Tame your annotations with MetAcsl: Specifying, Testing and Proving High-Level Properties

Virgile Robles[1][0000−0002−5838−134X], Nikolai Kosmatov[1,2][0000−0003−1557−2813], Virgile Prevosto[1][0000−0002−7203−0968], Louis Rilling[3][0000−0003−4520−6646], and Pascale Le Gall[4][0000−0002−8955−6835]

[1] Institut LIST, CEA, Université Paris-Saclay, Palaiseau, France
firstname.lastname@cea.fr
[2] Thales Research & Technology, Palaiseau, France
[3] DGA, France, louis.rilling@irisa.fr
[4] Laboratoire de Mathématiques et Informatique pour la Complexité et les Systèmes CentraleSupélec, Université Paris-Saclay, Gif-Sur-Yvette, France
pascale.legall@centralesupelec.fr

Abstract. A common way to specify software properties is to associate a contract to each function, allowing the use of various techniques to assess (e.g. to prove or to test) that the implementation is valid with respect to these contracts. However, in practice, high-level properties are not always easily expressible through function contracts. Furthermore, such properties may span across multiple functions, making the specification task tedious, and its assessment difficult and error-prone, especially on large code bases. To address these issues, we propose a new specification mechanism called meta-properties. Meta-properties are enhanced global invariants specified for a set of functions, capable of expressing predicates on values of variables as well as memory related conditions (such as separation) and read or write access constraints. This paper gives a detailed presentation of meta-properties and their support in a dedicated Frama-C plugin MetAcsl, and shows that they are automatically amenable to both deductive verification and testing. This is demonstrated by applying these techniques on two illustrative case studies.

1 Introduction

Function contracts are a common way of specifying the functional behavior of a program in a modular manner. In this setting, each function of the program is annotated with both preconditions (properties expected to be ensured by the caller of the function) and postconditions (properties that must be ensured after the function returns). Various assessment techniques exist to check the validity of a function implementation with respect to its contract.

This is the case in the Frama-C [1] framework, built for the analysis of C programs. Frama-C allows the user to specify function contracts in its companion specification language ACSL [2] and to express first-order properties on program

variables. A variety of *plugins* can be used to assess the validity of the C program with respect to these contracts, for example via static (deductive) verification or dynamic verification (runtime assertion checking).

Motivation However, some categories of properties over a C program are not easily expressible via function contracts. In particular, some properties, which we may call *global* properties, are spanning across multiple functions. For instance, we might want to ensure that all accesses to some data are guarded by a proper authentification mechanism. Writing contracts for each function encompassed by a global property is tedious and error-prone, especially on large code bases. In the end, there is no guarantee other than manual verification that the set of provided contracts correctly and completely expresses the global property: in the example above, checking that all accesses are indeed guarded by an appropriate annotation would quickly become very difficult. In this situation, even when all contracts are verified, it cannot be directly deduced with a high level of confidence that the global property is indeed true.

This can become even harder when the contract clauses related to the global property are mixed with other, usual clauses: when updating the contract of a function, it becomes very easy to invalidate the global property since there is no explicit link between this property and the associated contract clauses. This need for global properties arises in two different case studies we tackled, each one involving both safety and security properties over a whole library of functions.

To address these issues, we propose a new specification mechanism within Frama-C called *meta-properties* (whose main ideas were briefly presented in [3]), which provides the verification engineer with a means to easily specify global properties on a C program. An automated translation of meta-properties into ACSL annotations allows using existing analysis plugins for the assessment.

This paper provides a rigorous description of the notion of meta-property and its instantiation mechanism, along with multiple extensions that were implemented in the MetAcsl plugin, and an evaluation of assessment techniques for meta-properties using static and dynamic verification.

Contributions More precisely, the contributions of this paper include :

- a proper formalization of the notion of meta-property and its translation into ACSL (Sections 2 and 3),
- several extensions to basic meta-properties allowing the user to write more expressive properties on a larger class of programs (Section 4),
- a detailed description of a previous case study regarding confidentiality (introduced in [3]), and a new case study about smart houses, that are both specified using meta-properties (Section 5),
- a demonstration that meta-properties are amenable to a new assessment method, runtime verification (Section 6), and
- an evaluation of two assessment techniques—deductive verification and runtime assertion checking—on the two case studies (Section 6).

With respect to the previous tool demo paper [3] (which gave an informal presentation of meta-properties and briefly illustrated their usage for deductive verification only), the present paper gives a complete formal description of meta-properties, illustrated by several examples, provides more detail on several recent extensions, the proposed transformation-based approach and the considered case studies, and demonstrates the capacity of the proposed approach to be combined with both deductive verification and runtime assertion checking.

2 Specification of Meta-Properties

Meta-properties are a way of expressing a category of *high-level, global* program properties. It basically consists of a local property, a notion of scope (a set of target functions) indicating which parts of the program have to respect the property, and a notion of context determining what kinds of situations (e.g. which instructions) in the target functions must be constrained by the property. This section formalizes and illustrates this notion.

2.1 Definition of a Meta-Property

We assume that we are working on a *complete* C program, where all functions are defined in one of the source files composing the program. Moreover, statements have been normalized such that each instruction modifies at most one single memory location. In Frama-C, this normalization phase is enforced by the kernel.

In this setting, let $\mathcal{F}$ denote the set of all functions defined in the program under analysis. We define as usual the control-flow graph (CFG) of each function as a directed graph (V, E) where each vertex $v \in V$ is a *single* C instruction and an edge $e \in E$ from v_1 to v_2 indicates that after executing v_1 the program may execute v_2 (conditional statements may have two successors, while normal instructions, such as assignments, have exactly one).

A *meta-property* is then defined as a triple $(F, \mathcal{C}, P)$ where :

- $F \subseteq \mathcal{F}$ is the *target set*, delimiting the scope of the meta-property.
- $\mathcal{C}$ is a called a *context*. It is defined as a pair $(\mathcal{M}, \theta)$ where $\mathcal{M}$ is a (potentially empty) set of names that we call *meta-variables* and θ is a *contextualization mapping*. Given a C function `f` having a CFG (V, E), θ associates `f` with a set whose elements are pairs (e, m) where $e \in E$ and m is an *environment mapping* which maps each name of $\mathcal{M}$ to an ACSL term. Informally, the contextualization mapping defines a criterion for selecting a set of locations in a C function and may associate additional information to these locations, by setting values (ACSL terms) for some special variables, that we call *meta-variables*. Section 2.3 presents examples of contexts with their corresponding meta-variables, and Figure 2 gives a full example of one of them.
- P is an ACSL predicate over a subset of $\mathcal{G} \cup \mathcal{M}$, where $\mathcal{G}$ is the set of variables defined in the global scope of the program. Given a location-environment pair (e, m) returned by θ, we can construct an ACSL property, denoted P_m, where every meta-variable v is replaced by $m(v)$.

```
int A, B, C;
int level, secret_size; // level is the current confidentiality level
int* secret; // a secret array with secret_size elements

void main(); // main entry point
void def_level(int val); // set confidentiality level
void backdoor_root(); // backdoor that can always access secret
int read_secret(unsigned n); // return secret only if level is sufficient
/*@
  //A always remains equal to B in function main
  meta \prop, \name(AB_same), \targets({main}), \context(\strong_invariant),
    A == B;
  //The level can only be modified in def_level or backdoor_root
  meta \prop, \name(modif_level),
    \targets(\diff(\ALL, {def_level, backdoor_root})),
    \context(\writing), \separated(\written, &level);
  //The secret can only be read if level is at least ROOT_LEVEL
  meta \prop, \name(can_read_secret), \targets(\ALL), \context(\reading),
    \separated(\read, &secret[0 .. secret_size - 1]) ∨ level ≥ ROOT_LEVEL;
  //Function backdoor_root is never called
  meta \prop, \name(no_backdoor), \targets(\ALL), \context(\calling),
    \separated(\called, backdoor_door); */
```

Fig. 1: Examples of meta-properties and contexts

Given the target set F, the context $\mathcal{C} = (\mathcal{M}, \theta)$ and the property P, the meta-property $(F, \mathcal{C}, P)$ is interpreted as:

$$\forall f \in F,\ \forall (e, m) \in \theta(f),\ P_m \text{ holds on } e.$$

In other words, for every function of F and for every point in this function selected by θ, P must hold at this point when its meta-variables have been instantiated according to the environment mapping.

One simple example of meta-property, with no meta-variable, is the specification of a predicate P as a *strong invariant*: $M_{si}(P) = (\mathcal{F}, (\emptyset, \theta_{si}), P)$ where θ_{si} returns every edge of the CFG with a trivial environment mapping.

2.2 ACSL Syntax and First Examples

To specify meta-properties in Frama-C, we propose an extension of ACSL for explicitly providing each element of the triple $(F, \mathcal{C}, P)$. It is mandatory to *name* the property. This allows traceability between the meta-property and the generated ACSL assertions. Figure 1 gives a few examples of meta-properties that are detailed below. Concretely, a meta-property is defined as follows:

```
/*@ meta \prop, \name(...), \targets(...), \context(...), P; */
```

The target set is provided using the usual set syntax of ACSL. It can be explicit ($\{\texttt{f1}, \ldots, \texttt{fn}\}$), or use set operators such as `\union` or `\inter`. We also added the `\diff` operator for set difference, which does not exist in ACSL.

Since the goal for meta-properties is to be able to easily and automatically specify properties on large code bases, giving the explicit set of targets is rarely a practical solution. Instead we provide a special variable `\ALL` which refers to

$\mathcal{F}$ (the set of all functions in the program), and is very convenient, along with the `\diff` operation, to specify target sets of the form "all functions except...".

As an additional way to ease the delimitation of the targets, we provide two constructs `\callees` and `\callers`. `\callees(f)` is the set containing `f` and all functions (transitively) called by `f`. `\callers(f)` is the dual set containing `f` and all functions that (transitively) call `f`. It is especially useful when dealing with programs with clearly defined entry points.[1]

The combination of these simple constructs allows for a convenient way to specify the scope of a meta-property without having to rewrite the target set when new functions are added to the implementation.

2.3 Available Contexts

We define several contexts that the user can use when writing a meta-property, by indicating the context name in the `\context(...)` field. It turns out that these few and simple contexts, combined with the expressiveness of ACSL itself, are enough to write quite interesting properties.

Weak Invariant, Pre/Post-condition The `\precond` context returns only the starting edge of the CFG with no meta-variable, while `\postcond` does the same with the ending edges, and `\weak_invariant` combines both.

Strong Invariant As mentioned earlier, the `\strong_invariant` context simply provides a contextualization mapping returning every edge of the CFG of a given function without defining any meta-variables. However, it is sometimes necessary even for a strong invariant to be temporarily broken. Equality between two variables (e.g. `AB_same` in Figure 1) is an example of that, as there is no way to change the value of the two variables in a single instruction. To overcome this issue, we add a `lenient` modifier that can be applied on a block of code to exclude the edges inside it from the scope of strong invariants.

Upon Writing the `\writing` context is the pair $(\{\texttt{\textbackslash written}\}, \theta_w)$, where θ_w returns all edges of the CFG of a given function leading to an instruction that writes into the memory (through e.g. the assignment of a variable) with an environment that maps `\written` to the address modified by that instruction. The action of mapping θ_w is illustrated in Figure 2.

Since `\written` is a meta-variable of this context, it can then be used by the predicate P to form a useful meta-property. A simple example would be to forbid any local modification of some global variable, as shown by meta-property `modif_level` on Line 14 of Figure 1. It states that for any function that is not `def_level` or `backdoor_root`, whenever some memory location is modified locally, it must be unrelated to the global variable `level`. In ACSL, the `\separated(p1,p2)` predicate states that the memory locations referred to

[1] This feature relies on the Frama-C plugin Callgraph, which makes gross over-approximations of these sets in the presence of indirect calls (i.e. function pointers).

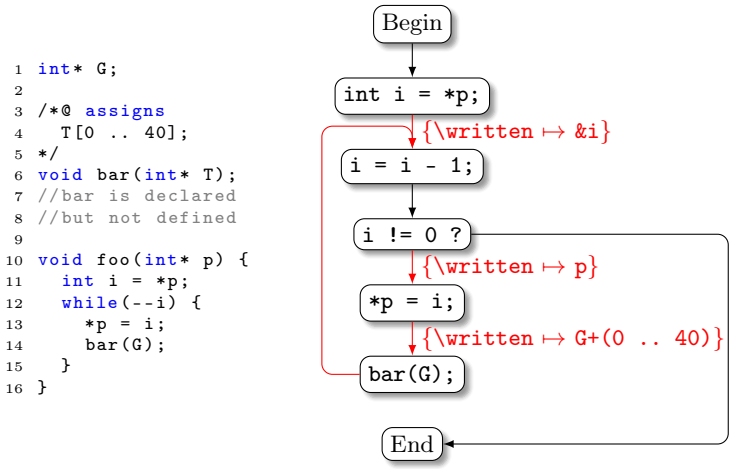

Fig. 2: Illustration of the contextualization mapping θ_w (for `\writing` context) for function `foo`. Red arrows indicate the edges returned by θ_w (leading to writing operations), and the corresponding environment mapping is shown by a label on such an edge (indicating that meta-variable `\written` is mapped to the (potentially) modified variable(s)).

by given (sets of) pointers `p1,p2` are physically disjoint (or *separated)*. Since we consider only local modification, a call to `def_level` inside another function not allowed to modify `level` does not violate `modif_level`, even if `def_level` itself modifies `level`. Thus, `modif_level` can be seen as enforcing the proper encapsulation of `level`.

Upon Reading The `\reading` context is identical to the `\writing` context except that it selects all edges leading to an instruction that *reads from* the memory and associates a meta-variable `\read` with the addresses being read by the instruction. It is illustrated by property `can_read_secret` in Figure 1.

Upon Calling In a similar fashion, the `\calling` context selects all edges preceding a function call and maps a meta-variables `\called` with the function (or function pointer) that is being called. It is used in the `no_backdoor` property.

3 Instantiation of Meta-Properties

Several existing Frama-C plugins provide useful and efficient analysis of ACSL-annotated C code, such as deductive verification [4] or runtime assertion checking [5]. Following the usual Frama-C approach of *tool collaboration*, we wish to take benefit of existing analyzers without re-implementing them for meta-properties. To do that, we designed a way to transform meta-properties into plain ACSL annotations while keeping links between the original meta-properties and

```
meta \prop,
  \name(G_is_constant),
  \targets({foo}),
  \context(\writing),
  \separated(\written, G);
```

```
void foo(int* p) {
  int i = *p;
  while(1) {
    /*@assert \sep(&i, G);*/
    i = i - 1;
    if(i ≠ 0) break;
    /*@assert \sep(p, G);*/
    *p = i;
    //Invalid assertion
    /*@assert \sep(G+(0..40), G);*/
    bar(p);
  }
}
```

Fig. 3: A meta-property and its instantiation for function `foo` of Figure 2

their ACSL translation. Hence, the existing tools can understand and analyze the translation and their results for the translated ACSL annotations can then be interpreted in terms of meta-properties.

Since meta-properties have more expressive power than ACSL, it is often impossible to transform a meta-property into a single ACSL annotation. In some cases, a meta-property is translated into function contract clauses (e.g. for weak invariants) but in most cases it has to be captured by assertions inserted directly into the body of a function. More precisely, we use the the ACSL annotation `/*@ assert P; */`, which means that `P` must hold at the particular point where the assertion is inserted. This allows a verification process that sticks closely to the definition of a meta-property $(F, \mathcal{C}, P)$: for each function f in F, once the set of edges of f defined by $\mathcal{C}$ is determined, it suffices to assert that for each edge/environment pair (e, m), P_m (P with meta-variables substituted with their values in m) must be valid in the context of e. The transformation process materializes this assertion by inserting a concrete ACSL annotation `/*@ assert` P_m`; */` at the point between two instructions corresponding to the edge e. Note however that P_m must be correctly typed at this point, as discussed in Section 4.1. This `assert` is called an *instantiation* of the meta-property. Translation of a meta-property then consists in instantiating it for each edge/environment pair, as illustrated by Figure 3, showing a simple *Upon Writing* meta-property and its translation for function `foo` of Figure 2. Note that, as most Frama-C plugins, we rely on the presence of an `assigns` clause specifying the side effects of a function whose body is unknown (as function `bar` in Figure 2).

This translation technique can be performed automatically and is correct by definition *with respect to the semantics given to meta-properties*: there is a one-to-one mapping between CFG edges and assertions in the instrumentation. We implemented it in the MetAcsl plugin of Frama-C.

Performance considerations While the proposed technique is simple, it entails that a meta-property is instantiated for each selected edge in each target function. Thus, the number of instantiations can quickly become high enough (for example when using the *Strong Invariant* context) to become a problem for the

Frama-C plugins that are expected to analyse the translated program, resulting in potentially long analysis times or loss of precision in the results.

However, we have observed that when a meta-property has been instantiated, there are a lot of cases where the resulting assertion is trivial to prove or disprove. For example if P is `\separated(\written, &A)` and P_m is `\separated(&B, &A)` where `A` and `B` are different variables (thus separated by definition), this instantiation is trivially valid and its actual insertion can be skipped.

Thus, MetAcsl performs a *simplification phase* for assertions where simple patterns such as the one mentioned above are recognized and replaced by their truth value, which is then propagated through the property. Hence, the instantiations left in the code are those that could not be simplified, and for which other plugins should attempt a more thorough verification. The quantitative evaluation of this simplification is discussed in Section 5.

4 Extensions of Meta-Properties

The basic definition of meta-properties presented in Section 2 enables the specification of many useful properties, as seen in the examples. However, the case studies (described in Section 5) showed that it had several limitations, in both expressiveness and adaptability to the structure of programs. To address these limitations, we introduce some extensions to meta-properties.

4.1 Typing Issues

A context is defined as a pair $(\mathcal{M}, \theta)$ where $\mathcal{M}$ is the set of meta-variables, i.e. names that are mapped to different ACSL terms at each point defined by θ. Each meta-variable can then be used in the property P to refer to those terms.

However, notice that while each mapped ACSL term has a well-defined type, the meta-variable itself does not. Or rather, its type is the union of types of the mapped terms. Yet, this set of types is not known in advance when writing a meta-property. Thus, nothing can be assumed for example about the type of `\written` when specifying a meta-property in the *Upon Writing* context, except that it is always a pointer type (since it refers to the set of addresses that are modified). Any other assumption would create a risk of typing error. For example, assuming there exists a C structure `struct S` with an `x` field, the presence of `\written->x == 0` in P would make any instance of the meta-property related to an assignment to a location of another type ill-defined, while `\separated(\written, &global_variable)` would not, since the `\separated` predicate only requires its parameters to be of pointer types.

While this suffices to express interesting properties such as separation, it does not allow reasoning about the value of the meta-variable. To address this issue, we introduce a construct to make assumptions about the type of a meta-variable while having a safeguard in case these assumptions were wrong. More precisely, we add two functions `\tguard` and `\fguard` that take an unsafe predicate (where a typing error might happen), behave as the identity if there is no

error, and return respectively `\true` and `\false` otherwise. This allows the user to specify the previous example as `\tguard(\written->x == 0)`. If a particular instance of `\written` is of the expected type `struct S` then its field is checked, else the property defaults to `\true` (i.e. we are only interested in modifications on locations of that type). Had `\fguard` been chosen instead of `\tguard`, any instantiation of that meta-property on a type that is not `struct S` would have defaulted to `\false`, effectively forbidding write operations to those types.

Intuitively, these functions should be used to guard any predicate that may be invalid for some instantiation of the meta-property. The default value to choose should reflect if these failures are expected in some cases or not: in our example, `\tguard` allowed and ignored failures while `\fguard` did not.

4.2 Labels in Meta-Properties

While meta-properties allow specifying a property when some event defined by the context happens (e.g. a memory operation) and the safeguarding constructs enable that property to talk about the values of the meta-variables, sometimes we need to talk about the *effect* of the memory operation on these values.

For example, one may want to globally guarantee that some initially null global variable `G` is initialized only once to a strictly positive value. However it is not possible to specify this without a mean to refer to `G` *before* and *after* each modification, which would be needed to characterize our notion of initialization.

In ACSL, one can use the `\at(expression, label)` construct to refer to the value of an expression *at a specific point of the program* identified by a *label*. An expression used without `\at` refers to its value at the point where it appears (which can be used explicitly with the `Here` label). There also exists two built-in labels `Pre` and `Post` referring respectively to the state before and after the current function.[2] Furthermore, any previously defined C label can be used as a label in `\at`. The `\at` construct can naturally be used in P, with labels `Pre`, `Post` and `Here` keeping their meanings.

To tackle the aforementioned problem, we define two additional labels that are specific to meta-properties and their context: `Before` and `After` that are used to refer to the states before and after the statements considered by the context, if any. These special labels may be mapped to actual ACSL labels by the contextualization mapping of a context when it makes sense to do so.

For example, the `\precond` does not define them. The `\writing` context maps `Before` to `Here` (since by definition the edge returned by the context *precedes* a statement modifying the memory) and `After` to a C label inserted after the statement modifying the memory.

With these labels, we can now write our previously problematic initialization meta-property:

```
1 meta \prop, \name(G_unique_initialization),
2   \targets(\ALL), \context(\writing),
3   \separated(\written, &G) ∨ (\at(G, Before) == 0 ∧ \at(G, After) > 0);
```

[2] Technically, `Post` can only be used in `assigns` statements or contract post-conditions.

which means that each instruction either does not modify `G` or modifies it such that its value is 0 before the modification and strictly positive after it.

4.3 Referring to Non-Global Values

As meta-properties are *global* properties that are not declared in the scope of any particular function, they can only refer to global variables and meta-variables. This is a strong limitation to the kind of properties that can be written, as some programs have few interesting objects declared in the global scope and typically pass them as arguments. To tackle this issue, we came up with two different mechanisms: the `\formal` construct and the notion of *local binding*.

Referring to function parameters If there is an object present in every target function of a meta-property, but as a consistently named function parameter (which is called a *formal*) instead of a global variable, we introduce the `\formal` keyword to refer to such a parameter in the property P of a meta-property. When `\formal(some_param)` appears in a meta-property, each instantiation of the meta-property triggers the check that `some_param` is indeed a formal of the current target function. If it is, the `\formal` call is safely replaced by `some_param`. Otherwise, a typing error is triggered at the point where it is used. Thus, `\formal` is best used when combined with the safeguarding constructs `\tguard` and `\fguard` (Section 4.1), since it allows the specification engineer to *assume* that a formal is consistently defined in every target function and use it in a property, but to safely default to a conservative property if this assumption is wrong.

```
meta \prop, \name(),
    \targets(\ALL),
        \context(\calling),
    \tguard(
        \separated(\called,
            \formal(pre_process))
        ∨ \separated(do_not_call,
            \formal(pre_process))
    );
```

For example, the above property specifies that if a function in the programs takes a function pointer `pre_process` as a parameter, then it can only be called if it is distinct from a `do_not_call` function. If this parameter does not exist in a function, then the property defaults to `\true` since there is nothing to verify.

Referring to bound names If it is not possible to rely on a consistent naming of formals across functions, we introduce a notion of *binding* to overcome this difficulty with some help from the user.

We introduce two special functions, `\bind` and `\bound`. The first one is to be used outside of a meta-property, in the body of a C function, to *bind* a name to the value of a C expression at that point. This name can then be used in a meta-property to formulate an interesting property about the value it refers to. A name can actually be bound multiple times to different value at different points of a program, meaning that the name inside a meta-property refers to the whole set of associated values. The whole process is illustrated in Figure 4. Notice that the bound values are constant but may be pointers referring to changing

memory. We are then specifying a property across all the memory states of the different instantiations, which makes `\bound` a meta-variable.

```
1  int lock;
2  int* create_cell() {
3    char* c = malloc(1);
4    //@ meta \bind(c, cells);
5    return c;
6  }
7  int safe_modify_cell
8  (int* cell, int val) {
9    if(!lock) {
10     lock = 1;
11     *cell = val;
12     lock = 0;
13     return 0;
14   }
15   else return -1;
16 }
17 void unsafe_modify_cell
18 (int* cl, int val) {
19   *cll = val;
20 }
21
22 /*@ //Pointers returned by create_cell
23   //are not modified if the lock is on
24   meta \prop,
25   \name(cell_modif_is_critical),
26   \targets(\ALL), \context(\writing),
27   \separated(\written, \bound(cells))
         ∨ lock;
28 */
```

Fig. 4: *Bindings* usage example

To actually instantiate (as described in Section 3) a meta-property with bindings, the program must be further instrumented using *ghost code*.

Ghost variables are declared for specification purposes only and cannot be used by the original C code, while ghost statements may only modify ghost variables. Thus, ghost code altogether cannot modify the original behavior of the code but may facilitate verification.

For each bound name, we allocate an associated ghost global array whose role is to store the set of associated values. Consequently, each instance of `\bind(v, n)` is replaced by a ghost instruction adding `v` to the array `n_set` associated to `n` and every instance of a predicate $P(n)$ involving a bound name is replaced by a quantified predicate $\forall v \in$ `n_set`$, P(v)$. This is illustrated in Figure 5, which is the translation of Figure 4. Notice that the type of the array is inferred from the `\bind` calls. As such, it is the responsibility of the user to ensure that every bound value is of the same type and to use the bound name appropriately.

5 Case studies and their Specification

We applied our technique to two different case studies for the purpose of evaluating its relevance on actual code and properties. First, we describe the content of these case studies and how useful properties about them are specified using meta-properties.[3] Then in Section 6, various assessment techniques are used to check the validity of the implementation with respect to these meta-properties.

5.1 Confidentiality

The first case study, which was submitted by an industrial partner, deals with a confidentiality-oriented page management system. We assume a system where a confidentiality level is associated to each memory page. Two different pages may have the same level but the set of confidentiality levels must be totally ordered.[4]

[3] The case studies and their specifications are available at `https://huit.re/metatap`.
[4] We assume a total order for simplicity, but it would also work with a partial one.

```
int lock;
//@ ghost char* cells_set = NULL;
//@ ghost size_t cells_set_size = 0;
int* create_cell() {
    char* c = malloc(1);
    //@ ghost add_to_array(cells_set, c);
    return c;
}
int safe_modify_cell(int* cell, int val) {
    if(!lock) {
        /*@ assert ∀ size_t i; i < cells_set_size ⇒
              \separated(&lock, cells_set[i]) ∨ !lock; */
        *cell = val;
        lock = 1;
        //@ assert ∀ i; ... \separated(cell, cells_set[i]) ∨ !lock;
        *cell = val;
        //@ assert ∀ i; ... \separated(&lock, cells_set[i]) ∨ !lock;
        lock = 0;
        return 0;
    }
    else return -1;
}
void unsafe_modify_cell(int* cl, int val) {
    //@ assert ∀ i; ... \separated(cl, cells_set[i]) ∨ !lock;
    *cl = val;
}
```

Fig. 5: Translation of Figure 4

We call *agent* any entity (a process, for example) which may happen to read or write from such pages, and give to each agent a confidentiality level as well.

The two basic guarantees that such a system should offer are:

C_1: An agent can never **read** from a page with a confidentiality level **higher** than its own (to preserve the confidentiality of the data written on the page),
C_2: An agent can never **write** to a page with a level **lower** than its own (to prevent the agent's data from being read by lower agents in the future).

Notice that these properties ensure *confidentiality* but not *integrity*, which is not considered here but could be similarly specified.

We wrote a simple implementation of this case study, where the system is modelled by a stateful API of functions to allocate, free, write to or read from pages. The confidentiality level of the calling agent is represented by a global variable, which is assumed to be securely modified when the context changes.

There are several other properties needed for C_1 and C_2 to be useful in ensuring confidentiality:

C_3: The confidentiality level of an allocated page remains constant,
C_4: The allocation status of a page can only be modified by the allocation and de-allocation functions,
C_5: Non allocated pages are neither accessed nor modified,
C_6: Non allocated pages do not retain old data.

We also consider an extension of this system introducing *encryption* as a means to decrease the confidentiality level of a page. Two functions to encrypt and decrypt a page are added to the API with a key based on the confidentiality level of the caller, and we weaken C_3 into:

```
1  //Never read from a higher confidentiality page
2  meta \prop, \name(C_1), \targets(\ALL),
3    \context(\reading),
4      forall_page(p,
5        page_allocated(p) ∧ user_level < page_level(p) ⇒
6        \separated(page_data(p), \read)
7      );
8  //The confidentiality of an allocated page is constant outside of encryption
9  meta \prop, \name(C_3'), \targets(\diff(\ALL, {page_encrypt, page_decrypt})),
10   \context(\writing),
11     forall_page(p,
12       page_allocated(p) ⇒ \separated(&p->confidentiality_level, \written)
13     );
14
15 //The content of a free page is always null
16 meta \prop, \name(C_6), \targets(\ALL),
17   \context(\strong_invariant),
18     forall_page(p, !page_allocated(p) ⇒ clean_page(p));
```

Fig. 6: Specification of some confidentiality properties using meta-properties

C_3': The confidentiality level of an allocated page remains constant, except in encryption/decryption functions.

All of these properties can be expressed using meta-properties, as illustrated in Figure 6 where properties C_1, C_3' and C_6 are specified. The `forall_page` predicate is a formula-shortening macro which quantifies over the globally-stored array of pages (both free and allocated).

5.2 Smart house

The second case study models a smart house command system on which we tried to specify and verify interesting safety and security properties.

The house is modelled as a set of rooms, each containing a door that can be locked or unlocked by authorized users, a window that can be opened or closed and an AC system that can be enabled or disabled. There is also an alarm that can be triggered by anyone in case of emergency.

We assume each room contains a terminal authenticating users and relaying their instructions to a central command system. The system is again modelled as a stateful API, this time with a single entry point where instructions from terminals are received and processed. We also add some administration functions that should not be called by terminals.

Some desirable properties that we specified for this system are:

S_1: Every door is unlocked when the alarm is ringing,
S_2: The AC system cannot be enabled when the window in the room is open,
S_3: A door can only be unlocked by authorized users,
S_4: The alarm cannot be silenced by users,

Their formalization into meta-properties is represented in Figure 7. Notice the use of both the `\formal` construct (Section 4.3) to refer to the parameters

```
#define USER_SET (\callees(receive_command))
/*@
meta \prop, \name(S_2), \targets(\ALL),
  \context(\strong_invariant),
  forall_room(r,
    r->window_state == 0
   ⇒ r->ac_state == AC_DISABLED
);

meta \prop, \name(S_3),
    \targets(USER_SET),
    \context(\writing),
    forall_room(r,
        \at(r->door_lock_state, Before) ≠ 0
        ∧ \at(r->door_lock_state, After) == 0
        ∧ \fguard(user_permissions[\formal(uid)] < r->clearance_needed)
        ⇒ \separated(\written, &r->door_lock_state)
    );

meta \prop, \name(S_4), \targets(USER_SET),
    \context(\writing),
    \at(alarm_status, Before) == ALARM_NONE
    ∨ \at(alarm_status, After) ≠ ALARM_NONE
    ∨ \separated(\written, &alarm_status);
*/
```

Fig. 7: Specification of some smarthouse-related properties using meta-properties

of the different functions (combined with `\fguard` to default to a false predicate if there is no such parameter), as well as the `\at` construct with labels `Before` and `After` (Section 4.2) in S_3 to express the fact that a door is locked before an instruction and unlocked after it, needed to express the notion of unlocking. Finally, notice the use of the `\callees` function in `USER_SET` to refer to the set of every function called by the single entry point (`receive_command`), in order to exclude the administration functions (if they are indeed not called) from the scope of the meta-properties.

6 Assessment of the Case Studies

Having two case studies specified with meta-properties, we want to evaluate the ability to assess them with the usual Frama-C tools, after translating meta-properties into native ACSL with MetAcsl. To that end, we wrote a correct C implementation of the different functions for both the confidentiality and smart house case studies. Then, to increase the number of benchmarks, we used a Frama-C plugin[5] to generate mutations of this correct implementation, providing a set of modified implementations, potentially invalid with respect to the meta-properties. In this way, we obtain respectively 126 and 69 mutants for the confidentiality and the smarthouse case study. The mutations consist in the replacement of binary operators, the negation of conditions and the modification of

[5] See `https://github.com/gpetiot/Frama-C-Mutation`.

Number of:	functions	meta-properties	generated asserts	gen. asserts with simplif.	invalid mutants/ total mutants
Confidentiality	11	11	408	273	42/126
Smarthouse	14	7	156	87	19/69

Fig. 8: Statistics about the MetAcsl instrumentation on case studies

Case study	Confidentiality		Smarthouse	
Assessment type	Wp	E-ACSL	Wp	E-ACSL
False Positives	0	29	0	6
False Negatives	0	0	0	0
Interrupted (RTE)	N/A	19	N/A	11

Fig. 9: Assessment of automatic approaches, relative to a manual verification

numerical values. They simulate frequent programming errors in the code. The specification for the mutants remains the same as for the initial implementation.

For each mutant, we manually check if the introduced mutation violates one of the meta-properties of the case study. If so, the mutant is considered invalid. The proportion of invalid mutants is reported in Figure 8 along with some quantitative information about the case studies and their instrumentation by MetAcsl. Here we can observe that the simplification phase described in Section 3 significantly reduces the number of generated assertions, thus easing the job of the tools that are subsequently run on the resulting translated programs.[6]

For each benchmark (initial version or one of the mutants, including all valid and invalid mutants), we first apply MetAcsl to generate an instrumented C program. We wish then to investigate whether, thanks to the instrumentation with MetAcsl, different Frama-C tools are able to assess the validity of the benchmarks with respect to the meta-properties.

We test two existing assessment techniques, namely deductive verification with the Wp plugin and runtime verification with the E-ACSL plugin. For both plugins, Figure 9 indicates the number of false positives (cases where the mutant is invalid but no violation was detected) and false negatives (cases where the mutant is valid, but flagged as violating a meta-property; this can happen in case of proof failure or faulty translation). We detail both techniques in the rest of the section.

6.1 Deductive Verification

Deductive verification allows users to formally prove that the implementation of a function is correct with respect to its specification. If a function is specified by annotations (a contract, invariants and/or assertions), then logical formulas encoding the semantics of these annotations and known as *proof obligations*

[6] For example, simplification saves 8 seconds on the deductive verification of the correct confidentiality implementation (for a total of 24 seconds).

(POs) can be generated and given to automated theorem provers. If all POs are validated, the body of the function fulfills its specification. This technique is implemented by Frama-C plugin Wp.

We attempt to run Wp on each benchmark. While a proof success is definitive, a proof failure may have different causes: the property to be proved may be false, there could be insufficient assumptions available to the prover or it could simply exceed the capacity of the prover in its allocated time. Thus if every proof failure is classified as a judgement of invalidity, false negatives are to be expected. To mitigate this phenomenon, we first manually annotated the case studies with partial function contracts for the correct implementations to be successfully proved.

The results are encouraging. Every valid mutant was successfully proved as valid, and the proof failed for each invalid mutant (see Figure 9)[7], thus confirming the correctness of the transformation. These results demonstrate that the spec-to-spec translation with MetAcsl creates a convenient, fully automatic toolchain for deductive verification of global properties in Frama-C. As usual for deductive verification, some additional annotations were necessary to prove the different functions (respectively 40 and 5 lines of specification were needed, loops being the main point of effort) but their number was much smaller than the number of relevant assertions automatically generated from the meta-properties.

6.2 Runtime Verification on Test Cases

We now wish to study if it is also possible to verify meta-properties at runtime—without any additional annotations—thanks to the E-ACSL [5] plugin for runtime assertion checking. It automatically translates an ACSL-annotated C program into another program that fails at runtime if an annotation is violated.

Since our two case studies are APIs without any `main` function, we wrote small test suites of complete programs that can be actually compiled and executed. In both cases, they contain simple functional tests and do not aim at covering every possible usage case. They feature sequences of respectively 40 and 20 calls to the APIs.

We then applied runtime assertion checking to the execution of every instrumented benchmark on all tests of both test suite. The results (Figure 9) are also promising and allowed us to identify several issues and future work directions.

The additional row refers to cases where the generated binary detected a violation of a safety property [8], thus stopping the execution and preventing us to know if a meta-property violation would have been detected or not. In the future, it would be desirable to filter out safety-violating mutants, and only keep mutations simply modifying the semantics of the code.

There are no false negatives, confirming that the instrumentation of both MetAcsl and E-ACSL does not introduce any bug. There is a significant

[7] The last row is not relevant for deductive verification, see Section 6.2.

[8] E-ACSL add checks to ensure that no runtime error (segfaults, overflow, ...) will occur and stops the program upon violation.

number of false positives (incorrect mutants for which no test failed). There are several reasons for this. First, our initial test suites are not complete and some mutants are not killed by these tests. In the future, we plan to address this by using the StaDy [6] plugin, which combines static and dynamic verification and allows the automatic generation of test cases that can exhibit counter-examples for invalid properties. The second reason is that E-ACSL only supports a subset of ACSL: some properties involving complex constructions such as the `\at` keyword are simply ignored by E-ACSL, thus they cannot possibly be violated at runtime. This support should be improved in the future.

This study demonstrates that it is easy to check meta-properties at runtime without extra annotation effort thanks to the combination of MetAcsl and E-ACSL, as long as the specified properties are supported by the tools. This is especially useful for properties that are not easily tractable with deductive verification: for example, a property using bindings (Section 4.3) might be very difficult to verify using Wp without writing extensive function contracts, while it can be immediately tested with E-ACSL.

7 Related Work

ACSL is a specification language inspired by previous efforts such as JML [7], a behavioural interface specification language for Java. JML has similar limitations regarding the expression of global properties on a software module, and we believe that our approach could be useful in this context as well. However a subset of meta-properties is already expressible in JML, such as weak invariants (using JML class invariants). Another high-level specification feature of JML is the notion of *constraint*, which allows the specification of a property relating the states before and after every method of a class, or a given set of methods (similar to our notion of target set).

The idea of extending a contract-based specification language to support high-level properties has been explored before. For example, Cheon and Perumandla [8] extended JML, allowing the specification of *protocols*, i.e. properties related to the order of call sequences. Protocols could be specified using the `\calling` context of meta-properties and ghost code to model an automaton, but may not be as usable as the simple syntax provided by this work. Another such example is the work of Trentelman and Huisman [9], which extends JML to enable the expression of temporal properties. Meta-properties can express a subset of temporal properties with the extension detailed in Section 4.2, but this is not the aim of the tool (the Frama-C plugins Aoraï [10] and CaFE [11] are already devoted to this task).

Within Frama-C, extending ACSL by writing a plugin that translates the extension back into normal ACSL has been used previously for the support of relational properties with Rpp [12] or temporal logic with Aorai [10].

The proposed transformation technique is related to the work of Pavlova et al. [13] as they generate annotations whose verification implies the validity of a high-level property as well. However, their specified properties are in the form of pre-/post-conditions on a well-defined set of *core* functions that are then

propagated throughout the code, while we define properties that are not always pertaining to some core functions and use a simpler propagation method.

The general idea of defining a high-level concept in the global scope and then *weaving it* into the implementation is analogous to the Aspect-Oriented Programming (AOP) [14] paradigm, as meta-properties can be seen as cross-cutting concerns on the specification side rather than the implementation side. Contexts can then be related to *pointcuts*, which in AOP are a set of control flow points where the code needed by the concern should be added.

8 Conclusion and Future Work

We proposed in this paper a complete description of a new specification mechanism for high-level properties in Frama-C. Meta-properties provide a useful extension to function contracts, offering the possibility to express a variety of high-level safety- and security-related properties, and reducing the risk of errors inherent to the manual specification of such properties, especially when updating the program specification or code. Today, meta-properties are capable to express the different types of high-level properties that motivated this work (e.g. isolation properties for verification of a hypervisor, confidentiality-oriented security properties, various global invariants, etc.).

The extensions to meta-properties we presented are helpful for expressing richer properties (e.g. referring to different states) on a larger class of C programs (by allowing the properties to refer to objects that are not necessarily in the global scope), as demonstrated for the specification of the two case studies.

We provided an automatic transformation-based method to enable the assessment of the meta-properties and showed that the result can successfully be assessed by existing deductive verification and runtime assertion checking techniques. This enables both users ready to put some effort into the specification of their program and users with a complete test suite, to assess the validity of their program with strong levels of confidence.

Finally, we emphasize that our goal is to propose a *pragmatic class of properties amenable to a high level of automation*; we do not claim they offer more expressiveness than other classes of properties on the logical level.

Future Work We plan to perform a formalization and a formal soundness proof for our transformation technique, thereby allowing MetAcsl to be reliably used for critical code verification. There is a plan to tackle existing industrial case studies to demonstrate the ability of meta-properties to specify programs that are not necessarily verification-friendly. Finally, we wish to refine the transformation technique in order to allow the proof of the generated specification to scale better when the number of meta-properties or the size of the code increases.

Acknowledgment This work was partially supported by the project VESSEDIA, which has received funding from the EU Horizon 2020 research and innovation programme under grant agreement No 731453. This work was also partially supported by ANR (grant ANR-18-CE25-0015-01). The work of the first author was partially funded by a Ph.D. grant of the French Ministry of Defense. Many thanks to the anonymous referees for their helpful comments.

References

[1] F. Kirchner, N. Kosmatov, V. Prevosto, J. Signoles, and B. Yakobowski. "Frama-C: A software analysis perspective". In: *Formal Aspects of Computing* (2015), pp. 573–609.

[2] P. Baudin, P. Cuoq, J.-C. Filiâtre, C. Marché, B. Monate, Y. Moy, and V. Prevosto. *ACSL: ANSI/ISO C Specification Language*. `https://frama-c.com/acsl.html`. 2018.

[3] V. Robles, N. Kosmatov, V. Prevosto, L. Rilling, and P. Le Gall. "MetAcsl: Specification and Verification of High-Level Properties". In: *Tools and Algorithms for the Construction and Analysis of Systems*. 2019, pp. 358–364.

[4] P. Baudin, F. Bobot, L. Correnson, and Z. Dargaye. *WP plugin manual*. `http://frama-c.com/wp.html`. 2010.

[5] J. Signoles, N. Kosmatov, and K. Vorobyov. "E-ACSL, a Runtime Verification Tool for Safety and Security of C Programs (tool paper)". In: *International Workshop on Competitions, Usability, Benchmarks, Evaluation, and Standardisation for Runtime Verification Tools*. 2017, pp. 164–173.

[6] G. Petiot, N. Kosmatov, B. Botella, A. Giorgetti, and J. Julliand. "How testing helps to diagnose proof failures". In: *Formal Aspects of Computing* (2018), pp. 629–657.

[7] G. T. Leavens, A. L. Baker, and C. Ruby. "JML: A Notation for Detailed Design". In: *Behavioral Specifications of Businesses and Systems*. 1999, pp. 175–188.

[8] Y. Cheon and A. Perumandla. "Specifying and Checking Method Call Sequences in JML". In: *International Conference on Software Engineering Research and Practice*. 2005, pp. 511–516.

[9] K. Trentelman and M. Huisman. "Extending JML Specifications with Temporal Logic". In: *International Conference on Algebraic Methodology and Software Technology*. AMAST. 2002, pp. 334–348.

[10] N. Stouls and J. Groslambert. *Vérification de propriétés LTL sur des programmes C par génération d'annotations*. Research Report (French). 2011.

[11] S. de Oliveira, V. Prevosto, and S. Bensalem. "CaFE: a model-checker collaboratif". In: *Approches Formelles dans l'Assistance au Developpement Logiciel*. 2017.

[12] L. Blatter, N. Kosmatov, P. Le Gall, V. Prevosto, and G. Petiot. "Static and Dynamic Verification of Relational Properties on Self-composed C Code". In: *International Conference on Tests and Proofs*. 2018.

[13] M. Pavlova, G. Barthe, L. Burdy, M. Huisman, and J. Lanet. "Enforcing High-Level Security Properties for Applets". In: *International Conference on Smart Card Research and Advanced Applications*. 2004, pp. 1–16.

[14] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J. Loingtier, and J. Irwin. "Aspect-Oriented Programming". In: *European Conference on Object-Oriented Programming*. 1997, pp. 220–242.