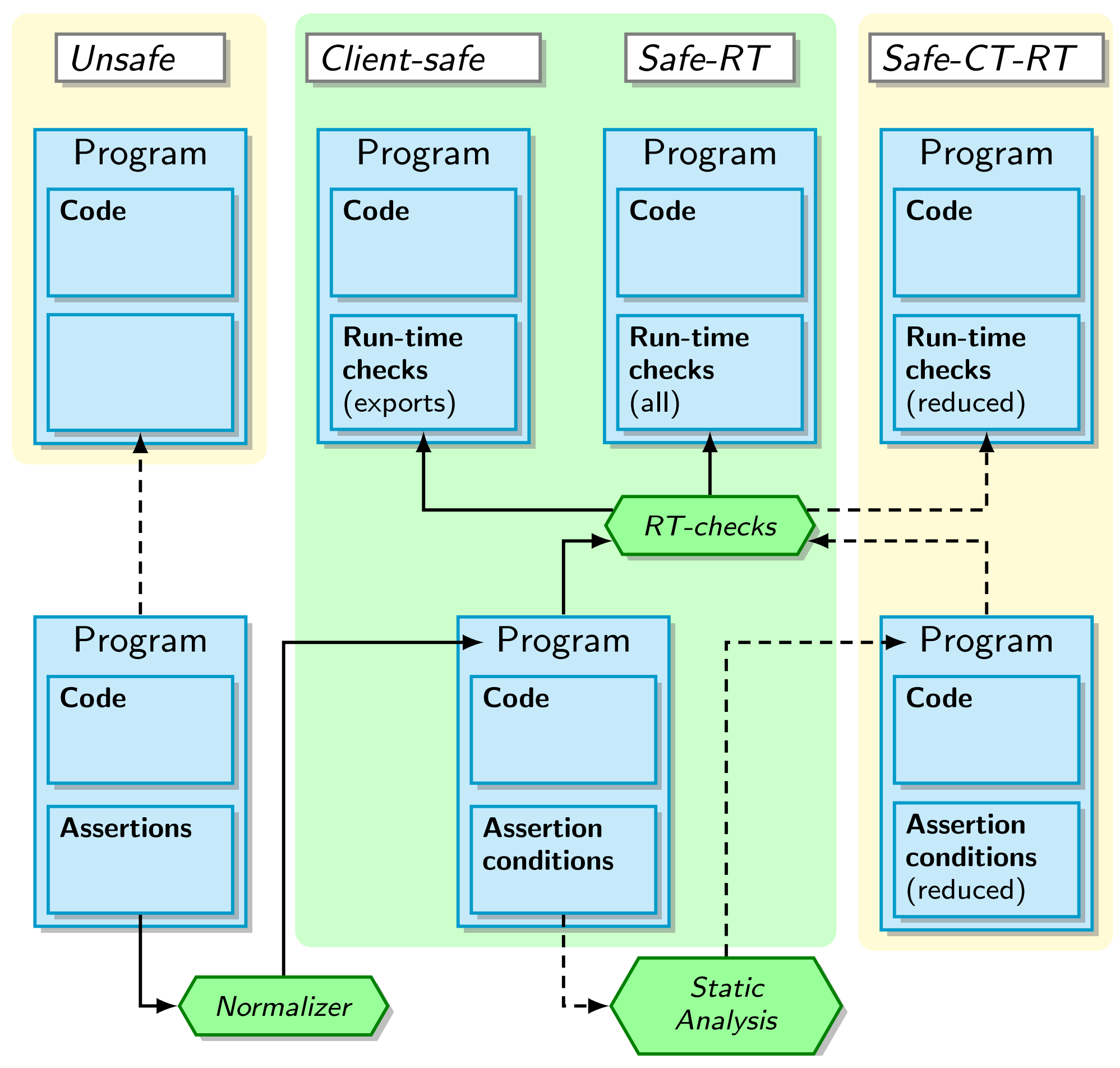


ICLP'15
Practical Runtime Checking via Unobtrusive Property Caching

Compile time

Parts of a program specification can be proved to hold true during compilation, thus there would be no need to check them at runtime. In our work we perform static analysis (based on the *abstract interpretation* technique) at compile time that checks specifications of variable sharing, variable freeness and terms shape (and many other properties).

We study runtime checking overhead decrease in the following 4 practical scenarios:
- no checks (baseline performance, no safety guarantees);
- checks only for public program interface (minimal overhead and safety guarantees);
- checks for both public and private program predicates (maximal overhead and safety guarantees);
- checks for the public interface and for the specifications of private predicates that were not proved at compile time.



submitted to PPDP'16
Reducing the Overhead of Assertion Run-time Checks via Static Analysis

This research has been partially funded by the EU FP7 agreement no 318337 ENTRA, Spanish MINECO TIN2012-39391 StrongSoft, TIN2008-05624 DOVES, Comunidad de Madrid TIC/1465 PROMETIDOS-CM, TIN2015-67522-C3-1-R TRACES projects, and the Madrid M141047003 N-GREENS program.