**METACSL: Specification and Verification of High-Level Properties**

Virgile Robles¹, Nikolai Kosmatov¹, Virgile Prevosto¹, Louis Rilling², and Pascale Le Gall³

¹ CEA, List, Software Reliability and Security Lab, Gif-sur-Yvette, France
   firstname.lastname@cea.fr
² DGA Maîtrise de l’information, Bruz, France
   firstname.lastname@irisa.fr
³ Laboratoire de Mathématiques et Informatique pour la Complexité et les Systèmes
   CentraleSupélec, Université Paris-Saclay, Gif-Sur-Yvette, France
   firstname.lastname@centralesupelec.fr

**Abstract.** Modular deductive verification is a powerful technique capable to show that each function in a program satisfies its specified contract. However, function contracts do not allow users to express some high-level (e.g. security-related) properties of a software module and do not provide a global view on which high-level properties are actually established. To address this issue, this tool demo paper proposes a new specification mechanism, called meta-properties, able to express a rich set of high-level properties. A meta-property can be seen as an enhanced global invariant specified for all or a subset 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 then be proved by deductive verification tools in a usual way. This technique has been implemented as a Frama-C plugin called MetAcsl and successfully applied to specify and prove several safety- and security-related meta-properties in two illustrative case studies.

**Keywords:** meta-properties, deductive verification, formal specification, Frama-C.

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**, that 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 [3] and its deductive verification plugin Wp, which allows proving a 
C code 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 soft-
ware product frequently evolves during the software 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 verifica-
tion 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 specifica-
tion 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 hyper-
visors and micro-kernels [4].

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 confidentiality 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.

Contributions The contributions of this paper 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),

1 These examples are publicly available at [https://github.com/Firobe/metacsl_examples](https://github.com/Firobe/metacsl_examples)
– 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).

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 applied to our case study. They will be explained in more details as we go along.

```
/*@
meta M1: ∀function f; \strong_invariant(f),
∀ int page; 0 ≤ page < PAGE_NB ⇒
metadata[page].status == FREE ∨ metadata[page].status == ALLOCATED;
meta M2: ∀function f; //Only page_encrypt can change levels of allocated pages
! \subset (f, { page_encrypt}) ⇒ \writing(f),
∀ int page; 0 ≤ page < PAGE_NB ∧ metadata[page].status == ALLOCATED
⇒ \separated(\written, \metadata[page].level);
meta M3: ∀function f; \reading(f), //Ensures P_read
∀ int page; 0 ≤ page < PAGE_NB ∧ metadata[page].status == ALLOCATED
∧ user_level == PUBLIC ∧ metadata[page].level == CONFIDENTIAL
⇒ \separated(\read, metadata[page].data + (0 .. PAGE_LENGTH - 1));
*/ //Meta-property ensuring P_write is defined similarly to M3
```

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

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, \( F \subseteq \mathcal{F} \) 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 \( G \cup M(c) \),
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 $c$ (see Section 2.2).

The actual MetACL 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.

The MetACL syntax for the specification of $F$ uses the builtin ACSL constructions $\forall$, possibly followed by $\subset$ with or without logic negation $\bar{}$ (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 $F_- = \{\text{page_encrypt}\}$ exclude 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 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** It ensures that $P$ holds upon any modification of the memory (both stack and heap). It provides a meta-variable $\text{written}$ that refers to the variable (or, 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.

---

2 Temporal weakening for a specific block is also supported; it can be useful e.g. to weaken the property $A==B$ inside the code fragment $A=a;B=b;$. 

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, here we can forbid the modification under some condition (namely, that the page is allocated), while assigns has no such mechanism readily available.

### Reading Context

Similar to the writing context, it ensures that the property holds whenever some memory location is read, and provides a meta-variable $\text{\textbackslash read}$ referring to the read location. It is used in property $M_3$ (lines 17–20), which expresses the guarantee $P_{\text{read}}$ 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 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, that 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 $P$. 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).

```c
/*@ requires P_{M_1}; */
struct Page* page_alloc() {
    struct Page* fp = find_free_page();
    if (fp != NULL) {
        fp->status = ALLOCATED;
        /*@ assert P_{M_1}; */
        fp->level = user_level;
    }
}
```

(a) Transformation for $M_1$

```c
/*@ requires P_{M_1}; */
struct Page* page_alloc() {
    struct Page* fp = find_free_page();
    if (fp != NULL) {
        /*@ assert \forall int page; 0 ≤ page < PAGE_NB ⇒ metadata[page].status == ALLOCATED */
        fp->status = PAGE_ALLOCATED;
        /*@ assert \forall int page; 0 ≤ page < PAGE_NB ⇒ metadata[page].status == ALLOCATED */
        fp->level = user_level;
    }
}
```

(b) Transformation for $M_2$

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

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 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 $P_{\text{read}}$ and $P_{\text{write}}$) 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, as well as an automatic transformation-based technique to verify these properties by a usual deductive verification approach. 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 to a verification engineer 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, diminishing the risk of making invalid an implicit high-level property spanned over several contracts. 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 The work of the first author was partially funded by a Ph.D. grant of the French Ministry of Armed Forces.

References

[1] P. Baudin, P. Cuoq, J.-C. Filiâtre, C. Marché, B. Monate, Y. Moy, and V. Prevosto. ACSL: ANSI/ISO C Specification Language. \url{https://frama-c.com/acsl.html} 2009.

[2] C. A. R. Hoare. “An axiomatic basis for computer programming”. In: Commun. ACM 12.10 (1969).

[3] F. Kirchner, N. Kosmatov, V. Prevosto, J. Signoles, and B. Yakobowski. “Frama-C: A software analysis perspective”. In: FAOC (2015).

[4] G. Klein, J. Andronick, K. Elphinstone, T. Murray, T. Sewell, R. Kolanski, and G. Heiser. “Comprehensive Formal Verification of an OS Microkernel”. In: TOCS 32.1 (2014).