The Mopsa static analysis platform, and our quest to ease implementation & maintenance

Raphaël Monat – SyCoMoRES team, Lille

`rmonat.fr`

Dagstuhl #25242
11 June 2025

Inria

Introduction

whoami

Research Scientist at Inria Lille.

whoami

Research Scientist at Inria Lille.

Research Interests

whoami

Research Scientist at Inria Lille.

Research Interests

- Static analysis: C, Python, multi-language paradigms

whoami

Research Scientist at Inria Lille.

Research Interests

- Static analysis: C, Python, multi-language paradigms
- Formal methods for public administrations

A Program Analysis Trichotomy

Sound	All errors in program reported by analyzer

A Program Analysis Trichotomy

All errors reported by analyzer are replicable in program

Complete

Sound

All errors in program reported by analyzer

A Program Analysis Trichotomy

Guaranteed Termination

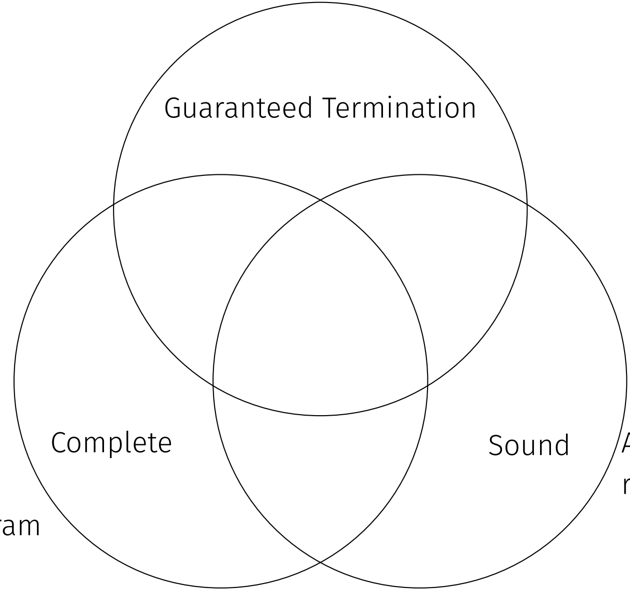
All errors reported by analyzer are replicable in program

Complete

Sound

All errors in program reported by analyzer

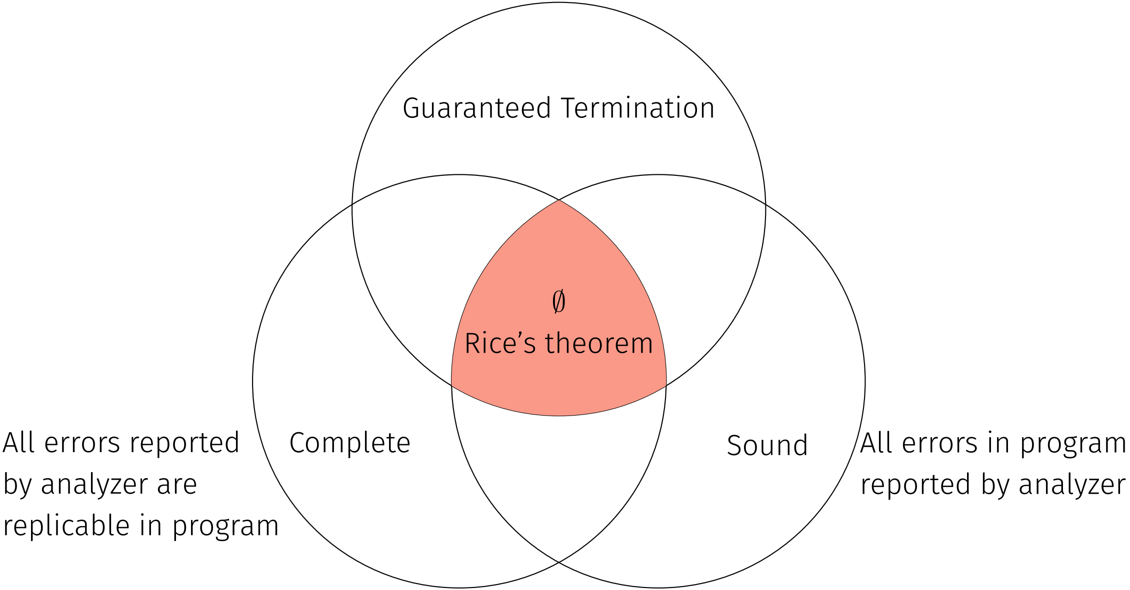
A Program Analysis Trichotomy



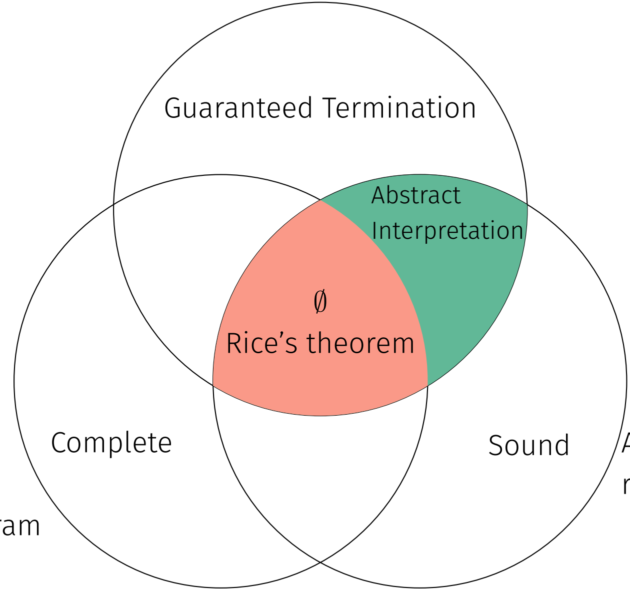
All errors reported by analyzer are replicable in program

All errors in program reported by analyzer

A Program Analysis Trichotomy



A Program Analysis Trichotomy



All errors reported by analyzer are replicable in program

All errors in program reported by analyzer

Motivation

Academic research around static analysis

Academic research around static analysis

Ideal analyzer

Academic research around static analysis

Ideal analyzer

- Sound, precise and scalable

Academic research around static analysis

Ideal analyzer

- Sound, precise and scalable
- Eases research:
  - Implementation
  - Experimental evaluation
  - Onboarding

Academic research around static analysis

Ideal analyzer

- Sound, precise and scalable
- Eases research:
  - Implementation
  - Experimental evaluation
  - Onboarding

Implementation hurdles

Academic research around static analysis

Ideal analyzer

- Sound, precise and scalable
- Eases research:
  - Implementation
  - Experimental evaluation
  - Onboarding

Implementation hurdles

- Debugging time-consuming

Academic research around static analysis

Ideal analyzer

- Sound, precise and scalable
- Eases research:
  - Implementation
  - Experimental evaluation
  - Onboarding

Implementation hurdles

- Debugging time-consuming
- Maintenance necessary to build upon previous work

Academic research around static analysis

Ideal analyzer

- Sound, precise and scalable
- Eases research:
  - Implementation
  - Experimental evaluation
  - Onboarding

Implementation hurdles

- Debugging time-consuming
- Maintenance necessary to build upon previous work

$\Longrightarrow$ Aiming for lowest possible implementation & maintenance costs

Outline

1. An overview of Mopsa
2. Avoiding regressions
3. Easing debugging
   - Developer-friendly interfaces
   - Testcase reduction
4. A plug-in system of analysis observers

An overview of Mopsa

Modular **O**pen **P**latform for **S**tatic **A**nalysis [Jou+19]

`gitlab.com/mopsa/mopsa-analyzer` or `opam install mopsa`

Started by ERC Consolidator Grant (2016-2021) of Antoine Miné (LIP6, SU)

Modular **O**pen **P**latform for **S**tatic **A**nalysis [Jou+19]

`gitlab.com/mopsa/mopsa-analyzer` or `opam install mopsa`

Started by ERC Consolidator Grant (2016-2021) of Antoine Miné (LIP6, SU)

Goals

- Explore new designs — Including multi-language support

Modular **O**pen **P**latform for **S**tatic **A**nalysis [Jou+19]

`gitlab.com/mopsa/mopsa-analyzer` or `opam install mopsa`

Started by ERC Consolidator Grant (2016-2021) of Antoine Miné (LIP6, SU)

Goals

- Explore new designs — Including multi-language support
- Ease development of relational static analyses
  - High expressivity $0 \leq i < \text{strlen}(s)$

Modular **O**pen **P**latform for **S**tatic **A**nalysis [Jou+19]

`gitlab.com/mopsa/mopsa-analyzer` or `opam install mopsa`

Started by ERC Consolidator Grant (2016-2021) of Antoine Miné (LIP6, SU)

Goals

- Explore new designs — Including multi-language support
- Ease development of relational static analyses
  - High expressivity $0 \leq i < \text{strlen}(s)$
- Open-source (LGPL)

Modular Open Platform for Static Analysis [Jou+19]

`gitlab.com/mopsa/mopsa-analyzer` or `opam install mopsa`

Started by ERC Consolidator Grant (2016-2021) of Antoine Miné (LIP6, SU)

Goals

- Explore new designs — Including multi-language support
- Ease development of relational static analyses
  - High expressivity $0 \leq i < \text{strlen}(s)$
- Open-source (LGPL)
- Can be used as an experimentation platform

Modular **O**pen **P**latform for **S**tatic **A**nalysis [Jou+19]

`gitlab.com/mopsa/mopsa-analyzer` or `opam install mopsa`

Started by ERC Consolidator Grant (2016-2021) of Antoine Miné (LIP6, SU)

Goals

- Explore new designs — Including multi-language support
- Ease development of relational static analyses
  High expressivity $0 \leq i < \text{strlen}(s)$
- Open-source (LGPL)
- Can be used as an experimentation platform

Currently, fully context-sensitive analyses

Contributors (2018–2025, chronological arrival order)

- A. Miné
- A. Ouadjaout
- M. Journault
- A. Fromherz
- D. Delmas
- R. Monat
- G. Bau
- F. Parolini
- M. Milanese
- M. Valnet
- J. Boillot

Contributors (2018–2025, chronological arrival order)

- **A. Miné**
- **A. Ouadjaout**
- M. Journault
- A. Fromherz
- D. Delmas
- **R. Monat**
- G. Bau
- F. Parolini
- M. Milanese
- M. Valnet
- J. Boillot

Maintainers in bold.

Mopsa design

Analysis = composition of abstract domains

Analysis = composition of abstract domains

unified domain signature $\Longrightarrow$ iterators are abstract domains

Mopsa design

Analysis = composition of abstract domains

unified domain signature $\implies$ iterators are abstract domains

- flexible architecture suitable for various programming paradigms

Analysis = composition of abstract domains

unified domain signature $\Longrightarrow$ iterators are abstract domains

- flexible architecture suitable for various programming paradigms
- separation of concerns

Analysis = composition of abstract domains

unified domain signature $\Longrightarrow$ iterators are abstract domains

- flexible architecture suitable for various programming paradigms
- separation of concerns
- allows reuse of domains across languages

Analysis = composition of abstract domains

unified domain signature $\Longrightarrow$ iterators are abstract domains

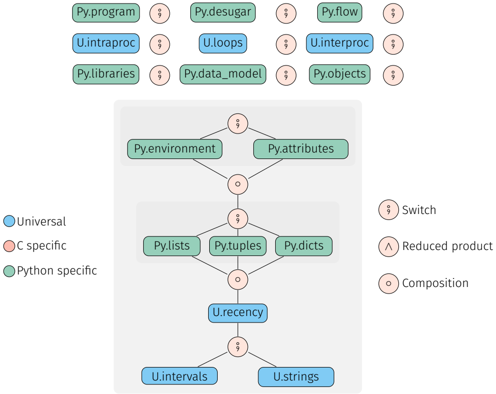
- flexible architecture suitable for various programming paradigms
- separation of concerns
- allows reuse of domains across languages
- defined as json files in `share/mopsa/configs`

Mopsa design

Analysis = composition of abstract domains

unified domain signature $\Longrightarrow$ iterators are abstract domains

- flexible architecture suitable for various programming paradigms
- separation of concerns
- allows reuse of domains across languages
- defined as json files in `share/mopsa/configs`



Summary of analyses

Languages

C [JMO18; OM20], Python [MOM20a; MOM20b]

Summary of analyses

Languages

C [JMO18; OM20], Python [MOM20a; MOM20b] Multilanguage Python+C [MOM21]

Summary of analyses

Languages

C [JMO18; OM20], Python [MOM20a; MOM20b] Multilanguage Python+C [MOM21]

WIP: Michelson [Bau+22], OCaml [VMM23; VMM25], Catala (date arithmetic [MFM24])...

Summary of analyses

Languages

C [JMO18; OM20], Python [MOM20a; MOM20b] Multilanguage Python+C [MOM21]

WIP: Michelson [Bau+22], OCaml [VMM23; VMM25], Catala (date arithmetic [MFM24])...

Properties

Summary of analyses

Languages

C [JMO18; OM20], Python [MOM20a; MOM20b] Multilanguage Python+C [MOM21]

WIP: Michelson [Bau+22], OCaml [VMM23; VMM25], Catala (date arithmetic [MFM24])...

Properties

- Absence of RTEs

Summary of analyses

Languages

C [JMO18; OM20], Python [MOM20a; MOM20b] Multilanguage Python+C [MOM21]

WIP: Michelson [Bau+22], OCaml [VMM23; VMM25], Catala (date arithmetic [MFM24])...

Properties

- **Absence of RTEs**
- Patch analysis [DM19]

Summary of analyses

Languages

C [JMO18; OM20], Python [MOM20a; MOM20b] Multilanguage Python+C [MOM21]

WIP: Michelson [Bau+22], OCaml [VMM23; VMM25], Catala (date arithmetic [MFM24])...

Properties

- **Absence of RTEs**
- Patch analysis [DM19]
- Endianness portability [DOM21]

Summary of analyses

Languages

C [JMO18; OM20], Python [MOM20a; MOM20b] Multilanguage Python+C [MOM21]

WIP: Michelson [Bau+22], OCaml [VMM23; VMM25], Catala (date arithmetic [MFM24])...

Properties

- **Absence of RTEs**
- Patch analysis [DM19]
- Endianness portability [DOM21]
- Non-exploitability [PM24]

Summary of analyses

Languages

C [JMO18; OM20], Python [MOM20a; MOM20b] Multilanguage Python+C [MOM21]

WIP: Michelson [Bau+22], OCaml [VMM23; VMM25], Catala (date arithmetic [MFM24])...

Properties

- **Absence of RTEs**
- Patch analysis [DM19]
- Endianness portability [DOM21]
- Non-exploitability [PM24]
- Sufficient precondition inference [MM24a; MM24b]

Software Verification Competition [Mon+24]

- Tools have to

Software Verification Competition [Mon+24]

- Tools have to
  - Decide whether a program is correct (large penalties if wrong)

Software Verification Competition [Mon+24]

- Tools have to
  - Decide whether a program is correct (large penalties if wrong)
  - Within limited machine resources (15 minutes CPU time, 8GB RAM)

Software Verification Competition [Mon+24]

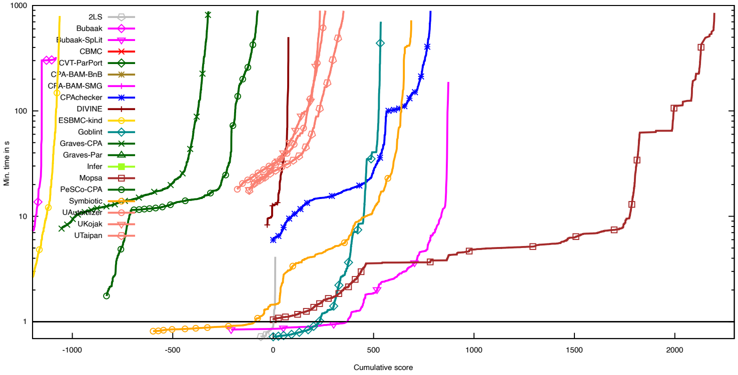
- Tools have to
  - Decide whether a program is correct (large penalties if wrong)
  - Within limited machine resources (15 minutes CPU time, 8GB RAM)
- Corpus of $\simeq$ 23,000 C benchmarks, now acts as a reference

Software Verification Competition [Mon+24]

- Tools have to
  - Decide whether a program is correct (large penalties if wrong)
  - Within limited machine resources (15 minutes CPU time, 8GB RAM)
- Corpus of $\simeq$ 23,000 C benchmarks, now acts as a reference
- For our second participation, Mopsa won the “Software Systems” track!

Software Verification Competition [Mon+24]

- Tools have to
  - Decide whether a program is correct (large penalties if wrong)
  - Within limited machine resources (15 minutes CPU time, 8GB RAM)
- Corpus of $\simeq$ 23,000 C benchmarks, now acts as a reference
- For our second participation, Mopsa won the “Software Systems” track!



Avoiding regressions

Detour: providing transparent analysis results

Raising the bar in static analyzer transparency

```
$ static-analysis-tool file
```

```
$ static-analysis-tool file
...
```

```
$ static-analysis-tool file
...
No errors found
```

Raising the bar in static analyzer transparency

```
$ static-analysis-tool file
...
No errors found
```

What has been checked? What has not?

Mopsa's approach to being transparent – at a high level

if $a^\# \not\sqsubseteq p^\#$ **then**
  add_alarm $a^\#$ $p^\#$

Mopsa's approach to being transparent – at a high level

$$\texttt{if}\ a^\# \not\sqsubseteq p^\#\ \texttt{then}\ \ \texttt{add\_alarm}\ a^\#\ p^\# \quad\rightsquigarrow\quad \texttt{if}\ a^\# \not\sqsubseteq p^\#\ \texttt{then}\ \ \texttt{add\_alarm}\ a^\#\ p^\#\ \ \texttt{else}\ \ \texttt{add\_safe\_check}\ p^\#$$

Mopsa’s approach to being transparent – example

Mopsa’s approach to being transparent

- Reporting status of all proofs / checks in every analyzed context

Mopsa's approach to being transparent – example

Mopsa's approach to being transparent

- Reporting status of all proofs / checks in every analyzed context
- Quantitative precision measure

$$\text{Selectivity} = \frac{\#\text{checks proved safe}}{\#\text{checks}}$$

Mopsa's approach to being transparent – example

Mopsa's approach to being transparent

- Reporting status of all proofs / checks in every analyzed context
- Quantitative precision measure

$$\text{Selectivity} = \frac{\#\text{checks proved safe}}{\#\text{checks}}$$

```c
int main() {
  int n = _mopsa_rand_s32();
  int y = -1;
  for(int x = 0; x < n; x++)
    y++;
}
```

Mopsa's approach to being transparent

- Reporting status of all proofs / checks in every analyzed context
- Quantitative precision measure

$$\text{Selectivity} = \frac{\#\text{checks proved safe}}{\#\text{checks}}$$

```c
int main() {
  int n = _mopsa_rand_s32();
  int y = -1;
  for(int x = 0; x < n; x++)
    y++;
}
```

Stmt
x++
y++
Selectivity

Mopsa's approach to being transparent

- Reporting status of all proofs / checks in every analyzed context
- Quantitative precision measure

$$\text{Selectivity} = \frac{\text{\#checks proved safe}}{\text{\#checks}}$$

```c
int main() {
  int n = _mopsa_rand_s32();
  int y = -1;
  for(int x = 0; x < n; x++)
    y++;
}
```

Stmt	Itv
x++	Safe
y++	Alarm
Selectivity	50%

Mopsa's approach to being transparent – example

Mopsa's approach to being transparent

- Reporting status of all proofs / checks in every analyzed context
- Quantitative precision measure

$$\text{Selectivity} = \frac{\#\text{checks proved safe}}{\#\text{checks}}$$

```c
int main() {
  int n = _mopsa_rand_s32();
  int y = -1;
  for(int x = 0; x < n; x++)
    y++;
}
```

Stmt	Itv	Poly
x++	Safe	Safe
y++	Alarm	Safe
Selectivity	50%	100%

Mopsa’s approach to being transparent – output

Benefits of the approach

Mopsa's approach to being transparent – output

Benefits of the approach

- Easy to implement

Mopsa's approach to being transparent – output

Benefits of the approach

- Easy to implement
- "2,756 alarms" $\rightsquigarrow$ 87% checks proved correct – "selectivity"

Mopsa's approach to being transparent – output

Benefits of the approach

- Easy to implement
- "2,756 alarms" $\rightsquigarrow$ 87% checks proved correct – "selectivity"
- ~~Program size~~ $\rightsquigarrow$ "expression complexity"

Mopsa's approach to being transparent – output

Benefits of the approach

- Easy to implement
- "2,756 alarms" ⇝ 87% checks proved correct – "selectivity"
- ~~Program size~~ ⇝ "expression complexity"

Analysis of coreutils `fmt`

```
Checks summary: 21247 total, ✔18491 safe, ✗129 errors, ⚠2627 warnings
  Stub condition: 690 total, ✔513 safe, ✗3 errors, ⚠174 warnings
  Invalid memory access: 8139 total, ✔7142 safe, ✗4 errors, ⚠993 warnings
  Division by zero: 499 total, ✔445 safe, ⚠54 warnings
  Integer overflow: 11581 total, ✔10177 safe, ⚠1404 warnings
  Invalid shift: 163 total, ✔163 safe
  Invalid pointer comparison: 37 total, ✗37 errors
  Invalid pointer subtraction: 85 total, ✗85 errors
  Insufficient variadic arguments: 1 total, ✔1 safe
  Insufficient format arguments: 26 total, ✔25 safe, ⚠1 warning
  Invalid type of format argument: 26 total, ✔25 safe, ⚠1 warning
```

Mopsa’s approach to being transparent – soundness assumptions

Soundness assumptions, through an example

```
extern int f(int *x)
```

Mopsa's approach to being transparent – soundness assumptions

Soundness assumptions, through an example

`extern int f(int *x)`, handling gradations

Mopsa's approach to being transparent – soundness assumptions

Soundness assumptions, through an example

`extern int f(int *x)`, handling gradations

1. Crash

Mopsa’s approach to being transparent – soundness assumptions

Soundness assumptions, through an example

`extern int f(int *x)`, handling gradations

1. Crash ✗

Mopsa's approach to being transparent – soundness assumptions

Soundness assumptions, through an example

`extern int f(int *x)`, handling gradations

1. Crash ✗
2. Ignore silently

Mopsa's approach to being transparent – soundness assumptions

Soundness assumptions, through an example

`extern int f(int *x)`, handling gradations

1. Crash ✗
2. Ignore silently ✗

Mopsa's approach to being transparent – soundness assumptions

Soundness assumptions, through an example

`extern int f(int *x)`, handling gradations

1. Crash ✗
2. Ignore silently ✗
3. Assume and report: f has no effect

Mopsa's approach to being transparent – soundness assumptions

Soundness assumptions, through an example

`extern int f(int *x)`, handling gradations

1. Crash ✗
2. Ignore silently ✗
3. Assume and report: f has no effect
4. Assume and report: f has any effect on its parameters

Mopsa's approach to being transparent – soundness assumptions

Soundness assumptions, through an example

`extern int f(int *x)`, handling gradations

1. Crash ✗
2. Ignore silently ✗
3. Assume and report: f has no effect
4. Assume and report: f has any effect on its parameters
5. Assume and report: f has any effect on its parameters and on globals

Mopsa's approach to being transparent – soundness assumptions

Soundness assumptions, through an example

`extern int f(int *x)`, handling gradations

1. Crash ✗
2. Ignore silently ✗
3. Assume and report: f has no effect
4. Assume and report: f has any effect on its parameters
5. Assume and report: f has any effect on its parameters and on globals

Related topic: soundiness paper [Liv+15]

Avoiding regressions

Leveraging analysis transparency

Avoiding regressions

$\Longrightarrow$ check for precision changes

Avoiding regressions

$\Longrightarrow$ check for precision changes

Benchmarks with precision oracles

- Know whether a given alarm should be raised
- Based on manual analysis, not scalable
- NIST's Juliet Benchmarks, SV-Comp labeling of tasks (coarse)
- Can provide absolute precision measure

Avoiding regressions

$\Longrightarrow$ **check for precision changes**

Benchmarks with precision oracles

- Know whether a given alarm should be raised
- Based on manual analysis, not scalable
- NIST's Juliet Benchmarks, SV-Comp labeling of tasks (coarse)
- Can provide absolute precision measure

Otherwise: relative precision measures, rely on our selectivity computation.

Comparing analysis reports

`mopsa-diff` script, used to compare:

- analysis report(s): either single output or set of outputs
- usecases: different configurations, different versions of Mopsa

Comparing analysis reports

`mopsa-diff` script, used to compare:

- analysis report(s): either single output or set of outputs
- usecases: different configurations, different versions of Mopsa

```
--- baseline/touch-many-symbolic-args-a4.json
+++ pplite/touch-many-symbolic-args-a4.json

- time: 589.0760
+ time: 675.1761

+ parse-datetime.y:1399.44-46: alarm: Invalid memory access
- parse-datetime.y:965.56-71: alarm: Invalid memory access
- parse-datetime.y:980.25-52: alarm: Invalid memory access
- parse-datetime.y:1003.23-50: alarm: Invalid memory access
- parse-datetime.y:921.56-71: alarm: Invalid memory access
- parse-datetime.c:1733.2-8: alarm: Invalid memory access
- parse-datetime.y:781.26-41: alarm: Invalid memory access
- parse-datetime.y:772.23-38: alarm: Invalid memory access
- parse-datetime.y:755.23-38: alarm: Invalid memory access
- parse-datetime.y:973.25-52: alarm: Invalid memory access
- parse-datetime.y:610.8-41: alarm: Invalid memory access
- parse-datetime.y:743.25-40: alarm: Invalid memory access
```

Comparing analysis reports

`mopsa-diff` script, used to compare:

- analysis report(s): either single output or set of outputs
- usecases: different configurations, different versions of Mopsa

```
--- baseline/touch-many-symbolic-args-a4.json
+++ pplite/touch-many-symbolic-args-a4.json

- time: 589.0760
+ time: 675.1761

+ parse-datetime.y:1399.44-46: alarm: Invalid memory access
- parse-datetime.y:965.56-71: alarm: Invalid memory access
- parse-datetime.y:980.25-52: alarm: Invalid memory access
- parse-datetime.y:1003.23-50: alarm: Invalid memory access
- parse-datetime.y:921.56-71: alarm: Invalid memory access
- parse-datetime.c:1733.2-8: alarm: Invalid memory access
- parse-datetime.y:781.26-41: alarm: Invalid memory access
- parse-datetime.y:772.23-38: alarm: Invalid memory access
- parse-datetime.y:755.23-38: alarm: Invalid memory access
- parse-datetime.y:973.25-52: alarm: Invalid memory access
- parse-datetime.y:610.8-41: alarm: Invalid memory access
- parse-datetime.y:743.25-40: alarm: Invalid memory access
```

```
139 reports compared
avg. time change        +52.065s
avg. speedup                -36%
new alarms                     2
removed alarms                32
new assumptions                0
removed assumptions            0
new successes                  0
new failures                   0
```

Detecting breaking changes using continuous integration

- `mopsa-diff` to compare with previous results

Detecting breaking changes using continuous integration

- `mopsa-diff` to compare with previous results
- Reusing all benchmarks from our experimental evaluations

CI, tests & benchmarks

Detecting breaking changes using continuous integration

- `mopsa-diff` to compare with previous results
- Reusing all benchmarks from our experimental evaluations

Benchmark selection

Our benchmarks are

CI, tests & benchmarks

Detecting breaking changes using continuous integration

- `mopsa-diff` to compare with previous results
- Reusing all benchmarks from our experimental evaluations

Benchmark selection

Our benchmarks are

- third-party real code

Detecting breaking changes using continuous integration

- `mopsa-diff` to compare with previous results
- Reusing all benchmarks from our experimental evaluations

Benchmark selection

Our benchmarks are

- third-party real code
- open-source – for the sake of reproducible science

CI, tests & benchmarks

Detecting breaking changes using continuous integration

- `mopsa-diff` to compare with previous results
- Reusing all benchmarks from our experimental evaluations

Benchmark selection

Our benchmarks are

- third-party real code
- open-source – for the sake of reproducible science
- unmodified*

Detecting breaking changes using continuous integration

- `mopsa-diff` to compare with previous results
- Reusing all benchmarks from our experimental evaluations

Benchmark selection

Our benchmarks are

- third-party real code
- open-source – for the sake of reproducible science
- unmodified*
  - Underscores practicality of our approach

CI, tests & benchmarks

Detecting breaking changes using continuous integration

- `mopsa-diff` to compare with previous results
- Reusing all benchmarks from our experimental evaluations

Benchmark selection

Our benchmarks are

- third-party real code
- open-source – for the sake of reproducible science
- unmodified*
  - Underscores practicality of our approach
  - * stubs can be added in marginal cases

Easing debugging

Developer-friendly interfaces

Where static analyzers usually start from

- Analysis output Too coarse

Where static analyzers usually start from

- Analysis output — Too coarse
- Printing abstract state using builtins — Not interactive

Where static analyzers usually start from

▶ Analysis output	Too coarse
▶ Printing abstract state using builtins	Not interactive
▶ Interpretation trace	Can be dozens of gigabytes of text

```
+ S [| set_program_name(argv[0]); |]
| | | + S [| add(argv0)
| | | |       argv0 = argv[0]; |]
| | | | + S [| add(argv0) |]
| | | | | + S [| add(argv0) |] in below(c.iterators.intraproc)
| | | | | | + S [| add(argv0) |] in C/Scalar
| | | | | | | + S [| add(offset⦃argv0⦄) |] in Universal
| | | | | | | o S [| add(offset⦃argv0⦄) |] in Universal done [0.0001s, 1 case]
| | | | | | o S [| add(argv0) |] in C/Scalar done [0.0001s, 1 case]
| | | | | | + S [| add(argv0) |] in below(c.memory.lowlevel.cells)
| | | | | | | + S [| add(offset⦃argv0⦄) |] in Universal
| | | | | | | o S [| add(offset⦃argv0⦄) |] in Universal done [0.0001s, 1 case]
| | | | | | o S [| add(argv0) |] in below(c.memory.lowlevel.cells) done [0.0001s, 1 case]
| | | | | o S [| add(argv0) |] in below(c.iterators.intraproc) done [0.0001s, 1 case]
| | | | o S [| add(argv0) |] done [0.0002s, 1 case]
| | | | + S [| argv0 = argv[0]; |]
| | | | | + S [| argv0 = (signed char *) @argv{0}:ptr; |] in below(c.iterators.intraproc)
| | | | | | + S [| argv0 = (signed char *) @argv{0}:ptr; |] in C/Scalar
| | | | | | | + S [| offset⦃argv0⦄ = (offset⦃@argv{0}:ptr⦄ + 0); |] in Universal
| | | | | | | | + S [| offset⦃argv0⦄ = (offset⦃@argv{0}:ptr⦄ + 0); |] in below(universal.iterators.intraproc)
```

An interactive engine acting as abstract debugger

GDB-like interface to the abstract interpretation of the program

An interactive engine acting as abstract debugger

GDB-like interface to the abstract interpretation of the program

Demo!

An interactive engine acting as abstract debugger

GDB-like interface to the abstract interpretation of the program

Demo!

- Breakpoints

An interactive engine acting as abstract debugger

GDB-like interface to the abstract interpretation of the program

Demo!

- Breakpoints
  - Program location

An interactive engine acting as abstract debugger

GDB-like interface to the abstract interpretation of the program

Demo!

- Breakpoints
  - Program location
  - Specific transfer function, analysis of subexpression

An interactive engine acting as abstract debugger

GDB-like interface to the abstract interpretation of the program

Demo!

- Breakpoints
  - Program location
  - Specific transfer function, analysis of subexpression
  - Alarm: jumping back to statement generating first alarm

An interactive engine acting as abstract debugger

GDB-like interface to the abstract interpretation of the program

Demo!

- Breakpoints
  - Program location
  - Specific transfer function, analysis of subexpression
  - Alarm: jumping back to statement generating first alarm
- Navigation

An interactive engine acting as abstract debugger

GDB-like interface to the abstract interpretation of the program

Demo!

- Breakpoints
  - Program location
  - Specific transfer function, analysis of subexpression
  - Alarm: jumping back to statement generating first alarm
- Navigation
- Observation of the abstract state

An interactive engine acting as abstract debugger

GDB-like interface to the abstract interpretation of the program

Demo!

- Breakpoints
  - Program location
  - Specific transfer function, analysis of subexpression
  - Alarm: jumping back to statement generating first alarm
- Navigation
- Observation of the abstract state
  - Full state

An interactive engine acting as abstract debugger

GDB-like interface to the abstract interpretation of the program

Demo!

- Breakpoints
  - Program location
  - Specific transfer function, analysis of subexpression
  - Alarm: jumping back to statement generating first alarm
- Navigation
- Observation of the abstract state
  - Full state
  - Projection on specific variables

An interactive engine acting as abstract debugger

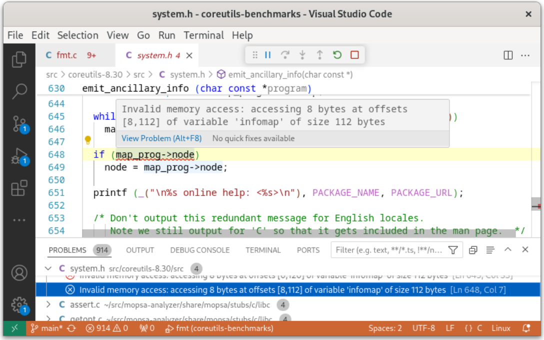
GDB-like interface to the abstract interpretation of the program

Demo!

- Breakpoints
  - Program location
  - Specific transfer function, analysis of subexpression
  - Alarm: jumping back to statement generating first alarm
- Navigation
- Observation of the abstract state
  - Full state
  - Projection on specific variables
- Some scripting capabilities

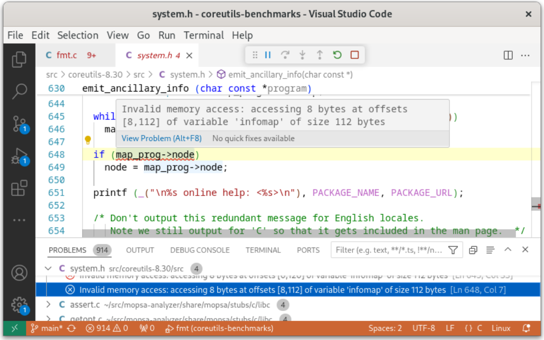
IDE support

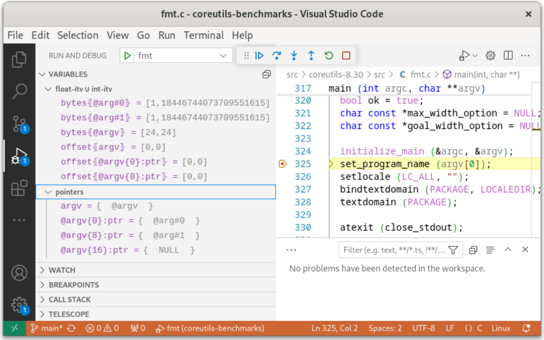
- Language Server Protocol for linters (report alarms)



IDE support

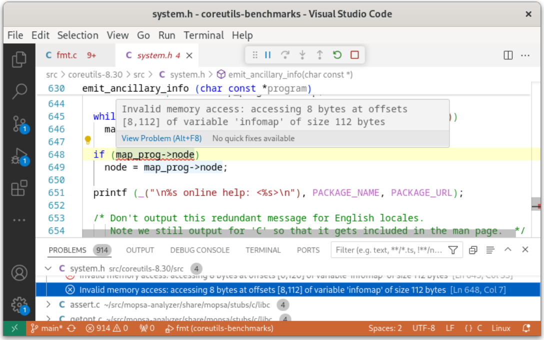
- Language Server Protocol for linters (report alarms)
- Debug Adapter Protocol providing interactive engine interface

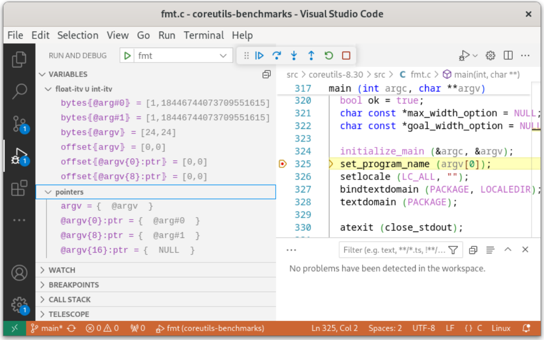




IDE support

- Language Server Protocol for linters (report alarms)
- Debug Adapter Protocol providing interactive engine interface
- Both protocols introduced by VSCode, supported by multiple IDEs





Easing debugging

Testcase reduction

Motivation

Testcase reduction

Motivation

- Static analyzers are complex piece of code and may contain bugs

Testcase reduction

Motivation

- Static analyzers are complex piece of code and may contain bugs
- In practice, some bugs will only be detected in large codebases

Testcase reduction

Motivation

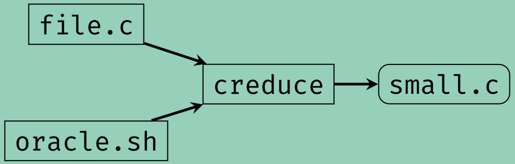
- Static analyzers are complex piece of code and may contain bugs
- In practice, some bugs will only be detected in large codebases
- Debugging extremely difficult: size of the program, analysis time

Testcase reduction

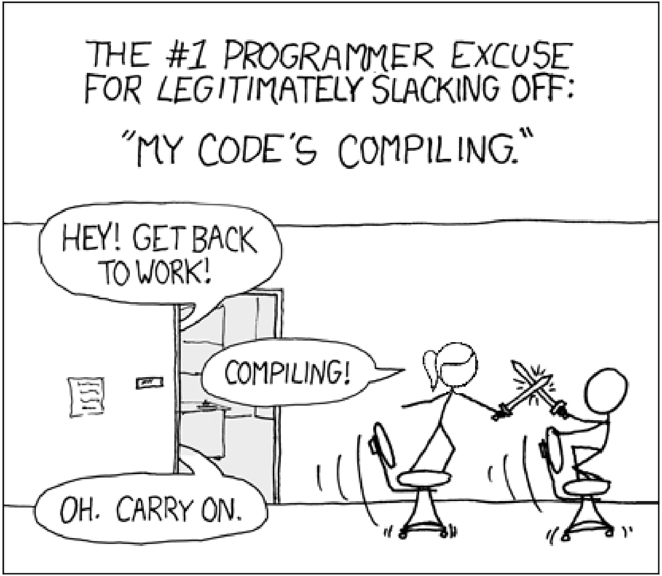
Motivation

- Static analyzers are complex piece of code and may contain bugs
- In practice, some bugs will only be detected in large codebases
- Debugging extremely difficult: size of the program, analysis time

Automated testcase reduction using `creduce` [Reg+12]



Testcase reduction – II



Testcase reduction – III

Internal errors debugging

- Highly helpful to significantly reduce debugging time of runtime errors (Apron mishandlings, raised exceptions, ...)
- Has been applied to coreutils programs, SV-Comp programs of 10,000+ LoC

Testcase reduction – III

Internal errors debugging

- Highly helpful to significantly reduce debugging time of runtime errors (Apron mishandlings, raised exceptions, ...)
- Has been applied to coreutils programs, SV-Comp programs of 10,000+ LoC

Reference	Origin	Original LoC	Reduced LoC	Reduction
Issue 76	SV-Comp	28,737	18	99.94%
Issue 81	SV-Comp	15,627	8	99.95%
Issue 134	SV-Comp	17,411	10	99.94%
Issue 135	SV-Comp	7,016	12	99.83%
M.R. 130	`coreutils`	77,981	20	99.97%
M.R. 145	`coreutils`	77,427	19	99.98%

Testcase reduction – IV

Differential-configuration debugging

```
$ mopsa-c -config=confA.json file.c
Alarm: assertion failure
$ mopsa-c -config=confB.json file.c
No alarm
```

Has been used to simplify cases in externally reported soundness issues

Handling multi-file projects

`creduce` reduces a specific file

One mitigation: generate a pre-processed, standalone file

Painful operation on large projects such as **`coreutils`**

Handling multi-file projects

`creduce` reduces a specific file

One mitigation: generate a pre-processed, standalone file

Painful operation on large projects such as `coreutils`

Mopsa supports multi-file C projects

- `mopsa-build`

Handling multi-file projects

`creduce` reduces a specific file

One mitigation: generate a pre-processed, standalone file

Painful operation on large projects such as `coreutils`

Mopsa supports multi-file C projects

- `mopsa-build`
  - Records compiler/linker calls and their options

Handling multi-file projects

`creduce` reduces a specific file

One mitigation: generate a pre-processed, standalone file

Painful operation on large projects such as `coreutils`

Mopsa supports multi-file C projects

- `mopsa-build`
  - Records compiler/linker calls and their options
  - Creates a compilation database

Handling multi-file projects

`creduce` reduces a specific file

One mitigation: generate a pre-processed, standalone file

Painful operation on large projects such as `coreutils`

Mopsa supports multi-file C projects

- `mopsa-build`
  - Records compiler/linker calls and their options
  - Creates a compilation database

$\rightsquigarrow$ `mopsa-build make` drop-in replacement for `make`

Handling multi-file projects

`creduce` reduces a specific file

One mitigation: generate a pre-processed, standalone file

Painful operation on large projects such as `coreutils`

Mopsa supports multi-file C projects

- `mopsa-build`
  - Records compiler/linker calls and their options
  - Creates a compilation database

  ⇝ `mopsa-build make` drop-in replacement for `make`
- `mopsa-c` leverages the compilation database

```
mopsa-c mopsa.db -make-target=fmt
```

Handling multi-file projects

`creduce` reduces a specific file

One mitigation: generate a pre-processed, standalone file

Painful operation on large projects such as `coreutils`

Mopsa supports multi-file C projects

- `mopsa-build`
  - Records compiler/linker calls and their options
  - Creates a compilation database

  ⇝ `mopsa-build make` drop-in replacement for `make`
- `mopsa-c` leverages the compilation database

```
mopsa-c mopsa.db -make-target=fmt
```

- Option to generate a single, preprocessed file

A plug-in system of analysis observers

Hooks: a plug-in system of analysis observers

Hooks	
Observe analyzer state	before/after any expression/statement analysis

Hooks: a plug-in system of analysis observers

Hooks

Observe analyzer state before/after any expression/statement analysis

Current hooks

- Logs: trace of interpretation performed by the analysis

Hooks: a plug-in system of analysis observers

Hooks

Observe analyzer state before/after any expression/statement analysis

Current hooks

- Logs: trace of interpretation performed by the analysis
- Thresholds for widening

Hooks: a plug-in system of analysis observers

Hooks

Observe analyzer state before/after any expression/statement analysis

Current hooks

- Logs: trace of interpretation performed by the analysis
- Thresholds for widening
- Coverage

Hooks: a plug-in system of analysis observers

Hooks

Observe analyzer state before/after any expression/statement analysis

Current hooks

- Logs: trace of interpretation performed by the analysis
- Thresholds for widening
- Coverage
- Heuristic unsoundness/imprecision detection

Hooks: a plug-in system of analysis observers

Hooks

Observe analyzer state before/after any expression/statement analysis

Current hooks

- Logs: trace of interpretation performed by the analysis
- Thresholds for widening
- Coverage
- Heuristic unsoundness/imprecision detection
- Profiling

Hooks: a plug-in system of analysis observers

Hooks

Observe analyzer state before/after any expression/statement analysis

Current hooks

- Logs: trace of interpretation performed by the analysis
- Thresholds for widening
- Coverage
- Heuristic unsoundness/imprecision detection
- Profiling

Coverage hooks

Coverage

- Global metric for the analysis' results
- Good to detect issues in the instrumentation of the fully context-sensitive analysis

No symbolic argument

```
./src/coreutils-8.30/src/fmt.c:
    'main' 76% of 72 statements analyzed
    'set_prefix' 100% of 12 statements analyzed
    'same_para' 100% of 1 statement analyzed
    'get_line' 100% of 30 statements analyzed
    'fmt' 100% of 7 statements analyzed
    'base_cost' 100% of 16 statements analyzed
    'line_cost' 100% of 10 statements analyzed
    'get_prefix' 100% of 18 statements analyzed
```

Symbolic arguments

```
./src/coreutils-8.30/src/fmt.c:
    'main' 100% of 72 statements analyzed
```

Heuristic unsoundness/imprecision detection

Detection of unsound transfer functions

Bottom shouldn't appear after some statements (such as assignments)

Detection of imprecise analysis

Warns when top expressions are created

Simplifies the search for sources of large imprecision (esp. with rewritings)

Profiling

Standard profiling

Measures which parts of Mopsa are the most time-consuming

Profiling

Standard profiling

Measures which parts of Mopsa are the most time-consuming

Abstract profiling hook

Measures which parts of the analyzed program are the most time-consuming

- Loop-level profiling
- Function-level profiling

Profiling

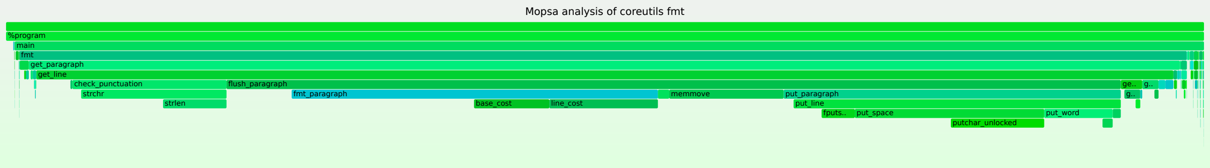
Standard profiling

Measures which parts of Mopsa are the most time-consuming

Abstract profiling hook

Measures which parts of the analyzed program are the most time-consuming

- Loop-level profiling
- Function-level profiling



Profiling – II

Apron vs PPLite on Coreutils touch

- PPLite is 14% slower but more precise (11 alarms removed). Why?

Profiling – II

Apron vs PPLite on Coreutils touch

- PPLite is 14% slower but more precise (11 alarms removed). Why?
- Suggestion from Enea Zaffanella: widening operator.

Profiling – II

Apron vs PPLite on Coreutils touch

- PPLite is 14% slower but more precise (11 alarms removed). Why?
- Suggestion from Enea Zaffanella: widening operator.
- Easy to confirm intuition!

Profiling – II

Apron vs PPLite on Coreutils touch

- PPLite is 14% slower but more precise (11 alarms removed). Why?
- Suggestion from Enea Zaffanella: widening operator.
- Easy to confirm intuition!

```
Loops profiling:
  ./src/coreutils-8.30/lib/argmatch.c:95.2-118.5: 3 times, [-3.00-] {+4.00+} avg. iterations [-(3, 3, 3)-] {+(4, 4, 4)+}
  ./src/coreutils-8.30/lib/posixtm.c:130.2-132.18: 12 times, [-2.00-] {+3.00+}
    avg. iterations [-(2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2)-] {+(3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3)+}
  ./src/coreutils-8.30/lib/posixtm.c:135.2-136.52: 12 times, [-2.00-] {+3.00+}
    avg. iterations [-(2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2)-] {+(3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3)+}
  ./src/coreutils-8.30/src/system.h:645.2-646.14: 3 times, [-2.00-] {+3.00+}
    avg. iterations [-(2, 2, 2)-] {+(3, 3, 3)+}
  parse-datetime.c:2636.2-2660.5: 16 times, [-2.00-] {+2.50+}
    avg.iterations [-(2, 2,-] {+(3, 3,+} 3, 1, [-2, 2,-] {+3, 3,+} 3, 1, [-2, 2,-] {+3, 3,+} 3, 1, [-2, 2,-] {+3, 3,+} 3, 1
  parse-datetime.c:2711.2-2716.5: 16 times, [-1.50-] {+1.75+}
    avg.iterations [-(1,-] {+(2,+} 2, 2, 1, [-1,-] 2, 2, [-1,-] {+2,+} 1, 2, 2, {+2,+} 1, [-1,-] {+2,+} 2, 2, 1)
  parse-datetime.y:1298.2-1300.15: 40 times, [-2.00-] {+3.00+}
    avg.iterations [-(2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
                   {+(3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3,
  parse-datetime.y:1304.2-1306.15: 40 times, [-2.00-] {+3.00+}
    avg.iterations [-(2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
                   {+(3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3,
```

Conclusion

Related work

Lots of folklore

Related work

Lots of folklore

- First work, applying and combining S.E. techniques for TAJS [AMN17]

Related work

Lots of folklore

- First work, applying and combining S.E. techniques for TAJS [AMN17]
- Frama-C & Goblint: flamegraphs, testcase reduction

Related work

Lots of folklore

- First work, applying and combining S.E. techniques for TAJS [AMN17]
- Frama-C & Goblint: flamegraphs, testcase reduction
- Symbolic profiling [BT18]

Related work

Lots of folklore

- First work, applying and combining S.E. techniques for TAJS [AMN17]
- Frama-C & Goblint: flamegraphs, testcase reduction
- Symbolic profiling [BT18]
- Leveraging LSP [LDB19]

Related work

Lots of folklore

- First work, applying and combining S.E. techniques for TAJS [AMN17]
- Frama-C & Goblint: flamegraphs, testcase reduction
- Symbolic profiling [BT18]
- Leveraging LSP [LDB19]
- Testing the soundness and precision of static analyzers [KCW19; TLR20; MVR23; Kai+24; Fle+24]

Related work

Lots of folklore

- First work, applying and combining S.E. techniques for TAJS [AMN17]
- Frama-C & Goblint: flamegraphs, testcase reduction
- Symbolic profiling [BT18]
- Leveraging LSP [LDB19]
- Testing the soundness and precision of static analyzers [KCW19; TLR20; MVR23; Kai+24; Fle+24]
- Debugging:

Related work

Lots of folklore

- First work, applying and combining S.E. techniques for TAJS [AMN17]
- Frama-C & Goblint: flamegraphs, testcase reduction
- Symbolic profiling [BT18]
- Leveraging LSP [LDB19]
- Testing the soundness and precision of static analyzers [KCW19; TLR20; MVR23; Kai+24; Fle+24]
- Debugging:
  - Mixing concrete+abstract [Do+18; MVR23]

Related work

Lots of folklore

- First work, applying and combining S.E. techniques for TAJS [AMN17]
- Frama-C & Goblint: flamegraphs, testcase reduction
- Symbolic profiling [BT18]
- Leveraging LSP [LDB19]
- Testing the soundness and precision of static analyzers [KCW19; TLR20; MVR23; Kai+24; Fle+24]
- Debugging:
  - Mixing concrete+abstract [Do+18; MVR23]
  - Sound abstract debugger in Goblint [Hol+24a; Hol+24b]

Conclusion

Our current approach to ease Mopsa's maintenance

- Non-regression testing of soundness & precision. CI on real-world software.

Conclusion

Our current approach to ease Mopsa's maintenance

- Non-regression testing of soundness & precision. CI on real-world software.
- Combination of existing techniques and new tools to debug & profile Mopsa

Conclusion

Our current approach to ease Mopsa's maintenance

- Non-regression testing of soundness & precision. CI on real-world software.
- Combination of existing techniques and new tools to debug & profile Mopsa
  "std. tools on the concrete execution of the *abstract interpreter*"

Conclusion

Our current approach to ease Mopsa's maintenance

- Non-regression testing of soundness & precision. CI on real-world software.
- Combination of existing techniques and new tools to debug & profile Mopsa
  "std. tools on the concrete execution of the *abstract interpreter*"
  $\rightsquigarrow$ "new tools on abstract execution of *target program*"

Conclusion

Our current approach to ease Mopsa's maintenance

- Non-regression testing of soundness & precision. CI on real-world software.
- Combination of existing techniques and new tools to debug & profile Mopsa
  "std. tools on the concrete execution of the *abstract interpreter*"
  $\rightsquigarrow$ "new tools on abstract execution of *target program*"

Future directions

Conclusion

Our current approach to ease Mopsa's maintenance

- Non-regression testing of soundness & precision. CI on real-world software.
- Combination of existing techniques and new tools to debug & profile Mopsa
  "std. tools on the concrete execution of the *abstract interpreter*"
  $\rightsquigarrow$ "new tools on abstract execution of *target program*"

Future directions

- More debugging tools?

Conclusion

Our current approach to ease Mopsa's maintenance

- Non-regression testing of soundness & precision. CI on real-world software.
- Combination of existing techniques and new tools to debug & profile Mopsa
  "std. tools on the concrete execution of the *abstract interpreter*"
  $\rightsquigarrow$ "new tools on abstract execution of *target program*"

Future directions

- More debugging tools?
- Exponential number of configurations

Conclusion

Our current approach to ease Mopsa's maintenance

- Non-regression testing of soundness & precision. CI on real-world software.
- Combination of existing techniques and new tools to debug & profile Mopsa
  "std. tools on the concrete execution of the *abstract interpreter*"
  $\rightsquigarrow$ "new tools on abstract execution of *target program*"

Future directions

- More debugging tools?
- Exponential number of configurations
- Testing non-leaf abstract domains? Apron-compatible abstract domains?

Conclusion

Our current approach to ease Mopsa's maintenance

- Non-regression testing of soundness & precision. CI on real-world software.
- Combination of existing techniques and new tools to debug & profile Mopsa
  "std. tools on the concrete execution of the *abstract interpreter*"
  $\rightsquigarrow$ "new tools on abstract execution of *target program*"

Future directions

- More debugging tools?
- Exponential number of configurations
- Testing non-leaf abstract domains? Apron-compatible abstract domains?
- Larger usability improvements?

References – I

[AMN17] Esben Sparre Andreasen, Anders Møller, and Benjamin Barslev Nielsen. **“Systematic approaches for increasing soundness and precision of static analyzers”.** In: ed. by Karim Ali and Cristina Cifuentes. ACM, 2017, pp. 31–36. DOI: 10.1145/3088515.3088521.

[Bau+22] Guillaume Bau et al. **“Abstract interpretation of Michelson smart-contracts”.** In: ed. by Laure Gonnord and Laura Titolo. ACM, 2022, pp. 36–43. DOI: 10.1145/3520313.3534660.

[BT18] James Bornholt and Emina Torlak. **“Finding code that explodes under symbolic evaluation”.** In: Proceedings of the ACM on Programming Languages OOPSLA (2018), pp. 1–26.

References – II

[DM19] David Delmas and Antoine Miné. **"Analysis of Software Patches Using Numerical Abstract Interpretation".** In: ed. by Bor-Yuh Evan Chang. Lecture Notes in Computer Science. Springer, 2019, pp. 225–246. DOI: `10.1007/978-3-030-32304-2_12`.

[Do+18] Lisa Nguyen Quang Do et al. **"Debugging static analysis".** In: IEEE Transactions on Software Engineering 7 (2018), pp. 697–709.

[DOM21] David Delmas, Abdelraouf Ouadjaout, and Antoine Miné. **"Static Analysis of Endian Portability by Abstract Interpretation".** In: Lecture Notes in Computer Science. Springer, 2021, pp. 102–123.

References – III

[Fle+24] Markus Fleischmann et al. "Constraint-Based Test Oracles for Program Analyzers". In: ed. by Vladimir Filkov, Baishakhi Ray, and Minghui Zhou. ACM, 2024, pp. 344–355. DOI: 10.1145/3691620.3695035.

[Hol+24a] Karoliine Holter et al. "Abstract Debuggers: Exploring Program Behaviors using Static Analysis Results". In: Onward! '24. Pasadena, CA, USA: Association for Computing Machinery, 2024, pp. 130–146. DOI: 10.1145/3689492.3690053.

[Hol+24b] Karoliine Holter et al. "Abstract Debugging with GobPie". In: ed. by Elisa Gonzalez Boix and Christophe Scholliers. ACM, 2024, pp. 32–33. DOI: 10.1145/3678720.3685320.

References – IV

[JMO18] Matthieu Journault, Antoine Miné, and Abdelraouf Ouadjaout. "Modular Static Analysis of String Manipulations in C Programs". In: ed. by Andreas Podelski. Lecture Notes in Computer Science. Springer, 2018, pp. 243–262. DOI: 10.1007/978-3-319-99725-4_16.

[Jou+19] M. Journault et al. "Combinations of reusable abstract domains for a multilingual static analyzer". In: New York, USA, July 2019, pp. 1–17.

[Kai+24] David Kaindlstorfer et al. "Interrogation Testing of Program Analyzers for Soundness and Precision Issues". In: ed. by Vladimir Filkov, Baishakhi Ray, and Minghui Zhou. ACM, 2024, pp. 319–330. DOI: 10.1145/3691620.3695034.

References – V

[KCW19] Christian Klinger, Maria Christakis, and Valentin Wüstholz. **“Differentially testing soundness and precision of program analyzers”.** In: ed. by Dongmei Zhang and Anders Møller. ACM, 2019, pp. 239–250. DOI: `10.1145/3293882.3330553`.

[LDB19] Linghui Luo, Julian Dolby, and Eric Bodden. **“MagpieBridge: A General Approach to Integrating Static Analyses into IDEs and Editors (Tool Insights Paper)”.** In: ed. by Alastair F. Donaldson. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019, 21:1–21:25. DOI: `10.4230/LIPICS.ECOOP.2019.21`.

[Liv+15] Benjamin Livshits et al. **“In defense of soundiness: a manifesto”.** In: Commun. ACM 2 (2015), pp. 44–46. DOI: `10.1145/2644805`.

References – VI

[MFM24] Raphaël Monat, Aymeric Fromherz, and Denis Merigoux. **“Formalizing Date Arithmetic and Statically Detecting Ambiguities for the Law”.** In: ed. by Stephanie Weirich. Lecture Notes in Computer Science. Springer, 2024, pp. 421–450. DOI: `10.1007/978-3-031-57267-8_16`.

[MM24a] Marco Milanese and Antoine Miné. **“Generation of Violation Witnesses by Under-Approximating Abstract Interpretation”.** In: ed. by Rayna Dimitrova, Ori Lahav, and Sebastian Wolff. Lecture Notes in Computer Science. Springer, 2024, pp. 50–73. DOI: `10.1007/978-3-031-50524-9_3`.

References – VII

[MM24b] Marco Milanese and Antoine Miné. **“Under-Approximating Memory Abstractions”.** In: ed. by Roberto Giacobazzi and Alessandra Gorla. Lecture Notes in Computer Science. Springer, 2024, pp. 300–326. DOI: 10.1007/978-3-031-74776-2_12.

[MOM20a] R. Monat, A. Ouadjaout, and A. Miné. **“Static Type Analysis by Abstract Interpretation of Python Programs”.** In: LIPIcs. 2020.

[MOM20b] R. Monat, A. Ouadjaout, and A. Miné. **“Value and allocation sensitivity in static Python analyses”.** In: ACM, 2020, pp. 8–13. DOI: 10.1145/3394451.3397205.

[MOM21] R. Monat, A. Ouadjaout, and A. Miné. **“A Multilanguage Static Analysis of Python Programs with Native C Extensions”.** In: 2021.

References – VIII

[Mon+24] Raphaël Monat et al. **"Mopsa-C: Improved Verification for C Programs, Simple Validation of Correctness Witnesses (Competition Contribution)".** In: Lecture Notes in Computer Science. Springer, 2024, pp. 387–392.

[MVR23] Mats Van Molle, Bram Vandenbogaerde, and Coen De Roover. **"Cross-Level Debugging for Static Analysers".** In: ed. by João Saraiva, Thomas Degueule, and Elizabeth Scott. ACM, 2023, pp. 138–148. DOI: 10.1145/3623476.3623512.

References – IX

[OM20] A. Ouadjaout and A. Miné. **“A Library Modeling Language for the Static Analysis of C Programs”.** In: ed. by David Pichardie and Mihaela Sighireanu. Lecture Notes in Computer Science. Springer, 2020, pp. 223–247. DOI: 10.1007/978-3-030-65474-0_11.

[PM24] Francesco Parolini and Antoine Miné. **“Sound Abstract Nonexploitability Analysis”.** In: Lecture Notes in Computer Science. Springer, 2024, pp. 314–337.

[Reg+12] John Regehr et al. **“Test-case reduction for C compiler bugs”.** In: ed. by Jan Vitek, Haibo Lin, and Frank Tip. ACM, 2012, pp. 335–346. DOI: 10.1145/2254064.2254104.

[TLR20] Jubi Taneja, Zhengyang Liu, and John Regehr. **“Testing static analyses for precision and soundness”.** In: ACM, 2020, pp. 81–93. DOI: 10.1145/3368826.3377927.

[VMM23] Milla Valnet, Raphaël Monat, and Antoine Miné. **“Analyse statique de valeurs par interprétation abstraite de programmes fonctionnels manipulant des types algébriques récursifs”.** In: ed. by Timothy Bourke and Delphine Demange. Praz-sur-Arly, France, Jan. 2023, pp. 211–242.

[VMM25] Milla Valnet, Raphaël Monat, and Antoine Miné. **“Compositional Static Value Analysis for Higher-Order Numerical Programs”.** In: ed. by Jonathan Aldrich and Alexandra Silva; Bergen, Norway: Dagstuhl Publishing, June 2025, p. 15. DOI: 10.4230/LIPIcs.ECOOP.2025.15.