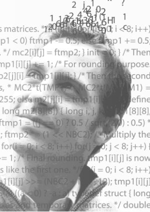
Software Analyzers

Abstract Interpretation of Recursive Logic Definitions for Efficient Runtime Assertion Checking

Thibaut Benjamin, Julien Signoles

TAP 2023

Université Paris-Saclay, CEA, List, F-91120, Palaiseau, France
University of Cambridge



RAC of Arithmetic Properties in E-ACSL

The E-ACSL Plugin

- Plugin part of the Frama-C software (Baudin, Bobot, et al. 2021)
  Collaborative platform for C code verification combining different analysis methods.

The E-ACSL Plugin

- Plugin part of the Frama-C software (Baudin, Bobot, et al. 2021)
  Collaborative platform for C code verification combining different analysis methods.
- Specification language: ACSL (Baudin, Filliâtre, et al. n.d.)

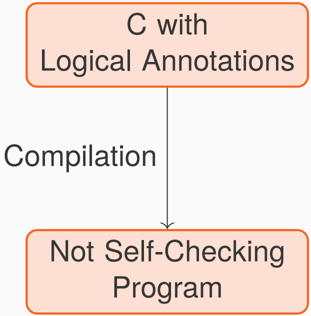
The E-ACSL Plugin

- Plugin part of the Frama-C software (Baudin, Bobot, et al. 2021)
  Collaborative platform for C code verification combining different analysis methods.
- Specification language: ACSL (Baudin, Filliâtre, et al. n.d.)
- E-ACSL is a runtime assertion checker
  Give it a program with assertions, and it gives you a monitored program

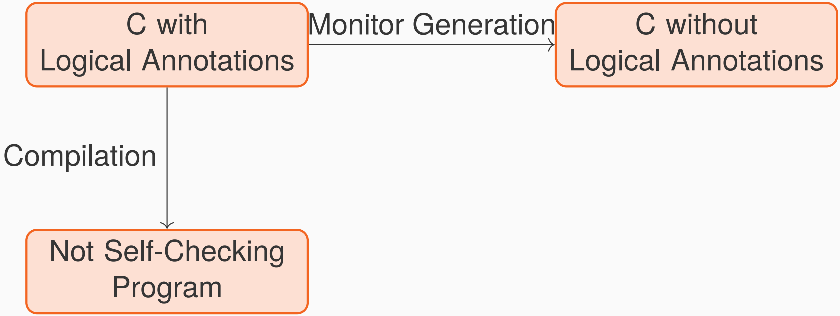
Basic Principle of Inline Monitoring

C with
Logical Annotations

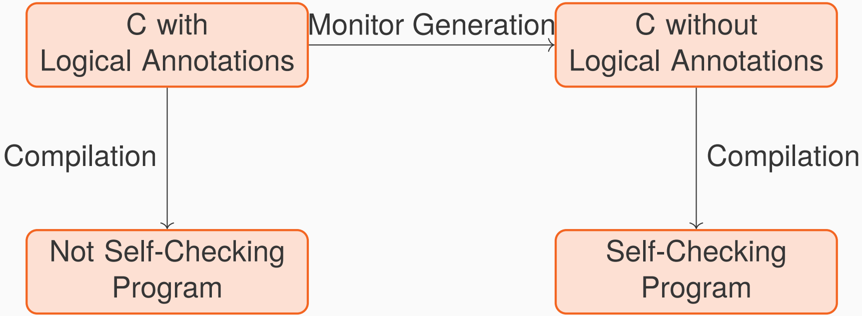
Basic Principle of Inline Monitoring



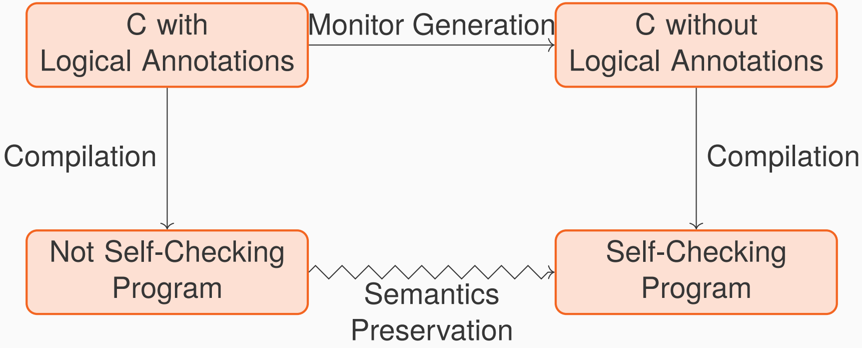
Basic Principle of Inline Monitoring



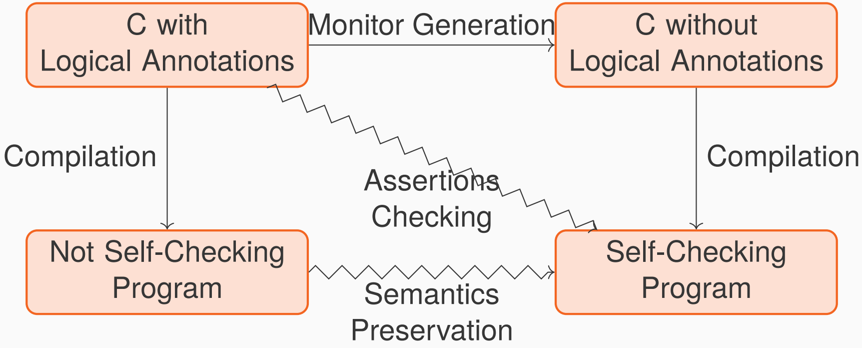
Basic Principle of Inline Monitoring



Basic Principle of Inline Monitoring



Basic Principle of Inline Monitoring



Minimal Example

```
int main (){
  int x = 5;
  //@ assert x + 1 == 6;
  return 0;
}
```

Minimal Example

```c
int main (){
  int x = 5;
  //@ assert x + 1 == 6;
  return 0;
}
```

→

```c
int main (){
  int x = 5;
  assert (x + 1 == 6);
  return 0;
}
```

Desired Properties

- Transparency : When all the assertions are satisfied, the presence of the monitor should not alter the functional behaviour of the program.

Desired Properties

- Transparency : When all the assertions are satisfied, the presence of the monitor should not alter the functional behaviour of the program.
- Soundness : When an assertion is violated, the program should accurately report it.

Desired Properties

- Transparency : When all the assertions are satisfied, the presence of the monitor should not alter the functional behaviour of the program.
- Soundness : When an assertion is violated, the program should accurately report it.
- Correctness : Transparency + Soundness

Desired Properties

- Transparency : When all the assertions are satisfied, the presence of the monitor should not alter the functional behaviour of the program.
- Soundness : When an assertion is violated, the program should accurately report it.
- Correctness : Transparency + Soundness
- Efficiency : The presence of the monitor should induce a reasonable overhead in time and resources.

Desired Properties

- Transparency : When all the assertions are satisfied, the presence of the monitor should not alter the functional behaviour of the program.
- Soundness : When an assertion is violated, the program should accurately report it.
- Correctness : Transparency + Soundness
- Efficiency : The presence of the monitor should induce a reasonable overhead in time and resources.
- Expressiveness : The more properties the monitor can handle the better

Today's scope

- E-ACSL supports in particular (Signoles 2022):
  - Arithmetic Assertions

Today's scope

- E-ACSL supports in particular (Signoles 2022):
    - Arithmetic Assertions
    - User-defined (Mutually Recursive) Functions and Predicates

Today's scope

- E-ACSL supports in particular (Signoles 2022):
  - Arithmetic Assertions
  - User-defined (Mutually Recursive) Functions and Predicates
  - Quantifications on Finite Domains

Today's scope

- E-ACSL supports in particular (Signoles 2022):
  - Arithmetic Assertions
  - User-defined (Mutually Recursive) Functions and Predicates
  - Quantifications on Finite Domains
  - Pointer Status and Memory Properties

Today's scope

- E-ACSL supports in particular (Signoles 2022):
  - Arithmetic Assertions
  - User-defined (Mutually Recursive) Functions and Predicates
  - Quantifications on Finite Domains
  - Pointer Status and Memory Properties

Today

Today's scope

- E-ACSL supports in particular (Signoles 2022):
  - Arithmetic Assertions
  - User-defined (Mutually Recursive) Functions and Predicates
  - Quantifications on Finite Domains
  - Pointer Status and Memory Properties

Today

- Specifically, we consider the following constructs:

Today's scope

- E-ACSL supports in particular (Signoles 2022):
  - Arithmetic Assertions
  - User-defined (Mutually Recursive) Functions and Predicates
  - Quantifications on Finite Domains
  - Pointer Status and Memory Properties

Today

- Specifically, we consider the following constructs:
  - Comparison and arithmetic operators
    `==, !=, <, <=, >, >=,   +, *, -, /`

Today's scope

- E-ACSL supports in particular (Signoles 2022):
  - Arithmetic Assertions
  - User-defined (Mutually Recursive) Functions and Predicates
  - Quantifications on Finite Domains
  - Pointer Status and Memory Properties

Today

- Specifically, we consider the following constructs:
  - Comparison and arithmetic operators
    `==, !=, <, <=, >, >=,   +, *, -, /`
  - Conditionals $p \, ? \, t_1 : t_2$

Today's scope

- E-ACSL supports in particular (Signoles 2022):
  - Arithmetic Assertions
  - User-defined (Mutually Recursive) Functions and Predicates
  - Quantifications on Finite Domains
  - Pointer Status and Memory Properties

Today

- Specifically, we consider the following constructs:
  - Comparison and arithmetic operators
    `==, !=, <, <=, >, >=,   +, *, -, /`
  - Conditionals $p \;?\; t_1 \;:\; t_2$
  - User-defined Functions and Predicates

```
logic integer f (integer x) = t     ...     f(y)
predicate p (integer x) = b         ...     p(y)
```

Related Works

- Formalisation of RAC : for JML (Cheon 2003)
  No mathematical integer at the time
  No user-defined logic functions

Related Works

- Formalisation of RAC : for JML (Cheon 2003)
  No mathematical integer at the time
  No user-defined logic functions

- Formlisation effort on E-ACSL
  In, particular: memory properties (Ly et al. 2020)

Correctness vs. Efficiency

Arithmetic Overflow Issues

Naive Approach:

```
// x is an int
//@ assert (x+1 == 0);
```

→

```
// x is an int
assert (x+1 == 0);
```

Arithmetic Overflow Issues

Naive Approach:

```
// x is an int
//@ assert (x+1 == 0);
```

$\rightarrow$

```
// x is an int
assert (x+1 == 0);
```

+ in the ACSL language
  ↳ mathematical integers

Arithmetic Overflow Issues

Naive Approach:

```
// x is an int
//@ assert (x+1 == 0);
```

→

```
// x is an int
assert (x+1 == 0);
```

- + in the ACSL language
  ↳ mathematical integers

- + in the C language
  ↳ machine integers

Arithmetic Overflow Issues

Naive Approach:

```
// x is an int
//@ assert (x+1 == 0);
```

→

```
// x is an int
assert (x+1 == 0);
```

- in the ACSL language
  ↳ mathematical integers
- in the C language
  ↳ machine integers

What if $x = 2^{31} - 1$?

Arithmetic Overflow Issues

Naive Approach:

```
// x is an int
//@ assert (x+1 == 0);
```

→

```
// x is an int
assert (x+1 == 0);
```

+ in the ACSL language
  ↳ mathematical integers

+ in the C language
  ↳ machine integers

What if x$= 2^{31} - 1$?

$2^{31} \stackrel{?}{=} 0$

false

Arithmetic Overflow Issues

Naive Approach:

```
// x is an int
//@ assert (x+1 == 0);
```

$\rightarrow$

```
// x is an int
assert (x+1 == 0);
```

+ in the ACSL language
  ↳ mathematical integers

+ in the C language
  ↳ machine integers

What if $x = 2^{31} - 1$?

$2^{31} \stackrel{?}{=} 0$

false

Arithmetic Overflow

Undefined Behavior

Solution: Arbitrary Precision Arithmetic

A correct translation generated using the GMP library:

```
// x is an int
//@ assert (x+1 == 0);
```

```
// x is an int
mpz_t y, z, o, r;
mpz_init_set_si(y, x);
mpz_init_set_si(o, 1);
mpz_init(r);
mpz_add(r, y, o);
mpz_init_set_si(z, 0);
int c = mpz_cmp(r, 0);
assert (c == 0);
mpz_clear(y);
mpz_clear(z);
mpz_clear(o);
mpz_clear(r);
```

The Dilemma

	machine integers	GMP integers
correct		✓
efficient	✓	
generates simple code	✓	

The Dilemma

	machine integers	GMP integers
correct		✓
efficient	✓	
generates simple code	✓	

Can we get the best of both worlds?

Central Idea

Run a static analysis on the annotation, to infer the runtime size of each term

Central Idea

Run a static analysis on the annotation, to infer the runtime size of each term

- If the term is small enough $\rightarrow$ machine integers

Central Idea

Run a static analysis on the annotation, to infer the runtime size of each term

- If the term is small enough $\rightarrow$ machine integers
- Otherwise $\rightarrow$ GMP integers

Central Idea

Run a static analysis on the annotation, to infer the runtime size of each term

- If the term is small enough $\rightarrow$ machine integers
- Otherwise $\rightarrow$ GMP integers

Translating annotations this way in the presence of user-defined functions is challenging.

Out of scope for today (Benjamin and Signoles 2023)

Interval Arithmetic

Build the static analysis with the following constraints

- Over-approximate the possible values, no need for precision

Interval Arithmetic

Build the static analysis with the following constraints

- Over-approximate the possible values, no need for precision
- Give an answer quickly in every situation

Interval Arithmetic

Build the static analysis with the following constraints

- Over-approximate the possible values, no need for precision
- Give an answer quickly in every situation

Interval Arithmetic

Build the static analysis with the following constraints

- Over-approximate the possible values, no need for precision
- Give an answer quickly in every situation

↳ Interval arithmetic, with rules like

$$\frac{\vdash t : [a, b] \qquad \vdash u : [c, d]}{\vdash t + u : [a + c; b + d]}$$

Abstract Interpretation for User-Defined Logic Functions

Dealing with Function

- This section deals with the user-defined functions and predicates.

Dealing with Function

- This section deals with the user-defined functions and predicates.
- Use standard techniques from abstract interpretation (Cousot 2022)

Dealing with Function

- This section deals with the user-defined functions and predicates.
- Use standard techniques from abstract interpretation (Cousot 2022)
- Adapt them in the context of efficient RAC

Dealing with Function

- This section deals with the user-defined functions and predicates.
- Use standard techniques from abstract interpretation (Cousot 2022)
- Adapt them in the context of efficient RAC
- We only present the case of functions.
  Predicates are the particular case of Boolean-valued functions.

Translation by Specialisation

- Specialise each logic function into its own C function

Translation by Specialisation

- Specialise each logic function into its own C function
- The same logic function may be specialised several times

Translation by Specialisation

- Specialise each logic function into its own C function
- The same logic function may be specialised several times
- Example:

```
//@ logic integer f (integer x) = x * 1024
...
//@ assert f(2097152) > f(256)
```

Specialise using GMP on the RHS, and `int` on the LHS

Inference in Context

- *Context* = mapping: formal parameter $\rightarrow$ interval

Inference in Context

- *Context* = mapping: formal parameter $\rightarrow$ interval
- Infer the interval of every term in context
  $\Gamma \vdash t : [a, b]$

Inference in Context

- *Context* = mapping: formal parameter $\rightarrow$ interval
- Infer the interval of every term in context
  $\Gamma \vdash t : [a, b]$
- Infer the interval of the body of a function assuming an interval for every argument :

```
/*@ logic integer f
    (integer x) = t */
...
//@ assert f(u) = 5
```

Inference in Context

- *Context* = mapping: formal parameter $\rightarrow$ interval
- Infer the interval of every term in context
  $\Gamma \vdash t : [a, b]$
- Infer the interval of the body of a function assuming an interval for every argument :

```
/*@ logic integer f
    (integer x) = t */
...
//@ assert f(u) = 5
```

$$\frac{\vdash u : J \qquad \{x : J\} \vdash t : I}{\vdash f(u) : I}$$

Recursive Functions

- ACSL allows for recursive functions

Recursive Functions

- ACSL allows for recursive functions

- Just apply the previous rule until you stop?

$$\frac{\vdash u : J \qquad \{x : J\} \vdash t : I}{\vdash f(u) : I}$$

Recursive Functions

- ACSL allows for recursive functions
- Just apply the previous rule until you stop?

$$\frac{\vdash u : J \qquad \{x : J\} \vdash t : I}{\vdash f(u) : I}$$

- This might not terminate, or converge slowly!
  It may not terminate on well-defined functions. Additionally, ACSL does not restrict syntactically the recursion schemes

Recursive Functions

- ACSL allows for recursive functions
- Just apply the previous rule until you stop?

$$\frac{\vdash u : J \qquad \{x : J\} \vdash t : I}{\vdash f(u) : I}$$

- This might not terminate, or converge slowly!
  It may not terminate on well-defined functions. Additionally, ACSL does not restrict syntactically the recursion schemes
- Way too many specialisations
  Essentially unrolls all recursive calls

Computing Fixpoints (I)

- 2 fixpoint computations
  - For the function's arguments

Computing Fixpoints (I)

- 2 fixpoint computations
  - For the function's arguments
  - For the function's output $(\vdash_f)$

Computing Fixpoints (I)

- 2 fixpoint computations
  - For the function's arguments
  - For the function's output $(\vdash_f)$
  - Those computations are simultaneous and intricate

Computing Fixpoints (I)

- 2 fixpoint computations
  - For the function's arguments
  - For the function's output ($\vdash_f$)
  - Those computations are simultaneous and intricate

- Function application = inductive case for argument fixpoint:

$$\frac{\Gamma|\{f : J \to I'\} \vdash u : J' \qquad \{f : J \cup J' \to I'\} \vdash_f f : I}{\Gamma|\{f : J \to I'\} \vdash f(u) : I}$$

Computing Fixpoints (I)

- 2 fixpoint computations
  - For the function's arguments
  - For the function's output $(\vdash_f)$
  - Those computations are simultaneous and intricate

- Function application = inductive case for argument fixpoint:

$$\frac{\Gamma|\{f : J \to I'\} \vdash u : J' \qquad \{f : J \cup J' \to I'\} \vdash_f f : I}{\Gamma|\{f : J \to I'\} \vdash f(u) : I}$$

- Base case for argument fixpoint

$$\frac{\Gamma|\{f : J \to I\} \vdash u : J' \qquad J' \subseteq J}{\Gamma|\{f : J \to I\} \vdash f(u) : I}$$

Computing Fixpoints (II)

Computing the interval output of a recursive function:

- Base case :

$$\frac{\{x : J\} | \{f : J \to I\} \vdash t : I' \qquad I' \subseteq I}{\{f : J \to I\} \vdash_f f : I}$$

Computing Fixpoints (II)

Computing the interval output of a recursive function:

- Base case :

$$\frac{\{x : J\} | \{f : J \to I\} \vdash t : I' \qquad I' \subseteq I}{\{f : J \to I\} \vdash_f f : I}$$

- Inductive case :

$$\frac{\{x : J\} | \{f : J \to I'\} \vdash t : I'' \qquad \{f : J \to I' \cup I''\} \vdash_f f : I}{\{f : J \to I'\} \vdash_f f : I}$$

Widening to Speed Up Convergence

- The fixpoint converge slowly, still may not terminate

Widening to Speed Up Convergence

- The fixpoint converge slowly, still may not terminate

- But they are very (too?) precise
  Sacrifice some precision for the sake of efficiency

Widening to Speed Up Convergence

- The fixpoint converge slowly, still may not terminate
- But they are very (too?) precise
  Sacrifice some precision for the sake of efficiency
- Use *widening*: classic technique in abstract interpretation.
  Operator ∇, s.t. $I \nabla I'$ contains $I \cup I'$ (+ additional conditions)

Widening to Speed Up Convergence

- The fixpoint converge slowly, still may not terminate
- But they are very (too?) precise
  Sacrifice some precision for the sake of efficiency
- Use *widening*: classic technique in abstract interpretation.
  Operator ∇, s.t. $I \nabla I'$ contains $I \cup I'$ (+ additional conditions)
- Replace $\cup$ by ∇ in the previous rules

Widening to Speed Up Convergence

- The fixpoint converge slowly, still may not terminate
- But they are very (too?) precise
  Sacrifice some precision for the sake of efficiency
- Use *widening*: classic technique in abstract interpretation.
  Operator ∇, s.t. $I \nabla I'$ contains $I \cup I'$ (+ additional conditions)
- Replace $\cup$ by ∇ in the previous rules
- In practice: $I \nabla I'$ extends I to the next machine integer boundary in the "direction of the growth"
  E.g. $[0, 5] \nabla [1, 6] = [0, 2^{31} - 1]$

Benchmarks

- This system is implemented and E-ACSL; we use it for benchmarks.

Benchmarks

- This system is implemented and E-ACSL; we use it for benchmarks.
- Benchmarks were run on minimal examples, since execution quickly fails without using this analysis.

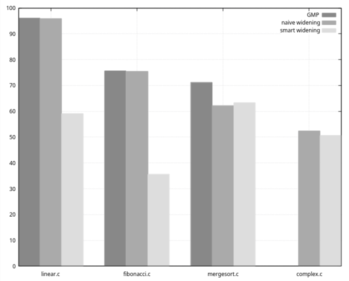


Figure: Evaluation of Monitor Efficiency.

Formal Guarantees

abstract semantics = interval inferred for a logic term in our system
concrete semantics = actual semantics of the ACSL language.

Theorem

Every term has an abstract semantics, and if the term has a concrete semantics, it is necessarily an element of the abstract semantics.

Conclusion

- Static analysis based on abstract interpretation in the interval domains for efficient RAC.

Conclusion

- Static analysis based on abstract interpretation in the interval domains for efficient RAC.

- 2 simultaneous fixpoint algorithms given by 4 inference rules.
  Proving formal correctness

Conclusion

- Static analysis based on abstract interpretation in the interval domains for efficient RAC.
- 2 simultaneous fixpoint algorithms given by 4 inference rules.
  Proving formal correctness
- Using widening to speed up convergence
  Terminates quickly even on ill-formed functions.

Conclusion

- Static analysis based on abstract interpretation in the interval domains for efficient RAC.
- 2 simultaneous fixpoint algorithms given by 4 inference rules.
  Proving formal correctness
- Using widening to speed up convergence
  Terminates quickly even on ill-formed functions.
- Implemented and benchmarked in Frama-C/E-ACSL

Perspectives

- Possibility to improve widening
  Possibly account for off-by-one? function-specific widening?

Perspectives

- Possibility to improve widening
  Possibly account for off-by-one? function-specific widening?

- Switch GMP/machine integers at runtime? (cf. ZArith in OCaml)
  if a function has to compute in GMP once after many iterations in machine integers.

Perspectives

- Possibility to improve widening
  Possibly account for off-by-one? function-specific widening?

- Switch GMP/machine integers at runtime? (cf. ZArith in OCaml)
  if a function has to compute in GMP once after many iterations in machine integers.

- Include this work with other aspects of E-ACSL
  Dealing with rational numbers, undefinedness, memory property

Perspectives

- Possibility to improve widening
  Possibly account for off-by-one? function-specific widening?

- Switch GMP/machine integers at runtime? (cf. ZArith in OCaml)
  if a function has to compute in GMP once after many iterations in machine integers.

- Include this work with other aspects of E-ACSL
  Dealing with rational numbers, undefinedness, memory property

- Apply this idea to other systems dealing with integers
  E.g. Simulation software

THANK YOU

References I

- Baudin, P., F. Bobot, et al. (2021). "The Dogged Pursuit of Bug-Free C Programs: The Frama-C Software Analysis Platform". In: *Communications of the ACM*.
- Baudin, P., J-C. Filliâtre, et al. (n.d.). *ACSL: ANSI/ISO C Specification Language*. Tech. rep. CEA List and Inria. URL: `https://frama-c.com/download/acsl.pdf`.
- Signoles, J. (2022). *E-ACSL Version 1.18. Implementation in Frama-C Plug-in E-ACSL 26.1*. `http://frama-c.com/download/e-acsl/e-acsl-implementation.pdf`.
- Cheon, Y. (2003). "A runtime assertion checker for the Java Modeling Language". PhD thesis. Iowa State University.
- Ly, D. et al. (2020). "Verified Runtime Assertion Checking for Memory Properties". In: *International Conference on Tests and Proofs (TAP)*.

References II

Benjamin, T. and J. Signoles (2023). "Formalizing an Efficient Runtime Assertion Checker for an Arithmetic Language with Functions and Predicates". In: *Symposium on Applied Computing*.

Cousot, Patrick (2022). *Principles of Abstract Interpretation*. MIT Press.