Abstract Interpretation and Properties of C Programs

Julien Signoles

Software Reliability & Security Lab

École des Jeunes Chercheurs en Programmation 2019 (EJCP’19)

June 26, 2019
Context

Software and Trust

Frama-C

ACSL

Frama-C for Software Assessment

Eva Plug-in

Derived Analyses

Conclusion
Software is Eating the World

do you trust it?

SOURCE LINES OF CODE (MILLIONS)

- Daily code output
- 787 Dreamliner
- F-22 Raptor
- Chevrolet Volt
- Google Chrome
- Google infrastructure
- Linux-3.6
- Windows XP

2000
State of the Art: Normative Documents

Normative documents

▷ safety
  ▷ EN 50128 for rail safety
  ▷ DO-178C for avionics safety
  ▷ IEC ISO 61513 for nuclear plants
  ▷ IEC 61508 reference

▷ security
  ▷ Common Criteria / ISO 15408
normative documents rely on **tests** and **code reviews** as verification techniques

tests and code reviews are **too costly** beyond a certain size and coverage criterion

need for **correct** tools
  - detect all potential issues
  - may issue spurious warnings
  - impossible for an automated tool to warn for all real issues and only for them (Rice Theorem, 1953)

several correct verification techniques deal with this issue
Verification Techniques

- Normative documents rely on tests and code reviews as verification techniques.
- Tests and code reviews are too costly beyond a certain size and coverage criterion.
- Need for correct tools:
  - Detect all potential issues.
  - May issue spurious warnings.
  - Impossible for an automated tool to warn for all real issues and only for them (Rice Theorem, 1953).
- Several correct verification techniques deal with this issue.
Verification Techniques

- normative documents rely on tests and code reviews as verification techniques
- tests and code reviews are too costly beyond a certain size and coverage criterion
- need for correct tools
  - ✓ detect all potential issues
  - ✗ may issue spurious warnings
    - ✗ impossible for an automated tool to warn for all real issues and only for them (Rice Theorem, 1953)
- several correct verification techniques deal with this issue
Verification Techniques

- normative documents rely on tests and code reviews as verification techniques
- tests and code reviews are too costly beyond a certain size and coverage criterion
- need for correct tools
  - ✓ detect all potential issues
  - ✗ may issue spurious warnings
  - ✗ impossible for an automated tool to warn for all real issues and only for them (Rice Theorem, 1953)
- several correct verification techniques deal with this issue
normative documents rely on **tests and code reviews** as verification techniques

tests and code reviews are **too costly** beyond a certain size and coverage criterion

need for **correct** tools

- ✓ detect all potential issues
- ❌ may issue spurious warnings
- ❌ impossible for an automated tool to warn for all real issues and only for them (*Rice Theorem, 1953*)

several correct verification techniques deal with this issue
Verification Techniques

cont’d

- **model checking** analyses a program’s model
  - ✔ automatic
  - ✗ state explosion

- **runtime verification** analyses a concrete run
  - ✔ automatic
  - ✗ not exhaustive

- **deductive verification** generates logic formulas (aka proof obligations) to be proven by other means
  - ✔ powerful
  - ✗ not fully automatic

- **abstract interpretation** over-approximates program’s behaviors
  - ✔ (almost) automatic
  - ✗ false alarms
Verification Techniques cont’d

- **model checking** analyses a program’s model
  - ✓ automatic
  - ✗ state explosion

- **runtime verification** analyses a concrete run
  - ✓ automatic
  - ✗ not exhaustive

- **deductive verification** generates logic formulas (aka proof obligations) to be proven by other means
  - ✓ powerful
  - ✗ not fully automatic

- **abstract interpretation** over-approximates program’s behaviors
  - ✓ (almost) automatic
  - ✗ false alarms
Verification Techniques cont’d

- **model checking** analyses a program’s model
  - ✔ automatic
  - ✗ state explosion

- **runtime verification** analyses a concrete run
  - ✔ automatic
  - ✗ not exhaustive

- **deductive verification** generates logic formulas (aka proof obligations) to be proven by other means
  - ✔ powerful
  - ✗ not fully automatic

- **abstract interpretation** over-approximates program’s behaviors
  - ✔ (almost) automatic
  - ✗ false alarms
Verification Techniques

cont’d

- **model checking** analyses a program’s model
  - ✔ automatic
  - ✗ state explosion

- **runtime verification** analyses a concrete run
  - ✔ automatic
  - ✗ not exhaustive

- **deductive verification** generates logic formulas (aka proof obligations) to be proven by other means
  - ✔ powerful
  - ✗ not fully automatic

- **abstract interpretation** over-approximates program’s behaviors
  - ✔ (almost) automatic
  - ✗ false alarms
Context

Software and Trust

Frama-C

ACSL

Frama-C for Software Assessment

Eva Plug-in

Derived Analyses

Conclusion
Credits

- Benjamin Monate
- Pascal Cuq
- Virgile Prevosto
- Julien Signoles
- Loïc Correnson
- Boris Yakobowsk
- François Bobot
- Patrick Baudin
- Anne Pacalet
- Philippe Herrmann
- David Bühler
- Valentin Perrelle
- André Maroneze
- 8 other current developers
- 17 other past developers

More information

- Frama-C User Manual
- Frama-C Plug-in Development Guide
Frama-C Distribution

Framework for analyses of source code written in ISO 99 C

[Kirchner & al in FAC’15]

- developed by CEA LIST since 2005
- almost open source (LGPL 2.1)
- first open-source release aka Hydrogen in 2008
- last open-source release aka 19-Potassium in June 2019

http://frama-c.com

- also proprietary extensions and distributions
- targets both academic and industrial usages
Frama-C, a Collection of Tools

Several tools inside a single platform

▶ plug-in architecture à la Eclipse [S. @F-IDE’15]
▶ tools provided as plug-ins
  ▶ 28 plug-ins in the latest open source distribution
  ▶ outside open source plug-ins (e.g. Frama-Clang)
  ▶ close source plug-ins, either at CEA (> 20) or outside
▶ plug-ins connected to a kernel
  ▶ provides an uniform setting
  ▶ provides general services
  ▶ synthesizes useful information
▶ analyzer combinations [Correnson & S. @FMICS’12]
Several tools inside a single platform

- **plug-in architecture à la Eclipse** [S. @F-IDE’15]
- tools provided as plug-ins
  - 28 plug-ins in the latest open source distribution
  - outside open source plug-ins (e.g. Frama-Clang)
  - close source plug-ins, either at CEA (> 20) or outside
- **plug-ins connected to a kernel**
  - provides an uniform setting
  - provides general services
  - synthesizes useful information
  - analyzer combinations [Correnson & S. @FMICS’12]
Frama-C, a Development Platform

- developed in OCaml
- was based on Cil [Necula & al @CC’02]
- library dedicated to analysis of C code
  development of plug-ins by third party
- powerful low-cost analyser
- dedicated plug-in for specific task (coding rules verifier)
- dedicated plug-in for fine-grain parameterization
- extension of existing analysers
Frama-C, a Development Platform

- developed in OCaml
- was based on Cil [Necula & al @CC’02]
- library dedicated to analysis of C code
  
  development of plug-ins by third party

- powerful low-cost analyser
- dedicated plug-in for specific task (coding rules verifier)
- dedicated plug-in for fine-grain parameterization
- extension of existing analysers
Context

Software and Trust
Frama-C
ACSL
Frama-C for Software Assessment

Eva Plug-in

Derived Analyses

Conclusion
Presentation

- based on the notion of **contract**, like in Eiffel
- allows users to specify **functional properties** of their code
- allows communication between various plugins
- independent from a particular analysis
- [https://github.com/acsl-language/acsl/](https://github.com/acsl-language/acsl/)

Basic Components

- first-order logic
- pure C expressions
- C types + $\mathbb{Z}$ (integer) and $\mathbb{R}$ (real)
- built-ins predicates and logic functions, e.g. over pointers.
▶ no overflow: all operations are done over \( \mathbb{Z} \)

▶ simple specification of absence of overflows (undefined behavior): \( \text{INT\_MIN} \leq x + y \leq \text{INT\_MAX} \)

▶ still bounded (machine) integers if required

  ▶ \( (\text{int})z \equiv z \mod 2^{8 \times \text{sizeof(int)}} \)

  ▶ \( \text{INT\_MIN} \leq (\text{int})z \leq \text{INT\_MAX} \)
Operations over $\mathbb{R}$: infinite precision

\[\text{round\_double}(r, \text{NearestEven})\] to explicitly choose rounding mode

predicates $\text{is\_finite}(d), \text{is\_plus\_infinity}(d), \text{is\_NaN}(d), \ldots$

function $\text{exact}(x)$: the value that C variable $x$ would have if all computations had been done using $\mathbb{R}$.

$\text{round\_error}$ is the distance between $x$ and $\text{exact}(x)$

typical specification:

$\text{round\_error}(\text{result}) \leq \text{acceptable\_limit}$
struct S {
    short x;
    int y;
} s[2];

\valid(s[0]+(0 .. 1))
\valid(((char*)s[0] + (0 .. 15))
!\initialized(*((char*)s[0].x+2))
\block_length(s[0]) == 16
\base_addr(s[0].y) == s
\offset(s[1].y) == 12
\separated(s[0], s[1])
Question

Assume \( \text{valid}(p+(0 .. 2)) \) with \( p \) a pointer to \textit{int}, and \texttt{sizeof(int)}==4. What is the most precise information we can say about \texttt{block_length(p)}?

Answers

- a \( \text{block_length}(p) == 2 \)
- b \( \text{block_length}(p) >= 3 \)
- c \( \text{block_length}(p) == 8 \)
- d \( \text{block_length}(p) >= 8 \)
- e \( \text{block_length}(p) == 12 \)
- f \( \text{block_length}(p) >= 12 \)
/*@ requires R(x);

ensures E(result,x);

behavior extra:
    assumes A(x);
    ensures more_result(result,x);
*/

int f(int x);
Function Contract

/*@ requires R(x); */

ensures E(result,x);

behavior extra:

assumes A(x);

ensures more_result;

int f(int x);
/*@ requires R(x); 
ensures E(result, x);

behavior extra:
  assumes A(x);
  ensures more_result(result, x);
*/

int f(int x);
Question

Assume an ACSL function `acsl_strlen` that returns the offset of the first '\0' char if it exists and -1 otherwise. What would be an appropriate requires for the standard library function `size_t strlen(const char* s)`?

Answers

- a) `acsl_strlen(s) >= 0`
- b) `acsl_strlen(s) >=0 && \valid(s + (0 .. acsl_strlen(s)))`
- c) `\valid(s + (0 .. acsl_strlen(s)))`
- d) `acsl_strlen(s) >= 0 && \valid(s)`
/*@ assert p == NULL || \valid(p); */
if (p) { *p = 42; }

if (0) { /*@ assert \false; */ exit (1); }
Assess a property at a given point

```c
/*@ assert p == NULL || valid(p); */
if (p) {
    *p = 42;
}

if (0) {
    /*@ assert false; */
    exit (1);
}
```
Assess a property at a given point

/*@ assert p == NULL || valid(p); */
if (p) { *p = 42; }

if (0) { /*@ assert \false; */ exit (1); }

Indicates dead code
Context

Software and Trust
Frama-C
ACSL
Frama-C for Software Assessment

Eva Plug-in

Derived Analyses

Conclusion
What is verified by Frama-C?

**Code Properties**
- Functional properties (contract)
- Absence of run-time errors
- Termination
- Dependencies
- Non-interference
- Temporal properties

**Perimeter of the verification**
- Which part of the code is under analysis?
- Which initial context?

**Trusted Code Base**
- ACSL Axioms
- Hypotheses made by analyzers
- Stub Functions
- Frama-C itself
Further Reading
Frama-C

- manuals & tutorials
  - Frama-C User Manual
  - Frama-C Plug-in Development Guide
  - ACSL By Example (Introduction to ACSL and WP proofs)

- papers
Context

Eva Plug-in

Abstract Interpretation
Basics
Refining the Analysis
Setting Analysis Context

Derived Analyses

Conclusion
Abstract interpretation is about

- abstracting away information
- ensuring answer in a reasonable time
- while retaining adequate precision
- and guaranteeing correct answers
Abstract interpretation is about:

- abstracting away information
- ensuring answer in a reasonable time
- while retaining adequate precision
- and guaranteeing correct answers
Abstract interpretation is about

▶ abstracting away information
▶ ensuring answer in a reasonable time
▶ while retaining adequate precision
▶ and guaranteeing correct answers
Abstract interpretation is about

- abstracting away information
- ensuring answer in a reasonable time
- while retaining adequate precision
- and guaranteeing correct answers
Abstract Interpretation in two pictures

- abstracting away information
- ensuring answer in a reasonable time
- while retaining adequate precision
- and guaranteeing correct answers
Abstract Interpretation in a Nutshell

- replace all possible concrete executions ...
- ... by one abstract execution
- analysis is guaranteed to terminate
- over-approximation and false alarms
- trade-off between precision and computation time
Abstract Interpretation in a Nutshell

- replace all possible concrete executions ...
- ... by one abstract execution
- analysis is guaranteed to terminate
- over-approximation and false alarms
- trade-off between precision and computation time
Abstract Interpretation in a Nutshell

- replace all possible concrete executions ...
- ... by one abstract execution
- analysis is guaranteed to terminate
- over-approximation and false alarms
- trade-off between precision and computation time
Abstract Interpretation in a Nutshell

- replace all possible concrete executions ...
- ... by one abstract execution
- analysis is guaranteed to terminate
- over-approximation and false alarms
- trade-off between precision and computation time
Abstract Interpretation in a Nutshell

▶ replace all possible concrete executions ...

▶ ... by one abstract execution

▶ analysis is guaranteed to terminate

▶ over-approximation and false alarms

▶ trade-off between precision and computation time
mpz_ptr syracuse(mpz_t res, const mpz_t arg) {
    mpz_t x;
    mpz_init_set_ui(res,0UL);
    mpz_init_set(x,arg);
    while (mpz_cmp_ui(x,1UL)>0) {
        mpz_out_str(stdout,10,x);
        putchar(\n');
        if (mpz_odd_p(x)) {
            mpz_mul_ui(x,x,3UL);
            mpz_add_ui(x,x,1UL);
        } else {
            mpz_cdiv_q_ui(x,x,2UL);
        }
        mpz_add_ui(res,res,1UL);
    }
    mpz_clear(x);
    return res;
}
Abstract Interpretation

Fundamental Notions

- **abstract domain**: over-approximation of possible values
  - intervals, e.g. $x \in [0; 9]$
  - octagons, e.g. $0 \leq x - y \leq 20$
- **abstract transfer function**: propagates an abstract domain over a CFG’s edge
- **join operator**: merges abstract states for nodes with multiple predecessors
  - $[0; 9] \sqcup [12; 42] = [0; 42]$
Abstract Interpretation
Correctness and Termination

- **correction**: do we include all concrete states in the end?
- **termination**: converge in a finite number of steps
- **abstract interpretation**: a systematic way to build correct and terminating analyses [Cousot & Cousot @POPL’77]
Widening

- for loop nodes, state grows slowly at each step
- convergence could require infinite time
- replace $\sqcup$ with **widening operator** $\sqcap$:
  - correctness
  - termination
  - no infinitely growing sequence

$x_0 \sqcap x_1 \sqcap \ldots \sqcap x_n \ldots$

---

$S_1$ (before) \ $S_4$ \ $S_1$ (after)

$y \in [0; 0] \sqcup [1; 1] = [0; 1]$
for loop nodes, state grows slowly at each step

convergence could require infinite time

replace $\sqcup$ with widening operator $\triangledown$:

- correctness
- termination
- no infinitely growing sequence

$x_0 \triangledown x_1 \triangledown \ldots \triangledown x_n \ldots$
Widening

- for loop nodes, state grows slowly at each step
- convergence could require infinite time
- replace $\sqcup$ with widening operator $\triangledown$:

  correctness
  
  termination
  
  no infinitely growing sequence

$x_0 \triangledown x_1 \triangledown \ldots \triangledown x_n \ldots$
Widening

- for loop nodes, state grows slowly at each step
- convergence could require infinite time
- replace \( \sqcup \) with widening operator \( \triangledown \):

**Correctness**

\[ x \sqcup y \sqsubseteq x \triangledown y \]

**Termination**

no infinitely growing sequence

\[ x_0 \triangledown x_1 \triangledown \ldots \triangledown x_n \ldots \]
Widening

- For loop nodes, state grows slowly at each step.
- Convergence could require infinite time.
- Replace $\sqcup$ with widening operator $\nabla$:

  $x \sqcup y \subseteq x \nabla y$

  No infinitely growing sequence.
  $x_0 \nabla x_1 \nabla \ldots \nabla x_n \ldots$

**lower bound stable:** don’t change

**$S_1$ (before) $S_4$ $S_1$ (after)**

$y \in [0, 2]$ $\nabla$ $[1, 3] = [0, +\infty]$
Widening

- for loop nodes, state grows slowly at each step
- convergence could require infinite time
- replace $\sqcup$ with widening operator $\nabla$

**Correctness**

$\forall y \in [0;2] \, \nabla \, \forall y \in [1;3] = [0;\infty]$  

**Upper bound grows: widen interval**

$S_1$ (before)

$S_4$ (after)

Termination

$\exists x_0 \nabla x_1 \nabla \ldots \nabla x_n \ldots$
Widening

- for loop nodes, state grows slowly at each step
- convergence could require infinite time
- replace $\sqcup$ with widening operator $\triangleleft$:
  - correctness
  - termination
- no infinitely growing sequence
- $x_0 \triangleleft x_1 \triangleleft \ldots \triangleleft x_n \ldots$
Recover some precision

- Widening can be very coarse
- Use narrowing after reaching fixpoint:
  - correctness \( y \subseteq (x \triangle y) \subseteq x \)
  - termination no infinitely decreasing sequence
- In practice, very often better to directly improve widening

\[
\begin{align*}
& S_0 \quad \text{arg} \leq 300? \\
& S_1 \quad \text{arg} > 300? \\
& S_2 \quad x = \text{arg}, y = 0 \\
& S_3 \quad y \leq x? \\
& S_4 \quad y > x? \\
& S_5 \quad y = y + 1 \\
& S_6
\end{align*}
\]

- \( y \in [0; 0] \) \( \triangleright \) \( [1; 1] \) = \( [0; +\infty] \)
- widen and propagate new bound
Narrowing

Recover some precision
- Widening can be very coarse
- Use narrowing after reaching fixpoint:
  - correctness  $y \subseteq (x \vartriangle y) \subseteq x$
  - termination  no infinitely decreasing sequence
- In practice, very often better to directly improve widening

$y \in [0; +\infty] \vartriangle [1; 301] = [0; +\infty]$

\(\infty\) may be too much try to narrow it down
Narrowing

Recover some precision

- Widening can be very coarse
- Use narrowing after reaching fixpoint:
  - correctness  \( y \subseteq (x \triangledown y) \subseteq x \)
  - termination no infinitely decreasing sequence

- In practice, very often better to directly improve widening
Narrowing

Recover some precision
- Widening can be very coarse
- Use narrowing after reaching fixpoint:
  correctness $y \subseteq (x \triangle y) \subseteq x$
  termination no infinitely decreasing sequence
- In practice, very often better to directly improve widening

\[
\begin{align*}
\text{arg} &\leq 300? \\
S_0 &\quad \text{arg} > 300? \\
x &= \text{arg} \quad y = 0 \\
S_1 &\quad y \leq x? \\
y > x? &\quad S_5 \\
S_2 &\quad y = y + 1 \\
S_3 &\quad \Delta \\
S_4 &\quad \text{check new fixpoint is reached}
\end{align*}
\]

$S_2 \ (\text{before})$ $\quad S_4 \quad S_2 \ (\text{after})$

$y \in [0;301] \triangle [1;301] = [0;301]$
Narrowing

Recover some precision
- Widening can be very coarse
- Use narrowing after reaching fixpoint:
  - correctness $y \subseteq (x \triangle y) \subseteq x$
  - termination no infinitely decreasing sequence
- In practice, very often better to directly improve widening
Narrowing

Recover some precision
- Widening can be very coarse
- Use narrowing after reaching fixpoint:
  correctness \( y \subseteq (x \triangle y) \subseteq x \)
  termination no infinitely decreasing sequence
- In practice, very often better to directly improve widening
Context

Eva Plug-in
Abstract Interpretation
Basics
Refining the Analysis
Setting Analysis Context

Derived Analyses

Conclusion
Credits
▶ Pascal Cuoq
▶ Boris Yakobowski
▶ David Bühler
▶ André Maroneze
▶ Valentin Perrelle
▶ A few other developers...

More information
▶ Eva Manual
Main Objective

Find the domains of the variables of a program

- based on abstract interpretation
- alarms on operations that may be invalid
- alarms on the specifications that may be invalid
- Correct: if no alarm is raised, no runtime error can occur
Some Eva’ specificities

- Precise handling of pointers

- Several representations for dynamic allocation (precision vs. time)

- Time and memory efficient (as much as achievable)

- Precise enough
  - for proving absence of runtime errors on some critical code
  - to serve as a back-end for other semantical analyses through its API
Abstract Domain

- small set of integers (by default, cardinal $\leq 8$)
- integer interval $\times$ modulo information
- finite floating-point interval

Examples

- $\{0; 40;\} = 0$ or $40$
- $[0..40] =$ an integer between $0$ and $40$ (inclusive)
- $[-\ldots-] =$ any integer (within the bounds of the corresponding integral type)
- $[3..39], 3\%4 = 3, 7, 11, 15, 19, 23, 27, 31, 35$ or $39$
- $[0.25..3.125] =$ floating-point between $0.25$ and $3.125$ (inclusive)
int x, y, t, m; double d;
extern char z; char z1;

void f(int c) {
    if (c) x = 40;
    for (int i = 0; i <= 40; i++) {
        Frama_C_show_each_loop_1(i);
        if (c == i) y = i;
    }
    z1 = z;
    t = z;
    m = 3;
    for (int i = 3; i <= 40; i+=4) {
        if (c == i) m = i;
    }
    if (c) { d = 0.25; } else { d = 3.125; }
}
frama-c -eva -main f 01-integer.c

[eva] 01-integer.c:7: Frama_C_show_each_loop_1: {0}
[eva] 01-integer.c:6: starting to merge loop iterations
[eva] 01-integer.c:7: Frama_C_show_each_loop_1: {0; 1}
[eva] 01-integer.c:7: Frama_C_show_each_loop_1: {0; 1; 2}
[eva] 01-integer.c:7: Frama_C_show_each_loop_1: [0..40]
[eva] 01-integer.c:7: Frama_C_show_each_loop_1: [0..40]
[eva] -------- VALUES COMPUTED --------
[eva:final-states] Values at end of function f:
  x {0; 40}
  y [0..40]
  t [−128..127]
  m [3..39], 3%4
  d [0.25 .. 3.125]
  z1 [——..——]
Integers in Eva Quiz

Question

if x is in the interval [-10 .. 10] before the execution of statement

```
if (x == 0) { y = 14; }
else { y = (x < 0 ? 13 : x + 2); }
```

What is the value associated to y after the statement?

Answers

- a [-8 .. 14]
- b [2 .. 13]
- c [2 .. 14]
- d [3 .. 14]
Memory Address

Base Address

- global variable
- formal parameter of main function
- literal string constant
- NULL
- ...

Addresses

- base address + offset (integer)
- each base has a maximal valid offset
- abstract values are sets of addresses
Examples of Addresses

Precise Base

- `{&p + \{4; 8\}}` = address of \( p \) shifted from 4 or 8 bytes
- `{&"foobar"; }` = address of literal string "foobar" (shifted from 0)
- `{&NULL + \{1024; \}}` = absolute location 1024

Imprecision

- **garbled mix of \&\{x_1; \ldots; x_n\}** = unknown address built upon arithmetic operations over integers and addresses \( x_1; \ldots; x_n \)
- **ANYTHING** = top of the lattice (should not occur in practice)
```c
int *x, *z, *t; const char *y; int p[3];
const char *string = "foobar";

void f(int c) {
    if (c) { x = &p[1]; }
    else { x = &p[2]; }
    y = string;
    z = (int *) 1024;
    t = (int *) ((int)x | 4096);
}
```
[eva] ——— VALUES COMPUTED ———
[eva: final states] Values at end of function f:
x {{ &p{[1], [2]} }}
y {{ "foobar" }}
z {1024}
t {{ garbled mix of &p
    (origin: Arithmetic {02-address.c:13}) }}
Abstract Domain

written address = valid left value

address
× initialized?
× not dangling pointer?

Example

```c
int x, y;
if (e) x = 2;
L: if (e) y = x + 1;
```

▶ At L, we know that x equals 2 iff it has been initialized
▶ Depending on the complexity of e, we know that y equals 3 if x equals 2
int X, Y, *p;
void f(int c) {
    int x, y;
    if (c <= 0) x = 2;
    L: if (c <= 0) y = x + 1; else y = 4;
    X = x;
    Y = y;
    p = c ? &X : &x;
}

int main(int c) {
    f(c);
    if (Y == 4) *p = 3;
    return 0;
}
assert \initialized(&x);
[eva:locals—escaping] 03—address—written.c:15: Warning: locals {x} escaping the scope of f through p
assert !\dangling(&p);
Question

if a is an array of size 3, initialized to 0, and c in [0..2] what would be the content of a after executing the following statement:

```c
if (c) { a[c] = c; } else a[1] = 3;
```

Answers

- **a**
  a[0] IN {0}, a[1] IN {0,1,3}, a[2] IN {0,2}

- **b**
  a[i] IN {0,1,2,3} for all indices

- **c**
  a[0] IN {0}, a[1] IN {0,1,2,3}, a[2] IN {0,1,2}

- **d**
  a[0] IN {0}, a[1] IN {1,3}, a[2] IN {2}
Adding other domains

- new domains can provide additional information:
  - equalities between values
  - values of symbolic locations
  - gauges, affine relations wrt number of loop steps
- possible to add new domains
- inter-domain communication done through queries:

```plaintext
val extract_expr : 
(exp -> value evaluated) 
-> state -> exp 
-> (value * origin) evaluated

val extract_lval : 
(exp -> value evaluated) 
-> state -> lval -> typ -> location 
-> (value * origin) evaluated
```
Context

Eva Plug-in
Abstract Interpretation
Basics
Refining the Analysis
Setting Analysis Context

Derived Analyses

Conclusion
-eva-precision n

**new** in 19.0 Potassium

automatically sets up some Eva options for quick arbitration between precision and speed

from 0 (fastest) to 11 (most precise)
Primary Options

- option **-eva-slevel**: allows Eva to explore $n$ separated paths before joining them

- option **-eva-slevel-function**: same as previous, but for a particular function

- annotation **loop unroll n**: for considering $n$ iterations of a loop separately

For specialists only

- option **-eva-ilevel**: maximum number of elements in the set before conversion into intervals (default = 8)

- option **-eva-plevel**: maximum number of distinct array cells (default = 200)
int S = 0;
int T[5];

int main(void) {
    int i;
    int *p = &T[0];
    for (i = 0; i < 5; i++) {
        S = S + i;
        *p++ = S;
    }
    return S;
}
Driving Eva through Annotations

- ACSL assertions can be used to restrict propagated domains

but only if Eva can interpret them

```
// potentially useful:
/*@ assert x % 2 == 0; */
// useless:
/*@ assert \exists integer y; x == 2 * y; */
```

- Case analysis using disjunctions (combined with -slevel)

```
/*@ assert x <= 0 || x > 0; */
```
int x, y;

void main (int c) {
    if (c) { x = 10; } else { x = 33; }
    if (!c) { x++; } else { x--; }
    if (c <= 0) { y = 42; } else { y = 36; }
    if (c > 0) { y++; } else { y--; }
}
without slevel
  \[ x \; \text{IN} \; \{9; \; 11; \; 32; \; 34\} \]
  \[ y \; \text{IN} \; \{35; \; 37; \; 41; \; 43\} \]

with slevel, no assertion
  \[ x \; \text{IN} \; \{9; \; 11; \; 34\} \]
  \[ y \; \text{IN} \; \{37; \; 41\} \]

with slevel and assertion
  
  \[ /*@ assert c <= 0 || c > 0; */ \]

  [value] Assertion got status valid.
  \[ x \; \text{IN} \; \{9; \; 34\} \]
  \[ y \; \text{IN} \; \{37; \; 41\} \]
Context

Eva Plug-in

Abstract Interpretation
Basics
Refining the Analysis

Setting Analysis Context

Derived Analyses

Conclusion
Which part of the code should be analyzed?

- **main f** starts the analysis at function f
- **-lib-entry** indicates that the initial global context is not 0-initialized

Use of a driver function with some builtins to provide non-determinism:

```c
void f_wrapper() {
    setup_analysis_context();
    f(arg_1, arg_2);
}
```
Example

```c
int search(char *a, char key) {
    char *orig = a;
    while (*a) {
        if (*a == key) return a - orig;
        a++;
    }
    return -1;
}
```
frama-c -eva [...]

[...]

[eva:alarm] 06-context.c:3: Warning: out of bounds read. assert \valid_read(a);
[eva:alarm] 06-context.c:4: Warning: out of bounds read. assert \valid_read(a);
  assert \base_addr(a) \equiv \base_addr(orig);
[eva] ====== VALUES COMPUTED =======
[eva:final-states] Values at end of function search:
  \!a \in \{ &S_a[0], [1] \}
  \!orig \in \{ \text{NULL} ; &S_a[0] \}
  \!\_\text{retres} \in \{-1 ; 0 ; 1\}
```c
#include "__fc_builtin.h"
#include <limits.h>

int search(char* a, char key);

char buffer[1024];

int driver() {
    buffer[1023] = 0;
    char key = Frama_C_interval(CHAR_MIN, CHAR_MAX);
    return search(buffer, key);
}
```
frama-c -eva -main driver -eva-slevel 1024 [...]  
[ ... no alarm ... ]

[eva] ====== VALUES COMPUTED ======
[eva:final-states] Values at end of function search:
  a ∈ {{ &buffer + [0..1023] }}
  orig ∈ {{ &buffer[0] }}
  __retres ∈ [-1..1022]
[eva:final-states] Values at end of function driver:
  Frama_C_entropy_source ∈ [--.--]
  buffer[0..1022] ∈ [--.--]
    [1023] ∈ {0}
  key ∈ [--.--]
Provide an “implementation” for Eva

▶ assumed to match the real implementation

▶ write stub directly in C (aimed at ease of analysis, not performance)

▶ provide an ACSL specification

▶ `-eva-use-spec f`

▶ use an Eva built-in (`-eva-builtin`)

▶ `-eva-builttins-list`
Assumptions made by Eva

Command-line Options

- **-eva-ignore-recursive-calls** assumes recursive calls have no effect
- **-all-rounding-modes** do not assume floating-point computations use same rounding as host machine

ACSL Properties

- alarms emitted by Eva
- annotations with **Unknown** status
  - try to prove them (as soon as they are not too complicated)
  - try to simplify Eva’ state by assuming that they are true
can we guarantee absence of defaults in large system-level code?

- Scada systems of 100+ kloc of C code
- Highest certification requirements (IEC60880 class 1)
- Pinpoint the undefined behaviors and help investigate their cause
- Structural properties on memory separation and cyclic behaviors
- 80% code coverage, 200 alarms
- [Ourghanlian in Nuclear Engineering & Technology, 2015]
Further Reading

- **manuals & tutorials**
  - Eva Manual
  - Frama-C Blog

- **papers**
Context

Eva Plug-in

Derived Analyses

Dependencies

Impact

Slicing

Conclusion
From Plug-in

tied to Eva

for each memory location \texttt{loc} possibly modified, returns its dependencies

i.e. the set of locations whose value might be used in computing the final value of \texttt{loc}

over-approximation: some dependencies might be spurious
int x1, x2, res1, res2;

/*@ assigns \result \from x; */
int do_it_min(int x);

/*@ assigns \result \from x; */
int do_it_max(int x);

int f() {
    if (x1 <= 100) {
        res1 = do_it_min(x1);
    }
    if (x2 >= 1000) {
        res2 = do_it_max(x2);
    }
    res1++;
    res2++;
    return x1 > x2 ? x1 : x2;
}
[from] Function do_it_max:
\result FROM x
[from] Function do_it_min:
\result FROM x
[from] Function f:
res1 FROM x1; res1
res2 FROM x2; res2
\result FROM x1; x2
frama-c -eva -eva-slevel 1000 -deps \
   Keccak-simple.c KeccakNISTInterface.c \
   KeccakSponge.c KeccakF-1600-reference.c test.c

[from] Function rho:
   context.state[0..199]
   FROM context.state[0..199];
   KeccakRhoOffsets[0..24]; A (and SELF)
[from] Function theta:
   context.state[0..199] FROM
   context.state[0..199]; A (and SELF)
[from] Function KeccakPermutationOnWords:
   context.state[0..199] FROM
   context.state[0..199];
   KeccakRoundConstants[0..23];
   KeccakRhoOffsets[0..24]; state (and SELF)
Context

Eva Plug-in

Derived Analyses
  Dependencies

Impact

Slicing

Conclusion
computes impact of a set of statements

i.e. the statements whose evaluation depend on the initial set

- data dependency (result of computation is impacted)
- address dependency (memory location is impacted)
- control dependency (branch may be taken or not)
```c
int x1, x2, res1, res2;

/*@ assigns \result \from x; */
int do_it_min(int x);

/*@ assigns \result \from x; */
int do_it_max(int x);

int f() {
    if (x1 <= 100) {
        res1 = do_it_min(x1);
    }
    if (x2 >= 1000) {
        //@ impact pragma stmt;
        res2 = do_it_max(x2);
    }
    res1++;
    res2++;
    return x1 > x2 ? x1 : x2;
}
```
frama-c -impact-pragmas f -impact-slicing \
impact.c -main f -lib-entry \
-then-last -print

```c
void f(void)
{
    if (x2 >= 1000)
        /*@ impact pragma stmt; */
        res2 = do_it_max(x2);
    res2 ++;
    return;
}
```
Context

Eva Plug-in

Derived Analyses

Dependencies

Impact

Slicing

Conclusion
slicing criterion

- value of a variable at a given point
- truth value of an ACSL assertion
- final state of the program

removes all statements that do not change the slicing criterion

same dependencies as impact, but used in the opposite direction
```c
int x1, x2, res1, res2;

/*@ assigns \result \from x; */
int do_it_min(int x);

/*@ assigns \result \from x; */
int do_it_max(int x);

int f() {
    if (x1 <= 100) {
        res1 = do_it_min(x1);
    }
    if (x2 >= 1000) {
        res2 = do_it_max(x2);
    }
    res1++;
    res2++;
    /*@ slice pragma expr res2; */
    return x1 > x2 ? x1 : x2;
}
```
frama-c -slice-pragma f \n  -main f -lib-entry slicing.c \n  -then-last -print

void f(void)
{
  if (x2 >= 1000)
    res2 = do_it_max(x2);
  res2 ++;
  //@ slice pragma expr res2; */ ;
  return;
}
Abstract Interpretation & Eva

- **abstract interpretation**: a systematic way to build correct and terminating analysis
  - abstract away information (over-approximations)
  - ensures answer in a reasonable time
  - while retaining adequate precision
  - and guaranteeing correct answers

- Eva
  - computes (over-approximations of) the domains of the variables of a program
  - emits alarms for potential undefined behaviors
  - proves their absence if no alarms
  - may also prove ACSL properties if expressable in its domains
Frama-C provides powerful analyzers for verifying properties over C programs.

<table>
<thead>
<tr>
<th>Absence of undefined behavior</th>
<th>Value Analysis</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dependencies between variables</td>
<td>From</td>
</tr>
<tr>
<td>Focus on a piece of code</td>
<td>Slicing / Impact</td>
</tr>
<tr>
<td>Functional properties</td>
<td>WP</td>
</tr>
<tr>
<td>Monitoring of unvalidated properties</td>
<td>E-ACSL</td>
</tr>
<tr>
<td>Custom analysis</td>
<td>Custom script</td>
</tr>
</tbody>
</table>

Combine them for even more powerful results :-)  
- Eva + WP: prove a few functions to remove false alarms  
- Eva + E-ACSL: check the remaining alarms at runtime  
Solutions to Quizzes
Question

Assume `\texttt{valid(p+(0 .. 2)) with p a pointer to int, and sizeof(int)==4. What is the most precise information we can say about `\texttt{block_length(p)}?` 

Answers

\begin{itemize}
  \item [a] `\texttt{block_length(p) == 2} \times$
  \item [b] `\texttt{block_length(p) >= 3}$
  \item [c] `\texttt{block_length(p) == 8}$
  \item [d] `\texttt{block_length(p) >= 8}$
  \item [e] `\texttt{block_length(p) == 12}$
  \item [f] `\texttt{block_length(p) >= 12}$
\end{itemize}
Question
Assume `valid(p+(0 .. 2))` with `p` a pointer to `int`, and `sizeof(int)==4`.
What is the most precise information we can say about `block_length(p)`?

Answers
- a `block_length(p) == 2`
- b `block_length(p) >= 3`  
- c `block_length(p) == 8`
- d `block_length(p) >= 8`
- e `block_length(p) == 12`
- f `block_length(p) >= 12`
Question

Assume $valid(p+(0 .. 2))$ with $p$ a pointer to $int$, and $sizeof(int)==4$. What is the most precise information we can say about $\block_length(p)$?

Answers

- a $\block_length(p) == 2$
- b $\block_length(p) >= 3$
- c $\block_length(p) == 8$ ×
- d $\block_length(p) >= 8$
- e $\block_length(p) == 12$
- f $\block_length(p) >= 12$
Question

Assume `valid(p+(0 .. 2))` with `p` a pointer to `int`, and `sizeof(int)==4`.
What is the most precise information we can say about
`block_length(p)`?

Answers

- **a** `block_length(p) == 2`
- **b** `block_length(p) >= 3`
- **c** `block_length(p) == 8`
- **d** `block_length(p) >= 8`
- **e** `block_length(p) == 12`
- **f** `block_length(p) >= 12`
Question

Assume $\text{valid}(p+(0 .. 2))$ with $p$ a pointer to int, and $\text{sizeof}(\text{int})==4$. What is the most precise information we can say about $\text{block_length}(p)$?

Answers

- $a$ $\text{block_length}(p) == 2$
- $b$ $\text{block_length}(p) >= 3$
- $c$ $\text{block_length}(p) == 8$
- $d$ $\text{block_length}(p) >= 8$
- $e$ $\text{block_length}(p) == 12$ ✗
- $f$ $\text{block_length}(p) >= 12$
Question
Assume $\text{valid}(p+(0 .. 2))$ with $p$ a pointer to int, and sizeof(int)==4.
What is the most precise information we can say about $\text{block_length}(p)$?

Answers
- a $\text{block_length}(p) == 2$
- b $\text{block_length}(p) >= 3$
- c $\text{block_length}(p) == 8$
- d $\text{block_length}(p) >= 8$
- e $\text{block_length}(p) == 12$
- f $\text{block_length}(p) >= 12$ ✓
Question

Assume an ACSL function `acsl_strlen` that returns the offset of the first '\0' char if it exists and -1 otherwise. What would be an appropriate requires for the standard library function `size_t strlen(const char* s)`?

Answers

- a. `acsl_strlen(s) >= 0` ✗
- b. `acsl_strlen(s) >= 0 && valid(s + (0 .. acsl_strlen(s)))`
- c. `valid(s + (0 .. acsl_strlen(s)))`
- d. `acsl_strlen(s) >= 0 && valid(s)`
**Question**

Assume an ACSL function `acsl_strlen` that returns the offset of the first '\0' char if it exists and -1 otherwise. What would be an appropriate requires for the standard library function `size_t strlen(const char* s)`?

**Answers**

- **a** `acsl_strlen(s) >= 0`
- **b** `acsl_strlen(s) >= 0 && \valid(s+ (0 .. acsl_strlen(s)))`
- **c** `\valid(s + (0 .. acsl_strlen(s)))`
- **d** `acsl_strlen(s) >= 0 && \valid(s)`
Question

Assume an ACSL function `acsl_strlen` that returns the offset of the first '\0' char if it exists and -1 otherwise. What would be an appropriate `requires` for the standard library function `size_t strlen(const char* s)`?

Answers

- a. `acsl_strlen(s) >= 0`
- b. `acsl_strlen(s) >= 0 && \valid(s + (0 .. acsl_strlen(s)))`
- c. `\valid(s + (0 .. acsl_strlen(s)))` ✗
- d. `acsl_strlen(s) >= 0 && \valid(s)`
Question

Assume an ACSL function `acsl_strlen` that returns the offset of the first `\0` char if it exists and `-1` otherwise. What would be an appropriate requires for the standard library function `size_t strlen(const char* s)`?

Answers

- a) `acsl_strlen(s) >= 0`
- b) `acsl_strlen(s) >= 0 && \valid(s + (0 .. acsl_strlen(s)))`
- c) `\valid(s + (0 .. acsl_strlen(s)))`
- d) `acsl_strlen(s) >= 0 && \valid(s)` ✗
Question

if $x$ is in the interval $[-10 .. 10]$ before the execution of statement

```c
if ($x == 0$) { $y = 14; }$
else { $y = (x < 0 ? 13 : x + 2); }$
```

What is the value associated to $y$ after the statement?

Answers

- [a] $[-8 .. 14]$  **X**
- [b] $[2 .. 13]$
- [c] $[2 .. 14]$
- [d] $[3 .. 14]$
Question

if $x$ is in the interval $[-10 .. 10]$ before the execution of statement

```c
if (x == 0) { y = 14; }
else { y = (x < 0 ? 13 : x + 2); }
```

What is the value associated to $y$ after the statement?

Answers

- $a$ [−8 .. 14]
- $b$ [2 .. 13] ✗
- $c$ [2 .. 14]
- $d$ [3 .. 14]

See solution
Question

if \( x \) is in the interval \([-10 .. 10]\) before the execution of statement

\[
\textbf{if } (x == 0) \{ \ y = 14; \ \} \\
\textbf{else} \ { y = (x < 0 ? 13 : x + 2); } \\
\]

What is the value associated to \( y \) after the statement?

Answers

- [ a ] [ -8 .. 14 ]
- [ b ] [ 2 .. 13 ]
- [ c ] [ 2 .. 14 ]
- [ d ] [ 3 .. 14 ]
Question

if x is in the interval \([-10 \ldots 10]\) before the execution of statement

```c
if (x == 0) { y = 14; }
else { y = (x < 0 ? 13 : x + 2); }
```

What is the value associated to y after the statement?

Answers

- **a** \([-8 \ldots 14]\)
- **b** \([2 \ldots 13]\)
- **c** \([2 \ldots 14]\)
- **d** \([3 \ldots 14]\) ✗
Question

if a is an array of size 3, initialized to 0, and c in [0..2] what would be the content of a after executing the following statement:

```c
if (c) { a[c] = c; } else a[1] = 3;
```

Answers

- a
  - a[0] IN {0}, a[1] IN {0,1,3}, a[2] IN {0,2}
  - Incorrect

- b
  - a[i] IN {0,1,2,3} for all indices

- c
  - a[0] IN {0}, a[1] IN {0,1,2,3} a[2] IN {0,1,2}

- d
  - a[0] IN {0}, a[1] IN {1,3}, a[2] IN {2}
Question

if a is an array of size 3, initialized to 0, and c in [0..2] what would be the content of a after executing the following statement:

```c
if (c) { a[c] = c; } else a[1] = 3;
```

Answers

- **a**
  
a[0] IN {0}, a[1] IN {0,1,3}, a[2] IN {0,2}

- **b**
  
a[i] IN {0,1,2,3} for all indices ✗

- **c**
  
a[0] IN {0}, a[1] IN {0,1,2,3} a[2] IN {0,1,2}

- **d**
  
a[0] IN {0}, a[1] IN {1,3}, a[2] IN {2}
Question

if a is an array of size 3, initialized to 0, and c in [0..2] what would be the content of a after executing the following statement:

```
if (c) { a[c] = c; } else a[1] = 3;
```

Answers

- a
  - a[0] IN {0}, a[1] IN {0,1,3}, a[2] IN {0,2}
- b
  - a[i] IN {0,1,2,3} for all indices
- c
  - a[0] IN {0}, a[1] IN {0,1,2,3} a[2] IN {0,1,2}
- d
  - a[0] IN {0}, a[1] IN {1,3}, a[2] IN {2}
Question

if a is an array of size 3, initialized to 0, and c in [0..2] what would be the content of a after executing the following statement:

```c
if (c) { a[c] = c; } else a[1] = 3;
```

Answers

- **a**
  
  a[0] IN {0}, a[1] IN {0,1,3}, a[2] IN {0,2}

- **b**
  
  a[i] IN {0,1,2,3} for all indices

- **c**
  
  a[0] IN {0}, a[1] IN {0,1,2,3} a[2] IN {0,1,2}

- **d**
  
  a[0] IN {0}, a[1] IN {1,3}, a[2] IN {2}

X