Design of a Modular Platform for Static Analysis

Antoine Miné   Abdelraouf Ouadjaout*   Matthieu Journault
Raphaël Monat

LIP6, Sorbonne Université, Paris

Journée de l'équipe APR
July 5, 2019

What is MOPSA?

MOPSA is a *__M__odular __O__pen __P__latform for __S__tatic __A__nalysis*

1. Based on **abstract interpretation**

∀	Sound	Cover all possible executions.
🕘	Efficient	Terminates in finite time.
💻	Automatic	No need to manual annotation.

2. Supports **multiple languages**

📄	Expressiveness	Keep the original AST of the program.
♻	Reusability	Reuse abstractions among languages.

3. Features a **flexible architecture**

🧩	Loose coupling	Divided into interchangeable components.
📦	Composition	Create complex components from simpler ones.
💬	Cooperation	Components can communicate and delegate tasks.

👎 May raise false alarms.

Abstract Interpretation

How can an abstract interpreter handle this program?

```
int main(int argc, char *argv[]) {
  char **p = argv; int i = 0;
  while(*p) {
    printf("length(argv[%d]) = %lu\n", i, strlen(*p));
    p++; i++;
  }
}
```

Abstract Interpretation

How can an abstract interpreter handle this program?

```
int main(int argc, char *argv[]) {
  char **p = argv; int i = 0;
  while(*p) {
    printf("length(argv[%d]) = %lu\n", i, strlen(*p));
    p++; i++;
  }
}
```

- Use computable abstractions to over-approximate states.

Numeric

$\texttt{argc} \in [1, \texttt{SIZE\_MAX} - 1]$

$\textit{size}(\texttt{argv}) = \texttt{argc} + 1$

Abstract Interpretation

How can an abstract interpreter handle this program?

```
int main(int argc, char *argv[]) {
  char **p = argv; int i = 0;
  while(*p) {
    printf("length(argv[%d]) = %lu\n", i, strlen(*p));
    p++; i++;
  }
}
```

- Use computable abstractions to over-approximate states.

Numeric	Pointers
$\mathtt{argc} \in [1, \mathtt{SIZE\_MAX} - 1]$	$\mathtt{argv}[0..\mathtt{argc} - 1] \rightsquigarrow \{\mathtt{@}\}$
$\mathit{size}(\mathtt{argv}) = \mathtt{argc} + 1$	$\mathtt{argv}[\mathtt{argc}] \rightsquigarrow \mathtt{NULL}$
$\mathit{size}(\mathtt{@}) \in [1, \mathtt{SIZE\_MAX}]$	

Abstract Interpretation

How can an abstract interpreter handle this program?

```
int main(int argc, char *argv[]) {
  char **p = argv; int i = 0;
  while(*p) {
    printf("length(argv[%d]) = %lu\n", i, strlen(*p));
    p++; i++;
  }
}
```

- Use computable abstractions to over-approximate states.

Numeric	Pointers	Strings
$\mathtt{argc} \in [1, \mathtt{SIZE\_MAX} - 1]$	$\mathtt{argv}[0..\mathtt{argc}-1] \rightsquigarrow \{@\}$	$\exists \mathtt{k} \in [0, \mathit{size}(@) - 1] : @[\mathtt{k}] = 0$
$\mathit{size}(\mathtt{argv}) = \mathtt{argc} + 1$	$\mathtt{argv}[\mathtt{argc}] \rightsquigarrow \mathtt{NULL}$	
$\mathit{size}(@) \in [1, \mathtt{SIZE\_MAX}]$		

Abstract Interpretation

How can an abstract interpreter handle this program?

```
int main(int argc, char *argv[]) {
  char **p = argv; int i = 0;
  while(*p) {
  ● printf("length(argv[%d]) = %lu\n", i, strlen(*p));
    p++; i++;
  }
}
```

- Use computable abstractions to over-approximate states.
- Infer inductive invariants.

Numeric	Pointers	Strings
$\mathtt{argc} \in [1, \mathtt{SIZE\_MAX} - 1]$	$\mathtt{argv}[0..\mathtt{argc} - 1] \rightsquigarrow \{\mathtt{@}\}$	$\exists \mathtt{k} \in [0, \mathit{size}(\mathtt{@}) - 1] : \mathtt{@}[\mathtt{k}] = 0$
$\mathit{size}(\mathtt{argv}) = \mathtt{argc} + 1$	$\mathtt{argv}[\mathtt{argc}] \rightsquigarrow \mathtt{NULL}$	
$\mathit{size}(\mathtt{@}) \in [1, \mathtt{SIZE\_MAX}]$	$\mathtt{p} \rightsquigarrow \{\mathtt{argv}\}$	
$0 \leq \mathit{offset}(\mathtt{p}) \leq \mathit{size}(\mathtt{argv}) - 1$		
$\mathit{offset}(\mathtt{p}) = \mathtt{i}$		

Abstract Interpretation

How can an abstract interpreter handle this program?

```
int main(int argc, char *argv[]) {
  char **p = argv; int i = 0;
  while(*p) {
    printf("length(argv[%d]) = %lu\n", i, strlen(*p));
    p++; i++;
  }
}
```

Valid string passed to strlen?

Integer overflow on i?

- Use computable abstractions to over-approximate states.
- Infer inductive invariants.
- Check safety rules.

Numeric	Pointers	Strings
$\mathtt{argc} \in [1, \mathtt{SIZE\_MAX} - 1]$	$\mathtt{argv}[0..\mathtt{argc} - 1] \rightsquigarrow \{@\}$	$\exists \mathtt{k} \in [0, \mathit{size}(@) - 1] : @[\mathtt{k}] = 0$
$\mathit{size}(\mathtt{argv}) = \mathtt{argc} + 1$	$\mathtt{argv}[\mathtt{argc}] \rightsquigarrow \mathrm{NULL}$	
$\mathit{size}(@) \in [1, \mathtt{SIZE\_MAX}]$	$\mathtt{p} \rightsquigarrow \{\mathtt{argv}\}$	
$0 \le \mathit{offset}(\mathtt{p}) \le \mathit{size}(\mathtt{argv}) - 1$		
$\mathit{offset}(\mathtt{p}) = \mathtt{i}$		

Part I

Multi-Language Support

Multi-Language Support

Extensible AST

Existing Solutions

- **Static** translation to an intermediate language.
  e.g. LLVM, SIL of Facebook Infer(C/C++/ObjectiveC/Java).

👍 Easy reuse of abstractions.

👎 Loss of source code information.

Mopsa Approach

- Keep the original AST of the program.
- Domains can **extend** the language to add new AST nodes.

```
type expr_kind +=
  | E_c_call of ...
```

```
type expr_kind +=
  | E_py_call of ...
```

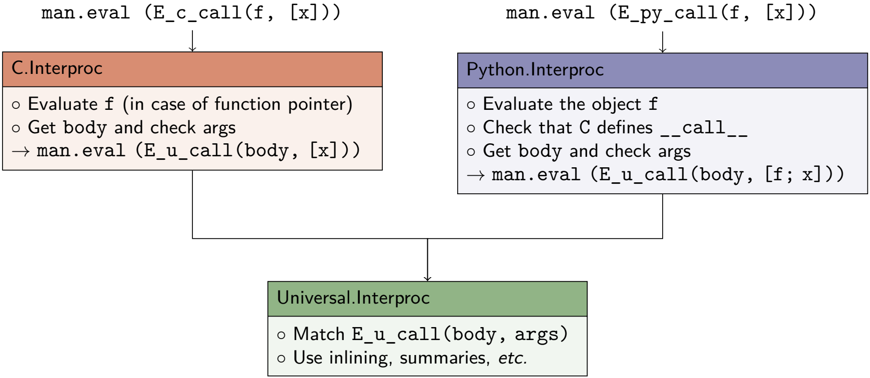
- Intermediate languages are added to share abstractions.
  Original and intermediate languages are in the same AST.
- Translation is **dynamic**, **semantic** and **cooperative**.

Multi-Language Support

Cooperation via Delegation

```
int f(int x)
{
  <...>
}
```

```
class F:
    def __call__(self, x):
        <...>
f = F()
```



`man.eval (E_c_call(f, [x]))`

C.Interproc

- ◦ Evaluate `f` (in case of function pointer)
- ◦ Get body and check args
- → `man.eval (E_u_call(body, [x]))`

`man.eval (E_py_call(f, [x]))`

Python.Interproc

- ◦ Evaluate the object `f`
- ◦ Check that `C` defines `__call__`
- ◦ Get body and check args
- → `man.eval (E_u_call(body, [f; x]))`

Universal.Interproc

- ◦ Match `E_u_call(body, args)`
- ◦ Use inlining, summaries, *etc.*

Part II

Flexible Architecture

Flexible Architecture

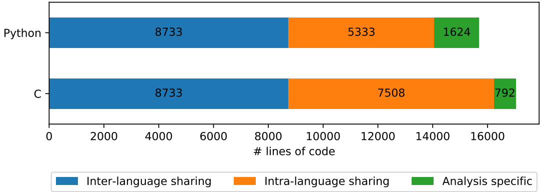
Domains

Domains

- Analyzer is decomposed into **domains** over extensible AST.
- Share domains as much as possible.
- Several domains can abstract the same thing, but differently.
  $\{\langle x \mapsto 0\rangle, \langle x \mapsto 2\rangle\}$ *can be abstracted as* $\langle x \mapsto [0,2]\rangle$ *or* $\langle x \mapsto 2\mathbb{Z}\rangle$.

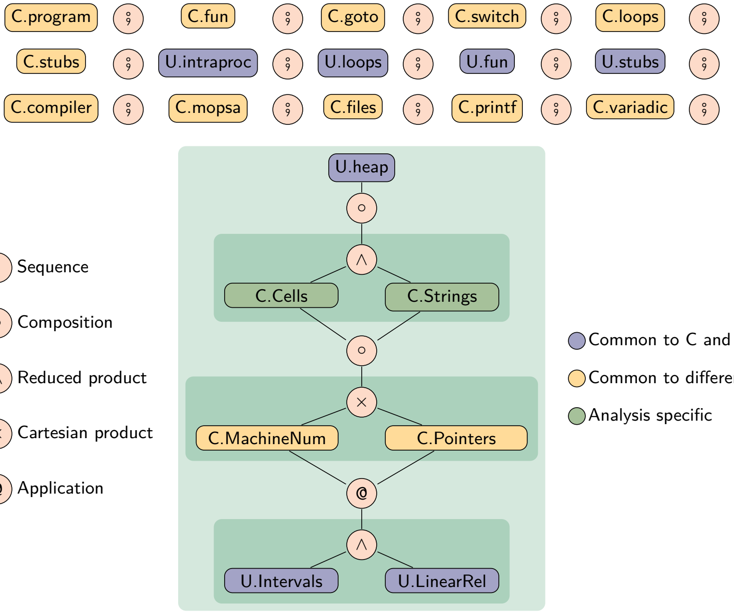
Unified Signature

- MOPSA defines a **unified** signature for all domains.
- Other simplified signatures are automatically lifted.



Flexible Architecture

Composition



C.program ⨟ C.fun ⨟ C.goto ⨟ C.switch ⨟ C.loops ⨟

C.stubs ⨟ U.intraproc ⨟ U.loops ⨟ U.fun ⨟ U.stubs ⨟

C.compiler ⨟ C.mopsa ⨟ C.files ⨟ C.printf ⨟ C.variadic ⨟

U.heap

∘

∧

C.Cells

C.Strings

∘

×

C.MachineNum

C.Pointers

@

∧

U.Intervals

U.LinearRel

⨟ Sequence

∘ Composition

∧ Reduced product

× Cartesian product

@ Application

🟣 Common to C and Python

🟡 Common to different C analysis

🟢 Analysis specific

Numeric Domains

LinearRel $\wedge$ Intervals

$$\textsc{Intervals} \overset{\text{def}}{=} \mathcal{V} \to (\mathbb{Z} \cup \{-\infty\} \times \mathbb{Z} \cup \{+\infty\})$$

$$\textsc{LinearRel} \overset{\text{def}}{=} \bigwedge_j \left(\sum_{i=1}^{n} \alpha_{ij} \mathtt{v}_i \geq \beta_j\right), \alpha_{ij}, \beta_j \in \mathbb{R}, \mathtt{v}_i \in \mathcal{V}$$

- INTERVALS is a fast domain provided by most analyzers.
- LINEARREL keeps track of linear relations.
  e.g. $\mathit{size}(\mathtt{argv}) = \mathtt{argc} + 1$
- LINEARREL and INTERVALS have the same concrete semantics.
  $\Longrightarrow$ can be combined with a meet product $\wedge$.

Need for Collaboration

Pre-state: $\mathtt{x} \mapsto [0, 10]$, $\mathtt{y} \mapsto [0, 10]$ $\wedge$ $\mathtt{x} = \mathtt{y}$, $0 \leq \mathtt{x} \leq 10$

$\downarrow$

$\mathbb{S}^\sharp[\![$ `if (x < 5)` $]\!]$

$\downarrow$

Post-state: $\mathtt{x} \mapsto [0, 4]$, $\mathtt{y} \mapsto [0, 10]$ $\wedge$ $\mathtt{x} = \mathtt{y}$, $0 \leq \mathtt{x} \leq 4$

- INTERVALS can not infer that `y < 5`.
- How to collaborate while preserving loose coupling?

Reduction Rules

LinearRel ∧ Intervals

- Reduction rules combine the results of two abstract domains.
- They are implemented **outside** domains to reduce coupling.

Pre-state

$\mathtt{x} \mapsto [0, 10]$ $\quad\wedge\quad$ $\mathtt{x} = \mathtt{y}$

$\mathtt{y} \mapsto [0, 10]$ $\quad\quad\quad$ $0 \leq \mathtt{x} \leq 10$

$\downarrow$

$\mathbb{S}^\sharp[\![\, \texttt{if (x < 5)} \,]\!]$

$\downarrow$

Post-state

$\mathtt{x} \mapsto [0, 4]$ $\quad\quad\quad$ $\mathtt{x} = \mathtt{y}$

$\mathtt{y} \mapsto [0, 10]$ $\quad\wedge\quad$ $0 \leq \mathtt{x} \leq 4$

The hidden constraint reduction

1. Collect variables V statically present in the statement.
   $V = \{\mathtt{x}\}$
2. Search for the set variables R in LinearRel related to V.
   $R = \{\mathtt{y}\}$, *since* $\mathtt{x} = \mathtt{y}$
3. Get the interval of R in LinearRel and refine Intervals.
   $\textsc{Intervals}[\mathtt{x} \mapsto [0, 10] \sqcap [0, 4]]$

Scalar Domains

Machine Numbers

MachineNum @ (LinearRel ∧ Intervals)

- INTERVALS and LINEARREL uses **mathematical** numbers.
- C uses **finite precision** numbers with modular arithmetics.
- MACHINENUM lifts C statements to a math semantic.
  → Operator @ applies a **stack** domain to an argument.

Scalar Domains

Machine Numbers

MachineNum @ (LinearRel ∧ Intervals)

- INTERVALS and LINEARREL uses **mathematical** numbers.
- C uses **finite precision** numbers with modular arithmetics.
- MACHINENUM lifts C statements to a math semantic.
  → Operator @ applies a **stack** domain to an argument.

Dynamic Lifting

Consider a variable `x` declared as `unsigned char`.

$$\begin{tikzcd}
 & \wr \texttt{x + 2}, \langle \texttt{x} \mapsto [0,253] \rangle, \varnothing \int \\
\mathbb{E}^\sharp[\![\texttt{x + 2}]\!] \langle \texttt{x} \mapsto [0,255] \rangle \arrow[ur, "x+2 \in [0,255]"] \arrow[dr, "x+2 \notin [0,255]"'] & \\
 & \wr \mathit{wrap}(\texttt{x + 2}, [0,255]), \langle \texttt{x} \mapsto [254,255] \rangle, \{\texttt{IntOverflow}\} \int
\end{tikzcd}$$

Scalar Domains

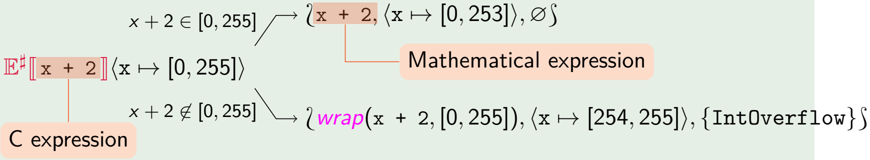
Machine Numbers

MachineNum @ (LinearRel ∧ Intervals)

- Intervals and LinearRel uses **mathematical** numbers.
- C uses **finite precision** numbers with modular arithmetics.
- MachineNum lifts C statements to a math semantic.
  → Operator @ applies a **stack** domain to an argument.

Dynamic Lifting

Consider a variable `x` declared as `unsigned char`.



$$\mathbb{E}^\sharp[\![\texttt{x + 2}]\!]\langle \texttt{x} \mapsto [0, 255]\rangle \begin{cases} \xrightarrow{x + 2 \in [0, 255]} \lbag \texttt{x + 2}, \langle \texttt{x} \mapsto [0, 253]\rangle, \emptyset \rbag \\ \xrightarrow{x + 2 \notin [0, 255]} \lbag \mathit{wrap}(\texttt{x + 2}, [0, 255]), \langle \texttt{x} \mapsto [254, 255]\rangle, \{\texttt{IntOverflow}\} \rbag \end{cases}$$

Mathematical expression

C expression

Scalar Domains

Machine Numbers

MachineNum @ (LinearRel ∧ Intervals)

Result Partitioning

MOPSA uses **monads and delegation** to handle $\wr r_1 \int \vee \cdots \vee \wr r_n \int$.

```
let eval exp man a =
  match exp.ekind with
  | E_var v -> Some (Eval.singleton exp a)
  | E_binop(op, e1, e2) ->
    man.eval e1 a >>= fun n1 a ->
    man.eval e2 a >>= fun n2 a ->
    let vmin, vmax = rangeof exp.etyp in
    let nexp = mk_binop n1 op n2 in
    let ret = assume (mk_in nexp vmin vmax) man a
        ~fthen:(fun a -> Eval.singleton nexp a)
        ~felse:(fun a ->
            let nexp’ = mk_wrap nexp vmin vmax in
            raise_alarm IntegerOverflow a |> Eval.singleton nexp’
          )
    in
    Some ret
  | _ -> None
```

Scalar Domains

Machine Numbers

MachineNum @ (LinearRel ∧ Intervals)

Result Partitioning

Mopsa uses **monads and delegation** to handle $\lbag r_1 \rbag \vee \cdots \vee \lbag r_n \rbag$.

```
let eval exp man a =
  match exp.ekind with
  | E_var v -> Some (Eval.singleton exp a)
  | E_binop(op, e1, e2) ->
    man.eval e1 a >>= fun n1 a ->
    man.eval e2 a >>= fun n2 a ->
    let vmin, vmax = rangeof exp.etyp in
    let nexp = mk_binop n1 op n2 in
    let ret = assume (mk_in nexp vmin vmax) man a
        ~fthen:(fun a -> Eval.singleton nexp a)
        ~felse:(fun a ->
            let nexp' = mk_wrap nexp vmin vmax in
            raise_alarm IntegerOverflow a |> Eval.singleton nexp'
          )
    in
    Some ret
  | _ -> None
```

`Variables do not overflow`

Scalar Domains

Machine Numbers

MachineNum @ (LinearRel ∧ Intervals)

Result Partitioning

Mopsa uses **monads and delegation** to handle $\wr r_1 \int \vee \cdots \vee \wr r_n \int$.

```
let eval exp man a =
  match exp.ekind with
  | E_var v -> Some (Eval.singleton exp a)
  | E_binop(op, e1, e2) ->
    man.eval e1 a >>= fun n1 a ->
    man.eval e2 a >>= fun n2 a ->
    let vmin, vmax = rangeof exp.etyp in
    let nexp = mk_binop n1 op n2 in
    let ret = assume (mk_in nexp vmin vmax) man a
        ~fthen:(fun a -> Eval.singleton nexp a)
        ~felse:(fun a ->
            let nexp' = mk_wrap nexp vmin vmax in
            raise_alarm IntegerOverflow a |> Eval.singleton nexp'
          )
    in
    Some ret
  | _ -> None
```

Evaluate e1 and bind each case $\wr$n1$\int$

Scalar Domains

Machine Numbers

MachineNum @ (LinearRel ∧ Intervals)

Result Partitioning

MOPSA uses **monads and delegation** to handle $\wr r_1 \int \vee \cdots \vee \wr r_n \int$.

```
let eval exp man a =
  match exp.ekind with
  | E_var v -> Some (Eval.singleton exp a)
  | E_binop(op, e1, e2) ->
    man.eval e1 a >>= fun n1 a ->
    man.eval e2 a >>= fun n2 a ->
    let vmin, vmax = rangeof exp.etyp in
    let nexp = mk_binop n1 op n2 in
    let ret = assume (mk_in nexp vmin vmax) man a
        ~fthen:(fun a -> Eval.singleton nexp a)
        ~felse:(fun a ->
            let nexp’ = mk_wrap nexp vmin vmax in
            raise_alarm IntegerOverflow a |> Eval.singleton nexp’
          )
    in
    Some ret
  | _ -> None
```

`Evaluate e2 and bind each case` $\wr$`n2`$\int$

Scalar Domains

Machine Numbers

MachineNum @ (LinearRel ∧ Intervals)

Result Partitioning

MOPSA uses **monads and delegation** to handle $\wr r_1 \int \vee \cdots \vee \wr r_n \int$.

```
let eval exp man a =
  match exp.ekind with
  | E_var v -> Some (Eval.singleton exp a)
  | E_binop(op, e1, e2) ->
    man.eval e1 a >>= fun n1 a ->
    man.eval e2 a >>= fun n2 a ->
    let vmin, vmax = rangeof exp.etyp in
    let nexp = mk_binop n1 op n2 in
    let ret = assume (mk_in nexp vmin vmax) man a
        ~fthen:(fun a -> Eval.singleton nexp a)
        ~felse:(fun a ->
            let nexp' = mk_wrap nexp vmin vmax in
            raise_alarm IntegerOverflow a |> Eval.singleton nexp'
          )
    in
    Some ret
  | _ -> None
```

Partition on condition
nexp ∈ [vmin, vmax]

Scalar Domains

Pointers

(Pointer × MachineNum) @ (LinearRel ∧ Intervals)

$$\textsc{Pointers}(\textsc{Num}) \overset{\text{def}}{=} (\mathcal{V}_{ptr} \to \wp(\mathcal{V}) \cup \{\texttt{NULL}, \texttt{INVALID}\}) \times \textsc{Num}$$

- $\textsc{Pointers}$ is also a **stack** domain.
  1. Each pointer is mapped to the set of pointed variables.
  2. The offset of a pointer $\texttt{p}$ is abstracted by *offset*($\texttt{p}$) in the argument numeric domain $\textsc{Num}$.

Scalar Domains

Pointers

(Pointer × MachineNum) @ (LinearRel ∧ Intervals)

$$\textsc{Pointers}(\textsc{Num}) \stackrel{\text{def}}{=} (\mathcal{V}_{ptr} \to \wp(\mathcal{V}) \cup \{\texttt{NULL}, \texttt{INVALID}\}) \times \textsc{Num}$$

- $\textsc{Pointers}$ is also a **stack** domain.
  1. Each pointer is mapped to the set of pointed variables.
  2. The offset of a pointer p is abstracted by *offset*(p) in the argument numeric domain $\textsc{Num}$.
- $\textsc{Pointers}$ and $\textsc{MachineNum}$ lift a **shared** numeric environment using a cartesian product (×).

  👍 Infer relations between offsets and numeric variables.

Example

```
char a[10] = "hello";
int i = _mopsa_rand(0,9);
char *p = &(a[i]); /* ⟨p ↦ {a}⟩, ⟨i ∈ [0, 9] ∧ offset(p) = i⟩ */
```

Block Domains

(Cells ∧ Strings) ∘ (Pointer × MachineNum) @ (LinearRel ∧ Intervals)

CELLS

- Represent each scalar sub-block individually.
- Tailored for low-level C (casts, type punning, unions).

```
union { uint16 ax; struct { uint8 al; uint8 ah; } bytes; } reg;
reg.ax = 0xabcd; /* ⟨reg[0 : 2] = 43981⟩ */
x = reg.bytes.al; /* ⟨x = 205⟩ */
```

STRINGS

- Keep track of the first `'\0'` byte in the block.
- Useful for checking valid strings.

```
char s[100];
int i, n = _mopsa_rand(1,10);
for(i = 0; i < n; i++) s[i] = 'a';
s[i] = '\0'; /* ⟨n ∈ [1, 10] ∧ length(s) = n = i⟩ */
```

Model of the C Standard Library

- One goal of Mopsa is to analyze large open-source projects.
- C standard library is used in (almost) everywhere.
- Mopsa uses annotations inspired from Frama-C's ACSL.
  1. Specify the behavior of the APIs.
  2. Express safety functional properties.

```
/*$
 * requires: ∃ int i ∈ [0, size(__s) - 1]: __s[i] == 0;
 * ensures : return ∈ [0, size(__s) - 1];
 * ensures : __s[return] == 0;
 * ensures : return > 0  ⟹
 *             ∀ int i ∈ [0, return - 1]: __s[i] != 0;
 */
size_t strlen(const char *__s);
```

Experiments

C Benchmarks

Juliet test suite for Common Weakness Enumerations (CWE)

Category	Description	Lines	Time	Alarms	Coverage
`CWE476`	Null pointer deref.	25k	2min26s	0	100%
`CWE369`	Division by zero	109k	7min20s	699	53%
`CWE190`	Integer overflow	440k	34min57s	760	73%

Python Benchmarks

Official Python performance benchmarks

Program	Lines	Time	Alarms
`fannkuch.py`	59	0.07s	0
`float.py`	63	0.06s	0
`spectral_n.py`	74	0.33s	1
`nbody.py`	157	1.5s	1
`chaos.py`	324	5.9s	1

Conclusion

Features

- MOPSA provides a composotional and flexible architecture for building sound static analyzers.
- Already used in research projects for the analysis on non-trivial C and Python programs.
- Support of different languages, reusable abstract domains with loose coupling.

Limitations and Future Work

- Improve coverage of builtin functions and standard libraries.
- Not tested on large codebases.
- Open questions: backward analysis, shape analysis.