Contiki-NG + E-ACSL in Cooja (wip)

VESSEDIA

Inria Lille

March 2019

1) Frama-C and E-ACSL

frama C
Software Analyzers

- Framework for Modular Analysis of C programs
- interoperable program analyzers for C program : static analysis, dynamic analysis
- CEA and INRIA
- OCaml language, LGPL + BSD licences
- modular plugin architecture :
  - kernel : CIL generates the C AST + ACSL (Ansi-C Specification Language) annotations
  - plugins :
    - WP : deductive verification → functional correctness
    - EVA : abstract interpretation → no runtime errors
    - E-ACSL : a kind of super-valgrind (memcheck, memory leak) but at source-level → no runtime errors
    - ...

github/travis era + critical softwares => Frama-C inside the continuous integration cycle infrastructure ?

E-ACSL :

- transforms the C program such that :
  - fails **at runtime** if an (E-)ACSL annotation is violated
  - doesn't change the behaviour otherwise

$\Rightarrow$ dynamical analysis relying on runtime libraries

RTE :

- generates automatically the (E-)ACSL annotations

```c
int main()
{
    int i = 0 ;
    int *ptr = &i;
    *ptr = 0 ;
    return 0;
}
```

```c
int main()
{
    int i = 0 ;
    int *ptr = &i;
    *ptr = 0 ;
    return 0;
}
```

```
frama-c -machdep gcc_x86_64 test.c -print -ocode test.frama.c
```

```c
int main()
{
    int i = 0 ;
    int *ptr = &i;
    *ptr = 0 ;
    return 0;
}
```

```
frama-c -machdep gcc_x86_64 test.c -print -ocode test.frama.c
```

```c
int main(void)
{
    int __retres;
    int i = 0;
    int *ptr = & i;
    *ptr = 0;
    __retres = 0;
    return __retres;
}
```

```c
int main()
{
    int i = 0 ;
    int *ptr = &i;
    *ptr = 0 ;
    return 0;
}
```

```c
int main()
{
    int i = 0 ;
    int *ptr = &i;
    *ptr = 0 ;
    return 0;
}
```

```
frama-c -machdep gcc_x86_64 -e-acsl-prepare -rte test.c -print -ocode test.rte.c
```

```c
int main()
{
    int i = 0 ;
    int *ptr = &i;
    *ptr = 0 ;
    return 0;
}
```

```
frama-c -machdep gcc_x86_64 -e-acsl-prepare -rte test.c -print -ocode test.rte.c
```

```c
int main(void)
{
    int __retres;
    int i = 0;
    int *ptr = & i;
    /*@ assert rte: mem_access: \valid(ptr); */
    *ptr = 0;
    __retres = 0;
    return __retres;
}
```

```c
int main()
{
    int i = 0 ;
    int *ptr = &i;
    *ptr = 0 ;
    return 0;
}
```

```
frama-c -machdep gcc_x86_64 -e-acsl-prepare -rte test.c -print -ocode test.rte.c
```

```c
int main(void)
{
    int __retres;
    int i = 0;
    int *ptr = & i;
    /*@ assert rte: mem_access: \valid(ptr); */
    *ptr = 0;
    __retres = 0;
    return __retres;
}
```

```
frama-c -machdep gcc_x86_64 test.rte.c -e-acsl -then-last -print -ocode test.e-acsl.c
```

```c
int main()
{
    int i = 0 ;
    int *ptr = &i;
    *ptr = 0 ;
    return 0;
}
```

```
frama-c -machdep gcc_x86_64 -e-acsl-prepare -rte test.c -print -ocode test.rte.c
```

```c
int main(void)
{
    int __retres;
    int i = 0;
    int *ptr = & i;
    /*@ assert rte: mem_access: \valid(ptr); */
    *ptr = 0;
    __retres = 0;
    return __retres;
}
```

```
frama-c -machdep gcc_x86_64 test.rte.c -e-acsl -then-last -print -ocode test.e-acsl.c
```

or directly :

```
frama-c -machdep gcc_x86_64 test.c -e-acsl-prepare -rte -then -e-acsl -then-last -print -ocode test.e-acsl.c
```

```
int main(void)
{
    int __retres;
    __e_acsl_memory_init((int *)0,(char ***)0,(size_t)8);
    int i = 0;
    __e_acsl_store_block((void *)(& i),(size_t)4);
    __e_acsl_full_init((void *)(& i));
    int *ptr = & i;
    __e_acsl_store_block((void *)(& ptr),(size_t)8);
    __e_acsl_full_init((void *)(& ptr));
    /*@ assert rte: mem_access: \valid(ptr); */
    {
        int __gen_e_acsl_initialized;
        int __gen_e_acsl_and;
        __gen_e_acsl_initialized = __e_acsl_initialized((void *)(& ptr),
        sizeof(int *));
        if (__gen_e_acsl_initialized) {
            int __gen_e_acsl_valid;
            __gen_e_acsl_valid = __e_acsl_valid((void *)ptr,sizeof(int),
            (void *)ptr,(void *)(& ptr));
            __gen_e_acsl_and = __gen_e_acsl_valid;
        }
        else __gen_e_acsl_and = 0;
        __e_acsl_assert(__gen_e_acsl_and,(char *)"Assertion",(char *)"main",
        (char *)"rte: mem_access: \\valid(ptr)",26);
    }
    __e_acsl_initialize((void *)ptr,sizeof(int));
    *ptr = 0;
    __retres = 0;
    __e_acsl_delete_block((void *)(& ptr));
    __e_acsl_delete_block((void *)(& i));
    __e_acsl_memory_clean();
    return __retres;
}
```

```
int main(void)
{
    int __retres;
    __e_acsl_memory_init((int *)0,(char ***)0,(size_t)8);
    int i = 0;
    __e_acsl_store_block((void *)(& i),(size_t)4);
    __e_acsl_full_init((void *)(& i));
    int *ptr = & i;
    __e_acsl_store_block((void *)(& ptr),(size_t)8);
    __e_acsl_full_init((void *)(& ptr));
    /*@ assert rte: mem_access: \valid(ptr); */
    {
        int __gen_e_acsl_initialized;
        int __gen_e_acsl_and;
        __gen_e_acsl_initialized = __e_acsl_initialized((void *)(& ptr),
        sizeof(int *));
        if (__gen_e_acsl_initialized) {
            int __gen_e_acsl_valid;
            __gen_e_acsl_valid = __e_acsl_valid((void *)ptr,sizeof(int),
            (void *)ptr,(void *)(& ptr));
            __gen_e_acsl_and = __gen_e_acsl_valid;
        }
        else __gen_e_acsl_and = 0;
        __e_acsl_assert(__gen_e_acsl_and,(char *)"Assertion",(char *)"main",
        (char *)"rte: mem_access: \\valid(ptr)",26);
    }
    __e_acsl_initialize((void *)ptr,sizeof(int));
    *ptr = 0;
    __retres = 0;
    __e_acsl_delete_block((void *)(& ptr));
    __e_acsl_delete_block((void *)(& i));
    __e_acsl_memory_clean();
    return __retres;
}
```

then compilation and link with :

- e-acsl runtime library checking the memory access
- customized version of malloc and gmp (multiprecision)

```c
#include <stdio.h>
int main()
{
    int i = 0 ;
    int *ptr = &i;
    *ptr = 0 ;
    puts("no problem\n");
    return 0;
}
```

```c
#include <stdio.h>
int main()
{
    int i = 0 ;
    int *ptr = &i;
    *ptr = 0 ;
    puts("no problem\n");
    return 0;
}
```

```
$ frama-c -machdep gcc_x86_64 test.c -e-acsl-prepare -rte -then -e-acsl -then-last -print -ocode test.e-acsl.c

$ gcc … test.e-acsl.c -o test.e-acsl
```

```c
#include <stdio.h>
int main()
{
    int i = 0 ;
    int *ptr = &i;
    *ptr = 0 ;
    puts("no problem\n");
    return 0;
}
```

```
$ frama-c -machdep gcc_x86_64 test.c -e-acsl-prepare -rte -then -e-acsl -then-last -print -ocode test.e-acsl.c

$ gcc … test.e-acsl.c -o test.e-acsl
$ ./test.e-acsl

noproblem
```

```c
#include <stdio.h>
int main()
{
    int i = 0 ;
    int *ptr = &i;
    *(ptr+1) = 0 ;
    puts("no problem\n");
    return 0;
}
```

```c
#include <stdio.h>
int main()
{
    int i = 0 ;
    int *ptr = &i;
    *(ptr+1) = 0 ;
    puts("no problem\n");
    return 0;
}
```

```
$ frama-c -machdep gcc_x86_64 test.c -e-acsl-prepare -rte -then -e-acsl -then-last -print -ocode test.e-acsl.c

$ gcc … test.e-acsl.c -o test.e-acsl
```

```c
#include <stdio.h>
int main()
{
    int i = 0 ;
    int *ptr = &i;
    *(ptr+1) = 0 ;
    puts("no problem\n");
    return 0;
}
```

```
$ frama-c -machdep gcc_x86_64 test.c -e-acsl-prepare -rte -then -e-acsl -then-last -print -ocode test.e-acsl.c

$ gcc … test.e-acsl.c -o test.e-acsl
$ ./test.e-acsl


Assertion failed at line 6 in function main.
The failing predicate is:
rte: mem_access: \valid(ptr + 1).
Aborted
```

2) Contiki-NG and Continuous Integration

Contiki-NG :

- OS for low-power wireless IoT devices
- C language
- BSD license
- TCP/IPv6 stack (uIPv6), coap, RPL routing protocol, 6LowPAN header compression, IEEE 802.15.4 radio
- protothread (mixture between multithreading and event-driven programming)

Git :

- Github workflow (develop branch + master branch + release branches)
- Github-Travis plugin with an Ubuntu 32bits Docker image

→ each commit has to pass the tests

Compilation tests :

- native (x86 architecture) compilation
- ARM architecture compilation

→ only a subset, only a few compilation options

Run tests :

- native (x86 architecture)

→ only 2 tests...

→ obviously no native ARM tests (no QEMU)...

- COOJA : a JAVA-based network simulator

→ native and ARM motes

→ test the whole IPv6-RPL-MAC 80215.4 stack with realist RPL DODAG routing topology

Contiki-NG can be compiled as :

- a firmware for an ARM IoT device
- a ELF executable for native host (Linux-x86)
- a dynamic library for COOJA

From now, we call :

- C hello world = a single `printf`
- Contiki-NG hello-world = C hello-world + full IPv6 pingable stack.

Contiki-NG can be compiled as :

- a firmware for an ARM IoT device
- a ELF executable for native host (Linux-x86)
- a dynamic library for COOJA

From now, we call :

- C hello world = a single `printf`
- Contiki-NG hello-world = C hello-world + full IPv6 pingable stack.

Demo1 :

- Contiki-NG hello-world native compilation + run
- E-ACSL'ed Contiki-NG hello-world compilation + run

Contiki-NG can be compiled as :

- a firmware for an ARM IoT device
- a ELF executable for native host (Linux-x86)
- a dynamic library for COOJA

From now, we call :

- C hello world = a single `printf`
- Contiki-NG hello-world = C hello-world + full IPv6 pingable stack.

Demo1 :

- Contiki-NG hello-world native compilation + run
- E-ACSL'ed Contiki-NG hello-world compilation + run

We want Contiki-NG as a dynamic library and run it in Cooja !
$\Rightarrow$ have to deal with Java

```
224 /** Shadow Layout {{{ */
225 /*****************************************************************************
226  * Memory Layout *************************************************************
227  *****************************************************************************
228   ---------------------------------------> Max address
229   Kernel Space
230   ---------------------------------------->
231   Non-canonical address space (only in 64-bit)
232   ---------------------------------------->
233   Environment variables [ GLIBC extension ]
234  ---------------------------------------->
235   Program arguments [ argc, argv ]
236  ---------------------------------------> Stack End
237   Stack [ Grows downwards ]
238  ---------------------------------------->
239   Thread-local storage (TLS) [ TDATA and TBSS ]
240  ---------------------------------------->
241   Shadow memory [ Heap, Stack, Global, TLS ]
242  ---------------------------------------->
243   Object mappings
244  ---------------------------------------->
245  ---------------------------------------->
246   Heap [ Grows upwards^ ]
247  ---------------------------------------> Heap Start [Initial Brk]
248   BSS Segment  [ Uninitialised Globals ]
249  ---------------------------------------->
250   Data Segment [ Initialised Globals   ]
251  ---------------------------------------->
252   RODataa [ Potentially ]
253  ---------------------------------------->
254   Text Segment [ Constants ]
255  ---------------------------------------> NULL (0)
256  *****************************************************************************
257 NOTE: Above memory layout scheme generally applies to Linux Kernel/gcc/glibc.
258   It is also an approximation slanted towards 64-bit virtual process layout.
259   In reality layouts may vary. Also, with mmap allocations heap does not
260   necessarily grows from program break upwards. Typically mmap will allocate
261   memory somewhere closer to stack. */
262
263 /* Struct representing a contigous memory region. Effectively this describes
264  * a memory segment, such as heap, stack or segments in the shadow memory
265  * used to track them. */
266 struct memory_segment {
267   const char *name; //!< Symbolic name
268   size_t    size; //!< Byte-size
269   uintptr_t start; //!< Least address
270   uintptr_t end; //!< Greatest address
271   mspace    mspace; // !< Mspace used for the partition
272   /* The following are only set if the segment is a shadow segment */
273   struct memory_segment *parent; //!< Pointer to the tracked segment
274   size_t shadow_ratio; //!< Ratio of shadow to application memory
275   /*!< Offset between the start of the tracked segment and the start of this
276      segment */
277   intptr_t shadow_offset;
278 };
279
280 typedef struct memory_segment memory_segment;
e_acsl_shadow_layout.h
```

$\Rightarrow$ have to modify E-ACSL to take in account the specific memory layout of a (JNI) dynamic library

3) Contiki-NG in Cooja ?

```java
import java.util.Scanner;

public class Loader {
    native public static int getpid();
    native public static void jnimain();

    public static void main(String[] args) {
        /* Load JNI library */
        String mainlibname = "framac.eacsl";
        if (args.length >= 1)
        mainlibname = args[0];
        System.loadLibrary(mainlibname);
        String libmainlibname = "lib" + mainlibname + ".so";
        System.out.println(libmainlibname + " loaded !");
        /* Print PID (JNI function) and wait for Enter */
        int pid = getpid();
        System.out.println("Pid " + pid);
        Scanner reader = new Scanner(System.in);
        System.out.println("Press Enter...");
        reader.nextLine();
        /* Main (JNI function) */
        jnimain();
    }
}
```

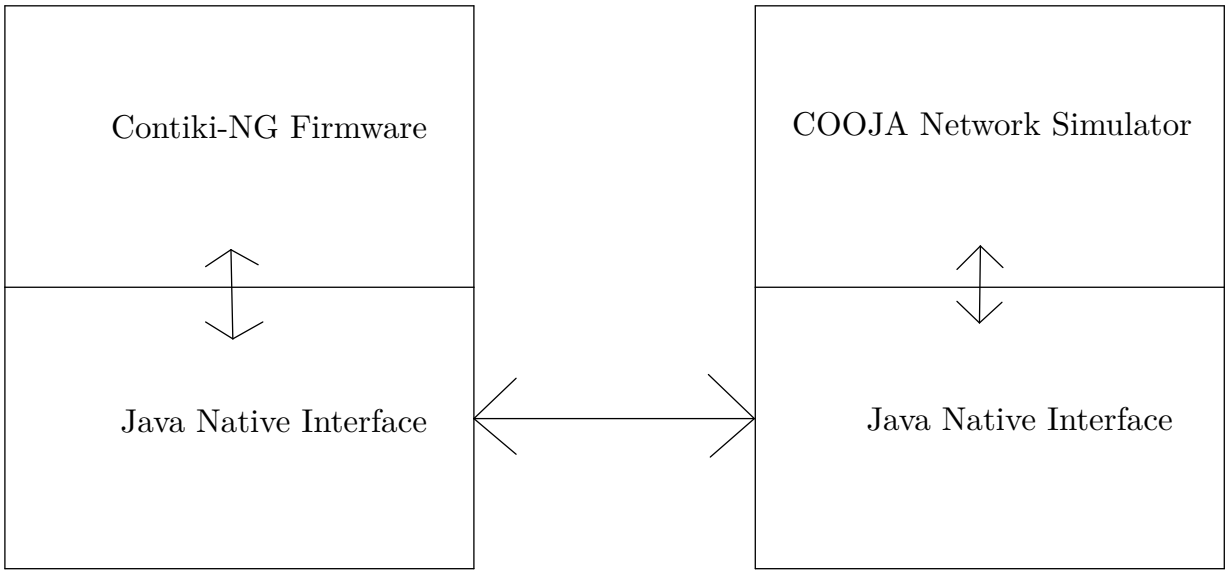
```java
import java.util.Scanner;

public class Loader {
    native public static int getpid();
    native public static void jnimain();

    public static void main(String[] args) {
        /* Load JNI library */
        String mainlibname = "framac.eacsl";
        if (args.length >= 1)
        mainlibname = args[0];
        System.loadLibrary(mainlibname);
        String libmainlibname = "lib" + mainlibname + ".so";
        System.out.println(libmainlibname + " loaded !");
        /* Print PID (JNI function) and wait for Enter */
        int pid = getpid();
        System.out.println("Pid " + pid);
        Scanner reader = new Scanner(System.in);
        System.out.println("Press Enter...");
        reader.nextLine();
        /* Main (JNI function) */
        jnimain();
    }
}
```

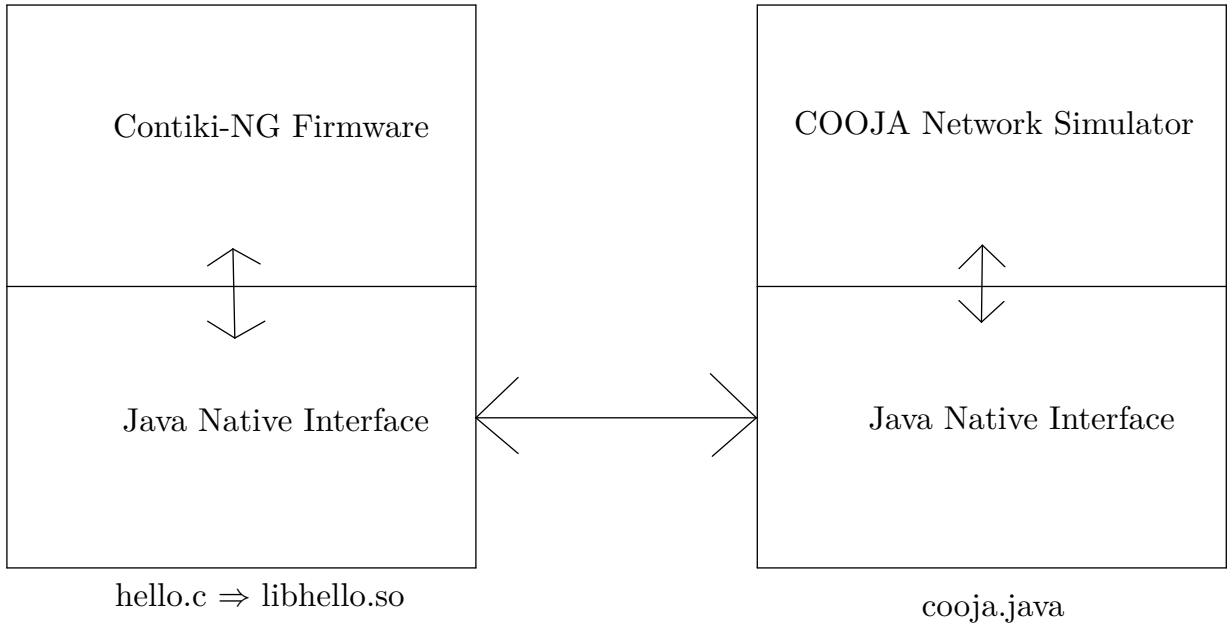
Demo2 (experiments with JAVA) :

- C hello-world loaded in Java
- E-ACSL'ed C hello-world loaded in Java
- E-ACSL'ed native Contiki-NG C hello-world loaded in Java



hello.c ⇒ libhello.so

cooja.java



Demo3 (experiments with Cooja) :

- Frama-C + Cooja tests/07-simulation-base/02-ringbufindex.csc
- Frama-C + E-ACSL + Cooja tests/07-simulation-base/02-ringbufindex.csc

4) Conclusion

Conclusion :

- E-ACSL can be used in a dynamic library
- still some work to do to make E-ACSL usable in Cooja