Evolving Frama-C Value Analysis

Frama-C Day 2016 — Boris Yakobowski, CEA Tech List

Frama-C Value Analysis: a Brief Recap

The Value Analysis plugin

- Abstract interpreter for C
- Initially focused on embedded programs
- Analyses of more generic code in the last years

- Support for most of the gory details of C (+ some extensions)
  - bitfields, packing, type punning, FAM, etc

- Currently not handled:
  - restrictions on dynamic allocation (`alloca`)
  - recursive functions (specification must be supplied and is used instead)
  - parts of the C standard library lack ACSL specifications

Current implementation

- Abstractions:
  - Integers (reduced product of: intervals, congruence, discrete sets);
  - Precise representation of pointers;
  - Powerful memory domain (handles arrays, structs and unions);
  - Initialization/danglingness information.
- Disjunctive propagation of states (`-slevel` option)
  - after conditionals
  - for different loop iterations of a loop
- Datastructures highly optimized (hash-consing...) + cache
- Excellent results on embedded code

Main limitation: the memory abstraction is non-relational and hard-wired

Novelties in Frama-C
Aluminium and Silicium

New auxiliary plugins (1/2)

Tuning-up slevel

- `loop plugin`
- per-function suggestions for the amount of `slevel` to use

```
for (i=7; i<50; i+=2) {
  t[i] = v ? 0 : i;
}
```

```
-val-slevel-merge-after-loop main
-slevel-function main:84
```

Flagging invalid instructions

- `nonterm plugin`
- detects instructions that never return
  - either non-terminating functions, or guaranteed alarms

```
void f() {
  int t[90];
  for (int i=0; i<=90; i++)
    t[i] = i;
}
```

```
[nonterm] warning: unreachable
  return statement for function f
```

Both plugins are available in Aluminium

New auxiliary plugins (2/2)

Handling variadic functions: `variadic` plugin

- handles known (`sprintf`, `scanf`...) or unknown functions
- specialized, non-variadic, prototype for each call-site
- plus pre- and post-conditions

Initial code: int *p; scanf("%d", p);

Result: (behavior corresponding to a successful scan):

```
/*@ requires  valid_read_string (format);
    requires \valid(param0);
    ensures \ initialized (param0); [...]
    assigns \result \from *__fc_stdin , *(format+(0 ..));
    assigns *param0 \from *__fc_stdin , *(format+(0 ..));  */
int scanf_0(char const *format, int *param0);
```

- Available in Aluminium. Activated by default in Silicium?

Dynamic Allocation

Sound & precise modelization

- builtins for `malloc`, `realloc` and `free`
- one new variable by allocation site
  - *allocation site* = entire callstack
  - "normal" variable on first allocation
  - decays to summarized variable when needed

```
while (i <= 10) {
  int *p = malloc(sizeof(int));
  *p = i; // same variable on each iteration, strong updates
  free (p);
}
```

Dynamic Allocation

Sound & precise modelization

- builtins for `malloc`, `realloc` and `free`
- one new variable by allocation site
  - *allocation site* = entire callstack
  - "normal" variable on first allocation
  - decays to summarized variable when needed

```
while (i <= 10) {
  int *p = malloc(sizeof(int));
  *p = i; // variable may correspond to multiple allocations, weak updates
  if (rand ()) free (p);
}
```

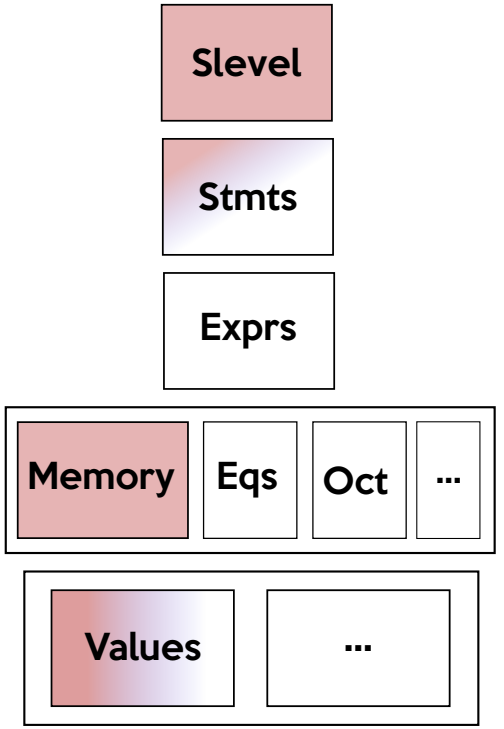
- Will be available in Silicium

EVA: Evolved Value Analysis

Major rewrite of large parts of Value

- By default: same abstractions (memory and values)
- Same iteration strategy
- Everything else has changed
  - new analysis domains
  - better backward propagation
  - ability to add new domains!

- Active by default in Aluminium



Better backward propagation

Goal: learning from conditionals and alarms

- e.g. if (x+3 <= c+y) or assert x*4 <= 13
- Before: selected syntactic patterns
- Now: systematic approach through atomic transformers
  - one transformer per operator
  - old patterns backported
  - new transformers (integer and floating-point +/−, |, &)

- Available in Aluminium; more transformers in Silicium

Symbolic equalities and locations

Symbolic equalities

- Store equalities between arbitrary expressions
  - Already used in Astrée and Verasco
  - Undo temporaries introduced by parsing

```
if (x >= 0) tmp = x; else tmp = -x;
if (tmp < 0.01) /* must use eq. on x and tmp */ { y = 1/x; }
```

Symbolic locations

- Store abstract value for e.g. `t[i]` in `if (t[i]-3 >= k)`.
- Complimentary to symbolic equations.

```
/* ''Grail'' when
     i is imprecise */
```

```
if (t[i] >= 2) {
  f(&t[i]);
}
```

```
void f(int *p) {
  int x = *p+1;
  /*@ assert x >= 3; */
}
```

- Symbolic equalities in Aluminium, locations in Silicium

Gauge domain [Venet 2012]

Affine relation with the loop counter(s)

- Phantom variable λ corresponding to the current iteration
- All variables are in relation *only* with λ
  - good scalability
- Good domain for the extremely frequent idiom

```
int *p = &t;
for (int i=0; i<N; i++) {
  *p++ = k;
}
```

- Will be available in Silicium

Bitwise values

- Represent constraints on some bits of a value
  - `if (x & 0xf0) { /* x? */ ... }`
  - numeric domains not sufficient in general
- Memory domain can represent sequences of values
  - e.g. $\{\&x\}_{bits\ 16-31}$; $[10..1024]3\%4_{bits\ 8-15}$; $0_{8\ bits}$
  - write abstract bitwise transformers on such values
- BDDs: more expressive – but more complex – possibility
- Available in Aluminium
- Next steps:
  - information in the least significant bits of a pointer
  - sign/exponent/mantissa of floating-point values

Relational numerical domains

Binding to the Apron library

- Relational numerical domains (polyhedra, octagon, etc...)
- Currently: integer variables
- **BTS-supplied example**: ⇐
  - The assertion gets proven!
  - unprovable without WP or a massive disjunction before

```
/*@ requires 0 < len <= 1024;
    requires 0 < n < 64; */
void main(size_t len, size_t n) {
  if (len >= 64 || len + n >= 64)
  {
    n = 64 - n;
    len -= n;
  }
  //@ assert len <= 1023;
}
```

- Proof-of-concept in Aluminium
- Numerous improvements in Silicium:
  - aggregates
  - better handling of expressions that overflow

Next Steps

Consolidating it all

Behind the scenes

- add sound support of option `-memexec-all` in new domains
  - required for scalability
- collaborative evaluation of logical assertions
  - including calling functions with only a specification
- even simpler APIs for new domains

User feedback

- saving the inferred abstract domains on disk
- displaying the new results in the GUI

Relational domains

- New domains for pointers/dynamic allocation:
  - **int** $*$p = &x+i; p+=k;
  - Relations with the size of an allocated base
    char $*$p = malloc(S); **while**(i<S) p[i]=...;
- Numerical domains
  - Handle aggregates: arrays and structs (Silicium)
  - Handle float/double (Phosphorus?)
  - Limit the number of variables in relation: "packing"
    - "Semantic" heuristics to choose the variables?
  - Binding to the Verified Polyhedra Library (Verimag)
- Mature all currently implemented domains

  Goals: scalability & expressiveness

Conclusion and Perspectives

From Magnesium to Silicium

- Major reimplementation of the legacy analyzer
  - Finally extensible!
  - Without new domains: comparable/better results but cleaner implementation!
- Stable Open API, hopefully in Frama-C Phosphorus (mid-2017)
  - Beta-testers welcome :-)
- Many other additions
- Many challenges for the next months!
  - mature the new domains
  - implement other complementary domains
- Further news on the blog: `http://blog.frama-c.com/`

Brought to you by

- EDF/AREVA/CEA collaborative project *QLCC*
- ANR *Vecolib* project
- TrustInSoft/CEA joint lab
- DGA Rapid *Aurochs* project

- The Frama-C/Value team, past and present

  P. Cuoq, B. Monate, B. Yakobowski, M. Lemerre, A. Maroneze, V. Perrelle, D. Bühler, and all our colleagues of the Frama-C team

Contact us!

if you are interested in a collaboration on new analysis domains