Combinations of Analysis Techniques for Sound and Efficient Software Verification

Habilitation Thesis Defense

Nikolai Kosmatov

Palaiseau, November 20th, 2018

Software analysis: from foundations to combinations

Theoretical foundations established in the 20th century:

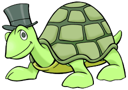
- Undecidability of program analysis [Rice, 1953]
- Floyd-Hoare logic [Floyd, 1967][Hoare, 1969]
- Weakest precondition calculus [Dijkstra, 1975]
- Symbolic execution for testing [King, 1976]
- Abstract interpretation [P.& R. Cousot, 1977]
- Model-checking [Emerson, Clarke, 1980][Queille, Sifakis, 1982]

Efficient tools and convincing practical applications appeared later:

- PolySpace to detect Ariane 5 bug after 1996 [Deutsch, 2003]
- Astrée used by Airbus [Cousot, ESOP 2005]
- Fluctuat used by Airbus [Delmas, FMICS 2009]
- Caveat used by Airbus to certify A380 [Randimbivovolovna, FM'99]
- Sage widely used by Microsoft [Godefroid, NDSS 2008]

Static vs. Dynamic analysis techniques

- for a long time, seen as orthogonal and used separately
- more recently, realization of potential synergy and complementarity



Static analysis

Analyzes the source code without executing it

- Instructions reported as safe are safe (complete)
- Detected *potential* errors can be safe (imprecise)

Dynamic analysis

Executes the program on some test data

- Detected errors are really errors (precise)
- Cannot cover all executions (incomplete)

This talk focuses on combinations of various analyses in Frama-C

Outline

Tool context: Frama-C, a platform for analysis of C code

From testing to static analysis

From executable specifications to counterexamples

A proof-friendly view of test coverage criteria

Conclusion and perspectives

Outline

Tool context: Frama-C, a platform for analysis of C code

From testing to static analysis

From executable specifications to counterexamples

A proof-friendly view of test coverage criteria

Conclusion and perspectives

A brief history

- 90's: CAVEAT, Hoare logic-based tool for C code at CEA
- 2000's: CAVEAT used by Airbus during certification process of the A380 (DO-178 level A qualification)
- 2002: Why and its C front-end Caduceus (at INRIA)
- 2006: Joint project on a successor to CAVEAT and Caduceus
- 2008: First public release of Frama-C (Hydrogen)
- 2012: New Hoare-logic based plugin WP developed at CEA
- Today: Frama-C v.17 Chlorine
  - Multiple projects around the platform
  - A growing community of users...
  - and of developers
- Used by, or in collaboration with, several industrial partners

Frama-C at a glance

frama C

Software Analyzers

- A Framework for Modular Analysis of C code
- Developed at CEA List
- Released under LGPL license
- Kernel based on CIL [Necula, CC 2002]
- ACSL annotation language
- Extensible plugin oriented platform
  - Collaboration of analyses over same code
  - Inter plugin communication through ACSL formulas
  - Adding specialized plugins is easy

Publications: [SEFM 2012, FAC 2015]

ACSL: ANSI/ISO C Specification Language

- Based on the notion of contract, like in Eiffel, JML
- Allows users to specify functional properties of programs
- Allows communication between various plugins
- Independent from a particular analysis
- Manual at `http://frama-c.com/acsl`

Basic Components

- First-order logic
- Pure C expressions
- C types + $\mathbb{Z}$ (integer) and $\mathbb{R}$ (real)
- Built-in predicates and logic functions particularly over pointers: `\valid(p)` `\valid(p+0..2)`, `\separated(p+0..2,q+0..5)`, `\block_length(p)`

Outline

Tool context: Frama-C, a platform for analysis of C code

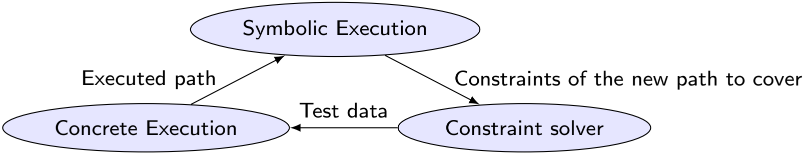
From testing to static analysis

From executable specifications to counterexamples

A proof-friendly view of test coverage criteria

Conclusion and perspectives

First activities at CEA: PathCrawler test generator



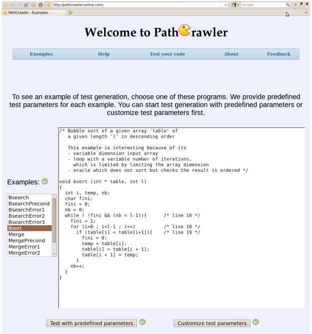
- Performs Dynamic Symbolic Execution (DSE)
- By Nicky Williams with B.Botella,M.Delahaye,N.K.,P.Mouy,M.Roger
- Uses code instrumentation, concrete and symbolic execution, constraint solving (relies on COLIBRI solver by Bruno Marre)
- Sound and relatively complete: doesn't approximate path constraints

My contributions: test generation strategies, interprocess communication, output features, treatment of preconditions, integration into Frama-C
Publications: [ISSRE 2008, AST 2009, JFPC 2010, RV 2013]

PathCrawler-online testing service

Main author of PathCrawler-online in 2009–2010

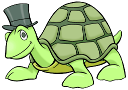
- With interns A.Kouider, N.Dugué, and later Richard Bonichon (author of the new interface in 2011–2012)
- Detailed results: concrete & symbolic outputs, path predicates, coverage...
- Challenge: executes users' code
- Widely used for teaching in Paris, Orléans, Orsay, Evry, Strasbourg, Bourges, Toulouse. . . , but also in China, Germany, USA, India, Iran, Austria, Canada...



Publications: [SOSE 2013, CSTVA 2011, IGI Global 2013]

The SANTE approach: motivation and goals

Detection of runtime errors: two approaches



Static analysis:
abstract interpretation based
value analysis

Issue: leaves unconfirmed errors that can be safe

Dynamic Analysis:
DSE based
test generation

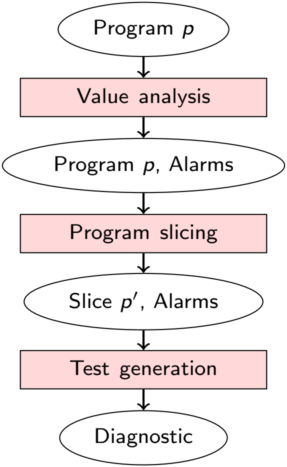
Issue: cannot detect all errors since test coverage is partial

Goal: Combine both techniques to detect runtime errors more efficiently

PhD work of Omar Chebaro in 2008-2011 (co-supervised with Alain Giorgetti, Jacques Julliand)

Publications: [Chebaro et al, TAP 2009, TAP 2010, SAC 2012, ASEJ 2014]

SANTE: Methodology for detection of runtime errors



- Value analysis detects alarms
- Slicing reduces the program (w.r.t. one or several alarms)
- Test generation (PathCrawler) on a reduced program to diagnose alarms (after adding error branches to trigger errors)
- Various slicing options based on alarm dependencies
- Diagnostic
  - bug if a counter-example is generated
  - if not, and all paths were explored, the alarm is safe
  - otherwise, unknown

SANTE: Experiments

- 9 benchmarks with known errors (from Apache, libgd, ...)

Alarm classification:

- all known errors found by SANTE
- SANTE leaves less unclassified alarms than VALUE or PathCrawler alone

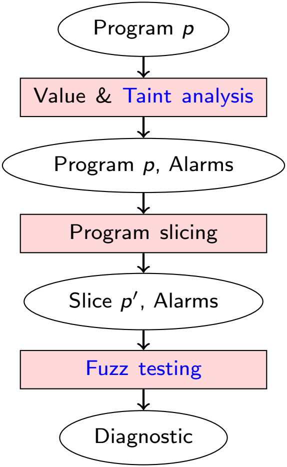
Program reduction:

- 32% in average, up to 89% for some examples
- program paths in counter-examples are in average 19% shorter

Execution time:

- Average speedup w.r.t. testing alone is 43% (up to 98% for some examples)

SANTE: Applications to security



- Reused in EU FP7 project STANCE (CEA List, Dassault, Search Lab, FOKUS,...)
- Taint analysis to identify most security-relevant alarms
- Fuzz testing (Flinder tool) for efficient detection of vulnerabilities
- Applied to the recent Heartbleed security flaw (found in 2014 in OpenSSL)

Publication: [Kiss et al., HVC 2015]

- Another application (in EU project VESSEDIA) in progress

SANTE: Selected Related Work

- CHECK'N'CRASH: combines ESC/JAVA, random testing JCRASHER [Csalner et al, ICSE 2005]
- DAIKON: detects likely invariants [Ernst et al., SCP 2007]
- DSD CRASHER: combines DAIKON, CHECK'N'CRASH [Smaragdakis et al, TAP 2007]
- SYNERGY, BLAST, YOGI: combine testing and partition refinement [Gulavani et al, FSE 2006][Beyer et al, STTT 2007]
- DYTA: follows the SANTE approach with CFG connectivity instead of slicing [Ge et al, ICSE 2011]
- [Chistakis et al, FM 2012] consider unsound static analysis with testing
- [Chimento et al, RV 2015] combine static analysis with runtime verification

Slicing: Soundness for verification

Research question: V&V on slices instead of the initial program

- If an error is found in a slice, is it present in the initial program?
- If there are no errors in a slice, what about the initial program?

Main results:

- a new soundness property of slicing
- a formal link between errors in the slice and the initial program
  - an error in the slice can only be hidden in the init. program by an erroneous or non-terminating stmt non-preserved in the slice
- formalization and proof in Coq

PhD work of Jean-Christophe Léchenet in 2015-2018 (co-supervised with Pascale Le Gall)

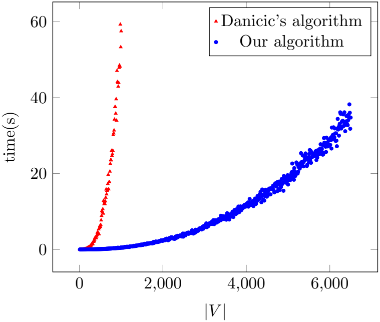
Publications: [Léchenet et al, FASE 2016, FAC 2018]

Towards generic slicing: Control dependencies

Research question: Efficient computation of control dependencies for a language with unstructured control-flow [Danicic et al. TCS 2011]

Main results:

- Formalization of Danincic's algorithm in Coq
- a new, more efficient algorithm to compute control dependencies
- its formalization and proof in Why3



PhD work of Jean-Christophe Léchenet in 2015-2018 (co-supervised with Pascale Le Gall)

Publication: [Léchenet et al, FASE 2018]

Outline

Tool context: Frama-C, a platform for analysis of C code

From testing to static analysis

From executable specifications to counterexamples

A proof-friendly view of test coverage criteria

Conclusion and perspectives

E-ACSL: Executable specifications and efficient runtime assertion checking

- Very active research project
- With Julien Signoles, 2 postdocs Mickaël Delahaye, Kostyantyn Vorobyov, 2 interns Guillaume Petiot, Arvid Jakobsson, PhD student Dara Ly...

- My contributions: combined analyses using E-ACSL, design of efficient memory models, detection of temoral errors, optimizations by static analysis, evaluation, CRV competitions...
- Two patents on efficient shadow-memory based solutions for memory monitoring (with K.Vorobyov, J.Signoles)

Publications: [SAC 2013, RV 2013, JFLA 2015, SAC 2015, SCP 2016, ISOLA 2016, RV 2017, ISMM 2017, TAP 2018, HILT 2018]

From ACSL to E-ACSL

ACSL was designed for static analysis tools only

- cannot execute some terms/predicates (e.g. unbounded quantification)
- cannot be used by dynamic analysis tools (e.g. testing or monitoring)

E-ACSL: Executable subset of ACSL:

- it is verifiable in finite time, suitable for runtime assertion checking
- limitations: only bounded quantification, no axioms, no lemmas
- Includes builtin memory-related predicates, for a pointer p:

Builtin predicate	Description
`\valid(p)`	p is a valid pointer
`\initialized(p)`	$*p$ has been initialized
`\block_length(p)`	Length of p's memory block
`\base_address(p)`	Base address of p's memory block
`\offset(p)`	Offset of p in its memory block

E-ACSL2C: monitoring optimized by static analysis

E-ACSL2C is a runtime verification tool for E-ACSL specifications:

- it translates annotated program p into another program p'
- p' exits with error message if an annotation is violated
- otherwise p and p' have the same behavior

Goal: avoid the monitoring of irrelevant statements

- Not necessary to monitor all memory locations

Solution: A pre-analysis of the input program

- Backward data-flow analysis
- Over-approximates the set of variables that must be monitored to verify memory-related annotations
- Identified irrelevant memory locations are not monitored
- Provides a significant speedup

Modular Deductive Verification in a Nutshell

```
// Pre_f assumed
f(<args>){
  code1;
// Pre_g to be proved
  g(<args>);
  // Post_g assumed
  code2;
}
// Post_f to be proved
```

```
// Pre_f assumed
f(<args>){
  code1;
// I to be proved
  while(C){
// I ∧ C assumed
    code3;
// I to be proved
  }
// I ∧ ¬C assumed
  code2;
}
// Post_f to be proved
```

A proof failure can be due to various reasons!

For convenience, we say:
A *subcontract* of f is the contract of a called function or loop in f.

Example: Several reasons for the same proof failure

```
/*@ requires n>=0 && \valid(t+(0..n-1));
    assigns \nothing;
    ensures \result != 0 <==>
      (\forall integer j; 0 <= j < n ==> t[j] == 0);
*/
int all_zeros(int t[], int n) {
  int k;
  /*@ loop invariant 0 <= k <= n;
      loop invariant \forall integer j; 0<=j<k ==> t[j]==0;
      loop assigns k;
      loop variant n-k;
  */
  for(k = 0; k < n; k++)
    if (t[k] != 0)
      return 0;
  return 1;
}
```

Can be proven with Frama-C/WP

Example: Several reasons for the same proof failure

Postcondition unproven…

```
/*@ requires n>=0 && \valid(t+(0..n-1));
    assigns \nothing;
    ensures \result == 0 <==>
      (\forall integer j; 0 <= j < n ==> t[j] == 0);
*/
int all_zeros(int t[], int n) {
  int k;
  /*@ loop invariant 0 <= k <= n;
      loop invariant \forall integer j; 0<=j<k ==> t[j]==0;
      loop assigns k;
      loop variant n-k;
  */
  for(k = 0; k < n; k++)
    if (t[k] != 0)
      return 0;
  return 1;
}
```

…because it is incorrect.

Example: Several reasons for the same proof failure

Postcondition unproven. . .

```
/*@ requires n>=0 && \valid(t+(0..n-1));
    assigns \nothing;
    ensures \result != 0 <==>
      (\forall integer j; 0 <= j < n ==> t[j] == 0);
*/
int all_zeros(int t[], int n) {
  int k;
  /*@ loop invariant 0 <= k <= n;
      loop invariant \forall integer j; 0<=j<k ==> t[j]==0;
      loop assigns k;
      loop variant n-k;
  */
  for(k = 0; k < n; k++)
    if (t[k] != 0)
      return 0;
  return 0;
}
```

. . . because the code is incorrect.

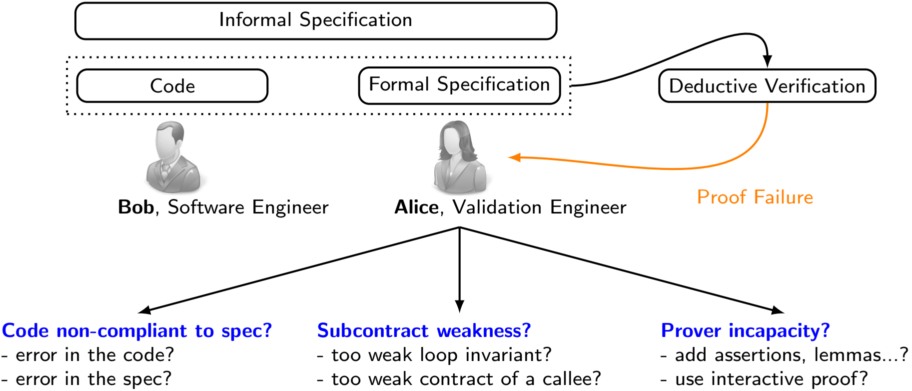
Example: Several reasons for the same proof failure

```
/*@ requires n>=0 && \valid(t+(0..n-1));
    assigns \nothing;
    ensures \result != 0 <==>
      (\forall integer j; 0 <= j < n ==> t[j] == 0);
*/
int all_zeros(int t[], int n) {
  int k;
  /*@ loop invariant 0 <= k <= n;
      loop invariant \forall integer j; 0<=j<k ==> t[j]==0;
      loop assigns k;
      loop variant n-k;
  */
  for(k = 0; k < n; k++)
    if (t[k] != 0)
      return 0;
  return 1;
}
```

Postcondition unproven. . .

. . . because a loop invariant is missing.

The STADY approach: Motivation and goals



Goals of STADY: a complete verification methodology to

- automatically and precisely diagnose proof failures,
- provide a counter-example to illustrate the issue

STADY: Methodology for diagnosis of proof failures

- Define three kinds of proof failures:
  - non-compliance (direct conflict betw. code and spec)
  - subcontract weakness (too weak contract for some loop or callee)
  - prover incapacity (the property holds, but is not proven)
- Perform dedicated instrumentation allowing to detect non-compliances and subcontract weaknesses
- Apply DSE-based test generation (PathCrawler) to try to find a counter-example and to classify the proof failure
- Indicate a more precise feedback (if possible, with a counter-example) to help the user to understand and to fix the proof failure

PhD work of Guillaume Petiot in 2012–2015 (co-supervised with Alain Giorgetti and Jacques Julliand), in collaboration with B.Botella, J.Signoles

Publications: [Petiot et al, TAP 2014, SCAM 2014, TAP 2016, FAC 2018]

Instrumentation for non-compliance detection

```
/*@ requires Pre_g;
    ensures Post_g; */

Type_f g(...) {

  code1;

}
```

$\rightarrow$

```
Type_f g(...) {
  fassert(Pre_g);

  code1;


  fassert(Post_g);
}
```

Principle:

- translate annotations into C code, similarly to runtime assertion checking, but in a way that DSE can trigger errors
- details in [Petiot et al, SCAM'14]

Instrumentation for subcontract weakness detection:

```
/*@ assigns x1,..,xN;
      ensures Post_g; */

Type_g g(...) {
  code3;
}

Type_f f(...) {
  code1;
  g(Args);
  code2;
}
```

$\rightarrow$

```
Type_g g_sw(...) {
  x1 = NonDet();
...
  xN = NonDet();
  Type_g ret=NonDet();
  fassume(Post_g);
 return ret;
} //respects Post_g
Type_g f(...) {
  code1
  g_sw(Args);
  code2;
}
```

- **Principle:** Replace the callee/loop code by **the most general code** respecting its contract, then try to trigger errors with DSE
- requires (`loop`) `assigns` clauses

STADY: Initial experiments

- 26 annotated (provable) programs (from [Burghardt, Gerlach])
- 2036 mutants generated (erroneous code, erroneous or missing annotation), 1574 unproven
- STADY is applied to classify proof failures

Alarm classification:

- STADY classified ∼95.5% proof failures

Execution time: comparable to WP

- WP takes in average 2.2 sec. per mutant (13 sec. per unproven mutant)
- STADY takes in average 2.2 sec. per unproven mutant

Partial coverage:

- Testing with partial coverage remains efficient in STADY

Complex counterex. can be found as well: STADY found a counterex. with runLen: 471,360,70,111,41,71 for Timsort [de Gouw, CAV 2015]

STADY: Selected related work

- SPARK extracts counterexamples from a solver counter-model [Dailler et al, J. Log. Alg. Meth. 2018]
- CBMC also exploits counter-models [Groce et al, CAV 2014]
- EIFFEL uses function inlining and loop unrolling [Tschannen et al., VSTTE 2013]
- DAFNY follows an approach similar to STADY for non-compliances [Christakis et al, TACAS 2016]
- Proof tree analysis for KEY [Gladisch, TAP 2009][Engel, Hähnle, TAP 2007]
- KEYTESTGEN generates tests from partial proofs in KEY [Ahrendt et al, KeyBook 2016]

Outline

Tool context: Frama-C, a platform for analysis of C code

From testing to static analysis

From executable specifications to counterexamples

A proof-friendly view of test coverage criteria

Conclusion and perspectives

Support of advanced test coverage criteria

- Very active research project
- With S.Bardin, V.Prevosto, N.Williams, B. Marre, L.Correnson (CEA), D.Mentré (MERCE), M.Papadakis (Luxembourg)...
- With 2 postdocs Mickaël Delahaye, Michaël Marcozzi, an intern Thibault Martin

Main results:

- Label specification mechanism to express a large range of coverage criteria
- An efficient test generation technique for labels
- The LTest toolset for labels: annotation, test generation, detection of infeasible test objectives, coverage evaluation
- HTOL (Hyperlabel Test Objective Language), the recent extension of labels supporting hyperproperties (MCDC), dataflow criteria...

Publications: [ICST 2014, TAP 2014, ICST 2015, ICST 2017a, ICST 2017b, ICSE 2018, ISOLA 2018]

Labels, a mechanism to specify test objectives

Basic definitions

Given a program P, a label l is a pair (loc, φ), where:

- φ is a well-defined predicate at location loc in P
- φ contains no side-effects

Example:

```
statement_1;
// l1:  x==y
// l2:  !(x==y)
if (x==y && a<b)
{...};
statement_3;
```

Benefit: express a large class of coverage criteria, allow for their efficient and generic support

LTest: a label-oriented test generation

DSE*: an efficient test generation technique for labels

- Tight instrumentation totally prevents “complexification”
- Iterative Label Deletion: discards some redundant paths
- Both techniques can be implemented in a black-box manner

DSE* dramatically improves test generation performances

- APEX reports an average overhead >272x [Jamrozik et al, TAP 2013]
- DSE* leads to an average overhead of 2.4x [Bardin et al, ICST 2014]

LTest: detection of infeasible test objectives

Coverage criteria (decision, mcdc, etc.) play a major role in testing

The enemy: Uncoverable test objectives

- waste generation effort, imprecise coverage ratios
- cause: structural coverage criteria are ... structural
- detecting uncoverable test objectives is undecidable

Recognized as a hard and important issue in testing

Idea. The test objective "reach location *loc* and satisfy predicate p" is uncoverable	$\Leftrightarrow$	the assertion `assert` $(\neg p)$; at location *loc* is valid

Apply static analysis techniques to show validity of assertions

LTest: Experiments with detection of infeasible labels

- automatic, sound and generic method
- new combination of existing verification techniques
- experiments for 12 programs and 3 criteria (CC, MCC, WM):
  - strong detection power (95%),
  - reasonable detection speed ($\leq$ 1s/obj.),
  - test generation speedup (3.8x in average),
  - more accurate coverage ratios (99.2% instead of 91.1% in average, 91.6% instead of 61.5% minimum)

PathCrawler/LTest: Towards an industrial adoption

MERCE, a research branch of Mitsubishi Electric, developed additional modules for automatic annotation of labels, generation of stubs, generation of test sheets targeting their industrial needs

MERCE evaluated the complete automatic tool

- on industrial code of 80,000 lines, 1,300 functions in 150 files
- the tool was able to parse and annotate 100% of the files and generate test cases for 86% of functions
- the generation took < 1 day instead of $\sim$230 days manually

Those very good results are very encouraging
for pushing the technology in business units
Publication: [Bardin et al, ISOLA 2018]

Outline

Tool context: Frama-C, a platform for analysis of C code

From testing to static analysis

From executable specifications to counterexamples

A proof-friendly view of test coverage criteria

Conclusion and perspectives

Other activities

Support of relational properties (in deduct.verif., testing, RAC, STADY)

- Relies on self-composition. Used in the EU VESSEDIA project.

PhD work of Lionel Blatter since 2015 (co-supervised with Pascale Le Gall, Virgile Prevosto). Publications: [TACAS 2017, TAP 2018].

Deductive verification of concurrent programs

- Relies on a code transformation simulating interleavings.

PhD work of Allan Blanchard in 2012–2016 (co-supervised with Frédéric Loulergue, Matthieu Lemerre). Publications: [FMICS 2015, SCAM 2016, VPT 2017, CSTVA 2016, COMPLAN 2018].

Verification of IoT software (in EU projects DEWI and VESSEDIA)

- Several modules of Contiki OS verified. Some errors detected.

In collaboration with 2 interns Frédéric Mangano, Alexandre Peyrard, and Allan Blanchard, Simon Duquennoy (INRIA), Frédéric Loulergue (NAU), Shahid Raza (RISE). [CRISIS 2016, RedIOT 2018, TAP 2018, NEM 2018].

Conclusion

- Combining Static and Dynamic analyses can be beneficial for various domains of software verification:
  - detection of runtime errors and security vulnerabilities,
  - deductive verification,
  - runtime assertion checking,
  - test generation, . . .
- Both ways: static helps dynamic and dynamic helps static
- Frama-C provides a rich and extensible framework for combined analyses

Challenges for combinations of analyses

Efficiency (in a large sense)

- Solving the target program more efficiently than each of the combined techniques
- Criteria include analysis time, number of detected defects, precision

Soundness

- Assumptions and conclusions of the combined techniques should be properly taken into account
- A (semi-)formal justification of soundness is desirable

Specification mechanisms

- Analysis should rely on well-defined and sufficiently expressive specification mechanisms
- New or adapted specification mechanisms can become necessary

Practical applicability

- Requires to carefully take into account the needs of the users, to communicate, to accompany in evaluation and application of tools

Perspectives:
A generic program slicer for unstructured programs

- Providing a dedicated slicer for a given language can be difficult
- Especially for unstructured control-flow (goto, break)

Goals:

- Create a generic program slicer in the presence of errors and nontermination
- Use a CFG-based program representation
- Connect to various programming languages
- Formalize and prove the core algorithm
- Use other analyses (value analysis) to increase precision

Perspectives:
Formal Proof of Soundness of Combined Analyses

- Previous work demonstrated the need of formal proof of soundness
- This is particularly important for combined analyses

Goals:

- A Coq framework of certified C analyzers sharing a unique semantics
- Mechanized formal proof of combined analyses
  - e.g. SANTE, STADY, E-ACSL2C, LTest...
  - First step: Ph.D. work of Dara Ly in progress for E-ACSL2C
- Use a CompCert semantics of C

Some of these objectives are part of the ANR CERTICAT project proposal

Perspectives:
Support for advanced test coverage criteria

- Previous work demonstrated the interest of generic specification mechanisms of coverage criteria (labels, hyperlabels...)
- However, hyperlabels are not yet fully supported

Goals:

- Extended support for hyperlabels in LTest (for test generation, detection of infeasible objectives, test assessment)
- Investigate extensions of HTOL to support yet unformalized industrially relevant criteria, and further push their industrial applications
- Study the usage of coverage criteria in a continuous development cycle

A 3-year international ANR-FNR (France-Luxembourg) grant of 760,000 € for SATOCROSS project was allocated to support this work direction