Instrumentation of Annotated C Programs for Test Generation

Guillaume Petiot*†, Bernard Botella*, Jacques Julliand†, Nikolai Kosmatov* and Julien Signoles*
* CEA, LIST, Software Reliability Laboratory, PC 174, 91191 Gif-sur-Yvette France
Email: `firstname.lastname@cea.fr`
† FEMTO-ST/DISC, University of Franche-Comté, 25030 Besançon Cedex France
Email: `firstname.lastname@femto-st.fr`

***Abstract*—Software verification and validation often rely on formal specifications that encode desired program properties. Recent research proposed a combined verification approach in which a program can be incrementally verified using alternatively deductive verification and testing. Both techniques should use the same specification expressed in a unique specification language. This paper addresses this problem within the FRAMA-C framework for analysis of C programs, that offers ACSL as a common specification language. We provide a formal description of an automatic translation of ACSL annotations into C code that can be used by a test generation tool either to trigger and detect specification failures, or to gain confidence, or, under some assumptions, even to confirm that the code is in conformity with respect to the annotations. We implement the proposed specification translation in a combined verification tool STADY. Our initial experiments suggest that the proposed support for a common specification language can be very helpful for combined static-dynamic analyses.**

Keywords: specification language, C program instrumentation, test generation, deductive verification, combinations of static and dynamic analyses, Frama-C

I. MOTIVATION

Software verification and validation usually rely on a formal specification language in which properties are encoded. In a scenario where a program is verified using a combination of verification tools, these tools must use the same specification written in a unique language. Our concern is combining deductive verification and testing within the FRAMA-C framework for analysis of C programs [1] using ACSL [2], a common specification language offered by FRAMA-C.

In the current deductive verification practice, when a deductive verification tool does not manage to prove an annotated program, it usually cannot provide a comprehensive reason of the proof failure. In those cases, the validation engineer cannot easily see if some specific specification clauses are wrong, or if some clauses are not strong enough, or if the C code under verification is incorrect, or if additional assertions or lemmas, or a greater timeout are necessary for the deductive verification tool to finish the proof. Human audit of the specification and the code in order to understand and fix the issue may take lots of time and effort, and requires highly experienced validation engineers.

Various verification scenarios that advantageously combine deductive verification and testing to verify that a program respects its specification have been proposed in [3]. They suggest an approach in which the validation engineer incrementally formalizes the specification and performs the proof combining deductive verification and testing. Thus, one scenario suggests that the engineer can validate that the program respects its (partial) specification during the specification process. Indeed, the specification task aiming at writing function contracts (with pre- and postconditions), loop invariants and variants, and additional assertions necessary to prove a program, is tedious and entails important risks of errors. So the user may want to validate (a part of) the postcondition assuming a precondition and before providing all loop invariants, contracts of called functions etc., without which the proof cannot succeed. At the same time, it is perfectly possible to test whether the program respects its partial specification. If test generation reveals a counter-example, the user can analyze it step-by-step and understand from its knowledge of the informal specification or program requirements if there is an error in the program or in the specification, and fix it. If test generation does not reveal any non-conformity, the user acquires some confidence that the program respects the formal contract, and is encouraged to pursue the development of the formal specification.

Another scenario proposes incremental verification of loops. After writing a loop contract (including loop invariants) for each loop, the validation engineer can run a deductive verification tool. As long as the loop contract is proven, the specification effort can be continued. However, when the proof of the loop contract fails, the user does not know if it is too strong, or too weak, or if there is an error in the program code, etc. Here again, the user can run test generation to check the loop contract and benefit from its feedback. If test generation reveals a counter-example where the loop contract fails, it helps the user to understand the reason of the proof failure and fix the specification or the code. If test execution does not reveal any error, the user can think that the loop contract is too weak to prove the program and continues enforcing it. In some cases and under certain assumptions, the absence of counter-examples confirms that the program respects its specification. Notice that the objective of this approach is certainly not *to fit the specification to (potentially erroneous) code,* but *to help the validation engineer to identify the problem (in the specification or in the code)* with a counter-example [3].

This combined verification approach requires that the test generation tool embedded in FRAMA-C correctly handle ACSL annotations. For this purpose, we propose a formally described translation of basic ACSL annotations into instru-

Work partially funded by EU FP7 (project STANCE, grant 317753).

mented C code. The objectives of such a formal description of the translation are two-fold. First, instrumentation based translation of specification into executable code requires significant effort to treat an expressive specification language, and merits to be strictly formalized to allow reusability of the results. Second, a formal description of translation rules allows to thoroughly check its correctness, and in our case helped us to find a couple of subtle errors that could have remained unnoticed without this formalization. Moreover, since the absence of counter-examples can be considered in some cases as a proof of correctness of a program w.r.t. its specification, a correct translation of the specification into executable code is crucial to ensure the correctness of test generation results (the underlying deductive verification and testing tools are usually assumed to be correct). Therefore, a formalization of the translation is a mandatory step to formally establish its correctness, and constitutes a first step to the design of sound combined static-dynamic analyses in the FRAMA-C framework.

Context. FRAMA-C [1] is a platform dedicated to analysis of C programs that includes various source code analyzers as separate plugins such as WP performing weakest-precondition calculus for deductive verification, VALUE performing value analysis by abstract interpretation, etc. FRAMA-C supports ACSL (ANSI C Specification Language) [2], a behavioral specification language allowing to express properties over C programs. In addition to providing formal specifications for C programs, ACSL annotations play a central role in communication between plugins: any analyzer can add annotations to be verified by other ones and notify other plugins about results of the analysis it performed by changing an annotation status. The status can indicate that the annotation is valid, valid under conditions, invalid or undetermined, and which analyzer established this result [4]. For combinations with dynamic analysis, we consider its executable subset E-ACSL [5]. E-ACSL can express function contracts (with pre/postconditions, guarded behaviors, completeness and disjointness of behaviors), assertions and loop contracts (with loop variants and invariants). It includes quantifications over bounded intervals of integers, mathematical integers and memory-related constructs (e.g. on validity and initialization).

PATHCRAWLER [8] is a structural (also known as *concolic* [9]) test generator for C programs, combining *conc*rete and symb*olic* execution. PATHCRAWLER is based on COLIBRI, a constraint solver implementing advanced features such as floating-point and modular integer arithmetics support. PATHCRAWLER provides coverage strategies like *k-paths* (feasible paths with at most k consecutive loop iterations) and *all-paths* (all feasible paths without any limitation on loop iterations). PATHCRAWLER is *sound*, meaning that each test case activates the test objective for which it was generated. This is verified by concrete execution. Unlike some other concolic tools, PATHCRAWLER does not approximate path constraints, and is *complete* in the following sense: when the tool manages to explore all feasible paths of the program, all features of the program are supported by the tool and constraint solving terminates for all paths, the absence of a test for some test objective means that this test objective is infeasible.

Contributions. We have implemented STADY, a tool that bridges the gap between STAtic (e.g. WP, VALUE) and DYnamic (e.g. PATHCRAWLER) analyzers of FRAMA-C. STADY translates E-ACSL annotations into executable instrumented C code in order to support test generation for C programs annotated in E-ACSL. It is implemented as a FRAMA-C plugin and uses PATHCRAWLER to generate test cases for the resulting instrumented C program. Since the analyzers rely on E-ACSL annotations to communicate with each other, performing a translation from E-ACSL into C is the most appropriate way to combine a test generator like PATHCRAWLER with other FRAMA-C analyzers. The target language of the translation is C because PATHCRAWLER is naturally capable to check expected properties expressed in this language (assertions, postconditions,...) and offers an original dedicated support for the precondition of the function under test (FUT) written in C [10].

The main contributions of this paper include:

- a formal description of the translation rules used to derive executable C code from an E-ACSL specification for applying test generation in combination with deductive verification. This formal description is given for C and E-ACSL, but its main ideas can be applicable for other specification formalisms and imperative programming language kernels;
- a brief presentation of the implementation of these rules in STADY as part of the FRAMA-C verification framework;
- a comparison of instrumentation for test generation and runtime assertion checking (RAC);
- an experiment report showing the efficiency of the combination of deductive verification and testing in STADY, in particular, for finding counter-examples for C programs according to E-ACSL specifications.

The paper is organized as follows. Sec. II presents the E-ACSL language (Sec. II-A), the treatment of mathematical integers (Sec. II-B), the running example (Sec. II-C) and gives general insights about the instrumentation (Sec. II-D). Next, the rules of the instrumentation are detailed for terms (Sec. III), for predicates (Sec. IV) and for annotations (Sec. V). Sec. VI compares some translation issues for test generation and RAC, and Sec. VII shows our experiments with STADY. Sec. VIII presents the related work, and finally, Sec. IX concludes.

II. INSTRUMENTATION PROCESS

This section presents the instrumentation process for C functions annotated in ACSL that we support during test generation for the annotated code. It will be convenient to consider that specific labels are introduced for each statement and annotation inside a function body, as well as for the beginning and the end of each function body and loop body. (Strictly speaking, labels cannot be put just before declarations or "`}`" in C, but this can be easily fixed by adding a skip instruction "`;`" after such labels.) We assume that functions respect the normal form defined by the syntactic entity *function* from the grammar in Fig. 1. In this figure, the superscript "*as* X" means that any occurrence of X in the analysis of the current rule should be replaced with the string referred by the superscripted syntactic entity. For example, if `foo` is the name *id* of the analyzed function in the rule $function$, labels $\texttt{Beg}_f$ and $\texttt{End}_f$ in this function's body are replaced by

```
function     ::=  /*@ (requires predicate;)*
                      (typically predicate;)*
                      (ensures predicate;)*   */
                  type^as T  id^as f  (params) {
                  Beg_f : T res;
                  decl*    stmt*
                  End_f : return res; }
decl         ::=  type id;
stmt         ::=  label : assert predicate;
              |   label^as l : (/*@ loop_annot */)^?
                        while ( bool_expr ) {
                          BegIter_l :    stmt*
                          EndIter_l : }
              |   (label :)^? not_while_stmt
loop_annot   ::=  (loop invariant predicate;)*
                  (loop variant term;)^?
```

Fig. 1. Grammar of an annotated C function

$\texttt{Beg}_{\texttt{foo}}$ and $\texttt{End}_{\texttt{foo}}$ providing unique labels for the beginning and the end of the function body. Array and pointer accesses are supposed to be written as `*(p+i)`. The terminal symbols are presented in a `typewriter font`. Underlined non-terminal symbols are not detailed because they are part of the C or ACSL languages. $Terms$ and $predicates$ are ACSL expressions, most of them are described in Sec. III and Sec. IV (see the ACSL documentation [2] for more detail).

A. *Overview of the* E-ACSL *Specification Language*

The specification language E-ACSL is a strict executable subset of ACSL, which is a behavioral specification language implemented in FRAMA-C. 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, 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 E-ACSL facilitates combinations of static and dynamic analyses.

E-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 talk about the result of a function, while **`\valid`** is a builtin predicate stating that its argument is a valid pointer. Quantifications are bounded by constraints to finite intervals of integers in order to remain executable. An EIFFEL-like contract [11] may be associated to each function in order to specify its pre- and postconditions. These contracts may be split into several named guarded behaviors for which the users may require completeness and/or disjointness. Assertions, loop invariants and loop variants may also be associated to statements. We now focus on two of the most important design choices of the language: integers and undefinedness.

1) Integers: In addition to all machine types, E-ACSL terms also include mathematical integers of type **`integer`**: integer constants and operators, as well as logic variables are of this type. Integer arithmetics is unbounded and never overflows. E-ACSL holds a small subtyping system to automatically coerce C integral types into mathematical integers.

```
1 if( x > 0 ) {
2   /*@ assert x+1 > 0; */ // never fails in unbounded arithm.
3   fassert(x+1 > 0);      // may fail in modular arithmetics
4 }
```

Fig. 3. Properties over integers: naive instrumentation gives false positive

For instance, if `x` is a C variable of type **`int`**, `x+1` and `1` are of type **`integer`** and a coercion from **`int`** to **`integer`** is introduced when typing `x` in this context. 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)(INT_MAX + 1)` is equal to `INT_MIN`, 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 INT_MIN <= x+y <= INT_MAX;*/` specifies in the easiest way that `x+y` must not overflow. Unless otherwise stated, "integer" will refer below to "unbounded integer".

2) Undefinedness: E-ACSL is executable. However, evaluation of undefined terms like `1/0` is not possible. To solve this issue, E-ACSL follows Chalin's Runtime Assertion Checking semantics [12] by stating that semantics of such terms is "undefined": E-ACSL uses a 3-valued logic [13] like SPARK2014 [14] or JML [15]. It is then the responsibility of the tools interpreting E-ACSL to ensure that an undefined term is never evaluated. 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).

B. *Handling Unbounded Integers*

Fig. 2 and Fig. 3 exhibit two examples where naive translation of annotations with mathematical integers leads to unsoundness. Let `x` be of type **`int`**. The assertion at line 1 of Fig. 2 is obviously false when `x = INT_MAX`. The naive translation of this assertion (line 3) uses machine integers with modular arithmetics (we assume a 32-bit architecture), so `x+1` remains less or equal to `INT_MAX` for any value of `x`, making it impossible to find the assertion violation. The correct translation for this annotation (sketched at lines 6–8) maintains the semantics of unbounded integer arithmetics using an external unbounded integer library (we use GMP, the GNU Multi-Precision library) to represent values that may overflow otherwise (like `INT_MAX+1` here). It creates and initializes necessary variables for unbounded integers, then computes and compares the values as unbounded integers. The second example (Fig. 3) defines an ACSL assertion (line 2) that is always correct here: for any positive integer `x`, its successor also is positive. The naive instrumentation (line 4) will exhibit an error for `x = INT_MAX`: due to modular arithmetics, `x+1` overflows and becomes negative, violating the assertion. A correct translation using unbounded integers (not detailed here) maintains the expected behavior, so that the assertion remains valid for any positive integer `x`. Using the naive translation for these two examples would result in a false negative in the first

```
1  //@ assert x+1 <= INT_MAX;  // fails with x = INT_MAX since INT_MAX + 1 does not overflow in ACSL
2
3  fassert(x+1 <= 2147483647); // naive instrumentation: never fails in modular arithmetics
4
5  // correct instrumentation with unbounded integers:
6  Z_t var_0; Z_init(var_0); Z_set(var_0, x); Z_t var_1; Z_init(var_1); Z_set(var_1, 1);
7  Z_t var_2; Z_init(var_2); Z_add(var_2, var_0, var_1); Z_clear(var_0); Z_clear(var_1);
8  Z_t var_3; Z_init(var_3); Z_set(var_3, 2147483647); int var_4 = Z_le(var_2, var_3);
9  Z_clear(var_2); Z_clear(var_3); fassert(var_4);
10
11 // correct instrumentation in abbreviated notation:
12 ☐var_0 = x; ☐var_1 = 1; ☐var_2 = var_0⊠ + var_1⊠; ☐var_3 = 2147483647; int var_4 = var_2⊠ <= var_3⊠; fassert(var_4);
```

Fig. 2. Properties over integers: naive instrumentation gives false negative

```
1  /*@ requires 0 <= n;
2      requires \valid(t+(0..n-1));
3      typically n <= 6;
4      ensures \result != 0 <==> \exists integer i; 0<=i<\old(n)
5                                   && \old(*(t+i))==\old(v); */
6  int is_present(int* t, int n, int v) {
7   Beg_is_present : int res = 0, i = 0;
8   l_0:
9    /*@ loop invariant 0 <= i && i <= n;
10       loop variant n - i; */
11   while (i < n) {
12     BegIter_l0 :
13     if(*(t+i) == v) { res = 1; break; }
14     i++;
15     EndIter_l0 :
16   }
17  End_is_present : return res;
18 }
```

Fig. 4. Annotated C function deciding if `v` is present in array `t` of size `n`

case, and in a false positive in the second case, undermining the tool's correctness and precision. The translation rules for E-ACSL constructs presented below respect unbounded integer semantics of E-ACSL mathematical integers and assume the usage of an external library as illustrated in Fig. 2. To simplify the notation of code insertions, we will use the abbreviated notation $^{\square}$`var` to indicate that the variable `var` must be declared and allocated (with `Z_t var; Z_init(var);`) at the beginning of the inserted code, and the notation `var`$^{\boxtimes}$ to indicate that the variable `var` must be de-allocated (with `Z_clear(var);`) at the end of the inserted code. We will also use underlined code fragments to indicate that the corresponding operation (assignment, comparison, ...) should be translated using corresponding functions from the unbounded integer library. So line 12 of Fig. 2 illustrates abbreviated notation of instrumentation for lines 6–9. As mentioned in Sec. II-A1, type coersions are automatically made explicit in the annotations in FRAMA-C, so `x+1` in line 1 of Fig. 2 becomes `(integer)x+1`.

C. Running Example

We present in Fig. 4 an example of FUT normalized according to the grammar of Fig. 1. It returns 1 when a given value is present in a given array, or 0 otherwise. The instrumented program obtained after translation of annotations of this function is presented (using abbreviated notation) in Fig. 5. The generated precondition function is defined at lines 1–6 (Fig. 5), it returns a nonzero value when the precondition holds. The **requires** clause in line 1 (Fig. 4) stating that the array size `n` is positive is translated as the condition in line 3 (Fig. 5). The **requires** clause in line 2 (Fig. 4) states that `(t+0)`,...,`(t+(n-1))` are valid pointers, and leads to the condition in line 4 (Fig. 5). The **typically** clause in line 3 of Fig. 4 (translated as line 5 of Fig. 5), is an extension of ACSL defining a precondition considered only for testing. It strengthens the precondition to restrict the (potentially too big) number of paths to be explored by test generation to user-controlled partial coverage. Here it bounds the state space of `n` (and thus the size of `t`) to $[0, 6]$ (it can be seen as a domain *finitization* [16]). The loop invariant in line 9 of Fig. 4 is translated as lines 13–14 of Fig. 5 to check that the invariant holds before the first loop iteration, and lines 21–22 of Fig. 5 to check the preservation of the invariant by any iteration. (An additional loop invariant `\forall integer k; 0<=k<i ==> \old(*(t+k))!=v;` necessary to formally prove the postcondition using deductive verification was not included in this simplified example). The loop variant line 10 in Fig. 4 is translated as line 15 of Fig. 5 to check that the variant is positive or zero before the loop, line 18 of Fig. 5 to back up the value of the variant at the beginning of the loop, and lines 23–25 of Fig. 5 to check that it strictly decreases but remains positive or zero, thus ensuring the termination of the loop. The postcondition at lines 4-5 of Fig. 4 is translated as lines 28–32 of Fig. 5, it states that `\result` is non zero if and only if there exists an element of `t` equal to `v`. The values of formal parameters `t`, `n` and `v` are saved in lines 10–11 (Fig. 5), a new array `old_val_t` saves the old values contained in `t`, it is allocated line 10, filled line 11 and deallocated line 33.

D. Principles of the Instrumentation

Let us describe the principles of instrumentation for an annotated function f respecting the grammar of Fig. 1. First, each input value x (a formal parameter x in $params$ or a global variable x) of type T is stored as `T old_x = x;` at the beginning of the instrumented FUT, i.e. at label Beg_f, in the $decl^*$ section. For an input array (or pointer) `x`, the values are stored in the dynamically allocated array `old_val_x` whose size is inferred from the `\valid` clause. We also generate an additional function named `f_precond` that is used to check the precondition of the FUT. For the FUT we ensure that the precondition is assumed by inserting `fassume(f_precond(x1, ..., xn));` at label Beg_f (cf line 11 in Fig. 5).

Second, any ACSL annotation of the form `kwd w` (where keyword `kwd` belongs to {**assert**, **requires**, **typically**, **ensures**, **loop variant**, **loop invariant**} and w is a predicate or a term) is translated. Some other ACSL constructs are not detailed here because they can be obtained from the described ACSL fragment. For example, behaviors can be rewritten as implications in **ensures** annotations, statement contracts can be rewritten as implicative **assert**'s and global (resp., loop or statement) **assigns** clauses can be rewritten as postconditions (resp., loop invariants or assertions) checking the non-modification of some variables.

Each pair $(label, annotation)$ is translated into a sequence of *code insertions* $(l_1, c_1) \cdot (l_2, c_2) \cdot \ldots \cdot (l_n, c_n)$, that represents

```
int is_present_precond(int* t, int n, int v) {
 Beg_is_present_precond :
  □var_0 = 0; □var_1 = n; int var_2 = var_0⊠ <= var_1⊠; if (!var_2) return 0;
  if (!(fvalidr(t,0,(n-1)))) return 0;
  □var_3 = n; □var_4 = 6; int var_5 = var_3⊠ <= var_4⊠; if (!var_5) return 0;
  return 1; }

int is_present(int* t, int n, int v) {
 Beg_is_present :
  int *old_t = t, *old_val_t = malloc(((n-1)+1)*sizeof(int)), old_n = n, old_v = v, res = 0, i = 0, iter_t;
  for(iter_t = 0; iter_t < n; iter_t++) *(old_val_t+iter_t) = *(t+iter_t); fassume(is_present_precond(t, n, v));
 l_0 :
  □var_0 = 0; □var_1 = i; int var_2 = var_0⊠ <= var_1⊠; int var_3 = var_2;
  if(var_3) { □var_4 = i; □var_5 = n; int var_6 = var_4⊠ <= var_5⊠; var_3 = var_6; } fassert(var_3);
  □var_7 = n; □var_8 = i; □var_9 = var_7⊠ - var_8⊠; int var_10 = 0 <= var_9⊠; fassert(var_10);
  while(i < n) {
   BegIter_l_0 :
    □var_11 = n; □var_12 = i; □var_13 = var_11⊠ - var_12⊠; □old_variant = var_13⊠;
    if(*(t+i) == v) { res = 1; break; } i++;
   EndIter_l_0 :
    □var_14 = 0; □var_15 = i; int var_16 = var_14⊠ <= var_15⊠; int var_17 = var_16;
    if(var_17) { □var_18 = i; □var_19 = n; int var_20 = var_18⊠ <= var_19⊠; var_17 = var_20; } fassert(var_17);
    □var_21 = n; □var_22 = i; □var_23 = var_21⊠ - var_22⊠; int var_24 = 0 <= var_23⊠; int var_25 = var_24;
    if(var_25) { □var_26 = n; □var_27 = i; □var_28 = var_26⊠ - var_27⊠; int var_29 = var_28⊠ < old_variant⊠; var_25 = var_29; }
    fassert(var_25);
  }
 End_is_present :
  □var_30 = 0; □var_31 = res; int var_32 = var_30⊠ != var_31⊠; □var_33 = 0; □var_34 = old_n;
  int var_35 = 0;
  for(□i_0 = var_33⊠; i_0 < var_34⊠ && !var_35; i_0++⊠)
    { □var_36 = i_0; int var_37 = var_36⊠; var_35 = *(old_val_t + var_37) == old_value; }
  fassert((!var_32 || var_35) && (!var_35 || var_32));
  free(old_val_t); return res; }
```

Fig. 5. Instrumented version of program of Fig. 4

a list of fragments of C programs $c_1, c_2, \ldots, c_n$ where the fragment c_i will be inserted into the instrumented program at label l_i. Program fragments c_i are parts of a correct program that might be incomplete if taken separately since a syntactically complete statement can be split into several insertions. When there are several fragments to insert in the same location l, they are inserted according to their order in the list. ACSL annotations are translated separately, and the resulting sequences of insertions are treated respecting the order of annotations in the source program.

Translation of annotations is defined by the rules in Sec. V. It requires translating ACSL terms and predicates. These transformations are described by the rules in Sec. III and Sec. IV. In these rules we use the following notation:

- p, p_1, p_2 and p_3 are E-ACSL predicates;
- t, t_1, t_2 and t_3 are E-ACSL terms and w is either a predicate or a term;
- c, c_1, c_2 and c_3 are fragments of C programs;
- e, e_1, e_2 and e_3 are C expressions;
- $l, l_1, l_2, l_3, Beg_f, End_f, BegIter_l$ and $EndIter_l$ are program labels;
- i is an identifier of a bounded variable in an ACSL predicate and an iterating counter in a C program, x is an identifier of a C variable;
- I, I_1, I_2 and I_3 are lists of code insertions (l_i, c_i).

We denote by `res`, `var_n`, `i_n`, `old_x`, `old_val_x` and `old_variant`, fresh variables, that is, identifiers different from all other identifiers of the instrumented program. When we use several times a rule introducing a fresh variable, all occurrences must be different. It can be easily implemented incrementing the number `n` in `var_n` or `i_n` and replacing `x` in `old_x` or `old_val_x` by the names of parameters of f or global variables.

For convenience, we assume that all bounded variables in annotations and all program variables are different from one another. In particular, we can translate bounded variables into C without renaming.

III. Term Translation

Let us denote $(l, t : T) \mapsto ((l_1, c_1) \cdot (l_2, c_2) \cdot \ldots \cdot (l_n, c_n), e)$ an instance of the partial function τ mapping an E-ACSL term t of type T at label l to the pair (I, e) where I is a list of code insertions (l_i, c_i), e is a C expression, and T can be $\mathbb{Z}$ (integer) or $ctype$. We only consider the following types as included in $ctype$: ptr, that can be a pointer of any type, and **`int`** that is the regular C type. Other integral C types such as **`char`**, **`long`**, **`unsigned`** could be supported in the same way, but are not considered for simplicity and readability purposes. The expression e is pure (i.e. has no side effects) and evaluates the value of the term at the given point. The evaluation of e often requires additional computations, that are performed by the inserted program fragments resulting from the predicate and term translation. For example, a quantified term requires the computation of a value throughout a **`for`** loop (see Fig. 9). In this case, the second element of the returned pair, e, is the value of a variable computed by the loop. When a term t can be directly translated without additional C code, the sequence of code insertions is empty and denoted $\emptyset$. For a term $t : \mathbb{Z}$, the translation result e is always a variable of type `Z_t` (that justifies de-allocation `e`⊠ e.g. in Fig. 7).

Fig. 6 describes the rules for identifiers (τ-VAR), for the **`\result`** term (τ-RES) and constants (τ-CONST). Two rules

$$\tau\text{-OLD}\ \frac{}{(l, \texttt{\textbackslash old(x)} : ctype) \mapsto (\emptyset, \texttt{old\_x})} \quad \tau\text{-VAR}\ \frac{}{(l, \texttt{x} : ctype) \mapsto (\emptyset, \texttt{x})} \quad \tau\text{-OLD-VAL}\ \frac{(l, t : int) \mapsto (I, e)}{(l, \texttt{\textbackslash old(*(x+t))} : ctype) \mapsto (I, \texttt{*(old\_val\_x+e)})}$$

$$\tau\text{-RES}\ \frac{}{(l, \texttt{\textbackslash result} : ctype) \mapsto (\emptyset, \texttt{res})} \quad \tau\text{-CONST}\ \frac{}{(l, \texttt{cst}:\mathbb{Z}) \mapsto ((l, \underline{{}^{\square}\texttt{var\_n = cst}}), \texttt{var\_n})}$$

Fig. 6. Translation rules for constants, identifiers, and **`\old`** terms

are considered for the ACSL construct **`\old`**. Applied to an identifier x, it is translated as the fresh variable `old_x` storing the value at the entry of the function. Applied to a memory access (rule τ-OLD-VAL), we use the dynamically allocated array `old_val_x` that memorizes the elements of an array `x` at the entry of the function (cf Sec. II-D). The special term **`\result`** translated by the rule τ-RES denotes the return value of the function in ACSL written into the fresh variable `res`. That variable is unique after the normalization of the abstract syntax tree (AST) by FRAMA-C. The rule τ-CONST states that an integer variable is defined to store the integer constant.

Fig. 7 details the rules for coercions, from integer to C type and from C type to integer.

Fig. 8 details the rules for unary operations (τ-UNOP$_*$), binary operations (τ-BINOP$_*$) and the ternary condition on terms (τ-IF). τ-UNOP$_1$ deals with pointer indirection ($\star$). In τ-UNOP$_2$, the logical not (`!`) does not involve integers. In τ-UNOP$_3$, *op* is the unary minus (`-`) or the bitwise complement ($\sim$) and is an operation from and to integer. In τ-BINOP$_1$, the left operand is a pointer so *op* must be `+` or `-`, whereas the right operand is an int (if it is an integer, it will be coerced to int using the rule τ-COERCE$_1$ for pointer arithmetics). In τ-BINOP$_2$, *op* is a comparison operator over integer values and the result of the comparison is stored in an **`int`**. In τ-BINOP$_3$, *op* is any of the arithmetic operators: `+`, `-`, `/`, `%`, `<<`, `>>`, `|`, `&`, `^` over integers. In the rule τ-IF, the evaluation of t_2 and t_3 are in conditional branches: only one of them is computed, depending on the evaluation of t_1.

Fig. 9 presents the rules for translating the builtin logic functions **`\sum`** (τ-SUM) and **`\numof`** (τ-NUMOF). The rule for function **`\product`** is similar to the rule τ-SUM. The rule τ-SUM over integers initializes a fresh integer variable `var_n`, initialized to 0, and increments its value with the value of the **`\lambda`**-term `t3` at each iteration. The rule τ-NUMOF also initializes a fresh integer variable `var_n` to 0, but increments it only when the (non-integer) **`\lambda`**-term `t3` is evaluated to a non-null expression.

IV. PREDICATE TRANSLATION

Similarly to the translation function τ for terms (see Sec. III), the translation function for predicates, denoted π, is defined as a partial function mapping a label and an ACSL predicate to a sequence of code insertions (l_i, c_i) and a C expression $e \in \{0, 1\}$. When a predicate p can be directly translated without additional C code, the sequence of code insertions is empty and denoted $\emptyset$.

We define in Fig. 10, Fig. 11 and Fig. 12 the transformation rules for the main ACSL predicates we handle.

Fig. 10 details the translation rules for the simplest predicates of the ACSL language. The rules π-TRUE and π-FALSE state that $true$ (resp. $false$) are translated into 1 (resp. 0). The rules π-EQUIV and π-NOT are compositional: the subpredicates are translated recursively, then the result of the translation of the predicate is rebuilt from the values of the subpredicates. The rules π-AND, π-OR and π-IMPL are reflective of the laziness of the ACSL semantics of those operators: the first operand is always evaluated (in I_1) but the second one is only evaluated (in I_2) when necessary. The rule The rule π-IF is the counterpart of τ-IF (cf Sec. III) for predicates. In our running example of Fig. 4, translating the postcondition **`\result`** `!= 0 <==>` **`\exists integer`** `i; 0<=i<n && *(t+i)==v` requires the translation of the predicates **`\result`** `!= 0` and **`\exists`**`(...)` that are respectively translated as `var_32` and `var_35`. These two predicates are put together to build the translation of the composed predicate, that is, according to π-EQUIV, `(!var_32 || var_35) && (!var_35 || var_32)` (line 32 of Fig. 5). In the rule π-REL, `op` is any of these operators: `<, <=, >, >=, ==, !=`.

Fig. 11 shows the translation rules for the **`\valid`** predicate that holds if its parameter points to a valid memory location and can be dereferenced. Basic usages are **`\valid`**`(t)` for checking the validity of a single pointer `t`, and **`\valid`**`(t1+(t2..t3))` for checking the validity of pointer `t1` within offset range `t_2..t_3`. The rules π-VALID and π-VALID-RANGE cover both usages. We assume that we can evaluate the validity of the memory location a pointer points to by the means of a function `fvalid`, and the validity of a pointer within an offset range by `fvalidr`. In our running example of Fig. 4, the predicate **`\valid`**`(t+(0..n-1))` in the **`requires`** clause of line 2 is translated to `fvalidr(t,0,(n-1))` (line 4 of Fig. 5). Full support of these predicates requires a precise low-level memory representation.

The rules π-EXISTS (Fig. 12) translates the existentially quantified predicate **`\exists`** other integer values. It updates a variable `var_n` (initially set to $false$) at each iteration of the loop until all values of `i` are considered or until `var_n` is evaluated to $true$. This rule also treats the universally quantified predicate (l, **`\forall integer`** `i; t1<=i<t2 ==> p`) since it is equivalent to (l, `! (`**`\exists integer`** `i; t1<=i<t2 && !p)`). In Fig. 4, the quantified predicate **`\exists integer`** `i; 0<=i<n && *(t+i)==v` of line 4 is translated in lines 29–31 of Fig. 5 and the fresh variable containing the value of the predicate after its evaluation is `var_35` (declared at line 29).

V. ANNOTATION TRANSLATION

Now we define the translation rules for annotations that allow to trigger annotation failures and enforce test generation of erroneous inputs. For each annotation, a fragment of a C program ending by a condition test is inserted into the program. The test generator will try to cover all feasible paths, thus activating the error if such inputs exist. The rules are of the following form:

$$\frac{(l_1, w) \mapsto (I_1, e_1) \quad \dots \quad (l_n, w) \mapsto (I_n, e_n)}{(l, \texttt{kwd w;}) \mapsto g_{kwd}(l, (I_1, e_1), \dots, (I_n, e_n))}$$

This rule pattern states that if the term or predicate w at label l_i is translated (by τ or π) as (I_i, e_i), then the

$$\tau\text{-COERCE}_1\quad \frac{(l, t:\mathbb{Z}) \mapsto (I,e)}{(l,(\mathit{int})(t:\mathbb{Z})) \mapsto (I\cdot(l, \textbf{int}\ \texttt{var\_n = }\underline{\texttt{e}^{\boxtimes}}\texttt{;}), \texttt{var\_n})}\qquad \tau\text{-COERCE}_2\quad \frac{(l,t:\mathit{int})\mapsto(I,e)}{(l,(\mathbb{Z})(t:\mathit{int}))\mapsto(I\cdot(l,\underline{{}^{\square}\texttt{var\_n = e}}\texttt{;}),\texttt{var\_n})}$$

Fig. 7. Translation rules for coercions

$$\tau\text{-BINOP}_1\quad \frac{(l,t_1:\mathit{ptr})\mapsto(I_1,e_1)\quad (l,t_2:\mathit{int})\mapsto(I_2,e_2)\quad \mathit{op}\in\{+,-\}}{(l,(\texttt{t1 op t2}):\mathit{ptr})\mapsto(I_1\cdot I_2,\texttt{e1 op e2})}\qquad \tau\text{-UNOP}_1\quad \frac{(l,t:\mathit{ptr})\mapsto(I,e)}{(l,(\texttt{* t}):\mathit{ctype})\mapsto(I,\texttt{* e})}$$

$$\tau\text{-BINOP}_2\quad \frac{(l,t_1:\mathbb{Z})\mapsto(I_1,e_1)\quad (l,t_2:\mathbb{Z})\mapsto(I_2,e_2)\quad \mathit{op}\in\{\texttt{==},\texttt{!=},\texttt{<},\texttt{<=},\texttt{>},\texttt{>=},\texttt{||},\texttt{\&\&}\}}{(l,(\texttt{t1 op t2}):\mathit{int})\mapsto(I_1\cdot I_2\cdot(l,\textbf{int}\ \texttt{var\_n = }\underline{\texttt{e1}^{\boxtimes}\ \texttt{op e2}^{\boxtimes}}\texttt{;}),\texttt{var\_n})}\qquad \tau\text{-UNOP}_2\quad \frac{(l,t:\mathit{int})\mapsto(I,e)}{(l,(\texttt{! t}):\mathit{int})\mapsto(I,\texttt{! e})}$$

$$\tau\text{-BINOP}_3\quad \frac{(l,t_1:\mathbb{Z})\mapsto(I_1,e_1)\quad (l,t_2:\mathbb{Z})\mapsto(I_2,e_2)\quad \mathit{op}\in\{\texttt{+},\texttt{-},\texttt{/},\texttt{\%},\texttt{<<},\texttt{>>},\texttt{|},\texttt{\&},\texttt{\^{}}\}}{(l,(\texttt{t1 op t2}):\mathbb{Z})\mapsto(I_1\cdot I_2\cdot(l,\underline{{}^{\square}\texttt{var\_n = e1}^{\boxtimes}\ \texttt{op e2}^{\boxtimes}}\texttt{;}),\texttt{var\_n})}\qquad \tau\text{-UNOP}_3\quad \frac{(l,t:\mathbb{Z})\mapsto(I,e)\quad \mathit{op}\in\{-,\sim\}}{(l,(\texttt{op t}):\mathbb{Z})\mapsto(I,\cdot(l,\underline{\texttt{e = op e}}\texttt{;}),\texttt{e})}$$

$$\tau\text{-IF}\quad \frac{(l,t_1:\mathbb{Z})\mapsto(I_1,e_1)\qquad (l,t_2:\mathbb{Z})\mapsto(I_2,e_2)\qquad (l,t_3:\mathbb{Z})\mapsto(I_3,e_3)}{(l,(\texttt{t1 ? t2 : t3}):\mathbb{Z})\mapsto(I_1\cdot(l,\textbf{if}\texttt{(}\underline{\texttt{e1}^{\boxtimes}\ \texttt{!= 0}}\texttt{) \{})\cdot I_2\cdot(l,\underline{{}^{\square}\texttt{var\_n = e2}^{\boxtimes}}\texttt{;\}}\ \textbf{else}\ \texttt{\{})\cdot I_3\cdot(l,\underline{{}^{\square}\texttt{var\_n = e3}^{\boxtimes}}\texttt{;\}}),\texttt{var\_n})}$$

Fig. 8. Translation rules for unary and binary operations

$$\tau\text{-SUM}\quad \frac{(l,t_1:\mathbb{Z})\mapsto(I_1,e_1)\qquad (l,t_2:\mathbb{Z})\mapsto(I_2,e_2)\qquad (l,t_3:\mathbb{Z})\mapsto(I_3,e_3)}{\begin{array}{l}(l,(\textbf{\textbackslash sum}\texttt{(t1, t2, }\textbf{\textbackslash lambda integer}\ \texttt{i; t3)}):\mathbb{Z})\mapsto\\ \quad(I_1\cdot I_2\cdot(l,\underline{{}^{\square}\texttt{var\_n = 0}}\texttt{;}\ \textbf{for}\texttt{(}\underline{{}^{\square}\texttt{i\_n = e1}^{\boxtimes}}\texttt{;}\ \underline{\texttt{i\_n <= e2}^{\boxtimes}}\texttt{;}\ \underline{\texttt{i\_n++}^{\boxtimes}}\texttt{)\{})\cdot I_3\cdot(l,\underline{\texttt{var\_n += e3}^{\boxtimes}}\texttt{;\}}),\texttt{var\_n})\end{array}}$$

$$\tau\text{-NUMOF}\quad \frac{(l,t_1:\mathbb{Z})\mapsto(I_1,e_1)\qquad (l,t_2:\mathbb{Z})\mapsto(I_2,e_2)\qquad (l,t_3:\mathit{int})\mapsto(I_3,e_3)}{\begin{array}{l}(l,(\textbf{\textbackslash numof}\texttt{(t1, t2, }\textbf{\textbackslash lambda integer}\ \texttt{i; t3)}):\mathbb{Z})\mapsto\\ \quad(I_1\cdot I_2\cdot(l,\underline{{}^{\square}\texttt{var\_n = 0}}\texttt{;}\ \textbf{for}\texttt{(}\underline{{}^{\square}\texttt{i\_n = e1}^{\boxtimes}}\texttt{;}\ \underline{\texttt{i\_n <= e2}^{\boxtimes}}\texttt{;}\ \underline{\texttt{i\_n++}^{\boxtimes}}\texttt{) \{}\cdot I_3\cdot(l,\textbf{if}\texttt{(e3)}\ \underline{\texttt{var\_n++}}\texttt{;\}}),\texttt{var\_n})\end{array}}$$

Fig. 9. Translation rules for builtin logic functions

$$\pi\text{-EQUIV}\quad \frac{(l,p_1)\mapsto(I_1,e_1)\qquad (l,p_2)\mapsto(I_2,e_2)}{(l,\texttt{p1 <==> p2})\mapsto(I_1\cdot I_2,\texttt{((!e1 || e2) \&\& (!e2 || e1))})}$$

$$\pi\text{-AND}\quad \frac{(l,p_1)\mapsto(I_1,e_1)\qquad (l,p_2)\mapsto(I_2,e_2)}{(l,\texttt{p1 \&\& p2})\mapsto(I_1\cdot(l,\textbf{int}\ \texttt{var\_n = e1;}\ \textbf{if}\texttt{(var\_n) \{})\cdot I_2\cdot(l,\texttt{var\_n = e2; \}}),\texttt{var\_n})}\qquad \pi\text{-TRUE}\quad \frac{}{(l,\textbf{\textbackslash true})\mapsto(\emptyset,\texttt{1})}$$

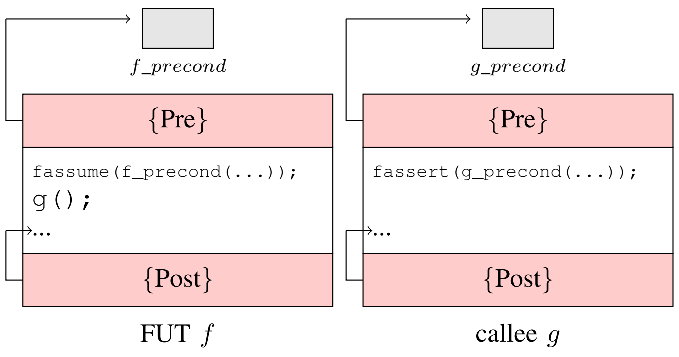
$$\pi\text{-OR}\quad \frac{(l,p_1)\mapsto(I_1,e_1)\qquad (l,p_2)\mapsto(I_2,e_2)}{(l,\texttt{p1 || p2})\mapsto(I_1\cdot(l,\textbf{int}\ \texttt{var\_n = e1;}\ \textbf{if}\texttt{(!var\_n) \{})\cdot I_2\cdot(l,\texttt{var\_n = e2; \}}),\texttt{var\_n})}\qquad \pi\text{-FALSE}\quad \frac{}{(l,\textbf{\textbackslash false})\mapsto(\emptyset,\texttt{0})}$$

$$\pi\text{-IMPL}\quad \frac{(l,p_1)\mapsto(I_1,e_1)\qquad (l,p_2)\mapsto(I_2,e_2)}{(l,\texttt{p1 ==> p2})\mapsto(I_1\cdot(l,\textbf{int}\ \texttt{var\_n = 1;}\ \textbf{if}\texttt{(e1) \{})\cdot I_2\cdot(l,\texttt{var\_n = e2; \}}),\texttt{var\_n})}\qquad \pi\text{-NOT}\quad \frac{(l,p)\mapsto(I,e)}{(l,\texttt{!p})\mapsto(I,\texttt{!e})}$$

$$\pi\text{-IF}\quad \frac{(l,t:\mathbb{Z})\mapsto(I_1,e_1)\qquad (l,p_2)\mapsto(I_2,e_2)\qquad (l,p_3)\mapsto(I_3,e_3)}{(l,\texttt{t ? p2 : p3})\mapsto(I_1\cdot(l,\textbf{int}\ \texttt{var\_n;}\ \textbf{if}\texttt{(}\underline{\texttt{e1}^{\boxtimes}\ \texttt{!= 0}}\texttt{) \{})\cdot I_2\cdot(l,\texttt{var\_n=e2; \}}\ \textbf{else}\ \texttt{\{})\cdot I_3\cdot(l,\texttt{var\_n=e3; \}}),\texttt{var\_n})}$$

$$\pi\text{-REL}\quad \frac{(l,t_1:\mathbb{Z})\mapsto(I_1,e_1)\quad (l,t_2:\mathbb{Z})\mapsto(I_2,e_2)\quad \mathit{op}\in\{\texttt{<},\texttt{<=},\texttt{>},\texttt{>=},\texttt{==},\texttt{!=}\}}{(l,\texttt{t1 op t2})\mapsto(I_1\cdot I_2\cdot(l,\textbf{int}\ \texttt{var\_n = }\underline{\texttt{e1}^{\boxtimes}\ \texttt{op e2}^{\boxtimes}}\texttt{;}),\texttt{var\_n})}$$

Fig. 10. Translation rules for simplest predicates



```
fassume(f_precond(...));
g();
...
```

```
fassert(g_precond(...));
...
```

Fig. 13. Pre-Post translation scheme for the FUT and called functions

property $kwd\ w$ involving w at program point l will be translated by some composition function of code insertions of each I_i and of the expressions e_i. The particular composition function, denoted g_{kwd}, depends on the annotation kind. In Fig. 14 and Fig. 15 we present the translation rules for each annotation kind. Suppose `fassert` is a C function checking an expression. We define the condition e that must be tested for each annotation. The test is expressed by applying the function `fassert`, that is expanded to a conditional **if**. Test generation tries to cover both branches. If the expected property is false, the failure is reported and the exploration switches to another branch.

$$\alpha\text{-ASSERT}\quad \frac{(l,p)\mapsto(I,e)}{(l,\textbf{assert}\ \texttt{p;})\mapsto I\cdot(l,\texttt{fassert(e);})}$$

$$\alpha\text{-POST}\quad \frac{(End_f,p)\mapsto(I,e)}{(End_f,\textbf{ensures}\ \texttt{p;})\mapsto I\cdot(End_f,\texttt{fassert(e);})}$$

$$\alpha\text{-PRE}\quad \frac{(Beg_{f\_precond},p)\mapsto(I,e)}{\left(Beg_f,\left\{\begin{array}{r}\textbf{typically}\ \texttt{p;}\\ \textbf{requires}\ \texttt{p;}\end{array}\right.\right)\mapsto I\cdot(Beg_{f\_precond},\textbf{if}\texttt{(!e)}\ \textbf{return}\ \texttt{0;})}$$

Fig. 14. Translation rules for assert, postcondition and precondition

Fig. 14 describes the translation rules for an assertion at label l (α-ASSERT), a postcondition (α-POST) and a precondition (α-PRE) that are considered to be stated resp. at labels End_f and Beg_f. The rule α-ASSERT simply checks the translated predicate at label l with the function `fassert`. The rule α-POST checks the predicate at the end of the function being translated. The rule α-PRE checks the predicate of a precondition in a separate function that we call `f_precond`, returning 0 if one of the **`requires`** or **`typically`** clauses does not hold, or 1 otherwise. This function has to be called with the same formal parameters as the function being translated. Since the precondition is *assumed* for the function under test and must be *ensured* for a called function, its result has to be *asserted* in the case of

$$\pi\text{-VALID}\quad \frac{(l, t : ptr) \mapsto (I, e)}{(l, \texttt{\textbackslash valid(t)}) \mapsto (I, \texttt{fvalid(e)})} \qquad \pi\text{-VALID-RANGE}\quad \frac{(l, t_1 : ptr) \mapsto (I_1, e_1)(l, t_2 : int) \mapsto (I_2, e_2)(l, t_3 : int) \mapsto (I_3, e_3)}{(l, \texttt{\textbackslash valid(t1+(t2..t3))}) \mapsto (I_1 \cdot I_2 \cdot I_3, \texttt{fvalidr(e1,e2,e3)})}$$

Fig. 11. Translation rules for memory validity predicate

$$\pi\text{-EXISTS}\quad \frac{(l, t_1 : \mathbb{Z}) \mapsto (I_1, e_1) \qquad (l, t_2 : \mathbb{Z}) \mapsto (I_2, e_2) \qquad (l, p) \mapsto (I, e)}{\begin{array}{l}(l, \texttt{\textbackslash exists integer i; t1 <= i < t2 \&\& p}) \mapsto \\ \quad (I_1 \cdot I_2 \cdot (l, \texttt{int var\_n = 0; for(}\underline{^{\square}\texttt{i\_n = e1}^{\boxtimes}}\texttt{; }\underline{\texttt{i\_n < e2}^{\boxtimes}}\texttt{ \&\& !var\_n; }\underline{\texttt{i\_n++}^{\boxtimes}}\texttt{) \{}) \cdot I \cdot (l, \texttt{var\_n = e; \}}), \texttt{var\_n})\end{array}}$$

Fig. 12. Translation rule for quantified predicate

a *callee* and *assumed* in the case of the function under test. Fig. 13 illustrates where pre/postconditions checks are inserted for a FUT f that calls g_1, that calls g_2, etc. So for each function, a check `fassert(h_precond(x1, x2, ..., xn));` (resp. `fassume(h_precond(x1, x2, ..., xn));`) is inserted at the label Beg_h if h is a *callee* (resp. FUT), where $x_1, x_2, ..., x_n$ are the formal parameters of h, and the `fassume` function restricts test generation to input values for which it returns true. In the example of Fig. 4, the **`requires`** clause lines 1-2 and the **`typically`** clause line 3 are translated to the lines 3–5 of Fig. 5. The validity of the `is_present_precond` precondition is *assumed* line 11 of Fig. 5. Finally, the **`ensures`** clause lines 4-5 is translated as lines 28–32 in Fig. 5.

The translation rules for the loop invariants (α-INVARIANT) and loop variants (α-VARIANT) for a loop at label l are presented in Fig. 15. The rule α-INVARIANT checks the predicate of the loop invariant before the loop, and after each iteration of the loop (i.e. at the predefined label $EndIter_l$). The rule α-VARIANT first checks that the variant term is positive or zero before the first loop iteration. Then, at the beginning of each loop iteration, it saves the previous value of the term in a fresh variable `old_variant`. Finally, it checks at the end of each iteration that the current value of the variant is still positive or zero and is strictly decreasing (compared to its value in the previous iteration). In the running example of Fig. 4, translating the **`loop invariant`** line 9 results in lines 13–14 and 21–22 in Fig. 5, while translating the **`loop variant`** line 10 results in lines 15, 18 and 23–25 in Fig. 5.

VI. Instrumentation for Test Generation vs Runtime Assertion Checking

Instrumentation based translation of ACSL annotations into C code has been implemented in two Frama-C [1] plugins: E-ACSL2C [5], [6] that generates an instrumented program for Runtime Assertion Checking (RAC), and StaDy, that instruments a program for Test Generation with PathCrawler [8]. This section discusses similarities and differences between both kinds of instrumentation.

Test generation and runtime checking both need to generate executable code and so consider only an executable subset of the specification language. Therefore, most rules defined in Sec. III, IV and V for test generation are also valid for RAC.

Precondition of the function under test (FUT). One difference is the treatment of the precondition of the FUT. In RAC, it is usually checked as any other annotation. In test generation, it is used to avoid testing the program on inadmissible values for which the program is not supposed to work correctly. Hence, the precondition of the FUT should be assumed during test generation to ensure that all generated test inputs respect the precondition of the FUT (cf Fig. 13).

Besides, the treatment of the precondition of the FUT in PathCrawler has two optimizations. First, an internal mechanism of unquantified and quantified preconditions allows a direct translation of ACSL preconditions into constraints supported in an efficient manner. Hence, for precondition patterns that can be expressed by this mechanism, a translation into C code is not necessary for PathCrawler. Second, for the remaining preconditions translated into a C function, PathCrawler offers a specific efficient mechanism [10]. A call to `fassume` (adding the constraints of the precondition to the constraint store *before* the path predicate in the FUT) can be replaced in PathCrawler by a dedicated support for late precondition (where precondition constraints are posted *after* other path constraints of the FUT). Thus the call to `fassume` for the FUT in PathCrawler is not needed (cf Fig. 13).

Memory-related constructs. Runtime checkers also require a complex instrumentation framework to treat memory-related constructs where each memory related operation is instrumented and relative memory block metadata is stored so that it can be extracted when it is necessary to evaluate a memory-related ACSL annotation [6]. Some of these constructs can be handled symbolically in concolic testing without additional instrumentation. The functions `fvalid` and `fvalidr` are builtin C functions defined by PathCrawler that return the value of validity of a pointer [17]. They actually support global variables and formal parameters of the function under test.

Unbounded integers. While translation of ACSL mathematical integers relying on an external library for unbounded integers (like GMP) is appropriate and sufficient for RAC, it will be quite inefficient if the library function code is directly handled by test generation. Indeed, test generation on the instrumented code would have to treat much more complex code, with lots of additional function calls, dynamic memory allocation and de-allocation, etc. This can be avoided again using symbolic execution of the test generation tool. PathCrawler offers dedicated builtin support for GMP numbers and operations that are efficiently translated into appropriate constraints on unbounded integers and handled by the underlying constraint solver.

Runtime errors. Straightforward translation of annotations into C may introduce runtime errors due to annotations with undefined terms (such as 1/0, cf Sec. II-A2). This issue can be easily solved for test generation exactly as proposed in [5] for RAC, by an additional guard generation phase by running the Frama-C/RTE plugin [1] on the instrumented code to add annotations preventing runtime errors, and finally running the instrumentation again on these new annotations. Runtime errors related to unbounded integer (division of a GMP integer by 0, overflow during a type coersion (**int**)(t:$\mathbb{Z}$), etc.) are not treated by RTE, but they can be easily prevented by adding suitable checks in the corresponding rules. For

$$\alpha\text{-INVARIANT}\quad \frac{(l,p)\mapsto(I_1,e_1)\qquad (EndIter_l,p)\mapsto(I_2,e_2)}{(l,\texttt{\textbf{loop invariant} p;})\mapsto I_1\cdot(l,\texttt{fassert(e1);})\cdot I_2\cdot(EndIter_l,\texttt{fassert(e2);})}$$

$$\alpha\text{-VARIANT}\quad \frac{(l,t)\mapsto(I_1,e_1)\qquad (BegIter_l,t)\mapsto(I_2,e_2)\qquad (EndIter_l,t)\mapsto(I_3,e_3)}{\begin{array}{l}(l,\texttt{\textbf{loop variant} t;})\mapsto\\ I_1\cdot(l,\texttt{fassert(}\underline{\texttt{0 <= e1}^{\boxtimes}}\texttt{);})\cdot I_2\cdot(BegIter_l,\underline{{}^{\square}\texttt{old\_variant = e2}^{\boxtimes}}\texttt{;})\cdot I_3\cdot(EndIter_l,\texttt{fassert(}\underline{\texttt{0 <= e3}}\ \texttt{\&\&}\ \underline{\texttt{e3}^{\boxtimes}\texttt{< old\_variant}^{\boxtimes}}\texttt{);})\end{array}}$$

Fig. 15. Transformation rules for loop annotations: invariant and variant

example	time (s.)	# paths
array-unsafe	1.299	9
count-up-down-unsafe	1.285	3
eureka-01-unsafe	1.355	48
for-bounded-loop1-unsafe	1.320	11
insertion-sort-unsafe	16.530	730
invert-string-unsafe	1.359	48
linear-search-unsafe	3.624	2766
matrix-unsafe	1.367	22
nec20-unsafe	1.463	1035
string-unsafe	1.362	48
sum01-bug02-base-unsafe	1.335	26
sum01-bug02-unsafe	1.327	36
sum01-unsafe	1.312	56
sum03-unsafe	1.291	46
sum04-unsafe	1.310	22
sum-array-unsafe	1.358	14
trex03-unsafe	1.358	21
sendmail-unsafe	1.396	77
vogal-unsafe	1.349	341

Fig. 16. Experiments with StaDy: Bug detection

example	mutants	$\neg$ equiv.	killed	success rate
merge-sort	96	92	88	95.65%
merge-arrays	68	63	59	93.65%
quick-sort	130	130	130	100%
binary-search	40	40	39	97.5%
bubble-sort	52	49	42	85.71%
insertion-sort	39	37	36	97.3%
array-safe	18	16	15	93.75%
bubble-sort-safe	64	58	55	94.83%
count-up-down-safe	14	13	13	100%
eureka-01-safe	60	60	60	100%
eureka-05-safe	36	36	36	100%
insertion-sort-safe	43	41	40	97.56%
invert-string-safe	47	47	47	100%
linear-search-safe	19	17	16	94.12%
matrix-safe	30	27	25	92.59%
nc40-safe	20	20	20	100%
nec40-safe	20	20	20	100%
string-safe	65	65	65	100%
sum01-safe	14	14	13	92.86%
sum02-safe	14	14	11	78.57%
sum03-safe	10	10	10	100%
sum04-safe	14	14	10	71.43%
sum-array-safe	17	17	15	88.24%
trex03-safe	56	56	56	100%
sendmail-safe	31	31	31	100%
vogal-safe	71	68	67	98.53%
Total	1088	1054	1019	**96.68%**

Fig. 17. Experiments with StaDy: Mutation testing

example, the downcast of a term t from integer to int (see the rule τ-COERCE$_1$ of Fig. 7) can be guarded by inserting `fassert(`$\underline{\texttt{INT\_MIN <= e}}$ `&&` $\underline{\texttt{e <= INT\_MAX}}$`)` before the assignment $\underline{\texttt{var\_n = e}}$.

Triggering errors during test generation. The function `fassert(cond)` is a C macro defined by PathCrawler that is expanded as a conditional **`if`**`(cond)`, testing if its parameter is true. Covering all feasible paths of the program will therefore enforce the generation of test inputs activating the *then* branch of this condition, and test inputs activating the *else* branch. So if there exist inputs such that `cond` is evaluated to $false$, a test case activating the `fassert` thus violating the corresponding annotation is generated. In other terms, a counter-example for that annotation is generated if such inputs exist.

VII. Experimental Results

The current implementation of StaDy supports a significant subset of E-ACSL including assertions, pre- and postconditions, loop invaliants and variants, quantifications, logic functions, integral and pointer types, and basic pointer operations. Pointer validity is currenty supported only for input arrays and pointers. StaDy currently does not support **`assigns`** clauses, **`\at`** terms, real numbers, as well as advanced memory-related constructs (e.g. **`\offset`**), complex pointer arithmetics such as `p1-p2` or `*(p-i)` and dynamic memory allocation due to the limitations of the underlying test generator.

To evaluate the efficiency of StaDy in a combined verification approach (cf Sec. I), we applied it on safe and unsafe programs from the TACAS 2014 Software Verification CompetitionFirst, we executed StaDy on 20 faulty programs that handle arrays with loops. The properties to invalidate originally expressed as C assertions, were manually rewritten in E-ACSL. Adequate E-ACSL preconditions were also added. The programs containing infinite loops and reachability properties to invalidate are not handled by StaDy due to the necessity to execute the program in PathCrawler. StaDy detected failures of all faulty properties in each considered program. Fig. 16 illustrates the time taken to invalidate the properties including all the steps of StaDy: instrumentation from the E-ACSL specifications and test generation in PathCrawler, and the number of explored paths.

Secondly, we used mutation testing to evaluate the ability of StaDy to find bugs in unsafe programs. We selected 20 safe programs of the same benchmark, and 6 additional safe programs from our own benchmarks. All of them were annotated in E-ACSL. They contain preconditions, postconditions, assertions, memory-related properties, loop variants and invariants. We used mutation testing on these safe programs to generate modified programs (*mutants*) and see if StaDy is able to *kill* (i.e. to find errors in) these mutants. The mutations performed on the source code mimic usual programming errors. They include modifications of numerical and/or pointer arithmetic operators, comparison operators, condition negation and logical operators (*and* and *or*). Fig. 17 gives the numbers of all and erroneous mutants, as well as the number and proportion of erroneous mutants killed by StaDy. StaDy showed an average success rate of 96.68%, going up to 100% on many examples. The missing percents are mostly due to a currently incomplete support of E-ACSL features by the underlying test generation tool.

VIII. Related Work

Cheon and Leavens [15], [18] presented a runtime checker for JML (that have inspired ACSL) and proposed an original way to handle the issue of undefinedness in assertions, and present translation rules for the assertions. Our work continues these efforts, but addresses the C language, focuses on translation for testing and considers a larger range of constructs (including some memory-related constructs, invariants and pre/postconditions).

Zee et al. [19] described a runtime checker for Jahob, a verification system for a subset of Java. They discuss the support of challenging constructs like quantifications, set comprehensions and previous program states (the `old` construct). Contrary to ACSL, the annotation language they rely on can express higher-order logic assertions (but their implementation is restricted to first-order quantifiers). The set comprehension feature is also present in ACSL, although not implemented in our tool and not defined in our rules. Some details about treating these features are discussed in [19] but their translation rules are not detailed. Our work proposes a formal description of the translation rules for testing.

Polikarpova et al. [20] presented a technique to generate executions of programs annotated with Boogie specifications. It exhibits test cases that are either counter-examples violating the specification or test witnesses validating the specification if no error has been exposed for the annotations. Compared to ACSL, the Boogie intermediate verification language contains additional constructs like nondeterministic assignments and map applications, but it does not have critical features like pointers. In their approach, the specification is not translated into an executable language for execution, but is directly evaluated symbolically and concretely using symbolic execution and SMT constraint solving, where the translation rules were not formally presented. Following a similar objective, our work offers a formalization of translation rules for C language that allows a black-box use of a testing tool. This choice facilitates tool integration and allows the user to use another test generator to validate properties, like KLEE [21].

Delahaye et al. [5] discussed the first insights about the translation of E-ACSL annotations into executable C programs. Kosmatov et al. [6] focused on the memory model of the C monitor and aimed at resolving the issue of memory-related E-ACSL constructs and scaling. To the best of our knowledge, a similar formalization of translation rules that allow to combine formal verification and test generation for C programs has never been presented before.

IX. Conclusion and Future Work

In this paper, we have given a formal description of the translation rules to derive executable C code from E-ACSL [5] specification for testing. This formalization offers a strict and easily reusable presentation of instrumentation-based support of an expressive specification language during test generation. Moreover, it constitutes a major step to formal verification of correctness of the annotation translation, that is necessary to ensure global correctness of derived combinations of static and dynamic analyses sharing the same specification formalism. Most parts of the formalization remain valid for RAC, and we have emphasized specific issues (related to unbounded integers, precondition of the FUT, memory-related constructs, etc.) that are different. We have implemented this translation in the StaDy tool as part of the Frama-C [1] verification framework. Our experiments show the efficiency of StaDy in providing more confidence in correctness, or in finding counter-examples, for C programs according to an E-ACSL specification. We believe that it can significantly facilitate the specification and verification task in an incremental verification approach combining deductive verification with testing. Future work includes formal proof of correctness, support of additional ACSL constructs like type invariants, data invariants and lemmas, that will strongly rely on the predicate translation presented in this paper, and further experiments with the StaDy tool on real-life C programs.

Acknowledgment. The authors thank the Frama-C and PathCrawler teams for providing the tools and support. Special thanks to Alain Giorgetti for many fruitful discussions, suggestions and advice.

References

[1] P. Cuoq, F. Kirchner, N. Kosmatov, V. Prevosto, J. Signoles, and B. Yakobowski, "Frama-C - a software analysis perspective," in *SEFM*, 2012.

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

[3] G. Petiot, N. Kosmatov, A. Giorgetti, and J. Julliand, "How test generation helps software specification and deductive verification in Frama-C," in *TAP*, 2014, to appear.

[4] L. Correnson and J. Signoles, "Combining analyses for C program verification," in *FMICS*, 2012.

[5] M. Delahaye, N. Kosmatov, and J. Signoles, "Common specification language for static and dynamic analysis of C programs," in *SAC*, 2013.

[6] N. Kosmatov, G. Petiot, and J. Signoles, "An optimized memory monitoring for runtime assertion checking of C programs," in *RV*, 2013.

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

[8] 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 *AST*, 2009.

[9] K. Sen and G. Agha, "CUTE and jCUTE: concolic unit testing and explicit path model-checking tools," in *CAV*, 2006.

[10] M. Delahaye and N. Kosmatov, "A late treatment of C precondition in dynamic symbolic execution testing tools," in *RV*, 2013.

[11] B. Meyer, *Object-Oriented Software Construction*. Prentice-Hall, 1988.

[12] P. Chalin, "Engineering a sound assertion semantics for the verifying compiler," *IEEE Trans. Software Eng.*, vol. 36, pp. 275–287, 2010.

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

[14] C. Dross, P. Efstathopoulos, D. Lesens, D. Mentré, and Y. Moy, "Rail, space, security: Three case studies for SPARK 2014," in *Proc. ERTS*, 2014.

[15] Y. Cheon, "A runtime assertion checker for the java modeling language," Ph.D. dissertation, Iowa State Univ., 2003.

[16] C. Boyapati, S. Khurshid, and D. Marinov, "Korat: Automated testing based on java predicates," in *ISSTA*, 2002.

[17] O. Chebaro, M. Delahaye, and N. Kosmatov, "Testing inexecutable conditions on input pointers in C programs with SANTE," in *ICSSEA*, 2012.

[18] Y. Cheon and G. T. Leavens, "A contextual interpretation of undefinedness for runtime assertion checking," in *AADEBUG*, 2005.

[19] K. Zee, V. Kuncak, M. Taylor, and M. Rinard, "Runtime checking for program verification," in *RV*, 2007.

[20] N. Polikarpova, C. A. Furia, and S. West, "To run what no one has run before: Executing an intermediate verification language," in *RV*, 2013.

[21] C. Cadar, D. Dunbar, and D. Engler, "KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs," in *OSDI*, 2008.