E-ACSL
Executable ANSI/ISO C Specification Language
Version 1.5-3
E-ACSL
Executable ANSI/ISO C Specification Language
Version 1.5-3

Julien Signoles
CEA LIST, Software Reliability Laboratory

©2011 CEA LIST
This work has been 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 .................................................. 18
    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 .............................. 19
    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 ......................................... 21
    2.4.1 Assertions ............................................. 21
    2.4.2 Loop annotations ...................................... 22
    2.4.3 Built-in construct \at ................................ 24
    2.4.4 Statement contracts ................................... 24
  2.5 Termination ................................................... 25
    2.5.1 Integer measures ..................................... 25
CONTENTS

2.5.2 General measures ................................. 25
2.5.3 Recursive function calls .......................... 25
2.5.4 Non-terminating functions ....................... 25
2.6 Logic specifications ................................. 25
  2.6.1 Predicate and function definitions ............... 25
  2.6.2 Lemmas ......................................... 26
  2.6.3 Inductive predicates ............................ 26
  2.6.4 Axiomatic definitions ........................... 26
  2.6.5 Polymorphic logic types ......................... 26
  2.6.6 Recursive logic definitions ..................... 26
  2.6.7 Higher-order logic constructions ............... 26
  2.6.8 Concrete logic types ............................ 26
  2.6.9 Hybrid functions and predicates ............... 26
  2.6.10 Memory footprint specification: reads clause ... 26
  2.6.11 Specification Modules .......................... 26
2.7 Pointers and physical addressing .................... 27
  2.7.1 Memory blocks and pointer dereferencing .......... 27
  2.7.2 Separation ...................................... 27
  2.7.3 Allocation and deallocation ..................... 27
2.8 Sets as first-class values ........................... 27
2.9 Abrupt termination ................................. 27
2.10 Dependencies information ........................... 27
2.11 Data invariants .................................... 27
  2.11.1 Semantics .................................... 27
  2.11.2 Model variables and model fields ............. 27
2.12 Ghost variables and statements .................... 28
  2.12.1 Volatile variables ............................. 28
  2.13 Undefined values, dangling pointers ............. 28

3 Libraries ............................................ 31

4 Conclusion .......................................... 33

A Appendices .......................................... 35
A.1 Changes .......................................... 36
  A.1.1 Version 1.5-3 ................................ 36
  A.1.2 Version 1.5-2 ................................ 36
  A.1.3 Version 1.5-1 ................................ 36
  A.1.4 Version 1.5-0 ................................ 37
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 1.5-3 of E-ACSL design based on ACSL version 1.5 [2]. 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, Benjamin Monate, Yannick Moy and Virgile Prevosto.
Chapter 1

Introduction

This document is a reference manual for E-ACSL. E-ACSL is an acronym for “Executable ANSI/ISO C Specification Language”. It is an “executable” subset of stable ACSL [2] implemented [4] 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 [2] 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 [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.
Chapter 2

Specification language

2.1 Lexical rules

No difference with AC SL.

2.2 Logic expressions

No difference with AC SL, 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 AC SL, while Figure 2.2 presents grammar of predicates. The only difference between E-AC SL and AC SL predicates are quantifications.

Quantification

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

Guarded integer quantification

Guarded universal quantification is denoted by

\[
\forall \tau x_1, \ldots, x_n; \quad a_1 \leq x_1 \leq b_1 \ldots \&\& a_n \leq x_n \leq b_n \Rightarrow e
\]

and guarded existential quantification by

\[
\exists \tau x_1, \ldots, x_n; \quad a_1 \leq x_1 \leq b_1 \ldots \&\& a_n \leq x_n \leq b_n \&\& e
\]

Therefore each quantified variable belongs to a finite constant 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.

Guarded iterator quantification

In order to iterate over non-integer types, E-AC SL introduces a notion of iterators over types: standard AC SL unguarded quantifications are only allowed over a type which an iterator is attached to.
\begin{figure}[h]
\centering
\begin{tabular}{l}
\textbf{bin-op} ::= \texttt{+} | \texttt{-} | \texttt{*} | \texttt{/} | \texttt{%} \\
\quad | \texttt{==} | \texttt{!=} | \texttt{<=} | \texttt{>=} | \texttt{>} | \texttt{<} \\
\quad \&\& | \texttt{||} \\
\quad \& | \texttt{-->} | \texttt{<-->} | \texttt{^} \\
\textbf{unary-op} ::= \texttt{+} | \texttt{-} \\
\quad \texttt{!} \\
\quad \texttt{-} \\
\quad \texttt{*} \\
\quad \& \\
\textbf{term} ::= \texttt{true} | \texttt{false} \\
\quad \texttt{integer} \\
\quad \texttt{real} \\
\quad \texttt{id} \\
\quad \texttt{unary-op \ term} \\
\quad \texttt{term \ bin-op \ term} \\
\quad \texttt{term \ [ \ term \ ]} \\
\quad \{ \texttt{term} \ \texttt{with} \ [ \ \texttt{term} \ ] = \texttt{term} \} \\
\quad \texttt{term} \ . \ \texttt{id} \\
\quad \{ \texttt{term} \ \texttt{with} \ . \ \texttt{id} = \texttt{term} \} \\
\quad \texttt{term} \ \texttt{-->} \ \texttt{id} \\
\quad ( \ \texttt{type-expr} \ ) \ \texttt{term} \\
\quad \texttt{id} \ ( \ \texttt{term} \ (, \ \texttt{term})^* \ ) \\
\quad ( \ \texttt{term} \ ) \\
\quad \texttt{term} \ ? \ \texttt{term} : \ \texttt{term} \\
\quad \texttt{let id = term ; term} \\
\quad \texttt{sizeof} \ ( \ \texttt{term} \ ) \\
\quad \texttt{sizeof} \ ( \ \texttt{C-type-expr} \ ) \\
\quad \texttt{id} : \ \texttt{term}
\end{tabular}
\caption{Grammar of terms}
\end{figure}
2.2. LOGIC EXPRESSIONS

| rel-op  | ::=  | = | != | <= | >= | > | < |
| pred    | ::=  | \true | \false  |
| term    | ( rel-op term )^+ |
| id      | ( term (, term )* ) |
| ( pred ) |
| pred && pred |
| pred || pred |
| pred ==> pred |
| pred <=> pred |
| ! pred |
| pred ^^ pred |
| term ? pred : pred |
| pred ? pred : pred |
| \let id = term ; pred |
| \let id = pred ; pred |
| \forall binders ; |
| integer-guards ==> pred |
| \exists binders ; |
| integer-guards && pred |
| \forall binders ; |
| iterator-guard ==> pred |
| \exists binders ; |
| iterator-guard && pred |
| \forall binders ; pred |
| \exists binders ; pred |
| id : pred |

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
CHAPTER 2. SPECIFICATION LANGUAGE

\[
\begin{align*}
\text{binders} &::= \text{binder} \ (, \ \text{binder})^* \\
\text{binder} &::= \text{type-expr} \ \text{variable-ident} \\
&\quad (, \ \text{variable-ident})^* \\
\text{type-expr} &::= \text{logic-type-expr} \ | \ C\text{-type-expr} \\
\text{logic-type-expr} &::= \text{built-in-logic-type} \\
&\quad | \ \text{id} \\
\text{built-in-logic-type} &::= \text{boolean} \ | \ \text{integer} \ | \ \text{real} \\
\text{variable-ident} &::= \text{id} \ | \ * \ \text{variable-ident} \\
&\quad | \ \text{variable-ident} [] \\
&\quad | \ (\ \text{variable-ident} )
\end{align*}
\]

Figure 2.3: Grammar of binders and type expressions

\[
\begin{align*}
\text{declaration} &::= \text{//@ iterator } \text{id} \ (, \ \text{wildcard-param} , \ \text{wildcard-param} ) : \\
&\quad \text{nexts terms} ; \ \text{guards predicates} \\
\text{wildcard-param} &::= \text{parameter} \\
&\quad | \ _ \\
\text{terms} &::= \text{term} \ (, \ \text{term})^* \\
\text{predicates} &::= \text{predicate} \ (, \ \text{predicate})^*
\end{align*}
\]

Figure 2.4: Grammar of iterator declarations

Iterators are introduced by a specific construct which attaches two sets — namely \text{nexts} and the \text{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 \(\tau\), \text{nexts} is a set of terms which take an argument of type \(\tau\) and return a value of type \(\tau\) which computes the next element in this type, while \text{guards} is a set of predicates which take an argument of type \(\tau\) and are valid (resp. invalid) to continue (resp. stop) the iteration.

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

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

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

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

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

\textbf{Unguarded quantification} They are only allowed over boolean and char.
2.2. LOGIC EXPRESSIONS

2.2.1 Operators precedence

No difference with ACSL.

Figure 2.5 summarizes operator precedences.

<table>
<thead>
<tr>
<th>class</th>
<th>associativity</th>
<th>operators</th>
</tr>
</thead>
<tbody>
<tr>
<td>selection</td>
<td>left</td>
<td><code>...</code>, <code>-&gt;</code></td>
</tr>
<tr>
<td>unary</td>
<td>right</td>
<td><code>!</code>, <code>+</code>, <code>-</code>, <code>*</code>, <code>(cast) sizeof</code></td>
</tr>
<tr>
<td>multiplicative</td>
<td>left</td>
<td><code>*</code>, <code>/</code>, <code>%</code></td>
</tr>
<tr>
<td>additive</td>
<td>left</td>
<td><code>+</code>, <code>-</code></td>
</tr>
<tr>
<td>shift</td>
<td>left</td>
<td><code>&lt;&lt;</code>, <code>&gt;&gt;</code></td>
</tr>
<tr>
<td>comparison</td>
<td>left</td>
<td><code>&lt;</code>, <code>&lt;=</code>, <code>&gt;</code>, <code>&gt;=</code></td>
</tr>
<tr>
<td>comparison</td>
<td>left</td>
<td><code>==</code>, <code>!=</code></td>
</tr>
<tr>
<td>bitwise and</td>
<td>left</td>
<td><code>&amp;</code></td>
</tr>
<tr>
<td>bitwise xor</td>
<td>left</td>
<td><code>^</code></td>
</tr>
<tr>
<td>bitwise or</td>
<td>left</td>
<td>`</td>
</tr>
<tr>
<td>bitwise implies</td>
<td>left</td>
<td><code>--&gt;</code></td>
</tr>
<tr>
<td>bitwise equiv</td>
<td>left</td>
<td><code>&lt;--&gt;</code></td>
</tr>
<tr>
<td>connective and</td>
<td>left</td>
<td><code>&amp;&amp;</code></td>
</tr>
<tr>
<td>connective xor</td>
<td>left</td>
<td><code>^^</code></td>
</tr>
<tr>
<td>connective or</td>
<td>left</td>
<td>`</td>
</tr>
<tr>
<td>connective implies</td>
<td>right</td>
<td><code>==&gt;</code></td>
</tr>
<tr>
<td>connective equiv</td>
<td>left</td>
<td><code>&lt;==&gt;</code></td>
</tr>
<tr>
<td>ternary connective</td>
<td>right</td>
<td><code>?</code>, <code>??</code>, <code>??</code></td>
</tr>
<tr>
<td>binding</td>
<td>left</td>
<td><code>\forall</code>, <code>\exists</code>, <code>\let</code></td>
</tr>
<tr>
<td>naming</td>
<td>right</td>
<td><code>:</code></td>
</tr>
</tbody>
</table>

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 $e$ 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 $p \implies q$ is also lazy since it is equivalent to $tp || q$. 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:

- \( \false \land \frac{1}{0} = \frac{1}{0} \)
- \( \forall \text{integer } x, -1 \leq x \leq 1 \implies \frac{1}{x} > 0 \)
- \( \forall \text{integer } x, 0 \leq x \leq 0 \implies \false \implies -1 \leq \frac{1}{x} \leq 1 \)
- \( \exists \text{integer } x, 1 \leq x \leq 0 \land -1 \leq \frac{1}{x} \leq 1 \)

In particular, the second one is invalid since the quantification is in fact an enumeration over a finite number of elements, it amounts to \( 1/-1 > 0 \land \frac{1}{0} > 0 \land \frac{1}{1} > 0 \). The first atomic proposition is invalid, so the rest of the conjunction (and in particular \( \frac{1}{0} \)) 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:

- \( \frac{1}{0} = \frac{1}{0} \land \false \)
- \( -1 \leq \frac{1}{0} \leq 1 \implies \true \)
- \( \exists \text{integer } x, -1 \leq x \leq 1 \land \frac{1}{x} > 0 \)

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.

- \( \text{(char)}127 \)
- \( \text{(char)}128 \)

Handling undefinedness in tools It is the responsibility of the tools 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 [2] and mainstream specification languages for runtime assertion checking like JML [9]. Consistency means that, if it exists and is defined, the E-ACSL predicate corresponding to a valid (resp. invalid) ACSL predicate is valid (resp. invalid). Thus it is possible to reuse tools interpreting ACSL like the FRAMA-C’s value analysis plug-in [6] in order to interpret E-ACSL, and it is also possible to perform runtime assertion checking of E-ACSL predicates in the same way than JML predicates. Reader interested by the implications (especially issues) of such a choice may read articles of Patrice Chalin [3, 4].

2.2.3 Typing

No difference with ACSL.
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 with 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 \old and \result

No difference with ACSL.

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

\old is usually difficult to implement, since it requires a deep copy of the location. Thus you would not wonder if most tools do not support it (or support it partially).

2.3.2 Simple function contracts

No difference with ACSL.

\assigns is usually difficult to implement, since it requires the implementation of a memory model. Thus you would not wonder if most tools do not support it (or support it partially).
CHAPTER 2. SPECIFICATION LANGUAGE

function-contract ::= requires-clause* decreases-clause? simple-clause* named-behavior* completeness-clause*

requires-clause ::= requires predicate ;
decreases-clause ::= decreases term (for ident)? ;
simple-clause ::= assigns-clause | ensures-clause
assigns-clause ::= assigns locations ;
locations ::= location (, location)* | \nothing
ensures-clause ::= ensures predicate ;
named-behavior ::= behavior id : behavior-body
behavior-body ::= assumes-clause* requires-clause* simple-clause*
assumes-clause ::= assumes predicate ;
completeness-clause ::= complete behaviors (id (, id)*)? ; | disjoint behaviors (id (, id)*)? ;

Figure 2.6: Grammar of function contracts

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

Figure 2.7: \old and \result in terms
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.

\[
\begin{align*}
tset &::= \emptyset & \text{empty set} \\
&| tset \rightarrow id \\
&| tset . id \\
&| * tset \\
&| \& tset \\
&| tset [ tset ] \\
&| \text{term .. term} & \text{range} \\
&| \\text{union} ( \ tset , \ tset)* & \text{union of locations} \\
&| \\text{inter} ( \ tset , \ tset)* & \text{intersection} \\
&| tset + tset \\
&| ( tset \\
&| \{ tset \mid \text{binder} ; \text{guards (&& \ pred)} \} & \text{set comprehension} \\
&| \{ \text{term} \} & \text{explicit singleton} \\
&| \text{term} & \text{implicit singleton} \\
pred &::= \\subset ( tset , \ tset ) & \text{set inclusion}
\end{align*}
\]

Figure 2.8: Grammar for sets of terms

**Example 2.5** The set \{ x \mid \text{integer } x; 0 \leq x \leq 9 \&\& 20 \leq x \leq 29 \} 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.*

2.4 Statement annotations

2.4.1 Assertions

*No difference with ACSL.*

Figure 2.9 summarizes grammar for assertions.
CHAPTER 2. SPECIFICATION LANGUAGE

\[
\text{compound-statement ::= \{ declaration* statement* assertion\} }
\]
\[
\text{statement ::= assertion statement}
\]
\[
\text{assertion ::= /*@ assert pred ; */}
\]
\[
\text{\quad | /*@ for id (, id)* : assert pred ; */}
\]

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.

\[
\text{statement ::= /*@ loop-annot */}
\]
\[
\text{\quad while ( expr ) statement}
\]
\[
\text{\quad | /*@ loop-annot */}
\]
\[
\text{\quad for ( expr ; expr ; expr ) statement}
\]
\[
\text{\quad | /*@ loop-annot */}
\]
\[
\text{\quad do statement}
\]
\[
\text{\quad while ( expr ) ;}
\]
\[
\text{loop-annot ::= loop-clause*}
\]
\[
\text{\quad loop-behavior*}
\]
\[
\text{\quad loop-variant?}
\]
\[
\text{loop-clause ::= loop-invariant}
\]
\[
\text{\quad | loop-assigns}
\]
\[
\text{loop-invariant ::= loop invariant pred ;}
\]
\[
\text{loop-assigns ::= loop assigns locations ;}
\]
\[
\text{loop-behavior ::= for id (, id)* :}
\]
\[
\text{\quad loop-clause* annotation for behavior id}
\]
\[
\text{loop-variant ::= loop variant term ;}
\]
\[
\text{\quad | loop variant term for id ; variant for relation id}
\]

Figure 2.10: Grammar for loop annotations

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

Loop invariants

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

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

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

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

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

Example 2.6 In the following, $bsearch(t,n,v)$ searches for element $v$ in array $t$ between indices $0$ and $n-1$.

```
1  /* @ requires n >= 0 & & valid(t*(0..n-1));
2    @ assigns \nothing;
3    @ ensures -1 <= \result <= n-1;
4    @ behavior success:
5    @ ensures \result >= 0 ==> t[\result] == v;
6    @ behavior failure:
7    @ assumes t_is_sorted : \forallall integer k1, int k2;
8    @ 0 <= k1 <= k2 <= n-1 ==> t[k1] <= t[k2];
9    @ ensures \result == -1 ==> 
10   @ \forallall integer k; 0 <= k < n ==> t[k] != v;
11 */
12 int bsearch(double t[], int n, double v) {
13   int l = 0, u = n - 1;
14   /* @ loop invariant 0 <= l & & u <= n-1; 
15    @ for failure: loop invariant
16    @ \forallall integer k; 0 <= k < n ==> t[k] == v ==> l <= k <= u; 
17    @ */
18   while (l <= u) {
19      int m = l + (u-l)/2; // better than (l+u)/2
20      if (t[m] < v) l = m + 1;
21      else if (t[m] > v) u = m - 1;
22      else return m;
23   }
24   return -1;
25 }
```

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

```
1  /* @ requires n >= 0 & & valid(t*(0..n-1));
2    @ assigns \nothing;
3    @ ensures -1 <= \result <= n-1;
4    @ behavior success:
5    @ ensures \result >= 0 ==> t[\result] == v;
6    @ behavior failure:
7    @ assumes t_is_sorted : \forallall integer k1, int k2;
8    @ 0 <= k1 <= k2 <= n-1 ==> t[k1] <= t[k2];
9    @ ensures \result == -1 ==> 
```
General inductive invariant

Syntax of these kinds of invariant is shown Figure 2.11

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

2.4.3 Built-in construct \at

\at is usually difficult to implement, since it requires a deep copy of the location. Thus you would not wonder if most tools do not support it (or support it partially).

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.

```plaintext
statement ::= /*@ statement-contract */ statement
statement-contract ::= (for id (, id)* : requires-clause* simple-clause* behavior-body*)
```

Figure 2.12: Grammar for statement contracts
2.5 Termination

No difference with ACSL, but no terminates clauses.

2.5.1 Integer measures

No difference with ACSL.

2.5.2 General measures

No difference with ACSL.

2.5.3 Recursive function calls

No difference with ACSL.

2.5.4 Non-terminating functions

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

2.6 Logic specifications

Limited to stable and computable features.

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

```
C-global-decl ::= /*@ logic-def+ */
logic-def ::= logic-const-def
| logic-function-def
| predicate-def
type-expr ::= id
logic-const-def ::= logic type-expr id = term ;
logic-function-def ::= logic type-expr id parameters = term ;
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, allocation and deallocation is unsupported.

2.7.1 Memory blocks and pointer dereferencing

No difference with ACSL.
\( \text{base\_addr}, \text{block\_length}, \text{valid} \) and \( \text{offset} \) 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 such feature in E-ACSL, since it is still experimental in ACSL.

2.7.3 Allocation and deallocation

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

2.8 Sets as first-class values

No difference with ACSL.

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 2.14 summarizes 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 2.15 summarizes grammar for declarations of model variables and fields.
CHAPTER 2. SPECIFICATION LANGUAGE

\[
\begin{align*}
\text{declaration} &::= */@\text{data-inv-decl} */ \\
data-inv-decl &::= \text{data-invariant} \mid \text{type-invariant} \\
data-invariant &::= \text{inv-strength}\? \text{global invariant} \\
    &\quad \text{id} : \text{pred} ; \\
type-invariant &::= \text{inv-strength}\? \text{type invariant} \\
    &\quad \text{id}(\text{C-type-expr \text{id}}) = \text{pred} ; \\
\text{inv-strength} &::= \text{weak} \mid \text{strong}
\end{align*}
\]

Figure 2.14: Grammar for declarations of data invariants

\[
\begin{align*}
\text{declaration} &::= \text{C-declaration} \\
    &\mid */@\text{model C-declaration} */ \quad \text{model variable} \\
\text{struct-declaration} &::= \text{C-struct-declaration} \\
    &\mid */@\text{model C-struct-declaration} */ \quad \text{model field}
\end{align*}
\]

Figure 2.15: 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 2.16 summarizes grammar for ghost statements which is the same than the one of ACSL.

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.

\text{initialized} and \text{specified} 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).
Figure 2.16: Grammar for ghost statements
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 [2].
Chapter 4

Conclusion

This document presents an Executable ANSI/ISO C Specification Language. It provides a subset of ACSL [2] implemented [1] 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.
A.1 Changes

A.1.1 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 guarded 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.

A.1.2 Version 1.5-2

- Section 2.2: remove laziness of operator $\Leftarrow$.
- Section 2.2: restrict guarded quantifications to integer.
- Section 2.2: add guarded 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..

A.1.3 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 `$\Leftarrow$`.
- 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.
A.1. CHANGES

- **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.

A.1.4 Version 1.5-0

- Initial version.


<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Grammar of terms</td>
<td>14</td>
</tr>
<tr>
<td>2.2</td>
<td>Grammar of predicates</td>
<td>15</td>
</tr>
<tr>
<td>2.3</td>
<td>Grammar of binders and type expressions</td>
<td>16</td>
</tr>
<tr>
<td>2.4</td>
<td>Grammar of iterator declarations</td>
<td>16</td>
</tr>
<tr>
<td>2.5</td>
<td>Operator precedence</td>
<td>17</td>
</tr>
<tr>
<td>2.6</td>
<td>Grammar of function contracts</td>
<td>20</td>
</tr>
<tr>
<td>2.7</td>
<td>\old and \result in terms</td>
<td>20</td>
</tr>
<tr>
<td>2.8</td>
<td>Grammar for sets of terms</td>
<td>21</td>
</tr>
<tr>
<td>2.9</td>
<td>Grammar for assertions</td>
<td>22</td>
</tr>
<tr>
<td>2.10</td>
<td>Grammar for loop annotations</td>
<td>22</td>
</tr>
<tr>
<td>2.11</td>
<td>Grammar for general inductive invariants</td>
<td>24</td>
</tr>
<tr>
<td>2.12</td>
<td>Grammar for statement contracts</td>
<td>24</td>
</tr>
<tr>
<td>2.13</td>
<td>Grammar for global logic definitions</td>
<td>25</td>
</tr>
<tr>
<td>2.14</td>
<td>Grammar for declarations of data invariants</td>
<td>28</td>
</tr>
<tr>
<td>2.15</td>
<td>Grammar for declarations of model variables and fields</td>
<td>28</td>
</tr>
<tr>
<td>2.16</td>
<td>Grammar for ghost statements</td>
<td>29</td>
</tr>
</tbody>
</table>
Index

?, 12, 13
_, 14
annotation, 19
assert, 19, 20
assigns, 18, 20
assumes, 18
behavior, 19
behavior, 18
behaviors, 18
boolean, 14
complete, 18
contract, 17, 22
data invariant, 25
decreases, 18\decreases, 23
disjoint, 18
do, 20
else, 27\empty, 19
ensures, 18\exists, 13
\false, 12, 13
for, 18, 20, 22\forall, 13
function behavior, 19
function contract, 17
ghost, 26
ghost, 27
global, 26
global invariant, 25
grammar entries
  C-global-decl, 23
  assertion, 20, 22
  assigns-clause, 18
  assumes-clause, 18
  behavior-clause, 18
bin-op, 12
binders, 14
binder, 14
built-in-logic-type, 14
completeness-clause, 18
compound-statement, 20
data-invariant, 26
data-invariant, 26
declaration, 14, 26, 27
decreases-clause, 18
direct-declarator, 27
ensures-clause, 18
function-contract, 18
ghost-selection-statement, 27
ghost-type-specifier, 27
integer-guard-op, 13
integer-guards, 13
iter, 13
inv-strength, 26
iterator-guard, 13
locations, 18
logic-const-def, 23
logic-def, 23
logic-function-def, 23
logic-type-expr, 14
loop-annot, 20
loop-assigns, 20
loop-behavior, 20
loop-clause, 20
loop-invariant, 20
loop-variant, 20
named-behavior, 18
parameters, 23
parameter, 23
postfix-expression, 27
predicate-def, 23
predicates, 14
pred, 13, 18, 19
rel-op, 13
requires-clause, 18
simple-clause, 18
INDEX

statement-contract, 22
statements-ghost, 27
statement, 20, 22, 27
struct-declaration, 26, 27
terms, 14
term, 12, 18
tset, 19
type-expr, 14, 23
type-invariant, 26
unary-op, 12
variable-ident, 14
wildcard-param, 14
guards, 14

hybrid
   function, 24
   predicate, 24

if, 27
integer, 14
\inter, 19
invariant, 20
   data, 25
   global, 25
   type, 25
invariant, 20, 22, 26
iterator, 14
\let, 12, 13
location, 25
logic, 23
logic specification, 23
loop, 20

model, 25
model, 26

nexts, 14
\nothing, 18
\old, 18

predicate, 23
real, 14
recursion, 24
requires, 18
\result, 18
sizeof, 12
specification, 23
statement contract, 22

strong, 26
\subset, 19

termination, 22
\true, 12, 13
type, 26
type invariant, 25
\union, 19

variant, 20
\variant, 23

weak, 26
while, 20
\with, 12