MetAcsl: Specification and Verification of High-Level Properties

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

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

Abstract. Modular deductive verification is a powerful technique capable to show that each function in a program satisfies its contract. However, function contracts do not provide a global view of which high-level (e.g. security-related) properties of a whole software module are actually established, making it very difficult to assess them. To address this issue, this paper proposes a new specification mechanism, called meta-properties. A meta-property can be seen as an enhanced global invariant specified for a set of functions, and capable to express predicates on values of variables, as well as memory related conditions (such as separation) and read or write access constraints. We also propose an automatic transformation technique translating meta-properties into usual contracts and assertions, that can be proved by traditional deductive verification tools. This technique has been implemented as a Frama-C plugin called MetAcsl and successfully applied to specify and prove safety- and security-related meta-properties in two illustrative case studies.

1 Introduction

Modular deductive verification is a well-known technique for formally proving that a program respects some user-defined properties. It consists in providing for each function of the program a *contract*, which basically contains a *precondition* describing what the function expects from its callers, and a *postcondition* indicating what it guarantees when it successfully returns. Logical formulas, known as *verification conditions* or *proof obligations* (POs), can then be generated and given to automated theorem provers. If all POs are validated, the body of the function fulfills its contract. Many deductive verification frameworks exist for various programming and formal specification languages. We focus here on Frama-C [1] and its deductive verification plugin Wp, which allows proving a C program correct with respect to a formal specification expressed in ACSL [1].

However, encoding *high-level* properties spanning across the entire program in a set of Pre/Post-based contracts is not always immediate. In the end, such high-level properties get split among many different clauses in several contracts, without an explicit link between them. Therefore, even if each individual clause is formally proved, it might be very difficult for a verification engineer, a code reviewer or a certification authority to convince themselves that the provided contracts indeed ensure the expected high-level properties. Moreover, a software product frequently evolves during its lifetime, leading to numerous modifications in the code and specifications. Maintaining a high-level (e.g. security-related) property is extremely complex without a suitable mechanism to formally specify and automatically verify it after each update.

The purpose of the present work is to propose such a specification mechanism for high-level properties, which we call *meta-properties*, and to allow their automatic verification on C code in Frama-C thanks to a new plugin called MetAcsl.

Motivation This work was motivated by several previous projects. During the verification of a hypervisor, we observed the need for a mechanism of specification and automatic verification of high-level properties, in particular, for global properties related to isolation and memory separation. Isolation properties are known as key properties in many verification projects, in particular, for hypervisors and micro-kernels.

A similar need for specific high-level properties recently arose from a case study on a confidentiality-oriented page management system submitted by an industrial partner. In this example, each page and each user (process) are given a confidentiality level, and we wish to specify and verify that in particular:

- (P_{read}) a user cannot read data from a page with a confidentiality level higher than its own;
- (P_{write}) a user cannot write data to a page with a confidentialtiy level lower than its own.

This case study will be used as a running example in this paper. As a second case study (also verified, but not detailed in this paper), we consider a simple smart house manager with several interesting properties such as: “a door can only be unlocked after a proper authentication or in case of alarm” or “whenever the alarm is ringing, all doors must be unlocked”. Again, these examples involve properties that are hard to express with function contracts since they apply to the entire program rather than a specific function.[1]

Contributions The contributions of this paper[2] include:

- a new form of high-level properties, which we call *meta-properties*, and an extension of the ACSL language able to express them (Sect. 2),
- a set of code transformations to translate meta-properties into native ACSL annotations that can be proved via the usual methods (Sect. 3),
- a Frama-C plugin MetAcsl able to parse C code annotated with meta-properties and to perform the aforementioned code transformations (Sect. 4),
- a case study: a confidentiality-oriented page system, where important security guarantees were expressed using meta-properties and automatically verified thanks to the code transformation with MetAcsl (Sect. 4).

---

[1] These examples are publicly available at `https://huit.re/metacas`.

[2] A longer version is available at `https://arxiv.org/abs/1811.10509`

```
 1 struct Page { //Page handler structure
 2   char* data; //First address of the page
 3   enum allocation status; //ALLOCATED or FREE (ensured by M1, lines 10-12)
 4   enum confidentiality level; /*Page level, CONFIDENTIAL or PUBLIC*/ }
 5 enum confidentiality user_level; //Current user process level
 6 struct Page metadata[PAGE_NB]; //All pages
 7 struct Page* page_alloc(void); //Allocates a page
 8 void page_read(struct Page*, char* buffer); //Reads a page
 9 void page_encrypt(struct Page*); //Encrypts a page in place, makes it PUBLIC
10 /*@ meta M1: ∀function f; \strong_invariant(f),
11     ∀ int page; 0 ≤ page < PAGE_NB ⇒
12     metadata[page].status == FREE ∨ metadata[page].status == ALLOCATED;
13   meta M2: ∀function f; //Only page_encrypt can change levels of allocated
         pages
14     ! \subset(f, {page_encrypt}) ⇒ \writing(f),
15     ∀ int page; 0 ≤ page < PAGE_NB ∧ metadata[page].status == ALLOCATED
16     ⇒ \separated(\written, &metadata[page].level);
17   meta M3: ∀function f; \reading(f), //Ensures Pread
18     ∀ int page; 0 ≤ page < PAGE_NB ∧ metadata[page].status == ALLOCATED
19     ∧ user_level == PUBLIC ∧ metadata[page].level == CONFIDENTIAL
20     ⇒ \separated(\read, metadata[page].data + (0 .. PAGE_LENGTH - 1));
21 */ //Meta-property ensuring  Pwrite is defined similarly to M3
```

Fig. 1: Partial meta-specification of a confidentiality case study

2 Specification of Meta-properties

A meta-property is a property meant to express high-level requirements. As such, it is not attached to any particular function but instead to a set of functions. It is thus defined in the global scope and can only refer to global objects.

To define a meta-property, the user must provide (i) the set of functions it will be applied to, (ii) a property (expressed in ACSL) and (iii) the *context*, *i.e.* a characterization of the situations in which they want the property to hold in each of these functions (everywhere in the function, only at the beginning and the end, upon writing in a variable, etc.). Furthermore, depending on the context, the property can refer to some special variables which we call *meta-variables*. Figure 1 features a few examples of meta-properties further explained below.

Let $\mathcal{F}$ denote the set of functions defined in the current program, and $\mathcal{P}$ the set of native ACSL properties. Formally, we can define a meta-property as a triple (c, F, P), where c is a context (see Sect. 2.2), $F \subseteq \mathcal{F}$ and $P \in \mathcal{P}$. Intuitively, we can interpret this triple as "$\forall f \in F$, P holds for f in the context c". For the meta-property to be well-formed, P must be a property over a subset of $\mathcal{G} \cup \mathcal{M}(c)$, where $\mathcal{G}$ is the set of variables available in the global scope of the program and $\mathcal{M}(c)$ is the set of meta-variables provided by the context c.

The actual MetAcsl syntax for defining a meta-property (c, F, P) is `meta [specification of F] c, P;` An example is given by property M_1 (cf. lines 10–12 in Figure 1), where $F = \mathcal{F}$, $c =$ `strong_invariant` and P is the predicate stating that the status of any page should be either `FREE` or `ALLOCATED`.

2.1 Target Functions and Quantification

Meta-properties are applied to a given *target set* of functions F defined as $F = F_+ \setminus F_-$ by providing explicit lists of considered and excluded functions $F_+, F_- \subseteq \mathcal{F}$. If not provided, F_+ and F_- are respectively equal to $\mathcal{F}$ and $\emptyset$ by default, *i.e.* the meta-property should hold for all functions of the program. F_- is useful when the user wants to target every function except a few, since they do not have to explicitly provide every resulting target function.

The MetAcsl syntax for the specification of F uses the built-in ACSL construction `\forall`, possibly followed by `\subset` with or without logic negation `!` (to express $f \in F_+$ and $f \notin F_-$). It can be observed in property M_2 (lines 13–16), where $F_+ = \mathcal{F}$ and $F_- = \{$`page_encrypt`$\}$ excludes only one function.

2.2 Notion of Context

The *context* c of a meta-property defines the states in which property P must hold, and may introduce *meta-variables* that can be used in the definition of P.

Beginning/Ending Context (Weak Invariant) A *weak invariant* indicates that P must hold at the beginning and at the end of each target function $f \in F$.

Everywhere Context (Strong invariant) A *strong invariant* is similar to a weak invariant, except that it ensures that P holds at *every point* [3] of each target function. For example, property M_1 specifies that at every point of the program, the status of any page must be either `FREE` or `ALLOCATED`.

Writing Context This ensures that P holds upon any modification of the memory (both stack and heap). It provides a meta-variable `\written` that refers to the variable (and, more generally, the memory location) being written to.

A simple usage of this context can be to forbid any direct modification of some global variable, as in property M_2. This property states that for any function that is not `page_encrypt`, the left-hand side of any assignment must be *separated* from (that is, disjoint with) the global variable `metadata[page].level` for any `page` with the `ALLOCATED` status. In other words, only the `page_encrypt` function is allowed to modify the confidentiality level of an allocated page.

An important benefit of this setting is a *non-transitive restriction of modifications* that cannot be specified using the ACSL clause `assigns`, since the latter is transitive over function calls and necessarily permits to modify a variable when at least one callee has the right to modify it. Here, since we only focus on *direct* modifications, a call to `page_encrypt` (setting to public the level of the page it has encrypted) from another function does not violate meta-property M_2.

Furthermore, modification can be forbidden *under some condition* (i.e. that the page is allocated), while `assigns` has no such mechanism readily available.

Reading Context Similar to the writing context, this ensures that the property holds whenever some memory location is read, and provides a meta-variable `\read` referring to the read location. It is used in property M_3 (lines 17–20), which expresses the guarantee P_{read} of the case study (see Sec. 1) by imposing a separation of a read location and the contents of allocated confidential pages when the user does not have sufficient access rights. As another example, an isolation of a page can be specified as separation of all reads and writes from it.

These few simple contexts, combined with the native features of ACSL, turn out to be powerful enough to express quite interesting properties, including memory isolation and all properties used in our two motivating case studies.

---

[3] More precisely, every *sequence point* as defined by the C standard.

3 Verification of Meta-properties

Figure 2 shows an (incorrect) toy implementation of two functions of Figure 1 that we will use to illustrate the verification of meta-properties M_1–M_3.

The key idea of the verification is the translation of meta-properties into native ACSL annotations, which are then verified using existing FRAMA-C analyzers. To that end, the property P of a meta-property (c, F, P) must be inserted as an assertion in relevant locations (as specified by context c) in each target function $f \in F$, and the meta-variables (if any) must be instantiated.

We define a specific translation for each context. For weak invariants, property P is simply added as both a precondition and a postcondition in the contract of f. This is also done for the strong invariant, for which P is additionally inserted after each instruction potentially modifying the values of the free variables in P[4] For example, Figure 3a shows the translation of M_1 on `page_alloc`. Our property (defined on lines 11–12 in Figure 1, denoted P_{M_1} here) is inserted after the modification of a `status` field (line 6) since the property involves these objects, but not after the modification of a `level` field (line 8).

```
 1 struct Page* page_alloc() {
 2   //try to find a free page
 3   struct Page* fp = find_free_page();
 4   //if a free page is found,
 5   //allocate it with current user
         level
 6   if(fp ≠ NULL) {
 7     fp->status = ALLOCATED;
 8     fp->level = user_level;
 9   }
10   return fp;
11 }
12 void page_read(struct Page* from,
       char* buffer) {
13   for(i∈ℕ = 0 ; i < PAGE_LENGTH ;
         ++i)
14     buffer[i] = from->data[i];
15 }
```

Fig. 2: Incorrect code w.r.t. M_2 and M_3

For *Writing* (resp. *Reading*) contexts, P is inserted before any instruction potentially making a write (resp. read) access to the memory, with the exception of function calls. In addition, each meta-variable is replaced by its actual value. For example, in the translation of M_2 on `page_alloc` (Figure 3b), the property is inserted before the two modifications of `fp`, and `\written` is replaced respectively by `fp->status` and `fp->level`. In this case M_2 does not hold. While its first instantiation (lines 4–6) is easily proved, it is not the case for the second one (lines 8–10). Indeed, there exists a `page` (the one being modified) that has a status set to `ALLOCATED` because of the previous instruction (line 7) and for which the `\separated` clause is obviously false. Hence, the assertion fails, meaning that the whole meta-property M_2 cannot be proved. The fix consists in swapping lines 6 and 7 in Figure 2. After that, all assertions generated from M_2 are proved.

A similar transformation for M_3 on `page_read` shows that the proof fails since the implementation allows an agent to read from any page without any check. Adding proper guards allows the meta-property to be proved. Conversely, if a meta-property is broken by an erroneous code update, a proof failure after *automatically* re-running METACSL helps to easily detect it.

---

[4] The AST is normalized so that every memory modification happens through an assignment. Then we conservatively determine if the object being assigned is one of the free variables of P: in presence of pointers, we assume the worst case.

```
1  /*@ requires P_M1;
2    ensures P_M1; */
3  struct Page* page_alloc() {
4    struct Page* fp =
         find_free_page();
5    if(fp ≠ NULL) {
6      fp->status = ALLOCATED;
7      /*@ assert P_M1;*/
8      fp->level = user_level;
9      //Line 8 cannot break P_M1
10   }
11 }
```

(a) Transformation for M_1

```
struct Page* page_alloc() {
  struct Page* fp = find_free_page();
  if(fp ≠ NULL) {
    /*@ assert ∀ int page; 0 ≤ page < PAGE_NB
      ⇒ metadata[page].status == ALLOCATED
      ⇒ \separated(fp->status,
           &metadata[page].level);*/
    fp->status = PAGE_ALLOCATED;
    /*@ assert ∀ int page; 0 ≤ page < PAGE_NB
      ⇒ metadata[page].status == ALLOCATED
      ⇒ \separated(fp->level,
           &metadata[page].level);*/
    fp->level = user_level;
  }
}
```

(b) Transformation for M_2

Fig. 3: Examples of code transformations for functions of Figure 2

4 Results on Case Studies and Conclusion

Experiments The support of meta-properties and the proposed methodology for their verification were fully implemented in OCaml as a Frama-C plugin called MetAcsl. We realized a simple implementation of the two case studies mentioned in Sect. 1) and were able to fully specify and automatically verify all aforementioned properties (in particular P_{read} and P_{write}) using MetAcsl. The transformation step is performed in less than a second while the automatic proof takes generally less than a minute.

Conclusion We proposed a new specification mechanism for high-level properties in Frama-C, as well as an automatic transformation-based technique to verify these properties by a usual deductive verification approach. The main idea of this technique is similar to some previous efforts e.g. [2] . Meta-properties provide a useful extension to function contracts offering the possibility to express a variety of high-level safety- and security-related properties. They also provide a verification engineer with an explicit global view of high-level properties being really proved, avoiding the risk to miss some part of an implicit property which is not formally linked to relevant parts of several function contracts, thus facilitating code review and certification. Another benefit of the new mechanism is the possibility to easily re-execute a proof after a code update. Initial experiments confirm the interest of the proposed solution.

Future Work We plan to establish a formal soundness proof for our transformation technique, thereby allowing MetAcsl to be reliably used for critical code verification. Other future work directions include further experiments to evaluate the proposed approach on real-life software and for more complex properties.

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. 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: *FAOC* (2015).

[2] M. Pavlova, G. Barthe, L. Burdy, M. Huisman, and J.-L. Lanet. “Enforcing High-Level Security Properties for Applets”. In: *CARDIS*. 2004.