Software Analyzers

# ACSL-importer Plug-in Manual

For Frama-C 32.1 (Germanium)

Patrick Baudin

# Contents

# Overview 1

This is the user manual of the ACSL [2, 1] importer plug-in of Frama-C[1]. The content of this document corresponds to the version 32.1 (Germanium), released on March 16, 2026, of Frama-C.

This plug-in adds to Frama-C support for out of C source files ACSL specification. Therefore it is useful only to people that intend to write ACSL specifications and do not want to write them inside comments of the C source files as documented in [2].

In addition, the plug-in allows to enter ACSL annotations directly via the Graphical User Interface.

[1] http://frama-c.com

# Usage 2

You may refer to the user manual of Frama-C [4] for non specific options of Frama-C.

After its installation, the plug-in is listed by `frama-c -plugins` as `ACSL importer`.

All options of the plug-in are prefixed by `-acsl-import`.

The main options are:

- `-acsl-import-help` lists all options specific to the plug-in
- `-acsl-import <filename list>` analyzes the content of files listed in its arguments and inserts the correctly typed specifications in the abstract syntax tree of Frama-C for further processing. The file name list is a comma separated list of file names. It can be repeated many times and its effect is cumulative.
- `-acsl-import-include-dirs <dirname list>` specifies the list of directories where the relative file names mentioned in `include` directives are searched.
- `-acsl-import-keep-unused-symbols` configures Frama-C kernel such that unused C symbols aren't removed before the import.
- `-acsl-import-no-run` doesn't run the plug-in, just configure its parameters.
- `-acsl-import-parse-only` parses the ACSL files without typing nor imports.
- `-acsl-import-typing-only` parses and types the ACSL files without imports.

For example if you want to check your specification written in the file named `specif.acsl` for the program in `example.c` using the value analysis plug-in[3], you may use the following command:

```
# frama-c -acsl-import specif.acsl example.c -then -eva
```

The option `-then` ensures that the specification is imported before running the value analysis.

## Grammar for external ACSL files

An external ACSL file is a text file containing any <specification> conforming to the grammar described in table 2.1.

Non terminals starting with <acsl_> correspond to non terminals without the <acsl_> in the original ACSL grammar[2].

The importation semantics is given by the following rules:

- `module f : <S>` means that the specification in `S` has to be imported into the C file named `f.c`
- `include "f.acsl"` means that the file named `f.acsl` has to be included verbatim into the current file, as if its content was inlined. The included file has to contain a valid `<specification>`. If `f.acsl` is a relative filename the plug-in will look for first inside the directory containing file containing the `include` directive and then in first directory of the list of directories given to the option `-acsl-import-include-dirs` that contains `f.acsl`.
- `let m = <T>;` defines a macro named `m`. All occurrences of the macro `m` in current scope will be expanded verbatim by the ACSL term `T`. The `let` is untyped but each expansion of `m` into `T`

```
<specification>          :=  (("module" <module_name> ":")* <module_specification>)* ;

<module_specification>   :=  <global_specification>* <fun_specification>* ;

<global_specification>   :=  <include_directive> | <macro_directive>
                           | <acsl_logical_axiomatic> | <acsl_logical_type>
                           | <acsl_logical_predicate> | <acsl_logical_function>
                           | <acsl_volatile_clause> ;

<include_directive>      :=  "include" ""<file_name>"" ";" ;

<macro_directive>        :=  ("global" ?) let" <identifier> "=" <acsl_term> ";" ;

<fun_specification>      :=  "function" <fun_name> ":"
                             (<fun_spec> | <stmt_spec> | <global_specification>)* ;

<fun_spec>               :=  "contract" ":" <acsl_fun_spec> ;

<stmt_spec>              :=  <stmt_ref> ":" (<acsl_loop_annotation>
                           | <acsl_code_annotation>
                           | "contract" ":" <acsl_stmt_spec>) ;

<stmt_ref>               :=  "at" (<loop_id>
                           | <stmt_id>
                           | <call_id>) ;

<loop_id>                :=  "loop" <number>
                           | "loop" "body" <number>;
                           | "loop" "stmt" <number>;

<stmt_id>                :=  <number>
                           | "return"
                           | <label_name> ;

<call_id>                :=  "asm" <number>
                           | "indirect_call" <number>*
                           | "call" <fun_name> <number>*
                           | "call" <number> ;
```

Table 2.1: External ACSL grammar

will be typed in its own context. The considered scope is the whole file for macro defined before the first `module` markup, the current module before the first `function` markup and the current function otherwise. Macros can be located into an included file to be shared between specifications of different imported files.

- `global let m = <T>;` defines a macro named `m` within the file scope.
- `function f : contract : <S>` means that the specification in `S` has to be imported as a specification for the C function named `f`
- `function f : <stmt_ref> : <S>` means that the specification in `S` has to be imported as a specification for the statement denoted by `stmt_ref` inside the body of the C function named `f`
- `at loop <i>` denotes the loop of number `i`, inside the current function, counting from the top

of the definition of the function and starting from 1. The ACSL loop annotations refer to the loop statement itself, whereas ACSL code annotations and statement contracts refer to the loop body.
- `at loop stmt <i>` as previous, but denotes always the loop stmt (notice that this statement never includes the inializer part of `for` loops),
- `at loop body <i>` as previous, but denotes always the loop stmt (notice that this statement never includes the increment part of `for` loops),
- `at asm <i>` denotes the asm call of number `i`, inside the current function, counting from the top of the definition of the function and starting from 1.
- `at indirect_call <i>` denotes the indirect call (via function pointer) of number `i`, inside the current function, counting from the top of the definition of the function and starting from 1. To refers all indirect calls from the current function, do not give a value `i`.
- `at call <fun_name> <i>` denotes the direct call to the function `fun_name` of number `i`, inside the current function, counting from the top of the definition of the function and starting from 1. To refers all direct calls to `fun_name` from the current function, do not give a value `i`.
- `at call <i>` denotes the direct call of number `i`, inside the current function, counting from the top of the definition of the function and starting from 1.
- `at <L>` denotes the statement designated by the C label `L`, inside the current function. In addition to the C labels the special keyword `return` is interpreted as pointing to the `return` statement of the C function.
- `at <i>` denotes the statement of number `i`, inside the current function, counting from the top of the definition of the function (in its internal form) and starting from 1. Note that this number may be counter-intuitive to compute and may depend on the version of the Frama-C platform.

Here is a small example C program contained in file `demo.c`:

```
int f(int i);
int g(int j);
void job(int *t, int A) {
  for(int i = 0; i < 50; i++) t[i] = f(i);
  for(int j = A; j < 100; j++) t[j] = g(j);
}
```

and here is the content of the file `demo.acsl`:

```
module demo :
 axiomatic A {
 type event = B | C(integer);
 predicate P(integer x);
 logic integer phi(event e);
 }

 function f:
contract:
  ensures P(phi(C(\result)));
  assigns \nothing;

function g:
contract: assigns \nothing;

function job:
at 1: assert 50 <= A <= 100;
at loop 1:
  loop invariant 0 <= i <= 50;
  loop invariant \forall integer k; 0 <= k < i ==> P(t[k]);
  loop assigns i,t[0..49];
at loop body 1:
  contract:
```

```
   ensures i == \old(i);
at loop stmt 1:
  contract:
    requires i==0;
    ensures i==50;

at loop 2:
  loop invariant A <= j <= 100;
  loop assigns j,t[A..99];

at return:
  assert \forall integer k; 0 <= k < 50 ==> P(t[k]);

at call f:
  contract:
    ensures P(phi(C(t[i])));
```

Here is the result of the :

```
# frama-c -acsl-import demo.acsl demo.c -print -no-unicode
[kernel] Parsing demo.c (with preprocessing)
[acsl-importer] Success for demo.acsl
[acsl-importer] Done: 1 file.
[kernel] Parsing demo.c (with preprocessing)
[acsl-importer] Success for demo.acsl
[acsl-importer] Done: 1 file.
/* Generated by Frama-C */
/*@
axiomatic A {
  type event = B | C(integer);
  predicate P(integer x) ;
  logic integer phi(event e) ;
  }
 */
/*@ ensures P(phi(C(\result)));
   assigns \nothing; */
int f(int i);

/*@ assigns \nothing; */
int g(int j);

void job(int *t, int A) {
  /*@ assert 50 <= A <= 100; */
  {
    int i = 0;
    /*@ requires i == 0;
       ensures i == 50; */
    /*@ loop invariant 0 <= i <= 50;
       loop invariant \forall integer k; 0 <= k < i ==> P(*(t + k));
       loop assigns i, *(t + (0 .. 49));
    */
    while (i < 50) {
      /*@ ensures i == \old(i); */
      {
        /*@ ensures P(phi(C(*(t + i)))); */
        *(t + i) = f(i);
      }
      i ++;
```

```
    }
  }
  {
    int j = A;
    /*@ loop invariant A <= j <= 100;
       loop assigns j, *(t + (A .. 99)); */
    while (j < 100) {
      {
       *(t + j) = g(j);
      }
      j ++;
    }
  }
  /*@ assert \forall integer k; 0 <= k < 50 ==> P(*(t + k)); */
  return;
}

int T[100];
void main(void) {
  job(T,50);
  return;
}
```

## Adding properties from the Graphical User Interface

Contextual menus are added when selecting statements or function definitions. It is possible to insert `acsl_logical_axiomatic`, `acsl_logical_type`, `acsl_logical_predicate`, `acsl_logical_function`, `acsl_volatile_clause`, `acsl_stmt_spec`, `acsl_loop_annotation` and `acsl_code_annotation`. Selection of such a menu opens a popup window allowing to type directly the corresponding annotation (i.e. `ensures P(t[0]);`).

## Addons for ACSL

The option `-acsl-import-addon-ensures-and-exits` adds a new clause `ensures_and_exits` to ACSL grammar. This keyword is syntactic sugar for writing two clauses (one `ensures` and one `exits`) in one.

The option `-acsl-import-addon-integer-cast` allows the introduction of casts from integer to C integral types. Warnings of the category `annot:integer-cast` are emitted when introducing such casts.

## Addons for code transformations

From Frama-C kernel, it is possible to identify two kinds of code transformations:

- transformations performed during the elaboration of the AST (Abstract Syntaxtic Tree) related to the parsed C files, and
- transformations performed over the AST.

The import of ACSL specification files can be viewed as a second class transformation adding to the ACSL annotation to the current AST. But, some imports need a code transformation of the first class for beeing performed (i.e. for identifying the statement body of the $i^{th}$ loop). All syntaxic code transformations of Frama-C kernel (including the deletion of unused C symbols) are processed before the import of ACSL specification files.

The option `-acsl-import-unroll-loop-conditions` transforms `while` loops and and `for` loops into `do-while` loops when the initial loops have no loop annotation.

The `while` loop

```
while (c)
  //@ invariant I;
  S;
```

is transformed into :

```
if (c)
  do //@ invariant I;
    S;
  while (c);
```

The `for` loop

```
for (init; c; incr)
  //@ invariant I;
  S;
```

is transformed into :

```
init ;
if (c)
  do //@ invariant I;
  { S;
    incr;
  } while (c);
```

For `do-while` loops, `invariant` clauses set on their loop body are equivalent to `loop invariant` clauses. So, the importer is able to transform `invariant` clauses set on every body of loops of every kind into `loop invariant` clauses of `do-while` loops.

This transformation is a first class transformation processed before the loop unrolling related to the option `-kernel-ulevel <n>`. Since the code transformation processed by the option `-kernel-ulevel <n>` is also a first class transformation, the use of the options `-kernel-ulevel -1 <acsl-import-options> -then -acsl-import-run -acsl-ulevel <n>` is required to perform loop unrolling after the import of ACSL specification files.

The option `-acsl-import-ulevel <n>` unrolls loops as `-kernel-ulevel <n>` does at the parsing stage, except that is now a second class transformation. This means that a code annotation imported to a statement of the body of an unrolled loops can be duplicated using option `-acsl-import-ulevel <n>` as if it was inside the C file and using the option `-kernel-ulevel <n>` instead. These `UNROLL` loop pragmas could be in the source code, imported from ACSL specification files or generated by the `-acsl-import-ulevel-spec` option.

The option `-acsl-import-ulevel-spec <spec`$_1$`,...,spec`$_s$`>` adds `UNROLL` loop pragmas that can be used by both `-acsl-import-ulevel` and `-kernel-ulevel` transformations. An unrolling specification `<m@f:tag@n>` adds an annotation '`loop unfold "tag", <n>;`' to the $m_{th}$ loop of the function `<f>`. An unrolling specification `<c@f:tag@n>` adds an annotation '`loop unfold "tag", <n>;`' to all loops of category `<c>` of the function `<f>`, where allowed loop categories are: `while`, `for` and `do-while`.
A specification is considered as a set of elementary specifications: `spec`$_1$, ..., `spec`$_s$.
Categories, function names and loop occurrence numbers can be omitted. The priority ordering used for choosing the (`"tag"` name, unrolling value `<n>`) pair is: `<m@f:tag@n>` > `<c@f:tag@n>` >

<f:tag@n> > <c:tag@n> > <:tag@n>.
The default value for optional tags is the empty string which leads to add unrolling loop pragmas without tags.
Nothing is done for loops having already a clause 'loop unfold ...'.

The command line

```
# frama-c -acsl-import-unroll-loop-conditions -ulevel-spec "fct:completely@2" file.c
```

will transforms the function

```
void fct (void) {
  S1;
  for (i=0; i<2; i++)
    Si;
  Sn ;
}
```

such that the remaining loop have to be *death* code:

```
void fct (void) {
  S1;
  i=0;
  if (i<2){ /*** loop condition unrolled ***/
    Si;   /*** first unrolling ***/
    i ++;
    if (! (i < 2)) goto unrolling_2_loop;
      Si;  /*** second unrolling ***/
      i ++;
      if (! (i < 2)) { goto unrolling_2_loop; }
      /*** the loop should be death code since completely unrolled ***/
      /*@ loop invariant \false;
          loop unfold "completely", 2; */
      while (1) {
        Si;
        i ++;
        if (! (i < 2)) break;
        }
unrolling_2_loop:
    }
  Sn ;
}
```

# Changes

## Frama-C 32.0 (Germanium)

- The plug-in is now part of the Frama-C open-source distribution
- Options are now prefixed with `-acsl-import`

## From plugin ACSL importer 1.11.27 (Frama-C 27 Cobalt)

- Minor changes for compatibility with Frama-C 27.

## From plugin ACSL importer 1.11.26 (Frama-C 25 Iron)

- The build system moves from `Make` files to `Dune` files.

## From plugin ACSL importer 1.11.25 (Frama-C 25 Manganese)

- Minor changes for compatibility with Frama-C 25.

## From plugin ACSL importer 1.11.24 (Frama-C 24 Chromium)

- Fixes import of the clause terminates.

## From plugin ACSL importer 1.10.23 (Frama-C 23 Vanadium)

- Fixes demo example (the global axiomatic to import was missing).

## From plugin ACSL importer 1.9.19 (Frama-C 19 Potassium)

- Fixes import of contracts related to labeled statements.

– Fixes scope rule for searching variables under `\at` construct.
– Fixes individual status for `ensures_and_exits` extended clause.

## From plugin ACSL importer 1.8.18 (Frama-C 18 Argon)

– Adding options `-acsl-keep-unused-symbols`, `-acsl-parse-only` and `acsl-typing-only` options.

## From plugin ACSL importer 1.7.17 (Frama-C 17 Chlorin

– Extended grammar allowing the import of statement contracts and code annotations to loops.

## From plugin ACSL importer 1.6.15 (Frama-C 15 Phosphorus)

– Introduction of global macro.

## From plugin ACSL importer 1.5.14 (Frama-C 14 Silicon)

– Introduction of scopes for macro binding.

## From plugin ACSL importer 1.4.13 (Frama-C 13 Aluminium)

– Extended grammar allowing specification imports to C function call statements and assembly code.
– Review of the grammar related to “module” directive.

## From prom plugin ACSL importer 1.1.8 (Frama-C 8 Oxygen)

– Section “Addons for code transformation” added.
– Section “Adding properties from the Graphical User Interface” added.

## From plugin ACSL importer 1.0.7 (Frama-C 7 Nitrogen)

– Section “Addons for ACSL” added.
– Initial document revision.

# Bibliography


[1] Patrick Baudin, Pascal Cuoq, Jean-Christophe Filliâtre, Claude Marché, Benjamin Monate, Yannick Moy, and Virgile Prevosto. *ACSL: ANSI/ISO C Specification Language*. Latest Frama-C implementation release at https://frama-c.com/download/frama-c-acsl-implementation.pdf.

[2] Patrick Baudin, Jean-Christophe Filliâtre, Claude Marché, Benjamin Monate, Yannick Moy, and Virgile Prevosto. *ACSL: ANSI/ISO C Specification Language*, 2008. Latest release at https://frama-c.com/download/acsl.pdf.

[3] Bühler, David and Cuoq, Pascal and Yakobowski, Boris with Lemerre, Matthieu and Maroneze André and Perrelle, Valentin and Prevosto, Virgile. *Eva - The Evolved Value Analysis plug-in*. CEA LIST, Software Reliability Laboratory. Latest release at https://frama-c.com/download/frama-c-eva-manual.pdf.

[4] Loïc Correnson, Pascal Cuoq, Florent Kirchner, André M Maroneze, Virgile Prevosto, Armand Puccetti, Julien Signoles, and Boris Yakobowski. *Frama-C User Manual*. CEA LIST, Software Safety Laboratory. Latest release at https://frama-c.com/download/frama-c-user-manual.pdf.