The Flatac Frama-c front-end

Radu Iosif, Florent Garnier

May 13, 2012

Outline

What is flatac ?

- Part of a toolchain that aims at proving that C programs don't generate memory faults and don't violates assertions.
- A front end that generates NTS based models of C programs.
- Coded as a Frama-C plugin.

Typical memory faults :

- Memory access outside and allocated memory zone of the heap
- Access to an array outside of its bounds
- Memory access using a non aligned address
- Double free
- Freeing an allocated segment using an pointer that does not points at the begining of the segment.
- Memory leaks

Two subkinds of properties :

- Properties concerning the memory shape (Simple Separation Logic):
  - Relation between pointer variables (Stack) and location variables (heaps).
  - Memory allocation.
  - Allocated Segment separation.
- Arithmetic properties :
  - Memory segment access within its bounds.
  - Memory address alignment (Congruence).

Tracked property

This front end aims at proving that C programs :

- Have no execution run that lead to memory fault.
- Have no exectution that violates some assertion expressed using arithmetic constraints.

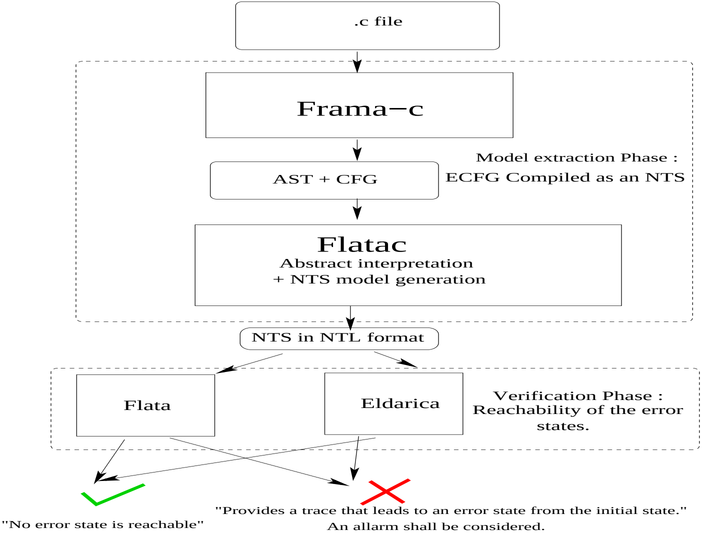
Flatac plugin: Front end of NTS error state reachability analysis

- Extracts models of C Programs using Abstract Interpretation Techniques.
- Adds Numerical Transitions Systems informations on the model for a posteri Verification Phase.

How to do that ?

1. Extracting an extended cfg from Frama-c cfg (Cil statements $\times$SSL memory abstractions)^2
2. Labelling the Ecfg transitions with Numerical Transition System expression –Guards, counter affectation and Function Calls.
3. If a SSL Abs value of a state is $\perp$, define this state as an error state.
4. Export the labelled Ecfg into Nts Format.
5. Ask an analysis tool –Flata, Eldarica, to check whether some error state is reachable from the entry point (main function).

Flatac in the tool-chain:



Abstract interpretation preliminary part

Simple Separation Logic formulae : Abstract domain.

$$\begin{array}{lcll}
\phi & := & \pi \updownarrow \sigma \mid \exists l.\phi & \text{Formulae} \\
\pi & := & x \mapsto l \mid x \mapsto \text{nil} \mid (l_1 = l_2) \mid \pi_1 \wedge \pi_2 & \text{Pure part} \\
\sigma & := & \text{Emp} \mid alloc(l) \mid \sigma_1 * \sigma_2 \perp & \text{Spatial part}
\end{array}$$

Properties of SSL

The problem that follows are decidable :

- Satisfiability (Valid configuration)
- Entailment, Equivalence.
- Memory leaks

Those problems are solved using rewriting techniques.

Example of SSL formulae

- $x \mapsto l_1 \wedge y \mapsto l \wedge z \mapsto \text{nil} \updownarrow \text{Emp}$
- $x \mapsto l_1 \wedge y \mapsto l \wedge z \mapsto \text{nil} \updownarrow alloc(l_1)$
- $x \mapsto l_1 \wedge y \mapsto l \wedge z \mapsto \text{nil} \updownarrow alloc(l_1) * alloc(l)$
- $x \mapsto l_1 \wedge y \mapsto l \wedge z \mapsto \text{nil} \updownarrow alloc(l_1) * alloc(l)$
- $x = y \wedge x \mapsto l_1 \wedge y \mapsto l \updownarrow alloc(l_1) * alloc(l)$ (Unsat)
- $x = y \wedge x \mapsto l_1 \wedge y \mapsto \text{nil} \updownarrow alloc(l_1)$ (Unsat)
- $true \updownarrow alloc(l)$ (Leak)

Flata-c Memory model

A memory model that associates counters to SSL variables :

SSL Variable	NTS counter	
$x \in PVar$	`x_offset`	offset
$l \in LVar$	`l_size`	segment size

In order to :

- Associate to segment their size.
- Associate to pointer their offset.
- To express guards on memory access.

Example

C "statement"	SSL formula	NTS transition
`int *x;`	$\exists lx \mapsto l \updownarrow \mathrm{Emp}$	`offset_x'=0`
`x=malloc(10);`	$\exists lx \mapsto l \updownarrow alloc(l)$	`size_l=10`
`x++;`	$\exists lx \mapsto l \updownarrow alloc(l)$	`offset_x'+=sizeof(int)`
`int y =*x;`	$\exists lx \mapsto l \updownarrow alloc(l)$	`offset_x'` $<$ size_l
	$\exists . lx \mapsto l \updownarrow alloc(l)$	havoc(y)

Example

C "statement"	SSL formula	`NTS transition`
`int *x;`	$\exists lx \mapsto l \updownarrow \text{Emp}$	`offset_x'=0`
`x=malloc(10);`	$\exists lx \mapsto l \updownarrow alloc(l)$	`size_l=10`
`x++;`	$\exists lx \mapsto l \updownarrow alloc(l)$	`offset_x'+=sizeof(int)`
`int y =*x;`	$\exists lx \mapsto l \updownarrow alloc(l)$	`offset_x'` $<$ size_l
	$\exists .lx \mapsto l \updownarrow alloc(l)$	havoc(y)
`x+=10;`	$\exists lx \mapsto l \updownarrow alloc(l)$	`offset_x'+=10*sizeof(int)`

Example

C "statement"	SSL formula	NTS transition
`int *x;`	$\exists lx \mapsto l \updownarrow \mathrm{Emp}$	`offset_x'=0`
`x=malloc(10);`	$\exists lx \mapsto l \updownarrow alloc(l)$	`size_l=10`
`x++;`	$\exists lx \mapsto l \updownarrow alloc(l)$	`offset_x'+=sizeof(int)`
`int y =*x;`	$\exists lx \mapsto l \updownarrow alloc(l)$	`offset_x'` $<$ size_l
	$\exists . lx \mapsto l \updownarrow alloc(l)$	havoc(y)
`x+=10;`	$\exists lx \mapsto l \updownarrow alloc(l)$	`offset_x'+=10*sizeof(int)`
`*x=42;`	$\perp$	`offset_x` $\geq$ size_l

Access to `*x` is out of bounds of allocated segment at `l`.

Cil representation of C programs

Cil provides and AST and CFG info from C files.

Most relevant information

For each function of the AST :

- Expressions.
- Locals and formal variables.
- Statements of the Control flow graph.

Control flow statemnt v.s. basic instructions

Control flow statement

- `if(expr,blockif,blockelse)`
- `switch(expr,case_list)`
- `while(expr)`

Instruction statement

- `lval=expr`
- `lval=funcall(name,exp list)`

Nts guards for valid memory access

Let `x` a *PVar* such that `x:`τ`*` Let
$\texttt{valid\_mem} : \texttt{Cil\_types.expr} \times SSL \mapsto NtsGuards$

Memory access guards Cil for atomic expressions

Cil expressions	Memory abstraction	Nts Guard
`*x`	$x \mapsto l \updownarrow alloc(l)$	`true`
`*x`	$x \mapsto l \updownarrow \mathrm{Emp}$	`false`
`*(x+i)`	$x \mapsto l \updownarrow alloc(l)$	$0 \leq i \times \texttt{sizeof}(\tau) < \texttt{l\_size}$
`tab[i]`	$true \updownarrow \mathrm{Emp}$	$0 \leq i < \texttt{tab\_size}$

Nts guards for valid memory access

Cil expression type definition (Non exhaustive)

```
UnOp(UOp,expr)
BinOp(BOp,expg,expd)
UOP=UnMin| BNot | Neg...
BOP=BAnd|BOr...
    Plus|Minus|Prod|Div...
    PlusPI|MinusPI|MinusPP...
```

valid_mem of Cil expression

Cil expr	valid_mem
UnOp(UOp,expr)	valid_mem(expr)
BinOp(BOp,expg,expd)	valid_mem(expg)∧valid_mem(expg)

Model extraction :

Input : Cil AST and Control flow graph
Generated Model : Extended CFG,
$(S_i, S_f, S_{err}, S, \rightarrow \in (S \times R \times S))$
where :

- $S \in$ (Cil_types.stmt$\times$ Abs),
- $Abs =$ Set of SSL formula $\bigcup \perp$,
- R is a set of possibly guarded NTS transitions.

Compiling expressions

$$\boxed{\begin{array}{lcl} base_\phi(x) & := & l \text{ if } x \mapsto l \in PP(\phi) \\ base_\phi(\texttt{NULL}) & := & \text{nil} \quad \forall \phi \\ base_\phi(P + I) & := & base_\phi(P) \end{array}}$$

Compiling Integer expressions

$$
\begin{array}{lcl}
[\![n]\!]_\phi & := & n \\
[\![i]\!]_\phi & := & i_{\mathrm{cnt}} \\
[\![\mathtt{NULL}]\!]_\phi & := & 0 \\
[\![x]\!]_\phi & := & x_{off} \\
[\![x_1 - x_2]\!]_\phi & := & x_{1\,off} - x_{2\,off} \\
[\![P + I]\!]_\phi & := & [\![P]\!]_\phi + [\![I]\!]_\phi \times \text{ sizeof}\,(\tau),\ \text{ where } P : \tau * \\
[\![P_1 - P_2]\!]_\phi & := & ([\![P_1]\!]_\phi - [\![P_2]\!]_\phi) /\text{ sizeof}\,(\tau),\ \text{ where } P_1 : \tau * \\
[\![I_1 + I_2]\!]_\phi & := & [\![I_1]\!]_\phi + [\![I_2]\!]_\phi \\
[\![I_1 \times I_2]\!]_\phi & := & [\![I_1]\!]_\phi \times [\![I_2]\!]_\phi
\end{array}
$$

Compiling Boolean expressions

$$
\begin{array}{lcl}
[\![P_1 == P_2]\!]_\phi & := & \begin{cases} [\![P_1]\!]_\phi == [\![P_2]\!]_\phi & \text{if } \mathit{base}_\phi(P_1) \equiv \mathit{base}_\phi(P_2) \\ \bot & \text{else} \end{cases} \\
[\![P_1 != P_2]\!]_\phi & := & \begin{cases} [\![P_1]\!]_\phi != [\![P_2]\!]_\phi & \text{if } \mathit{base}_\phi(P_1) \equiv \mathit{base}_\phi(P_2) \\ \bot & \text{else} \end{cases} \\
[\![P_1 \bowtie P_2]\!]_\phi & := & \begin{cases} [\![P_1]\!]_\phi \bowtie [\![P_2]\!]_\phi & \text{if } \mathit{base}_\phi(P_1) \equiv \mathit{base}_\phi(P_2) \\ \bot & \text{else} \end{cases}
\end{array}
$$

Extraction : Basic statment

Var(v)=expr

$$\phi \xrightarrow{\texttt{valid\_mem(expr) and v'=}[\![expr]\!]_{\phi}} \phi$$

$$\phi \xrightarrow{\neg\texttt{valid\_mem(expr)}} \mathit{true} \updownarrow \perp$$

Extraction : Basic statment

`lval=expr`

Considered lvals are array element and referenced mem cells.

`valid_mem(expr)`$\wedge$`valid_mem(lval)`

true $\updownarrow$ Emp

true $\updownarrow$ Emp

true $\updownarrow$ $\perp$

$\neg($`valid_mem(expr)`$\wedge$`valid_mem(lval)`$)$

`lval=expr`

Extraction : Basic statement

`lval=call(fun,arg1,...,argn)`

Call of `fun` where `{P} fun {Q}`.

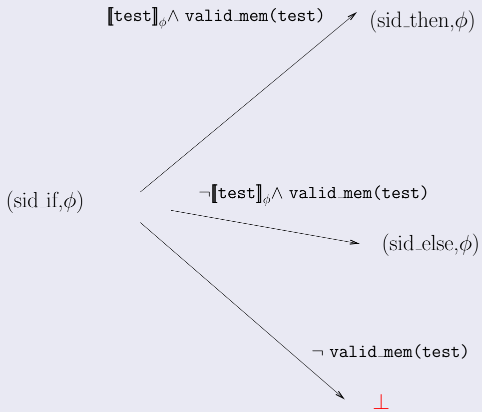
$$(\mathrm{sid}, P * R) \xrightarrow{\bigwedge_{0 \le i \le n} \texttt{valid\_mem}(arg_i) \text{ and } \mathrm{fun}(\llbracket arg_1 \rrbracket_{P*R}, \ldots, \llbracket arg_n \rrbracket_{P*R})} (\mathrm{next\_sid}, Q * R)$$

$$(\mathrm{sid}, P * R) \xrightarrow[\exists i, \neg\, \texttt{valid\_mem}(arg_i)]{} \perp$$

Exctraction : Control Flow operation

`if(test, stmtif, stmtelse)`

Expression `test` might perform some illegal operations.



Among other things

- Validity of integer values : Initialization, difference between two pointers, (Valid, Not Valid, Don't Know)
- Transitions not generated when guards can be statically proved false.

Verification Phase : Reachability Analysis

- Exporting the Ecfg Hierarchical Numerical Transition System.
- Reachability analisys of the error states by FLATA and/or ELDARICA
- If some error state is reachable : An alarm is raised.
- If no error states is reachable : The program is free of the memory fault we consider.