frama C
Software Analyzers

Developing Frama-C Plug-ins in OCaml

Julien Signoles
Software Safety Lab.

cea list

Ocaml Users Meeting

2011/04/15

Frama-C at a glance

http://frama-c.com

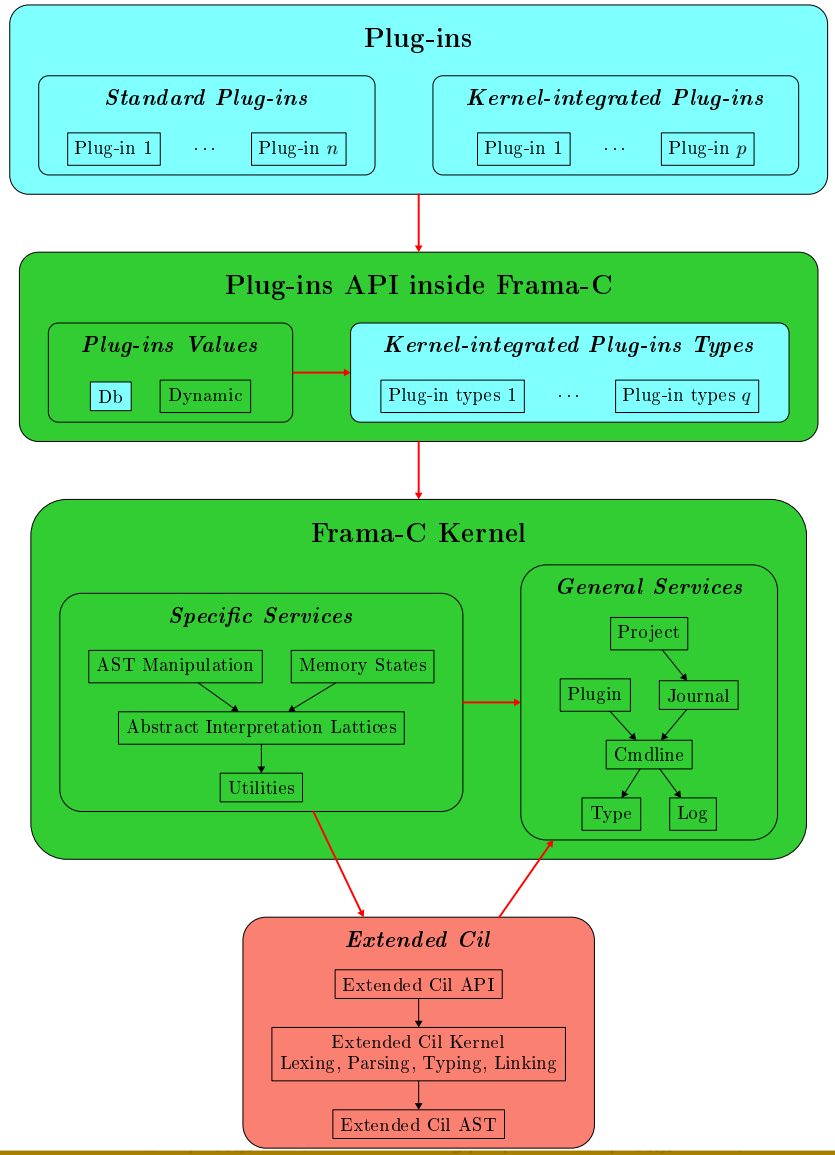
- platform dedicated to source code analysis of C programs
- ANSI/ISO C 99 + a formal specification language ACSL
- developed by CEA LIST and Inria Proval since 2005
- open source and released under LGPL v2.1
- extensible platform through plug-ins
- 1 plug-in = 1 analyser
- collaboration between analysers
- several static analysers
- both academic and industrial purposes

Several tools inside one platform

Frama-C development overview

- fully written in OCaml (> 100 kloc)
- minimal number of dependencies (long-term support+easier installation) :
  - Fork of CIL (Necula and *al*, Berkeley)
  - OCamlGraph
  - LablGtk2
- big API for developing analysers
- support : API documentation, Plug-in Development Guide, mailing-list, BTS, wiki, ...

Frama-C architecture



A simple script

running example :

Loop counter

loop_counter.ml:

```
let main () =
  Format.printf "Counting number of loops...@."

(* execute your plug-in among the others *)
let () = Db.Main.extend main
```

$ frama-c -load-script loop_counter

Registering your plug-in

```
(* register your plug-in to access basic services *)
include Plugin.Register
  (struct
    let name = "loop counter"
    let shortname = "loop"
    let help =
      "Count the number of loops in the program"
   end)
```

- default options (`-loop-help`, `-loop-debug`, `-loop-verbose`)
- way to add command line options
- way to display messages

Messages

Format-like functions for user messages :

- `feedback`, `result`, `warning`, `abort`, `error`, `fatal`, ...
- consistent message taxonomy anywhere in Frama-C

```
let nb_loops = ref 0
let main () =
  feedback ~level:2 "Counting number of loops...";
  result "Program contains %d loops" !nb_loops
```

Command Line Options

Functors for registering parameters

```
(* boolean option '-loop' initialized to [false] *)
module Enable =
  False(struct
    let option_name  = "-loop"
    let help = "enable count of loops in the program"
    let kind = 'Correctness
  end)

let main () =
  (* compute iff the option is set *)
  if Enable.get () then begin
    feedback ~level:2 "Counting number of loops...";
    result "Program contains %d loops" !nb_loops
  end
```

Makefile and configure

Plug-in Makefile and configure

- may include a generic Makefile
- only set few variables
- may also write easily a configure from a generic one

```
# Frama-C installation directories
FRAMAC_SHARE  :=$(shell frama-c.byte -print-path)
FRAMAC_LIBDIR :=$(shell frama-c.byte -print-libpath)
PLUGIN_NAME   := Loop_counter
PLUGIN_CMO    := loop_counter
include $(FRAMAC_SHARE)/Makefile.dynamic

$ make && sudo make install
$ frama-c -loop
```

frama C Software Analyzers

Visitor

Counting loops while visiting

```
let nb_loops () =
  nb_loops := 0;
  let count_loop = object
    (* visit the AST in place by inheritance *)
    inherit Visitor.frama_c_inplace
    (* only implement what is required *)
    method vstmt s = match s.Cil_types.skind with
    | Cil_types.Loop _ -> incr nb_loops; Cil.DoChildren
    | _ -> Cil.DoChildren
  end
  in
  (* visit the AST with our custom visitor *)
  Visitor.visitFramacFile count_loop (Ast.get());
  !nb_loops
```

Provide access to your analyser to other plug-ins

- safe dynamic typing facilities provided by the Frama-C kernel
- the only way to define mutually-recursive plug-ins
- one interface, several implementations

```
let nb_loops =
  (* register function 'Loop_counter.nb_loops'*)
  (* of type 'unit -> int' in the API *)
  Dynamic.register
    ~journalize:false
    ~plugin:"loop_counter"
    "nb_loops"
    (Datatype.func Datatype.unit Datatype.int)
    nb_loops
```

Journalisation

- journal = OCaml script replaying user actions
- automatically generate on need/on user request
- help debugging
- kind of macro language
- quick plug-in prototyping

```
let nb_loops =
  Dynamic.register
    ~journalize:true
    ~plugin:"loop_counter"
    "main"
    (Datatype.func Datatype.unit Datatype.int)
    nb_loops
```

Project

- project = one AST + associated global states
- as many projects as you want
- default project : no need to add extra parameters to analysers
- only require to register global states

```
(* a reference to an int option by project *)
module Nb_loops =
  State_builder.Option_ref
    (Datatype.Int)
    (struct
      let name = "Loop_counter.Nb_loops"
      let dependencies = [ Ast.self ]
      let kind = `Correctness
     end)
```

GUI

possible to extend the Frama-C GUI in many ways

```
(* dedicated panel for our plug-in *)
let loop_counter_panel =
  let box = GPack.hbox () in
  ... (* adding widgets to the box *)
  let refresh () = ... in
  "Loop counter", box#coerce, Some refresh

let main main_ui =
 (* attach our new panel to the GUI *)
  main_ui#register_panel loop_counter_panel

(* run our main when creating the GUI *)
let () = Design.register_extension main
```

Conclusion

http://frama-c.com

- platform dedicated to source code analysis of C programs
- high-level powerful services
- handle all C constructs
- existing community of plug-in developers
- both academic and industrial purposes
- documentations and supports

Join the growing community