# CEA Contribution & Future Work


Julien Signoles

CEA LIST

Software Safety Labs


Hi-Lite Final Meeting

May 2013, the 29th

Simplifying the use of formal methods

1. what's the status before Hi-Lite (Apr. 2010)
2. what's done during Hi-Lite
   - E-ACSL language
   - E-ACSL plug-in
   - improving Frama-C
3. what's planned to do next

# Situation in April 2010

- Frama-C Boron
- only designed for static analyzers
- mostly no way to combine them to verify program properties
- ACSL specification language
- PathCrawler test generation tool

# Situation in May 2013

- Frama-C Fluorine: Boron + 4
- now designed both for static and dynamic analyzers
- combining analyses is effective
- ACSL + E-ACSL specification languages
- PathCrawler as a Frama-C Plug-in

What was done during these 3 years in Hi-Lite?

# Initial Needs

## Testing tools:

- require a precondition specifying valid inputs
- require an oracle to decide whether a test is correct

## Abstract interpreters:

- require a precondition and assertions to be precise

## Program proving tools:

- require a formal specification
- based on pre/post-conditions

Combining them requires a common specification language

# Proposal

- E-ACSL: Executable ANSI/ISO C Specification Language
  - builds a bridge between static and dynamic analysis tools
  - based on pre-existing ACSL language used by Frama-C
- E-ACSL plug-in converts E-ACSL specifications into C code
  - Frama-C plug-in
  - runtime assertion checking
  - helpful for debuging specification
  - may easily be used by any analysis tool for C

# E-ACSL: Executable-ACSL

## Benefits:

- being executable allows to be understandable by dynamic tools (testing tools, monitors)
- being based on ACSL allows to be supported by existing Frama-C analyzers
- being translatable into C allows to be supported by other analysis tools for C

## Differences with ACSL:

- few restrictions
- one extension: iterators over recursive datastructures
- compatible semantics changes

# E-ACSL Restrictions

- quantifications must be guarded

  `\forall` $\tau$ $x_1, \ldots, x_n$;
  $a_1$ `<=` $x_1$ `<=` $b_1$ `&&` ... `&&` $a_n$ `<=` $x_n$ `<=` $b_n$
  `==> p`

  `\exists` $\tau$ $x_1, \ldots, x_n$;
  $a_1$ `<=` $x_1$ `<=` $b_1$ `&&` ... `&&` $a_n$ `<=` $x_n$ `<=` $b_n$
  `&& p`

- sets must be finite
- loop invariants are simply equivalent to 2 assertions
- no way to express termination properties
- backwards C labels only

## Iterators over C recursive datastructures

```
// type of binary trees
struct btree {
  int val;
  struct btree *left, *right;
};

// declare an iterator over a binary tree
/*@ iterator access(_, struct btree *t):
  @   nexts t->left, t->right;
  @   guards \valid(t->left), \valid(t->right); */

// is_even(t) is valid iff all values in the binary tree t are even
/*@ predicate is_even(struct btree *t) =
  @   \forall struct btree *tt;
  @     access(tt, t) ==> tt->val % 2 == 0; */
```

# E-ACSL Integers

- mathematical integers to preserve ACSL semantics
- many advantages compared to bounded integers
  - automatic theorem provers work much better with such integers than with bounded integers arithmetics
  - specify without implementation details in mind
  - still possible to use bounded integers when required
  - much easier to specify overflows

# E-ACSL Undefinedness

- ACSL logic is total and 1/0 is logically significant
  - help the user to write simple specification like u/v == 2
  - 1/0 is defined but not executable
- E-ACSL logic is 3-valued
  - the semantics of 1/0 is "undefined"
  - lazy operators `&&`, `||`, `_?_:_`, `==>`
  - correspond to Chalin's Runtime Assertion Checking semantics
  - consistent with ACSL: valid (resp. invalid) E-ACSL predicates remain valid (resp. invalid) in ACSL

# E-ACSL plug-in at a Glance

- convert E-ACSL annotations into C code
- implemented as a Frama-C plug-in

```c
int div(int x, int y) {
  /*@ assert y-1 != 0; */
  return x / (y-1);
}
```

E-ACSL →

```c
int div(int x, int y) {
  /*@ assert y-1 != 0; */
  e_acsl_assert(y-1 != 0);
  return x / (y-1);
}
```

# E-ACSL plug-in at a Glance

- convert E-ACSL annotations into C code
- implemented as a Frama-C plug-in

```
int div(int x, int y) {
  /*@ assert y-1 != 0; */
  return x / (y-1);
}
```

E-ACSL →

```
int div(int x, int y) {
  /*@ assert y-1 != 0; */
  e_acsl_assert(y-1 != 0);
  return x / (y-1);
}
```

- the general translation is more complex than it may look
  - `\result` requires to introduce extra-variables
  - `\at(x,L)` requires to introduce code at L
  - ...

# E-ACSL Plug-in Integer Support

- use GMP library for mathematical integers

```
/*@ assert y-1 == 0; */
mpz_t e_acsl_1, e_acsl_2, e_acsl_3, e_acsl_4;
int e_acsl_5;
mpz_init_set_si(e_acsl_1, y);                      // e_acsl_1 = y
mpz_init_set_si(e_acsl_2, 1);                      // e_acsl_2 = 1
mpz_init(e_acsl_3);
mpz_sub(e_acsl_3, e_acsl_1, e_acsl_2);   // e_acsl_3 = y-1
mpz_init_set_si(e_acsl_4, 0);                      // e_acsl_4 = 0
e_acsl_5 = mpz_cmp(e_acsl_3, e_acsl_4); // (y-1) == 0
e_acsl_assert(e_acsl_5 == 0);                   // runtime check
mpz_clear(e_acsl_1); mpz_clear(e_acsl_2); // deallocate
mpz_clear(e_acsl_3); mpz_clear(e_acsl_4);
```

# E-ACSL Plug-in Integer Support

- use GMP library for mathematical integers

```
/*@ assert y-1 == 0; */
mpz_t e_acsl_1, e_acsl_2, e_acsl_3, e_acsl_4;
int e_acsl_5;
mpz_init_set_si(e_acsl_1, y);                   // e_acsl_1 = y
mpz_init_set_si(e_acsl_2, 1);                   // e_acsl_2 = 1
mpz_init(e_acsl_3);
mpz_sub(e_acsl_3, e_acsl_1, e_acsl_2);          // e_acsl_3 = y-1
mpz_init_set_si(e_acsl_4, 0);                   // e_acsl_4 = 0
e_acsl_5 = mpz_cmp(e_acsl_3, e_acsl_4);         // (y-1) == 0
e_acsl_assert(e_acsl_5 == 0);                   // runtime check
mpz_clear(e_acsl_1); mpz_clear(e_acsl_2);       // deallocate
mpz_clear(e_acsl_3); mpz_clear(e_acsl_4);
```

- design a type system to detect when GMP is really required
- infer a correct interval for any term, as small as possible
- almost no GMP in practice :-)

must prevent introducing RTE when translating annotations

```
int foo(int u, int v) {
  /*@ assert u/v == 2; */
  return u/v;
}
```

# E-ACSL Plug-in RTE Detection

must prevent introducing RTE when translating annotations

```
int foo(int u, int v) {
  /*@ assert u/v == 2; */
  return u/v;
}
```

E-ACSL →

```
int foo(int u, int v) {
  /*@ assert u/v == 2; */
  e_acsl_assert(u/v == 2);
  return u/v;
}
```

# E-ACSL Plug-in RTE Detection

must prevent introducing RTE when translating annotations

```
int foo(int u, int v) {
  /*@ assert u/v == 2; */
  return u/v;
}
```

E-ACSL →

```
int foo(int u, int v) {
  /*@ assert u/v == 2; */
  e_acsl_assert(u/v == 2);
  return u/v;
}
```

↓ RTE plug-in

```
int foo(int u, int v) {
  /*@ assert v != 0; */
  /*@ assert u/v == 2; */
  e_acsl_assert(u/v == 2);
  return u/v;
}
```

# E-ACSL Plug-in RTE Detection

must prevent introducing RTE when translating annotations

```
int foo(int u, int v) {
  /*@ assert u/v == 2; */
  return u/v;
}
```

E-ACSL →

```
int foo(int u, int v) {
  /*@ assert u/v == 2; */
  e_acsl_assert(u/v == 2);
  return u/v;
}
```

↓ RTE plug-in

```
int foo(int u, int v) {
  /*@ assert v != 0; */
  /*@ assert u/v == 2; */
  e_acsl_assert(u/v == 2);
  return u/v;
}
```

← E-ACSL

```
int foo(int u, int v) {
  /*@ assert v != 0; */
  e_acsl_assert(v != 0);
  /*@ assert u/v == 2; */
  e_acsl_assert(u/v == 2);
  return u/v;
}
```

# E-ACSL Plug-in Memory Observation

- memory-related constructs like `\valid` require to know the memory structure at runtime
- C library for memory observation
- used by E-ACSL Plug-in
- once again the translation is quite heavy
- backward dataflow analysis to instrument the code only when required

# E-ACSL Publications

- J. Signoles.
  E-ACSL User Manual.
  May 2013.
- M. Delahaye, N. Kosmatov and J. Signoles.
  Common Specification Language for Static and Dynamic Analysis of C Programs.
  SAC'13. March 2013.
- N. Kosmatov, G. Petiot and J. Signoles.
  Optimized Memory Monitoring for Runtime Assertion Checking of C Programs.
  Submitted article.
- N. Kosmatov and J. Signoles.
  Runtime Assertion Checking with Frama-C.
  Submitted tutorial.

# Combining Analysins within Frama-C

- how to ensure the safety of an annotated program
- by using several customizable analyzers
- based on different techniques?
- a "consolidation algorithm" merges all the results coming from the different analyzers with their different configurations
- potential results are:
  - valid
  - unknown
  - invalid
  - inconsistent
  - a variety of refinement (never tried, dead annotations, ...)

- the consolidation algorihm is correct

  if each analyzer is correct, then the algorithm returns “Valid” (resp. “Invalid”) for a valid (resp. invalid) property. It returns “Inconsistent” if there are both a proof of validity and invalidity.

- the consolidation algorithm is complete

  if each analyzer is correct and indicates the right hypotheses, and if one analyzer does not indicate “Dont know” under recursively valid hypotheses, then the computed status is either “Valid” or “Invalid”.

# Frama-C Publications

- L. Correnson and J. Signoles.
  Combining Analyses for C Program Verification.
  FMICS'12. Aug. 2012.
- P. Cuoq, F. Kirchner, N. Kosmatov, V. Prevosto, J. Signoles and B. Yakobowski.
  Frama-C, A Software Analysis Perspective.
  SEFM'12. Oct. 2012.
  Selected for journal publication.
- P. Cuoq, D. Doligez and J. Signoles.
  Lightweight Typed Customizable Unmarshaling.
  ML'11. Sep. 2011.

# Future Work

## E-ACSL development

- support missing constructs:
  - assigns and loop assigns
  - logic functions and predicates
  - loop invariants
  - complete and disjoint behaviors
  - ...
- temporal memory safety (balancing of malloc/free, ...)
- memory profiling
- improve the instrumentation: more optimizations
- proof of E-ACSL optimized instrumentation

frama C
Software Analyzers

# Future Work

## Application of E-ACSL

- E-ACSL was initially designed for runtime assertion checking
- debugging specifications
  - before proving program
  - teaching
- monitoring
  - security application
  - combining monitoring and static analysis
  - demo this afternoon
- combining test and static analysis

- 1 opened Phd position
  - Formalization of E-ACSL within Coq
- 1 submitted French ANR project
  - combining static and dynamic analyses
  - fully centered around E-ACSL
- 1 European Artemis project being submitted
  - security-oriented
  - E-ACSL for monitoring on a simulator
- Sec4Safe
  - when, where, what, who? :-)
- Other projects?

# Future Work

## Also...

- Tool collaborations
- Language Collaborations
  - Mixed C/Ada program verification
- Tools/analysis/language Collaborations in a certification context
  - 1 opened Phd position (combining test and proof)

SRI's Evidential Tool Bus

# Conclusion

- E-ACSL: new executable specification language for C
- implemented as a Frama-C plug-in
- combining analysis is now effective within Frama-C
- 3 articles + 1 submitted, 1 short paper, 1 submitted tutorial
- several potential applications
- a lot of works remain (both theoretical and practical)