Frama-C Value Analysis

Séminaire CAP'TRONIC

Virgile Prevosto
virgile.prevosto@cea.fr

June 18th, 2015

Outline

Introduction

Abstract domains
- Arithmetic
- Memory

Methodology
- Basic commands
- Parameters

Value Analysis Plugin

Credits

- Pascal Cuoq
- Boris Yakobowski
- Matthieu Lemerre
- A few other developers...

More information

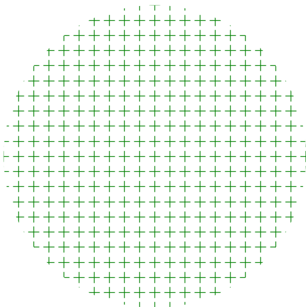
- `http://frama-c.com/download/frama-c-value-analysis.pdf`

Main Objective

Find the domains of the variables of a program

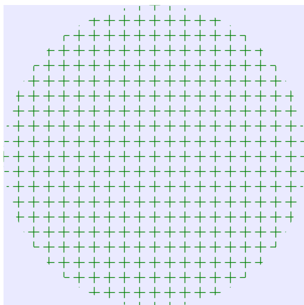
- based on abstract interpretation
- alarms on operations that may be invalid
- alarms on the specifications that may be invalid
- Correct: if no alarm is raised, no runtime error can occur

Abstract Interpretation in a Nutshell



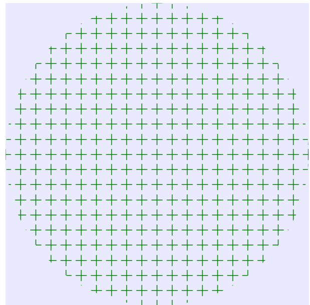
- Replace all possible concrete execution ...
- ... by one abstract execution
- Analysis is guaranteed to terminate
- Over-approximation and false alarms
- Trade-off between precision and computation time

Abstract Interpretation in a Nutshell



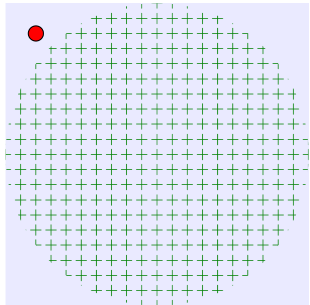
- Replace all possible concrete execution …
- … by one abstract execution
- Analysis is guaranteed to terminate
- Over-approximation and false alarms
- Trade-off between precision and computation time

Abstract Interpretation in a Nutshell



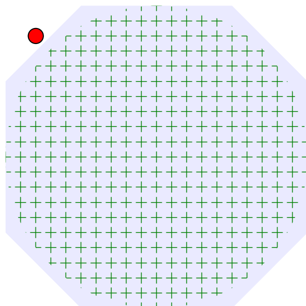
- Replace all possible concrete execution …
- … by one abstract execution
- Analysis is guaranteed to terminate
- Over-approximation and false alarms
- Trade-off between precision and computation time

Abstract Interpretation in a Nutshell



- Replace all possible concrete execution ...
- ... 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 execution ...
- ... by one abstract execution
- Analysis is guaranteed to terminate
- Over-approximation and false alarms
- Trade-off between precision and computation time

Some specificities

- Precise handling of pointers
- Several representation 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 analyzes through its API

Integer and Floating Point Arithmetic

Corresponding Abstract Domain

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

Examples

- $\{0; 40; \}$ = 0 or 40
- $[0..40]$ = an integer between 0 and 40 (inclusive)
- $[-..-]$ = any integer (within the bound 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)

next

Code sample

```
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; }
}
```

Value Result

```
frama-c -val -main f integer.c

[value] Called Frama_C_show_each_loop_1({0; 1})
[value] Called Frama_C_show_each_loop_1({0; 1; 2})
[value] Called Frama_C_show_each_loop_1([0..16])
[value] Called Frama_C_show_each_loop_1([0..40])
[value] ====== VALUES COMPUTED ======
  x IN {0; 40}
  y IN [0..40]
  z1 IN [--..--]
  t IN [-128..127]
  m IN [3..39],3%4
  d IN [0.25 .. 3.125]
```

Memory Address

Base Address

$$\begin{array}{l} \text{Global variable} \\ \uplus \text{ Formal parameter of main function} \\ \uplus \text{ literal string constant} \\ \uplus \text{ NULL} \\ \uplus \ldots \end{array}$$

Addresses

- Base address + Offset (integer)
- Each base has a maximal valid offset
- Abstract Values are sets of addresses

next

Examples of Addresses

Precise Base

- $\{\{\&p + \{4; 8\}\}\}$ = address of p shifted from 4 or 8 octets
- $\{\{\&"foobar"; \}\}$ = 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

Code Sample

```
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);
}
```

Value Result

```
[value] ====== VALUES COMPUTED ======
[value] Values at end of function f:
  x IN {{ &p{[1], [2]} }}
  y IN {{ "foobar" }}
  z IN {1024}
  t IN
   {{ garbled mix of &{p}
    (origin: Arithmetic
      {examples/value/address.c:16}) }}
```

Write to an Address

Abstract Domain

written address = valid left value

$$\begin{array}{rl} & \text{adress} \\ \times & \text{initialized?} \\ \times & \textit{not dangling pointer?} \end{array}$$

Example

```
{
    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

Code Sample

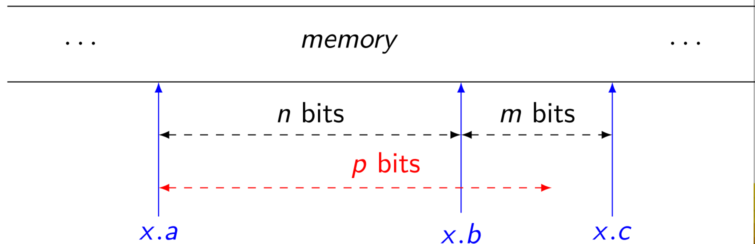
```
int X,Y, *p;
void f(int c) {
     int x,y;
     if (c<=0) x = 2;
 L:  if (c<=0) y = x + 1; else y = 4;
     X = x;
     Y = y;
     p = c ? &X : &x;
}

int main(int c) {
     f(c);
     if (Y==4) *p = 3;
     return 0;
}
```

Value Result

```
examples/value/address_written.c:8:
[kernel] warning:
  accessing uninitialized left-value:
  assert \initialized(&x);
examples/value/address_written.c:16:
[kernel] warning:
  accessing left-value that
  contains escaping addresses:
  assert !\dangling(&p);
[value] Values at end of function main:
  X IN {2; 3} or UNINITIALIZED
  Y IN {3; 4}
  p IN {{ &X }} or ESCAPINGADDR
  __retres IN {0}
```

Concrete Memory

- Seen as big array of bits
- read/write a value v at address i = read/write v at index i over n bits.
- n depends upon the type of v
- potential overlap
- example: $x.a$ extends over n bits, $x.b$ over m bits. Writing at $x.a$ over p bytes with $n < p < m$ will partially erase $x.b$



Abstract Memory State

base address $\rightharpoonup$ offsetmap

Example

$$
\begin{array}{l}
S \mapsto \{\ [0..31] \mapsto \{\&x \mapsto 0 \text{ (initialized, } \textit{not dangling})\} \\
\qquad\quad [32..47] \mapsto \{\mathit{NULL} \mapsto 12 \text{ (initialized, } \textit{not dangling})\}\ \} \\
x \mapsto \{\ [0..31] \mapsto \{\mathit{NULL} \mapsto \{3;24\} \text{ (initialized, } \textit{not dangling})\}\ \}
\end{array}
$$

Displaying Values

- Expected type is used to display values

Exemple

$$
\begin{array}{l}
S.\mathit{mypointer} \in \{\{\&x\}\} \\
\quad .\mathit{myshort} \in 12 \\
\qquad\qquad x \in \{\ 3;\ 24\ \}
\end{array}
$$

next

Abstract Memory

```
struct S {
      int* mypointer;
      short myshort;
} S;
char T[sizeof(struct S)];
int x;
void main (int c) {
      if(c) { x = 3; } else { x = 24; }
      S.mypointer = &x;
      S.myshort = 12;
      *(int*)T = &x;
      *(short *)((int*)T + 1) = 12;
}
```

Value Result

```
[value] ====== VALUES COMPUTED ======
[value] Values at end of function main:
  S.mypointer IN {{ &x }}
   .myshort IN {12}
   .[bits 48 to 63] IN {0}
  T[bits 0 to 31] IN {{ (? *)&x }}
   [bits 32 to 47] IN {12}
   [6..7] IN {0}
  x IN {3; 24}
```

Overlapping

```
int c,x;

char t[6];

void main(void) {
      t[0] = c ? 1 : 2;
      *(int*)(t+1) = c ? 3 : 4;
      *(t+3) = 5;
      x = *(int*)(t+1);
}
```

Value result

```
[value] Values at end of function main:
  x[bits 0 to 15]# IN {4}%32, bits 0 to 15
   [bits 16 to 23] IN {5}
   [bits 24 to 31]# IN {4}%32, bits 24 to 31
  t[0] IN {2}
   [bits 8 to 23]# IN {4}%32, bits 0 to 15
   [3] IN {5}
   [4]# IN {4}%32, bits 24 to 31
   [5] IN {0}
```

Main options

- `-main`: specifies the entry point of the analysis (default: `main` function)
- `-lib-entry`: Library mode: globals are not assumed to be 0-initialized
- `-val`: launch value, starting at the specified entry point

abs.c

```
// returns the absolute value of x
int abs ( int x ) {
  if ( x >=0 )
    return x ;
```

- `frama-c -main abs -val -lib-entry abs.c`

Automation

Is abstract interpretation an automated plug-in?

- yes...
- and no!
- must be driven carefully to give meaningful results
- requires some expertise and some time

Help Value Analysis to Understand the Code

- Pay attention to missing code (external library) or code that is not understood (asm)
  - write C code (stub), that can be understood by Value and approximates the missing part well enough with respect to the desired property
  - give an ACSL specification
- Give an appropriate context
  - Write an appropriate entry point to initialize global variables and formal parameters
  - Sometime possible to use dedicated options (`-context-*`)

next

Code Sample

```
void getchar(char* p);

int main(char* string) {
     char c;
     int idx = 0;
     getchar(&c);
     while(string[idx]) {
          if (string[idx] == c)
            return idx;
          idx++;
     }
     return -1;
}
```

Value result without specification

```
examples/value/stub.c:7:
[kernel] warning: out of bounds read.
  assert \valid_read(string+idx);
examples/value/stub.c:8: [kernel] warning:
  accessing uninitialized left-value:
  assert \initialized(&c);
examples/value/stub.c:8:
 [kernel] warning: out of bounds read.
  assert \valid_read(string+idx);
[value] Values at end of function main:
  c IN [--..--] or UNINITIALIZED
  idx IN {0; 1}
  __retres IN {-1; 0; 1}
```

Value result with specification

```
[value] Values at end of function main:
  c_0 IN [--..--]
  idx IN [0..255]
  __retres IN [-1..254]
[value] Values at end of function real_main:
  s[0..254] IN [--..--]
   [255] IN {0}
```

Enhance precision

Loops

- option `-ulevel`: syntactic loop unrolling
- option `-slevel`: allows Value to explore *n* separated paths before joining them
- option `-wlevel`: number of loop steps before performing widening (default is 3, use with caution)

Driving Value through Annotations

- ACSL assertions can be used to restrict propagated domains
- but only if Value can interpret it

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

- Case analysis using disjunctions

next

Code Sample

```
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--; }
}
```

Value Result

without slevel

```
x IN {9; 11; 32; 34}
y IN {35; 37; 41; 43}
```

with slevel, no assertion

```
x IN {9; 11; 34}
y IN {37; 41}
```

with slevel and assertion

```
/*@ assert c<=0 || c > 0; */

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

A more complete example

- Demo on PolarSSL 1.3.7 code
- Actually detected by Trust in Soft (`http://trust-in-soft.com/`), a spin-off of CEA
- 12 bugs reported and fixed since v1.1.8 according to PolarSSL Changelog