E-ACSL Version 1.12
Implementation in Frama-C plug-in E-ACSL
version Chlorine-20180501
E-ACSL

Executable ANSI/ISO C Specification Language

Version 1.12 – Frama-C plug-in E-ACSL version Chlorine-20180501

Julien Signoles

CEA LIST, Software Reliability Laboratory

©2011-2018 CEA LIST
This work has been initially supported by the ‘Hi-Lite’ FUI project (FUI AAP 9).
## Contents

1 Introduction .......................................................... 11
   1.1 Organization of this document ................................ 11
   1.2 Generalities about Annotations ............................... 11
   1.3 Notations for grammars ....................................... 11

2 Specification language ............................................... 13
   2.1 Lexical rules .................................................... 13
   2.2 Logic expressions .............................................. 13
      2.2.1 Operators precedence .................................. 17
      2.2.2 Semantics ............................................... 17
      2.2.3 Typing ................................................... 19
      2.2.4 Integer arithmetic and machine integers ............... 19
      2.2.5 Real numbers and floating point numbers .............. 19
      2.2.6 C arrays and pointers .................................. 19
      2.2.7 Structures, Unions and Arrays in logic ............... 19
      2.2.8 String literals .......................................... 19
   2.3 Function contracts ............................................. 19
      2.3.1 Built-in constructs `\old` and `\result` ............... 19
      2.3.2 Simple function contracts ............................. 21
      2.3.3 Contracts with named behaviors ....................... 21
      2.3.4 Memory locations and sets of terms .................... 21
      2.3.5 Default contracts, multiple contracts ............... 21
   2.4 Statement annotations ......................................... 22
      2.4.1 Assertions ................................................ 22
      2.4.2 Loop annotations ....................................... 22
      2.4.3 Built-in construct `\at` ................................ 24
      2.4.4 Statement contracts .................................... 25
   2.5 Termination .................................................... 25
      2.5.1 Integer measures ...................................... 25
This is a preliminary design of the E-ACSL language, a deliverable of the task 3.4 of the FUI-9 project Hi-Lite (http://www.open-do.org/projects/hi-lite).

This is the version 1.12 of E-ACSL design based on ACSL version 1.12 [1]. Several features may still evolve in the future.

Acknowledgements

We gratefully thank all the people who contributed to this document: Patrick Baudin, Bernard Botella, Loïc Correnson, Pascal Cuoq, Johannes Kanig, David Mentré, Benjamin Monate, Yannick Moy and Virgile Prevosto.
This document is a reference manual for the E-ACSL implementation provided by the E-ACSL plug-in [10] (versionChlorine-20180501) of the FRAMA-C framework [5]. 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.

Not all of the features mentioned in this document are currently implemented in the FRAMA-C’s E-ACSL plug-in. Those who aren’t yet are signaled as in the following line:

This feature is not currently supported by FRAMA-C’s E-ACSL plug-in.\(^1\)

As a summary, Figure 1.1 synthetizes main features that are not currently implemented into the FRAMA-C’s E-ACSL plug-in.

1.2 Generalities about Annotations

No difference with ACSL.

1.3 Notations for grammars

No difference with ACSL.

\(^1\)Additional remarks on the feature may appear as footnote.
## CHAPTER 1. INTRODUCTION

<table>
<thead>
<tr>
<th>typoing</th>
<th>mathematical reals</th>
</tr>
</thead>
<tbody>
<tr>
<td>terms</td>
<td><code>true</code> and <code>false</code></td>
</tr>
<tr>
<td></td>
<td>bitwise operators</td>
</tr>
<tr>
<td></td>
<td>let binding</td>
</tr>
<tr>
<td></td>
<td><code>t</code>-sets</td>
</tr>
<tr>
<td>predicates</td>
<td>exclusive or operator</td>
</tr>
<tr>
<td></td>
<td>let bindings</td>
</tr>
<tr>
<td></td>
<td>quantifications over non-integer types</td>
</tr>
<tr>
<td></td>
<td><code>\text{\textit{separated}}</code></td>
</tr>
<tr>
<td></td>
<td><code>\text{\textit{specified}}</code></td>
</tr>
<tr>
<td>annotations</td>
<td>behavior-specific annotations</td>
</tr>
<tr>
<td></td>
<td>loop assigns</td>
</tr>
<tr>
<td></td>
<td>loop variants</td>
</tr>
<tr>
<td></td>
<td>global annotations</td>
</tr>
<tr>
<td>behavior clauses</td>
<td>assigns</td>
</tr>
<tr>
<td></td>
<td>decreases</td>
</tr>
<tr>
<td></td>
<td>abrupt termination</td>
</tr>
<tr>
<td></td>
<td>complete and disjoint behaviors</td>
</tr>
</tbody>
</table>

Figure 1.1: Summary of not-yet-implemented features.
Chapter 2

Specification language

2.1 Lexical rules

No difference with ACSL.

2.2 Logic expressions

No difference with ACSL, but guarded quantification.

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.

Reals are not correctly supported by the E-ACSL plug-in right now. Only floating point
numbers are supported: real constants and operations are seen as C floating point constants
and operations.

Quantification

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

Guarded integer quantification Guarded universal quantification is denoted by
\[ \forall \tau x_1, \ldots, x_n ; \]
\[ a_1 \leq x_1 \leq b_1 \ldots \& \& a_n \leq x_n \leq b_n \]

and guarded existential quantification by
\[ \exists \tau x_1, \ldots, x_n ; \]
\[ a_1 \leq x_1 \leq b_1 \ldots \& \& a_n \leq x_n \leq b_n \]
\& \& 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.
### CHAPTER 2. SPECIFICATION LANGUAGE

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

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

boolean operations

**unary-op ::=** + | -
| ! | ~
| * | & |

unary plus and minus

boolean negation

bitwise complementation

pointer dereferencing

address-of operator

**term ::=** literal
| id |
| unary-op term |
| term bin-op term |
| term [ term ] |
| \{ term \with \{ term = term \} |
| term . id |
| \{ term \with . id = term \} |
| term -> id |
| ( type-expr ) term |
| id ( term (, term)* ) |
| ( term ) |
| term ? term : term |
| \let id = term ; term |
| sizeof ( term ) |
| sizeof ( C-type-name ) |
| id : term |
| string : term |

literal constants

variables

array access

array functional modifier

structure field access

field functional modifier

cast

function application

parentheses

ternary condition

local binding

syntactic naming

syntactic naming

---

**Figure 2.1: Grammar of terms**
2.2. LOGIC EXPRESSIONS

| rel-op ::= | == | != | <= | >= | > | < |
| pred ::= | \true | \false | comparisons |
| term (rel-op term)+ | predicate application |
| id ( term (, term)* ) | |
| ( 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
**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 $\tau$. Both sets must have the same cardinal. This construct is described by the grammar of Figure 2.4. For a type $\tau$, *nexts*

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

**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.

```plaintext
struct btree {
  int val;
  struct btree *left , *right;
};
//@ iterator access (_, struct btree *t):
@  nexts t->left , t->right;
```
2.2. LOGIC EXPRESSIONS

@ guards \valid(t\rightarrow\text{left}), \valid(t\rightarrow\text{right}); */

/*@ predicate is_even(struct btree *t) = */
@ \forall tt struct btree *tt; access(tt, t) \implies 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.

![Operator precedence figure]

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 \( \tau \) is undefined if \( e \) is
not representable in \( \tau \).

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].
2.3 FUNCTION CONTRACTS

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).

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).

2.2.8 String literals

*No difference with ACSL.*

2.3 Function contracts

*No difference with ACSL, but no terminates and abrupt clauses.*

Figure 2.6 shows grammar of function contracts. This is a simplified version of ACSL one without terminates and abrupt clauses. Section 2.5 (resp. 2.9) explains why E-ACSL has no terminates (resp. abrupt) clause.

2.3.1 Built-in constructs $\texttt{old}$ and $\texttt{result}$

*No difference with ACSL.*

Figure 2.7 summarizes grammar extension of terms with $\texttt{old}$ and $\texttt{result}$.
\begin{align*}
\text{function-contract} &::= \text{requires-clause}^* \\
 &\quad \text{decreases-clause}^? \text{ simple-clause}^* \\
 &\quad \text{named-behavior}^* \text{ completeness-clause}^* \\
\text{requires-clause} &::= \text{requires} \ pred ; \\
\text{decreases-clause} &::= \text{decreases} \ term \ \text{(for id)}^? ; \\
\text{simple-clause} &::= \text{assigns-clause} \mid \text{ensures-clause} \\
\text{assigns-clause} &::= \text{assigns} \ \text{locations} ; \\
\text{locations} &::= \text{location} \ (, \ \text{location})^* \mid \text{\nothing} \\
\text{location} &::= \text{tset} \\
\text{ensures-clause} &::= \text{ensures} \ pred ; \\
\text{named-behavior} &::= \text{behavior} \ \text{id} : \ \text{behavior-body} \\
\text{behavior-body} &::= \text{assumes-clause}^* \ \text{requires-clause}^* \ \text{simple-clause}^* \\
\text{assumes-clause} &::= \text{assumes} \ pred ; \\
\text{completeness-clause} &::= \text{complete} \ \text{behaviors} \ (\text{id} \ (, \ \text{id}))^? ; \\
 &\quad \mid \text{disjoint} \ \text{behaviors} \ (\text{id} \ (, \ \text{id}))^? ;
\end{align*}

Figure 2.6: Grammar of function contracts

\begin{align*}
\text{term} &::= \text{\old} \ ( \text{term} ) \ \text{old value} \\
 &\mid \text{\result} \ \text{result of a function} \\
\text{pred} &::= \text{\old} \ ( \ \text{pred} )
\end{align*}

Figure 2.7: \text{\old} and \text{\result} in terms
2.3. FUNCTION CONTRACTS

2.3.2 Simple function contracts

*No difference with ACSL.*

\texttt{\textbackslash 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.

\[
\begin{align*}
tset & ::= \emptyset & \text{empty set} \\
    & | \quad \text{tset} \rightarrow \text{id} \\
    & | \quad \text{tset} . \text{id} \\
    & | \quad \ast \quad \text{tset} \\
    & | \quad \& \quad \text{tset} \\
    & | \quad \text{tset} \lbrack \text{tset} \rbrack \\
    & | \quad \text{term} \quad \text{,} \quad \text{term} & \text{range} \\
    & | \quad \text{\textbackslash union} \ ( \text{tset} \ (, \text{tset})^* ) & \text{union of locations} \\
    & | \quad \text{\textbackslash inter} \ ( \text{tset} \ (, \text{tset})^* ) & \text{intersection} \\
    & | \quad \text{tset} \ + \ \text{tset} \\
    & | \quad ( \ \text{tset} \ ) \\
    & | \quad \{ \ \text{tset} \mid \text{binders} \ (, \text{pred})^? \}^a & \text{set comprehension} \\
    & | \quad \{ \ (\text{tset} \ (, \text{tset})^*)^? \}^b \\
    & | \quad \text{term} & \text{implicit singleton} \\
pred & ::= \text{\textbackslash subset} \ ( \text{tset} \ , \text{tset} ) & \text{set inclusion}
\end{align*}
\]

*\text{tset}*: set of locations

*\text{pred}*: set inclusion

\[a^*\text{the given term cannot itself be a set}\]

\[b^*\text{the given terms cannot themselves be a set}\]

Figure 2.8: Grammar for sets of terms

\[\text{Example 2.5} \quad \text{The set}\ \{ \ x \mid \text{integer} \ x; \ 0 \leq x \leq 9 \ \| \ 20 \leq x \leq 29 \ \} \text{denotes the set of all integers between 0 and 9 and between 20 and 29.}\]

2.3.5 Default contracts, multiple contracts

*No difference with ACSL.*
CHAPTER 2. SPECIFICATION LANGUAGE

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 ::= /*@ assert pred ; */ |
            /*@ for id (, id)* : assert 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 ::= loop invariant pred ;

loop-assigns ::= loop assigns locations ;

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

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

Figure 2.10: Grammar for loop annotations
**2.4. STATEMENT 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 2.6** In the following, bsearch(t,n,v) searches for element $v$ in array $t$ between indices 0 and n-1.

```c
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 k; 0 <= k < n \implies t[k] == v \implies 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.
CHAPTER 2  SPECIFICATION LANGUAGE

assertion ::= /*@ invariant pred ; */
| /*@ for id (, id)* : invariant pred ; */

Figure 2.11: Grammar for general inductive invariants

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 program point p, the program point p' denoted by id must be executed after p the program execution flow.

Example 2.7 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.
2.4.4 Statement contracts

No difference with ACSL, but no abrupt clauses.

Figure 2.6 shows grammar of statement contracts. Like function contracts, this is a simplified version of ACSL with no abrupt clauses. All other constructs are unchanged.

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.

```
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

2.6.1 Predicate and function definitions

No difference with ACSL.

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 require 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 addressing

*No difference with ACSL, but separation.*

Figure 2.14 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.*

\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).
### 2.7.2 Separation

*No difference with ACSL.*

\texttt{\textbackslash separated} 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).

### 2.7.3 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).*

**Warning:** this section is still almost experimental in ACSL. Thus it might still evolve in the future.

### 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.

---

**Figure 2.14:** Grammar extension of terms and predicates about memory

| 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 |
2.9 Abrupt termination

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

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 2.16 summarizes grammar for declarations of data invariants.

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

*Figure 2.16: 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 2.17 summarizes 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 2.18 summarizes grammar for ghost statements which is the same than the one of ACSL.

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

direct-declarator ::= C-direct-declarator
  | direct-declarator
    ( C-parameter-type-list? )
    /*@ ghost
    ( ghost-parameter-list )
    */

ghost-type-specifier ::= C-type-specifier
  | logic-type

direct-declarator ::= C-direct-declarator
  | direct-declarator
    ( C-parameter-type-list? )
    /*@ ghost
    ( ghost-parameter-list )
    */

ghostselection-statement ::= C-selection-statement
  | if ( C-expression )
    statement
    /*@ ghost else
    ghost-statement
    */

struct-declaration ::= C-struct-declaration
  | /*@ ghost
  struct-declaration */

Figure 2.18: Grammar for ghost statements
2.12.1 Volatile variables

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

2.13 Undefined values, dangling pointers

No difference with ACSL.

\texttt{initialized} and \texttt{dangling} 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).

2.14 Well-typed pointers

No such feature in E-ACSL, since it is still experimental in ACSL.
Chapter 3
Libraries

Disclaimer: this chapter is yet empty. It is left here to give an idea of what the final document will look and 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 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.
Appendix A

Appendices
A.1 Changes

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 2.14: 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.
A.1. CHANGES

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 $\Leftarrow\Rightarrow$.
- **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 $\\text{valid}$.
- **Section 2.2**: lazy operators $\&\&$, $\|\|$, $\Leftrightarrow$, $\Rightarrow$ and $\Leftarrow\Rightarrow$.
- **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.

A.2 Changes in E-ACSL Implementation

Version Chlorine-20180501

- Section 2.2: support of \let binding.

Version 0.5

- Section 2.7.3: support of \freeable.

Version 0.3

- Section 2.4.2: support of loop invariant.

Version 0.2

- Section 2.2: support of bitwise complementation.
- Section 2.7.1: support of \valid.
- Section 2.7.1: support of \block_length.
- Section 2.7.1: support of \base_addr.
- Section 2.7.1: support of \offset.
- Section 2.13: support of \initialized.

Version 0.1

- Initial version.


List of Figures

1.1 Summary of not-yet-implemented features ........................................ 12

2.1 Grammar of terms ............................................................... 14
2.2 Grammar of predicates ......................................................... 15
2.3 Grammar of binders and type expressions ..................................... 16
2.4 Grammar of iterator declarations ............................................. 16
2.5 Operator precedence ............................................................ 17
2.6 Grammar of function contracts ............................................... 20
2.7 \old{} and \result{} in terms .................................................... 20
2.8 Grammar for sets of terms ...................................................... 21
2.9 Grammar for assertions ......................................................... 22
2.10 Grammar for loop annotations ............................................... 22
2.11 Grammar for general inductive invariants .................................. 24
2.12 Grammar for statement contracts ........................................... 25
2.13 Grammar for global logic definitions ....................................... 26
2.14 Grammar extension of terms and predicates about memory ............... 28
2.15 Notations for built-in list datatype .......................................... 29
2.16 Grammar for declarations of data invariants ................................ 29
2.17 Grammar for declarations of model variables and fields ................... 30
2.18 Grammar for ghost statements ............................................... 30
Index

? , 12, 13
_, 14
\allocable , 26
\allocation , 26
annotation , 20
assert , 20
assigns , 18, 20
assumes , 18
\at , 22
\base_addr , 26
behavior , 19
behavior , 18, 23
behaviors , 18
\block_length , 26
boolean , 14
complete , 18
codercontract , 17, 23
data invariant , 27
decreases , 18
\decreases , 23
disjoint , 18
do , 20
doelse , 28
\empty , 19
ensures , 18
\exists , 13
\false , 12, 13
for , 18, 20, 22, 23
\forall , 13
\freeable , 26
\fresh , 26
function behavior , 19
function contract , 17
ghost , 28
ghost , 28
global , 27
global invariant , 27
grammar entries
 C-compound-statement , 20
 C-global-decl , 24
 C-statement , 20
assertion , 20, 22
assigns-clause , 18
assumes-clause , 18
behavior-body-stmt , 23
behavior-body , 18
bin-op , 12
binders , 14
binder , 14
built-in-logic-type , 14
completeness-clause , 18
data-inv-decl , 27
data-invariant , 27
declaration , 14, 27, 28
decreases-clause , 18
direct-declarator , 28
ensures-clause , 18
function-contract , 18
ghost-contract , 18
ghost-type-specifier , 28
integer-guard-op , 13
integer-guards , 13
interv , 13
inv-strength , 27
iterator-guard , 13
literal , 12
location-addresses , 26
location-address , 26
locations , 18
location , 18
logic-const-def , 24
logic-def , 24
logic-function-def , 24
logic-predicate-def , 24
logic-type-expr , 14
loop-annot , 20
loop-assigns , 20
loop-behavior, 20
loop-clause, 20
loop-invariant, 20
loop-variant, 20
named-behavior-stmt, 23
named-behavior, 18
one-label, 26
parameters, 24
parameter, 24
postfix-expression, 28
predicates, 14
pred, 13, 18, 19, 26
rel-op, 13
requires-clause, 18
simple-clause, 18
statement-contract, 23
statements-ghost, 28
statement, 20, 23, 28
struct-declaration, 28
terms, 14
term, 12, 18, 26, 27
tset, 19
two-labels, 26
type-expr, 14, 24
type-invariant, 27
unary-op, 12
variable-ident, 14
wildcard-param, 14
guards, 14
hybrid
  function, 25
  predicate, 25
if, 28
integer, 14
\int, 19
invariant, 21
  data, 27
  global, 27
  type, 27
invariant, 20, 22, 27
iterator, 14
\let, 12, 13
location, 26
logic, 24
logic specification, 24
loop, 20
model, 27

model, 28
nexts, 14
\nothing, 18
\null, 26
\offset, 26
\old, 18
predicate, 24
real, 14
recursion, 25
requires, 18
\result, 18
\separated, 26
sizeof, 12
specification, 24
statement contract, 23
strong, 27
\subset, 19
termination, 23
\true, 12, 13
type, 27
type invariant, 27
\union, 19
\valid, 26
\valid_read, 26
variant, 20
\variant, 23
weak, 27
while, 20
\with, 12