Code analysis with Frama-C Value Analysis

Stance Training Session – Course 1

Virgile Prevosto

March 28th, 2013

Outline

- Introduction
- Abstract domains
  - Arithmetic
  - Memory
- Methodology
  - Basic commands
  - Parameters
- Extensions

Value Analysis Plugin

Credits

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

More information

- http://frama-c.com/download/frama-c-value-analysis.pdf
- http://blog.frama-c.com/index.php?tag/value
- http://blog.frama-c.com/index.php?tag/skein

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
- can also be used in interpreter mode

Some specificities

- Precise handling of pointers
- Several representation for dynamic allocation (precision vs. time)
- GUI: can show possible values of any location at any program point.
- 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 et 3.125 (inclusive)

Memory Address

Base Address

$$
\begin{aligned}
&\text{Global variable} \\
\uplus\ &\text{Formal parameter of main function} \\
\uplus\ &\text{literal string constant} \\
\uplus\ &\text{NULL} \\
\uplus\ &\dots
\end{aligned}
$$

Addresses

- Base address $\rightharpoonup$ arithmetic value
- Fonctional abstract domain
- Equivalent to an associative *map*
- can be used as a rvalue for any type
- Pointer to integer cast will loose precision

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

Write to an Address

Abstract Domain

written address = valid left value

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

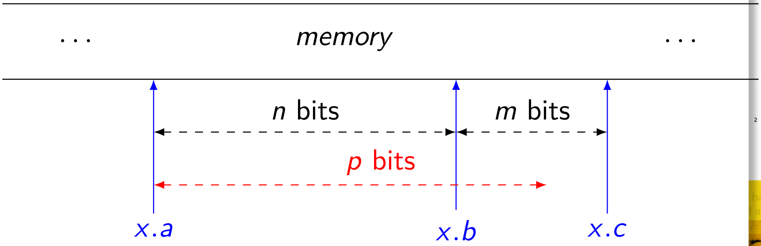
Exemple

```
{
   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

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 $x.b$



Contiguous Memory Zones

$$\text{offsetmap} = \text{interval} \rightharpoonup \text{value}$$

Exemple

First 32 bits contain address of x, next 16 contain 12.

$$[0..31] \mapsto \{\{\&x \text{ (initialized, } \textit{not dangling}\text{)}\}\}$$
$$[32..47] \mapsto 12$$

Remark

- Integers and pointers share the same representation
- Values in memory can be integers
- $12 \triangleq \textit{NULL} \mapsto 12$ (initialized, *not dangling*)

Abstract Memory State

base address $\rightharpoonup$ offsetmap

Example

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

Displaying Values

- Expected type is used to display values

Exemple

$$
\begin{aligned}
S.\mathit{mypointer} &\in \{\{\&x\}\} \\
.\mathit{myshort} &\in 12 \\
x &\in \{\ 3;\ 24\ \}
\end{aligned}
$$

Abstract Memory and Overlapping

```
int c,x;
char t[6];

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

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

```c
int R;
void abs(int x) {
      R = x >= 0 ? x : -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

Another example

simple.c (frama-c -val simple.c)

```c
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;
}
```

Get feedback from Value Analysis

- with the GUI
- with `Frama_C_show_each_test(...)`
- with `Frama_C_dump_each()`

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-*`)

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

Plugins based on Value

- Lightweight Analyzers
  - Call graphs
  - Constant propagation
  - Occurrence
- Side Effects and Dependencies
  - Functional dependencies
  - Imperative effects
  - Operational effects
  - Scope of assignments
- Code specialization
  - Slicing
  - Sparecode
  - Impact