Try-Mopsa: Relational Static Analysis in Your Pocket*

Raphaël Monat[1] (✉)

Univ. Lille, Inria, CNRS, Centrale Lille, UMR 9189 CRIStAL, F-59000 Lille, France

Abstract. Static analyzers are complex pieces of software with large dependencies. They can be difficult to install, which hinders adoption and creates barriers for students learning static analysis. This work introduces Try-Mopsa: a scaled-down version of the Mopsa static analysis platform, compiled into JavaScript to run purely as a client-side application in web browsers. Try-Mopsa provides a responsive interface that works on both desktop and mobile devices. Try-Mopsa features all the core components of Mopsa. In particular, it supports relational numerical domains. We present the interface, changes and adaptations required to have a pure JavaScript version of Mopsa. We envision Try-Mopsa as a convenient platform for onboarding or teaching purposes.

Keywords: Static Analysis · Abstract Interpretation · Usability · Teaching.

1 Introduction

Static analyzers are complex pieces of software, usually building on a large number of dependencies. For example, the Mopsa static analysis platform [18] requires among others: two parsing libraries (Menhir and libclang), the Zarith library to handle arbitrary precision arithmetic, and the Apron library to handle relational numerical domains. When facing a large number of users (or students), there is always a chance to encounter some installation issues. While good packaging or containerization can certainly limit those, installing a new tool still consumes time and resources. In any case, the installation process hinders both testing and adoption of new static analysis tools.

One remedy to this issue is to provide zero-install ways to try a software, for example by enabling tool usage through a web browser. This article presents Try-Mopsa, a scaled-down version of the Mopsa static analysis platform that runs entirely as a client-side web application. It relies on a responsive interface to support a wide variety of devices (from smartphones to computers), and supports all core features of Mopsa, including relational domains and an interactive engine acting as an abstract debugger. Try-Mopsa is available online [30]. Thanks to

* This work is partially supported by grant agreement ANR-24-CE25-7956-01 RAISIN from the French Agence Nationale de la Recherche, and by an Amazon Research Award, Fall 2024.

its purely client-side implementation, Try-Mopsa scales to a large number of concurrent users.

We provide a brief overview of Mopsa in Section 2. Then, we describe in Section 3 how we adapted Mopsa to make it runnable in a web page. Section 4 provides an overview of the resulting web interface, Section 5 evaluates several features of the implementation, and Section 6 discusses related work.

2 A Brief Overview of Mopsa

Mopsa is a Modular Open Platform for Static Analysis, rooted within the abstract interpretation framework [6]. It aims at providing a convenient platform for static analysis learners, developers and users. Although Mopsa explores some new perspectives for the design of static analyzers, it is stable and precise enough to be on-par with state-of-the-art academic program analyzers participating to the Software-Verification Competition [2, 33]. Journault et al. [18] describe the core of Mopsa's principles, and Monat [29, Chapter 3] provides an in-depth introduction to Mopsa's architecture. We briefly describe three features of interest for this article:

Multilanguage support. Mopsa supports the analysis of multiple programming languages. Currently, it supports the analysis of an in-house toy imperative language (called "Universal"), of C [39] and of Python [31, 34].

User-defined analysis combination. As an analysis platform, Mopsa offers a wide variety of abstract domains to choose from. Users define in a configuration file which abstract domains they want to enable, and how they should be combined (reduced product, ...).

Tailored for relational domains. Relational domains greatly improve the expressiveness of analyses by being able to infer constraints between variables. In addition, every abstract domain can introduce ghost variables and add constraints on those, which can be handled by an underlying relational domain.

We show in Figure 1 the main components of Mopsa, and describe them below. All components are supported by Try-Mopsa, except those filled in gray. We discuss unsupported components and potential replacements in Section 3.

The frontend handles the parsing of a program into an abstract syntax tree (AST). As we mentioned above, Mopsa currently supports the analysis of three programming languages.

The analysis builder takes a JSON configuration file describing the choice of abstract domains and their combinators, makes sure it is valid, and enables the corresponding abstract domains in the toplevel analysis. It also sets passed options (either for the framework or for the enabled abstract domains).

Once the parsing is done and the abstract domains are combined according to the specified configuration, the toplevel analysis starts. It runs the combination of chosen abstract domains, such as iterators handling loops and function calls, trace and state partitioning, numerical abstract domains and string abstractions.

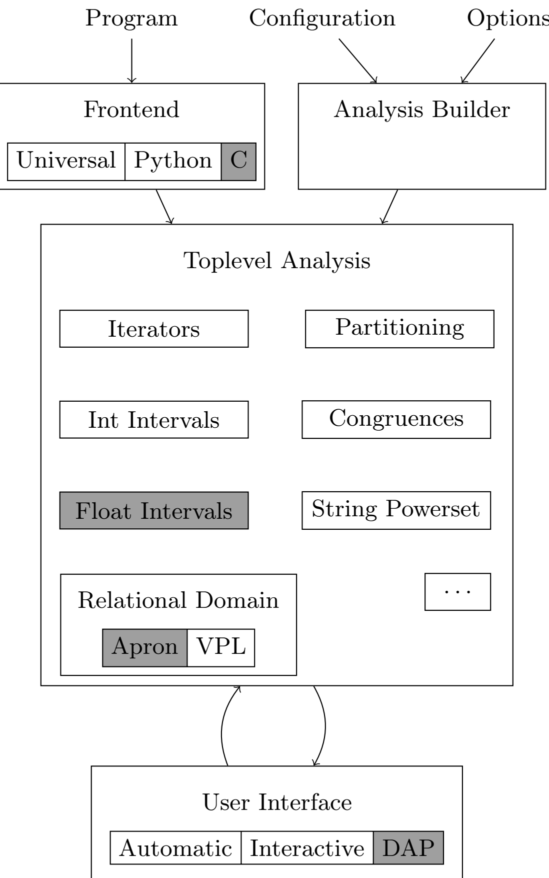

Fig. 1. Components of Mopsa. All components are supported by Try-Mopsa, except those filled in gray.

Different user interfaces are offered by Mopsa. The automatic interface is the classic one: the analysis runs to completion and then displays the results. The interactive engine lets user navigate the abstract execution of the program, where analysis computations are performed on-the-fly accordingly. This interface acts as a `gdb`-like abstract debugger, and supports breakpoints, program navigation, printing of abstract states, ... The DAP interface provides a debug adapter protocol interface, similar to the interactive engine but allowing interactions with IDEs supporting this protocol. Monat et al. [32, Section 5] describes the former two interfaces in more details.

3 Try-Mopsa: Under the Hood

This section describes the implementation of Try-Mopsa. Section 3.1 focuses on the backend part, explaining how we adapted Mopsa to be able to compile it to JavaScript. Section 3.2 shows how this compiled JavaScript is integrated within a web page to provide a user-friendly interface.

3.1 Compiling Mopsa to JavaScript

According to `cloc` [8], the current version of Mopsa consists of 87,856 lines of code. The overwhelming majority of the codebase (89%) is written in OCaml. We rely on the Js_of_ocaml (JSOO) compiler [46] to compile these components of Mopsa to JavaScript. This compiler option is selected in the build system as a drop-in replacement to the standard OCaml compiler creating executables. This compilation process already supports most Mopsa features, including the parsing of user-defined analysis combination, fixpoint and interprocedural iterators, heap and string abstractions, trace and state partitioning.

However, some crucial features of Mopsa rely on external dependencies: we use Menhir or libclang to parse programs, the Zarith library to handle arbitrary precision arithmetic, and the Apron library to handle relational numerical domains. We now discuss how these components have been adapted to compile to JavaScript.

Parsing libraries. Mopsa relies on the Menhir library [40] to implement parsers for the Universal and the Python programming languages. As Menhir is written in pure OCaml, JSOO natively supports compiling it to JavaScript.

Try-Mopsa currently does not support C parsing. Indeed, Mopsa depends on LLVM and its libclang library to parse C programs, and our C parser contains 4,182 lines of C++ glue code. While JSOO can interface with manually written JavaScript stubs, and some prototype versions of LLVM have been compiled to WebAssembly, we believe the current level of support for these processes would require too much manual work.

Arbitrary-precision integer arithmetic. Our integer abstractions (intervals, congruences) rely on arbitrary-precision arithmetic to avoid overflows and unsoundness. Mopsa relies on the Zarith [26] library to implement these mathematical integers. Try-Mopsa makes use of a third-party alternative implementation in JavaScript of the Zarith interface, called Zarith_stubs_js [14] and natively supported by the compilation process of JSOO.

Relational Domains. Mopsa depends on the Apron library [17] to support relational numeric domains such as octagons [24] and polyhedra [7]. However, Apron abstract domains are written in C/C++: compiling them to JavaScript, just like our C frontend, would be quite cumbersome. For Try-Mopsa, we chose to replace Apron with the alternative Verified Polyhedra Library (VPL) [3]. Given that some domains of the VPL are pure OCaml implementations, JSOO is able

to compile it to JavaScript, enabling relational abstract domains to run within web browsers. We wrote a new VPL wrapper, making it compatible with Mopsa's interface, and adding support for the `fold/expand` operators [12] required by Mopsa but not built in the VPL.

Floating-point Abstractions. Our current implementation of floating-point intervals sets various rounding modes in the floating-point unit to be sound [23]. Since this is not supported by JavaScript, these intervals are currently not supported by Try-Mopsa. Alternative implementations of floating-point intervals, not relying on specific rounding modes could be developed in future work to alleviate this limitation.

3.2 Handling Web Interactions

Try-Mopsa is split between a toplevel, handling the page interaction with the user and a web worker, handling the actual computations of the analysis. Thanks to this decoupling, the web worker and the page rendering lies within different threads of the web browser, so longer computations do not freeze the browser's rendering process.

Web Worker. The web worker relies primarily on the JavaScript executable obtained through the process described in the previous section.

The compiled version of Mopsa relies on standard output to interact with the user. Additionally, the interactive engine reads user commands from the standard input. We rely on JSOO's utilities to set up callbacks intercepting any standard input/output and redirecting them to the toplevel. Thanks to this approach, Mopsa did not require any modification to work in the browser.

We rely on the virtual filesystem mechanism of JSOO to embed internal files Mopsa may expect to find during its analysis. This currently concerns Python stubs for various library modules.

Toplevel. The toplevel renders the initial dynamic elements of the page and updates them in response to user action.

It uses the Ace editor [42], and the Ezjs_ace bindings [38] to display the input program and the analysis results. We have extended the Ace editor to provide syntax highlighting for the Universal language, to allow collapsing of printed elements within the analysis result, and to render ANSI escape codes in HTML, as those are used by Mopsa to print in color.

The toplevel also draws a form so that user can select options. The form is automatically generated from the option metadata encoded within Mopsa (which are fetched through a query to the web worker). The options depend on which abstract domains are activated, and thus on the configuration chosen by the user. The toplevel ensures that the option form is redrawn whenever the configuration is changed.

When a user starts an analysis, the toplevel creates a web worker to perform it. The user may choose to interrupt the analysis, in which case the top-level then signals the web worker to stop.

Interactive engine. The interactive engine of Mopsa follows an interaction loop by awaiting user input (in a synchronous, blocking fashion) and providing corresponding results. However, web interfaces typically rely on asynchronous processes. We explored options such as implementing an asynchronous interactive engine. This would have required the decoupling of the interactive interface from the analysis computations, which would have been a large implementation effort introducing breaking changes. In the end, we chose to rely on the sync-message [27] JavaScript library enabling synchronous communication between the web worker to the toplevel, should some user input be required.

4 Try-Mopsa: Interface Overview

The interface of Try-Mopsa is displayed in Figures 2, 4 and 5 and can be tried online [30]. The interface is responsive, to ensure it can be used on a wide variety of screens, from smartphones to desktops.

The landing page shown in Figure 2 provides a code editor and displays analysis results. By default, input programs are written in our toy imperative language. Users can also switch to the Python analysis if they wish to, and they can load some example programs. The analysis output consists of abstract states displayed when the `print()` instruction is analyzed, as well as a report on the runtime errors potentially detected by the analyzer. If the interactive engine has been enabled, the interaction happens in this same box. Note that on wider screens, the responsive design displays the code editor and analysis results side-by-side.

In Figure 2, we loaded the example program `str_alphabet2.u`, reproduced textually in Figure 3. Note that in the web editor, the definition of `to_string`, acting as a cast, is folded to highlight the editor capabilities.

The program starts with the string `s`, initialized to the letter "a". It then runs a loop until the number defined by `'a' + i` reaches the end of the lowercase alphabet. Similarly to C, in Universal, characters (between single quotes) are interpreted as integers through their ASCII code. This loop may stop non-deterministically at any iteration (line 6). At each loop iteration, the string `s` is concatenated (with the @ operator) with the singleton string whose character corresponds to the integer `'a' + i`. Thus the program non-deterministically computes a string `s` among $S = \{"a", "ab", "abc", \ldots, "abc \cdots xyz"\}$. The assertion at line 14 checks a simpler property: that `s` belongs S but up to any permutation of the characters.

Figure 4 shows the configuration editor, letting users specify their choice of abstract domains and combinators. Similarly to the code editor, users can also load a configuration from a pre-defined list. We comment on the loaded configuration, `string_product_relational.json`. This configuration features three notable abstract domains, working together. The string length domain handles ghost variables representing the length of each string. The string summarization handles ghost variables representing a summary of the contents of each string, through the ASCII codes of the contained characters. These two domains are

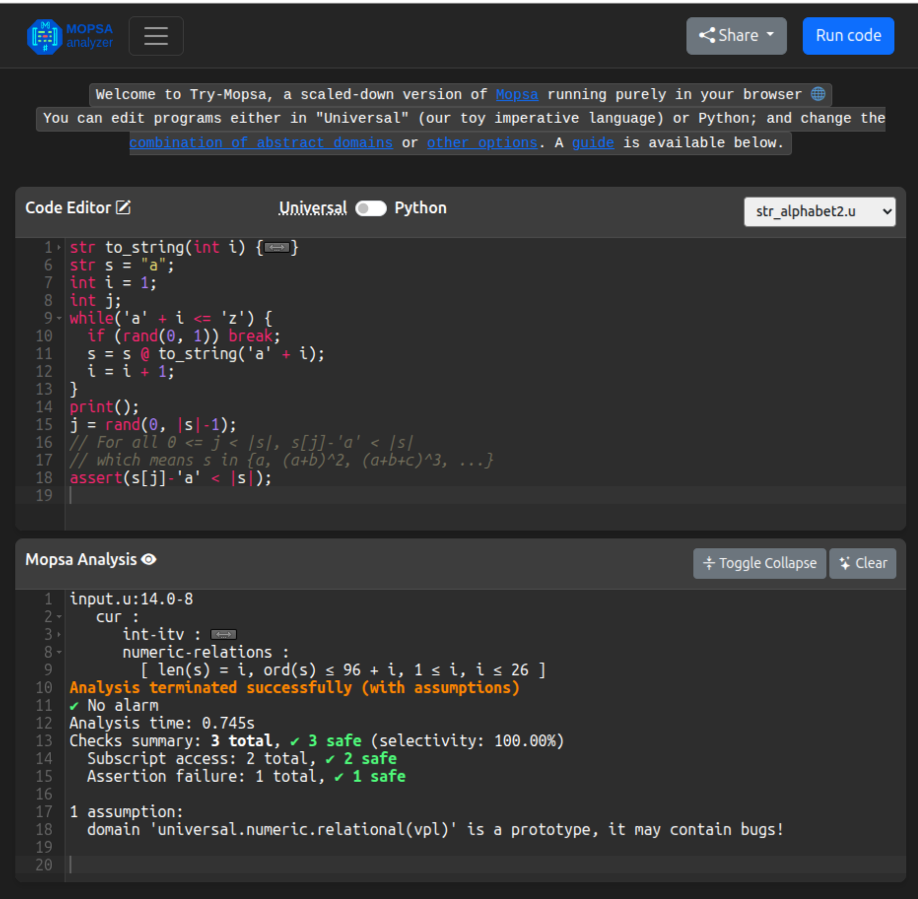

Fig. 2. Interface of Try-Mopsa: landing page, with program editor and analysis output.

```
1   str to_string(int i) { /* omitted */ }
6   str s = "a";
7   int i = 1;
8   int j;
9   while('a' + i <= 'z') {
10    if (rand(0, 1)) break;
11    s = s @ to_string('a' + i);
12    i = i + 1;
13  }
14  print();
15  j = rand(0, |s|-1);
16  // For all 0 <= j < |s|, s[j]-'a' < |s|
17  // which means s in {a, (a+b)^2, (a+b+c)^3, ...}
18  assert(s[j]-'a' < |s|);
```

Fig. 3. Program `str_alphabet2.u` used as example.

Configuration

Configuration Editor

string_product_relational.json

```
{
    "language": "universal",
    "domain": {
        "switch": [
            // Iterators
            "universal.iterators.program",
            "universal.iterators.intraproc",
            "universal.iterators.loops",
            "universal.iterators.interproc.inlining",
            "universal.iterators.unittest",
            {
                "product": [
                    "universal.toy.string_length",
                    "universal.toy.string_summarization"
                ],
                "reductions": [
                    "universal.toy.string_reduction"
                ]
            },
            // Numeric abstraction
            {
                "product": [
                    {
                        "nonrel": "universal.numeric.values.intervals.integer"
                    },
                    "universal.numeric.relational"
                ],
                "reductions": [
                    "universal.numeric.reductions.intervals_rel"
                ]
            }
        ]
    }
}
```

Fig. 4. Interface of Try-Mopsa: configuration editor, allowing to specify the choice of abstract domains and their combinators.

defined as a reduced product as they infer complementary pieces of information. Finally, a relational numerical domain is enabled. It handles the constraints passed by the aforementioned string domains, which allows it to infer relations between those different ghost variables, and program variables.

The result of analyzing `str_alphabet2.u` using `string_product_relational.json` is displayed at the bottom of Figure 2. In this case, Try-Mopsa is able to prove the complex assertion at line 18, which mixes information about the contents and the length of string `s`. Let us briefly comment on the abstract state inferred after the loop and printed at lines 1-9 of the analysis result in Figure 2. We can see that the relational domain inferred equality between the length of the string `s` and the integer variable `i`. We can further notice an interesting relation between the contents of `s` ($\mathrm{ord}(s)$) and its length: $'a' \leq \mathrm{ord}(s) \leq 'a' + \mathrm{len}(s) - 1$. This invariant has been obtained thanks to the cooperation between the string

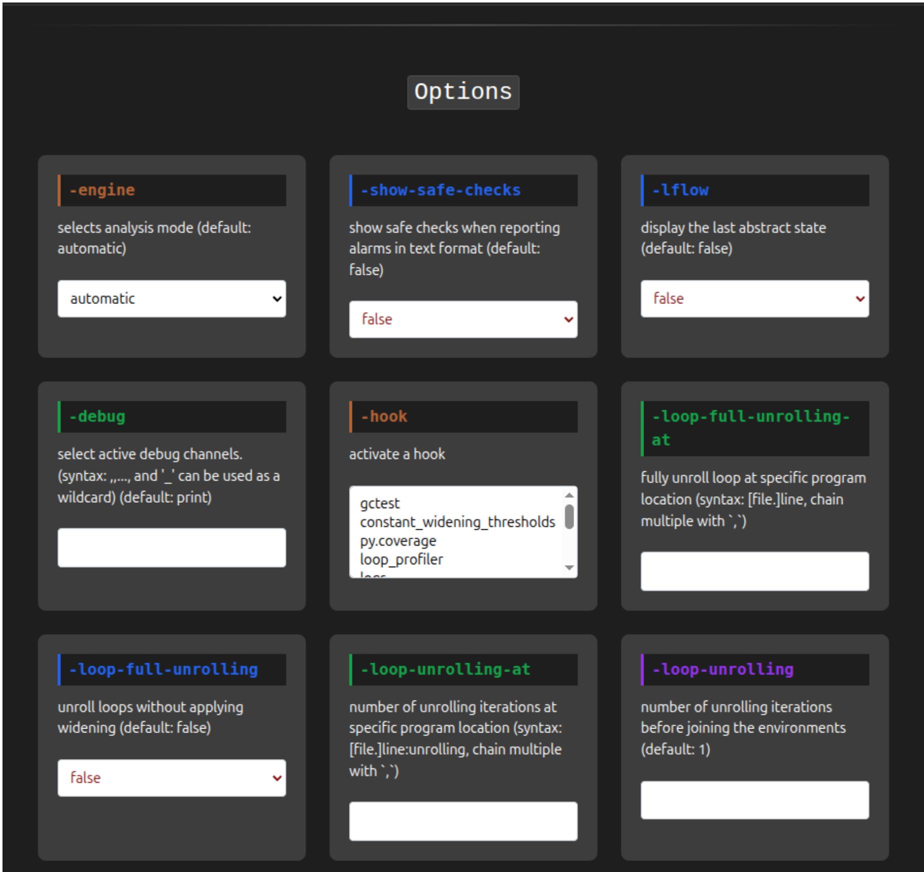

Fig. 5. Interface of Try-Mopsa: option selection.

abstract domains and the relational abstract domain. Note that the second inequality is exactly the invariant proved by Mopsa at line 18.

Figure 5 displays various analysis options to tweak the analysis. As we mentioned in the previous section, these options are dynamically generated by the toplevel, depending on which abstract domains are activated. In this example, the first five options are defined at the framework level, while the last four options customize the behavior of the loop iterator. Note that the options argument can be selected through different interfaces depending on the expected output (e.g., choice among fixed list, integer, string).

To further improve its usability, Try-Mopsa can also be used as a progressive web app, meaning that it can be installed locally on a device and run without relying on the network. In addition, a share button (in the top-right corner of Figure 2) lets users share their current program, configuration and option choices

Table 1. Comparison of analysis times with relational abstract domains enabled. The mean and standard deviation have been computed over 10 runs.

Program	Binary execution	Firefox execution
attributes.py	0.09s ± 0.00	0.52s ± 0.03
fspath.py	0.09s ± 0.00	0.49s ± 0.02
list.py	0.15s ± 0.01	0.88s ± 0.03
loop.py	0.12s ± 0.01	0.72s ± 0.03
recency.py	0.08s ± 0.01	0.63s ± 0.03
str_alphabet.u	0.04s ± 0.01	0.20s ± 0.01
str_alphabet2.u	0.23s ± 0.01	0.81s ± 0.01
str_conc_loop.u	0.08s ± 0.01	0.29s ± 0.01
str_conc_loop2.u	0.04s ± 0.01	0.15s ± 0.01

by encoding it into a URL. This can be useful for example to simplify practical sessions where no manual loading is required.

5 Implementation Discussion

This section briefly discusses the implementation size, maintainability, performance and browser compatibility of Try-Mopsa.

Implementation. The webpage, including extensions of the Ace editor for our purposes, is written in around 1,500 lines of HTML, CSS and JavaScript. The toplevel and worker of Try-Mopsa consist of 670 lines of OCaml code. The implementation within Mopsa of the VPL binder adds around 500 lines of OCaml code. Once compiled with full optimizations, the worker and all the Mopsa code relying on it are compiled into a relatively compact JavaScript file of 3.1 megabytes.

Maintainability. We argue that the implementation of Try-Mopsa is maintainable with respect to the standard implementation of Mopsa. Indeed, Try-Mopsa does not introduce breaking changes, meaning future development can be shared in the same repository.

In addition, most components of Mopsa were reused as-is, thanks in particular to the reuse of the same input-output loop as a terminal user of Mopsa would. The only major component that has been added is the support for relational abstract domains through the VPL library. This implementation provides adequate results on the toy examples we provide along with Try-Mopsa.

Performance. The goal of Try-Mopsa is to ease tool demonstrations and testing without installation, where raw analysis performance is not a priority. Nevertheless, we compared the analysis times between the natively generated executable, and the same code running in Firefox. We ran this comparison on the toy, default examples of Try-Mopsa, with configurations using relational abstract domains.

We show in Table 1 the nine most significant programs to analyze. On average, Try-Mopsa is five times slower than native binary execution. While this is a noticeable slowdown, we believe the analysis times – below one second – make Try-Mopsa still usable for demonstration or teaching purposes. Future work could focus on using JSOO's recent WebAssembly code generation features to obtain more efficient code.

Browser compatibility. We use the Playwright framework [43] to test the browser compatibility of Try-Mopsa. The tests run multiple usage scenarios and check the results are visible and as expected. These tests run during our continuous integration process and have helped identify several usability issues, particularly on mobile browsers. We currently test three desktop browsers (Chrome, Firefox, Safari), and two mobile browsers (Chrome, Safari), using different viewports corresponding to popular iPhone and Android devices, either used in portrait or in landscape orientation. There are currently 5 test scenarios, resulting in 35 tests depending on the chosen browser and viewport. Additional tests ensure that all examples programs can be analyzed in the relevant example configurations proposed in our interface. We currently provide 13 examples programs for Universal (resp. 5 for Python), and 11 configurations for Universal (resp. 6 for Python).

6 Related Work

In previous works, developers relied on server-side computations to provide web interfaces for static analyzers relying on relational numerical domains. This includes for example Interproc [15, 16], Banal [22, 25] and FuncTion [44, 45]. While client-server implementations require less work to adapt a static analyzer to the web, these implementations raise more security concerns and can be less scalable in the number of users. In addition, these implementations need to maintain a compatible server service over the years. This proves to be difficult: at the time of writing, the majority of these approaches are no longer operational.

More recently, pure client-side web interfaces have been developed for static analyzers. These scale much better with the number of concurrent users, and their "only" dependency is a web browser – one of the most widespread pieces of software. Lermusiaux and Montagu [19, 20] provide a demo web interface for their Salto analyzer of OCaml code, following previous demonstrators accompanying works around the static analysis of functional languages [35, 36]. In their BINSEC [9] tutorial at PLDI 2025, Recoules and Bardin [41] provide a web tutorial where users can choose between running BINSEC in CLI or running analysis scenarios directly in their browser. To the best of our knowledge, Try-Mopsa is the only web interface supporting the analysis of multiple languages, relational domains, and an interactive exploration of the analysis.

Negrini et al. [37] describe how they use their static analysis platform LiSA within their static analysis courses. Students are tasked with installing LiSA, and implementing several simple abstract domains. Try-Mopsa aims at providing a zero-install static analyzer relying on complex abstract domains – such as

polyhedra – to illustrate how such a tool works. Evaluating the impact of Try-Mopsa on real teaching cohorts is left as future work.

Following studies about developer use of static analyzers [5, 11], there are quite a few works discussing interfaces for static analyzers. MagpieBridge [21] aims at simplifying the display of analysis results within IDEs supporting the Language-Server Protocol. Several interfaces have also been developed to debug static analyzers [10], provide an abstract debugger [13], or mix concrete and abstract debuggers [28].

Outside of the static analysis community, Canou et al. [4] describe the technical components requiring to run an OCaml MOOC who attracted thousands of learners. It relies in particular on a pure client-side development environment avoiding installation woes students could otherwise face. Arias et al. [1] also provide a pure JavaScript version for the Rocq proof assistant, which aims at improving literate programming formatting and reducing installation burdens, in an educational context.

7 Conclusion

This article introduced Try-Mopsa, a pure, client-side implementation of Mopsa running in the browser or as a progressive web app. Try-Mopsa supports all the main core components of Mopsa, including support for relational numerical domains and its abstract debugger. Try-Mopsa is designed to work on a wide range of devices, from smartphones to desktop, thanks to a responsive design and continuous testing of the support of popular browsers and various resolutions. We envision Try-Mopsa to be a convenient tool to demonstrate static analysis capabilities, for teaching purposes or more generally to attract new users.

Acknowledgments. We are grateful to Aymane Ismail for prototyping an early version of Try-Mopsa; and to Alexandre Maréchal for his help in using the VPL. We thank Aymeric Fromherz, Louis Gesbert, Vincent Botbol, Louis Rustenholz, Antoine Miné, Abdelraouf Ouadjaout and the whole Mopsa team for their helpful discussion and comments on this work.

Bibliography

[1] Emilio Jesús Gallego Arias, Benoît Pin, and Pierre Jouvelot. "jsCoq: Towards Hybrid Theorem Proving Interfaces". In: *UITP*. 2016. DOI: 10.4204/EPTCS.239.2.

[2] Dirk Beyer. "Competition on Software Verification and Witness Validation: SV-COMP 2023". In: *TACAS*. 2023. DOI: 10.1007/978-3-031-30820-8_29.

[3] Sylvain Boulmé, Alexandre Maréchal, David Monniaux, Michaël Périn, and Hang Yu. "The Verified Polyhedron Library: an Overview". In: *SYNASC*. 2018. DOI: 10.1109/SYNASC.2018.00014.

[4] Benjamin Canou, Roberto Di Cosmo, and Grégoire Henry. “Scaling up functional programming education: under the hood of the OCaml MOOC”. In: *Proc. ACM Program. Lang.* ICFP (2017). DOI: 10.1145/3110248.

[5] Maria Christakis and Christian Bird. “What developers want and need from program analysis: an empirical study”. In: *ASE*.

[6] Patrick Cousot and Radhia Cousot. “Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints”. In: *POPL*. 1977. DOI: 10.1145/512950.512973.

[7] Patrick Cousot and Nicolas Halbwachs. “Automatic Discovery of Linear Restraints Among Variables of a Program”. In: *POPL*. 1978. DOI: 10.1145/512760.512770.

[8] Albert Danial. *cloc: v2.06*. 2025. URL: https://github.com/AlDanial/cloc/.

[9] Adel Djoudi and Sébastien Bardin. “BINSEC: Binary Code Analysis with Low-Level Regions”. In: *TACAS*. 2015. DOI: 10.1007/978-3-662-46681-0_17.

[10] Lisa Nguyen Quang Do, Stefan Krüger, Patrick Hill, Karim Ali, and Eric Bodden. “Debugging Static Analysis”. In: *IEEE Trans. Software Eng.* 7 (2020). DOI: 10.1109/TSE.2018.2868349.

[11] Lisa Nguyen Quang Do, James R. Wright, and Karim Ali. “Why Do Software Developers Use Static Analysis Tools? A User-Centered Study of Developer Needs and Motivations”. In: *IEEE Trans. Software Eng.* 3 (2022). DOI: 10.1109/TSE.2020.3004525.

[12] Denis Gopan, Frank DiMaio, Nurit Dor, Thomas W. Reps, and Shmuel Sagiv. “Numeric Domains with Summarized Dimensions”. In: *TACAS*. 2004. DOI: 10.1007/978-3-540-24730-2_38.

[13] Karoliine Holter, Juhan Oskar Hennoste, Patrick Lam, Simmo Saan, and Vesal Vojdani. “Abstract Debuggers: Exploring Program Behaviors using Static Analysis Results”. In: *Onward!* 2024. DOI: 10.1145/3689492.3690053.

[14] Jane Street Group, LLC. *JavaScript stubs for the Zarith Library*. Version 0.17.0. July 2024. URL: https://github.com/janestreet/zarith_stubs_js.

[15] Bertrand Jeannet. “Relational interprocedural verification of concurrent programs”. In: *Softw. Syst. Model.* 2 (2013). DOI: 10.1007/S10270-012-0230-7.

[16] Bertrand Jeannet. *Web Interface for the Interproc Analyzer*. 2009. URL: https://pop-art.inrialpes.fr/interproc/interprocweb.cgi.html.

[17] Bertrand Jeannet and Antoine Miné. “Apron: A Library of Numerical Abstract Domains for Static Analysis”. In: *CAV*. 2009. DOI: 10.1007/978-3-642-02658-4_52.

[18] Matthieu Journault, Antoine Miné, Raphaël Monat, and Abdelraouf Ouadjaout. “Combinations of Reusable Abstract Domains for a Multilingual Static Analyzer”. In: *VSTTE*. 2019. DOI: 10.1007/978-3-030-41600-3_1.

[19] Pierre Lermusiaux and Benoît Montagu. “Detection of Uncaught Exceptions in Functional Programs by Abstract Interpretation”. In: *ESOP*. 2024. DOI: 10.1007/978-3-031-57267-8_15.

[20] Pierre Lermusiaux and Benoît Montagu. *Web Demo for the Salto Analyzer*. 2025. URL: https://salto.gitlabpages.inria.fr/demo/salto.html.

[21] Linghui Luo, Julian Dolby, and Eric Bodden. “MagpieBridge: A General Approach to Integrating Static Analyses into IDEs and Editors (Tool Insights Paper)”. In: *ECOOP*. 2019. DOI: 10.4230/LIPICS.ECOOP.2019.21.

[22] Antoine Miné. “Inferring Sufficient Conditions with Backward Polyhedral Under-Approximations”. In: *NSAD@SAS*. 2012. DOI: 10.1016/J.ENTCS.2012.09.009.

[23] Antoine Miné. “Relational Abstract Domains for the Detection of Floating-Point Run-Time Errors”. In: *ESOP*. 2004. DOI: 10.1007/978-3-540-24725-8_2.

[24] Antoine Miné. “The octagon abstract domain”. In: *High. Order Symb. Comput.* 1 (2006). DOI: 10.1007/S10990-006-8609-1.

[25] Antoine Miné. *Web Interface for Sufficient Condition Polyhedral Prototype Analyzer*. 2012. URL: https://mine.perso.lip6.fr/banal/.

[26] Antoine Miné, Xavier Leroy, and Pascal Cuoq. *Zarith: arithmetic and logical operations over arbitrary-precision integers*. Version 1.14. July 2024. URL: https://github.com/ocaml/Zarith.

[27] Alex Mojaki. *sync-message*. Version 0.0.12. Oct. 2023. URL: https://github.com/alexmojaki/sync-message.

[28] Mats Van Molle, Bram Vandenbogaerde, and Coen De Roover. “Cross-Level Debugging for Static Analysers”. In: *SLE*. 2023. DOI: 10.1145/3623476.3623512.

[29] Raphaël Monat. “Static Type and Value Analysis by Abstract Interpretation of Python Programs with Native C Libraries”. PhD thesis. Sorbonne Université, France, 2021.

[30] Raphaël Monat. *Try-Mopsa*. 2025. URL: https://try-mopsa.rmonat.fr.

[31] Raphaël Monat, Abdelraouf Ouadjaout, and Antoine Miné. “A Multilanguage Static Analysis of Python Programs with Native C Extensions”. In: *SAS*. 2021. DOI: 10.1007/978-3-030-88806-0_16.

[32] Raphaël Monat, Abdelraouf Ouadjaout, and Antoine Miné. “Easing maintenance of academic static analyzers”. In: *Int. J. Softw. Tools Technol. Transf.* 6 (2024). DOI: 10.1007/S10009-024-00770-1.

[33] Raphaël Monat, Abdelraouf Ouadjaout, and Antoine Miné. “Mopsa-C: Modular Domains and Relational Abstract Interpretation for C Programs (Competition Contribution)”. In: *TACAS*. 2023. DOI: 10.1007/978-3-031-30820-8_37.

[34] Raphaël Monat, Abdelraouf Ouadjaout, and Antoine Miné. “Static Type Analysis by Abstract Interpretation of Python Programs”. In: *ECOOP*. 2020. DOI: 10.4230/LIPICS.ECOOP.2020.17.

[35] Benoît Montagu and Thomas P. Jensen. "Stable relations and abstract interpretation of higher-order programs". In: *Proc. ACM Program. Lang.* ICFP (2020). DOI: 10.1145/3409001.

[36] Benoît Montagu and Thomas P. Jensen. "Trace-based control-flow analysis". In: *PLDI*. 2021. DOI: 10.1145/3453483.3454057.

[37] Luca Negrini, Vincenzo Arceri, Luca Olivieri, Agostino Cortesi, and Pietro Ferrara. "Teaching Through Practice: Advanced Static Analysis with LiSA". In: *FMTea*. 2024. DOI: 10.1007/978-3-031-71379-8_3.

[38] OCamlPro. *Ezjs_ace: bindings for the Ace editor*. Version 0.1.1. Oct. 2020. URL: https://github.com/ocamlpro/ezjs_ace.

[39] Abdelraouf Ouadjaout and Antoine Miné. "A Library Modeling Language for the Static Analysis of C Programs". In: *SAS*. 2020. DOI: 10.1007/978-3-030-65474-0_11.

[40] François Pottier and Yann Régis-Gianas. *Menhir: an LR(1) parser generator for OCaml*. Version 20250903. Sept. 2025. URL: https://gitlab.inria.fr/fpottier/menhir.

[41] Frédéric Recoules and Sébastien Bardin. *BINSEC Tutorial at PLDI'25: Adapting Symbolic Execution for Binary-level Security*. 2025. URL: https://binsec.github.io/tutorial-pldi2025/.

[42] The Ace Developers. *Ace (Ajax.org Cloud9 Editor)*. Version 1.43.3. Sept. 2025. URL: https://github.com/ajaxorg/ace.

[43] The Playwright Development Team. *Playwright: framework for Web Testing and Automation*. Version 1.55.0. Sept. 8, 2025. URL: https://github.com/microsoft/playwright.

[44] Caterina Urban. "FuncTion: An Abstract Domain Functor for Termination - (Competition Contribution)". In: *TACAS*. 2015. DOI: 10.1007/978-3-662-46681-0_46.

[45] Caterina Urban. *Web Interface for FuncTion*. 2015. URL: https://www.di.ens.fr/~urban/FuncTion.html.

[46] Jérôme Vouillon and Vincent Balat. "From bytecode to JavaScript: the Js_of_ocaml compiler". In: *Softw. Pract. Exp.* 8 (2014). DOI: 10.1002/SPE.2187.

This work is licensed under a Creative Commons "Attribution 4.0 International" license.