Software Analyzers

E-ACSL

Executable ANSI/ISO C Specification Language

Version 1.16

E-ACSL

Executable ANSI/ISO C Specification Language

Version 1.16

Julien Signoles

CEA LIST, Software Reliability Laboratory

This work has been initially supported by the 'Hi-Lite' FUI project (FUI AAP 9).

Contents

Foreword

This document describes version 1.16 of the E-ACSL specification language. It is based on the ACSL specification language [1]. Features of both languages may still evolve in the future, even if we do our best to preserve backward compatibility.

Acknowledgements

We gratefully thank all the people who contributed to this document: Patrick Baudin, Bernard Botella, Loïc Correnson, Pascal Cuoq, Basile Desloges, Johannes Kanig, Fonenantsoa Maurica, David Mentré, Benjamin Monate, Yannick Moy and Virgile Prevosto.

Chapter 1

Introduction

This document is a reference manual for E-ACSL. E-ACSL is an acronym for "Executable ANSI/ISO C Specification Language". It is an "executable" subset of *stable* ACSL [1] implemented [2] in the Frama-C platform [5]. "Stable" means that no experimental ACSL feature is supported by E-ACSL. Contrary to ACSL, each E-ACSL specification is executable: it may be evaluated at runtime.

In this document, we assume that the reader has a good knowledge of both ACSL [1] and the ANSI C programming language [7, 8].

1.1 Organization of this document

This document is organized in the very same way that the reference manual of ACSL [1].

Instead of being a fully new reference manual, this document points out the differences between E-ACSL and ACSL. Each E-ACSL construct which is not pointed out must be considered to have the very same semantics than its ACSL counterpart. For clarity, each relevant grammar rules are given in BNF form in separate figures like the ACSL reference manual does. In these rules, constructs with semantic changes are displayed in blue.

1.2 Generalities about Annotations

No difference with ACSL.

1.3 Notations for grammars

No difference with ACSL.

Chapter 2

Specification language

2.1 Lexical rules

No difference with ACSL*.*

2.2 Logic expressions

No difference with ACSL*, but guarded quantificatication..*

More precisely, grammars of terms and binders presented respectively Figures 2.1 and 2.3 are the same than the one of ACSL, while Figure 2.2 presents grammar of predicates. The only difference between E-ACSL and ACSL predicates are quantifications.

Quantification

E-ACSL quantification must be computable. They are limited to two limited forms.

Guarded integer quantification Guarded universal quantification is denoted by

```
\forall τ x1,...,xn;
  a1 <= x1 <= b1 ... && an <= xn <= bn
  ==> p
```

and guarded existential quantification by

```
\exists τ x1,...,xn;
  a1 <= x1 <= b1 ... && an <= xn <= bn
  && p
```

Each variable must be guarded exactly once and the guard of x_i must appear before the guard of x_j if $i < j$ (*i.e.* order of guards must follow order of binders).

Following the definition, each quantified variable belongs to a finite interval. Since finite interval is only computable in practice for integers, this form of quantifier is limited to `integer` and its subtype. Thus there is no guarded quantification over `float`, `real`, C pointers or logic types.

```
literal   ::= \true | \false                           boolean constants
            | integer                                  integer constants
            | real                                     real constants
            | string                                   string constants
            | character                                character constants

bin-op    ::= + | - | * | / | % | << | >>
            | == | != | <= | >= | > | <
            | && | || | ^^                             boolean operations
            | & | | | --> | <--> | ^                   bitwise operations

unary-op  ::= + | -                                    unary plus and minus
            | !                                        boolean negation
            | ~                                        bitwise complementation
            | *                                        pointer dereferencing
            | &                                        address-of operator

term      ::= literal                                  literal constants
            | id                                       variables
            | unary-op term
            | term bin-op term
            | term [ term ]                            array access
            | { term \with [ term ] = term }           array functional modifier
            | term . id                                structure field access
            | { term \with . id = term }               field functional modifier
            | term -> id
            | ( type-expr ) term                       cast
            | id ( term (, term)* )                    function application
            | ( term )                                 parentheses
            | term ? term : term                       ternary condition
            | \let id = term ; term                    local binding
            | sizeof ( term )
            | sizeof ( C-type-name )
            | id : term                                syntactic naming
            | string : term                            syntactic naming
```

Figure 2.1: Grammar of terms

```
           rel-op  ::=  ==  |  !=  |  <=  |  >=  |  >  |  <

             pred  ::=  \true  |  \false
                     |  term (rel-op term)+                   comparisons
                     |  id ( term (, term)* )                 predicate application
                     |  ( pred )                              parentheses
                     |  pred && pred                          conjunction
                     |  pred || pred                          disjunction
                     |  pred ==> pred                         implication
                     |  pred <==> pred                        equivalence
                     |  ! pred                                negation
                     |  pred ^^ pred                          exclusive or
                     |  term ? pred : pred                    ternary condition
                     |  pred ? pred : pred
                     |  \let id = term ; pred                 local binding
                     |  \let id = pred ; pred
                     |  \forall binders ;
                        integer-guards ==> pred               univ. integer quantification
                     |  \exists binders ;
                        integer-guards && pred                exist. integer quantification
                     |  \forall binders ;
                        iterator-guard ==> pred               univ. iterator quantification
                     |  \exists binders ;
                        iterator-guard && pred                exist. iterator quantification
                     |  \forall binders ; pred                univ. quantification
                     |  \exists binders ; pred                exist. quantification
                     |  id : pred                             syntactic naming
                     |  string : pred                         syntactic naming

   integer-guards  ::=  interv (&& interv)*

           interv  ::=  (term integer-guard-op)+
                        id
                        (integer-guard-op term)+

 integer-guard-op  ::=  <=  |  <

   iterator-guard  ::=  id ( term , term )
```

Figure 2.2: Grammar of predicates

binders	::=	*binder* (, *binder*)*	
binder	::=	*type-expr* *variable-ident* (,*variable-ident*)*	
type-expr	::=	*logic-type-expr* | *C-type-name*	
logic-type-expr	::=	*built-in-logic-type*	
	|	*id*	type identifier
built-in-logic-type	::=	boolean | integer | real	
variable-ident	::=	*id* | `*` *variable-ident*	
	|	*variable-ident* `[]`	
	|	`(` *variable-ident* `)`	

Figure 2.3: Grammar of binders and type expressions

Iterator quantification In order to iterate over non-integer types, E-ACSL introduces a notion of *iterators* over types: standard ACSL unguarded quantifications are only allowed over a type which an iterator is attached to.

Iterators are introduced by a specific construct which attaches two sets — namely `nexts` and the `guards` — to a binary predicate over a type τ. Both sets must have the same cardinal. This construct is described by the grammar of Figure 2.4. For a type τ,

declaration	::=	`//@ iterator` *id* `(` *wildcard-param* `,` *wildcard-param* `)` `:` `nexts` *terms* `;` `guards` *predicates* `;`
wildcard-param	::=	*parameter*
	|	`_`
terms	::=	*term* (, *term*)*
predicates	::=	*predicate* (, *predicate*)*

Figure 2.4: Grammar of iterator declarations

`nexts` is a set of terms which take an argument of type τ and return a value of type τ which computes the next element in this type, while `guards` is a set of predicates which take an argument of type τ and are valid (resp. invalid) to continue (resp. stop) the iteration.

Furthermore, the guard of a quantification using an iterator must be the predicate given in the definition of the iterator. This abstract binary predicate takes two arguments of the same type. One of them must be unnamed by using a wildcard (character underscore `'_'`). The unnamed argument must be binded to the guantifier, while the other corresponds to the term from which the iteration begins.

Example 2.1 *The following example introduces binary trees and a predicate which is valid if and only if each value of a binary tree is even.*

```
struct btree {
  int val;
  struct btree *left, *right;
};
```

```
/*@ iterator access(_, struct btree *t):
  @   nexts t->left, t->right;
  @   guards \valid(t->left), \valid(t->right); */

/*@ predicate is_even(struct btree *t) =
  @   \forall struct btree *tt; access(tt, t) ==> tt->val % 2 == 0; */
```

Unguarded quantification They are only allowed over boolean and char.

2.2.1 Operators precedence

No difference with ACSL*.*

Figure 2.5 summarizes operator precedences.

class	associativity	operators
selection	left	`[···] -> .`
unary	right	`! ~ + - * & (cast) sizeof`
multiplicative	left	`* / %`
additive	left	`+ -`
shift	left	`<< >>`
comparison	left	`< <= > >=`
comparison	left	`== !=`
bitwise and	left	`&`
bitwise xor	left	`^`
bitwise or	left	`|`
bitwise implies	left	`-->`
bitwise equiv	left	`<-->`
connective and	left	`&&`
connective xor	left	`^^`
connective or	left	`||`
connective implies	right	`==>`
connective equiv	left	`<==>`
ternary connective	right	`···?···:···`
binding	left	`\forall \exists \let`
naming	right	`:`

Figure 2.5: Operator precedence

2.2.2 Semantics

No difference with ACSL*, but undefinedness and same laziness than* C*.*

More precisely, while ACSL is a 2-valued logic with only total functions, E-ACSL is a 3-valued logic with partial functions since terms and predicates may be "undefined".

In this logic, the semantics of a term denoting a C expression e is undefined if e leads to a runtime error. Consequently the semantics of any term t (resp. predicate p) containing a C expression e leading to a runtime error is undefined if e has to be evaluated in order to evaluate t (resp. p).

Example 2.2 *The semantics of all the below predicates are undefined:*

- `1/0 == 1/0`
- `f(*p)` *for any logic function* `f` *and invalid pointer* `p`

Furthermore, C-like operators `&&`, `||`, `^^` and `_ ? _ : _` are lazy like in C: their right members are evaluated only if required. Thus the amount of undefinedness is limited. Consequently, predicate `p ==> q` is also lazy since it is equivalent to `!p || q`. It is also the case for guarded quantifications since guards are conjunctions and for ternary condition since it is equivalent to a disjunction of implications.

Example 2.3 *Below, the first, second and fourth predicates are invalid while the third one is valid:*

- `\false && 1/0 == 1/0`
- `\forall integer x, -1 <= x <= 1 ==> 1/x > 0`
- `\forall integer x, 0 <= x <= 0 ==> \false ==> -1 <= 1/x <= 1`
- `\exists integer x, 1 <= x <= 0 && -1 <= 1/x <= 1`

In particular, the second one is invalid since the quantification is in fact an enumeration over a finite number of elements, it amounts to `1/-1 > 0 && 1/0 > 0 && 1/1 > 0`. *The first atomic proposition is invalid, so the rest of the conjunction (and in particular 1/0) is not evaluated. The fourth one is invalid since it is an existential quantification over an empty range.*

A contrario *the semantics of predicates below is undefined:*

- `1/0 == 1/0 && \false`
- `-1 <= 1/0 <= 1 ==> \true`
- `\exists integer x, -1 <= x <= 1 && 1/x > 0`

Furthermore, casting a term denoting a C expression e to a smaller type τ is undefined if e is not representable in τ.

Example 2.4 *Below, the first term is well-defined, while the second one is undefined.*

- `(char)127`
- `(char)128`

Handling undefinedness in tools It is the responsibility of each tool which interprets E-ACSL to ensure that an undefined term is never evaluated. For instance, they may exit with a proper error message or, if they generate C code, they may guard each generated undefined C expression in order to be sure that they are always safely used.

This behavior is consistent with both ACSL [1] and mainstream specification languages for runtime assertion checking like JML [9]. Consistency means that, if it exists and is defined, the E-ACSL predicate corresponding to a valid (resp. invalid) ACSL predicate is valid (resp. invalid). Thus it is possible to reuse tools interpreting ACSL like the Frama-C's value analysis plug-in [6] in order to interpret E-ACSL, and it is also possible to perform runtime assertion checking of E-ACSL predicates in the same way than JML predicates. Reader interested by the implications (especially issues) of such a choice may read articles of Patrice Chalin [3, 4].

2.2.3 Typing

No difference with ACSL*, but no user-defined types.*

It is not possible to define logic types introduced by the specification writer (see Section 2.6).

2.2.4 Integer arithmetic and machine integers

No difference with ACSL*.*

2.2.5 Real numbers and floating point numbers

No difference with ACSL*.*

Exact real numbers and even floating point numbers are usually difficult to implement. Thus you would not wonder if most tools do not support them (or support them partially).

Real numbers beyond rationals are currently not supported by the E-ACSL plug-in. Only rationals (in $\mathbb{Q}$) and floating point numbers are supported.

2.2.6 C arrays and pointers

No difference with ACSL*.*

Ensuring validity of memory accesses is usually difficult to implement, since it requires the implementation of a memory model. Thus you would not wonder if most tools do not support it (or support it partially).

2.2.7 Structures, Unions and Arrays in logic

No difference with ACSL*.*

Logic arrays without an explicit length are usually difficult to implement. Thus you would not wonder if most tools do not support them (or support them partially).

The \length function is currently not supported by the E-ACSL plug-in.

The comparison of unions and structures is currently not supported by the E-ACSL plug-in.

2.3 Function contracts

No difference with ACSL, *but no* terminates.

Figure 2.6 shows grammar of function contracts. This is a simplified version of ACSL one without terminates clauses. Section 2.5 explains why E-ACSL has no terminates clause.

function-contract	::=	*requires-clause** *decreases-clause*? *simple-clause** *named-behavior** *compl*
clause-kind	::=	check | admit
requires-clause	::=	*clause-kind*? requires *pred* ;
decreases-clause	::=	decreases *term* (for *id*)? ;
simple-clause	::=	*assigns-clause* | *ensures-clause* | *allocation-clause* | *abrupt-clause*
assigns-clause	::=	assigns *locations* ;
locations	::=	*location* (, *location*) * | \nothing
location	::=	*tset*
ensures-clause	::=	*clause-kind*? ensures *pred* ;
named-behavior	::=	behavior *id* : *behavior-body*
behavior-body	::=	*assumes-clause** *requires-clause** *simple-clause**
assumes-clause	::=	assumes *pred* ;
completeness-clause	::=	complete behaviors (*id* (, *id*)*)? ; | disjoint behaviors (*id* (, *id*)*)? ;

Figure 2.6: Grammar of function contracts

2.3.1 Built-in constructs \old and \result

No difference with ACSL.

Figure 2.7 summarizes grammar extension of terms with \old and \result.

term	::=	\old (*term*)	old value
	|	\result	result of a function
pred	::=	\old (*pred*)	

Figure 2.7: \old and \result in terms

2.3.2 Simple function contracts

No difference with ACSL.

\assigns *is usually difficult to implement, since it requires the implementation of a memory model. Thus you would not wonder if most tools do not support it (or support it partially).*

2.3.3 Contracts with named behaviors

No difference with ACSL.

2.3.4 Memory locations and sets of terms

No difference with ACSL*, but ranges and set comprehensions are limited in order to be finite.*

Figure 2.8 describes grammar of sets of terms. The only differences with ACSL are that both lower and upper bounds of ranges are mandatory and that the predicate inside set comprehension must be guarded and bind only one variable. In that way, each set of terms is finite and their members easily identifiable.

tset	::=	`\empty`	empty set
	|	*tset* `->` *id*	
	|	*tset* `.` *id*	
	|	`*` *tset*	
	|	`&` *tset*	
	|	*tset* `[` *tset* `]`	
	|	*term* `..` *term*	range
	|	`\union` `(` *tset* (`,` *tset*)* `)`	union of locations
	|	`\inter` `(` *tset* (`,` *tset*)* `)`	intersection
	|	*tset* `+` *tset*	
	|	`(` *tset* `)`	
	|	`{` *tset* `|` *binders* (`;` *pred*)? `}`	set comprehension
	|	`{` (*tset* (`,` *tset*)*)? `}`	
	|	*term*	implicit singleton
pred	::=	`\subset` `(` *tset* `,` *tset* `)`	set inclusion
	|	*term* `\in` *tset*	set membership

Figure 2.8: Grammar for sets of terms

Example 2.5 *The set* `{ x | integer x; 0 <= x <= 9 || 20 <= x <= 29 }` *denotes the set of all integers between 0 and 9 and between 20 and 29.*

Ranges are currently only supported in memory built-ins described in Section 2.7.1, 2.13 and 2.14.

Example 2.6 *The predicate* `\valid(&t[0 .. 9])` *is supported and denotes that the ten first cells of the array* `t` *are valid. Writing the term* `&t[0 .. 9]` *alone, outside any memory built-in, is not yet supported.*

2.3.5 Default contracts, multiple contracts

No difference with ACSL.

2.4 Statement annotations

2.4.1 Assertions

No difference with ACSL.

Figure 2.9 summarizes grammar for assertions.

```
C-compound-statement ::= { declaration* statement* assertion+ }
         C-statement ::= assertion statement
      assertion-kind ::= assert | clause-kind
           assertion ::= /*@ assertion-kind pred ; */
                      |  /*@ for id (, id)* : assertion-kind pred ; */
```

Figure 2.9: Grammar for assertions

2.4.2 Loop annotations

No difference with ACSL*, but loop invariants lose their inductive nature.*

Figure 2.10 shows grammar for loop annotations. There is no syntactic difference with ACSL.

```
     statement ::= /*@ loop-annot */
                   while ( C-expression ) C-statement
                |  /*@ loop-annot */
                   for
                   ( C-expression ; C-expression ; C-expression )
                   statement
                |  /*@ loop-annot */
                   do C-statement
                   while ( C-expression ) ;

    loop-annot ::= loop-clause*
                   loop-behavior*
                   loop-variant?

   loop-clause ::= loop-invariant
                |  loop-assigns

loop-invariant ::= clause-kind? loop invariant pred ;

  loop-assigns ::= loop assigns locations ;

 loop-behavior ::= for id (, id)* :
                   loop-clause*                    annotation for behavior id

  loop-variant ::= loop variant term ;
                |  loop variant term for id ;      variant for relation id
```

Figure 2.10: Grammar for loop annotations

loop assigns *is usually difficult to implement, since it requires the implementation of a memory model. Thus you would not wonder if most tools do not support it (or support it partially).*

Loop invariants

The semantics of loop invariants is the same than the one defined in ACSL, except that they are not inductive. More precisely, if one does not take care of side effects (semantics of specifications about side effects in loop is the same in E-ACSL than the one in ACSL), a loop invariant I is valid in ACSL if and only if:

- I holds before entering the loop; and
- if I is assumed true in some state where the loop condition c is also true, and if execution of the loop body in that state ends normally at the end of the body or with a "continue" statement, I is true in the resulting state.

In E-ACSL, the same loop invariant I is valid if and only if:

- I holds before entering the loop; and
- if execution of the loop body in that state ends normally at the end of the body or with a "continue" statement, I is true in the resulting state.

Thus the only difference with ACSL is that E-ACSL does not assume that the invariant previously holds when one checks that it holds at the end of the loop body. In other words a loop invariant `I` is equivalent to put an assertion `I` just before entering the loop and at the very end of the loop body.

Example 2.7 *In the following,* `bsearch(t,n,v)` *searches for element* `v` *in array* `t` *between indices* `0` *and* `n-1`*.*

```
/*@ requires n >= 0 && \valid(t+(0..n-1));
  @ assigns \nothing;
  @ ensures -1 <= \result <= n-1;
  @ behavior success:
  @   ensures \result >= 0 ==> t[\result] == v;
  @ behavior failure:
  @   assumes t_is_sorted : \forall integer k1, int k2;
  @         0 <= k1 <= k2 <= n-1 ==> t[k1] <= t[k2];
  @   ensures \result == -1 ==>
  @       \forall integer k; 0 <= k < n ==> t[k] != v;
  @*/
int bsearch(double t[], int n, double v) {
  int l = 0, u = n-1;
  /*@ loop invariant 0 <= l && u <= n-1;
    @ for failure: loop invariant
    @   \forall integer k; 0 <= k < n ==> t[k] == v ==> l <= k <= u;
    @*/
  while (l <= u ) {
    int m = l + (u-l)/2; // better than (l+u)/2
    if (t[m] < v) l = m + 1;
    else if (t[m] > v) u = m - 1;
    else return m;
  }
  return -1;
}
```

In E-ACSL*, this annotated function is equivalent to the following one since loop invariants are not inductive.*

```
assertion ::= /*@ clause-kind? invariant pred ; */
            | /*@ for id (, id)* : clause-kind? invariant pred ; */
```

Figure 2.11: Grammar for general inductive invariants

```
/*@ requires n >= 0 && \valid(t+(0..n-1));
  @ assigns \nothing;
  @ ensures -1 <= \result <= n-1;
  @ behavior success:
  @   ensures \result >= 0 ==> t[\result] == v;
  @ behavior failure:
  @   assumes t_is_sorted : \forall integer k1, int k2;
  @         0 <= k1 <= k2 <= n-1 ==> t[k1] <= t[k2];
  @   ensures \result == -1 ==>
  @       \forall integer k; 0 <= k < n ==> t[k] != v;
  @*/
int bsearch(double t[], int n, double v) {
  int l = 0, u = n-1;
  /*@ assert 0 <= l && u <= n-1;
    @ for failure: assert
    @   \forall integer k; 0 <= k < n ==> t[k] == v ==> l <= k <= u;
    @*/
  while (l <= u ) {
    int m = l + (u-l)/2; // better than (l+u)/2
    if (t[m] < v) l = m + 1;
    else if (t[m] > v) u = m - 1;
    else return m;
    /*@ assert 0 <= l && u <= n-1;
      @ for failure: assert
      @   \forall integer k; 0 <= k < n ==> t[k] == v ==> l <= k <= u;
      @*/ ;
  }
  return -1;
}
```

General inductive invariant

Syntax of these kinds of invariant is shown Figure 2.11

In E-ACSL, these kinds of invariants put everywhere in a loop body is exactly equivalent to an assertion.

2.4.3 Built-in construct \at

No difference with ACSL, *but no forward references.*

The construct `\at(t,id)` (where `id` is a regular C label, a label added within a ghost statement or a default logic label) follows the same rule than its ACSL counterpart, except that a more restrictive scoping rule must be respected in addition to the standard ACSL scoping rule: when evaluating `\at(t,id)` at a propram point p, the program point p' denoted by `id` must be executed after p the program execution flow.

Example 2.8 *In the following example, both assertions are accepted and valid in* ACSL, *but only the first one is accepted and valid in* E-ACSL *since evaluating the term* `\at(*(p+\at(*q,Here)),L1)` *at* `L2` *requires to evaluate the term* `\at(*q,Here)` *at* `L1`*: that is forbidden since* `L1` *is executed before* `L2`*.*

```
/*@ requires \valid(p+(0..1));
  @ requires \valid(q);
  @*/
void f(int *p, int *q) {
  *p = 0;
  *(p+1) = 1;
  *q = 0;
  L1: *p = 2;
  *(p+1) = 3;
  *q = 1;
  L2:
  /*@ assert (\at(*(p+\at(*q,L1)),Here) == 2); */
  /*@ assert (\at(*(p+\at(*q,Here)),L1) == 1); */
  return ;
}
```

For the time being, `\at` can be applied to any term or predicate that uses quantified variables, let-binded variables and C variables.

Example 2.9 *The* `\at` *construct of the following example is supported.*

```
main(void) {
  int m = 2;
  int n = 7;;
  K: ;
  n = 875;
  /*@ assert
      \let k = 3;
      \exists integer u; 9 <= u < 21 &&
      \forall integer v; -5 < v <= (u < 15 ? u + 6 : k) ==>
        \at(n + u + v > 0, K); */ ;
  return 0;
}
```

However, quantified variables that use C variables in their bounds and let-binded variables that use C variables in their definition are not yet supported.

Example 2.10 *The* `\at` *construct of the following example is* not yet *supported since the quantified variable* `i` *uses the C variable* `n` *in the definition of its upper bound.*

```
/*@ ensures \forall int i; 0 <= i < n-1 ==> \old(t[i]) == t[i+1]; */
void reverse(int *t, int n) { }
```

2.4.4 Statement contracts

No difference with ACSL.

Figure 2.12 shows grammar of statement contracts.

statement	::=	`/*@` *statement-contract* `*/` *statement*
statement-contract	::=	(`for` *id* (`,` *id*)* `:`)? *requires-clause** *simple-clause-stmt** *named-behavior-stmt** *completeness-clause**
simple-clause-stmt	::=	*simple-clause* | *abrupt-clause-stmt*
named-behavior-stmt	::=	`behavior` *id* `:` *behavior-body-stmt*
behavior-body-stmt	::=	*assumes-clause** *requires-clause** *simple-clause-stmt**

Figure 2.12: Grammar for statement contracts

2.5 Termination

No difference with ACSL*, but no* `terminates` *clauses.*

2.5.1 Integer measures

No difference with ACSL*.*

2.5.2 General measures

No difference with ACSL*.*

2.5.3 Recursive function calls

No difference with ACSL*.*

2.5.4 Non-terminating functions

No such feature in E-ACSL*, since it is still experimental in* ACSL*.*

2.6 Logic specifications

Limited to stable and computable features.

Figure 2.13 presents grammar of logic definitions. This is the same than the one of ACSL without polymorphic definitions, lemmas, nor axiomatics.

2.6.1 Predicate and function definitions

No difference with ACSL*.*

C-global-decl	::=	`/*@` *logic-def*$^+$ `*/`
logic-def	::=	*logic-const-def*
	|	*logic-function-def*
	|	*logic-predicate-def*
type-expr	::=	*id*
logic-const-def	::=	logic *type-expr* *id* = *term* ;
logic-function-def	::=	logic *type-expr* *id* *parameters* = *term* ;
logic-predicate-def	::=	predicate *id* *parameters*$^?$ = *pred* ;
parameters	::=	(*parameter* (, *parameter*)*)
parameter	::=	*type-expr* *id*

Figure 2.13: Grammar for global logic definitions

2.6.2 Lemmas

No such feature in E-ACSL*: lemmas are user-given propositions. They are written usually to help theorem provers to establish validity of specifications. Thus they are mostly useful for verification activities based on deductive methods which are out of the scope of* E-ACSL*. Furthermore, they often requires human help to be proven, although* E-ACSL *targets are automatic tools.*

2.6.3 Inductive predicates

No such feature in E-ACSL*: inductive predicates are not computable if they really use their inductive nature.*

2.6.4 Axiomatic definitions

No such feature in E-ACSL*: by nature, an axiomatic is not computable.*

2.6.5 Polymorphic logic types

No such feature in E-ACSL*, since it is still experimental in* ACSL*.*

2.6.6 Recursive logic definitions

No difference with ACSL*.*

2.6.7 Higher-order logic constructions

No such feature in E-ACSL*, since it is still experimental in* ACSL*.*

2.6.8 Concrete logic types

No such feature in E-ACSL*, since it is still experimental in* ACSL*.*

2.6.9 Hybrid functions and predicates

No difference with ACSL.

Hybrid functions and predicates are usually difficult to implement, since they require the implementation of a memory model (or at least to support \at*). Thus you would not wonder if most tools do not support them (or support them partially).*

2.6.10 Memory footprint specification: reads clause

No such feature in E-ACSL*, since it is still experimental in* ACSL.

2.6.11 Specification Modules

No difference with ACSL.

2.7 Pointers and physical adressing

No difference with ACSL*, but separation.*

Figure 2.14 shows the additional constructs for terms and predicates which are related to memory location.

term	::=	\null
	|	\base_addr *one-label*? (*term*)
	|	\block_length *one-label*? (*term*)
	|	\offset *one-label*? (*term*)
	|	\allocation *one-label*? (*term*)
pred	::=	\allocable *one-label*? (*term*)
	|	\freeable *one-label*? (*term*)
	|	\fresh *two-labels*? (*term*, *term*)
	|	\valid *one-label*? (*location-address*)
	|	\valid_read *one-label*? (*location-address*)
	|	\separated (*location-address* , *location-addresses*)
one-label	::=	{ *id* }
two-labels	::=	{ *id*, *id* }
location-addresses	::=	*location-address* (, *location-address*)*
location-address	::=	*tset*

Figure 2.14: Grammar extension of terms and predicates about memory

2.7.1 Memory blocks and pointer dereferencing

No difference with ACSL.

\base_addr*,* \block_length*,* \valid*,* \valid_read *and* \offset *are usually difficult to implement, since they require the implementation of a memory model. Thus you would not wonder if most tools do not support them (or support them partially).*

term	::=	`[| |]`	empty list
	|	`[|` *term* `(,` *term*`)*` `|]`	list of elements
	|	*term* `^` *term*	list concatenation (overloading bitwise-xor operator)
	|	*term* `*^` *term*	list repetition

Figure 2.15: Notations for built-in list datatype

2.7.2 Separation

No difference with ACSL.

`\separated` *is usually difficult to implement, since it requires the implementation of a memory model. Thus you would not wonder if most tools do not support it (or support it partially).*

2.7.3 Dynamic allocation and deallocation

All these constructs are usually difficult to implement, since they require the implementation of a memory model. Thus you would not wonder if most tools do not support them (or support them partially). No difference with ACSL.

2.8 Sets and lists

2.8.1 Finite sets

No difference with ACSL.

2.8.2 Finite lists

No difference with ACSL.

Figure 2.15 shows the notations for built-in lists.

2.9 Abrupt termination

No difference with ACSL.

Figure 2.16 shows grammar of abrupt termination.

2.10 Dependencies information

No such feature in E-ACSL, *since it is still experimental in* ACSL.

2.11 Data invariants

No difference with ACSL.

Figure 2.17 summarizes grammar for declarations of data invariants.

abrupt-clause	::=	*exits-clause*
exits-clause	::=	`exits` *pred* `;`
abrupt-clause-stmt	::=	*breaks-clause* | *continues-clause* | *returns-clause*
breaks-clause	::=	breaks *pred* ;
continues-clause	::=	continues *pred* ;
returns-clause	::=	returns *pred* ;
term	::=	\exit_status

Figure 2.16: Grammar of contracts about abrupt terminations

declaration	::=	`/*@` *data-inv-decl* `*/`
data-inv-decl	::=	*data-invariant* | *type-invariant*
data-invariant	::=	*inv-strength*? global invariant *id* : *pred* ;
type-invariant	::=	*inv-strength*? type invariant *id* (*C-type-name* *id*) = *pred* ;
inv-strength	::=	`weak` | strong

Figure 2.17: Grammar for declarations of data invariants

2.11.1 Semantics

No difference with ACSL.

2.11.2 Model variables and model fields

No difference with ACSL.

Figure 2.18 summarizes grammar for declarations of model variables and fields.

declaration	::=	*C-declaration*	
	|	`/*@` model *parameter* `;` `*/`	model variable
	|	`/*@` model *C-type-name* `{` *parameter* `;`? `}` `;` `*/`	model field

Figure 2.18: Grammar for declarations of model variables and fields

2.12 Ghost variables and statements

No difference with ACSL, *but no specific construct for volatile variables.*

Figure 2.19 summarizes grammar for ghost statements which is the same than the one of ACSL.

C-type-qualifier	::=	*C-type-qualifier*	
	|	`\ghost`	only in ghost
ghost-type-specifier	::=	*C-type-specifier*	
	|	*logic-type*	
declaration	::=	*C-declaration*	
	|	`/*@ ghost` *ghost-declaration* `*/`	
direct-declarator	::=	*C-direct-declarator*	
	|	*direct-declarator* `(` *C-parameter-type-list*? `)` `/*@ ghost` `(` *ghost-parameter-list* `)` `*/`	ghost args
postfix-expression	::=	*C-postfix-expression*	
	|	*postfix-expression* `(` *C-argument-expression-list*? `)` `/*@ ghost` `(` *ghost-argument-expression-list* `)` `*/`	call with ghosts
statement	::=	*C-statement*	
	|	*statements-ghost*	
statements-ghost	::=	`/*@ ghost` *ghost-statement*+ `*/`	
ghost-selection-statement	::=	*C-selection-statement*	
	|	`if` `(` *C-expression* `)` *statement* `/*@ ghost else` *ghost-statement*+ `*/`	
struct-declaration	::=	*C-struct-declaration*	
	|	`/*@ ghost` *struct-declaration* `*/`	ghost field

Figure 2.19: Grammar for ghost statements

2.12.1 Volatile variables

No such feature in E-ACSL, *since it is still experimental in* ACSL.

2.13 Initialization and undefined values

No difference with ACSL.

\initialized *is usually difficult to implement, since it requires the implementation of a memory model. Thus you would not wonder if most tools do not support it (or support it partially).*

2.14 Dangling pointers

No difference with ACSL.

\dangling *is usually difficult to implement, since it requires the implementation of a memory model. Thus you would not wonder if most tools do not support it (or support it partially).*

2.15 Well-typed pointers

No such feature in E-ACSL, *since it is still experimental in* ACSL.

2.16 Logic attribute annotations

No such feature in E-ACSL, *since it is still experimental in* ACSL.

2.17 Preprocessing for ACSL

No difference with ACSL.

Chapter 3

Libraries

Disclaimer: this chapter is empty on purpose. It is left here to be consistent with the ACSL reference manual [1].

Chapter 4

Conclusion

This document presents an Executable ANSI/ISO C Specification Language. It provides a subset of ACSL [1] implemented [2] in the Frama-C platform [5] in which each construct may be evaluated at runtime. The specification language described here is intented to evolve in the future in two directions. First it is based on ACSL which is itself still evolving. Second the considered subset of ACSL may also change.

Appendix A

Appendices

A.1 Changes

Version 1.16

- Update according to ACSL 1.16
  - **Section 2.3:** add the `check` and `admit` clause kinds.
  - **Section 2.4.1:** add the `check` and `admit` clause kinds.
  - **Section 2.4.2:** add the `check` and `admit` clause kinds.
  - **Section 2.4.2:** add the `check` and `admit` clause kinds.

Version 1.15

- Update according to ACSL 1.15:
  - **Section 2.12:** add the \ghost qualifier.

Version 1.14

- Update according to ACSL 1.14:
  - **Section 2.4.1:** add the keyword `check`.

Version 1.13

- Update according to ACSL 1.13:
  - **Section 2.3.4:** add syntax for set membership.

Version 1.12

- Update according to ACSL 1.12:
  - **Section 2.3.4:** add subsections for build-in lists.
  - **Section 2.4.4:** fix syntax rule for statement contracts in allowing completeness clauses.
  - **Section 2.7.1:** add syntax for defining a set by giving explicitly its element.
  - **Section 2.15:** new section.

Version 1.9

- **Section 2.7.3:** new section.
- Update according to ACSL 1.9.

Version 1.8

- **Section 2.3.4:** fix example 2.5.
- **Section 2.7:** add grammar of memory-related terms and predicates.

Version 1.7

- Update according to ACSL 1.7.
- **Section 2.7.2:** no more absent.

Version 1.5-4

- Fix typos.
- **Section 2.2:** fix syntax of guards in iterators.
- **Section 2.2.2:** fix definition of undefined terms and predicates.
- **Section 2.2.3:** no user-defined types.
- **Section 2.3.1:** no more implementation issue for \old.
- **Section 2.4.3:** more restrictive scoping rule for label references in \at.

Version 1.5-3

- Fix various typos.
- Warn about features known to be difficult to implement.
- **Section 2.2:** fix semantics of ternary operator.
- **Section 2.2:** fix semantics of cast operator.
- **Section 2.2:** improve syntax of iterator quantifications.
- **Section 2.2.2:** improve and fix example 2.3.
- **Section 2.4.2:** improve explanations about loop invariants.
- **Section 2.6.9:** add hybrid functions and predicates.

Version 1.5-2

- **Section 2.2:** remove laziness of operator <==>.
- **Section 2.2:** restrict guarded quantifications to integer.
- **Section 2.2:** add iterator quantifications.
- **Section 2.2:** extend unguarded quantifications to char.
- **Section 2.3.4:** extend syntax of set comprehensions.
- **Section 2.4.2:** simplify explanations for loop invariants and add example..

Version 1.5-1

- Fix many typos.
- Highlight constructs with semantic changes in grammars.
- Explain why unsupported features have been removed.
- Indicate that experimental ACSL features are unsupported.
- Add operations over memory like `\valid`.
- **Section 2.2:** lazy operators `&&`, `||`, `^^`, `==>` and `<==>`.
- **Section 2.2:** allow unguarded quantification over boolean.
- **Section 2.2:** revise syntax of `\exists`.
- **Section 2.2.2:** better semantics for undefinedness.
- **Section 2.3.4:** revise syntax of set comprehensions.
- **Section 2.4.2:** add loop invariants, but they lose their inductive ACSL nature.
- **Section 2.5.2:** add general measures for termination.
- **Section 2.6.11:** add specification modules.

Version 1.5-0

- Initial version.

Bibliography

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

[2] Patrick Baudin, Pascal Cuoq, Jean-Christophe Filliâtre, Claude Marché, Benjamin Monate, Yannick Moy, and Virgile Prevosto. *ACSL, Implementation in Frama-C.* https://frama-c.com/download/frama-c-acsl-implementation.pdf.

[3] Patrice Chalin. Reassessing JML's logical foundation. In *Proceedings of the 7th Workshop on Formal Techniques for Java-like Programs (FTfJP'05)*, Glasgow, Scotland, July 2005.

[4] Patrice Chalin. A sound assertion semantics for the dependable systems evolution verifying compiler. In *Proceedings of the International Conference on Software Engineering (ICSE'07)*, pages 23–33, Los Alamitos, CA, USA, 2007. IEEE Computer Society.

[5] Loïc Correnson, Pascal Cuoq, Florent Kirchner, André Maroneze, Virgile Prevosto, Armand Puccetti, Julien Signoles, and Boris Yakobowski. *Frama-C User Manual.* https://frama-c.com/download/frama-c-user-manual.pdf.

[6] Pascal Cuoq, Boris Yakobowski, Matthieu Lemerre, André Maroneze, Valentin Perelle, and Virgile Prevosto. *Frama-C's Evolved Value Analysis analysis plug-in.* https://frama-c.com/fc-plugins/eva.html.

[7] International Organization for Standardization (ISO). *The ANSI C standard (C99).* http://www.open-std.org/JTC1/SC22/WG14/www/docs/n1124.pdf.

[8] Brian Kernighan and Dennis Ritchie. *The C Programming Language (2nd Ed.).* Prentice-Hall, 1988.

[9] Gary T. Leavens, K. Rustan M. Leino, Erik Poll, Clyde Ruby, and Bart Jacobs. JML: notations and tools supporting detailed design in Java. In *OOPSLA 2000 Companion, Minneapolis, Minnesota*, pages 105–106, 2000.

List of Figures

Index