Formal Verification for Security Certification: From a First Success to Sustainable Industrial Usage

Adel Djoudi[1][0000−0002−8238−6490] and Nikolai Kosmatov[2][0000−0003−1557−2813]

[1]Thales Cybersecurity and Digital Identity, Meudon, France
[2]Thales Research and Technology, cortAIx Labs, Palaiseau, France
firstname.lastname@thalesgroup.com

Abstract. The application of formal methods in an industrial context remains a delicate and time-consuming task, and each successful project contributes to improving productivity. This experience report summarizes five years of applying formal verification for Common Criteria-based security certification of a JavaCard Virtual Machine at Thales. Since the first successful formal verification—directly over the source code with the Frama-C verification platform—was achieved in 2021, five highest-level Common Criteria certifications (at EAL6 and EAL7 levels) have been obtained during the past five years. The strong guarantees of functional correctness and security provided by source code-based verification are essential for security-critical products such as smart cards. This paper traces the evolution of the project from an initial success to the continuous application of formal verification, and discusses the main challenges, applied solutions, and opportunities for further improvement.

Keywords: deductive verification, security properties, JavaCard virtual machine, Frama-C, Common Criteria certification, smart cards.

1 Introduction

Many aspects of modern society have become strongly dependent on software-based devices and services. Examples of such critical products include smart cards, which are widely used today for identity documents, banking, and communication. In the presence of increasing cybersecurity risks, it is crucial to guarantee functional correctness and security properties of critical products and services. Formal verification can provide such guarantees.

Although the application of formal methods in industry has become more widespread, industrial users still face difficulties when applying them to large-scale projects while maintaining long-term productivity. Sharing achievements and challenges from these applications within the formal methods community is valuable to both researchers and practitioners. This paper is motivated by this goal and reports on the experience with the formal verification of a critical smart card software at Thales.

In 2021, Thales realized a world-first formal verification [22] of a JavaCard Virtual Machine (JCVM) *directly on the real-life code* using the Frama-C verification platform [31]. It was done for security certification of a smart card product according to the Common Criteria (CC) for Information Technology Security Evaluation [16], which provide an international standard for security certification. It includes seven levels of assurance evalution (EAL1–EAL7). The highest levels (EAL6–EAL7) require a formal Security Policy Model (SPM) along with a rigorous mathematical proof of functional and security properties (including confidentiality and integrity). Overall, since the initial success in 2021 [22], five highest-level certifications of the JavaCard Virtual Machine have been realized by Thales during the past five years [1–5]. For the latest certification [5], Thales incorporated post-quantum cryptography, thus featuring the first quantum-resistant smart card in Europe to receive high-level security certification [42]. The goal is to help government services protect sensitive data—such as those on ID cards, health cards, and driver's licenses—and ensure that citizens' identities remain secure even in the presence of emerging quantum threats.

Contributions. This paper presents an experience report on five years of applying deductive verification for security certification of smart card software at Thales. It traces the evolution of the project from the initial success towards sustained, long-term industrial use. We outline the main difficulties we encountered and the solutions we adopted to improve the overall productivity of formal verification in this project. These solutions encompass both methodological advances and tool enhancements. Our key contributions include the specification of two new Frama-C features—recently implemented by the Frama-C team—namely, the Uncast plugin, which automates code transformations involving pointer casts, and the GenScript solution, which automates proof script generation. In addition, we present a novel open-source Frama-C plugin, Frama-C-lsp[1], which we developed in order to provide a convenient IDE for annotated C code based on VS Code. Finally, we discuss remaining challenges and outline opportunities for further improvements.

Outline. Section 2 presents the context and motivation of this work. Section 3 presents the aspects of the project related to formal specification and organization of source files. A novel VS Code extension for C and ACSL based on the Frama-C-lsp plugin is introduced in Sect. 4. Industrial expectations for productive applications of formal verification are emphasized in Sect. 5. Section 6 provides statistics and outlines challenges related to proof performance. Finally, Sect. 7 presents a conclusion and future work.

[1] Available at https://github.com/ThalesGroup/frama-c-lsp

2 Context and Motivation

2.1 The Frama-C Verification Platform

Frama-C [31, 32] is a verification platform for ISO C99 programs [29] developed by CEA List with contribution of Inria. It offers several analysis tools organized as plugins, working on top of a common kernel. Our project mainly relies on the WP plugin of Frama-C dedicated to modular deductive verification of C code [10].

Frama-C users can specify program annotations in ACSL (ANSI C Specification Language) [8,12], a formal specification language for C programs. Typically, for deductive verification of a C program in Frama-C, each function should be annotated with a *(function) contract* expressed in ACSL. In the contract of a function f, a *precondition* (expressed with an ACSL clause **`requires`**) defines a property expected to hold before any call to f. It is assumed on entry during the verification of f, but must be proved before any call to f during the verification of a caller. Dually, a *postcondition* (**`ensures`** clause) of f states a property that must hold after the function call: it must be proved during the verification of f but is assumed after the call to f in a caller. A frame rule (**`assigns`** clause) in the contract of f specifies the memory locations that f is allowed to modify. In a loop contract, a *loop invariant* (**`loop invariant`** clause) specifies a property that must hold before the loop and after every loop iteration. Lemmas and local assertions can be used to facilitate the proof by stating a useful intermediate property. A *(local) assertion* (expressed with an ACSL clause **`assert`**) must hold at the point where it is inserted. Stated at the global level, *lemmas* are proved once and can then be used in different proof contexts. ACSL offers the `integer` type for unbounded mathematical integers in addition to bounded machine types.

Given a C program with a (partial) formal specification expressed by ACSL annotations, the WP plugin [10] tries to prove that the program respects the provided annotations. To do that, WP generates *verification conditions* (or *proof goals*, or *proof obligations*) that are then either proved by WP itself (thanks to its internal formula simplification engine called Qed [18]) or sent via the Why3 tool [27] to external provers (SMT solvers). In our project, we use the Alt-Ergo solver [17]. If all proof goals are proved, the given program is guaranteed to respect the given ACSL specification. When automatic proof fails, the interactive proof editor of WP can be used to manually record a *proof script* by applying predefined proof tactics. Frequently used tactics include: `unfold` to unfold a predicate definition; `bittestrange` and `modmask` to rewrite a bit-level operation, resp., using ranges or modulo operations; `range` to enumerate a finite range of values; `split` and `cut` to split a goal into several subgoals using, resp., either logical operations and conditionals, or an intermediate hypothesis; `overflow` to consider in-range and out-range cases of machine integer conversion; `filter` to erase irrelevant hypotheses in a proof context; `instance` to instantiate a quantified variable to a specific value, etc. A detailed description of tactics can be found in [7]. After the application of a few carefully chosen tactics, SMT solvers can manage to prove the transformed proof goals.

```
1 /*@ requires n ≥ 0 ∧ \valid(t + (0..n-1));
2     assigns nothing;
3     ensures result ≠ 0 ⇔
4         (∀ ℤ j; 0 ≤ j < n ⇒ t[j] == 0);
5 */
6 int all_zeros(int *t, int n) {
7     int k=0;
8     /*@ loop invariant 0 ≤ k ≤ n;
9         loop invariant ∀ ℤ j; 0 ≤ j < k ⇒ t[j] == 0;
10        loop assigns k;
11        loop variant n-k;
12    */
13    while(k < n){
14        if (t[k] ≠ 0)
15            return 0;
16        k++;
17    }
18    return 1;
19 }
```

Fig. 1: Function `all_zeros` with ACSL specification.

To check whether the provided specification contains inconsistencies (e.g. contradictory assumptions), the user can activate their detection by the WP plugin using so-called *smoke tests* [11]. Basically, WP introduces a special check-and-forget assertion of false (that can be expressed in ACSL as **`check`** `\false;`) and tries to prove it with a short timeout. A successful proof indicates an inconsistency. In addition, to ensure that the program under verification cannot provoke *undefined behaviors* (also called *runtime errors*), WP relies on the RTE plugin of Frama-C to generate additional assertions to exclude the risk of undefined behaviors. Their proof is an essential step of formal verification, both to ensure the soundness of verification and to avoid security vulnerabilities due to undefined behaviors.

Finally, the MetAcsl plugin [40, 41] of Frama-C can be used to express and verify security properties. It allows the user to specify *global properties* (also called *metaproperties*, or *High-Level Requirements*) and translates them into local assertions (for instance, for each reading or writing operation) that can then be verified by other tools (like WP).

2.2 Example of ACSL Specification

Consider the function `all_zeros`, presented in Fig. 1. It receives an array t and its size n, and returns a nonzero value if all elements of t are zero, and 0 otherwise. We use pretty-printing of some ACSL keywords and symbols, for instance, `integer`, `==>` and `>=` as $\mathbb{Z}$, $\Rightarrow$ and $\geq$.

The contract of the function includes a precondition on line 1, which requires that that n is non-negative and the array contains valid memory locations at indices $0..(n-1)$. These constraints must be satisfied by the caller. The assigns clause on line 2 enforces that no non-local variables are modified, ensuring for instance that implementations altering the array would be deemed incorrect. The

postcondition on lines 3–4 specifies that the function's return value is nonzero if and only if all array elements are equal to zero.

In the presence of a loop, the loop contract on lines 8–12 is necessary for a successful deductive verification. The loop invariant on line 8 gives an interval of values for k. The loop invariant on line 9 specifies that elements at indices $0..(k-1)$ are zero after each complete iteration (non-interrupted with a return). The loop assigns clause identifies variables that may change within the loop, and the loop variant $(n-k)$ establishes a decreasing positive value providing an upper bound for the number of remaining loop iterations and helping the tool to prove termination of the loop. With this loop contract, Frama-C/WP can verify the correctness of the function, including safe array access on line 14 and the absence of arithmetic overflow on line 16.

2.3 Formal Verification of a JCVM at Thales

JavaCard Virtual Machine (JCVM) [38] is a crucial component present on most current smart cards. Its functional role is to interpret *JavaCard bytecode*, that is, a sequence of *opcodes*, each one specifying one operation of a JavaCard program. Its security role is to ensure mutual *applet isolation* [37]: the applets of mutually non-trusted vendors are not allowed to read or modify each other's data.

In 2021, Thales realized formal verification [22] of a JCVM for Common Criteria certifications of its smart card products. For the first time, the verification was realized directly on the source code, which brings many benefits to the certification process in terms of model understanding and confidence in verified properties [23]. It reduces the gap between the proven properties and the code.

This initial proof relied on several carefully chosen solutions detailed in previous publications [22, 24]: *ghost code* to model memory features, to avoid existential quantifiers and to facilitate bit-level reasoning; *code transformations* to address the limitations of Frama-C/WP; *proof scripts* for goals for which automatic proof fails; *preservation predicates* to facilitate reasoning about memory updates; *metaproperties* to express and verify security properties. This initial proof took approximately 3 person-years. We refer the reader to [22] for more detail and technical aspects.

2.4 Motivation: Productivity of Formal Verification in Industry

While an initial proof of a new project is a key milestone—one for which an industrial company may be willing to invest a significant amount of human effort—it does not ensure the sustained, long-term use of formal verification throughout the project. We observe that sustainable application of formal verification requires a higher level of automation across the entire process (including specification writing and comprehension, proof replay, reporting), as well as improved support for maintaining verified code (including the creation and evolution of proof scripts).

Indeed, industrial users must cope with a variety of evolving factors: the evolution of the codebase (e.g., to integrate new features or updates), the evolution of the formal specification (e.g., to express additional properties or to extend the

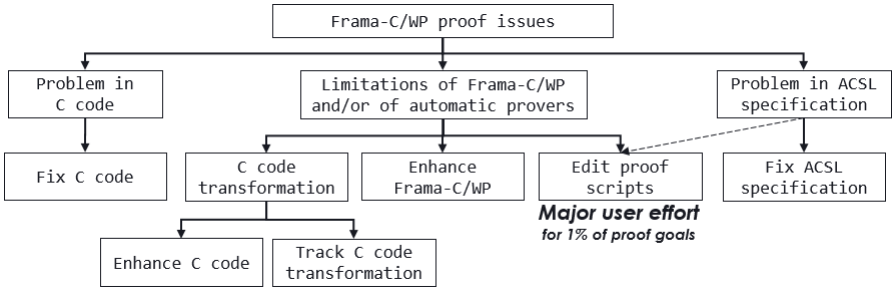

Fig. 2: Handling proof issues during deductive verification with Frama-C/WP.

verification scope), the evolution of verification tools (which may break manually written proof scripts or require new ones), and the evolving expectations of evaluators and certification authorities. Each of these changes can trigger substantial proof rework. Rather than performing a one-shot proof of a single project with a fixed set of properties and tools, industrial verification engineers must verify many similar, evolving projects using evolving tools, while carefully tracking all proof-related artifacts (annotations, verification scope, proof scripts, documentation, etc.). Given the size of industrial codebases and the frequency of such evolutions, it is hardly possible to reinvest the same level of effort repeatedly throughout the project lifecycle. Improving the overall productivity of formal verification in industrial contexts is therefore a key requirement for its sustained, long-term application. This concern has been a constant focus of our project over the years.

3 Structured Formal Specification

In this section, we first provide an overview of the main challenges of deductive verification on large-scale industrial projects. We then focus on several challenges and solutions related to the formal specification of the C code and the corresponding properties. Some issues are discussed further in the following sections.

3.1 Deductive Verification Workflow

The theoretical workflow of deductive verification appears straightforward: specify the C code in ACSL and prove it with Frama-C/WP. In practice—especially for large industrial projects involving complex properties—numerous iterations are necessary to reach a meaningful formal specification and address potential proof issues. Such issues include *proof failures* and *warnings* reported by WP, each of which requires careful manual analysis. Figure 2 illustrates the typical workflow of addressing proof issues.

A proof failure may reveal an error in the code or in the ACSL specification (cf. the left and right branches in Fig. 2), both of which must be corrected. *The*

ACSL specification needs to be complete, precise and free of any inconsistency. Achieving this is a major challenge in deductive verification. A careful organization and preparation of the annotated code is key to achieve this goal on large projects. A convenient IDE for browsing and understanding the annotated code (further discussed in Sect. 4) is necessary in practice. Some specification issues can be identified by manual review of ACSL annotations (in particular, completeness or precision issues) or by failing smoke tests (in particular, specification inconsistencies, cf. Sec. 2.1).

Due to proof incompleteness [39], even with a correct and complete ACSL specification, *automatic provers may fail to discharge complex proof goals.* This constitutes a second major challenge. In such cases, manually written proof scripts can be used to prove the remaining goals (cf. Sec. 2.1). However, as discussed in Sect. 2.4, in an evolving industrial context, this activity is highly time-consuming and costly in terms of proof maintenance, even when only a small fraction of goals requires proof scripts (1–3% in our project). Proof scripts are futher discussed in Sect. 5.

Last but not least, known limitations of Frama-C or WP in handling certain C and ACSL constructs as well as limitations of the memory model of WP when reasoning about pointer casts, restrict *the set of instructions and properties that can be formally verified.* In such cases, we rewrite the code in an equivalent way to make it suitable for formal verification. These *code transformations* should be introduced in a structural way and carefully tracked until they are either integrated into the production code or rendered obsolete by tool enhancements.

We also regularly provide feedback to the Frama-C developers in order to enhance the tool and extend the set of supported constructs.

3.2 Redundancy of the ACSL Specification: Good and Bad Practices

Repeating the same ACSL clauses in different function contracts is often necessary to specify required properties, which can lead to a certain *redundancy of contracts* and become a source of errors. As our goal is to verify functional and security properties, we express some of them using ACSL predicates. Instances of these predicates must then be included in several function contracts. To avoid redundant clauses in function contracts, we define *parametrized macros* that encapsulate commonly used predicates, and we include these macros in the relevant contracts. The macros are automatically unfolded to the required clauses by the preprocessor.

Another important solution for avoiding redundant clauses and automating the generation of necessary assertions relies on the MetAcsl plugin. We specify many important security properties as globally defined *metaproperties* [40, 41], which state confidentiality and integrity properties as constraints on all reading and writing operations. Such metaproperties are then automatically instantiated by the MetAcsl plugin into local checks [40,41] before the corresponding read and write operations.

A common approach to avoid redundant intermediate properties relies on the use of lemmas. Interestingly, *an intensive use of lemmas proved to be inefficient*

in our project. We compared a version that makes extensive use of lemmas (61 lemmas) to express intermediate properties with another version that uses only three lemmas and instead relies on 519 local assertions to express such properties where needed. Replacing lemmas with assertions reduced the overall proof time by 1 h 20 min. This result indicates that lemmas can significantly slow down the proof by introducing additional (and often useless) reasoning paths for the solver. Consequently, in our project we favor local assertions (repeated when needed) to simplify proof contexts and improve overall proof efficiency.

3.3 Frama-C/WP Limitations and the Uncast Plugin

In our project, we use the Typed memory model of WP [7] for its ability to provide a good trade-off between expressivity of provable properties and the capacity to discharge proof goals with automatic provers. The major limitation of this memory model is its inability to handle pointer casts and union types. The majority of unsupported operations is related to heterogeneous pointer casts (including byte splits or concatenations) that must be rewritten before the proof.

This observation motivated a new Frama-C plugin, called Uncast[2], that automatically rewrites common patterns of heterogeneous pointer casts into an equivalent code using arithmetic operations supported by the tool. An example of such a rewriting is the following (assuming a big-endian architecture):

```
1 typedef unsigned char u1; typedef unsigned short u2;
2 ...
3 u1 arr[10]; // Read bytes 4,5 of arr as an unsigned short into data:
4 // u2 data = *(u2*)(arr + 4); // before rewriting of casts (original code);
5 u2 data = (u2)( (*(arr + 4))*256 + *(arr + 5) ); // after rewriting of casts.
```

In this example, the real-life code before rewriting (shown on line 4) first constructs a pointer to the byte at offset 4 in array `arr`, than casts it to an unsigned short pointer (`u2*` type) to read the unsigned short value starting at that address, and therefore represented by the two bytes at offsets 4 and 5. The rewriting (line 5) avoids this pointer cast by reading the bytes at offsets 4 and 5 separately in order to compute the required unsigned short value arithmetically, without any pointer casts.

The Uncast plugin was specified by Thales and implemented by the Frama-C team. In addition, the transformations of all patterns were formally verified using Eva [13], the static analysis plugin of Frama-C. The use of Uncast brings significant benefits to our project: *it automates a large number of code transformations and provides strong confidence in their correctness.*

Examples of other limitations of Frama-C/WP include function pointers, unions, setjmp/longjmp instructions, volatile variables, and pointer-to-integer conversion. These constructs still require *manual code transformations* today.

3.4 Analysis Scope vs. Analysis View

In large-scale industrial projects, formal verification is often applied only to a subset of the codebase, excluding less critical components or parts that have

[2] The Uncast plugin is currently not yet open-source.

not yet been verified. Moreover, the verified code may contain *rewritten parts* or *additional hypotheses* (on the unproven parts or the environment), which should not pollute the original code. ACSL **`admit`** clauses can be used to introduce hypotheses as local properties. *Stubs* (i.e. declared functions with ACSL contracts but without function definition) can be used to abstract away low-level or irrelevant features and to introduce hypotheses about them. It is therefore convenient to distinguish and carefully track two aspects of the code: a *scope* and a *view*.

Scope. An *(analysis) scope* consists of the set of function definitions located in a set of input source files that are submitted to formal verification with Frama-C/WP. The scope is selected through a preprocessed macro in order to ignore other functions—considered out of scope—in the input source files. Notice that ACSL contracts of out-of-scope functions called from within the scope functions are needed for the proof, and included as stubs. Once the scope it selected, both its *original* and *analysis* versions (or *views*) should be tracked for traceability.

View. The *(analysis) view* of the scope applies to it all other code transformations required to complete the formal analysis with Frama-C/WP on functions inside the scope. These code transformations are necessary to tackle limitations of Frama-C/WP to handle some C constructs. The analysis view also adds all necessary ACSL annotations and ghost code. We use preprocessed macros to switch the scope from the original view to the analysis view. This enforces the traceability of all code transformations (clearly indicated in the code by the occurrences of the view changing macro) with no impact on the original code.

Source Code Preparation. On request of the development teams, source code preparation must ensure a structured integration of ACSL annotations into the original codebase without any effect on the original code compilation. Frama-C relies on an external C compiler (GCC by default) to preprocess C source files. We use configuration files (called *machdep files*) to set up compilation features for our targeted machine architectures, such as endianness and sizes of integer and pointer types. The code preparation for the analysis requires inclusion of carefully identified source files and analysis-specific headers, and relies on suitable macro definitions.

4 Integrated Development Environment for Frama-C/WP

4.1 Motivation

Frama-C graphical user interface (GUI) provides only a static display of the analyzed C code. Only a limited interactive mode to edit proof scripts and (re)run the proof for goals generated in the current session is provided. This is inconvenient for an intensive usage of Frama-C on large C codebases with

ACSL annotations. *Code editing*—typically to add or update ACSL annotations—must be done in a separate text editor with no support for navigation through ACSL constructs. In addition, multiple runs of Frama-C—currently executed in a separate console—are often necessary to complete deductive verification with WP. Running the proof from an IDE would be far more convenient.

Moreover, advanced use of Frama-C requires careful configuration of analysis options, such as command-line options to configure possible interactions between several plugins or options to tune the parsing and processing of targeted C code. The usual practice is to configure these options in a Makefile after getting enough knowledge about option ordering, priorities and code structure. An IDE dedicated to Frama-C, providing a configuration file with patterns of preconfigured options for common analyses, would greatly facilitate configuration setup.

4.2 The IDE Design

The Language Server Protocol (LSP), devised by Microsoft [35], provides a solution to avoid reimplementing the same programming-language features for every code editor and for each language. The LSP specifies a structured communication system between a client (the text editor) and a server that provides language-specific features. This communication is standardized by several message formats built on top of JSON-RPC[3].

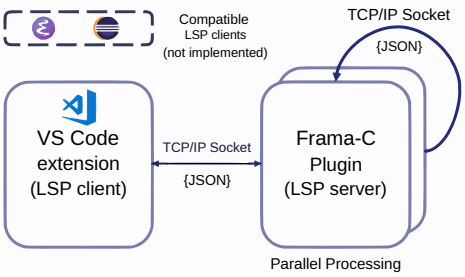

Fig. 3: VS Code extension for C/ACSL.

LSP server. We implement the C/ACSL LSP server as a new Frama-C plugin, Frama-C-lsp, in order to benefit from the full set of functionalities provided by the Frama-C API for handling C and ACSL. The design of this new LSP plugin constitutes the main contribution of our IDE solution. The plugin implements parallel processing of several Frama-C sessions (cf. right side of Fig. 3). A permanent handler Frama-C session connects to the LSP client through TCP/IP sockets locally on the host machine. It initiates a permanent listening loop on a dedicated connection port to manage incoming and outgoing requests.

Upon receiving a request, the handler session invokes temporary Frama-C sessions with appropriate options to produce the required results. These temporary sessions are executed transparently in the background and send their results back to the permanent handler session, which in turn forwards the response to the client. Communication between the handler session and the temporary Frama-C sessions is also performed via local TCP/IP sockets using the JSON format.

If an error occurs during the execution of a temporary session, the handler catches the exception and returns an appropriate LSP error response. This

[3] Remote Procedure Call protocol: https://www.jsonrpc.org/

server architecture avoids any interference with Frama-C's internal code representation throughout IDE usage. Incremental code parsing is handled seamlessly by spawning a new Frama-C session for each request. Our concerns about degraded user experience due to significant execution latency were dissipated after several trials, which demonstrated efficient execution performance. Overall, the user experience is largely enhanced compared to running multiple sessions of Frama-C on command line through a separate console.

LSP client: VS Code editor. We implement the LSP client as a VS Code extension called `acsl-lsp` [4]. Other editors such as Emacs and Eclipse are also LSP compatible and may implement an equivalent LSP client. In addition to a few useful LSP features, we implement in our VS Code extension other handy features specific to the use of Frama-C:

- *Syntax highlighting*: VS Code allows to apply syntax highlighting based on TextMate (TM) grammar. It allows to define patterns and scopes to highlight various syntax elements such as keywords, strings, and comments. We extend an existing syntax highlighting solution for C and C++ [5].
- *Diagnostics*. A file parsing by Frama-C is performed each time a file is opened or saved. We implement event listeners in our Frama-C plugin to catch any errors or warnings raised during code parsing. All diagnostics (errors and warnings) are displayed to the user in the form of squiggles at corresponding lines of code (and listed in the "Problems" tab).
- *Goto definition/declaration*. The editor moves the focus and navigates to the definition/declaration of symbols within the code. A proper information message is displayed if no definition/declaration is found.
- *Display-CIL*. Display the normalized C program pre-processed by Frama-C.
- *Display-CallGraph*. Display the call-graph computed by the callgraph plugin.
- *Display-Metric*. Display code metrics computed by the metrics plugin.
- *Display-ProofObligation*. Display the proof context(s) computed by WP plugin for a selected ACSL annotation if applicable.
- *Prove*: Trigger proof attempts with WP plugin given a function name, an ACSL annotation label and a timeout in seconds. The proof verdict is displayed in the "Goals" view of the editor.
- *SmokeTests*: Trigger proof attempts with WP plugin for automatically generated smoke tests. Their proof verdict is displayed in the "Goals" view of the editor.

5 Proof Effort and Productivity

In software development, the monitoring of key metrics is useful to optimize code quality, reduce maintenance costs, predict future development efforts and detect deviations from best coding practices. The key metrics for our project during

[4] https://marketplace.visualstudio.com/items?itemName=innov-org.acsl-lsp
[5] https://marketplace.visualstudio.com/items?itemName=ms-vscode.cpptools

Year	FCv	C	ACSL	Metapr.	Goals	Scripts (auto)	Proof time	GenScript time
2021	25.0	7 175	26 700	54	70 335	2 383 (0)	10 h 07 m	-
2022	25.0	8 211	44 600	48	74 800	936 (0)	09 h 10 m	-
2023	27.1	7 974	27 677	54	82 890	738 (0)	04 h 10 m	-
2024	29.0	9 877	29 638	53	90 477	459 (289)	07 h 11 m	08 h 30 m
2025	30.0	11 223	31 223	58	96 736	913 (647)	06 h 58 m	03 h 55 m

Fig. 4: Main proof metrics for selected milestone versions, indicating: the year; version of Frama-C; size (loc) of C code and ACSL specs; numbers of metaproperties, of all generated proof goals, of goals proved with all scripts (and with automatically generated ones if any); time of complete proof and of GenScript.

the last five years are summarized in Fig. 4. All metrics are collected on Linux (Ubuntu) machines with 32Gb RAM and 12 cores CPU.

5.1 Scope Extension Effort

The initial version in 2021 was proved using Frama-C 25.0. We extended the scope of analyzed C code from 7 175 lines of code up to 11 223 lines in a recent version proved with Frama-C 30.0. The size of ACSL annotations reached a peak of five times the size of analyzed code in 2022. Refactoring of redundant ACSL annotations with global preprocessed macros reduced the size of ACSL annotations to approx. three times the size of analyzed code. Metaproperies remained stable in several consecutive evaluation projects. This confirms that well-designed metaproperies remain relevant without being affected by extensions of the scope of analyzed code. The number of generated proof goals by Frama-C/WP kept a quite linear progression w.r.t. the size of analyzed code. Careful design of ghost code and ACSL annotations [22] along with code refactoring of complex functions allowed to reduce the number of necessary proof scripts (from ∼3% in 2021 to ∼1% in 2025).

5.2 Automation of Proof Script Creation and GenScript

An important concern identified since the first versions of our project was the prohibitive effort to devise and maintain a large number of manual proof scripts (2 383 in 2021). Even if the number of necessary scripts is relatively small compared to the total number of proof goals (and decreased since 2021), it still requires an important effort to maintain them. Proof scripts are fragile and need to be updated even for limited changes in the code and each new version of Frama-C (cf. Sect. 3).

Following our request to the Frama-C developers, a new feature (referred to as GenScript) has been introduced since Frama-C 29.0 (released in 2024) to automatically generate proof scripts. The generation is based on specific user-provided ACSL annotations called *(proof) strategies* [19] that basically indicate a sequence of tactics to try for certain goals. We achieved a very promising automation rate with 289 proof scripts generated automatically out of 459 in 2024

Tactic	filter	overflow	split	modmask	unfold	shift	bittestrange	cut	range	instance	ratio
All occurr. 2023/fc27.1	179	238	1167	82	1323	68	6	130	2	178	4.6
Scripts 2023/fc27.1	86	128	452	35	315	43	6	63	2	105	1.7
All occurr. 2025/fc30.0	535	998	2043	90	2341	215	36	104	191	210	7.2
Scripts 2025/fc30.0	219	102	531	48	367	58	8	72	63	124	1.7
Auto level 2025/fc30.0											

Fig. 5: Total number of occurrences of each tactic in proof scripts (line All occurr.) and number of scripts that use at least one occurrence of the tactic (line Scripts), shown for 2023 and 2025 versions. The last column (ratio) shows the average number of tactic applications per script (line All occurr.), and the average number of different tactics per script (line Scripts). The last line indicates the current level of automation with GenScript (estimated by the authors).

(and 647 out of 913 in 2025). The main advantage of automatically generated scripts is that their generation from strategies is *less fragile* and less dependent on minor code changes or the used Frama-C version. *The effort to design proof strategies is spent only once.*

Our attempts to further exploit automatic script generation are currently limited by some missing features in the current implementation of GenScript that we discuss below in this section. Ideally, there should be no manual proof scripts to maintain in order to be able to fully integrate the formal verification approach with Frama-C/WP in a continuous integration process.

It is interesting to assess and further improve the efficiency of GenScript to generate proof scripts. Currently, the generation time (3 h 55 min spent in our latest milestone) is incomparable with numerous days of human effort that were required to design the scripts manually.

5.3 A Closer Look at Proof Tactics

Figure 5 shows proof tactics usage statistics in our proof scripts for 2023 and 2025 versions, before and after integration of GenScript.

The `unfold` and `split` tactics are the most used in proof scripts. The `instance` tactic—not yet supported by GenScript—is probably the most complicated to apply (the user should correctly instantiate a quantified variable), so a relatively large number of its applications confirms the complexity of proof goals. A large number of applications of other tactics (such as `overflow` or `filter`) is due to a good automation and a clear role.

The ratio column shows that, in average, 1.7 distinct tactics are used in a proof script in both cases, with and without GenScript. However, the total script length becomes relatively high (7.2 tactic applications per script in average) when some scripts are generated automatically with GenScript, while this ratio is lower (4.6 tactic applications per script) for manually edited scripts. This is mainly

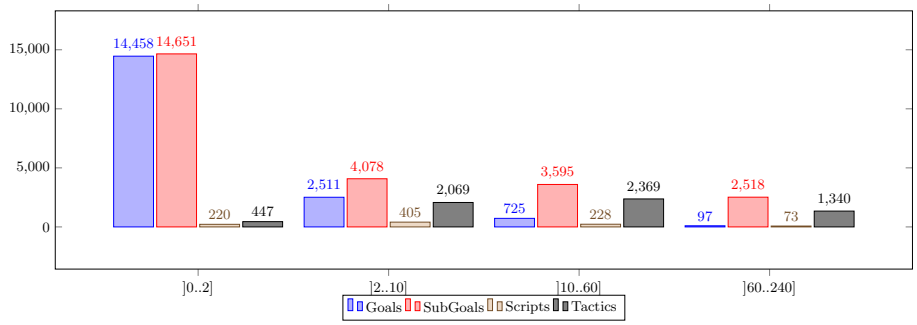

Fig. 6: Number of proof goals and corresponding scripts classified per ranges of proof time.

due to a limited control over multiple applications of tactics with the currently available GenScript implementation. Based on our experimental assessment, we provide (in the last line of Fig. 5) a level of satisfaction with the automation currently provided by GenScript for each tactic. An automation of `instance` and a better control over generated scripts for other tactics (in particular for `split`) should allow us to reach *full automation of proof script generation.*

5.4 Getting Proof Verdicts as Fast as Possible

Proof time for all proof goals depends on several parameters. The main parameter remains the complexity of analyzed code. A switch from a little-endian architecture (in 2022) to a big-endian one (in 2023), easier for the solver, reduced the proof time by ∼40 min. Code refactoring to reduce *cyclomatic complexity* of a few functions also contributed to improve proof efficiency.

Time of a full proof session of Frama-C/WP over all source files at once is 6 h 58 m for the latest version of the project. This duration is quite high and allows only daily proof runs to get a complete full verdict of proof status. In practice, *we rely on distributed proof sessions on several machines,* each one running Frama-C/WP over only one source file. This distributed (parallel) computation allows us to get a whole proof verdict in less than 30 min, which is better suited for a productive continuous integration and development process.

6 Proof Efficiency and Performance

Frama-C/WP offers a detailed reporting of proof results in json format. For each proof goal, the time spent by Qed and Alt-Ergo to discharge the goal is reported. If a proof script is used to discharge a proof goal, the proof verdict and proof time are reported for each subgoal generated by application of proof tactics. This section highlights some aspects of the project related to proof performance.

6.1 Number of Proof Goals

Proof performance depends on the number of goals to be proven. Some design choices at the level of ACSL annotations have a direct impact on the number of generated proof goals and time required to prove them.

Case of Lemmas. Even if lemmas are proved once and used in different proof contexts, we noticed decreased proof performance with lemmas since solvers check the applicability of lemmas in many irrelevant proof contexts (see Sect. 3.2). Hence, we favor assertions to insert intermediate properties only when needed.

Case of Metaproperies. More than half (60%) of the proof goals are generated for metaproperties. The general treatment of a metaproperty with a reading or writing context consists in generating an assertion at each relevant (resp., read or write) instruction in target functions. For complex functions, this treatment can sometimes become inefficient due to a large number of assertions and a complex logic inside the function. In such cases, we rely on a more efficient alternative based on **`assigns...\from...`** clauses. Indeed, an **`assigns...\from...`** clause gives the lists of variables that the function is allowed, resp., to modify and to read. In some cases, it provides sufficient information to justify some metaproperties at the call site, provided that **`assigns...\from...`** clauses are proved. The **`assigns...`** part is directly proved by WP, but the **`\from ...`** part is not yet handled. So, we add (very simple) metaproperies to prove the **`\from ...`** clauses—that is, non-reading of any non-listed variables—in targeted functions. This solution allowed us (again in 2023) to reduce the proof time by 1 h and to avoid costly proof goals inside complex functions.

Case of Scripts. The majority (80%) of proof goals are discharged by Qed alone with negligible reported time (0 s). This is an important achievement of proof automation with Frama-C/WP, but the remaining goals still represent a large number. In this section, we focus only on the (other 20% of) goals that require (in addition to Qed) an SMT prover (in our case, Alt-Ergo) to be proven. We set a global timeout per goal to 240 s to favor automatic proof without scripts.

We consider a partitioning of the set of proof goals into four disjoint subsets according to the necessary proof time to discharge each of them, with four intervals: $]0s..2s]$, $]2s..10s]$, $]10s..60s]$, $]60s..240s]$. Figure 6 shows with blue bars the number of goals in each subset for our latest version. The difference between the number of goals and the number of proof scripts (in brown) gives the number of goals automatically proved with Alt-Ergo without scripts. Only $97 - 73 = 24$ goals are automatically proved by Alt-Ergo in the time range $]60s..240s]$. We still favor automatic proof for them to avoid extra proof scripts to maintain.

220 proof scripts allow to prove corresponding proof goals in the $]0s..2s]$ time range. About half of proof scripts (405) prove corresponding proof goals in the $]2s..10s]$ time range. It is important to monitor the efficiency of proof scripts

because costly scripts are often difficult to maintain and to fix when they are broken.

Proof tactics that split the proof goal into subgoals are often useful but must be used with care to avoid duplication of proofs over generated subgoals instead of reducing the complexity of initial goals. The difference between the number of goals and the number of subgoals (in red) in Fig. 6 illustrated the number of extra subgoals created by tactics.

The ratio between the number of scripts and the number of tactics (black bars) is a relevant indicator on the manual effort required to create proof scripts. While this ratio is reasonable (2 tactics in average) for efficient scripts in the $]0s..2s]$ proof time range, it is dramatically higher for the other scripts (resp., 5, 10, 18 tactics in average per script in $]2s..10s]$, $]10s..60s]$, $]60s..240s]$ time ranges). As a guideline, *the sum of proof time for subgoals of each goal should be less than the considered timeout,* which is the case for our scripts. Otherwise, trying a greater solver timeout could be more efficient (and avoid a script).

6.2 Proof Time

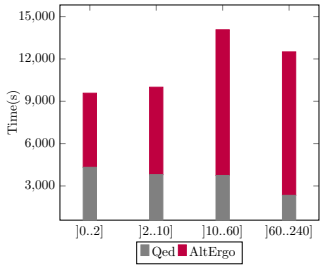

Fig. 7: Proof goals partitioned w.r.t. ranges of proof time.

In our latest version, it takes less than 15 min for Frama-C/WP to generate proof goals and for MetAcsl to instantiate metaproperties. This is negligible compared to 6 h 58 m of total proof time in one Frama-C session. Let us consider the same four subsets of goals as in Sect. 6.2 and focus on the reported CPU time to prove goals (without including time for their generation). The total CPU time to prove all goals (12 h 53 m) is higher than the real time (6 h 58 m) also because we let Frama-C run solvers for some goals in parallel on up to 9 cores. Figure 7 shows the total CPU time spent by Qed and Alt-Ergo to discharge each subset of goals in our latest version.

We observe that the time spent by Qed is not negligible. Unlike for Alt-Ergo, it decreases for costly provable goals but still it is not linearly proportional to the number of goals in each subset. Qed simplifications are very efficient in average for simpler subgoals in the time range $]0s..2s]$ compared to more costly provable goals. We also observed that it is beneficial to disable one Qed simplification (`-no-qed-pruning`) to avoid some costly (and unproductive) simplification attempts. Another observation is that, above 10 s of proof time for each goal, Alt-Ergo takes a long time to discharge the goals. This is correlated to the usage of proof scripts and emphasizes the need to better diagnose the challenging proof contexts for Alt-Ergo. The last but not least observation is that the more the scripts are complex, the more they are costly to prove and penalize the global proof efficiency. Interestingly, it takes negligible CPU time to prove 80% of proof goals discharged by Qed only; and it takes much less CPU time (2 h 40 m) to prove

14 458 goals in the time range $]0s, 2s]$ than the CPU time (10 h 13 m) to prove the remaining 3 333 goals in the other time ranges together.

7 Conclusion

Thanks to the impressive progress made by researchers and tool developers in the past decades, formal verification has become feasible in an increasing number of industrial projects today [28], using various verification tools, for instance, KeY [9, 21], VerCors [6, 36], Frama-C [25], SPARK [15, 26], VCC [33], Dafny [14, 34], VeriFast [30]. However, an initial success—for which investing an important human effort in an industrial verification project can be acceptable—does not guarantee the sustained use of formal verification in long-term projects. This paper presents an experience report on the path towards sustained industrial use of formal verification for the certification of a JavaCard Virtual Machine at Thales. Our achievements have contributed to reducing proof maintenance effort and increasing its performance, and the proposed solutions could benefit to researchers and practitioners in the formal methods community and further enhance proof tools and methodologies.

A finalization of the initiated automatization tasks is mandatory in order to completely endorse the adoption of deductive verification in an industrial continuous development process. In the context of our project, future work includes development and adoption of new versions of GenScript, the automatic script generation mechanism in Frama-C/WP. With a better control over patterns of tactic application and the support of new tactics, the ultimate objective of a 100% automation for proof script generation should be achievable. The proof maintenance effort will be completely under control. The development of more efficient and flexible memory models should reduce the need for code transformations. Another future task would be an advanced study of costly provable goals at the level of SMT solvers in order to suggest possible enhancement of Alt-Ergo performance on our use cases. Beyond the global proof performance improvement, the objective is to examine the ultimate possibility to refine the trade-off between our ACSL design choices and proof performance. Finally, we hope that our open-source IDE solution for Frama-C/WP would benefit to a large number of Frama-C users. It may facilitate the usage of the verification tool by non-expert engineers and students, and we warmly welcome all relevant contributions to further improve it.

Acknowledgement. Part of this work was supported by ANR (grants ANR-22-CE39-0014, ANR-22-CE25-0018, ANR-24-ASM2-0001). The development of the Frama-C-Isp plugin was initiated by Djamila Mohamed during her internship at Thales. The authors thank the Frama-C team members for their support, and the anonymous reviewers for their helpful comments.

References

1. ANSSI: Common Criteria evaluation certificate delivered on 2021-10-14 (ref. ANSSI-CC-2021/42, EAL6+), https://messervices.cyber.gouv.fr/visas/ANSSI-CC-2021-42-certificat.pdf
2. ANSSI: Common Criteria evaluation certificate delivered on 2022-04-22 (ref. ANSSI-CC-2020/33-S01, EAL7), https://messervices.cyber.gouv.fr/visas/ANSSI-CC-2020-33-S01-certificat.pdf
3. ANSSI: Common Criteria evaluation certificate delivered on 2023-08-28 (ref. ANSSI-CC-2023/31, EAL6+), https://messervices.cyber.gouv.fr/visas/ANSSI-CC-2023-31-certificat.pdf
4. ANSSI: Common Criteria evaluation certificate delivered on 2023-11-13 (ref. ANSSI-CC-2023/45, EAL7), https://messervices.cyber.gouv.fr/visas/ANSSI-CC-2023-45-certificat.pdf
5. ANSSI: Common Criteria evaluation certificate delivered on 2025-09-29 (ref. ANSSI-CC-2025/32, EAL6+), https://messervices.cyber.gouv.fr/visas/ANSSI-CC-2025-32-certificat.pdf
6. Armborst, L., Bos, P., van den Haak, L.B., Huisman, M., Rubbens, R., Sakar, Ö., Tasche, P.: The VerCors verifier: A progress report. In: Proc. of the 36th International Conference on Computer Aided Verification (CAV 2024). LNCS, vol. 14682, pp. 3–18. Springer (2024). https://doi.org/10.1007/978-3-031-65630-9_1
7. Baudin, P., Bobot, F., Correnson, L., Dargaye, Z., Blanchard, A.: WP Plug-in Manual (2025), https://frama-c.com/download/frama-c-wp-manual.pdf
8. Baudin, P., Cuoq, P., Filliâtre, J.C., Marché, C., Monate, B., Moy, Y., Prevosto, V.: ACSL: ANSI/ISO C Specification Language (2025), http://frama-c.cea.fr/acsl.html
9. Beckert, B., Sanders, P., Ulbrich, M., Wiesler, J., Witt, S.: Formally verifying an efficient sorter. In: Proc. of the 30th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2024). LNCS, vol. 14570, pp. 268–287. Springer (2024). https://doi.org/10.1007/978-3-031-57246-3_15
10. Blanchard, A., Bobot, F., Baudin, P., Correnson, L.: Chapter 4: Formally verifying that a program does what it should: The WP plug-in. In: Kosmatov, N., Prevosto, V., Signoles, J. (eds.) Guide to Software Verification with Frama-C. Core Components, Usages, and Applications, pp. 187–261. Computer Science Foundations and Applied Logic Book Series, Springer (2024). https://doi.org/10.1007/978-3-031-55608-1_4
11. Blanchard, A., Correnson, L., Djoudi, A., Kosmatov, N.: No smoke without fire: Detecting specification inconsistencies with Frama-C/WP. In: Proc. of the 18th International Conference on Tests and Proofs (TAP 2024), co-located with the 26th International Symposium on Formal Methods (FM 2024). LNCS, vol. 15153, pp. 65–83. Springer (Sep 2024). https://doi.org/10.1007/978-3-031-72044-4_4
12. Blanchard, A., Marché, C., Prevosto, V.: Chapter 1: Formally expressing what a program should do: The ACSL language. In: Kosmatov, N., Prevosto, V., Signoles, J. (eds.) Guide to Software Verification with Frama-C. Core Components, Usages, and Applications, pp. 3–80. Computer Science Foundations and Applied Logic Book Series, Springer (2024). https://doi.org/10.1007/978-3-031-55608-1_1

13. Bühler, D., Maroneze, A., Perrelle, V.: Chapter 3: Abstract interpretation with the eva plug-in. In: Kosmatov, N., Prevosto, V., Signoles, J. (eds.) Guide to Software Verification with Frama-C. Core Components, Usages, and Applications, pp. 131–186. Computer Science Foundations and Applied Logic Book Series, Springer (2024). https://doi.org/10.1007/978-3-031-55608-1_3
14. Cassez, F., Fuller, J., Quiles, H.M.A.: Deductive verification of smart contracts with Dafny. In: Proc. of the 27th International Conference on Formal Methods for Industrial Critical Systems (FMICS 2022). LNCS, vol. 13487, pp. 50–66. Springer (2022). https://doi.org/10.1007/978-3-031-15008-1_5
15. Cluzel, G., Georgiou, K., Moy, Y., Zeller, C.: Layered formal verification of a TCP stack. In: Proc. of the IEEE Secure Development Conference (SecDev 2021). pp. 86–93. IEEE (2021). https://doi.org/10.1109/SecDev51306.2021.00028
16. Common Criteria: Common criteria for information technology security evaluation. Part 3: Security assurance components. Tech. rep., CCMB-2017-04-003 (2017), https://www.commoncriteriaportal.org/files/ccfiles/CCPART3V3.1R5.pdf
17. Conchon, S., et al.: The Alt-Ergo automated theorem prover., http://alt-ergo.lri.fr
18. Correnson, L.: Qed. Computing what remains to be proved. In: NASA Formal Methods (NFM 2014). LNCS, vol. 8430, pp. 215–229. Springer (2014). https://doi.org/10.1007/978-3-319-06200-6_17
19. Correnson, L., Blanchard, A., Djoudi, A., Kosmatov, N.: Automate where automation fails: Proof strategies for Frama-C/WP. In: Proc. of the 30th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2024), Held as Part of the European Joint Conferences on Theory and Practice of Software (ETAPS 2024). LNCS, vol. 14570, pp. 331–339. Springer (Apr 2024). https://doi.org/10.1007/978-3-031-57246-3_18
20. Correnson, L., Cuoq, P., Kirchner, F., Maroneze, A., Prevosto, V., Puccetti, A., Signoles, J., Yakobowski, B.: Frama-C User Manual, http://frama-c.com/download/frama-c-user-manual.pdf
21. de Boer, M., de Gouw, S., Klamroth, J., Jung, C., Ulbrich, M., Weigl, A.: Formal specification and verification of JDK's identity hash map implementation. Formal Aspects Comput. **35**(3), 18:1–18:26 (2023). https://doi.org/10.1145/3594729
22. 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
23. 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)
24. Djoudi, A., Hána, M., Kosmatov, N.: Chapter 16: Proof of security properties: Application to JavaCard virtual machine. In: Kosmatov, N., Prevosto, V., Signoles, J. (eds.) Guide to Software Verification with Frama-C. Core Components, Usages, and Applications, pp. 659–683. Computer Science Foundations and Applied Logic Book Series, Springer (2024). https://doi.org/10.1007/978-3-031-55608-1_16

25. Dordowsky, F.: An experimental study using ACSL and Frama-C to formulate and verify low-level requirements from a DO-178C compliant avionics project. Electronic Proceedings in Theoretical Computer Science **187**, 28–41 (2015). https://doi.org/10.4204/EPTCS.187.3
26. Dross, C., Moy, Y.: Auto-active proof of red-black trees in SPARK. In: Proc. of the 9th International Symposium on NASA Formal Methods (NFM 2017). LNCS, vol. 10227, pp. 68–83 (2017). https://doi.org/10.1007/978-3-319-57288-8_5
27. Filliâtre, J., Paskevich, A.: Why3 – where programs meet provers. In: Proc. of the 22nd European Symp. on Programming (ESOP 2013). LNCS, vol. 7792, pp. 125–128. Springer (2013). https://doi.org/10.1007/978-3-642-37036-6_8
28. Hähnle, R., Huisman, M.: Deductive software verification: From pen-and-paper proofs to industrial tools. In: Computing and Software Science – State of the Art and Perspectives, LNCS, vol. 10000, pp. 345–373. Springer (2019). https://doi.org/10.1007/978-3-319-91908-9_18
29. ISO/IEC JTC1/SC22/WG14: 9899:TC3: Programming Languages—C (2007), http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1256.pdf
30. Jacobs, B., Smans, J., Philippaerts, P., Vogels, F., Penninckx, W., Piessens, F.: VeriFast: A powerful, sound, predictable, fast verifier for C and Java. In: Proc. of the Third International Symposium on NASA Formal Methods (NFM 2011). LNCS, vol. 6617, pp. 41–55. Springer (2011). https://doi.org/10.1007/978-3-642-20398-5_4
31. Kirchner, F., Kosmatov, N., Prevosto, V., Signoles, J., Yakobowski, B.: Frama-C: A software analysis perspective. Formal Asp. Comput. pp. 1–37 (2015). https://doi.org/10.1007/s00165-014-0326-7
32. 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
33. Leinenbach, D., Santen, T.: Verifying the microsoft Hyper-V hypervisor with VCC. In: Proc. of the Second World Congres on Formal Methods (FM 2009). LNCS, vol. 5850, pp. 806–809. Springer (2009). https://doi.org/10.1007/978-3-642-05089-3_51
34. Leino, K.R.M.: Program Proofs. The MIT Press (2023)
35. Microsoft: Language Server Protocol, https://microsoft.github.io/language-server-protocol/
36. Oortwijn, W., Huisman, M.: Formal verification of an industrial safety-critical traffic tunnel control system. In: Proc. of the 15th International Conference on Integrated Formal Methods (IFM 2019). LNCS, vol. 11918, pp. 418–436. Springer (2019). https://doi.org/10.1007/978-3-030-34968-4_23
37. Oracle: Java Card Platform: Runtime Environment Specification, Classic Edition, Version 3.1. Tech. rep., Oracle, Oracle (feb 2021), https://docs.oracle.com/javacard/3.1/related-docs/JCCRE/JCCRE.pdf
38. Oracle: Java Card Platform: Virtual Machine Specification, Classic Edition, Version 3.1. Tech. rep., Oracle, Oracle (feb 2021), https://docs.oracle.com/javacard/3.1/related-docs/JCVMS/JCVMS.pdf
39. Rice, H.G.: Classes of recursively enumerable sets and their decision problems. Trans. Amer. Math. Soc. **74**, 358–366 (1953)
40. Robles, V., Kosmatov, N., Prevosto, V., Rilling, L., Gall, P.L.: MetAcsl: Specification and verification of high-level properties. In: International Conference on

Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2019). LNCS, vol. 11427, pp. 358–364. Springer (2019). https://doi.org/10.1007/978-3-030-17462-0_22
41. Robles, V., Kosmatov, N., Prevosto, V., Rilling, L., Le Gall, P.: Methodology for specification and verification of high-level properties with MetAcsl. In: 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
42. Thales Press Release: Thales launches Europe's first certified smartcard ready for the Quantum Age, https://www.thalesgroup.com/en/news-centre/press-releases/thales-launches-europes-first-certified-smartcard-ready-quantum-age

A Appendix: Supplementary Material

This appendix is added for convenience of the reviewers, not for publication. It provides some additional technical details.

A.1 Scope and View Interaction

For any part of the codebase, it can be either in or out of the analysis scope (that is, either viewed by Frama-C for formal verification or not). When it is in the analysis scope, its interpretation by Frama-C depends on whether the analysis view of the scope is activated or not. Warnings returned by Frama-C and WP when parsing code out of the analysis scope provide interesting metrics on necessary manual effort required to extend the scope (and the analysis view) to that code.

Unfortunately, parsing code both in and out of the analysis scope at once with Frama-C results in type conflicts when the analysis view of the scope is activated. Indeed, the activation of the analysis view creates duplicate declarations for some global variables and functions for which a type change is needed for the proof. This type change has a pervasive effect on all the code. In the case of function declarations when type definition of parameters is changed for instance, the activation of the analysis view entails manual modification of corresponding function declarations and their calls in the code. In order to avoid this manual modification, we avoid duplicate declarations for relevant global variables and functions by using alternative declarations instead. For instance, when the definition of type `t` is changed from type **`int`** in the original view into type **`int`**`*` in the analysis view, these alternative declarations are encapsulated in global C macros with the same name as in the original view in order to override each occurrence of the original type definition with the new one. And the type usage in the code (both in the analysis view and the original view) is not changed.

Example of original type definition and function declaration in the original view:

```
1 typedef int t;
2 void function (t);
```

Example of alternative type definition and function declaration in the analysis view:

```
1 typedef int* t_view;
2 #define t t_analysis_view
3 #define function function_analysis_view
4 void function (t);
```

This structure of the analysis view is very convenient and avoids pervasive changes in the original code. The macro preprocessing allows to get the analysis view active in the analysis scope while being inactive for the rest of the code when parsed by Frama-C without triggering type conflicts.

A.2 Source Code Preparation for Analysis with Frama-C/WP

In terms of usage, Frama-C provides four main executables, describes in the Frama-C manual [20].

1. frama-c: batch command used to run the source code analyses through invocation of analysis plugins;
2. frama-c-gui (and recently ivette): graphical user interface (GUI) offering limited yet necessary interactive usage mode;
3. frama-c-config: light batch command used to retrieve practical configuration information on the current frama-c installation;
4. frama-c-script: an experimental set of tools aiming to help the user to prepare the source code for analysis, often used for the Eva plugin.

Unfortunately, the frama-c-script executable does not currently offer enough source code preparation features to automatically address all configuration issues faced when applying deductive verification with WP plugin. We share here some useful steps to prepare C source code for formal deductive verification with the WP plugin in a realistic industrial context.

Frama-C relies on an external C compiler (GCC by default) to preprocess C source files. We use configuration files (called *machdep files*) to set up compilation features compatible with our targeted machine architectures. Such features include endianness and sizes of integer and pointer types. The main challenge here is to ensure a structured integration of ACSL annotations into the original codebase without any effect on the original code compilation. ACSL annotations introduce new dependencies on global variables and ghost code that must be handled seamlessly with the following standard steps for source code preparation:

1. identify source files (.c) to be considered in the analysis scope;
2. ignore function definitions (or specific instructions) outside of the analysis scope with a specific macro;
3. identify header files (.h) to be considered in the analysis scope of Frama-C along with their related include paths (Frama-C option `-cpp-extra-args` is used to specify include paths);
4. for each function in the analysis scope declared in `module/file.h`, add an ACSL function contract on a copy of its prototype in a Frama-C-specific header file `module/frama_c/fc_file.h`. The Frama-C-specific header files must be included separately for Frama-C/WP analysis without any impact on original code compilation. Static inline functions should be inlined by Frama-C in order to avoid dependencies when defining ACSL function contracts for them.
5. define global header files for definition of ACSL predicates and global (ghost) variables. These header files should be named with a specific prefix (e.g. `fc_`) and put in a specific folder (e.g. `frama_c`) at the project root directory. Header files specific to Frama-C (prefixed with `fc_`) should have no dependencies on other header file. File dependencies of original code must remain unchanged to avoid introducing any complex cyclic dependencies in original code that might be difficult de debug.

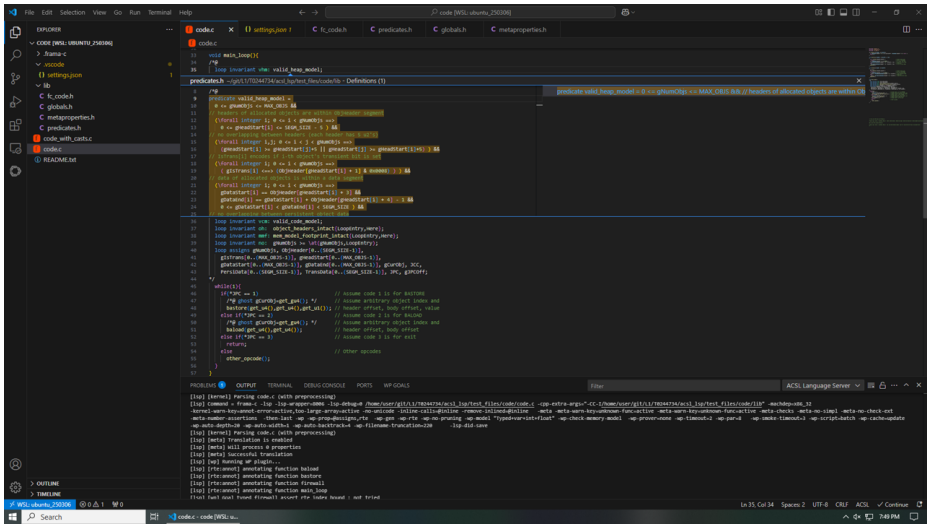

Fig. 8: Definition of an ACSL predicate displayed on request at its usage location.

A.3 Examples of IDE features with VS Code extension

In addition to a basic, yet very useful, syntax highlighting of C and ACSL constructs, our VS Code extension for Frama-C/WP offers quite advanced features to edit C code and ACSL specifications and to check the proof status. Here we give two examples of these features. The code samples used here [6] include those considered in our previous publication [22].

GoToDefinition LSP feature. The user may get the definition of an ACSL predicate at its usage location in a function contract, for instance, as shown in Fig. 8. Note that our VS Code extension only provides location of ACSL predicate definition. The display rendering is fully taken in charge by VS Code. Other LSP compatible clients have their own display rendering. The Output view at the bottom of the VS Code window provides the console output of underlying Frama-C command(s) used transparently in the background to get the display results.

Proof feature. One advanced feature of our VS Code extension is to run the proof with Frama-C/WP of ACSL annotations. The proof verdict for each generated proof goal is displayed in the WP Goals view as shown in Fig. 9. The proof may be run for one or several function(s) and ACSL annotations when applicable with a specific timeout for SMT solvers. We use WP options `-wp-fct`, `-wp-prop`, `-wp-timeout` to parametrize the underlying Frama-C command.

[6] Available at https://github.com/ThalesGroup/frama-c-lsp

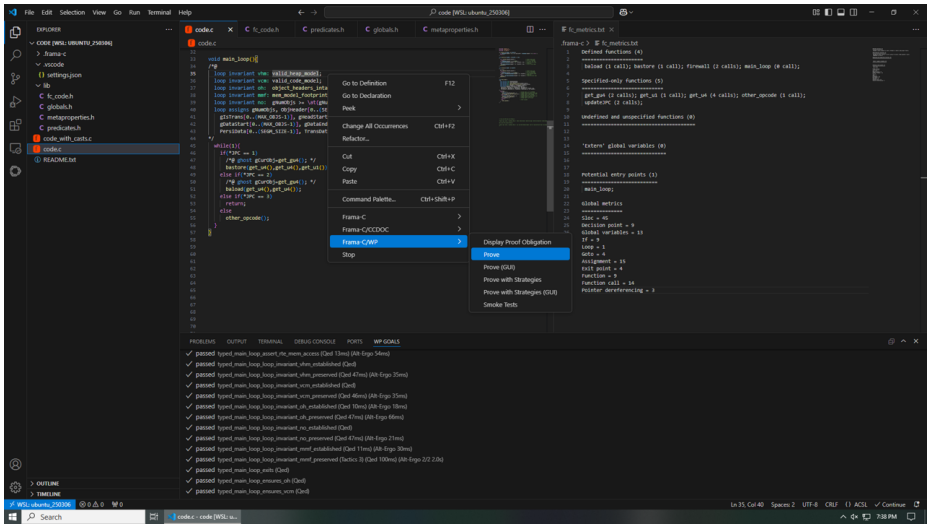

Fig. 9: Triggering proof of ACSL annotations from VS Code extension.

The user may configure the complete set of (default) Frama-C options for the current VS Code workspace in file `.vscode/settings.json`.