# High-Level Program Properties in Frama-C: Definition, Verification and Deduction

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

[1] Tarides, Paris, France
virgile.robles@protonmail.ch
[2] Thales Research & Technology, Palaiseau, France
nikolai.kosmatov@thalesgroup.com
[3] Université Paris-Saclay, CEA, List, Palaiseau, France
virgile.prevosto@cea.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.** In deductive verification of software, a contract is typically associated to each function and the implementation is shown to respect it. Such contracts for C programs can be expressed in the ACSL specification language. However, high-level (or global) program properties, in particular security properties, cannot be conveniently expressed as function contracts. This paper provides an overview of recent efforts to specify and verify global program properties in the Frama-C verification platform using a dedicated Frama-C plug-in called MetAcsl. Its verification approach relies on a translation of high-level properties into low-level ACSL annotations inserted in relevant program locations, followed by the verification of the resulting annotations. While this approach is expressive and powerful—and has already been effectively used in industrial applications—it can also be costly in terms of the number of the resulting low-level annotations. Deduction of high-level properties from other ones and from other annotations is thus desired. We discuss initial work on deduction of high-level properties and outline further research perspectives in this area.

**Keywords:** high-level properties · formal specification · deductive verification · Frama-C · MetAcsl.

## 1 Introduction

Deductive verification allows verification engineers to prove that a given program respects its formal specification. A formal specification is typically expressed in terms of *contracts*, where each function receives a *function contract* expressing the expected behavior of the function and including *preconditions* (expected to hold before the function call) and *postconditions* (expected to be ensured when the function exits). Recent years have seen many successful case studies

using deductive verification [10], in which a large range of safety, security and functional properties have been proved using various verification tools.

Among such tools for C programs, the Frama-C [13,14] framework offers a set of analyzers implementing a large range of verification techniques. In particular, the users of Frama-C can specify a given C program in its companion specification language ACSL [2] and prove that the program respects the provided specification using its deductive verification plug-in Wp.

In addition to function contracts and assertions that describe program properties in specific program points, the user may need to express and verify *high-level* (or *global*) *program properties* that concern the whole set (or a large subset) of program locations. Examples of such properties include strong and weak global invariants and security properties (such as integrity and confidentiality).

In our recent work [18,20,19,17], we introduced *High-Level ACSL Requirements* (HILAREs), also referred to as *meta-properties*[5], as a means to specify global properties, as well as a dedicated Frama-C plug-in MetAcsl to translate them into regular low-level ACSL annotations inserted in relevant program points. The proposed verification technique consists then in verifying the resulting annotations: if all the resulting annotations hold, the global property holds as well.

This approach appeared to be expressive and powerful, and was effectively used in industrial applications [6]. On large projects, though, it can lead to numerous low-level annotations, whose overall verification can become very costly. It turns out that in some cases this verification can be avoided by reasoning directly at high level to directly deduce a HILARE from other ones or from function contracts.

*Contributions.* This paper provides a panorama of the recent efforts [18,20,19] on specification and verification of global program properties in the Frama-C verification platform using MetAcsl. In addition, we present initial work on deduction of high-level properties and point out further research perspectives on this topic. The latter work is an original contribution of this paper: it was realized in the PhD thesis of the first author [17], but has not been published yet in another format.

*Outline.* The paper is organized as follows. The motivation for global properties is presented in Sect. 2. Next, we describe solutions for their definition using HILAREs and their verification using MetAcsl, resp., in Sects. 3 and 4. Deduction of HILAREs is presented in Sect. 5. Sect. 6 discusses some related work. Finally, Sect. 7 provides a conclusion and some future work directions.

[5] This term was used in initial publications on MetAcsl and is still used by some users, even if the term High-Level ACSL Requirements (HILAREs) is arguably more suitable.

## 2 High-Level Program Properties: Motivation

Some categories of properties over a C program, which we call *high-level properties* or *global properties*, are not easily expressible via function contracts. Such properties span across multiple functions and typically concern the whole set or a large subset of program locations. For instance, we may want to ensure a confidentiality property stating that all accesses to some data element throughout the program are always guarded by a proper authorization mechanism. Among other concrete examples of global properties, we can cite the following:

- a non-privileged user never reads a confidential data page;
- a privileged user never writes to a public data page;
- the confidentiality level of a page cannot be changed unless some condition holds;
- the privilege level of a user cannot be changed unless some condition holds;
- a free page cannot be read or written, and must contain zeros;
- a decryption function cannot be called unless some condition holds.

Incorporating such a global property into the contract of each relevant function can be tedious and error-prone, especially on large code bases. Moreover, even when it is technically possible to express a global property by multiple low-level annotations, the relevant annotations are typically distributed over several functions. Verification engineers and evaluators have in general no easy means to check[6]—other than by a tedious manual review of all annotations—that the set of provided contracts correctly and completely expresses the target global property. In the aforementioned example of a confidentiality property, checking that all accesses are indeed guarded by an appropriate annotation for a large project would quickly become very difficult. In this case, even when all annotations are proved, it can be hard to affirm with a high level of confidence that the high-level property is indeed true.

This verification can become even more cumbersome when the annotations related to the global property are mixed with the usual clauses: if the code or the specification of a function are updated, it is very easy to break the global property because it is not explicitly linked to the associated contract clauses.

In this context, HILAREs and the MetAcsl plug-in have been designed to address these issues, by letting verification engineers easily specify global properties on a C program and provide traceability between a HILARE and the corresponding low-level ACSL annotations that are verified by usual Frama-C plug-ins. The specification mechanism is presented in Sect. 3, while the verification approach is described in Sect. 4.

[6] Except in the simplest cases when a variable is never modified at all and this can be specified by `assigns` clauses, but this solution does not work in a more general case, e.g., if it can be modified only under a given condition, or to restrict read accesses.

## 3 Definition of High-Level Properties Using Meta-properties

A HILARE contains three components. First, we find its *target*, that is the set of functions to which it should apply. Second, we have to define the *context* of the HILARE, which indicates, inside the target functions, the exact program points where the HILARE needs to be instantiated. Finally, we must express the property itself, a normal ACSL predicate, which, depending on the context, can refer to some meta-variables that will be replaced with actual ACSL terms at each instantiation point.

Concretely, the MetAcsl plug-in uses Frama-C's ACSL extension mechanism [22, Sect. 4.16], and a HILARE is written under the following form.

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

In addition to the three main components mentioned above, each meta-property must have a name, which is crucial to ensure traceability between the meta-property and its instantiations.

Let us now describe more precisely the notions of target and context by expressing as proper HILAREs some of the examples of high-level properties mentioned in the previous section, as shown in Fig. 1. As it is often done, some ACSL notation (e.g. `\forall`, `&&`, `==>`, `<=`, `!=`) is pretty-printed (resp., as $\forall$, $\wedge$, $\Rightarrow$, $\leq$, $\neq$). First, property `confidential_read` (lines 15–22) specifies that a non-privileged user cannot read the content of a private page. More precisely, the HILARE applies to all functions, except `init`, that might need to perform specific tasks. The `\reading` context indicates that the HILARE will be instantiated each time a read access is performed. In such a context, the `\read` meta-variable denotes the address of the memory location that is read. Here, we state that such a location must be separated from the data of any allocated page whose level is strictly higher than the current user's level.

Our second example property, `constant_conf_level` (lines 24–30), exists to ensure that the confidentiality level of a page cannot be set outside of the three functions that are allowed to manipulate it. This time, we use the `\writing` context, so that the HILARE will be instantiated each time a write access occurs. For that, we use the `\written` meta-variable to state that the written location must be separated from the `status` field of any allocated page.

Finally, `encdec_uncalled` (lines 32–38) is using the `\calling` context and its `\called` meta-variable to forbid any call to the `encrypt` or `decrypt` function to occur outside of `page_encrypt` and `page_decrypt`.

Outside of the three contexts mentioned in these examples, MetAcsl introduces a few other ones, notably `weak_invariant` (the meta-property is instantiated at the entry and return point of each target function) and `strong_invariant` (the meta-property is instantiated at each instruction). A more complete description of the available contexts and their associated meta-variables can be found in Robles' PhD thesis [17] and in MetAcsl's manual [21].

```
1  struct Page { //Page handler structure
2    char* data; //First address of the page
3    enum allocation status; //ALLOCATED or FREE
4    enum confidentiality level; // Page level, PUBLIC or PRIVATE
5  };
6  enum confidentiality user_level; //Current user process level
7  struct Page pages[PAGE_NB]; //All pages
8  int init(); // Initialize pages
9  struct Page* page_alloc(void); //Allocates a page
10 void decrypt(char* data, unsigned key, size_t size);
11 void encrypt(char* data, unsigned key, size_t size);
12 void page_decrypt(struct Page*); //Decrypt a page in place, makes it PRIVATE
13 void page_encrypt(struct Page*); //Encrypts a page in place, makes it PUBLIC
14
15 /*@ //Never read from a higher confidentiality page
16  meta \prop, \name(confidential_read),
17    \targets(\diff(\ALL, init)), \context(\reading),
18    ∀ int p;
19      0 ≤ p < PAGE_NB ∧ pages[p].status == ALLOCATED ∧
20        user_level < pages[p].level
21     ⇒ \separated(\read, pages[p].data + (0 .. PAGE_SIZE - 1));
22 */
23
24 /*@ //Only page_encrypt/page_decrypt can manipulate a page's level field
25 meta \prop, \name(constant_conf_level),
26   \targets(\diff(\ALL, {init, page_encrypt, page_decrypt})),
27   \context(\writing),
28   ∀ int p; 0 ≤ p < PAGE_NB ∧ pages[p].status == ALLOCATED ⇒
29     \separated(\written,pages[p].level);
30 */
31
32 /*@ //The encryption/decryption primitives are only called within
33     // page_encrypt and page_decrypt
34 meta \prop, \name(encdec_uncalled),
35   \targets(\diff(\ALL, {page_encrypt, page_decrypt})),
36   \context(\calling),
37   !\subset(\called,{encrypt,decrypt});
38 */
```

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

## 4 Verification of High-Level Properties via Translation into Low-Level Annotations

When the meta-properties have been written by the user, MetAcsl has to instantiate them at the appropriate program points. For each HILARE, computing its set of instantiations is done in three steps, according to the definition seen in the previous section. First, the set of applicable functions is computed by considering the `\targets` of the HILARE. Second, for each target function, edges of the Control-Flow Graph (CFG) that are relevant for the `\context` of the HILARE are selected, and, where relevant, the meta-variables are bound to their actual value for the given edge. Finally, for each selected edge, an ACSL annotation is generated according to the property stated in the HILARE and the value of the meta-variables.[7]

[7] While MetAcsl's action is formalized on a CFG, the actual plugin inserts annotations in the original source code directly.

As an example, we can go back to the `confidential_read` property from Fig. 1 and see how it is instantiated in a function `page_read` that copies the content of a `Page` (if confidentiality conditions are met) into a buffer. Its definition is shown in Fig. 2, together with its CFG and the edges that are selected by the `\reading` context of `confidential_read`. More precisely, the `\reading` context

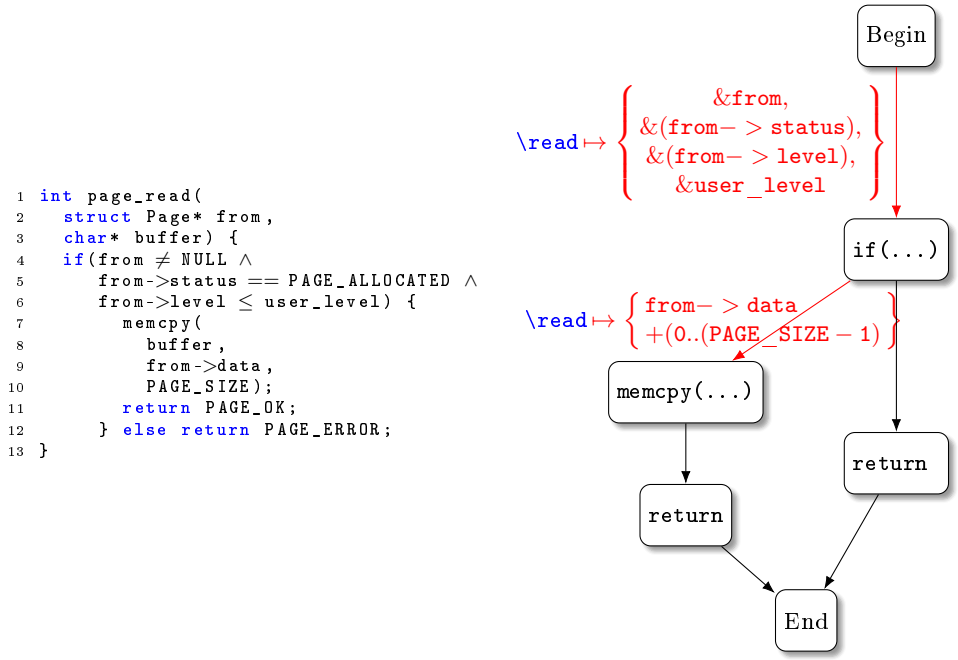


Fig. 2: Function `page_read` and its CFG, with red arrows indicating edges selected by the `\reading` context and the corresponding mapping of `\read` meta-variable to a set of pointers

selects all edges whose end node is an instruction that performs some read accesses. For each such edge, the `\read` meta-variable is then bound to the set of (addresses of) memory locations whose content may be read by the instruction. In our example, this is the case for two instructions. First, the condition of the `if` (lines 4–6) inspects the `from` pointer itself, as well as the `status` and `level` field (provided the pointer is valid), and `user_level`.

Second, we have the call to `memcpy` (lines 7–10). This second edge deserves some attention. Generally, edges leading to calls are only selected for the `\calling` context. However, `memcpy` is a bit special, in the sense that it is a function from the standard C library, and, as such, it has no implementation but only a prototype and an ACSL contract. In particular, its contract contains the following `assigns` clause.

```
1 /*@ ...
```

```
2   assigns ((char*)dest)[0..n - 1] \from ((char*)src)[0..n-1];
3   ... */
4 extern void *memcpy(void *restrict dest, const void *restrict src, size_t n);
```

This clause specifies that `memcpy` can write to any byte of the `dest` buffer and read any byte of the `src` buffer. In presence of a call to such a library function without implementation, MetAcsl considers, by default[8], that read accesses (and write accesses for the `\writing` context) are done by the caller itself. In other words, an HILARE with `\reading` or `\writing` context and targeting the caller will by default lead to a generated assertion upon such a call, based on an over-approximation of the read and written locations specified by the `assigns ... \from ...` clauses in the callee's contract. In our example, the binding to the `\read` meta-variable is computed by replacing the formal parameters of `memcpy` (`src` and `n`) by their actual expression (`from->data` and `PAGE_SIZE` respectively) in the `\from` term of the clause, leading to the mapping shown in Fig. 2.

Once the appropriate edges have been selected and the binding to the meta-variables have been computed, it just remains to generate an ACSL assertion at each selected edge and for each value associated by the binding, by replacing the meta-variables with their value in the predicate of the HILARE. Fig. 3 presents the instrumented code of the `page_read` function for the `confidential_read` property. As mentioned in Sect. 3, the generated assertions are named with `confidential_read:` and `meta:` for traceability. It can also be noted that, in the case of the `if` statement, MetAcsl generates four separated assertions instead of a single one since `\read` should be instantiated with each value of the set of four locations. This split gives annotations that are more easily verified by the Frama-C analyzers that are run on the instrumented code.

Namely, once the code has been instrumented, it is possible to use any of the standard analysis plug-ins (Eva, Wp, or E-ACSL) in order to try to validate the generated ACSL annotations.

## 5 Deduction of High-Level Properties: Initial Ideas and First Attempts

The verification technique for the HILARE language presented in the previous section is fundamentally *local*: it transforms a global requirement into a set of local properties in the source code, and verification activities are then done separately for each local property. The resulting set of properties is sometimes quite large despite efforts to discard trivially true or false statements early. While this approach is not always perfect, it cannot be completely avoided for proving global requirements about a code base without further global information: the low-level code *must* be observed in some way.

However, when some global requirements have *already* been established on a code base (for example by specifying and proving HILAREs in the usual way),

[8] This default behavior can be modified by available MetAcsl options.

```
1  int page_read(struct Page* from, char* buffer) {
2    /*@ assert confidential_read: meta:
3        ∀ int p;
4          0 ≤ p < PAGE_NB ∧ pages[p].status == ALLOCATED ∧
5            user_level < pages[p].level
6          ⇒ \separated(&from, pages[p].data + (0 .. PAGE_SIZE - 1));
7     */
8    /*@ assert confidential_read: meta:
9        ∀ int p;
10         0 ≤ p < PAGE_NB ∧ pages[p].status == ALLOCATED ∧
11           user_level < pages[p].level
12         ⇒ \separated(&from->status, pages[p].data + (0 .. PAGE_SIZE - 1));
13    */
14   /*@ assert confidential_read: meta:
15       ∀ int p;
16         0 ≤ p < PAGE_NB ∧ pages[p].status == ALLOCATED ∧
17           user_level < pages[p].level
18         ⇒ \separated(&from->level, pages[p].data + (0 .. PAGE_SIZE - 1));
19    */
20   /*@ assert confidential_read: meta:
21         ∀ int p;
22         0 ≤ p < PAGE_NB ∧ pages[p].status == ALLOCATED ∧
23           user_level < pages[p].level
24         ⇒ \separated(&user_level, pages[p].data + (0 .. PAGE_SIZE - 1));
25    */
26   if(from ≠ NULL ∧ from->status == PAGE_ALLOCATED ∧
27      from->level ≤ user_level) {
28       /*@ assert confidential_read: meta:
29           ∀ int p;
30            0 ≤ p < PAGE_NB ∧ pages[p].status == ALLOCATED ∧
31              user_level < pages[p].level
32            ⇒ \separated(
33                    from->data+(0 .. PAGE_SIZE -1),
34                    pages[p].data + (0 .. PAGE_SIZE - 1));
35          */
36       memcpy(buffer,from->data,PAGE_SIZE);
37       return PAGE_OK;
38     } else return PAGE_ERROR;
39 }
```

Fig. 3: Instrumented code for the `page_read` function and `confidential_read` property.

it feels natural to attempt *deducing* other global facts without having to resort to local analysis.

This section provides a motivating example, explains how and why a deduction framework can resolve the example in a sound way, and presents how such a deduction framework is implemented within the MetAcsl plug-in [21]. This section is an overview of the work described in Chapter 6 of the PhD thesis of the first author [17] that has not been published separately before.

A motivating example was encountered during verification efforts on the bootloader of a small embedded platform. This bootloader was structured as an automaton, with a function representing each state transition, itself calling potentially hundreds of other utility functions indirectly. Most of the actual state modifications were in the transition functions themselves rather than their callees though, and the effort was focused towards proving that the logic flow of

```
1  int state;
2
3  // A function which does not modify state,
4  // calling many other such functions
5  void util();
6
7
8  /*@
9      ensures state == 42;
10 */
11 void transition() {
12     // a new state is computed
13     ...
14     //@ assert KNOWN: state == 42;
15     util();
16     // calling utility functions does preserve the state value
17     //@ assert GOAL: state == 42;
18 }
```

Fig. 4: Motivating example for a deduction framework

these functions was correct. This can be represented in a simplified form by the code in Fig. 4.

Here, we have a transition function that should always end up with global variable `state` at value 42. Suppose that we have already proved that up until a final call to an external function `util`, the state value is as expected (assertion `KNOWN` on line 14). The only effort left to prove the postcondition is to prove that calling `util` conserves the value of `state` (illustrated by the `GOAL` assertion, which would imply the postcondition).

With a HILARE, it is easy to specify that function `util` and its callees do not write to variable `state` at all, with the `\writing` context:

```
/*@ meta \prop,
        \name(state_untouched_by_util),
        \targets(\callees(util)),
        \context(\writing),
    \separated(\written, &state); */
```

No matter the content of `util` and its callees, it can also be assumed that the proof of `state_untouched_by_util` is relatively easy given that `state` is indeed not modified in any of these functions. In particular, in general it is not necessary to write loop invariants to prove such a HILARE with this simple pattern.

However, specifying and proving that HILARE does not immediately help us prove the `GOAL` assertion in `transition` (line 13 in Fig. 4). While one can easily mentally deduce that if a function and all its callees do not modify a memory location then its value does not change after calling it, the HILARE does not provide the local information necessary to automatically prove the assertion with deductive verification.

The HILARE we would actually need to prove `GOAL` is the following, which adds `ensures state ==\old(state);` as a postcondition of `util` and each of its callees:

```
/*@ meta \prop,
```

```
2         \name(state_conserved_by_util),
3         \targets(\callees(util)),
4         \context(\postcond),
5     state == \old(state); */
```

However, in general, an automatic proof of these postconditions would fail. Their proof would require significant additional efforts. In particular, it would requiring providing loop annotations for each of the hundreds of callees, as well as `assigns` clause in their contracts.

This was unsatisfying[9] for a property that should seemingly be a natural consequence of the initial HILARE. What we would like is being able to *deduce* `state_conserved_by_util` from `state_untouched_by_util`, which is easy to prove in comparison, and then use it for further verification efforts (for example, to prove `GOAL`). In general we want to automatically apply a deduction pattern:

*Deduction pattern.* If a variable $v$ is not modified by a function $f$ nor any of its callees, then the value of $v$ is the same before and after any call of $f$.

Note that this pattern is related to `assigns` clauses of ACSL contracts, in the sense that if such clauses are provided, Wp would usually be able to prove that quite easily. However, providing these `assigns` clauses for all functions involved can be quite cumbersome, and is much stronger that what we want to achieve: we're only interested in ensuring that $v$ has not changed, not in determining the set of memory locations that might have changed.

We want to construct a framework where the end user can just add the line `\flags(proof:deduce)` to a HILARE in order to have MetAcsl try to automatically deduce it, rather than translating the HILARE to local assertions and delegating the task to other tools. To meet that objective of performance, the methodology we adopt is pragmatic: we want to be able to deduce simple properties automatically and efficiently, while having a formal guarantee that the deduction is sound. Furthermore, we want to design the framework in a way that it is extensible: it should be possible to add new, potentially more complex deduction patterns later. To address this goal, we proposed an approach that is threefold:

1. establish a mechanized formal model of the HILARE language using the Why3 [9] platform and prove that the deduction pattern described above is valid within that model;
2. translate this deduction pattern into an actionable and efficient Prolog deduction engine that can apply it;
3. extend MetAcsl to automatically translate a HILARE-specified program to constraints (also called knowledge base) for the Prolog engine, which can

[9] Note that for some more complex patterns there is no solution at all, for instance, for confidentiality: when a HILARE stating total absence of reading operations of some variable `Secret` in callees could be used to deduce a weaker HILARE stating the absence of unauthorised reading of `Secret` in the caller, that is, stating that reading `Secret` is allowed only under some condition.

then assess the validity of some properties using the previously established deduction patterns.

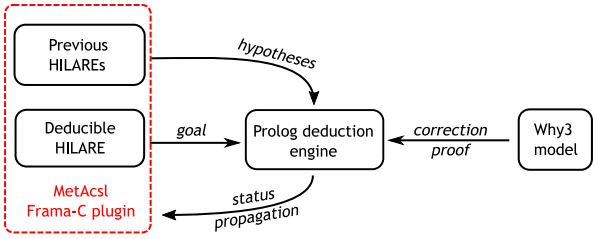


Fig. 5: Illustration of the deduction methodology

This approach is illustrated in Fig. 5. During the verification of a program annotated with HILAREs, one of which has the `deduce` flag, the following happens:

1. both the HILARE to be deduced and the other previous HILAREs are translated into data that is understood by the external deduction engine (as Prolog facts and rules, see paragraph *Translation of the program* below);
2. the Prolog deduction engine attempts to prove the desired HILARE based on the premise that the previous HILAREs are valid, and on a set of high-level deduction patterns (such as the ones presented previously). It is guaranteed to terminate but is not complete, as it handles only some specific HILARE structures and deduction patterns;
3. the deduction engine reports its result to MetAcsl, which propagates the status to the HILARE in Frama-C. If the translation to local assertions of that HILARE is enabled (with the `translation:yes` flag), then all the local assertions are considered valid if the deduction succeeded.

*Soundness.* The Why3 component of the framework has no impact at "runtime" (i.e., during the verification of HILAREs on concrete programs) but exists to formally ensure that Step 2 is sound. In that way, each part of the framework can be trusted. Currently, a weaker link (from the soundness point of view) is the interface between the Why3 model and the Prolog deduction engine, which is done manually: one must ensure that the deduction patterns encoded in the Prolog engine are exactly the ones proved within the Why3 model. A proof of the deduction approach can only be made within a formal system describing the HILARE language and ACSL themselves. This could have been done entirely in ACSL, however the Why3 platform and its WhyML language [9] are particularly appropriate since one can express a pure formal system and theorems easily, without having to deal with C specificities. The Coq proof assistant [24] was also a good candidate but the fact that Why3 can rely on external solvers to (try to)

automatically prove theorems was a clear advantage, and we did not need the full expressiveness of Coq.

The Why3 development is based on the formalization of HILAREs presented in [17, Chap. 3] and is itself described in details, with a few patterns other than the one mentioned above in [17, Chap. 6]. We model programs as blocks of instructions, where instructions are reduced to either *Read*, *Write* (to a specific location) or *Call*. We formalize the action of these instructions on the program state, the concept of assertion (a function that evaluates part of the state and returns a boolean), etc. Generally, the model keeps values completely abstract and just defines where state changes or does not, which is enough to prove our current patterns. Based on these primitive constructs, we are able to state our example pattern in Fig. 6.

```
lemma negative_assigns:
    forall prog: program, l: location, f: block, calls: S.fset block,
        pre post: state, v: value.                    (* IF *)
    valid_program prog ->                              (* all called functions are defined *)
    callees prog f calls ->                            (* 'calls' is the set of f callees *)
    untouched f l ->                                   (* f does not modify locally l *)
    (forall f'. S.mem f' calls -> untouched f' l) ->(* same for all callees, THEN *)
    state_block prog f pre post ->                     (* the value of l after and before f *)
    (M.mapsto l v pre <-> M.mapsto l v post)           (* is the same *)
```

Fig. 6: The example deduction pattern stated in Why3

This pattern is stated over any program: for the proof, we exhibited and used a conveniently structured induction principle on programs based on their call graph. Since our modelled call graphs are directed acyclic graphs (no recursion), we leverage the existing topological ordering of the graph's vertices as a total and well-founded order to deduce a Noetherian induction principle over functions of a call grap. It means that we can prove functional properties of the model by choosing a particular function $g$, and proving that the property holds on $g$ assuming that it already holds for every callee $h$ of $g$. Given that principle, small lemmas about what can be deduced within the model are incrementally built from the ground up, leading to the proof of the pattern.[10]

*Translation of the program.* Before high-level deduction can be used for a given program, a few properties about this particular program are automatically extracted and expressed as a set of Prolog facts, that can later be used by the Prolog engine for automatic deduction. In particular, we extract its structure, that is the set of its functions and the call graph. Furthermore, since the precondition for our deduction pattern is that the non-modification of the state by

[10] The model, its proofs and the corresponding Prolog rules are available in the MetAcsl distribution online, in folders `deduce` and `proofs`: `https://git.frama-c.com/pub/meta`

relevant functions is already proved through HILAREs, we must also represent any previously established HILARE as a fact in the knowledge base, so that it can be used by the deduction engine. This is performed by a simple pattern analysis: specified HILAREs matching known patterns are translated while others are dropped. Finally, the goal must also be translated, not as a fact but as a query. Going back to our motivating example, the program of Fig. 4 is translated into knowledge base of Fig. 7, where `meta_ground` and `meta_valid` are predicates of the Prolog engine described below.

```
% Export of the set of functions
targets({f_transition, f_util, f_util_callee_1, ...}).

% Export of the call graph
calls(f_transition, f_util).
calls(f_util, f_util_callee_1).
...
calls(f_util, f_util_callee_n).


% Translation of state_untouched_by_util
meta_ground("Writing", not_written(state), {f_util, f_util_callee_1, ...}).

% Translation of state_conserved_by_util. This is what we want to prove.
go :- meta_valid("Postcond", negative_assigns(state),
                 {f_util, f_util_callee_1, ...}).
```

Fig. 7: Knowledge base for the Prolog engine (automatically generated from Fig. 4)

*Prolog deduction engine.* While the Why3 model mentioned above exists to formally prove that deductions made with the patterns are sound, the Prolog deduction engine is here to actually use these patterns to make the deductions. The rationale for choosing Prolog to write the deduction engine is twofold:

- Prolog is a natural choice for the design and the usage of small CLP (Constraint Logic Programming) languages, such as {log} [8]. Hence, transposing the statement of the Why3 lemma is easy.
- If built correctly, an ad-hoc deduction engine written in Prolog is highly efficient. An early attempt was to directly use Why3 as a deduction engine, but the highly general nature of this platform made deduction on our large yet simple samples frustratingly inefficient. In particular, the formulas given to the engine usually involve reasoning over large sets (of function names), for which none of the automated provers used by Why3 as back-end were able to give satisfying results. On the contrary, {log} performed much better.

The solver is structured into three layers: the ground, intermediate, and high-level layers. The ground corresponds to what has been described above: listing known facts about the program and previously established properties. This part, by nature, is not hard-coded in the solver and must be generated at runtime.

Based on these known facts, there is a first layer in the engine that lay down very basic deduction *rules*, which we call the *intermediate layer*. It can essentially infer that a HILARE is valid if a stronger form of it is already established, or if it is simply the combination of multiple identical properties with different target sets.

On top of this simple layer, the engine defines rules for making less trivial high-level deductions based on the theorems established in the previous section. This layer is meant to be a transposition of the deduction patterns that matches the lemmas proved in Why3 as closely as possible. A small excerpt of this layer transposing the deduction pattern we want to use for our motivating example is provided in Figure 8.

```
% Try to deduce as a negative_assigns
meta_valid("Postcond", negative_assigns(L), S) :-
    % Functions do not modify L
    meta_inter("Writing", not_written(L), S) &
    % Callees are within S
    callees_restrict(S, S).
```

Fig. 8: Part of the high-level layer of deduction

The engine can then be queried to efficiently check if a particular HILARE can be deduced based on the known facts of the programs, or even list all deducible facts.

*Interaction with MetAcsl.* Given a program, a list of established HILAREs and a new HILARE marked with the `deduce` flag, MetAcsl can then build a knowledge base and query the Prolog engine. On success, it sets the status of the new HILARE (and potentially of its generated ACSL annotations) to valid, helping to prove further properties about the program. For example, this would generate an already proved assertion in our motivating program from which Frama-C/Wp-C could immediately derive our goal assertion.

*Extensibility.* While this section presents a relatively narrow deduction pattern for simplicity, other more complex ones were added following the same process (proof of correctness within Why3, and transposition to Prolog). Other examples of such deduction patterns include the following: if a function `f` is only ever called by functions for which a predicate `P` is a precondition and that do not modify the variables used by `P`, then `P` is also a precondition of `f`. This pattern, along with a third one, was used successfully for the verification efforts mentioned at the beginning of this section (see [17, Chaps. 6 and 7]). More generally, adding new patterns is non-trivial: the Why3 formalization and Prolog solver may need to be extended with new constructs for the introduced patterns to make sense. However, we believe that the more patterns are added to the framework, the easier it will be to add new ones since the Prolog and Why3 models will already

be complete enough to simply focus on the use case. This would also reduce the knowledge an end-user needs to have about the overall deduction engine, whereas today a HILARE marked for deduction has little chance to be proved successfully if it does not precisely match one of the existing deduction patterns.

## 6 Related Work

HILAREs partially overlap with previous extensions of contract-based specification languages such as JML [15], a behavioural specification language for Java programs. JML has been extended in [5] to specify protocols and in [25] to express temporal properties. Although protocols and a subset of temporal properties can be expressed with HILAREs, our syntax is not as easy to use as theirs. On the other hand, those extensions cannot be used to easily specify arbitrary high-level requirements on programs. The general idea of defining a high-level concept in the global scope and then *weaving it* into the program has been partially explored before. For instance, [16] enables enforcing high-level security properties by performing a code transformation weaved throughout the implementation. But the properties considered are very specific to the JavaCard programming language, both in terms of specification and verification: the code transformation is based on the assumption that there are a set of *core* functions acting as the main interfaces to the smart card. While the verification mechanism in [16] is quite similar to our approach, it lacks generality at the specification level. Similarly, ACSL [2] has been extended in different ways in the past, targeting different requirements than this work. To cite only a few, [23] introduced Aoraï to allow the specification and verification of complex temporal properties partially supported by our approach while [3] explored the specification and verification of relational properties, i.e. properties linking several program calls. Trace contract were recently studied in [11,4].

The notion of cross-cutting global properties is analogous to the Aspect-Oriented Programming (AOP) [12] paradigm. This paradigm allows writing code (called *concerns*) at the global level while specifying a set of control flow points (called *pointcuts*) where the code needed by the concern should be inserted. For example, this may be useful for easily adding logging to an already existing implementation. Our approach can be seen as writing cross-cutting concerns at specification level rather than code level. Indeed, contexts can be related to AOP's pointcuts: they are a criterion for specifying a set of control flow points and performing a local specification action. Several other works explore what they identify as Aspect-Oriented Specification, in different ways than ours: [27] explores how to specify programs written in AOP with a notion of invariant, and proposes a translation of that specification to JML, while [1] introduces another way to reason about AOP programs. Compared to our work, these efforts focus only on reasoning about programs that are already written in an AOP style, i.e., where some pervasive programming logic is *already* centralized at a single point, whereas we want to tackle arbitrary programs and only centralize at the *specification level*.

Closer to our work, [26] proposes another specification approach that can be translated to JML, allowing the specification of generalized weak invariants over arbitrary classes and methods, by specifying so-called *assertion aspects* inspired by AOP concepts, which translate to local annotations. While our work has the same inspiration (trying to centralize similar assertions spanning many functions), this approach lacks the expressiveness we are aiming for, especially in terms of memory management.

## 7 Conclusion

This paper has presented an overview of recent efforts for specification and verification of high-level properties with Frama-C. It describes a specification mechanism in the form of High-Level ACSL Requirements (HILARE), also called meta-properties. It also presents a transformation based approach translating HILAREs into regular ACSL annotations for their verification, which enables applying available verification tools to the resulting annotations. Finally, we presented initial efforts on deduction of HILAREs from other ones and from ACSL annotations.

*Industrial Application.* The HILARE based approach and the MetAcsl tool have been successfully used in a large-scale industrial verification project [6,7]. It was performed for a security-critical smart card product in order to perform its rigorous Common Criteria based certification at the highest assurance levels EAL6–EAL7[11]. Security of a smart card strongly relies on the requirement that the underlying JavaCard virtual machine ensures necessary isolation properties.

In that project, formal verification of a JavaCard Virtual Machine implementation was performed by Thales using the Frama-C verification toolset. It strongly relies on the MetAcsl tool. The target security properties expressed as HILAREs include integrity and confidentiality. The target implementation contains over 7 000 lines of C code. In particular, based on only 36 HILAREs manually specified by verification engineers, more than 400 000 lines of ACSL annotations were automatically generated by MetAcsl (leading to over 27 400 proof goals) and formally proved by Wp. This industrial verification project illustrates the potential of industrial applications of the approach described in this work.

*Future Work.* There are several interesting research avenues that build upon the work presented in this paper.

Regarding the specification language of HILAREs, while they enable the specification of high-level requirements, they are still quite close to the code: the user must precisely qualify the set of targets, the kind of operations targeted by the context and correctly refer to the global state. This allows specifying

[11] The EAL7 certificate delivered by the French certification body ANSSI is available at `https://cyber.gouv.fr/sites/default/files/document_type/Certificat-CC-2023_45fr_0.pdf`.

properties in the global scope but can sometimes seem verbose, and hard to proofread for a domain expert with little knowledge of the code base.

This can be alleviated with macros: C macros can abstract away lists of targets, HILARE patterns for frequent kinds of properties, and even parts of predicates in order to replace code-level expressions by higher-level names. However, this is a limited approach as it relies on the C preprocessor, which only manipulates text instead of semantic tokens and is notoriously hard to work with.

A good way of improving this approach could be to develop an abstraction system of macros (or, possibly, dedicated specification artifacts): there could be abstract sets of functions, predicates, HILARE patterns that are statically defined and type-checked instead of simply replacing text. One could then imagine that the specification of an application should only be written using this higher-level library, only punctually adding new domain-specific ones, so that the specification remains legible for all. This could also enable the generation of specification from other sources such as abstract models from other applications, where the "glue" between the higher-level specification and the concrete code is materialized by a set of macros defined afterwards.

The deduction framework presented in Sect. 5 is still experimental and has several limitations. The automatic translation from MetAcsl to Prolog handles only simple cases and has to be extended. There are only three simple deduction patterns currently implemented in the solver and proved in Why3: there is a lot of room for extending the scope of the deduction. One obvious extension is the addition of more deduction patterns to cover more situations where deduction would be desirable. More generally, the fact that the framework currently only deals with deduction of specific ad-hoc patterns of HILAREs is a limitation. It would be desirable for the framework to have a more general purpose. A full formalization and proof of correctness is another research direction.

Finally, further applications of the proposed specification and verification approach will allow to better understand its benefits and limitations.

*Acknowledgment.* Part of this work was supported by ANR (grants ANR-22-CE39-0014, ANR-22-CE25-0018). We thank Louis Rilling for his valuable contribution to some parts of this work, as well as the anonymous referees for helpful comments.

## References

1. Bagherzadeh, M., Rajan, H., Leavens, G.T., Mooney, S.: Translucid contracts: Expressive specification and modular verification for aspect-oriented interfaces. In: Proceedings of the Tenth International Conference on Aspect-Oriented Software Development. p. 141–152. AOSD '11, Association for Computing Machinery (2011). https://doi.org/10.1145/1960275.1960293
2. Baudin, P., Filliâtre, J.C., Marché, C., Monate, B., Moy, Y., Prevosto, V.: ACSL: ANSI/ISO C Specification Language, http://frama-c.com/acsl.html

3. Blatter, L., Kosmatov, N., Le Gall, P., Prevosto, V., Petiot, G.: Static and dynamic verification of relational properties on self-composed c code. In: Proc. of the 12th International Conference on Tests and Proofs (TAP 2018), Held as Part of STAF 2018. LNCS, vol. 10889, pp. 44–62. Springer (Jun 2018). https://doi.org/10.1007/978-3-319-92994-1_3
4. Bubel, R., Gurov, D., Hähnle, R., Scaletta, M.: Trace-based deductive verification. In: 24th International Conference on Logic for Programming, Artificial Intelligence and Reasoning. vol. 94, pp. 73–95. EasyChair (2023). https://doi.org/10.29007/vdfd
5. Cheon, Y., Perumandla, A.: Specifying and checking method call sequences in JML. In: International Conference on Software Engineering Research and Practice (SERP 2005). pp. 511–516. CSREA Press (2005)
6. Djoudi, A., Hána, M., Kosmatov, N.: Formal verification of a JavaCard virtual machine with Frama-C. In: Proc. of the 24th International Symposium on Formal Methods (FM 2021). LNCS, vol. 13047, pp. 427–444. Springer (2021). https://doi.org/10.1007/978-3-030-90870-6_23, long version available at https://nikolai-kosmatov.eu/publications/djoudi_hk_fm_2021.pdf
7. Djoudi, A., Hána, M., Kosmatov, N., Kříženecký, M., Ohayon, F., Mouy, P., Fontaine, A., Féliot, D.: A bottom-up formal verification approach for common criteria certification: Application to JavaCard virtual machine. In: Proc. of the 11th European Congress on Embedded Real-Time Systems (ERTS 2022) (2022)
8. Dovier, A., Omodeo, E.G., Pontelli, E., Rossi, G.: {log}: A language for programming in logic with finite sets. The Journal of Logic Programming **28**(1), 1–44 (1996). https://doi.org/10.1016/0743-1066(95)00147-6
9. Filliâtre, J.C., Paskevich, A.: Why3 — Where Programs Meet Provers. In: European Symp. on Programming (ESOP) (2013). https://doi.org/10.1007/978-3-642-37036-6_8
10. Hähnle, R., Huisman, M.: Deductive Software Verification: From Pen-and-Paper Proofs to Industrial Tools, LNCS, vol. 10000, pp. 345–373. Springer (2019). https://doi.org/10.1007/978-3-319-91908-9_18
11. Hähnle, R., Scaletta, M., Kamburjan, E.: Herding cats. In: Software Engineering and Formal Methods. pp. 3–8. Springer (2023). https://doi.org/10.1007/978-3-031-47115-5_1
12. Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C.V., Loingtier, J.M., Irwin, J.: Aspect-oriented programming. In: European Conference on Object-Oriented Programming. LNCS, vol. 1241, pp. 220–242. Springer (1997). https://doi.org/10.1007/BFb0053381
13. Kirchner, F., Kosmatov, N., Prevosto, V., Signoles, J., Yakobowski, B.: Frama-C: A Software Analysis Perspective. Formal Aspects of Computing (2015). https://doi.org/10.1007/s00165-014-0326-7
14. Kosmatov, N., Prevosto, V., Signoles, J. (eds.): Guide to Software Verification with Frama-C. Core Components, Usages, and Applications. Computer Science Foundations and Applied Logic Book Series, Springer (2024). https://doi.org/10.1007/978-3-031-55608-1
15. Leavens, G.T., Baker, A.L., Ruby, C.: JML: A notation for detailed design. In: Behavioral Specifications of Businesses and Systems, vol. 523, pp. 175–188. Springer (1999). https://doi.org/10.1007/978-1-4615-5229-1_12
16. Pavlova, M., Barthe, G., Burdy, L., Huisman, M., Lanet, J.L.: Enforcing high-level security properties for applets. In: International Conference on Smart Card Research and Advanced Applications. pp. 1–16. Springer (2004). https://doi.org/10.1007/1-4020-8147-2_1

17. Robles, V.: Specifying and Verifying High-Level Requirements on Large Programs: Application to Security of C Programs. Ph.D. thesis, Univ. Paris-Saclay (2022), https://theses.hal.science/tel-03626084/
18. Robles, V., Kosmatov, N., Prevosto, V., Rilling, L., Le Gall, P.: MetAcsl: Specification and verification of high-level properties. In: Proc. of the 25th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2019), Held as Part of the European Joint Conferences on Theory and Practice of Software (ETAPS 2019). LNCS, vol. 11427, pp. 358–364. Springer (Apr 2019). https://doi.org/10.1007/978-3-030-17462-0_22
19. Robles, V., Kosmatov, N., Prevosto, V., Rilling, L., Le Gall, P.: Methodology for specification and verification of high-level properties with MetAcsl. In: Proc. of the 9th IEEE/ACM International Conference on Formal Methods in Software Engineering (FormaliSE 2021). pp. 54–67. IEEE (2021). https://doi.org/10.1109/FormaliSE52586.2021.00012
20. Robles, V., Kosmatov, N., Prevosto, V., Rilling, L., Le Gall, P.: Tame your annotations with MetAcsl: Specifying, testing and proving high-level properties. In: Proc. of the Int. Conf. on Tests and Proofs (TAP). LNCS, vol. 11823, pp. 167–185. Springer (2019). https://doi.org/10.1007/978-3-030-31157-5_11
21. Robles, V., et al.: MetAcsl Frama-C plug-in, https://git.frama-c.com/pub/meta
22. Signoles, J., Antignac, T., Correnson, L., Lemerre, M., Prevosto, V.: Plug-in Development Guide For Frama-C 28.1 (Nickel). Tech. rep., CEA, List (2024), https://frama-c.com/download/frama-c-plugin-development-guide.pdf
23. Stouls, N., Prevosto, V.: Aoraï plug-in tutorial (2023), https://frama-c.com/download/frama-c-aorai-manual.pdf
24. The Coq Development Team: The Coq Proof Assistant Reference Manual Version 8.19 (2024), https://coq.inria.fr/doc/V8.19.0/refman/
25. Trentelman, K., Huisman, M.: Extending JML specifications with temporal logic. In: International Conference on Algebraic Methodology and Software Technology. LNCS, vol. 2422, pp. 334–348. Springer (2002). https://doi.org/10.1007/3-540-45719-4_23
26. Yamada, K., Watanabe, T.: An aspect-oriented approach to modular behavioral specification. Electronic Notes in Theoretical Computer Science **163**(1), 45–56 (2006). https://doi.org/https://doi.org/10.1016/j.entcs.2006.07.002, proceedings of the First Workshop on Aspect-Based and Model-Based Separation of Concerns in Software Systems (ABMB 2005)
27. Zhao, J., Rinard, M.: Pipa: A behavioral interface specification language for aspect. In: Pezzè, M. (ed.) Fundamental Approaches to Software Engineering. LNCS, vol. 2621, pp. 150–165. Springer (2003). https://doi.org/10.1007/3-540-36578-8_11