Journal manuscript No.
(will be inserted by the editor)

Easing Maintenance of Academic Static Analyzers

Raphaël Monat · Abdelraouf Ouadjaout · Antoine Miné

Submitted: 17/07/2024

Abstract Academic research in static analysis produces software implementations. These implementations are time-consuming to develop and some need to be maintained in order to enable building further research upon the implementation. While necessary, these processes can be quickly challenging. This article documents the tools and techniques we have come up with to simplify the maintenance of Mopsa since 2017. Mopsa is a static analysis platform that aims at being sound. First, we describe an automated way to measure precision that does not require any baseline of true bugs obtained by manually inspecting the results. Further, it improves transparency of the analysis, and helps discovering regressions during continuous integration. Second, we have taken inspiration from standard tools observing the concrete execution of a program to design custom tools observing the abstract execution of the analyzed program itself, such as abstract debuggers and profilers. Finally, we report on some cases of automated testcase reduction.

Keywords Static Analysis · Abstract Interpretation · Software Engineering

1 Introduction

One of the products of academic research in static analysis is the implementation of tools, which are used to illustrate and evaluate approaches. During their lifetime, these tools have to be maintained – in particular to enable further research. This research can be performed by the same authors, new members of a same group or by other groups wanting to build upon this tool. While necessary, debugging and maintenance of static analyzers can quickly turn out to be time-consuming, as static analyzers perform highly technical reasoning. In addition, this maintenance is not the main purpose of academic jobs – and similarly, industrial developers will be looking to reduce their maintenance costs as much as possible. Due to its nature, the maintenance problem is common to all researchers in the community. Some practices are considered as folklore, which might explain why practices developed after years of experience are not systematically documented nor shared between groups. In this paper, we document the tools and techniques we use to simplify our maintenance of Mopsa, the static analysis platform we have been developing since 2017. We hope this article, following the precursor work of Andreasen et al. (2017), will have a twofold impact, by being useful to other researchers (and especially newcomers such as students), and by motivating other groups to share their practices.

Contributions.

- We describe a novel way of reporting analyses results, and measuring precision based on selectivity (Section 3). It is automatic, enhances transparency of the analysis, and takes into account the complexity of the expressions in the analyzed program (and not merely the program size). We can then leverage analyses results to detect soundness and precision regressions, thanks to a set of real-world, open-source benchmarks used in our continuous integration.
- In Section 4, we show how plug-in observers to the analysis can help in the development of coverage and profiling tools working on the abstract interpretation of the program, while standard tools provide different results by working on the concrete execution of the analyzer.

Univ. Lille, Inria, CNRS, Centrale Lille, UMR 9189 CRIStAL, F-59000 Lille, France ·
Independent researcher, Grenoble, France ·
LIP6, Sorbonne Université, F-75005, Paris, France

- We showcase an abstract debugger interface allowing interactive exploration of the abstract interpretation of the analyzed program, facilitating use and debugging (Section 5). This abstract debugger is available both as a command-line interface and in IDEs supporting the Debug Adapter Protocol.
- We report on our experience in applying automated testcase reduction tools (initially used on compilers) on static analyzers (Section 6). These tools ease the process of isolating, analyzing and fixing precision and soundness issues in the analyzer. Indeed, this process is essentially manual and time-consuming, and bugs are often initially identified on example runs too large to inspect manually. Collaboration between Mopsa and automated testcase reduction is a two-way street: Mopsa enables seamless testcase reduction for multi-file projects with complex commands such as `GNU coreutils`. This process is enabled thanks to the `mopsa-build` utility instrumenting the compilation process, which can then be leveraged to output a single preprocessed file usable by testcase reduction tools. The generated file corresponds to a kind of AST-level linking.

In particular, Sections 4 and 5 highlight a systematic connection between standard tools observing the *concrete* execution of the *abstract interpreter* and custom tools (abstract debuggers, profilers) we developed, which observe the *abstract* execution of the *analyzed* program itself.

2 An Overview of Mopsa

Mopsa (*Modular Open Platform for Static Analysis*) is a publicly-available (Miné et al. (2024)) and open-source framework for the development of static analyzers based on the theory of abstract interpretation by Cousot and Cousot (1977). Journault et al. (2019); Monat (2021) describe its flexible and modular architecture that makes it extensible in many aspects. Mopsa aims at simplifying the exploration of new ideas and the development of static analyzers, while providing mature implementations for selected mainstream languages. Mopsa participated in 2023 and 2024 to the Software-Verification Competition (SV-Comp), following contributions from Monat et al. (2023, 2024b). In 2024, Mopsa obtained the first place in the SoftwareSystems track of SV-Comp.

2.1 Languages

Mopsa supports the analysis of multiple languages, such as C (Journault et al. (2018); Ouadjaout and Miné (2020)), Python (Monat et al. (2020a,b)), combination of Python and C (Monat et al. (2021)) and Michelson (Bau et al. (2022)) languages. Unlike most multi-language static analyzers, Mopsa does not rely on translating programs to a fixed intermediate representation before starting the analysis. These intermediate representations are generally very low-level (e.g., three-address code, stack machine, small C subsets). While intermediate representations simplify the reuse of abstractions among different languages, they may suffer from information loss – such as high-level control-flow structures and data structures – during the initial syntactic translation. For example, LLVM forgets whether integer types are signed or unsigned, while transformation to 3-address code puts a strain on relational domains to maintain precision, as shown by Namjoshi and Pavlinovic (2018). In addition, a fixed intermediate representation may not support all kinds of languages and paradigms (e.g, dynamically typed, object oriented programming languages, functional languages), limiting the generality of the framework.

On the contrary, Mopsa has an extensible AST (*Abstract Syntax Tree*) that represents the union of the original ASTs of the supported languages. This approach preserves the original semantics of the program, allowing developers to reason on it directly. However, Mopsa is not a disjoint union of separate analyzers, but tries to reuse abstractions between different languages. This is done via two key mechanisms: *semantic rewriting* and *delegation*.

Instead of an initial syntactic translation, as performed by classic analyzer frontends, Mopsa can rewrite a statement to another languages *during the analysis*. This dynamic translation can exploit the inferred values of variables to produce more precise, or more efficient transformations. For example, an integer addition in C can be translated to a mathematical addition if the sum of the operands fits within the range of the expression type. Similarly, an addition in Python can be translated to a mathematical addition if the operands have only values of integer types, and do not implement the special methods `__add__` and `__radd__`. Mopsa provides ready-to-use abstractions for mathematical integers, such as intervals and polyhedra. Therefore, both C and Python analyses can *delegate* the processing of the translated expression to one of these domains, which enables abstraction reuse between different analyses.

2.2 Precision

Sound static analyzers may be imprecise due to coarse over-approximations. This problem can be overcome by designing more accurate abstractions. These improvements are generally very localized, targeting specific

language constructs, such as how integer values are encoded or how loops are iterated. Therefore, it is important to decompose the global abstraction into smaller pieces, called *domains*, that are responsible for different parts of the language. Domains handling the same statements or expressions should be easily exchangeable without impacting the remaining analysis, or seamlessly combined in a reduced product to enhance precision.

In Mopsa, the definition of the global abstraction is done with a JSON file called a *configuration*, that lists the domains to include in the analysis as well as their order of execution and relations; and domains implement a common unified OCaml signature.

Unified Signature. From a developer perspective, the unified signature simplifies the integration of new domains. This signature provides an API to allow cooperation between domains while preserving low coupling among them. For example, when delegating the execution of a statement, a domain cannot call another domain directly. Instead, it asks the framework to find within the configuration the appropriate domain(s) implementing the transfer function of the statement.

Configurations. From a user perspective, the JSON format of configurations gives a simplified way to define a new analysis by combining existing domains. Mopsa supports different types of *combinators* to compose domains, such as sequences and reduced products. Many ready-to-use configurations with various tradeoffs between precision and efficiency are also provided.

2.3 Properties

Similarly to most static analyzers, Mopsa can verify classic reachability properties, such as absence of runtime errors in C and uncaught exceptions in Python. In addition, some analyses are experimented in Mopsa, for in-progress research projects. In particular, some target more complex reachability properties. Parolini and Miné (2024) verify the user-exploitability of alarms in C codes. Delmas et al. (2021) developed ways to prove portability of C programs between architectures with different endianness. Monat et al. (2024a) have developed an analysis targeting the rounding-sensitivity of date computations, in the context of legal implementations using the Catala programming language (Merigoux et al. (2021)).

These kinds of reachability properties rely on computing necessary post-conditions with an over-approximating forward analysis. The aim of these kinds of analysis is certifying the correctness of the program. Mopsa was recently extended to support the computation of sufficient preconditions via an under-approximating backward analysis described by Milanese and Miné (2024). This kind of analysis is able to generate counter-examples for certifying program incorrectness.

3 Measuring Precision

The precision of static analyzers is a cornerstone of experimental evaluations developed to evaluate the benefits of new approaches. It can also be leveraged to detect changes and regressions during tool development. We provide a quick survey of classic approaches to measuring precision in static analysis, before introducing a new way to automatically compute precision, which improves the transparency of the analysis and which can be numerically quantified. Then, we highlight how our approach can be naturally leveraged to compare analysis results, and how this comparison is used to detect changes during development, through continuous integration.

3.1 Traditional Approaches to Measuring Precision

The precision of a static analyzer is usually judged by separating the *true bugs* it found from the number of *false alarms* it raised.[1] In practice, this measure requires a *baseline* to be established. This baseline requires tedious manual work discriminating alarms, which is almost impossible to realize on new analyses of large-scale projects. We start by describing the case of manually annotated benchmarks where an absolute precision measure can thus be computed. We then survey usual ways to measure precision when no baseline is known – i.e, the number of *true bugs* is not known.

Absolute precision: the case of manually annotated benchmarks. Some specific benchmarks have been manually crafted, or studied, to know where the *true bugs* lie. This is for example the case of NIST's Juliet test suite for static analyzers by Black (2018), which contains tests labeled either as safe (no runtime errors) or buggy (with a single runtime error). In that case, establishing an absolute precision measure is possible, provided of course that the classification is correct. We leverage parts of the Juliet test suite to detect potential regressions of Mopsa in our continuous integration (cf. Section 3.4). We show in Table 1 our current precision results, measured as the percentage of tests where Mopsa is optimally precise.

[1] Note that in some cases, precision is measured through the proxy of another construction, such as generated call graphs Smaragdakis et al. (2011); Helm et al. (2024).

When no baseline exists: usual approaches to measuring precision. In most cases however, manually checking all alarms to separate true alarms from the others is not possible for the purpose of experimental evaluations made in academia. In those cases, other precision measures exist. A first measure, which can be useful on small testcases, is recording a boolean holding when the program is safe. This measure however is extremely coarse, which makes it unpractical to use during experimental evaluations or to discover improvements in an analysis. A second approach is to report the number of alarms. In our experience, this absolute number is however not informative: the numbers are difficult to put in context and might puzzle the community. This can be mitigated by measuring number of alarms per total lines of code, but this measure will be highly application-dependent.

Note that other approaches to quantify precision can be leveraged when the goal is to compare the results of two analyses of a same program. One can measure and compare abstract states at similar points of the analyses. This measure however yields more questions than it solves: which abstract states should be compared? For example, comparing all abstract states in the same contexts might be time-consuming due to the sheer number of states. In addition, some precision changes will propagate from one state to its successor(s), which may skew the measure. Additional issues can arise when incomparable domains are used. In related work, the clam static analyzer from Gurfinkel and Navas (2021); Gurfinkel et al. (2015) provides a `clam-diff` utility to compare two different analyses of a same program, which relies on a semantic comparison of the numerical abstract states computed in each analysis. This approach has more granularity than the one we currently use, and Arceri et al. (2023) have used it to evaluate precision changes in their work. However, this approach may be too sensitive and it could be interesting to quantify the difference between some comparisons. For example, the abstract state $a_1 = x < 1$ is included in $a_2 = x \leq 1$, which itself is included in $a_3 = \top$, but the change from a_2 to a_3 denotes a bigger precision loss. To the best of our knowledge, this problem of quantifying the difference between two abstract states has only been considered by Sotin (2010).

3.2 Reporting static analysis results in an automatic and transparent fashion

We have developed an approach to report static analysis results in Mopsa which is both automatic and enhances the transparency of the verifications performed by the analysis.

```
(* a# abstract state,          →    if a# ⋤ p# then
   p# safety property *)              add_alarm a# p#
if a# ⋤ p# then                     else
  add_alarm a# p#                     add_safe_check p#
```

Fig. 1: High-level implementation change, to move from reporting alarms to a transparent report of alarms and successful checks.

```
Checks summary: 21368 total, ✔ 18616 safe, ✗ 64 errors, ⚠ 2688 warnings
  Stub condition: 690 total, ✔ 513 safe, ✗ 3 errors, ⚠ 174 warnings
  Invalid memory access: 8139 total, ✔ 7143 safe, ✗ 4 errors, ⚠ 992 warnings
  Division by zero: 499 total, ✔ 445 safe, ⚠ 54 warnings
  Integer overflow: 11581 total, ✔ 10180 safe, ⚠ 1401 warnings
  Invalid shift: 163 total, ✔ 163 safe
  Invalid pointer comparison: 121 total, ✔ 84 safe, ⚠ 37 warnings
  Invalid pointer subtraction: 122 total, ✔ 37 safe, ✗ 57 errors, ⚠ 28 warnings
  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
Selectivity: 87.13%
```

Fig. 2: Analysis report summary for the analysis of `coreutils fmt`.

```
1  int main(int argc, char** argv) {
2    int y = -1;
3    for(int x = 0; x < argc; x++)
4      y++;
5  }
```

(a) Toy C example.

Stmt. \ Analysis	Intervals	Polyhedra
`x++`	Safe	Safe
`y++`	Alarm	Safe
Selectivity	50%	100%

(b) Selectivity measurement, in the case of integer overflow detection for the toy example of Section 3.2, analyzed either using intervals or a relational polyhedra abstract domain.

Fig. 3: Illustrating selectivity computation on a toy C example.

Traditionally, static analyzers only report alarms, which correspond to failed proofs of safety: program locations[2] where the abstract state does not satisfy a property the analyzer checks (such as absence of runtime errors). Our approach consists in also logging successful proofs, which we name safe checks (of a given a property). In the implementation, this change is conceptually easy to add, and boils down to the snippets shown in Figure 1. The analysis can then report all checks it performed, including alarms, and in a complementary way, the number of safe checks. An example of real analysis report is shown in Figure 2. From this report, we can derive a numeric notion of precision we call *selectivity*, measuring the ratio of successful proofs Mopsa has been able to perform. In the case of the analysis of `coreutils fmt` in Figure 2, the selectivity is 87%.

[2] and callstacks, for our fully context-sensitive analyses.

```
--- 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
```

(a) Comparing two analyses on a single program.

```
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
```

(b) Comparing two analyses on all coreutils.

Fig. 4: Mopsa-diff output comparing the impact of different relational domains on some `coreutils` programs.

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

We illustrate these notions of (safe) checks and selectivity computation on a toy example in Figure 3. In that case, we assume we are only interested in checking integer overflows, so we only check the two increments `x++` and `y++`. In a configuration using a non-relational interval analysis, Mopsa will be able to prove that `x` stays in the range of signed integers, but will fail to prove it for `y`. In that case, the selectivity is thus one half. Moving to a relational analysis relying on polyhedra, Mopsa will be able to infer that `y` is bounded by `x`, and hence prove that both operations are safe from overflows, resulting in a selectivity of 100%.

This approach has several benefits: it requires a lightweight implementation and provides transparent results, where users can clearly read in the analysis reports what the analysis has been able to verify. Finally, selectivity provides a relative measure of precision, making it more informative than reporting an absolute number of alarms. This measure depends on the complexity of the analyzed program expression and statements, rather than more arbitrary measures such as a program size.

3.3 Comparing analyses results using `mopsa-diff`

`mopsa-diff` is a tool we developed which can compare analyses reports: it can either focus on comparing two analyses of a given program, or on comparing analyses results on a set of programs. Comparing two analyses of a common program can be used to detect breaking changes (in terms of soundness or precision) when the implementation of the analyzer changes. Similarly, we can compare the performance-precision benefits of changing configurations. `mopsa-diff` can also be lifted to inspect the impact of a configuration change to a set of benchmarks for example.

These two modes are displayed in Figure 4, where we compare the analysis of some `coreutils` program with two relational numerical domains (chosen for illustrative purposes): PPLite v0.11 from Becchi and Zaffanella (2020) and the NewPolka implementation of Apron v0.9.14 from Jeannet and Miné (2009). Figure 4a compares two analyses of `coreutils touch`. The diff-like output shows that the PPLite version is able to remove 11 alarms compared to new Polka, although it is a bit slower, and adds another alarm. Figure 4b provides an overall comparison on all coreutils. We can notice that a few alarms are removed by this version, although two new alarms are generated. The analysis is overall slower. No new soundness assumptions[3] are recorded, and the PPLite analysis does not crash on any program NewPolka is able to analyze.

[3] During an analysis, Mopsa may make specific assumptions impacting the soundness of the analysis. Mopsa takes great care to report any such assumptions, and issues a special warning for the user to check them when there are some, for the sake of transparency. For example, Mopsa assumes by default that external C functions have no side effects on their parameters or global variables. We believe this approach to be a practical take on the principles highlighted in the soundiness manifesto of Livshits et al. (2015).

Benchmark	# Tests	Total LOC	Time	Precision
CWE121	2,508	234,930	3,064s	22.13%
CWE122	1,556	166,664	1,948s	25.84%
CWE124	758	93,372	961s	36.94%
CWE126	600	75,984	769s	46.83%
CWE127	758	89,022	963s	37.07%
CWE190	3,420	440,749	4,356s	78.13%
CWE191	2,622	340,884	3,236s	78.87%
CWE369	497	83,238	674s	70.42%
CWE415	190	17,990	228s	100.00%
CWE416	118	14,782	142s	67.80%
CWE469	18	1,520	22s	100.00%
CWE476	216	20,427	254s	100.00%

Table 1: Results of Mopsa analysis on Juliet benchmarks (non-relational configuration, no partitioning). CWE121 contains 2,508 tests, which overall take 3,064s to analyze. Mopsa is able to pass 22.13% with a precise analysis, and is imprecise in the 77.87% remaining cases.

Benchmark	LOC (C)	LOC (Python)
coreutils	148,214	0
pyperformance	0	4,215
fpp	0	3,140
bitarray	2,969	2,474
cdistance	912	979
levenshtein	5,120	357
llist	2,757	1,686
noise	636	631
pyahocorasick	2,933	1,336

Table 2: Link and lines of code (loc) of each project Mopsa uses in its continuous integration. LOCs have been measured using cloc from Danial (2021).

3.4 Detecting breaking changes during continuous integration

We leverage the analyses reports and `mopsa-diff` in our continuous integration to detect breaking changes affecting the soundness and precision of the analysis. This is done by comparing obtained results with baseline results. The set of benchmarks used in our continuous integration corresponds to all open-source projects we have analyzed in past experimental evaluations of our works. We strive not to modify the source code in order to stick close to a real-world usage. In some highly exceptional cases, we may rely on stubs to improve the results. We currently have just one stub for all benchmarks used in our continuous integration. Tables 1 to 3 show the benchmarks we currently use. Running times have been measured on a desktop machine relying on an Intel Core i7-12700.

	Benchmark	Time	Selectivity	# checks
coreutils	basename	33.79s	98.65%	11,731
	comm	42.67s	97.32%	12,654
	dircolors	34.82s	99.74%	20,062
	dirname	21.68s	99.61%	11,307
	echo	19.26s	99.43%	11,010
	false	14.50s	99.72%	10,774
	getlimits	34.62s	98.54%	11,711
	hostid	18.05s	99.65%	11,303
	id	32.69s	99.04%	12,338
	link	23.03s	99.52%	11,572
	logname	20.36s	99.66%	11,307
	mkfifo	34.87s	99.20%	11,807
	mknod	34.98s	99.11%	12,513
	nice	23.36s	99.55%	11,463
	nohup	26.98s	99.27%	11,734
	nproc	17.37s	99.44%	11,533
	printenv	23.59s	99.50%	11,202
	pwd	22.04s	99.62%	11,502
	rmdir	39.00s	99.22%	11,699
	runcon	18.55s	99.66%	11,215
	seq	42.68s	95.87%	14,310
	sleep	23.79s	99.46%	11,546
	stdbuf	32.16s	98.46%	12,526
	sync	24.53s	99.60%	11,273
	tee	35.69s	98.76%	12,057
	timeout	32.28s	98.51%	12,420
	true	9.55s	99.72%	10,774
	uname	20.61s	99.52%	11,943
	unlink	16.17s	99.63%	11,497
	users	20.82s	99.06%	11,668
	whoami	13.03s	99.66%	11,329
	yes	19.82s	99.45%	11,216
pyperformance	chaos	8.55s	98.86%	6,415
	fannkuch	0.31s	98.75%	321
	float	0.10s	99.48%	574
	go	143.72s	97.67%	7,552
	hexiom	39.30s	98.33%	5,392
	nbody	1.28s	99.73%	1,100
	raytrace	49.99s	98.95%	5,694
	regex_v8	18.30s	99.56%	32,638
	richards	11.86s	99.65%	3,710
	scimark	22.74s	99.09%	6,397
	spectral_norm	1.30s	99.43%	697
	unpack_sequence	3.46s	100.00%	8,094
fpp	choose	155.27s	99.76%	91,417
	processInput	3.84s	99.78%	4,914
	bitarray	499.26s	89.57%	288,475
	cdistance	58.26s	96.54%	46,512
	levenshtein	27.26s	85.35%	15,519
	llist	78.70s	98.92%	136,237
noise	perlin	5.80s	99.20%	4,273
	simplex	6.06s	99.45%	4,586
	pyahocorasick	29.92s	89.67%	20,415

Table 3: Benchmarks used in Mopsa's non-regression testsuite. `coreutils` benchmarks have been analyzed with fully symbolic arguments, and a relational analysis. Other projects have been analyzed in a non-relational setting.

4 Instrumenting the Analysis

Hooks are plug-ins that can observe, and influence, the analysis. When enabled, these hooks are called before and after every statement is analyzed, and they can peek at the input and output abstract states. Hooks do not have access to the private, internal representation of abstract domains, but communicate through a public interface of queries (described by Journault et al. (2019)).

These hooks have a wide variety of usages ranging from providing interpretation traces to improving the precision by detecting relevant thresholds which can be used by the widening. In the context of debugging and maintaining a static analyzer, these hooks are handy because they work at the level of the *abstract execution of the input program*, while standard profiling and coverage tools can provide information about the *execution of the static analyzer*. In the remainder of this section, we showcase computations of abstract coverage, abstract profiling, and heuristic detection of unsoundness and large imprecision.

4.1 Coverage

This hook computes the abstract coverage of statements that have been reached by the analysis. It gives a birds-eye view of where the analysis went, and can easily be leveraged to detect insufficient modeling assumptions reducing the search space of the analysis, as the user can quickly spot functions that should be reachable but are not. These modeling assumptions can be due to the modeling of command-line arguments.

As an example, the analysis of `coreutils fmt` shows that 76% of its `main` function is covered, when the analysis simply assumes that no arguments are passed to the utility. Given that `fmt` is a command-line utility, assuming that no arguments are passed is too restrictive. We can thus move to an analysis which performs a symbolic modeling of arguments: it considers that the analyzed program arguments are an array of arbitrary size (within either system bounds or user-supplied bounds) containing strings of arbitrary size and contents covering all actual possible usage. In this case of symbolic modeling of arguments, we reach 100% coverage for the `main` function. Note that by default, the analysis makes no restriction on the arguments of the `main` entry point (i.e, arguments are modeled symbolically by default).

This hook can help users find soundness issues related to their configuration and instrumention of the analysis.

4.2 Profiling

Loops and function calls are the two reasons why the same block of code might need to be analyzed a large number of times (due to loop iterations in one case, and context-sensitivity in the other case). They are thus the main source of analysis cost, and it is important to be able to pinpoint which ones might be problematic in a specific analysis. Mopsa thus provides two profiling hooks: one for loops and one for function calls.

The loop hooks tracks the number of times a given loop is called, the number of iterations needed to reach a fixpoint, as well as the total time spent analyzing each loop. This is helpful to fine-tune the widening, to track slowdowns in the postfixpoint computation performed by Mopsa, and more extremely, non-termination issues.

The function profiling hook tracks the number of times a function is analyzed as well as the time spent analyzing it. This hook is particularly important as the current analyses written in Mopsa are fully context-sensitive, meaning they analyze functions by virtual inlining. The hook thus helps identifying which functions take the most time to analyze. In particular, the collected data can be transformed into the flamegraph graphical representation from Gregg (2016); an example is shown in Figure 5. Tracking the number of times the function is analyzed is also important, as a frequent reason a function is reported as taking a large time is when the function is called many times, within nested loops, even if each function analysis is actually fast.

Hooks can produce and return information during abstract computations, without having to wait for the analysis to terminate. In particular, they work on partial executions of the analysis, which is especially useful for profiling programs where the analysis is unexpectedly long. It will only give a partial picture but will highlight the relevant loops and functions taking time to be analyzed.

4.3 Heuristic Unsoundness/Imprecision Detection

We developed plugins performing heuristic detection of unsound or highly imprecise behaviors, which are reported to the user. Most users will not precisely know the behavior of all domains an analysis configuration uses, especially due to the highly distributed nature of transfer functions in Mopsa. It is thus interesting to have hooks acting as runtime mechanisms that can warn the user of unsound or highly imprecise behavior, making it easier to pinpoint a source of imprecision or unsoundness.

The unsoundness detector acts as a sanity check. It currently verifies the following property: an assignment

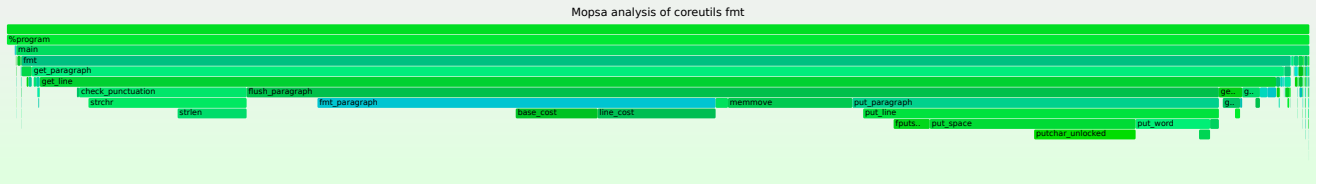


Fig. 5: Flamegraph obtained using the abstract function profiling of Mopsa, when analyzing `coreutils fmt`.

from a non-bottom state cannot return bottom[4]. Other properties could easily be added to this detector in future work. While the coverage hook finds modeling and configuration errors from users, this detector finds soundness issues due to bugs in the analyzer. These errors are critical, and corrected as soon as possible, as the analyses are constructed to be sound.

The imprecision detector warns when the analysis of a (sub)expression yields top, which is the source of large imprecision. Precision issues are less critical in Mopsa, as they usually result in a performance-precision tradeoff.

5 Abstract Debugging through an Interactive Engine

Traditional approaches to debugging static analyzers do not scale well. We briefly survey basic techniques – all supported by Mopsa – and their shortcomings. A first approach is to check the analysis output. It is usually quite coarse, and when unexpected behaviors happen, the output is not sufficient to get explanations. Another approach is to provide some builtin functions asking for the abstract interpreter to print its current state. Those can then be added in the source program to show specific abstract states. Bühler et al. (2024) mention the use of `Frama_C_show_each` in Frama-C; Mopsa provides a similar `mopsa_print`. While this approach helps making the result understandable, the cost is somewhat high: the modification of the source code requires to restart the analysis each time, which can be prohibitive when programs are large. If the program location where the printing happens is reached in a lot of different contexts, the output may turn out to be too verbose to be useful. Additionally, this restart of the analysis can become more complicated if the precise location of the origin of the error is different from the location where it becomes manifest. In that case, it must be discovered by trial and error, by inserting logging commands and running the analysis again many times. A complementary approach is to store relevant analysis states and information so that users can inspect them afterwards. Thus, a single analysis run can produce a log that can be examined at different program locations, even if these locations are not known in advance. The Goblint tool from Saan et al. (2024a) provides an interactive output allowing to explore various abstract states after the analysis is finished, either as an HTML page or through the Debug Adapter Protocol available in different IDEs and pioneered by VSCode. In Mopsa, we can record an interpretation trace, showing the order in which expressions and statements are analyzed, optionally with the abstract state. This however can quickly become too big to process. For example, the interpretation trace of analyzing `coreutils fmt` in Mopsa is around 12GB of text.

In this section, we showcase an abstract debugger, providing an interactive interface to the user. The user decides how they want to navigate the abstract execution of the program, and computations are performed on-the-fly accordingly. This debugger provides an interface similar to `gdb`, except that it works on the abstract execution of the program while `gdb` would work on a concrete execution. As Mopsa currently works as an interpreter on the AST (and not as a generic equation solver), the flow of execution of the analysis is close to that of the concrete execution of the program, and easy to understand. The abstract execution can be navigated using diverse strategies, such as going to the next statement, entering inner analysis of functions, or continuing until the next breakpoint. It is also possible to observe intra-instruction analysis, and rewriting operations that are at the heart of Mopsa. Breakpoints can be program locations, functions, transfer functions or the next detected alarm. The abstract state can be printed, as well as projections of abstract information related to selected abstract variables.

Combined with a terminal multiplexer such as `tmux`, this interactive interface can be used to perform some side-by-side debugging of the analysis of a same program in two different configurations. Thanks to some wrapper scripts we have developed, commands for the interactive engine have to be entered once and will be given to both sides of the analysis, further easing debugging.

We show how the interactive engine can be used in Figure 6, on an example where we analyze `coreutils`

[4] Except if the expression contains a runtime error, such as a division by zero.

```
Welcome to Mopsa 1.0~pre4!
Type 'help' to get the list of commands.

mopsa >> b #a; c
1 new alarm detected:

✗ Check #1:
~/mopsa/share/mopsa/stubs/c/libc/string.c: In function 'strlen':
~/mopsa/share/mopsa/stubs/c/libc/string.c:186.13-38: error: Stub condition

  186:  * requires: valid_string_or_fail(__s);
                   ^^^^^^^^^^^^^^^^^^^^^^^^^^

  Non-terminating string
  Callstack:
        from ~/mopsa/share/mopsa/stubs/c/libc/string.c:395.3-35: strlen
        from ./src/coreutils-8.30/lib/progname.c:59.10-30: strrchr
        from ./src/coreutils-8.30/src/fmt.c:325.2-28: set_program_name
        from ./src/coreutils-8.30/src/fmt.c:317.0-4: main
```

(a) Starting interface of interactive engine, with first command typed, to stop the analysis at the first alarm.

```
~/mopsa/share/mopsa/stubs/c/libc/string.c:186.13-38
   185  /*$
 ► 186   * requires: valid_string_or_fail(__s);
   187   * ensures:  return in [0, size(__s) - offset(__s));
   188   * ensures:  __s[return] == 0;
   189   * ensures:  forall size_t j in [0, return): __s[j] != 0;
   190   */
   191  size_t strlen (const char *__s);
S⟦ requires (valid_ptr_or_fail(__s) and
      (∃ i ∈[0 .. ((bytes(__s) - offset(__s)) - 1)] :
        (__s[_i] == 0) otherwise raise("Non-terminating string"))); ⟧
mopsa >>
```

(b) The engine jumped back to the beginning of the statement generating the alarm.

```
mopsa >> p __s
 float-itv ∪ (int-itv ∧ congruences) :
    bytes⦃@arg#0⦄ : [1,18446744073709551615] ∧ ℤ,
    offset⦃__s:~/mopsa/share/mopsa/stubs/c/libc/string.c:191.15-30⦄ :
      [0,0] ∧ 0,
    string-length⦃@arg#0⦄ : [0,18446744073709551614] ∧ ℤ ,
 pointers :
    __s:~/mopsa/share/mopsa/stubs/c/libc/string.c:191.15-30 :
      {  @arg#0  }
```

(c) Interval analysis: Inspecting abstract information related to `__s`.

```
mopsa >> b progname.c:59; b string.c:186; c; c
~/mopsa/share/mopsa/stubs/c/libc/string.c:186.13-38
   185  /*$
 ► 186   * requires: valid_string_or_fail(__s);
   187   * ensures:  return in [0, size(__s) - offset(__s));
   188   * ensures:  __s[return] == 0;
   189   * ensures:  forall size_t j in [0, return): __s[j] != 0;
   190   */
   191  size_t strlen (const char *__s);
S⟦ requires (valid_ptr_or_fail(__s) and
      (∃ i ∈[0 .. ((bytes(__s) - offset(__s)) - 1)] :
        (__s[_i] == 0) otherwise raise("Non-terminating string"))); ⟧
```

(d) Relational analysis started, with breakpoints to reach the alarm detected in Figure 6a.

```
mopsa >> p __s
 float-itv ∪ (int-itv ∧ congruences) :
    bytes⦃@arg#0⦄ : [1,18446744073709551615] ∧ ℤ,
    offset⦃__s:~/mopsa/share/mopsa/stubs/c/libc/string.c:191.15-30⦄ : [0,0] ∧ 0
    string-length⦃@arg#0⦄ : [0,18446744073709551614] ∧ ℤ ,
 numeric-relations(Symbolic arguments pack) :
   {  bytes⦃@arg#0⦄ ≤ 18446744073709551615,
      bytes⦃@arg#0⦄ ≥ string-length⦃@arg#0⦄ + 1,
      string-length⦃@arg#0⦄ ≥ 0  },
 pointers :
    __s:~/mopsa/share/mopsa/stubs/c/libc/string.c:191.15-30 :
      {  @arg#0  }
```

(e) Relational analysis: Inspecting abstract information related to `__s`.

Fig. 6: Interactive engine workflow example on the analysis of `coreutils fmt`.

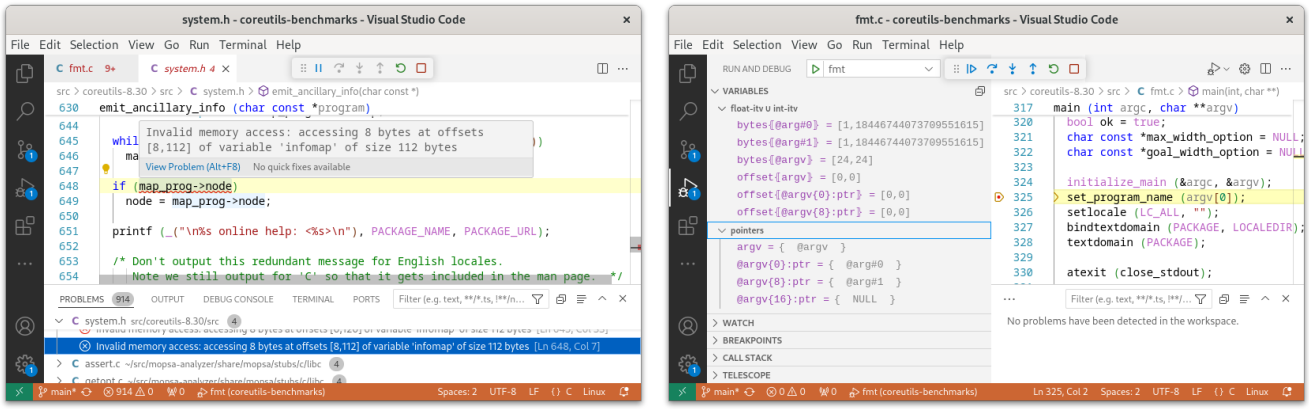
`fmt`. The analysis is fully context-sensitive, and supposes symbolic arguments are passed to `fmt`.

The interactive engine is shown just after it was started in Figure 6a. The prompt (represented by `mopsa >>`) has been given two commands: `breakpoint #alarm` and `continue` (abbreviated as `b #a` and `c`, and chained using a semicolon). This adds a breakpoint at the next encountered alarm and runs the analysis until this breakpoint is reached. The analysis raises an alarm in a call to builtin function `strrchr` at `progname.c:59`.

Once the first alarm is reached, the engine jumps back to the state and statement reached just before the alarm is triggered, easing inspection and understanding of the issue. Thanks to the functional implementation of Mopsa, this jumping back in time is easy to implement.[5] The state is shown in Figure 6b. The analysis currently checks that `strlen` can be called, through some preconditions encoded in the stub contract language of Ouadjaout and Miné (2020). The targeted precondition (at line 186) aims at verifying that the string passed to `strlen` is valid. The analysis is unable to prove the string is correctly encoded with a `'\0'` at its end, as expressed by the existentially quantified formula.

We can take a look at the abstract state to understand why the formula above cannot be proved correct (Figure 6c). There is no need to read the full abstract state, we can just ask for relevant abstract information related to `__s` with `print __s` (`print` can be abbreviated by `p`). We learn it points to a memory block of undetermined size (`bytes(@arg#0)`), and the position of the first 0 in the string (i.e, the auxiliary variable representing string lengths, `string-length(@arg#0)`, as defined by Journault et al. (2018)) is not known precisely either. Due to the non-relationality of the interval abstract domain, the analysis is thus unable to prove that the string `__s` has a terminating character.

[5] We are experimenting with ways to enable going further backwards in the analysis, which would provide the interface of a reverse debugger.



(a) The center panel shows source code annotated with one of the alarms reported by Mopsa through the LSP, and the list of alarms in the bottom panel.

(b) Using the DAP, a user can set a breakpoint at line 325 (right panel), and then inspect the program state (left panel).

Fig. 7: Analysis of `coreutils fmt` through VSCode's interface.

Let us restart the analysis in a relational setting, relying on the polyhedra abstract domain. In order to scale we use static packing (introduced by Bertrane et al. (2015)), a technique that keeps multiple polyhedra of small dimensions rather than a high-dimension polyhedra. We introduce here a specific pack handling the symbolic arguments used to analyze the program. In Figure 6d, we add breakpoints to reach the program location where the alarm was raised with the interval analysis (Figure 6a).

If we break to the program position where the alarm was raised with the interval analysis, we notice the numeric relations of Figure 6e express exactly what is needed to prove the existentially quantified formula from Figure 6b. This change of configuration means the first alarm has been removed by moving to a more expressive domain.

From command-line interfaces to IDEs. The whole interface of Mopsa, from its batch mode providing alarm reports to its abstract debugger, are available through the command-line. We provide similar interfaces for IDE users, by leveraging the Language Server Protocol (LSP) and the Debug Adapter Protocol (DAP) originally developed by VSCode. In particular, we can report results of our analysis directly on the source code thanks to the LSP Figure 7a, just as linters do. We use the DAP to provide interactive, abstract debugger sessions within the IDE, highlighted in Figure 7b.

6 Automated Testcase Reduction

Testcase reduction is an automated approach aiming at minimizing a test while keeping a specific property. One tool performing automated testcase reduction for C is `creduce`, developed by Regehr et al. (2012). It has originally been applied to compilers (where a test is an input program), and enjoys widespread use in this case. For example, GCC Wiki Contributors (2011) provide a guide to testcase reduction and asks for reduced testcases in their bug reports. We report our successful use of `creduce` on static analyzers, which are similarly highly complex pieces of code, manipulating potentially large input programs.

The `creduce` workflow is summarized in Figure 8. The user provides an input testcase (`file.c`) exhibiting an unwanted behavior (such as a crash). An oracle, written as a shell script, describes whether the unwanted behavior still occurs in partially reduced versions. `creduce` will then loop until it reaches a minimal testcase satisfying the oracle. One challenge of putting automated testcase reduction into practice is to establish sufficiently robust, yet automated, testcase oracles.

We start by describing two ways to leverage automated testcase reduction to ease debugging in Mopsa in Section 6.1. Our first usecase concerns internal errors from the analyzer that can happen deep into long analyses. Our second usecase pinpoints soundness issues by comparing two analyses, yielding different results on a same program. We finish by highlighting in Section 6.2 that the collaboration between Mopsa and `creduce` is a two-way street, as Mopsa can simplify testcase reduction for multi-file projects with complex commands such as `GNU coreutils`.

6.1 Leveraging Automated Testcase Reduction

Internal error reduction. We have successfully used `creduce` to pinpoint and fix internal errors within Mopsa

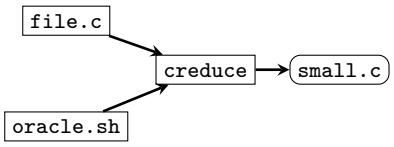


Fig. 8: creduce pipeline.

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

Table 4: Internal error reduction examples, taken from the Gitlab repository of Mopsa.

which result in uncaught OCaml exceptions terminating the analysis. In that case, the oracle we use first checks that the produced file can be compiled without errors[6], and then checks that the same internal error is raised during the analysis of the reduced testcase. In our experience, this kind of reduction only requires the exception name and message to be passed to the oracle. We provide in Table 4 examples of issues or merge requests where automated testcase reduction has been used. These cases stem from benchmarks from the Software-Verification Competition, or from `coreutils` programs. Lines of code correspond to the number of lines of preprocessed C code (cf. Section 6.2), formatted using `clang-format`, and measured through command `wc -l`. In our experience, reduction time is less than 12 hours, meaning that it can easily be run overnight and save a considerable amount of human time. It helped us solve issues that were otherwise out of reach. For example, issue 81 from Table 4 was reduced from one linux device driver code from the Software Verification Competition, and caused by an implementation bug in the string length domain from Journault et al. (2018).

Differential-configuration reduction. The testcase reduction for internal errors is the canonical usecase of such tools. We have had some recent successes in *differential-configuration reduction*, easing the debugging of cases where two different configurations of Mopsa (Section 2.2) yield contradictory analysis results on a given program. We have typically applied it when an analysis is unsound, and where the culprit (abstract domain or reduction) is included in one configuration and not the other. This is particularly useful given the large number of combinations of abstract domains Mopsa enables through its modular design. We have used this approach to simplify some soundness issues reported by external users (# 179, # 182), who then integrated it in their process when reporting further issues (# 184, # 185). Thanks to this approach, we have minimal programs in which to debug the source of unsoundness. For example, the reduction from issue 182 allowed to quickly identify the division from the integer powerset abstraction as the source of unsoundness which was then easily fixed. We have currently not explored how `creduce` could help us pinpoint precision improvements.

[6] `creduce` can produce C programs with illegal syntax or types, that are naturally rejected (early) by a compiler's frontend.

6.2 Leveraging Mopsa to Ease Multi-file Reduction

One of the current usability barriers to automated testcase reduction through `creduce` and its sibling `cvise` (developed by Pflanzer and Liška (2024)) is the support of multi-file projects. Indeed, `creduce` requires the explicit list of files to be reduced[7]. This list can be difficult to establish on large open-source projects, such as coreutils, where a build system like `make` takes care of compiling the various sources into an executable, through a list of complex rules. In addition, some large projects may use different files with different compilation options, which would create an additional difficulty in using standard `creduce`.

Mopsa natively supports the analysis of multi-file C projects, through a utility called `mopsa-build` which creates a compilation database by instrumenting the compilation process. This process can include `configure` scripts, `make`, `cmake`, etc. `mopsa-build` overrides environment variables to record all compiler and linker calls, as well as the options that were passed to them. Due to its seamless nature, `mopsa-build` can be used as a drop-in replacement of various build systems such as `make`: `mopsa-build make`. Then, this compilation database can be leveraged by the C analysis to analyze a specific target of the build system. An important side-effect of this process is the ability of the analysis to generate a single preprocessed file[8], which does heavily simplify automated testcase reduction.

[7] https://github.com/csmith-project/creduce/blob/31e855e290970cba0286e5032971509c0e7c0a80/creduce/creduce.in#L197

[8] The single file is generated as a kind of source-level (or AST-level) linking.

7 Related Work

To the best of our knowledge Andreasen et al. (2017) are the first to describe approaches they use to improve their static analysis of JavaScript. They describe combinations of techniques relying on delta-debugging, soundness testing (through comparison with concrete values) and blended analysis (injection of concrete values to restrict the abstract state). The automated testcase reduction we use is close to the delta-debugging techniques they rely on, but otherwise our approaches seem complementary.

Static analyzer interfaces. Our interactive engine can act as an abstract debugger, either from the command-line interface, or through IDEs supporting the debug-adapter protocol (DAP). Some other analyzers can report their alarms through the language-server protocol (LSP) developed initially for linters. A specific interface called MagpieBridge has been developed by Luo et al. (2019) to simplify the integration of static analyzers within LSP. Molle et al. (2023) showcase a cross-level debugger, working on the analyzed program and enabling conditional breakpoints expressed either on the analyzed program or the state of the analyzer itself. The Goblint static analyzer provides a different kind of abstract debugger developed by Holter et al. (2024). It provides an interactive, graphical exploration of the abstract states in the control-flow graph of the program once the analysis has successfully finished. Both abstract debuggers from Goblint and Mopsa provide an IDE integration through the Debug Adapter Protocol. In their documentation of the Goblint analyzer, Saan et al. (2024b) also mention automated testcase reduction, in particular to debug fixpoint termination issues.

A blog post from the Frama-C team by Maroneze (2020) highlights that automated testcase reduction simplified their interaction with industrial clients having private codebases. At least for runtime errors, these industrial clients can run automated testcase reduction by themselves. The reduction will yield a highly simplified testcase that will not leak important parts of the initial, private codebase, which can then be sent to Frama-C developers without any confidentiality issues.

Testing the soundness and precision of static analyzers. Bugariu et al. (2018) perform automated testing of numerical abstract domains, by checking that some chosen properties should be verified. We rely on a similar yet simplified approach in Section 4.3, through an encoding of a heuristic rule to detect unsoundness during an analysis. However, our approach is not specific to numerical abstract domains. Goblint relies on a similar approach implemented by Saan and Schwarz (2022) to check that at least one branch in conditionals is not dead.

Klinger et al. (2019) describe ways to automatically compare the soundness and precision of different C static analyzers on programs from the Software-Verification Competition (SV-Comp). In particular, the original program can be mutated to check the results at different program points, and a notion of $\delta-$unsoundness is established, to reduce spurious warnings. In our case, we have only considered different configurations of Mopsa, but not compared it using testcase reduction to other static analysis tools. Taneja et al. (2020) develop SMT-based algorithms that can detect soundness and precision errors of some of LLVM dataflow analyses.

Formally verified static analyzers, such as the work of Jourdan et al. (2015), will not require debugging of unsound results by construction. However, we believe the techniques we presented could still be interesting to investigate precision issues.

8 Conclusion

This article documents and shares the practices we have established for the maintenance of the Mopsa static analyzer during the last 7 years. In particular, we rely on a measure of precision than can be computed automatically on software without baselines on the number of true bugs. This approach increases the transparency of the analysis, and simplifies regression detection. We have shown different tools (profiler, debugger) focusing on the abstract execution of the analyzed program, and reported use of automated testcase reduction to simplify our debugging. Following the work of Andreasen et al. (2017), we hope this article will inspire other groups and encourage other researchers to document and share their practices.

There are still some challenges revolving around the development of Mopsa. The number of different configurations to analyze a given language can grow quite quickly due to the modular architecture of Mopsa. We are currently performing regression tests on selected, specific configurations to reduce the computational cost. Code maintenance and debugging can still take some time, and onboarding material takes a lot of time to create and maintain. Finally, we are looking into providing an install-free version of Mopsa – through a web page for example – meaning that prospective users can quickly test it without any installation.

References

Andreasen ES, Møller A, Nielsen BB (2017) Systematic approaches for increasing soundness and precision of static analyzers. In: Ali K, Cifuentes C (eds) Proceedings of the 6th ACM SIGPLAN International Workshop on State

Of the Art in Program Analysis, SOAP@PLDI 2017, Barcelona, Spain, June 18, 2017, ACM, pp 31–36, DOI: 10.1145/3088515.3088521

Arceri V, Dolcetti G, Zaffanella E (2023) Speeding up static analysis with the split operator. In: Ferrara P, Hadarean L (eds) Proceedings of the 12th ACM SIGPLAN International Workshop on the State Of the Art in Program Analysis, SOAP 2023, Orlando, FL, USA, 17 June 2023, ACM, pp 14–19, DOI: 10.1145/3589250.3596141

Bau G, Miné A, Botbol V, Bouaziz M (2022) Abstract interpretation of Michelson smart-contracts. In: Proc.~of the 11th ACM SIGPLAN International Workshop on the State Of the Art in Program Analysis (SOAP'22), ACM, pp 36–43, DOI: 10.1145/3520313.3534660

Becchi A, Zaffanella E (2020) Pplite: Zero-overhead encoding of NNC polyhedra. Inf Comput 275:104620, DOI: 10.1016/J.IC.2020.104620

Bertrane J, Cousot P, Cousot R, Feret J, Mauborgne L, Miné A, Rival X (2015) Static analysis and verification of aerospace software by abstract interpretation. Found Trends Program Lang 2(2-3):71–190, DOI: 10.1561/2500000002

Black PE (2018) Juliet 1.3 test suite: Changes from 1.2. US Department of Commerce, National Institute of Standards and Technology

Bugariu A, Wüstholz V, Christakis M, Müller P (2018) Automatically testing implementations of numerical abstract domains. In: Huchard M, Kästner C, Fraser G (eds) Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018, Montpellier, France, September 3-7, 2018, ACM, pp 768–778, DOI: 10.1145/3238147.3240464

Bühler D, Maroneze A, Perrelle V (2024) Abstract Interpretation with the EvaEva (Frama-C plug-in) Plug-in, Springer International Publishing, Cham, pp 131–186. DOI: 10.1007/978-3-031-55608-1_3

Cousot P, Cousot R (1977) Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: POPL, ACM, pp 238–252, DOI: 10.1145/512950.512973

Danial A (2021) cloc: v1.92. DOI: 10.5281/zenodo.5760077

Delmas D, Ouadjaout A, Miné A (2021) Static analysis of endian portability by abstract interpretation. In: SAS, Springer, Lecture Notes in Computer Science, vol 12913, pp 102–123, DOI: 10.1007/978-3-030-88806-0_5

GCC Wiki Contributors (2011) Gcc wiki – a guide to testcase reduction. URL https://gcc.gnu.org/wiki/A_guide_to_testcase_reduction

Gregg B (2016) The flame graph. Commun ACM 59(6):48–57, DOI: 10.1145/2909476, URL https://doi.org/10.1145/2909476

Gurfinkel A, Navas JA (2021) Abstract interpretation of LLVM with a region-based memory model. In: VSTTE, Springer, Lecture Notes in Computer Science, vol 13124, pp 122–144, DOI: 10.1007/978-3-030-95561-8_8

Gurfinkel A, Kahsai T, Komuravelli A, Navas JA (2015) The seahorn verification framework. In: CAV (1), Springer, Lecture Notes in Computer Science, vol 9206, pp 343–361, DOI: 10.1007/978-3-319-21690-4_20

Helm D, Keidel S, Kampkötter A, Düsing J, Roth T, Hermann B, Mezini M (2024) Total recall? how good are static call graphs really? In: ISSTA 2024

Holter K, Hennoste JO, Lam P, Saan S, Vojdani V (2024) Abstract debuggers: Exploring program behaviors using static analysis results. In: Edwards J (ed) Proceedings of the 2024 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software, Association for Computing Machinery, Onward! 2024, to appear

Jeannet B, Miné A (2009) Apron: A library of numerical abstract domains for static analysis. In: CAV, Springer, Lecture Notes in Computer Science, vol 5643, pp 661–667, DOI: 10.1007/978-3-642-02658-4_52

Jourdan J, Laporte V, Blazy S, Leroy X, Pichardie D (2015) A formally-verified C static analyzer. In: POPL, ACM, pp 247–259, DOI: 10.1145/2676726.2676966

Journault M, Miné A, Ouadjaout A (2018) Modular static analysis of string manipulations in C programs. In: Podelski A (ed) Static Analysis - 25th International Symposium, SAS 2018, Freiburg, Germany, August 29-31, 2018, Proceedings, Springer, Lecture Notes in Computer Science, vol 11002, pp 243–262, DOI: 10.1007/978-3-319-99725-4_16

Journault M, Miné A, Monat R, Ouadjaout A (2019) Combinations of reusable abstract domains for a multilingual static analyzer. In: VSTTE, Springer, Lecture Notes in Computer Science, vol 12031, pp 1–18, DOI: 10.1007/978-3-030-41600-3_1

Klinger C, Christakis M, Wüstholz V (2019) Differentially testing soundness and precision of program analyzers. In: Zhang D, Møller A (eds) Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2019, Beijing, China, July 15-19, 2019, ACM, pp 239–250, DOI: 10.1145/2644805

Livshits B, Sridharan M, Smaragdakis Y, Lhoták O, Amaral JN, Chang BE, Guyer SZ, Khedker UP, Møller A, Vardoulakis D (2015) In defense of soundiness: a manifesto. Commun ACM 58(2):44–46, DOI: 10.1145/2644805

Luo L, Dolby J, Bodden E (2019) Magpiebridge: A general approach to integrating static analyses into ides and editors (tool insights paper). In: Donaldson AF (ed) 33rd European Conference on Object-Oriented Programming, ECOOP 2019, July 15-19, 2019, London, United Kingdom, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, LIPIcs, vol 134, pp 21:1–21:25, DOI: 10.4230/LIPICS.ECOOP.2019.21

Maroneze A (2020) Debugging Frama-C analyses: Better privacy with C-reduce. URL https://frama-c.com/2020/04/02/creduce.html

Merigoux D, Chataing N, Protzenko J (2021) Catala: a programming language for the law. Proc ACM Program Lang 5(ICFP):1–29, DOI: 10.1145/3473582

Milanese M, Miné A (2024) Generation of violation witnesses by under-approximating abstract interpretation. In: VMCAI (1), Springer, Lecture Notes in Computer Science, vol 14499, pp 50–73, DOI: 10.1007/978-3-031-50524-9_3

Miné A, Ouadjaout A, Journault M, Fromherz A, Monat R, Parolini F, Milanese M, Boillot J (2024) Mopsa: Modular open static analysis platform. URL https://gitlab.com/mopsa/mopsa-analyzer/

Molle MV, Vandenbogaerde B, Roover CD (2023) Cross-level debugging for static analysers. In: Saraiva J, Degueule T, Scott E (eds) Proceedings of the 16th ACM SIGPLAN International Conference on Software Language Engineering, SLE 2023, Cascais, Portugal, October 23-24, 2023, ACM, pp 138–148, DOI: 10.1145/3623476.3623512

Monat R (2021) Static type and value analysis by abstract interpretation of python programs with native C libraries. (analyse statique, de type et de valeur, par interprétation abstraite, de programmes python utilisant des librairies C). PhD thesis, Sorbonne University, Paris, France, URL https://tel.archives-ouvertes.fr/tel-03533030

Monat R, Ouadjaout A, Miné A (2020a) Static type analysis by abstract interpretation of python programs. In: ECOOP, Schloss Dagstuhl - Leibniz-Zentrum

für Informatik, LIPIcs, vol 166, pp 17:1–17:29, DOI: 10.4230/LIPICS.ECOOP.2020.17
Monat R, Ouadjaout A, Miné A (2020b) Value and allocation sensitivity in static python analyses. In: SOAP@PLDI, ACM, pp 8–13, DOI: 10.1145/3394451.3397205
Monat R, Ouadjaout A, Miné A (2021) A multilanguage static analysis of python programs with native C extensions. In: SAS, Springer, Lecture Notes in Computer Science, vol 12913, pp 323–345, DOI: 10.1007/978-3-030-88806-0_16
Monat R, Ouadjaout A, Miné A (2023) Mopsa-c: Modular domains and relational abstract interpretation for C programs (competition contribution). In: TACAS Part II, Springer, Lecture Notes in Computer Science, vol 13994, pp 565–570, DOI: 10.1007/978-3-031-30820-8_37
Monat R, Fromherz A, Merigoux D (2024a) Formalizing date arithmetic and statically detecting ambiguities for the law. In: ESOP (2), Springer, Lecture Notes in Computer Science, vol 14577, pp 421–450, DOI: 10.1007/978-3-031-57267-8_16
Monat R, Milanese M, Parolini F, Boillot J, Ouadjaout A, Miné A (2024b) Mopsa-c: Improved verification for C programs, simple validation of correctness witnesses (competition contribution). In: TACAS Part III, Springer, Lecture Notes in Computer Science, vol 14572, pp 387–392, DOI: 10.1007/978-3-031-57256-2_26
Namjoshi KS, Pavlinovic Z (2018) The impact of program transformations on static program analysis. In: SAS, Springer, Lecture Notes in Computer Science, vol 11002, pp 306–325, DOI: 10.1007/978-3-319-99725-4_19
Ouadjaout A, Miné A (2020) A library modeling language for the static analysis of C programs. In: Pichardie D, Sighireanu M (eds) SAS, Springer, Lecture Notes in Computer Science, vol 12389, pp 223–247, DOI: 10.1007/978-3-030-65474-0_11
Parolini F, Miné A (2024) Sound abstract nonexploitability analysis. In: VMCAI (2), Springer, Lecture Notes in Computer Science, vol 14500, pp 314–337, DOI: 10.1007/978-3-031-50521-8_15
Pflanzer M, Liška M (2024) CVise: Super-parallel python port of the C-Reduce. URL https://github.com/marxin/cvise/
Regehr J, Chen Y, Cuoq P, Eide E, Ellison C, Yang X (2012) Test-case reduction for C compiler bugs. In: Vitek J, Lin H, Tip F (eds) ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '12, Beijing, China - June 11 - 16, 2012, ACM, pp 335–346, DOI: 10.1145/2254064.2254104
Saan S, Schwarz M (2022) URL https://github.com/goblint/analyzer/issues/826
Saan S, Schwarz M, Erhard J, Tilscher S, Holter K, Vogler R, Apinis K, Vojdani V (2024a) Goblint. DOI: 10.5281/zenodo.5735006, URL https://github.com/goblint/analyzer
Saan S, Schwarz M, Erhard J, Tilscher S, Holter K, Vogler R, Apinis K, Vojdani V (2024b) Goblint analyzer documentation on testcase reduction. URL https://goblint.readthedocs.io/en/stable/developer-guide/debugging/#debugging-issues-with-larger-programs
Smaragdakis Y, Bravenboer M, Lhoták O (2011) Pick your contexts well: understanding object-sensitivity. In: POPL, ACM, pp 17–30, DOI: 10.1145/1926385.1926390
Sotin P (2010) Quantifying the Precision of Numerical Abstract Domains. Research report, URL https://inria.hal.science/inria-00457324
Taneja J, Liu Z, Regehr J (2020) Testing static analyses for precision and soundness. In: CGO '20: 18th ACM/IEEE International Symposium on Code Generation and Optimization, San Diego, CA, USA, February, 2020, ACM, pp 81–93, DOI: 10.1145/3368826.3377927