Speeding up Static Analysis with the Split Operator

Vincenzo Arceri
vincenzo.arceri@unipr.it
University of Parma
Parma, Italy

Greta Dolcetti
greta.dolcetti@studenti.unipr.it
University of Parma
Parma, Italy

Enea Zaffanella
enea.zaffanella@unipr.it
University of Parma
Parma, Italy

Abstract

In the context of static analysis based on Abstract Interpretation, we propose a new abstract operator modeling the *split* of control flow paths: the goal of the operator is to enable a more efficient analysis when using abstract domains that are computationally expensive, having no effect on precision. Focusing on the case of conditional branches guarded by numeric linear constraints, we provide a preliminary experimental evaluation showing that, by using the split operator, we can achieve significant efficiency improvements for a static analysis based on the domain of convex polyhedra. We also briefly discuss the applicability of this new operator to different, possibly non-numeric abstract domains.

CCS Concepts: • **Theory of computation → Program analysis**; • **Software and its engineering → Automated static analysis**.

Keywords: Abstract Interpretation, Static Analysis, Abstract Operators

ACM Reference Format:
Vincenzo Arceri, Greta Dolcetti, and Enea Zaffanella. 2023. Speeding up Static Analysis with the Split Operator. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

In static analysis tools based on Abstract Interpretation (AI) it is common to define a neat separation between the AI framework and the abstract domains: the two components communicate by invoking abstract domain operators, such as meet, join, widening and narrowing, whose results are combined to correctly characterize the abstract semantics of the program. Among the operations needed during static analysis, one allows to *filter* an abstract element $A \in \mathbb{A}$ over a given predicate $p \in \text{Pred}$, returning a refined abstract element A_p approximating those states of A that satisfy p:

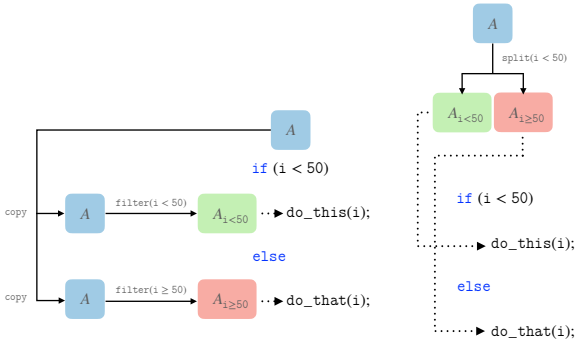
$$\text{filter} : \mathbb{A} \times \text{Pred} \to \mathbb{A}.$$

This operator goes by different names depending on the context. Abstract domains adopt a rather specific terminology, which often highlights the kind of predicates that are supported; for instance, the numeric domains in Apron [18] provide `meet_lincons_array` for filtering on a set of linear constraints; similarly, we can find `add_cons` for the polyhedra domains in the PPLite library [8] and `add_congruences` for the grid domain [2] in the PPL library [4]. Static analysis tools usually adopt a more generic name, sometimes promoting the operator to an abstract instruction in their program representation: for instance, we have `Comparison` in IKOS AR [9], `S_assume` in MOPSA [20] and `assume` in both Clam/Crab [16] and LiSA [12]. The filter operator can be used to enforce properties that are known to hold at a specific program point: for instance, after a call to a mock library function, we can filter the abstract state on the post-condition of the function. The same operator is also commonly used to model the conditional split of the control flow of the program. For instance, the code fragment

```
if (i < 50) do_this(i); else do_that(i);
```

can be modeled as shown in Figure 1a: the input abstract element A is cloned to obtain two copies; these are then filtered on the predicate $p = (i < 50)$ for the `then` branch and on its complement $\neg p = (i \geq 50)$ for the `else` branch.

This implementation approach is both correct and precise, but in some contexts could incur avoidable inefficiencies, as it



(a) Using filter operator **(b)** Using split operator

Figure 1. Abstract modeling of a conditional branch.

Conference'17, July 2017, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

replicates most of the work done to evaluate the complementary predicates in the two program branches. The overhead is probably negligible, hence acceptable, as long as considering the most efficient abstract domains, such as the domain of intervals [10]. Things might be different when adopting more expensive and precise abstract domains, such as the domain of convex polyhedra [11]; in these cases, it might be worth exploring alternative implementation strategies to better preserve efficiency. In particular, in Figure 1b we show an alternative approach where the conditional branch is modeled using the *split* operator:

$$\text{split}\colon \mathbb{A} \times \text{Pred} \to \mathbb{A} \times \mathbb{A}.$$

This new abstract domain operator, initially proposed in [7], filters the input abstract element on a predicate p and on its complement $\neg p$ *at the same time*, allowing for the factorization of any replicated computational effort. Note that, if no optimization is possible, the abstract domain can just resort to the default implementation, which clones the abstract element and invokes (twice) the filter operator.

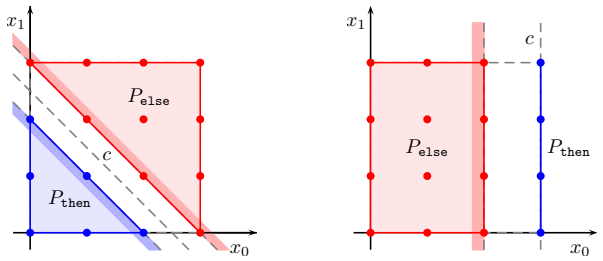
While this idea is both simple and intuitively promising, to the best of our knowledge its effectiveness has never been evaluated in the context of classical AI-based static program analysis:[1] the goal of the current work is to provide such an evaluation. To this end, we first describe the design changes that are required, both at the interface and at the implementation levels, in order to accomodate the new split operator into an existing static analysis tool. Then, we show the results of our preliminary experiments, which focus on splits defined on numerical predicates and on the abstract domain of convex polyhedra.

In principle, the high level specification of the split operator allows for a general adoption, independently from the considered abstract domain. However, since its end goal is to enable efficiency improvements (leaving correctness and precision unaffected), its actual effectiveness necessarily depends on *profitability* considerations (e.g., the computational cost of filtering on a specific class of predicates and the frequency of these operations in a typical analysis). Hence, we will also briefly discuss the applicability of the approach to more general contexts.

2 Splitting Polyhedra on Linear Constraints

In this section we briefly discuss how a split operator defined on a numerical predicate can be efficiently implemented on the domain of convex polyhedra [11]. For space reasons, we will only *hint* at the optimizations, without a proper explanation of their details: these are available in release 0.10.1 of the open source library PPLite [5, 6, 8].

The split operator proposed in [7] for the domain of convex polyhedra focuses on the case of *rational splits*: since in this case the constraints added to the `then` and `else` branches are complementary, the new abstract operator can factorize most of the work, leading to a rather systematic halving of the computational effort.[2] However, often the predicate guarding a conditional branch is defined on variables having an integral datatype: in this case the use of a rational split produces a safe result, but typically incurs a precision loss. Therefore, we now focus on *integral splits*. Consider first the case of a split based on a (strict or non-strict) linear inequality constraint.



(a) Split on $c \equiv (2x_0 + 2x_1 \leq 5)$ **(b)** Split on $c \equiv (x_0 = 3)$

Figure 2. Examples of integral splits.

Example 2.1. Let $P \equiv \{0 \leq x_0 \leq 3, 0 \leq x_1 \leq 3\}$ be a polyhedron in $\mathbb{R}^2$ and consider a conditional branch guarded by constraint $c \equiv (2x_0 + 2x_1 \leq 5)$. If the variables are known to be integral, this guard can be refined to $c_{\texttt{then}} \equiv (x_0 + x_1 \leq 2)$; the corresponding integral complement constraint is $c_{\texttt{else}} \equiv (x_0 + x_1 \geq 3)$. Hence, we obtain $P_{\texttt{then}} \equiv \{0 \leq x_0, 0 \leq x_1, x_0 + x_1 \leq 2\}$ and $P_{\texttt{else}} \equiv \{x_0 \leq 3, x_1 \leq 3, x_0 + x_1 \geq 3\}$, shown in Figure 2a.

The two constraints $c_{\texttt{then}}$ and $c_{\texttt{else}}$ are not complementary when considered in the real relaxation $\mathbb{R}^2$ and hence the optimized implementation of the integral split operator cannot be as effective as that for the rational case. However, since the two constraints have the same slope, the implementation can still factor out most of the (uselessly duplicated) work done when computing the scalar products of each generator of P with the two constraints, reducing the overall computational cost. Note that the described integral refinement process is generally incomplete: there are cases where one of the branches has no integral solution, but the abstract domain is unable to detect it due to the real relaxation (unless performing further expensive checks).

We now consider the case of an integral split based on a linear (dis-) equality constraint. This case turns out to be trickier, since the domain of polyhedra, like most numerical domains based on convex approximations, is unable to precisely filter on a disequality constraint.

Example 2.2. For polyhedron P of Example 2.1, consider a branch guarded by constraint $c \equiv (x_0 = 2)$. We can be

[1] The experimental evaluation reported in [7] targets the analysis of a particular class of hybrid systems.

[2] Readers interested in the details of the optimized implementation of rational splits are referred to [7, Section 5].

precise on the equality branch $P_{\texttt{then}} \equiv \{x_0 = 2, 0 \leq x_1 \leq 3\}$; on the other branch we may try to lower the disequality into a pair of inequalities, intuitively computing $P^{<}_{\texttt{else}} \equiv \{0 \leq x_0 \leq 1, 0 \leq x_1 \leq 3\}$ and $P^{>}_{\texttt{else}} \equiv \{x_0 = 3, 0 \leq x_1 \leq 3\}$; unfortunately, this effort towards precision is later made useless by the join computation $P_{\texttt{else}} = P^{<}_{\texttt{else}} \uplus P^{>}_{\texttt{else}} = P$.

Some attempts can be made to identify those lucky cases where a disequality can be successfully refined into an inequality constraint, as in the following example.

Example 2.3. When splitting polyhedron P of Example 2.1 on constraint $c \equiv (x_0 = 3)$, we obtain $P_{\texttt{then}} \equiv \{x_0 = 3, 0 \leq x_1 \leq 3\}$, $P^{<}_{\texttt{else}} \equiv \{0 \leq x_0 \leq 2, 0 \leq x_1 \leq 3\}$ and $P^{>}_{\texttt{else}} \equiv \bot$; hence, as shown in Figure 2b, $P_{\texttt{else}} = P^{<}_{\texttt{else}} \uplus P^{>}_{\texttt{else}} = P^{<}_{\texttt{else}}$.

The approach varies depending on the considered static analysis tool. For instance, Crab is able to detect a lucky case when the disequality constraint $c^{\neq}$ satisfies rather specific conditions:

(a) $c^{\neq} \equiv (x_i \neq x_j)$ and P implies either $c^{\leq} \equiv (x_i \leq x_j)$ or $c^{\geq} \equiv (x_i \geq x_j)$; or
(b) $c^{\neq} \equiv (x_i \neq expr)$ and there exists $k \in \mathbb{Z}$ such that P implies $c_k \equiv (expr = k)$,[3] and P also implies either $c^{\leq} \equiv (x_i \leq k)$ or $c^{\geq} \equiv (x_i \geq k)$.

What should be noted here is that, in the lack of a specific operator for the integral split, the analysis tool is forced to perform several calls to lower level abstract operators (entailment checks, evaluations of the value range of a linear expression, etc.), with a corresponding multiplication of the overheads that are inherently incurred when interfacing the static analysis tool with a *generic* abstract domain component; hence, the more precise (and expensive) domains will likely witness a degradation of their overall efficiency.

3 Enabling Splits in a Static Analysis

For our experiments we have chosen Clam/Crab, which is the Abstract Interpretation engine included in the SeaHorn framework [15]. The Clam component uses Clang/LLVM to obtain the LLVM bitcode of the program under analysis and then generates the corresponding CrabIR representation [16]; this is processed by the Crab component, which computes the abstract semantics according to the chosen analysis configuration. The latter includes, among many other parameters, the choice of the abstract domain: Crab supports many (combinations of) abstract domains and includes interfaces towards the domains provided by libraries Apron [18] and ELINA [22].

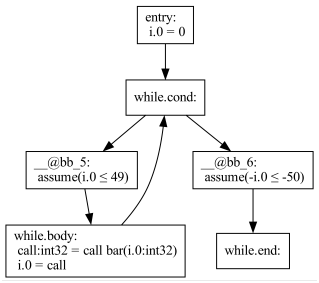
The addition of a new abstract operator requires that suitable changes are applied to both the abstract domain and the fixpoint engine components of the static analysis tool.

[3] I.e., all the variables x_j occurring in *expr* are bound in P to constant values.

Changes in the abstract domain component. We have first implemented the rational and integral split operators for the polyhedra domains included in the PPLite library [5, 6, 8] and we made them available (by adding a non-generic, *ad hoc* function) in the corresponding Apron interface wrapper. Then, we have extended the generic abstract domain interface in Crab by adding a new method for the split operator: this invokes the new abstract operator when the interface is instantiated with a PPLite domain, while resorting to the unoptimized implementation (based on the filter operator) when it is instantiated with the other domains.

```
void foo() {int i=0; while(i<50) i=bar(i);}
```

(a) A simple C function.



(b) Original CrabIR CFG. **(c)** CrabIR CFG with split.

Figure 3. The addition of split statements in CrabIR.

Changes in the fixpoint approximation engine. The adaptation of the fixpoint engine required more work than expected. This is mainly due to a CrabIR language design choice (inherited from IKOS AR form [9]) whereby all conditional branches are expressed in a declarative way, combining a non-deterministic branch with the addition, in each target of the branch, of `assume` abstract statements encoding the branch condition (see the CFG in Figure 3b for an example); roughly speaking, the unoptimized implementation of the conditional branch is *hard coded* in the CrabIR representation, preventing the application of the split operator. As a workaround, we enriched CrabIR by adding a new abstract statement, called `split`, and then modified Clam to generate this new statement whenever translating a numerical conditional branch (see the CFG in Figure 3c). Branches based on Boolean and pointer tests are not affected and hence maintain the declarative encoding.

The other main change to the fixpoint computation engine regards the choice of where to store the invariants computed during the analysis. Exploiting the declarative encoding of conditional branches, by default Crab annotates each node in the CFG with the pairs $\langle pre, post \rangle$ of invariants that are valid at the start (*pre*) and at the end (*post*) of the node. However, when the CFG is modified by introducing `split` statements this approach is no longer adequate, because we would need

to store two different invariants ($post_{\texttt{then}}$ and $post_{\texttt{else}}$) at the exit of those nodes ending with a `split` statement. Therefore, we modified the engine so as to store an invariant along each *edge* of the CFG, as well as storing an invariant on those nodes where a widening/narrowing is computed. At the end of the analysis, in a *finalization phase*, the edge invariants are used to compute the $\langle pre, post \rangle$ pairs of each CFG node, which requires computing many joins; while this simplistic approach incurs avoidable inefficiencies, it preserves the expectations of the other Clam/Crab components, that have not been modified.

It is worth noting that our current prototype is considering a *subset* of all the numerical branches that, in principle, could benefit from the split operator. Namely:

- *rational* splits are disregarded because they only make sense when branching on a floating point condition; currently, Clam/Crab provides limited support for floating point datatypes and safely ignores these conditions, yielding a pure non-deterministic branch;
- similarly, Clam/Crab safely ignores integral splits when the corresponding predicates are linear inequalities defined on *unsigned* variables; this is done to avoid potential safety issues related to the wrap-around semantics as defined in C-like languages;
- finally, our prototype currently disregards those *implicit* numerical branches encoded in CrabIR `select` statements, which implement conditional assignments.

As a consequence we conjecture that, in our experimental evaluation, the efficiency improvements obtainable thanks to the split operator are underestimated.

4 Experimental Evaluation

Our prototype analyzer was obtained by modifying the `dev14` branch of Clam/Crab,[4] which is based on LLVM 14. In our experiments we tried to apply minimal changes to the default configuration of the analyzer: in particular, we instructed LLVM to systematically inline function calls, so as to improve the call context sensitivity of the analysis. Note that, by default, Clam/Crab instructs LLVM to lower all switch statements, which are thus translated to chains of conditional branches and hence can benefit of the split optimization.

Table 1 reports the timing results for some of the tests considered in our *preliminary* experimental evaluation. We analyzed programs coming from two different sources. First, we considered 39 C source files distributed with PAGAI [17] which are variants of benchmarks taken from the SNU real-time benchmark suite for WCET (worst-case execution time) analysis; the results for the 5 tests whose analysis time was greater than a second are shown in the top half of the table. Then, we enriched our benchmark suite by also considering a few Linux drivers from the SVCOMP repository;[5] in this case, we applied no specific selection criterion and just cherry-picked a few drivers, shown in the lower half of the table, having reasonable analysis time.

	without splits		with splits	time
(abridged) test name	PPLite	ELINA	PPLite	ratio
adpcm	108.2	(⋆) 1.6	105.9	1.02
prog9000	31.8	241.1	42.0	0.76
nsichneu	30.9	40.8	21.1	1.46
decompress	5.3	5.5	2.3	2.30
filter	2.5	2.3	2.5	1.00
mmc-host	487.1	435.1	412.9	1.18
firewire	98.7	(⋆) 92.0	22.7	4.35
hwmon-abituguru3	87.5	91.0	52.8	1.66
media-usb-tm6000	23.2	22.3	12.4	1.87
media-pci-ttpci	12.3	12.9	10.8	1.14
power-bq2415x	10.1	10.9	11.6	0.87
hid-usb	8.6	10.1	6.0	1.43

Table 1. Overall static analysis time without/with splits.

The 2nd column in Table 1 shows the baseline for the efficiency evaluation, i.e., the overall analysis time in seconds when using `F_Poly`, the Cartesian factored convex polyhedra domain of PPLite; note that we include time spent in pre-analysis phases (e.g., parsing, LLVM bitcode generation and Clam preprocessing steps), while excluding post-analysis phases (e.g., assertion checks based on the results of the analysis).[6] In the 3rd column we show the time obtained when using the Cartesian factored convex polyhedra domain of ELINA [22]: this is done to highlight that our starting point for the efficiency comparison is in line with what is considered the most efficient library for convex polyhedra. Note however that ELINA cannot be used as a proper baseline, as it is well known that, by using machine integers (rather than the arbitrary precision integers adopted by PPLite), it sometimes raises overflow exceptions, after which it returns an over-approximation of the actual result, hence affecting both the efficiency and the precision of the analysis; these cases are highlighted in the table using (⋆).

The 4th column of the table reports the overall analysis time obtained when enabling the split optimization in our prototype, while the last column shows the speedup with respect to the baseline. On the considered tests, we obtain significant speedups, sometimes beyond our own expectations. In a few cases we also obtain slowdowns: these seem to be mainly caused by the unoptimized invariant finalization phase described in the previous section; by disabling this phase (leaving invariants on CFG edges), the analysis time of tests prog9000 and power-bq2415x can be reduced by 9.1 and 3.2 seconds, respectively.

A remarkable side effect of the split optimization is a significant reduction in peak memory usage, as highlighted by the data shown in Table 2. The 2nd and 5th columns of the

[4] https://github.com/seahorn/clam/tree/dev14
[5] https://github.com/sosy-lab/sv-benchmarks/c/ldv-linux-4.2-rc1

[6] Experiments have been performed on a laptop with an Intel Core i7-3632QM CPU, 16 GB of RAM and running GNU/Linux 5.15.0-60.

(abridged) test name	without splits nodes	without splits mem	splits	with splits nodes	with splits mem	mem ratio
adpcm	146	87	34	93	85	1.01
prog9000	1491	1629	275	947	1415	1.15
nsichneu	2004	1591	625	754	628	2.53
decompress	1032	177	266	638	87	2.15
filter	1121	263	187	809	254	1.04
mmc-host	26254	9705	3772	19701	1102	8.81
firewire	9925	7690	2038	6842	282	27.27
hwmon-abituguru3	2653	271	521	2088	130	2.08
media-usb-tm6000	15447	3300	1453	12886	200	16.50
media-pci-ttpci	9057	524	156	5610	147	3.56
power-bq2415x	23763	351	1985	20518	256	1.37
hid-usb	7303	185	1478	4704	119	1.55

Table 2. Number of nodes, splits and maximum RSS.

table show the number of nodes in the CrabIR CFGs generated without and with the split optimization (note that, in both cases, we refer to the CFGs *after* the LLVM inlining phase). As intuitively suggested by the CFGs in Figure 3, the decrease in the number of nodes is directly caused by the introduction of the numerical `split` statements, whose number is reported in the 4th column. In the 3rd and 6th columns of Table 2 we provide an estimation of the peak memory usage by reporting the maximum RSS (Resident Set Size) for the two aforementioned configurations (in MB); the last column of the table shows the ratio of the two memory measurements. A recent paper [19] proposes a technique to reduce the memory footprint of the static analysis tool IKOS (which shares with Crab the main design of the fixpoint approximation engine). Since the technique adopted in [19] is completely independent from the split optimization, we conjecture that the two optimizations can be applied together, combining their improvements.

Our experimental evaluation also confirmed that the abstract execution of the split statements using the new split abstract domain operator has no effect on precision: using the `clam-diff` helper tool, we observed no regression when systematically comparing the invariants produced by the baseline and the optimized analyses.

5 Conclusions

This paper proposes a new abstract domain operator that is able to speed up the static analysis when splitting the control flow path on a predicate and its complement. Even though our current prototype can only handle a subset of the control flow splits of the program, it is able to obtain non-trivial memory and time improvements on several tests, including both synthetic benchmarks and more realistic programs.

Future work can investigate several directions. First of all, the current prototype can be extended to enable the optimization on more kinds of numerical split statements, e.g., the correct handling of branches on unsigned integral expressions and the implicit branches in `select` statements.

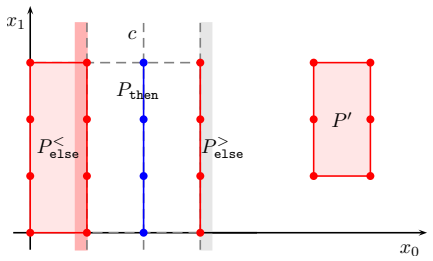


Figure 4. Example of split on a powerset domain.

Second, we plan to evaluate the applicability of the approach to other abstract domains; as briefly discussed before, while the overall idea is quite general, its effectiveness depends on profitability considerations. In particular, we believe that abstract domains supporting the representation of *disjunctive* information are promising candidates for an optimized split operator. As a first example, one could consider the finite powerset of convex polyhedra [3]. Note that, when using a disjunctive domain, it is clearly possible to workaround the limitations of convex over-approximations and thus improve the precision of splits. For instance, in Figure 4 we consider the split of the powerset $S = \{P, P'\}$ on the equality constraint c, where P and c are those described in Example 2.2; hence, by avoiding the convex polyhedral hull approximation, we can obtain $S_{\texttt{then}} = \{P_{\texttt{then}}\}$ and $S_{\texttt{else}} = \{P^{<}_{\texttt{else}}, P^{>}_{\texttt{else}}, P'\}$; note that P', which is not directly affected by the split operator, can be simply "moved" into $S_{\texttt{else}}$, avoiding useless and costly copy operations.

Other disjunctive abstract domains that are probably worth considering are LDDs [14] and RDDs [13] (Linear and Range Decision Diagrams), both available in Crab. We also plan to investigate the application of split operators to non-numeric domains, such as ROBDD-based domains for Boolean formulae or DFA-based abstract domains for string analysis [1, 21]; for instance, we could explore an optimized split operator for conditional branches based on string predicates, such as `str.startsWith("prefix")`, whose default implementations on the domain of DFAs are often expensive.

Finally, it would be interesting to extend our experimental evaluation to different static analysis tools; while we conjecture that similar results can be obtained for other tools analysing the low level program representation (e.g., IKOS [9] and PAGAI [17]), it is more difficult to predict the effectiveness of the optimization for those tools targeting the AST-based high level representations (e.g., MOPSA [20]).

Acknowledgments

The authors would like to express their gratitude to Jorge A. Navas for his help in getting them acquainted with the inner workings of Clam/Crab and for developing helper tool `clam-diff` to compare the precision of different analyses.

References

[1] Vincenzo Arceri, Isabella Mastroeni, and Sunyi Xu. 2020. Static Analysis for ECMAScript String Manipulation Programs. *Appl. Sci.* 10 (2020), 3525. https://doi.org/10.3390/app10103525

[2] Roberto Bagnara, Katy Louise Dobson, Patricia M. Hill, Matthew Mundell, and Enea Zaffanella. 2006. Grids: A Domain for Analyzing the Distribution of Numerical Values. In *Logic-Based Program Synthesis and Transformation, 16th International Symposium, LOPSTR 2006, Venice, Italy, July 12-14, 2006, Revised Selected Papers (Lecture Notes in Computer Science, Vol. 4407)*, Germán Puebla (Ed.). Springer, 219–235. https://doi.org/10.1007/978-3-540-71410-1_16

[3] Roberto Bagnara, Patricia M. Hill, and Enea Zaffanella. 2007. Widening operators for powerset domains. *Int. J. Softw. Tools Technol. Transf.* 9, 3-4 (2007), 413–414. https://doi.org/10.1007/s10009-007-0029-y

[4] Roberto Bagnara, Patricia M. Hill, and Enea Zaffanella. 2008. The Parma Polyhedra Library: Toward a complete set of numerical abstractions for the analysis and verification of hardware and software systems. *Sci. Comput. Program.* 72, 1-2 (2008), 3–21. https://doi.org/10.1016/j.scico.2007.08.001

[5] Anna Becchi and Enea Zaffanella. 2018. A Direct Encoding for NNC Polyhedra. In *Computer Aided Verification - 30th International Conference, CAV 2018, Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, Part I (Lecture Notes in Computer Science, Vol. 10981)*, Hana Chockler and Georg Weissenbacher (Eds.). Springer, 230–248. https://doi.org/10.1007/978-3-319-96145-3_13

[6] Anna Becchi and Enea Zaffanella. 2018. An Efficient Abstract Domain for Not Necessarily Closed Polyhedra. In *Static Analysis - 25th International Symposium, SAS 2018, Freiburg, Germany, August 29-31, 2018, Proceedings (Lecture Notes in Computer Science, Vol. 11002)*, Andreas Podelski (Ed.). Springer, 146–165. https://doi.org/10.1007/978-3-319-99725-4_11

[7] Anna Becchi and Enea Zaffanella. 2019. Revisiting Polyhedral Analysis for Hybrid Systems. In *Static Analysis - 26th International Symposium, SAS 2019, Porto, Portugal, October 8-11, 2019, Proceedings (Lecture Notes in Computer Science, Vol. 11822)*, Bor-Yuh Evan Chang (Ed.). Springer, 183–202. https://doi.org/10.1007/978-3-030-32304-2_10

[8] Anna Becchi and Enea Zaffanella. 2020. PPLite: Zero-overhead encoding of NNC polyhedra. *Inf. Comput.* 275 (2020), 104620. https://doi.org/10.1016/j.ic.2020.104620

[9] Guillaume Brat, Jorge A. Navas, Nija Shi, and Arnaud Venet. 2014. IKOS: A Framework for Static Analysis Based on Abstract Interpretation. In *Software Engineering and Formal Methods - 12th International Conference, SEFM 2014, Grenoble, France, September 1-5, 2014. Proceedings (Lecture Notes in Computer Science, Vol. 8702)*, Dimitra Giannakopoulou and Gwen Salaün (Eds.). Springer, 271–277. https://doi.org/10.1007/978-3-319-10431-7_20

[10] Patrick Cousot and Radhia Cousot. 1977. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages, Los Angeles, California, USA, January 1977*, Robert M. Graham, Michael A. Harrison, and Ravi Sethi (Eds.). ACM, 238–252. https://doi.org/10.1145/512950.512973

[11] Patrick Cousot and Nicolas Halbwachs. 1978. Automatic Discovery of Linear Restraints Among Variables of a Program. In *Conference Record of the Fifth Annual ACM Symposium on Principles of Programming Languages, Tucson, Arizona, USA, January 1978*, Alfred V. Aho, Stephen N. Zilles, and Thomas G. Szymanski (Eds.). ACM Press, 84–96. https://doi.org/10.1145/512760.512770

[12] Pietro Ferrara, Luca Negrini, Vincenzo Arceri, and Agostino Cortesi. 2021. Static analysis for dummies: experiencing LiSA. In *SOAP@PLDI 2021: Proceedings of the 10th ACM SIGPLAN International Workshop on the State Of the Art in Program Analysis, Virtual Event, Canada, 22 June, 2021*, Lisa Nguyen Quang Do and Caterina Urban (Eds.). ACM, 1–6. https://doi.org/10.1145/3460946.3464316

[13] Graeme Gange, Jorge A. Navas, Peter Schachte, Harald Søndergaard, and Peter J. Stuckey. 2021. Disjunctive Interval Analysis. In *Static Analysis - 28th International Symposium, SAS 2021, Chicago, IL, USA, October 17-19, 2021, Proceedings (Lecture Notes in Computer Science, Vol. 12913)*, Cezara Dragoi, Suvam Mukherjee, and Kedar S. Namjoshi (Eds.). Springer, 144–165. https://doi.org/10.1007/978-3-030-88806-0_7

[14] Arie Gurfinkel and Sagar Chaki. 2010. Boxes: A Symbolic Abstract Domain of Boxes. In *Static Analysis - 17th International Symposium, SAS 2010, Perpignan, France, September 14-16, 2010. Proceedings (Lecture Notes in Computer Science, Vol. 6337)*, Radhia Cousot and Matthieu Martel (Eds.). Springer, 287–303. https://doi.org/10.1007/978-3-642-15769-1_18

[15] Arie Gurfinkel, Temesghen Kahsai, Anvesh Komuravelli, and Jorge A. Navas. 2015. The SeaHorn Verification Framework. In *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part I (Lecture Notes in Computer Science, Vol. 9206)*, Daniel Kroening and Corina S. Pasareanu (Eds.). Springer, 343–361. https://doi.org/10.1007/978-3-319-21690-4_20

[16] Arie Gurfinkel and Jorge A. Navas. 2021. Abstract Interpretation of LLVM with a Region-Based Memory Model. In *Software Verification - 13th International Conference, VSTTE 2021, New Haven, CT, USA, October 18-19, 2021, and 14th International Workshop, NSV 2021, Los Angeles, CA, USA, July 18-19, 2021, Revised Selected Papers (Lecture Notes in Computer Science, Vol. 13124)*, Roderick Bloem, Rayna Dimitrova, Chuchu Fan, and Natasha Sharygina (Eds.). Springer, 122–144. https://doi.org/10.1007/978-3-030-95561-8_8

[17] Julien Henry, David Monniaux, and Matthieu Moy. 2012. PAGAI: A Path Sensitive Static Analyser. In *Third Workshop on Tools for Automatic Program Analysis, TAPAS 2012, Deauville, France, September 14, 2012 (Electronic Notes in Theoretical Computer Science, Vol. 289)*, Bertrand Jeannet (Ed.). Elsevier, 15–25. https://doi.org/10.1016/j.entcs.2012.11.003

[18] Bertrand Jeannet and Antoine Miné. 2009. Apron: A Library of Numerical Abstract Domains for Static Analysis. In *Computer Aided Verification, 21st International Conference, CAV 2009, Grenoble, France, June 26 - July 2, 2009. Proceedings (Lecture Notes in Computer Science, Vol. 5643)*, Ahmed Bouajjani and Oded Maler (Eds.). Springer, 661–667. https://doi.org/10.1007/978-3-642-02658-4_52

[19] Sung Kook Kim, Arnaud J. Venet, and Aditya V. Thakur. 2020. Memory-Efficient Fixpoint Computation. In *Static Analysis - 27th International Symposium, SAS 2020, Virtual Event, November 18-20, 2020, Proceedings (Lecture Notes in Computer Science, Vol. 12389)*, David Pichardie and Mihaela Sighireanu (Eds.). Springer, 35–64. https://doi.org/10.1007/978-3-030-65474-0_3

[20] Raphaël Monat, Abdelraouf Ouadjaout, and Antoine Miné. 2021. A Multilanguage Static Analysis of Python Programs with Native C Extensions. In *Static Analysis - 28th International Symposium, SAS 2021, Chicago, IL, USA, October 17-19, 2021, Proceedings (Lecture Notes in Computer Science, Vol. 12913)*, Cezara Dragoi, Suvam Mukherjee, and Kedar S. Namjoshi (Eds.). Springer, 323–345. https://doi.org/10.1007/978-3-030-88806-0_16

[21] Luca Negrini, Vincenzo Arceri, Pietro Ferrara, and Agostino Cortesi. 2021. Twinning Automata and Regular Expressions for String Static Analysis. In *Verification, Model Checking, and Abstract Interpretation - 22nd International Conference, VMCAI 2021, Copenhagen, Denmark, January 17-19, 2021, Proceedings (Lecture Notes in Computer Science, Vol. 12597)*, Fritz Henglein, Sharon Shoham, and Yakir Vizel (Eds.). Springer, 267–290. https://doi.org/10.1007/978-3-030-67067-2_13

[22] Gagandeep Singh, Markus Püschel, and Martin T. Vechev. 2017. Fast polyhedra abstract domain. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, Giuseppe Castagna and Andrew D. Gordon (Eds.). ACM, 46–59. https://doi.org/10.1145/3009837.3009885