Common Specification Language for Static and Dynamic Analysis of C Programs*

Mickaël Delahaye[1] Nikolai Kosmatov[2] Julien Signoles[2]
[1]UJF-Grenoble 1, LIG, UMR 5217, 38041 Grenoble France
[2]CEA, LIST, Software Reliability Laboratory, PC 174, 91191 Gif-sur-Yvette France
[1]firstname.lastname@imag.fr, [2]firstname.lastname@cea.fr

ABSTRACT

Various combinations of static and dynamic analysis techniques were recently shown to be beneficial for software verification. A frequent obstacle to combining different tools in a completely automatic way is the lack of a common specification language. Our work proposes to translate a Pre-Post based specification into executable C code. This paper presents E-ACSL, subset of the ACSL specification language for C programs, and its automatic translator into C implemented as a FRAMA-C plug-in. The resulting C code is executable and can be used by a dynamic analysis tool. We illustrate how the PATHCRAWLER test generation tool automatically treats such pre- and postconditions specified as C functions.

1. INTRODUCTION

A usual input of a software verification tool includes a program with its (partial) specification. Testing tools need at least a *precondition* (or *test context*) specifying admissible input data on which the program should be tested, and usually require an *oracle*, deciding if the results of the execution on a given test are correct. Detecting potential runtime errors by abstract interpretation also needs a precondition to improve its precision. Tools for program proving require a formal specification (or *contract*) with pre/postconditions, loop invariants, etc. Although the specification is extremely important for the verification process, its format varies from one tool to another, especially between static and dynamic analysis tools. That makes it difficult to combine them in a completely automatic way, while recent research (*e.g.* [11, 16, 23, 13, 7]) showed that combinations of static and dynamic analysis can be beneficial for software verification.

A concrete example is SANTE [7] which efficiently combines the value analysis plug-in [5] of FRAMA-C[1] [12] and the structural test generation tool PATHCRAWLER [4] for detection of runtime errors in C programs. While all static analyzers of FRAMA-C share a common specification language, called ACSL [2], PATHCRAWLER requires a precondition specified in another format and an oracle defined by a C function. Rewriting the precondition of the target C function in the PATHCRAWLER format remains the only manual step of the SANTE method.

The primary objective of our work is to specify E-ACSL [21], an expressive sub-language of ACSL that can be translated into C, compiled and used as executable specification. Our second goal is to develop its automatic translator E-ACSL2C into C [22].

This approach brings several benefits. To the best of our knowledge, E-ACSL is the first formal behavioral specification language for C that builds a bridge between static and dynamic analysis tools and avoids manual rewriting of a formal program specification for testing. Second, choosing a sub-language of ACSL has the advantage of being supported by existing FRAMA-C analyzers. Third, translating into C rather than into a specific format of a particular tool allows the usage by other analysis tools for C. Fourth, the possibility to observe the status of an annotation during a concrete execution may be very helpful while writing a correct specification of a given program, *e.g.* for later program proving. Finally, an executable specification makes it possible to check runtime assertions that cannot be verified statically and to establish a link between monitoring tools and static analysis tools.

The contributions of this paper include a presentation of E-ACSL, a novel executable specification language for C programs, and its advantages (Sec. 1, 2), an overview of its translation into C (Sec. 3) and an illustration of how the resulting pre- and postconditions in the form of C functions are automatically treated by the test generation tool PATHCRAWLER (Sec. 4).

2. EXECUTABLE ANSI/ISO C SPECIFICATION LANGUAGE: E-ACSL

E-ACSL [21] is a strict subset of ACSL [2], which is a behavioral specification language implemented in FRAMA-C [12], a platform dedicated to analysis of C programs. ACSL takes the best of the specification languages of both CADUCEUS [15] (itself inspired by JML [8]) and CAVEAT [3], two pioneer tools of C program proving.

On the one hand, designed as a subset of ACSL, E-ACSL preserves ACSL semantics. Therefore, existing FRAMA-C analyzers supporting ACSL continue to be used with E-ACSL without any change. On the other hand, thanks to its characteristics explained later in this section, the E-ACSL language is *executable*, that is, all its annotations can be translated into C and executed at runtime. Thus it can be used by dynamic analyses and monitors. Due to these two specific features (preserving ACSL semantics and being executable), E-ACSL eases combinations of static and dynamic analyses.

Overview of ACSL.

ACSL [2] is expressive enough to be able to express most functional properties of C programs. It has already been used in many

*This work has been partially funded by the FUI9 'Hi-Lite' project.
[1]`http://frama-c.com`

SAC'13 March 18-22, 2013, Coimbra, Portugal.

```
1  int A[10];
2  /*@ requires \forall integer i; 0<=i<9 ==> A[i]<=A[i+1];
3      behavior elt_present:
4        assumes \exists integer j; 0<=j<10 && A[j]==elt;
5        ensures \result == 1;
6      behavior elt_absent:
7        assumes \forall integer j; 0<=j<10 ==> A[j]!=elt;
8        ensures \result == 0; */
9  int search(int elt){ // linear search in a sorted array
10   int k;
11   /*@ loop invariant 0 <= k <= 10;
12     @ loop invariant
13     @   \forall integer i; 0 <= i < k ==> A[i] < elt; */
14   for(k = 0; k < 10; k++)
15     if(A[k] == elt) return 1; // element found
16     else if(A[k] > elt) return 0; // not found (A sorted)
17   return 0; // not found
18 }
```

Figure 1: Function `search` looks for element `elt` in sorted array `A`

projects, including large-scale industrial ones [12].

ACSL is based on a typed first-order logic in which terms may contain *pure* (*i.e.* side-effect free) C expressions and special keywords. For instance, the `\result` keyword allows the user to speak about the result of a function, while `\valid` is a built-in predicate stating that its argument is a valid pointer. ACSL terms include sets of terms which can express groups of memory locations in a very convenient way. For instance, `*(t+(0..99))` is the term denoting the first 100 cells of an array `t`. In addition to all machine types, terms also include mathematical integers (discussed later in this section) and reals.

An EIFFEL-like contract [19] may be associated to each function in order to specify its pre- and postconditions. Contracts may also include termination clauses (specifying termination properties) and an assigns clause (to specify which memory locations may be modified by the function). For example, clause `assigns *(t+(0..99));` in a function contract states that the function can only modify the first 100 elements of `t`. These contracts may be split into several named guarded behaviors for which the users may require completeness and/or disjointness. Contracts may also be associated to statements, as well as assertions, loop invariants and loop variants. Annotations also include definitions of (inductive) predicates, axiomatics, lemmas, logic functions, data invariants and ghost code.

Running Example: Linear Search.

Fig. 1 shows a simple C function `search` with an ACSL function contract (that is also an E-ACSL contract) enclosed into `@`-comments. The clause `requires` (line 2) is the precondition stating that the global array `A` must be sorted when this function is called. The function has two different behaviors. If `elt` belongs to `A` (clause `assumes` of behavior `elt_present` at line 4), then the function must return `1` (clause `ensures` of this behavior at line 5). Otherwise (clause `assumes` of behavior `elt_absent` at line 7), the function must return `0` (line 8). The code also contains two loop invariants (lines 11-13).

Overview of E-ACSL.

The requirement of being executable brings some natural limitations on ACSL annotations that can be supported in E-ACSL. One of them is related to quantifications. E-ACSL syntactically limits quantifications to range over finite domains of integers in order to be computable. Thus the general syntactic patterns of universal and existential quantifications in E-ACSL are as follows:

```
1  \forall τ x1,...,xn;
2    a1 <= x1 <= b1 && ... && an <= xn <= bn
3    ==> p
4
5  \exists τ x1,...,xn;
6    a1 <= x1 <= b1 && ... && an <= xn <= bn
7    && p
```

where τ denotes the type of integral variables $x_1, \ldots, x_n$, and a_i, b_i may be any E-ACSL terms. In ACSL, such guards specifying an interval for each binding (cf lines 2, 6) are not mandatory. Notice that the quantifications in Fig. 1 respect these patterns.

Moreover, E-ACSL allows the user to define dedicated iterators over recursive C datastructures (like linked lists, trees), and then to quantify over them as in the following example for binary trees. (This is the only feature of E-ACSL which is not yet part of ACSL.)

```
1  // type of binary trees
2  struct btree {
3    int val;
4    struct btree *left, *right;
5  };
6
7  // declare an iterator over a binary tree
8  /*@ iterator access(_, struct btree *t):
9    @   nexts t->left, t->right;
10   @   guards \valid(t->left), \valid(t->right); */
11
12 // is_even(t) is valid iff all values
13 // in the binary tree t are even
14 /*@ predicate is_even(struct btree *t) =
15   @   \forall struct btree *tt;
16   @     access(tt, t) ==> tt->val % 2 == 0; */
```

The `nexts` fields define how to access the immediate successors, whenever the corresponding `guards` hold. In the same spirit as for quantifications over integers, sets of memory locations are also syntactically limited to finite sets.

Loop invariants in E-ACSL lose their inductive nature which is usually required to prove them by induction. Indeed, preservation of a loop invariant after any iteration should be proved and cannot be established after one particular execution. Roughly speaking, without the inductive scheme, a loop invariant in E-ACSL is equivalent to two assertions: the first one before entering the loop and the second one at the end of each iteration of the loop body.

Furthermore, in E-ACSL there are no lemmas (which usually express mathematical non-executable properties) nor axiomatics (non-executable by nature). There is also no way to express termination properties of a loop or a recursive function, since detecting non-termination in finite time at runtime is not possible. Finally, E-ACSL does not include so-called experimental features of ACSL that are likely to evolve in the future: the decision whether they should be included in E-ACSL is postponed to the time when they become stable.

All the other features of ACSL are fully supported in E-ACSL, including mathematical integers, predicates and functions over C pointers. Last but not least, a major semantical difference between ACSL and E-ACSL is the presence of undefined terms in E-ACSL (discussed later in this section).

Integers.

In ACSL, integer constants and operators, as well as logic variables of type `integer`, denote *mathematical integers:* integer arithmetics is unbounded and never overflows. ACSL holds a small subtyping system to automatically coerce C integral types into mathematical integers. This design was chosen for several reasons. First, one of the main goals of FRAMA-C is program proving by discharging proof obligations to automatic theorem provers. Such provers usually work much better with mathematical arithmetics than with *modular arithmetics*, that is, bounded arithmetics with

overflows. Second, specifications are usually written without any implementation detail in mind, and potential overflows are implementation details. Third, it is still possible to use bounded modular arithmetics when required by using explicit casts: for instance, `(int)(MAXINT + 1)` is equal to `MININT`, the smallest representable value of `int`. Fourth, this choice makes it much easier to talk about potential overflows in specifications: for example, thanks to mathematical arithmetics, `/*@ assert MININT <= x+y <= MAXINT;*/` specifies in the easiest way that `x+y` must not overflow. To preserve ACSL semantics, E-ACSL uses the same rules for arithmetics, even if that leads to some implementation issues (discussed in Sec. 3).

Undefinedness.

Another difficulty comes from the fact that any ACSL term is logically significant, including those like `1/0`. Such terms are not problematic in ACSL. They help the user to write simple specifications like `u/v == 2` without taking care of the nullity of `v`: if `v` might be equal to `0`, then this predicate would be just invalid. In E-ACSL, such terms and predicates are however problematic since their translations into C provoke runtime errors: that is not acceptable. This problem extends to all terms and predicates which contain a C expression leading to a runtime error.

In order to solve it, we follow Chalin's Runtime Assertion Checking semantics [6] by stating that semantics of such terms is "undefined": E-ACSL uses a 3-valued logic [18] like ALFA [9] or JML [8]. It is the responsibility of the tools interpreting E-ACSL to ensure that an undefined term is never evaluated. Undefinedness is nevertheless consistent with ACSL since, if it exists and is defined, the E-ACSL predicate corresponding to a valid (resp. an invalid) ACSL predicate is valid (resp. invalid) as well. Thus re-using ACSL-compatible analyzers is preserved. An indirect consequence of this design is that operators `&&`, `||`, `_?_:_` and `==>` in E-ACSL are lazy (like the C counterparts for the first three of them).

3. AUTOMATIC TRANSLATOR TO C

We have developed a FRAMA-C plug-in, called E-ACSL2C in this paper, which automatically translates E-ACSL into C. Its released open-source version[2] already handles a significant part of E-ACSL [22], while the current SVN version extends it to even more features that we discuss in this section. E-ACSL2C takes as input an annotated C program and returns the same program in which annotations have been converted into C code dedicated to runtime assertion checking: this code fails at runtime if and only if an annotation is violated.

The primary goal of E-ACSL2C is runtime assertion checking. But, while ACSL is the *lingua franca* of FRAMA-C for static analyses collaboration [12], E-ACSL also aims to become the *lingua franca* of FRAMA-C for collaboration of static and dynamic analyses, the main purpose of this paper. The results obtained by different FRAMA-C analyzers can be then consolidated in the FRAMA-C platform using the algorithm presented in [10].

The main idea of translation with E-ACSL2C is illustrated by Fig. 2, where the function `e_acsl_assert` fails at runtime if its argument corresponding to the E-ACSL assertion condition is false. Unfortunately this simple translation scheme does not work in the general case for several reasons.

Special Keywords.

First, special E-ACSL keywords are not directly translatable into C expressions. For instance, handling `\result` requires to introduce a variable to store the result of the function while handling

```
1 int div(int x, int y) {
2   /*@ assert y-1 != 0; */
3   return x / (y-1);
4 }
```

```
1 int div(int x, int y) {
2   /*@ assert y-1 != 0; */
3   e_acsl_assert(y-1 != 0);
4   return x / (y-1);
5 }
```

Figure 2: Naive E-ACSL2C translation. Original code (left) *vs* translated code (right).

quantifications requires to generate a `for`-loop. Furthermore, the translation is not necessarily local to the program point where the predicate must hold. For instance, converting the term `\at(x,L)` (representing the value of `x` at label `L`) requires to generate code at program point `L` to store the value of `x` at `L`. Translation of all these terms is already supported in the current version [22]. Fig. 3 provides a slightly more complex example of a non local E-ACSL2C translation.

```
1  int A, B;
2  /*@ ensures A == \old(B);
3    @ ensures B == \old(A);
4    @ ensures \result ==
5    @   (A>B?A:B); */
6  int max_swap(void) {
7    int tmp = A;
8    A = B;
9    B = tmp;
10   if (A >= B) return A;
11   return B;
12 }
```

```
1  int A, B;
2  /*@ ensures A == \old(B);
3    @ ensures B == \old(A);
4    @ ensures \result ==
5    @   (A>B?A:B); */
6  int max_swap(void) {
7    int tmp_A, tmp_B, res;
8    int tmp;
9    tmp_A = A; // \old(A)
10   tmp_B = B; // \old(B)
11   tmp = A;
12   A = B;
13   B = tmp;
14   if (A >= B) res = A;
15   res = B;   // \result
16   e_acsl_assert
17     (A == tmp_B);
18   e_acsl_assert
19     (B == tmp_A);
20   { int tmp_if;
21     if (A>B) tmp_if = A;
22     else tmp_if = B;
23     e_acsl_assert
24       (res == tmp_if); }
25   return res;
26 }
```

Figure 3: More complex translation.

Runtime Errors in Annotations.

The second difficulty is related to possible runtime errors. We must generate additional guards to prevent execution of undefined terms (cf *Undefinedness* section above). E-ACSL2C already handles some particular cases. Let us present here a simple three-step solution we propose to solve this issue in a systematic way.

First, run E-ACSL2C to generate (non guarded) C code. For example, here is a function `foo` and its E-ACSL2C translation.

```
1 int foo(int u, int v) {
2   /*@ assert u/v == 2; */
3   return u/v;
4 }
```

```
1 int foo(int u, int v) {
2   /*@ assert u/v == 2; */
3   e_acsl_assert(u/v == 2);
4   return u/v;
5 }
```

Second, run the existing FRAMA-C plug-in RTE[3], which systematically generates an ACSL annotation to prevent any potential runtime error. RTE detects a risk of runtime error and inserts necessary assertions. Third, convert these new assertions into C code

[2] `http://frama-c.com/eacsl.html`

[3] `http://frama-c.com/rte.html`

with E-ACSL2C again. For function `foo`, the resulting code after the second and the third steps is the following.

```
1 int foo(int u, int v) {
2   /*@ assert v != 0; */
3   /*@ assert u/v == 2; */
4   e_acsl_assert(u/v == 2);
5   return u/v;
6 }
```

```
1 int foo(int u, int v) {
2   /*@ assert v != 0; */
3   e_acsl_assert(v != 0);
4   /*@ assert u/v == 2; */
5   e_acsl_assert(u/v == 2);
6   return u/v;
7 }
```

In this way, we handle any undefined term safely.

Implementing Integers.

The third issue is that the simplified translation scheme of Fig. 2 is in general incorrect since it uses modular arithmetics instead of mathematical one. To address this issue, we use the GMP library[4], which provides arbitrary precision arithmetics operating on signed integers. But it entails calling GMP functions for variable initialization, mathematical operations, and memory deallocation. In the previously released version, instead of the single conditional of Fig. 2, the generated code for the assertion is that of Fig. 4. Thus using GMP integers leads to a much more complex translation: it is the price to pay for correctness according to ACSL semantics.

```
mpz_t e_acsl_1, e_acsl_2, e_acsl_3, e_acsl_4;
int e_acsl_5;
mpz_init_set_si(e_acsl_1, y);            // e_acsl_1 = y
mpz_init_set_si(e_acsl_2, 1);            // e_acsl_2 = 1
mpz_init(e_acsl_3);
mpz_sub(e_acsl_3, e_acsl_1, e_acsl_2);
// e_acsl_3 = y-1
mpz_init_set_si(e_acsl_4, 0);            // e_acsl_4 = 0
e_acsl_5 = mpz_cmp(e_acsl_3, e_acsl_4);  // (y-1) == 0
e_acsl_assert(e_acsl_5 == 0);            // runtime check
mpz_clear(e_acsl_1); mpz_clear(e_acsl_2);// deallocate
mpz_clear(e_acsl_3); mpz_clear(e_acsl_4);
```

Figure 4: Translation using GMP integers.

However, for simple examples this translation may be unnecessarily complex. Fortunately, the upcoming version of E-ACSL2C embeds a type system, summarized in Fig. 5, that automatically infers for any integer term t an interval into which t fits for sure. For a (mathematical) integer constant, we infer the corresponding singleton interval. For all arithmetic operators (as illustrated in Fig. 5 for $+$ operator), we can easily compute the smallest interval from the deduced intervals of their operands. The interval of a left value, like a variable, is easily computed as the value range of its C integral type, taking into account the considered architecture. The interval of a conditional is computed by joining the intervals of the left and right parts: it is the smallest interval containing both of them. It is the reverse for casts: the inferred interval is the smallest interval containing either the values of the type, or the interval of the term. Potential invalid downcasts in case of overflows are prevented by annotations generated by RTE as explained before.

Intervals may be efficiently computed. Furthermore, if an interval becomes too big to be representable within a C type, we approximate it by $[-\infty, +\infty]$ to speed up the computation.

Afterwards, according to the considered architecture, it is easy to deduce the smallest C type (if any) which can be safely used for representing the translation of t into C, or to use GMPs otherwise. On our small examples, the generated code is similar to Fig. 2 and 3. For Fig. 2, it casts `int` values to `long long` (assuming a standard architecture) to be sure that `y-1` does not overflow. In practice, this type reasoning avoids GMP operations in a very big number of cases.

[4] `http://gmplib.org`

$$\frac{n \in \mathbb{Z}}{\vdash n : [n;n]}\ constant \qquad \frac{\vdash t1 : [l1;u1] \quad \vdash t2 : [l2;u2]}{\vdash t1 + t2 : [l1 + l2; u1 + u2]}\ plus$$

$$\frac{x \text{ of type } \tau}{\vdash x : [\text{min\_val}(\tau); \text{max\_val}(\tau)]}\ C\ variable$$

$$\frac{\vdash t1 : \_ \quad \vdash t2 : [l2;u2] \quad \vdash t3 : [l3;u3]}{\vdash t1?t2 : t3 \ : \ [\min(l2,l3); \max(u2,u3)]}\ conditional$$

$$\frac{\vdash t : [l1;u1] \quad \text{min\_val}(\tau) = l2 \quad \text{max\_val}(\tau) = u2}{\vdash (\tau)t : [\max(l1,l2); \min(u1,u2)]}\ cast$$

Figure 5: Type system (main rules)

Memory Observation Library.

Handling memory-related E-ACSL constructs (like **`\valid`**) requires to know the structure of valid memory blocks at runtime. We implemented a special C library for memory observation, and link it to the code generated by E-ACSL2C. In addition, the translated C code is instrumented in order to record and update the state of valid memory blocks and initialized locations.

Fig. 6 presents a very small example introducing how the memory recording and observation functions are used in the generated code when a memory location is allocated (call of `__store_block`), deallocated (`__delete_block`), initialized (`__full_init`), or checked for being valid (`__initialized` and `__valid`). For global variables, recording functions are called from `main`.

```
1 int *A;
2 void main(void) {
3   /*@ assert \valid(A); */
4   *A = 0;
5 }
```

```
1 int *A;
2 void main(void) {
3   int e_acsl_1, e_acsl_2;
4   __store_block(&A,4);
5   __full_init(&A);
6   /*@ assert \valid(A); */
7   e_acsl_1 =
8     __initialized(&A,
9       sizeof(int*));
10  if (e_acsl_1)
11    e_acsl_2 = _valid(A,
12      sizeof(int));
13  else
14    e_acsl_2 = 0;
15  e_acsl_assert(e_acsl_2);
16  __full_init(A);
17  *A = 0;
18  __delete_block(&A);
19 }
```

Figure 6: Translation involving memory observation.

The current translation appears unnecessarily heavy in some cases because it requires to instrument each statement that modifies the memory (even unrelated to memory blocks involved in annotations) in order to update the runtime memory record. To optimize memory space and time performances of the C code generated by E-ACSL2C, we are currently implementing an efficient backward dataflow analysis which computes an over-approximation of the statements which are required to preserve the semantics of E-ACSL memory-related constructs. Therefore only these statements have to be instrumented.

```
1 // the code of Fig.1 above
2 int search_precond(int elt){
3   int i, forall_OK;
4   for(i = 0, forall_OK = 1; i < 9; i++)
5     if(!(A[i] <= A[i+1])) // fail: A is not ordered
6       { forall_OK = 0; break; }
7   return forall_OK; }
```

Figure 7: File `search.c` containing the C code of Fig. 1 and the precondition translated from E-ACSL into C by E-ACSL2C

4. PATHCRAWLER SPECIFICATION

Functions `search_precond` (Fig. 7) and `search_postcond` (Fig. 8) present the simplified result of the automatic translation into C by E-ACSL2C for the specification of the function `search` of Fig. 1. The `\forall` clause in the precondition (Fig. 1, line 2) is translated by a loop (Fig. 7, lines 4–6). The behavior `elt_present` is translated by the lines 3–9 of Fig. 8. The `exists` clause at the assumption of this behavior (Fig. 1, line 4) is translated by the loop at lines 3–4 of Fig. 8. When it is true, the postcondition of the behavior (Fig. 1, line 5) must be checked (Fig. 8, lines 5–9). Similarly, the behavior `elt_absent` is translated by the lines 10–16 of Fig. 8. The actual automatic translation is slightly different: it is a bit more verbose and inlines the translated C code at the beginning (for the precondition) and at the end (for the postcondition) of the function instead of putting it into separate functions.

Let us illustrate how the pre- and postconditions translated into C by E-ACSL2C can be automatically used by PATHCRAWLER. The C precondition can be added into the C file containing the function under test like shown in Fig. 7. The reader may submit the complete file `search.c` of Fig. 7 to PATHCRAWLER online[5]. The C precondition `search_precond` will be automatically detected and taken into account during test generation to generate tests satisfying the precondition.

The C postcondition may be used inside an oracle to provide a verdict for each generated test. The default oracle, generated automatically by PATHCRAWLER, always gives the `unknown` verdict. It can be customized and replaced by the code given in Fig. 8. The C postcondition `search_postcond` is directly used in the oracle function `oracle_search` (whose name and signature must be respected) to provide a verdict. With this oracle, a meaningful verdict (success or failure) will be provided for each test.

The function `oracle_search` can be easily generated automatically so that the whole process from an E-ACSL specification to a C specification treatable by a testing tool becomes automatic. Therefore, E-ACSL provides a common specification language for static and dynamic analysis tools.

5. RELATED WORK

Although the C language is one of the most popular programming languages, its semantics and low-level operations are still major obstacles to apply formal methods. However, there has been active research in this domain.

Necula et al. [20] propose to check the absence of buffer overflows in C code through a dedicated type system. When their type system is not able to discard threats, dynamic checks are injected into the code to allow dynamic verification. In a similar manner, Hackett et al. [17] propose to augment the traditional type system of C to express various conditions on inputs and outputs of C functions that can be, in part, checked statically. E-ACSL is a full-fledged specification language for C, which allows specifying assertions,

[5] `http://pathcrawler-online.com`

```
1  int search_postcond(int elt, int result){
2    int j, exists_OK, forall_OK;
3    for(j = 0, exists_OK = 0; j < 10; j++)
4      if(A[j] == elt) { exists_OK = 1; break; }
5    if(exists_OK == 1) // element present in A
6      if(result == 1) // search returned 1 (found)
7        return 1; // postcondition verified
8      else
9        return 0; // postcondition fails
10   for(j = 0, forall_OK = 1; j < 10; j++)
11     if(!(A[j] != elt)) { forall_OK = 0; break; }
12   if(forall_OK == 1) // element not present in A
13     if(result == 0) // search returned 0 (not found)
14       return 1; // postcondition verified
15     else
16       return 0; // postcondition fails
17 }
18 void oracle_search(int elt, int Pre_elt, int result){
19   int postcond_status = search_postcond(elt, result);
20   if(postcond_status)       // postcondition verified
21     {pathcrawler_verdict_success();return;}
22   else                      // postcondition fails
23     {pathcrawler_verdict_failure();return;}
24 }
```

Figure 8: File `oracle_search.c` containing the oracle based on the postcondition of Fig. 1 translated from E-ACSL into C by E-ACSL2C

invariants and pre/postconditions in first-order logic. This allows us to use the same specification for various static analyses (abstract interpretation, deductive verification, etc.).

Cheon [8] proposes a behavioral specification language for Java, called JML. Inspired from JML, ACSL is first aimed at static verification, whereas JML was originally used for dynamic and later for static verification. JML lets the user choose which kind of integers and attached semantics they use through various modes from mathematical integers to checking overflows in the specification [**?**], whereas ACSL and E-ACSL support only mathematical integers, and can encode the other modes.

Other examples of executable specification languages are SPEC# [1] and the closely related Code Contracts for .NET [14]. Here it is possible to express the specification in the same language as the code. If using the same language may seem easier to the programmer, it has some drawbacks. Indeed, some conditions, like integer overflows, are difficult to express, because the underlying programming language does not allow to express it. As discussed earlier, those particular cases can be expressed in ACSL, thus they constitute challenges that E-ACSL and E-ACSL2C address.

ALFA [9] is a programming and specification language, derived from Ada, which also aims at filling the gap between static and dynamic analysis. It is designed and developed in the Hi-Lite[6] project. In addition to a static verification tool, ALFA comes with a compiler that permits dynamic verification of conditions. As JML, ALFA supports several arithmetic modes.

A major difference between our work and the above specification languages for other target languages, is the memory observation. Indeed Java, Ada, and C# (except for the unmanaged code, which cannot be verified with Spec# or Code Contracts) enforce strong properties of its memory models which ease design of languages and development of tools. Therefore, E-ACSL2C must embed a complex runtime memory observation system which is usually not required in other languages.

Thus E-ACSL stands out from all previous approaches for the C language because of its greater expressiveness and the possibility of using the same specification for both dynamic analysis and advanced static analysis.

[6] `http://www.open-do.org/projects/hi-lite`

6. CONCLUSION

We proposed an expressive formal specification language E-ACSL for C supported by static analyzers of FRAMA-C, that can be automatically translated into executable C specifications. We emphasized particular issues related to specific keywords, quantifications, mathematical integers, memory-related annotations and undefined terms. We presented our solutions for these issues, most of which are already implemented and available in the released (or current SVN) version of the E-ACSL2C translator. Moreover, we identified potential disadvantages in the current translation and proposed several improvements.

The specifications translated by E-ACSL2C are usable for runtime assertion checking and treatable by a testing tool for C programs. We illustrated how a translated specification is automatically treated in PATHCRAWLER, that avoids to manually rewrite the specification in another format for a testing tool. The experiments with the combined method SANTE in [7] showed that SANTE is more precise than a static analyzer and more efficient in terms of time and number of detected bugs than test generation alone. The present work offers a common specification language for static and dynamic analysis tools and will help to develop and better automatize their combinations. Future work includes finalizing the development of E-ACSL2C, its integration into the SANTE tool and further exploration of combined techniques for software verification.

Acknowledgments. The authors thank the members of the Frama-C and PathCrawler teams for useful discussions and support. Special thanks to Johannes Kanig, Yannick Moy and the anonymous referees for useful remarks and suggestions of improvement.

7. REFERENCES

[1] M. Barnett, K. R. M. Leino, and W. Schulte. The Spec# programming system: An overview. In *the Construction and Analysis of Safe, Secure, and Interoperable Smart Devices, International Workshop (CASSIS 2004)*, volume 3362 of *LNCS*, pages 49–69. Springer, 2004.

[2] P. Baudin, J. C. Filliâtre, T. Hubert, C. Marché, B. Monate, Y. Moy, and V. Prevosto. *ACSL: ANSI/ISO C Specification Language, v1.6*, Sept. 2012. URL: http://frama-c.com/acsl.html.

[3] P. Baudin, A. Pacalet, J. Raguideau, D. Schoen, and N. Williams. CAVEAT: a tool for software validation. In *the 2002 International Conference on Dependable Systems and Networks (DSN 2002)*, page 537. IEEE Computer Society, 2002.

[4] B. Botella, M. Delahaye, S. Hong-Tuan-Ha, N. Kosmatov, P. Mouy, M. Roger, and N. Williams. Automating structural testing of C programs: Experience with PathCrawler. In *the 4th International Workshop on Automation of Software Test (AST 2009)*, pages 70–78. IEEE Computer Society, 2009.

[5] G. Canet, P. Cuoq, and B. Monate. A value analysis for C programs. In *the Ninth IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM 2009)*, pages 123–124. IEEE Computer Society, 2009.

[6] P. Chalin. Engineering a sound assertion semantics for the verifying compiler. *IEEE Transactions on Software Engineering*, 36:275–287, 2010.

[7] O. Chebaro, N. Kosmatov, A. Giorgetti, and J. Julliand. Program slicing enhances a verification technique combining static and dynamic analysis. In *the ACM Symposium on Applied Computing (SAC 2012)*, pages 1284–1291. ACM, 2012.

[8] Y. Cheon. *A Runtime Assertion Checker for the Java Modeling Language*. Iowa State Univ., 2003. URL: http://cs.iastate.edu/~leavens/JML/Relatedpapers/index.html.

[9] C. Comar, J. Kanig, and Y. Moy. Integrating formal program verification with testing. In *the Embedded Real-Time Software and Systems Congress (ERTS 2012)*, 2012.

[10] L. Correnson and J. Signoles. Combining Analyses for C Program Verification. In *the 17th International Workshop on Formal Methods for Industrial Critical Systems (FMICS 2012)*, volume 7437 of *LNCS*, pages 108–130. Springer, 2012.

[11] C. Csallner and Y. Smaragdakis. Check'n'crash: combining static checking and testing. In *the 27th International Conference on Software Engineering (ICSE 2005)*, pages 422–431. ACM, 2005.

[12] P. Cuoq, F. Kirchner, N. Kosmatov, V. Prevosto, J. Signoles, and B. Yakobowski. Frama-C, a program analysis perspective. In *the 10th International Conference on Software Engineering and Formal Methods (SEFM 2012)*, volume 7504 of *LNCS*, pages 233–247. Springer, 2012.

[13] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao. The Daikon system for dynamic detection of likely invariants. *Sci. Comput. Program.*, 69(1–3):35–45, 2007.

[14] M. Fähndrich, M. Barnett, and F. Logozzo. Embedded contract languages. In *the 2010 ACM Symposium on Applied Computing (SAC 2010)*, pages 2103–2110. ACM, 2010.

[15] J.-C. Filliâtre and C. Marché. The Why/Krakatoa/Caduceus platform for deductive program verification. In *the 19th International Conference on Computer Aided Verification (CAV 2007)*, volume 4590 of *LNCS*, pages 173–177. Springer, 2007.

[16] B. S. Gulavani, T. A. Henzinger, Y. Kannan, A. V. Nori, and S. K. Rajamani. SYNERGY: a new algorithm for property checking. In *the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2006)*, pages 117–127. ACM, 2006.

[17] B. Hackett, M. Das, D. Wang, and Z. Yang. Modular checking for buffer overflows in the large. In *28th International Conference on Software Engineering (ICSE 2006)*, pages 232–241. ACM, 2006.

[18] B. Konikowska, A. Tarlecki, and A. Blikle. A three-valued logic for software specification and validation. *Fundam. Inform.*, pages 411–453, 1991.

[19] B. Meyer. *Object-Oriented Software Construction*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1988.

[20] G. C. Necula, S. McPeak, and W. Weimer. CCured: type-safe retrofitting of legacy code. In *29th ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages (POPL 2002)*, pages 128–139. ACM, 2002.

[21] J. Signoles. *E-ACSL: Executable ANSI/ISO C Specification Language*, Jan. 2012. URL: http://frama-c.com/download/e-acsl/e-acsl.pdf.

[22] J. Signoles. *E-ACSL Version 1.5-4. Implementation in Frama-C Plug-in E-ACSL version 0.1*, Jan. 2012. URL: http://frama-c.com/download/e-acsl/e-acsl-implementation.pdf.

[23] Y. Smaragdakis and C. Csallner. Combining static and dynamic reasoning for bug detection. In *the First International Conference on Tests and Proofs (TAP 2007)*, volume 4454 of *LNCS*, pages 1–16. Springer, 2007.