Software Analyzers

E-ACSL
Executable ANSI/ISO C Specification Language

Version 1.22

Julien Signoles

Contents

Foreword

This document describes version 1.22 of the E-ACSL specification language. It is based on the ACSL specification language [2]. Features of both languages may still evolve in the future, even if we do our best to preserve backward compatibility. In particular, some features are considered *experimental*, meaning that their syntax and semantics is not yet fixed. These features are marked with Experimental.

Acknowledgements

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

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

Introduction 1

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 ACSL [2] implemented [3] in the Frama-C platform [7]. 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 [2] and the ANSI C programming language [8, 9].

1.1 Organization of this document

This document is organized in the very same way that the reference manual of ACSL [2].

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.

Specification language 2

2.1 Lexical rules

No difference with ACSL.

2.2 Logic expressions

No difference with ACSL, but the quantifications must be guarded.

More precisely, the grammars of terms and binders presented respectively Figures 2.1 and 2.3 are the same than the ones of ACSL, while Figure 2.2 presents the grammar of predicates. The only differences introduced by E-ACSL with respect to ACSL are the fact that the quantifications that must be guarded and the introduction of iterators.

Quantification

The general form of quantifications (called generalized quantifications below), as described in Fig. 2.2, is restricted to a few *finite enumerable types*: the types of bound variables must be C integer types, enum types, pointer types, or their aliases.

Generalized quantification over large types (for instance, types containing 2^{32} *elements). are unlikely evaluated efficiently at runtime.*

In addition to generalized quantifications, a restricted form of guarded quantifications described in Fig. 2.4 is also recognized *for (possibly infinite) enumerable types* (typically, `integer`). In guarded quantifications, each bound variable must be guarded exactly once and, if its bounds depend on other bound variables, these variables must be guarded earlier or guarded by the same guard. Additionnally, guards are limited to bound variables, meaning that the only allowed identifiers *id* are variable identifiers enclosed in the binder list.

Example 2.1 *The following predicates are (labeled) guarded quantifications:*

- `sorted: \forall integer i, j; 0 <= i <= j < len ==> a[i] <= a[j]`
- `is_c: \exists u8 *q; p <= q < p + len && *q == (u8)c`

Iterator quantification

For iterating over other data structures, E-ACSL introduces a notion of *iterators* over types that are introduced by a specific construct which attaches two sets — namely `nexts` and `guards` — to a binary predicate over a type τ. This construct is described by the grammar of Figure 2.5. For a type τ, `nexts` is a set of terms, and `guards` a set of predicates of the same cardinal. Each term in `nexts` is a function taking an argument of type τ and returning a value of type τ which is a successor of its argument. Each

<table>
<tr><td>literal</td><td>::=</td><td><code>\true</code> | <code>\false</code></td><td>boolean constants</td></tr>
<tr><td></td><td>|</td><td>integer</td><td>(lexical) integer constants</td></tr>
<tr><td></td><td>|</td><td>real</td><td>(lexical) real constants</td></tr>
<tr><td></td><td>|</td><td>string</td><td>(lexical) string constants</td></tr>
<tr><td></td><td>|</td><td>character</td><td>(lexical) character constants</td></tr>
<tr><td>bin-op</td><td>::=</td><td><code>+</code> | <code>-</code> | <code>*</code> | <code>/</code> | <code>%</code></td><td></td></tr>
<tr><td></td><td>|</td><td><code>==</code> | <code>!=</code> | <code><=</code> | <code>>=</code> | <code>></code> | <code><</code></td><td></td></tr>
<tr><td></td><td>|</td><td><code>&&</code> | <code>||</code> | <code>^^</code></td><td>boolean operations</td></tr>
<tr><td></td><td>|</td><td><code><<</code> | <code>>></code></td><td></td></tr>
<tr><td></td><td>|</td><td><code>&</code> | <code>|</code> | <code>--></code> | <code><--></code> | <code>^</code></td><td>bitwise operations</td></tr>
<tr><td>unary-op</td><td>::=</td><td><code>+</code> | <code>-</code></td><td>unary plus and minus</td></tr>
<tr><td></td><td>|</td><td><code>!</code></td><td>boolean negation</td></tr>
<tr><td></td><td>|</td><td><code>~</code></td><td>bitwise complementation</td></tr>
<tr><td></td><td>|</td><td><code>*</code></td><td>pointer dereferencing</td></tr>
<tr><td></td><td>|</td><td><code>&</code></td><td>address-of operator</td></tr>
<tr><td>term</td><td>::=</td><td>literal</td><td>literal constants</td></tr>
<tr><td></td><td>|</td><td>id</td><td>variables, function names</td></tr>
<tr><td></td><td>|</td><td>unary-op term</td><td></td></tr>
<tr><td></td><td>|</td><td>term bin-op term</td><td></td></tr>
<tr><td></td><td>|</td><td>term <code>[</code> term <code>]</code></td><td>array access</td></tr>
<tr><td></td><td>|</td><td><code>{</code> term</td><td></td></tr>
<tr><td></td><td></td><td><code>\with</code> <code>[</code> term <code>]</code> <code>=</code> term <code>}</code></td><td>array functional modifier</td></tr>
<tr><td></td><td>|</td><td>term <code>.</code> id</td><td>structure field access</td></tr>
<tr><td></td><td>|</td><td><code>{</code> term <code>\with</code> <code>.</code> id <code>=</code> term <code>}</code></td><td>field functional modifier</td></tr>
<tr><td></td><td>|</td><td>term <code>-></code> id</td><td></td></tr>
<tr><td></td><td>|</td><td><code>(</code> type-expr <code>)</code> term</td><td>cast</td></tr>
<tr><td></td><td>|</td><td>id <code>(</code> term (<code>,</code> term)* <code>)</code></td><td>function application</td></tr>
<tr><td></td><td>|</td><td><code>(</code> term <code>)</code></td><td>parentheses</td></tr>
<tr><td></td><td>|</td><td>term <code>?</code> term <code>:</code> term</td><td>ternary condition</td></tr>
<tr><td></td><td>|</td><td><code>\let</code> id <code>=</code> term <code>;</code> term</td><td>local binding</td></tr>
<tr><td></td><td>|</td><td><code>sizeof</code> <code>(</code> term <code>)</code></td><td></td></tr>
<tr><td></td><td>|</td><td><code>sizeof</code> <code>(</code> C-type-expr <code>)</code></td><td></td></tr>
<tr><td></td><td>|</td><td>id <code>:</code> term</td><td>syntactic naming</td></tr>
<tr><td></td><td>|</td><td>string <code>:</code> term</td><td>syntactic naming</td></tr>
<tr><td>poly-id</td><td>::=</td><td>id</td><td></td></tr>
<tr><td>ident</td><td>::=</td><td>id</td><td></td></tr>
</table>

Figure 2.1: Grammar of terms. The terminals *id*, *C-type-name*, and various literals are the same as the corresponding C lexical tokens.

```
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

```
guarded-quantif  ::=  \forall binders ; (guards ==>)+ pred
                  |   \exists binders ; guards && pred
guards           ::=  interv  (&& interv)*
interv           ::=  term  (guard-op  id)+  guard-op  term
guard-op         ::=  <=  |  <
```

Figure 2.4: Grammar of guarded quantifications.

```
iterator        ::=  \forall binders ; iterator-guard ==> pred
                 |   \exists binders ; iterator-guard && pred
iterator-guard  ::=  id ( term , term )
declaration     ::=  //@ iterator id ( wildcard-param , wildcard-param ) :
                     nexts terms ; guards predicates ;
wildcard-param  ::=  parameter
                 |   _
terms           ::=  term  (, term)*
predicates      ::=  predicate  (, predicate)*
```

Figure 2.5: Grammar of iterator declarations

predicate in the set `guards` takes an element of type τ, and is valid (resp. invalid) to indicate that the iteration should continue on the corresponding successor (resp. stop at the given argument).

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 bound to the quantifier, while the other corresponds to the term from which the iteration begins.

Example 2.2 *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; */
```

2.2.1 Operators precedence

No difference with ACSL.

Figure 2.6 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 2.6: 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 2.3 *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 2.4 *All the predicates below are well defined. The first, second and fourth predicates are invalid, whereas 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 <= 1/0 <= 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 the 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 2.5 *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, it may exit with a proper error message or, if it generates C code, it 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 [2] and mainstream specification languages for runtime assertion checking like JML [10]. 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 (e.g., Frama-C's Eva [4] or Wp [1] 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 the articles of Patrice Chalin on that topic [5, 6].

2.2.3 Typing

No difference with ACSL.

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, but no quantification over real numbers and floating point numbers.

Exact real numbers and even operations over floating point numbers are usually difficult to implement. Thus, most tools may not support them (or may support them partially).

More precisely, most real numbers are not representable at runtime with an infinite precisions. Consequently, most operations over them (e.g., equality) cannot be computed at runtime with an arbitrary precision. In such cases, it is the responsibility of each tool which interprets E-ACSL to specify the level of precision (e.g., $1e^{-6}$) up to which it is sound, and/or to emit undefinitive verdicts (e.g., “unknown”).

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, most tools may not support it (or may 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, most tools may not support them (or may support them partially).

2.3 Function contracts

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

Figure 2.7 shows the 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.

2.3.1 Pre- and Post- state

No difference with ACSL.

Figure 2.8 summarizes the grammar extension of terms with `\old` and `\result`.

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, most tools may not support it (or may support it partially).*

2.3.3 Contracts with named behaviors

No difference with ACSL.

function-contract	::=	*requires-clause** *decreases-clause*$^?$ *simple-clause** *named-behavior** *completeness-clause**
clause-kind	::=	`check` | `admit`
requires-clause	::=	*clause-kind*$^?$ `requires` *pred* `;`
decreases-clause	::=	`decreases` *term* (`for` *ident*)$^?$ `;`
simple-clause	::=	*assigns-clause* | *ensures-clause*
	|	*allocation-clause* | *abrupt-clause*
assigns-clause	::=	`assigns` *locations* `;`
locations	::=	*locations-list* | `\nothing`
locations-list	::=	*location* (`,` *location*) *
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 2.7: Grammar of function contracts

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

Figure 2.8: `\old` and `\result` in terms

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 2.9 describes the grammar of sets of terms. There are two differences with ACSL:

– ranges necessarily have lower and upper bounds;
– a guard for each binder is required when defining set comprehension. The requested constraints for guards are the very same than the ones for quantifications.

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

Figure 2.9: Grammar for sets of terms

Example 2.6 *The set* `{ x | integer x; 0 <= x <= 10 && x % 2 == 0}` *denotes the set of even integers between 0 and 10.*

2.3.5 Default contracts, multiple contracts

No difference with ACSL.

2.4 Statement annotations

2.4.1 Assertions

No difference with ACSL.

Figure 2.10 summarizes the grammar for assertions.

2.4.2 Loop annotations

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

Figure 2.11 shows the grammar for loop annotations. There is no syntactic difference with ACSL.

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

Figure 2.10: Grammar for assertions

statement	::=	`/*@` *loop-annot* `*/` *C-iteration-statement*	
loop-annot	::=	*loop-clause** *loop-behavior** *loop-variant*?	
loop-clause	::=	*loop-invariant* | *loop-assigns*	
	|	*loop-allocation*	
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 2.11: Grammar for loop annotations

`loop allocation` *and* `loop assigns` *are usually difficult to implement, since they require the implementation of a memory model. Thus, most tools may not support them (or may support them 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 (the 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 the 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 the 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 putting 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.

```
/*@ 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

The syntax of this kind of invariant is shown Figure 2.12.

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

Figure 2.12: Grammar for general inductive invariants

In E-ACSL, a general inductive invariant may be written everywhere in a loop body, and is exactly equivalent to writing 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 reached before p in the program execution flow; and

– when evaluating `\at(t,id)`, for each C left-value x that contributes to the definition of a (non-ghost) logic variable involved in t, the equality `\at(x,id) == \at(x,Here)` must hold, i.e. the value of x must not be modified between the program points `id` and `Here`.

Below, the first example illustrates the first constraint, whereas the second example illustrates the second constraint.

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 ;
}
```

Example 2.9 *In the following example, the first assertion is supported, while the second one is not supported. Indeed, in the second assertion, the guard defining the logic variable* `u` *depends on* `n` *whose value is modified between* `L1` *and* `L2`*.*

```
main(void) {
  int m = 2;
  int n = 7;;
 L1: ;
  n = 4;
 L2:
  /*@ assert
      \let k = m + 1;
      \exists integer u; 9 <= u < 21 &&
      \forall integer v; -5 < v <= (u < 15 ? u + 6 : k) ==>
        \at(n + u + v > 0, K); */ ;
  /*@ assert
      \let k = m + 1;
      \exists integer u; n <= u < 21 && // [u] depends on [n]
      \forall integer v; -5 < v <= (u < 15 ? u + 6 : k) ==>
        \at(n + u + v > 0, L1); */ ;
  return 0;
}
```

2.4.4 Statement contracts

No difference with ACSL.

Figure 2.13 shows the 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 2.13: Grammar for statement contracts

2.5 Termination

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

2.5.1 Measure

No difference with ACSL.

2.5.2 Integer measures

No difference with ACSL.

2.5.3 General measures

No difference with ACSL.

2.5.4 Recursive function calls

No difference with ACSL.

2.5.5 Non-terminating functions

No such feature in E-ACSL: whether a function is guaranteed to terminate if some predicate p holds is not a monitorable property.

2.5.6 Measures and non-terminating functions

No difference with ACSL.

2.6 Logic specifications

No difference with ACSL.

Figure 2.14 presents the grammar of logic definitions.

2.6.1 Predicate and function definitions

No difference with ACSL.

C-external-declaration	::=	`/*@` *logic-def*$^{+}$ `*/`	
logic-def	::=	*logic-const-def*	
	|	*logic-function-def*	
	|	*logic-predicate-def*	
	|	*lemma-def*	
	|	*data-inv-def*	
type-var	::=	*id*	
type-expr	::=	*type-var*	type variable
	|	*id* $<$ *type-expr* (, *type-expr*)* $>$	polymorphic type
type-var-binders	::=	$<$ *type-var* (, *type-var*)* $>$	
poly-id	::=	*id* *type-var-binders*	polymorphic object identifier
logic-const-def	::=	`logic` *type-expr* *poly-id* = *term* `;`	
logic-function-def	::=	`logic` *type-expr* *poly-id* *parameters* = *term* `;`	
logic-predicate-def	::=	`predicate` *poly-id* *parameters*$^{?}$ = *pred* `;`	
parameters	::=	`(` *parameter* (, *parameter*)* `)`	
parameter	::=	*type-expr* *id*	
lemma-def	::=	*clause-kind*$^{?}$ `lemma` *poly-id* `:` *pred* `;`	

Figure 2.14: Grammar for global logic definitions

2.6.2 Lemmas

No difference with ACSL.

Lemmas are verified before running the function `main` but after initializing global variables.

2.6.3 Inductive predicates

EXPERIMENTAL

No difference with ACSL.

Figure 2.15 presents the grammar of inductive predicates.

logic-def	::=	*inductive-def*
inductive-def	::=	`inductive` *poly-id* *parameters*$^?$ `{` *indcase** `}`
indcase	::=	`case` *poly-id* `:` *pred* `;`

Figure 2.15: Grammar for inductive predicates

Inductive predicates are usually difficult to implement, since they require a fix-point calculation, which is not viable in practice. Thus, most tools may not support them (or may support them partially). Inductively defined predicates in all their generality are thus not monitorable; however a restricted subset as described below is supported.

Notably this subset includes predicates whose constructors (*indcase* in the above grammar) have a form corresponding to definite Horn clauses (http://en.wikipedia.org/wiki/Horn_clause): for an inductively defined n-ary predicate `P` its constructors are of the form `\forall ...; h`$_1$ `==> ... ==> h`$_k$ `==> P(a`$_1$`, ..., a`$_n$`)`, with the following restrictions:

- every occurrence of `P` (apart from the conclusion `P(a`$_1$`, ..., a`$_n$`)`) is at the root of one of the hypotheses. This implies that it cannot occur in negated form. `h`$_1$,...,`h`$_k$.
- all the arguments `a`$_1$,...,`a`$_n$ of the conclusion are *simple*, i.e. they are either constants or (universally) quantified variables.
- any quantified variable occurring in one of the hypotheses `h`$_1$,...,`h`$_k$ occurs in the conclusion.

Let us call this the *simple subset* of supported predicates.

Example 2.10 (supported) *This definition belongs to the simple subset as described above.*

```
inductive gcd(integer n, integer m, integer r) {
    case gcd_zero: \forall integer x; gcd(x, 0, x);
    case gcd_S: \forall integer x, y, z;
        y != 0 ==> gcd(y, x % y, z) ==> gcd(x, y, z);
}
```

For the constructor `gcd_zero` *the chain of implications is empty, which is permissible; all of the conclusion's arguments are either quantified variables (*`x`*) or constants (*`0`*).*

The constructor `gcd_S` *has three quantified variables, which occur in the hypotheses as well as in the conclusion. The conclusion has the correct form, as all of its arguments are quantified variables. The predicate* `gcd` *occurs (positively) as the root of a hypothesis.*

Example 2.11 (unsupported) *This definition does not belong to the simple subset. The quantified variable* `c` *occurs in one of the hypotheses but does not appear in the conclusion.*

```
inductive eq(integer x, integer y) {
  case c: \forall integer a, b, c; a == c ==> b == c ==> eq(a, b);
}
```

Example 2.12 (supported) *This definition belongs to the simple subset as described above. In the constructor* `zero`*, the quantified variable does not occur in the conclusion. This poses no problem as long as it does not occur in any conclusion.*

```
inductive even(integer x) {
  case zero: \forall integer a; even(0);
  case pos: \forall integer a; a >= 2 ==> even(a-2) ==> even(a);
  case neg: \forall integer a; a <= -2 ==> even(a+2) ==> even(a);
}
```

Example 2.13 (unsupported) *This definition does not belong to the simple subset, as* `P` *occurs in a hypothesis but not at its root, as it is negated.*

```
inductive even(integer x) {
  case zero: \forall integer a; even(0);
  case pos: \forall integer a; a >= 2 ==> !even(a-1) ==> even(a);
  case neg: \forall integer a; a <= -2 ==> !even(a+1) ==> even(a);
}
```

This simple subset is extended in some important ways giving rise to the *extended subset* of supported predicates:

- there may be `\let` expressions inserted in the chain of implications.
- one of `P`'s arguments may be *complicated*, i.e. it does not need to be a constant or a quantified variable. The position of the complicated argument has to be identical for all the constructors.
- the quantified variables occurring in the formulas have to obey certain boundness conditions, such as: a quantified variable occuring in the complicated argument, needs to be bound first by a recursive occurrence of P in a hypothesis.

As these conditions (especially the boundness conditions) are too intricate to explain here, let's consider a few more examples in order to convey an intuition for which inductive definitions are supported and which are not.

Example 2.14 (supported) *This definition does not belong to the simple subset, since the second argument* `f1+f2` *of the constructor* `other`*'s conclusion is not simple. It does however belong to the extended subset as described above.*

```
inductive fibo(integer i, integer x) {
   case zero: fibo(0, 0);
   case one: fibo(1, 1);
   case other: \forall integer n, f1, f2;
       n>1 ==> fibo(n-1, f1) ==>
       \let nm2 = n-2; fibo(nm2, f2) ==> fibo(n, f1+f2);
}
```

The quantified variables `f1` *and* `f2` *occuring in the complicated argument are both bound by the two preceding hypotheses:* `fibo(n-1, f1)` *binds* `f1` *while* `fibo(nm2), f2)` *binds* `f2`.

Note also that the chain of hypotheses is interrupted by a `\let` *binding, which is permitted.*

Example 2.15 (unsupported) *This (nonsensical but correct) reformulation of the previous example is not in the subset of supported definitions.*

```
inductive fibo(integer i, integer x) {
   case zero: \forall integer a; fibo(0, a+0-a);
   case one: \forall integer a; fibo(a+1-a, 1);
   case other: \forall integer n, f1, f2;
       n+f1>1+f1 ==> fibo(n-1, f1) ==> fibo(n-2, f2) ==> fibo(n, f1+f2);
}
```

Here we observe multiple problems:

1. *In the* `zero` *constructor,* `a` *occurs in a complicated argument without having been bound by a hypothesis.*
2. *In the* `one` *constructor the first argument is complicated while it is the second argument that is complicated in the constructors* `zero` *and* `other`*.*
3. *In the* `other` *constructor,* `f1` *occurs in a hypothesis before having been bound by the hypothesis* `fibo(n-1, f1)`

2.6.4 Axiomatic definitions

EXPERIMENTAL

No difference with ACSL.

Figure 2.16 presents the grammar of axiomatic definitions.

logic-def	::=	*axiomatic-decl*	
axiomatic-decl	::=	`axiomatic` *id* `{` *logic-decl** `}`	
logic-decl	::=	*logic-def*	
	|	*logic-type-decl*	
	|	*logic-const-decl*	
	|	*logic-predicate-decl*	
	|	*logic-function-decl*	
	|	*axiom-def*	
logic-type-decl	::=	`type` *logic-type* `;`	
logic-type	::=	*id*	
	|	*id* *type-var-binders*	polymorphic type
logic-const-decl	::=	`logic` *type-expr* *poly-id* `;`	
logic-function-decl	::=	`logic` *type-expr* *poly-id* *parameters* `;`	
logic-predicate-decl	::=	`predicate` *poly-id* *parameters*$^?$ `;`	
axiom-def	::=	`axiom` *poly-id* `:` *pred* `;`	

Figure 2.16: Grammar for axiomatic declarations

Axiomatic definitions in all their generality are not monitorable. Therefore, future versions of this document will restrict them syntactically and/or through semantic criteria.

2.6.5 Polymorphic logic types

No difference with ACSL.

2.6.6 Recursive logic definitions

No difference with ACSL.

2.6.7 Higher-order logic constructions

EXPERIMENTAL

No difference with ACSL.

Figure 2.17 introduces new term constructs for higher-order logic.

term	::=	`\lambda` *binders* `;` *term*	abstraction
	|	*ext-quantifier* `(` *term* `,` *term* `,` *term* `)`	
	|	`{` *term* `\with` `[` *range* `]` `=` *term* `}`	
ext-quantifier	::=	`\max` | `\min` | `\sum`	
	|	`\product` | `\numof`	

Figure 2.17: Grammar for higher-order constructs

Abstractions are only implemented for extended quantifiers, such as the term `\sum(1, 10, \lambda integer k; k)`.

2.6.8 Concrete logic types

Experimental

No difference with ACSL.

Figure 2.18 introduces new constructs for defining logic types and the associated new terms.

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, most tools may not support them (or may support them partially).*

2.6.10 Memory footprint specification: `reads` clause

Experimental

No difference with ACSL.

Figure 2.19 introduces `reads` clauses.

read clauses are usually difficult to implement, since they require the implementation of a memory model. Thus, most tools may not support them (or may support them partially).

2.6.11 Specification Modules

No difference with ACSL.

2.7 Pointers and physical adressing

No difference with ACSL.

Figure 2.20 shows the additional constructs for terms and predicates which are related to memory location.

2.7.1 Memory blocks and pointer dereferencing

No difference with ACSL.

All memory-related built-in functions and predicates are usually difficult to implement, since they require the implementation of a memory model. Thus, most tools may not support them (or may support them partially).

<table>
<tr><td>logic-def</td><td>::=</td><td><code>type</code> logic-type = logic-type-def <code>;</code></td><td></td></tr>
<tr><td>logic-type-def</td><td>::=</td><td>record-type</td><td></td></tr>
<tr><td></td><td>|</td><td>sum-type</td><td></td></tr>
<tr><td></td><td>|</td><td>product-type</td><td></td></tr>
<tr><td></td><td>|</td><td>function-type</td><td></td></tr>
<tr><td></td><td>|</td><td>type-expr</td><td>type abbreviation</td></tr>
<tr><td>record-type</td><td>::=</td><td><code>{</code> type-expr id (<code>;</code> type-expr id)* <code>;</code>? <code>}</code></td><td></td></tr>
<tr><td>function-type</td><td>::=</td><td><code>(</code> (type-expr (<code>,</code> type-expr)*)? <code>)</code> <code>-></code> type-expr</td><td></td></tr>
<tr><td>sum-type</td><td>::=</td><td><code>|</code>? constructor (<code>|</code> constructor)*</td><td></td></tr>
<tr><td>constructor</td><td>::=</td><td>id</td><td>constant constructor</td></tr>
<tr><td></td><td>|</td><td>id <code>(</code> type-expr (<code>,</code> type-expr)* <code>)</code></td><td>non-constant constructor</td></tr>
<tr><td>product-type</td><td>::=</td><td><code>(</code> type-expr (<code>,</code> type-expr)+ <code>)</code></td><td>product type</td></tr>
<tr><td>term</td><td>::=</td><td>term <code>.</code> id</td><td>record field access</td></tr>
<tr><td></td><td>|</td><td><code>\match</code> term <code>{</code> match-cases <code>}</code></td><td>pattern-matching</td></tr>
<tr><td></td><td>|</td><td><code>(</code> term (<code>,</code> term)+ <code>)</code></td><td>tuples</td></tr>
<tr><td></td><td>|</td><td><code>{</code> (<code>.</code> id <code>=</code> term <code>;</code>)+ <code>}</code></td><td>records</td></tr>
<tr><td></td><td>|</td><td><code>\let</code> <code>(</code> id (, id)+ <code>)</code> <code>=</code> term <code>;</code> term</td><td></td></tr>
<tr><td>match-cases</td><td>::=</td><td>match-case+</td><td></td></tr>
<tr><td>match-case</td><td>::=</td><td><code>case</code> pat <code>:</code> term</td><td></td></tr>
<tr><td>pat</td><td>::=</td><td>id</td><td>constant constructor</td></tr>
<tr><td></td><td>|</td><td>id <code>(</code> pat (<code>,</code> pat)* <code>)</code></td><td>non-constant constructor</td></tr>
<tr><td></td><td>|</td><td>pat <code>|</code> pat</td><td>or pattern</td></tr>
<tr><td></td><td>|</td><td>_</td><td>any pattern</td></tr>
<tr><td></td><td>|</td><td>literal | <code>{</code> (<code>.</code> id <code>=</code> pat)* <code>}</code></td><td>record pattern</td></tr>
<tr><td></td><td>|</td><td><code>(</code> pat (<code>,</code> pat)* <code>)</code></td><td>tuple pattern</td></tr>
<tr><td></td><td>|</td><td>pat <code>as</code> id</td><td>pattern binding</td></tr>
</table>

Figure 2.18: Grammar for concrete logic types and pattern-matching

logic-function-decl	::=	`logic` *type-expr* *poly-id* *parameters* *reads-clause* `;`
logic-predicate-decl	::=	`predicate` *poly-id* *parameters*$^{?}$ *reads-clause* `;`
reads-clause	::=	`reads` *locations*
logic-function-def	::=	`logic` *type-expr* *poly-id* *parameters* *reads-clause* `=` *term* `;`
logic-predicate-def	::=	`predicate` *poly-id* *parameters*$^{?}$ *reads-clause* `=` *pred* `;`

Figure 2.19: Grammar for logic declarations with `reads` clauses

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*$^{?}$ `(` *locations-list* `)`
	|	`\valid_read` *one-label*$^{?}$ `(` *locations-list* `)`
	|	`\separated` `(` *location* `,` *locations-list* `)`
	|	`\object_pointer` *one-label*$^{?}$ `(` *locations-list* `)`
	|	`\pointer_comparable` *one-label*$^{?}$ `(` *term* `,` *term* `)`
one-label	::=	`{` *label-id* `}`
two-labels	::=	`{` *label-id*, *label-id* `}`

Figure 2.20: Grammar extension of terms and predicates about memory

2.7.2 Separation

No difference with ACSL.

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

2.7.3 Dynamic allocation and deallocation

No difference with ACSL.

All these constructs are usually difficult to implement, since they require the implementation of a memory model. Thus, most tools may not support them (or may support them partially).

Figure 2.21 introduces grammar for dynamic allocations and deallocations.

allocation-clause	::=	`allocates` *dyn-allocation-addresses* `;`
	|	`frees` *dyn-allocation-addresses* `;`
loop-allocation	::=	`loop allocates` *dyn-allocation-addresses* `;`
	|	`loop frees` *dyn-allocation-addresses* `;`
dyn-allocation-addresses	::=	*locations*

Figure 2.21: Grammar for dynamic allocations and deallocations

2.8 Sets and lists

2.8.1 Finite sets

No difference with ACSL.

2.8.2 Finite lists

No difference with ACSL.

Figure 2.22 shows the notations for built-in lists.

2.9 Abrupt termination

No difference with ACSL.

Figure 2.23 shows the grammar of abrupt terminations.

2.10 Dependencies information

Experimental

No difference with ACSL.

Figure 2.24 shows the grammar for dependencies information.

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

Figure 2.22: Notations for built-in list datatype

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

Figure 2.23: Grammar of contracts about abrupt terminations

2.11 Data invariants

No difference with ACSL.

Figure 2.25 summarizes grammar for declarations of data invariants.

strong invariants are unlikely evaluated efficiently at runtime.

2.11.1 Semantics

No difference with ACSL.

2.11.2 Model variables and model fields

No difference with ACSL.

Figure 2.26 summarizes the grammar for declarations of model variables and fields.

2.12 Ghost variables and statements

No difference with ACSL.

Figure 2.27 summarizes the grammar for ghost statements which is the same than the one of ACSL.

2.12.1 Volatile variables

Figure 2.28 summarizes the grammar for volatile constructs.

assigns-clause	::=	`assigns` *locations-list* (`\from` *locations*)$^?$ `;`
	|	`assigns` *term* `\from` *locations* `=` *term* `;`

Figure 2.24: Grammar for dependencies information

data-inv-def	::=	*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 2.25: Grammar for declarations of data invariants

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, most tools may not support it (or may 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, most tools may not support it (or may support it partially).*

2.15 Well-typed pointers

No such feature in E-ACSL: it would require the implementation of a C type system at runtime.

2.16 Preprocessing for ACSL

No difference with ACSL.

logic-def	::=	`model` *parameter* `;`	model variable
	|	`model` *C-type-name* `{` *parameter* `;`$^?$ `}` `;`	model field

Figure 2.26: Grammar for declarations of model variables and fields

C-type-qualifier	::=	`\ghost`	only in ghost
C-type-specifier	::=	*logic-type*	
logic-def	::=	`ghost` *C-declaration*	
C-direct-declarator	::=	*C-direct-declarator*	function declarator
		`(` *C-parameter-type-list*$^?$	
		`) /*@ ghost (`	
		C-parameter-type-list	with ghost params
		`) */`	
C-postfix-expression	::=	*C-postfix-expression*	function call
		`(` *C-argument-expression-list*$^?$	
		`) /*@ ghost (`	
		C-argument-expression-list	with ghost args
		`)`	
		`*/`	
C-statement	::=	`/*@ ghost`	
		C-statement$^+$	ghost code
		`*/`	
	|	`if (` *C-expression* `)`	
		statement	
		`/*@ ghost`	
		`else` *C-statement*	ghost alternative
		C-statement*	unconditional ghost code
		`*/`	
C-struct-declaration	::=	`/*@ ghost`	
		C-struct-declaration	ghost field
		`*/`	

Figure 2.27: Grammar for ghost statements

logic-def	::=	`//@ volatile` *locations* (`reads` *ident*)$^?$ (`writes` *ident*)$^?$ `;`

Figure 2.28: Grammar for volatile constructs

3 Libraries

Disclaimer: this chapter is empty on purpose. It is left here to be consistent with the ACSL reference manual [2].

Conclusion 4

This document presents an Executable ANSI/ISO C Specification Language. It provides a subset of ACSL [2] implemented [3] in the Frama-C platform [7] in which each construct may be evaluated at runtime. The specification language described here is intended 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.

Appendices

A

A.1 Changes

Version 1.22

- **Section 2.6.3:** add support for a subset of inductive definitions.

Version 1.21

- No changes: changes in ACSL 1.21 do not impact E-ACSL.

Version 1.20

- No changes: changes in ACSL 1.20 do not impact E-ACSL.

Version 1.19

- Update according to ACSL 1.19
  - **Section 2.7.1:** add the `\object_pointer` and `\pointer_comparable` built-in predicates.

Version 1.18

- No changes: changes in ACSL 1.18 do not impact E-ACSL.

Version 1.17

- **Section 2.2:** xor `^^` is not lazy.
- **Section 2.2:** new extended syntax for quantifications.
- **Section 2.2.5:** additional remark about real numbers and operations over them.
- **Section 2.3.4:** new extended syntax for set comprehensions.
- **Section 2.4.3:** more restrictive scoping rule for `\at` constructs.
- **Section 2.6:** add lemmas and data invariants.
- **Section 2.6.3:** add inductive predicates experimentally: the accepted subset will be refined in a future version.
- **Section 2.6.4:** add axiomatic declarations experimentally: the accepted subset will be refined in a future version.
- **Section 2.6.5:** add polymorphic logic types.
- **Section 2.6.7:** add higher-order logic constructions.
- **Section 2.6.8:** add concrete logic types.

- **Section 2.6.10:** add `read` clauses.
- **Section 2.10:** add dependencies information.
- **Section 2.12.1:** add volatile constructs.

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 2.6.
- **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 2.4.
- **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 2.5.3:** add general measures for termination.
- **Section 2.6.11:** add specification modules.

Version 1.5-0

- Initial version.

Bibliography

[1] Patrick Baudin, François Bobot, Loïc Correnson, Zaynah Dargaye, and Allan Blanchard. *Wp Plug-in Manual*. https://frama-c.com/fc-plugins/wp.html.

[2] 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.

[3] 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.

[4] David Bühler, Pascal Cuoq, Boris Yakobowski, Matthieu Lemerre, André Maroneze, Valentin Perelle, and Virgile Prevosto. *Eva — The Evolved Value Analysis Plug-in*. https://frama-c.com/fc-plugins/eva.html.

[5] 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.

[6] 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.

[7] 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.

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

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

[10] 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