Design of a Modular Platform for Static Analysis

Antoine Miné
antoine.mine@lip6.fr
Laboratoire d'Informatique de Paris 6
Sorbonne Université, CNRS
Paris, France

Abdelraouf Ouadjaout
abdelraouf.ouadjaout@lip6.fr
Laboratoire d'Informatique de Paris 6
Sorbonne Université, CNRS
Paris, France

Matthieu Journault
matthieu.journault@lip6.fr
Laboratoire d'Informatique de Paris 6
Sorbonne Université, CNRS
Paris, France

Abstract

We present the design and implementation of Mopsa, a platform that simplifies the construction of semantic static analyzers by abstract interpretation. Mopsa computes sound program invariants and reports run-time errors, undefined behaviors, and uncaught exceptions. Mopsa differs from existing platforms by its highly modular and extensible design: semantic abstractions of numeric values, pointers, objects, control, as well as syntax-driven iterators, are defined in small, reusable domains with loose coupling, that can be combined and reused to a greater extent than in previous work. Moreover, Mopsa aims at supporting several languages (currently, subsets of both C and Python) while sharing abstraction components as much as possible. Mopsa is a work in progress, and not yet capable of analyzing full programs; nevertheless, we report early experimental results on verification benchmarks.

Keywords Static analysis, Abstract interpretation, Tool

Acknowledgments

This work is partially supported by the European Research Council under Consolidator Grant Agreement 681393 – Mopsa.

1 Introduction

Abstract interpretation [5], a theory of the approximation of semantics, allows designing sound static analyzers that infer automatically program properties. It has enjoyed a growing success, as witnessed by commercial tools used in the industry: Polyspace Verifier, Astrée [12], Sparrow [15], Julia [16], etc. Open-source industrial-strength frameworks to design analyzers by abstract interpretation have also been proposed: Frama-C [8], IKOS [3], Infer [4], etc. Most tools focus on analyzing families of C-like or Java-like languages, and do not target dynamic languages (e.g., Python, JavaScript) and *a forciori* cannot handle both kinds of languages. Moreover, while all tools are based on a modular combination of abstractions, as advocated by abstract interpretation [6], these are often restricted to specific abstractions: e.g., Astrée [7] achieves a reduced product of numeric abstractions but has a monolithic memory abstraction; Frama-C's abstractions can be composed either through coarse-grain plug-ins, or as non-relational value abstractions [2]. To try and address these issues, and push abstract interpretation frameworks further, we have started the design of a *Modular Open Platform for Static Analysis* (Mopsa) in OCaml. It is based on a standard interpreter by induction on the syntax and a collection of abstractions (see, e.g., [1]) but with unique features:

— all domains share a common extensible Abstract Syntax Tree datatype; it can express both high-level syntax close to the source languages, and simpler intermediate ones used internally by abstractions (e.g., numeric fragments);
— we eschew static simplifications common to frameworks (e.g., LLVM [13]); transformations are performed dynamically during interpretation and benefit from inferred facts;
— all domains share a common interface and are easy to compose; notably, iterators are viewed as domains, and domains are responsible for statement simplification;
— domains need only handle a fragment of the AST; they can be reused after extending the AST to new languages.

Mopsa is a work in progress: its architectural foundations are implemented, but it is not ready to analyze realistic programs. As a proof of concept, we have implemented analyses for both a fragment of C and Python, showing it can support legacy abstractions (e.g., numeric domains, low-level C memory models [14]) and is a great help developing novel ones (e.g., modular C analysis [11], Python analysis [9]). Sections 2 and 3 present the domain signature and domain composition; Sect. 4 reports on early experiments (analyzing part of Juliet for C and Python's regression tests); Sect. 5 concludes.

2 Unified Domains Signature

Abstract domains are OCaml modules implementing the `DOMAIN` signature in Fig. 1. In the following, we describe the main types and functions, and state the core design goals.

Lattice. Each domain defines a type `t` characterizing its abstract state, as well as corresponding lattice operators (`join`, `widening`, etc.). In Mopsa, the internal abstract state of a domain is private: other domains have no *direct* access. This separation ensures a low coupling between domains.

Managers. During the analysis of one statement, a domain may require computing the post-condition of statements handled by other domains. To allow inter-domain communication without sacrificing modularity, each domain transfer function is not defined over its private type `t`, but on the product of types from all domains. To ensure that each domain can be programmed independently from the others, the

TAPAS'18, August 28, 2018, Freiburg im Breisgau, Germany
2018.

```
module type DOMAIN = sig
  (* Lattice definition *)
  type t
  val bottom   : t
  val top      : t
  val leq      : t -> t -> bool
  val join     : t -> t -> t
  val meet     : t -> t -> t
  val widening : t -> t -> t
  (* Transfer functions *)
  val exec : stmt -> ('a, t) man -> 'a flow -> 'a post option
  val eval : expr -> ('a, t) man -> 'a flow -> 'a evals option
  val ask  : 'r query -> ('a, t) man -> 'a flow -> 'r option
  (* Interface *)
  val exec_interface : zone interface
  val eval_interface : (zone * zone) interface
end
```

Figure 1. Unified signature for abstract domains.

product is a type parameter `'a`, and we use an encoding of polymorphic records in OCaml: the manager `('a, t) man` is a record providing lattice operators on `'a`, transfer functions over all domains in `'a`, and accessor functions `get: 'a -> t` and `set: t -> 'a -> 'a` to allow a domain to access and update its private abstraction within the global abstraction.

Flows. Mopsa operates by induction on the syntax of the program. To handle non-local control flows, we use continuations (similarly to Astrée [1]): we collect not only environments reaching the current program location, but also those at previously encountered jump locations. Suspended flows are merged back into the current flow when reaching the corresponding jump target. Expressing flows as continuations makes it possible to abstract very complex non-local control, such as generators in Python [9]. Flow continuations are implemented as maps from *flow tokens* into the global abstraction `'a`. Tokens belong to an extensible type, thus allowing a domain to add new control flow abstractions independently from the remaining domains.

Example 2.1. We give in Fig. 2 an example of a transfer function for handling forward `goto` statements in C. Flows are enriched with a new token `T_goto` annotated with the target label. When reaching a statement `goto label`, the current environments are moved to token `T_goto label`, before being reset to $\perp$ because the next control location becomes unreachable. Encountering `label: stmt` merges environments associated to `T_goto label` into current environments, and returns a post-condition of `stmt` via the manager. The default case returns `None`, indicating that other domains should handle the other constructions. Such a pure iterator domain does not maintain any abstract state by itself, hence `t = unit`. Despite its brevity, the code in Fig. 2 is the complete transfer function of a domain that adds C-style

```
type token += T_goto of Ast.label

let exec stmt man flow =
  match stmt with
  | S_c_goto(label) -> (* Case of  S♯⟦goto label⟧ *)
    let cur_env = Flow.find T_cur man flow in
    let flow' = Flow.add (T_goto label) cur_env man flow in
    let flow'' = Flow.set T_cur man.bottom man flow' in
    Post.return flow''
  | S_c_label(label, stmt) -> (* Case of  S♯⟦label: stmt⟧ *)
    let goto_env = Flow.find (T_goto label) man flow in
    let flow' = Flow.add T_cur goto_env man flow in
    let flow'' = man.exec stmt flow' in
    Post.return flow''
  | _ -> None
```

Figure 2. Transfer function for forward `goto` statements.

forward `goto` to an analysis. It is completely decoupled from other domains, and thus highly reusable.

Evaluations. Domains in Mopsa can implement dynamic rewriting rules to simplify expressions by exploiting the available abstract information. A classic use is transforming modular arithmetic from C expressions into mathematical arithmetic, easier to handle in relational numeric domains, after ensuring the absence of overflows in the current state.

After a statement or expression is transformed by one domain, it is executed again *via* the manager on all existing domains. We must take care to avoid infinite rewriting loops. Mopsa introduces the notion of *zones* to avoid this problem. Zones define (possibly overlapping) AST fragments, such as expressions with modular arithmetic or purely mathematical expressions. When adding a new language, the extensible type `Zone.t` of zones can be enriched along with the extensible type of statements and expressions. Calls to transfer functions and rewriting tag input and output AST with zone information, and the manager routes the calls accordingly to relevant domains. It is then easy to ensure that abstract computations and rewriting progress until completion.

Another contribution of Mopsa is the ability to handle natively disjunctive evaluations. Domains can return different results for different parts of the input abstract environment. Consider the following example from Python analysis:

Example 2.2. Fig. 3 shows the evaluation of an index access on Python lists. We assume that lists are abstracted with summary and length variables over-approximating the contents and the number of elements, as done in [9]. When analyzing a read access, the function `getitem_cases` is used to partition the pre-condition w.r.t. the length. The first flow `flow1` represents the case when index `i` is outside the size of `self`. The associated evaluation is an empty expression since an exception is raised. The second flow `flow2` is obtained by filtering the pre-condition verifying that the list contains exactly one element. In this case, the summary variable is

```
(* Case of  E♯⟦list.__getitem__(self, i)⟧ *)
let flow1, flow2, flow3 = getitem_cases self i man flow in
let evl1 =
  let flow' = man.exec (mk_raise "IndexError") flow1 in
  Eval.empty flow'
in
let evl2 = Eval.singleton (summary self) flow2 in
let evl3 =
  let tmp = mk_tmp () in
  let flow' = man.exec (expand tmp (summary self)) flow3 in
  Eval.singleton tmp flow' ~cleaner:[remove_var tmp]
in
Eval.or_list [evl1; evl2; evl3]
```

Figure 3. Evaluation of index access on lists in Python.

returned as a sound simplification of index access. Finally, when the list contains at least two elements, a temporary variable is created as a copy of the summary variable using the expand function, similarly to the case of summarized arrays [10]. With the expression, a cleaning function is returned that will be called to remove the temporary variable once all rewritings are done.

Queries. When a domain needs specific information maintained by another domain, it can fetch them via *queries*, similarly to Astrée [7]. The type `_ query` is an extensible public GADT type that can be enriched by domains. For example, an interval query can be defined by:

```
type _ query += Q_interval: expr -> Itv.t query
```

Numeric domains handle requests to `Q_interval` in the transfer function `ask`, and client domains retrieve this information via their manager by calling `man.ask (Q_interval e) flow`. GADT typing ensures that the returned value has type `Itv.t`.

3 Domain Composition

The global abstraction of an analysis instance is constructed at run-time by composing domains according to a user-given configuration file. An illustrative example is depicted in Fig. 4 representing a simplified version of an analysis of C. Three types of generic composers are used to connect leaf domains:

Iterators. This composer iterates sequentially over a set of domains, which is materialized with ↠ in Fig. 4. Lattice operators are defined pointwise, and transfer functions are called in sequence until one succeeds: domain i is invoked only if previous domains $j < i$ returned `None`. This composition is useful to combine a set of iterators defining transfer functions for disjoint parts of the AST, such as loops, function calls, etc. In addition, it allows plugging new iteration schemes easily, e.g., replacing the induction-based iterators with CFG-based ones.

Reduced products. The classic method employed in most static analyzers (e.g., [1]) for creating reduced products is via cascading binary functors. Mopsa provides instead a generic n-ary reduced product composer that enables better propagation of reduction channels. Lattice operators are defined pointwise and transfer functions of argument domains are called in parallel. All post-conditions are given to a user-defined reduction operator, that can access directly abstract elements of any domain in the pool and can exploit the published reduction channels to refine the final post-condition.

Stacks. Another novelty in Mopsa is a stacked reduced product of n functors sharing the same subordinate domain instance. This form of composition is useful when several domains depend on an external domain to delegate management of parts of their abstract state. Consider the case of cell and smashing abstractions from [1, 14] in Fig. 4. Both domains need a pointer domain for address resolution and a numeric domain for abstracting numeric environments. By sharing the same numeric domain instance, relations between cells and smashed variables can be inferred.

4 Experiments

We instantiated two analyzers, one for C (Fig. 4) and one for Python (Fig. 5). We note that the configurations share several domains (underlined), including iterators, heap and numeric abstractions. C benchmarks were taken from the Juliet Test Suite (v 1.3) for C/C++. We selected all the C tests corresponding to the following categories: `CWE190` on integer overflows, `CWE369` on divisions by zero, and `CWE476` on null pointer dereferences. Each category provides several test cases, themselves split in two programs: a bad program containing an error, and its corrected version. Python benchmarks were performed on regression tests from the official Python 3.6.3 distribution, that test the builtins of the language and the standard library. The current configuration for the Python analysis does not provide an analyzer able to handle all the 500 regression tests; we selected 9 tests in the scope of our configuration (see [9] for more information). Tables 1 and 2 provide results of these experimentations: ✓ are good programs for which no alarm was raised, ✓ are bad programs for which an alarm was raised, ✗ are good programs for which the analyzer raised an alarm (false positive), ✱ are programs that could not be handled by the analyzer (e.g., by lack of stubs). False positives in `CWE369` are due to the use of a non-partitioned numerical domain for the analysis of programs testing divisions by zero; thanks to the modularity of the analyzer, this could be avoided by adding a partitioning domain.

5 Conclusion and Future Work

We have presented the design principles of Mopsa, a new platform for static analysis development through compositions of highly reusable abstractions. It is not yet feature-complete, but it is already successfully used in research projects to support the analysis of non-trivial C and Python

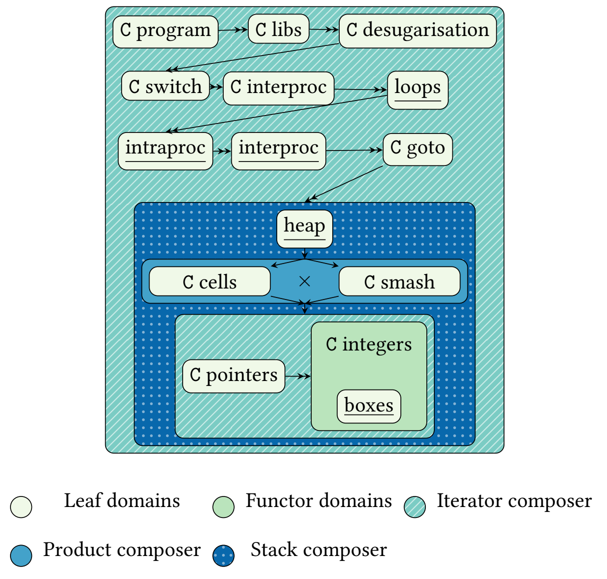

Figure 4. C analysis configuration.

Table 1. Benchmarks on regression tests of Python 3.6.

Regression test	Lines	Tests	Time	✓	✗	✱	Coverage
`test_augassign`	273	7	645ms	4	2	1	85.71%
`test_baseexception`	141	10	20ms	6	0	4	60.00%
`test_bool`	294	26	47ms	12	0	14	46.15%
`test_builtin`	454	21	360ms	3	0	18	14.29%
`test_contains`	77	4	418ms	1	0	3	25.00%
`test_int_literal`	91	6	29ms	6	0	0	100.00%
`test_int`	218	8	88ms	3	0	5	37.50%
`test_list`	106	9	88ms	3	0	6	33.33%
`test_unary`	39	6	11ms	2	0	4	33.33%
Total	1693	97	1.71s	40	2	55	43.30%

codes [9, 11]. In the future, we plan to extend the support for C standard libraries and Python built-ins to be able to analyze realistic codes. Other research projects on Mopsa include static analysis of novel properties beyond safety, such as a portability analysis. We plan to release Mopsa as open-source.

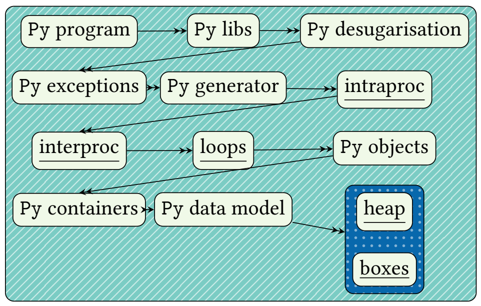

Figure 5. Python analysis configuration.

Table 2. Benchmarks on CWE from the Juliet test suite.

tests	Loc	Tests	Time	✓	✓	✗	✱	Coverage
`CWE190`	440k	6840	11.0mn	2584	3213	0	1043	84.75%
`CWE369`	109k	1368	3.30mn	76	380	304	608	55.55%
`CWE476`	25k	522	0,207mn	270	252	0	0	100%
Total	574k	8730	14.5mn	2930	3845	304	1651	77.60%

References

[1] J. Bertrane, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, and X. Rival. 2010. Static Analysis and Verification of Aerospace Software by Abstract Interpretation. In *AIAA Infotech@Aerospace*. AIAA, 1–38.

[2] S, Blazy, D. Bühler, and B. Yakobowski. 2017. Structuring Abstract Interpreters Through State and Value Abstractions. In *Verification, Model Checking, and Abstract Interpretation*. Springer, 112–130.

[3] G. Brat, J. A. Navas, N. Shi, and A. Venet. 2014. IKOS: A Framework for Static Analysis Based on Abstract Interpretation. In *Software Engineering and Formal Methods*. Springer, 271–277.

[4] C. Calcagno, D. Distefano, J. Dubreil, D. Gabi, P. Hooimeijer, M. Luca, P. O'Hearn, I. Papakonstantinou, J. Purbrick, and D. Rodriguez. 2015. Moving Fast with Software Verification. In *NFM*. Springer, 3–11.

[5] P. Cousot and R. Cousot. 1977. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Proc. of POPL'77*. ACM, 238–252.

[6] P. Cousot and R. Cousot. 1979. Systematic design of program analysis frameworks. In *Proc. of POPL'79*. ACM Press, 269–282.

[7] P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. 2006. Combination of abstractions in the Astrée static analyzer. In *Proc. of ASIAN'06 (LNCS)*, Vol. 4435. Springer, 272–300.

[8] P. Cuoq, F. Kirchner, N. Kosmatov, V. Prevosto, J. Signoles, and B. Yakobowski. 2012. Frama-C: A software analysis perspective. *Formal Aspects of Computing* 27 (2012), 573–609.

[9] A. Fromherz, A. Ouadjaout, and A. Miné. 2018. Static value analysis of Python programs by abstract interpretation. In *Proc. of NFM'18 (LNCS)*. Springer, 185–202.

[10] D. Gopan, F. DiMaio, N. Dor, T. Reps, and M. Sagiv. 2004. Numeric Domains with Summarized Dimensions. In *Proc. of TACAS'04 (LNCS)*, Vol. 2988. Springer, 512–529.

[11] M. Journault, A. Ouadjaout, and A. Miné. 2018. Modular static analysis of string manipulations in C programs. In *Proc. of SAS'18 (LNCS)*.

[12] D. Kästner, S. Wilhelm, S. Nenova, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, and X. Rival. 2010. Astrée: Proving the absence of runtime errors. In *Proc. of ERTS2 2010*.

[13] C. Lattner and V. Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proc. of CGO'04*.

[14] A. Miné. 2006. Field-sensitive value analysis of embedded C programs with union types and pointer arithmetics. In *Proc. of LCTES'06*. ACM, 54–63.

[15] H. Oh, K. Heo, W. Lee, W. Lee, and K. Yi. 2012. Design and Implementation of Sparse Global Analyses for C-like Languages. *SIGPLAN Not.* 47, 6 (June 2012), 229–238.

[16] F. Spoto. 2005. JULIA: A Generic Static Analyser for the Java Bytecode. In *Proc. of FTfJP'2005*. 17.