

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

Adel DJOUDI, Nikolai KOSMATOV

FM 2026 i-Day  
20 May 2026, Tokyo, Japan

[www.thalesgroup.com](http://www.thalesgroup.com)



# Motivation: Sustained Formal Verification in Industry

## A successful industrial application of formal verification on a new project is a key milestone

- Significant effort can be required and invested at this step

## Proof rework is often required due to the evolving context

- evolution of the codebase (e.g., to integrate new features or updates),
- evolution of the formal specification (e.g., to express additional properties or to extend the verification scope),
- evolution of verification tools (which may break some proofs or require new proof scripts),
- evolving expectations of evaluators and certification authorities.

## Industrial engineers have to verify several evolving projects using evolving tools

- hardly possible to reinvest the same level of effort repeatedly throughout the project lifecycle

## Sustained, long-term use of formal verification brings new productivity challenges

- A higher level of automation (e.g. specification writing and comprehension, proof replay, reporting)
- Improved support for maintaining verified code (including the creation and maintenance of proof scripts)
- ...

# Context: Common Criteria (CC) Certification for Integrated Circuits and Smart cards



ID documents



- EAL7
- EAL6
- EAL5
- EAL4
- EAL3
- EAL2
- EAL1

**Formal verification**  
required for **EAL6/EAL7**  
certification



Certification authority

**Goal: Meet highest-level certification standards expected by customers**

**Target:** Isolation of Applications (Firewall) of an Embedded Operating System (Java Card Virtual Machine)

**Approach:** We apply deductive verification directly on a C implementation using Frama-C/WP

# Feedback from successive certifications

## > Strong points of source-code based verification

- ▶ **Straightforward correspondence between the security functions and formal model**
- ▶ The approach perfectly adapted to the CC evaluation process
- ▶ Immediate understanding of formal entities (e.g. JCVm memory model)
- ▶ No refinement, thus no relation between multiple models to be evaluated
- ▶ The implementation challenges the formal model by construction

## > Issues and points of attention

- ▶ **Code complexity directly transferred to the model**
- ▶ Sensitivity to tool scalability issues
- ▶ Organization of a high number of manual annotations

- **FM 2021:** A.Djoudi, N.Kosmatov “Formal Verification of a JavaCard Virtual Machine with Frama-C”
- **ERTS 2022:** A.Djoudi, M.Hána, N.Kosmatov, M.Kříženecký, F.Ohayon, P.Mouy, A.Fontaine and D.Féλιot “A Bottom-Up Formal Verification Approach for Common Criteria Certification: Application to JavaCard Virtual Machine” (jointly with ANSSI and CEA-Leti, **best paper award**)
- **Book chapter in 2024:** A.Djoudi, M.Hána, N.Kosmatov. Proof of Security Properties: Application to JavaCard Virtual Machine (chapter in Springer book Guide to Software Verification with Frama-C)

Pending  
MAV1.2



October  
2025  
MAV5.2



November  
2023  
MAV5.1



August  
2023  
MAV5.1



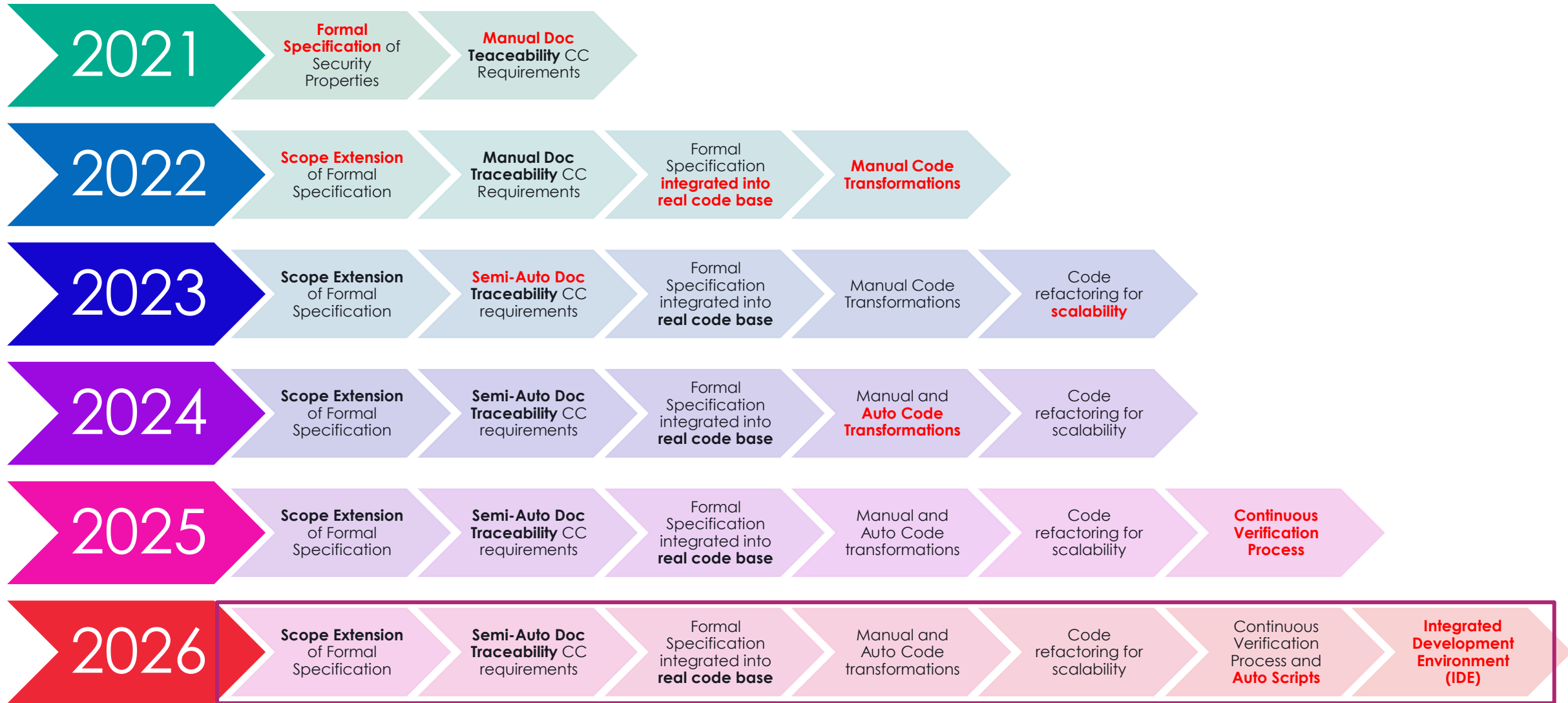
October  
2022  
MAV5.0



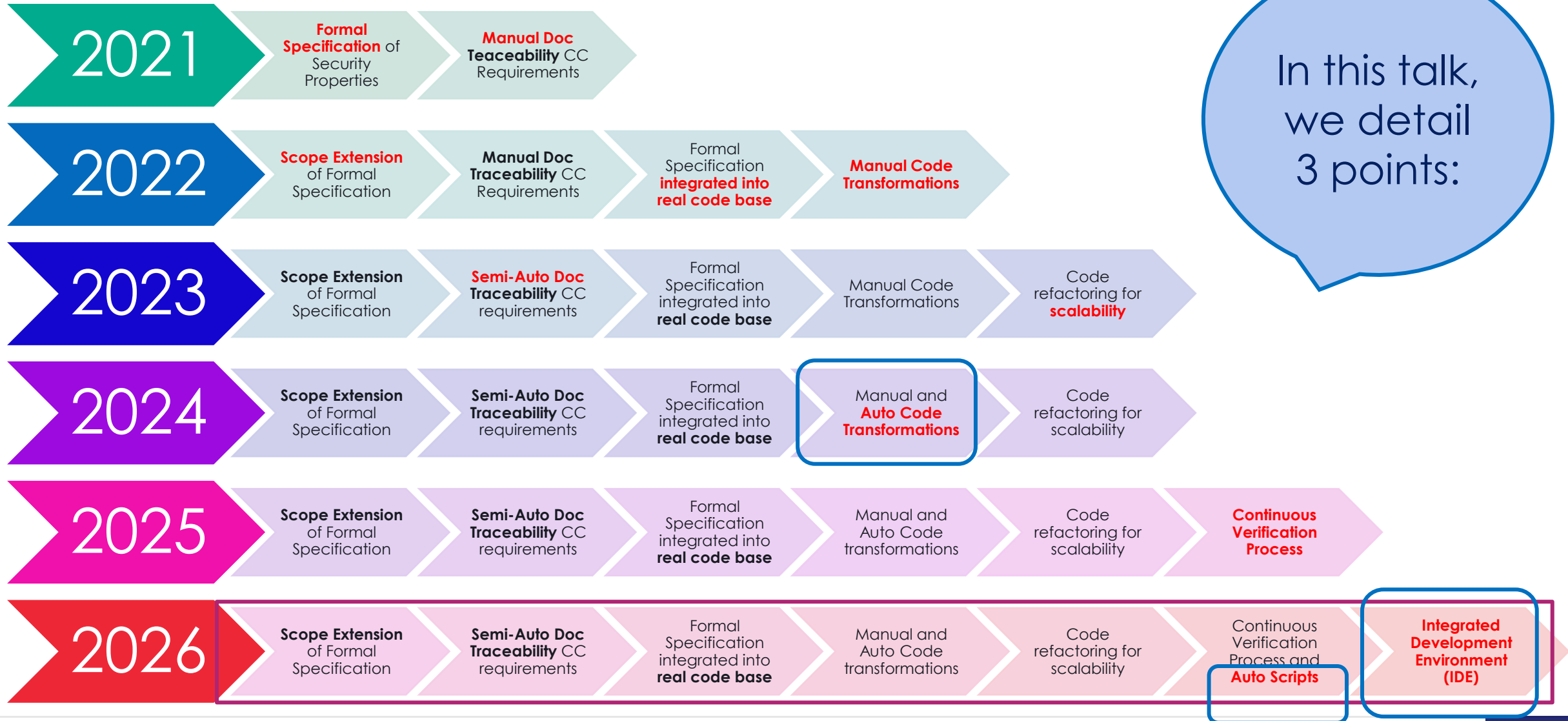
October  
2021  
MAV5.0



# Five years, five successful certifications: Challenges and Achievements



# Five years, five successful certifications: Challenges and Achievements



# Working Example

```
int all_zeros(int *t, int n) {  
    int k=0;  
  
    while(k < n){  
  
        if (t[k] != 0) return 0;  
  
        k++;  
    }  
    return 1;  
}
```

frama-c file.c  
[kernel] Parsing file.c (with preprocessing)

# Working Example with Frama-C/WP: Functional properties

```
/*@ requires n ≥ 0 ∧ \valid(t + (0..n-1));  
    terminates \true;  
    exits \false;  
    ensures \result != 0 ⇔ (∀ Z j; 0 ≤ j < n ⇒ t[j] == 0);  
    assigns \nothing;  
*/
```



function contract in ACSL

```
int all_zeros(int *t, int n) {  
    int k=0;
```

```
    /*@ loop invariant 0 ≤ k ≤ n;  
        loop invariant ∀ Z j; 0 ≤ j < k ⇒ t[j] == 0;  
        loop assigns k;  
        loop variant n-k;  
    */
```



loop contract in ACSL

```
    while(k < n){  
  
        if (t[k] != 0) return 0;  
  
        k++;  
    }  
    return 1;  
}
```


```
frama-c file.c -wp  
[kernel] Parsing file.c (with preprocessing)  
[wp] Proved goals: 13 / 13  
Terminating:      1  
Unreachable:      1  
Qed:               9 (5ms-2ms-17ms)  
Alt-Ergo 2.6.2:   2 (25ms-43ms)
```

# Working Example with Frama-C/WP + RTE: Functional properties + Absence of runtime errors

```
/*@ requires n ≥ 0 ∧ \valid(t + (0..n-1));
    terminates \true;
    exits \false;
    ensures \result != 0 ⇔ (∀ Z j; 0 ≤ j < n ⇒ t[j] == 0);
    assigns \nothing;
*/
int all_zeros(int *t, int n) {
    int k=0;
    /*@ loop invariant 0 ≤ k ≤ n;
        loop invariant ∀ Z j; 0 ≤ j < k ⇒ t[j] == 0;
        loop assigns k;
        loop variant n-k;
    */
    while(k < n){
        /*@ assert rte: mem_access: \valid_read(t + k); */

        if (t[k] != 0) return 0;
        /*@ assert rte: signed_overflow: k + 1 ≤ 2147483647; */
        k++;
    }
    return 1;
}
```

 function contract in ACSL

 loop contract in ACSL

```
frama-c file.c -wp -wp-rte
[kernel] Parsing file.c (with preprocessing)
[wp] Proved goals: 15 / 15
Terminating:      1
Unreachable:     1
Qed:              10 (4ms-2ms-20ms)
Alt-Ergo 2.6.2:  3 (44ms-54ms)
```

# Working Example with Frama-C/WP + RTE + MetAcsl: ... + Security properties

```
/*@ meta \prop, \name(confidentiality), \targets(\ALL), \context(\reading), \separated(\read, &secret[0..9]); */  
char secret[10];
```



Global security property

```
/*@ requires n ≥ 0 ∧ \valid(t + (0..n-1));  
terminates \true;  
exits \false;  
ensures \result != 0 ⇔ (∀ Z j; 0 ≤ j < n ⇒ t[j] == 0);  
assigns \nothing;
```



function contract in ACSL

```
*/
```

```
int all_zeros(int *t, int n) {
```

```
int k=0;
```

```
/*@ loop invariant 0 ≤ k ≤ n;  
loop invariant ∀ Z j; 0 ≤ j < k ⇒ t[j] == 0;  
loop assigns k;  
loop variant n-k;
```

```
*/
```

```
while(k < n){
```

```
/*@ assert rte: mem access: \valid read(t + k); */
```

```
/*@ assert confidentiality: meta: \separated(t + k, &secret[0 .. 9]); */
```

```
if (t[k] != 0) return 0;
```

```
/*@ assert rte: signed_overflow: k + 1 ≤ 2147483647; */
```

```
k++;
```

```
}
```

```
return 1;
```

```
}
```



loop contract in ACSL

```
frama-c file.c -meta -then-last -wp -wp-rte  
[kernel] Parsing file.c (with preprocessing)  
[wp] Proved goals: 16 / 16  
Terminating: 1  
Unreachable: 1  
Qed: 11 (5ms-3ms-23ms)  
Alt-Ergo 2.6.2: 3 (29ms-39ms-54ms)
```

# Automatic code transformations with Uncast plugin

## > Motivation: Code transformations are necessary for the proof with Frama-C/WP for

- Heterogeneous pointer casts – the most frequent ones
- Conversions from pointer to integer
- Function pointers casts
- Endianness swapping

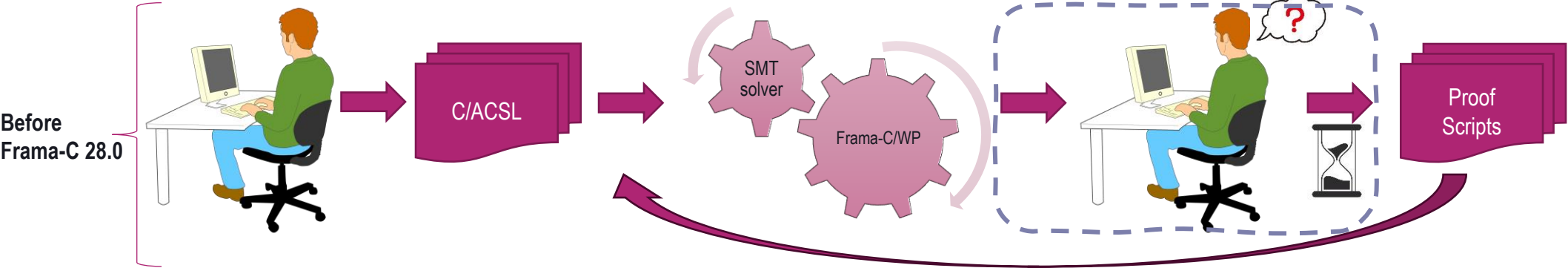
## > Solution: Automatic code transformations for heterogeneous pointer casts with Uncast

Original code	Automatic code rewriting with Uncast (for big-endian architecture)
<pre>unsigned char arr[10];  unsigned short data = *(unsigned short*)(arr + 4);</pre>	<pre>unsigned char arr[10];  unsigned short data = (unsigned short)((*(arr + 4))*256 + *(arr + 5));</pre>

## > Uncast transformations are formally proved to be correct

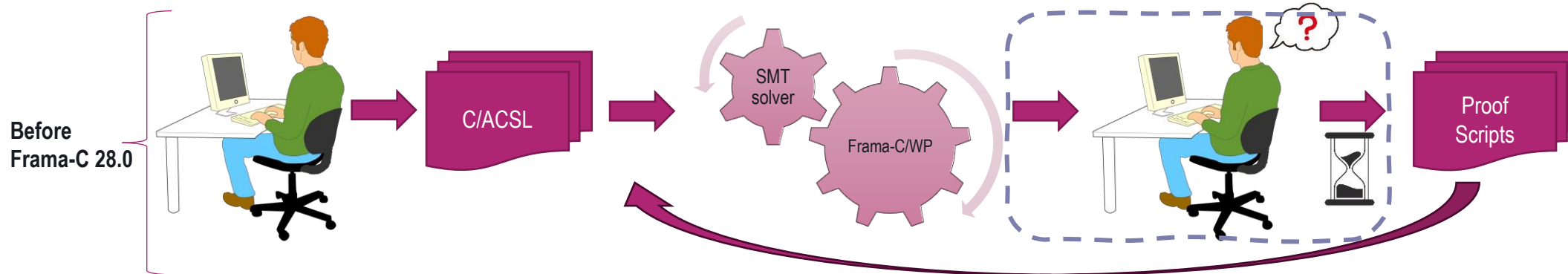
# Manual vs. Automatic Proof Script Creation

Motivation: manual proof script creation is hard

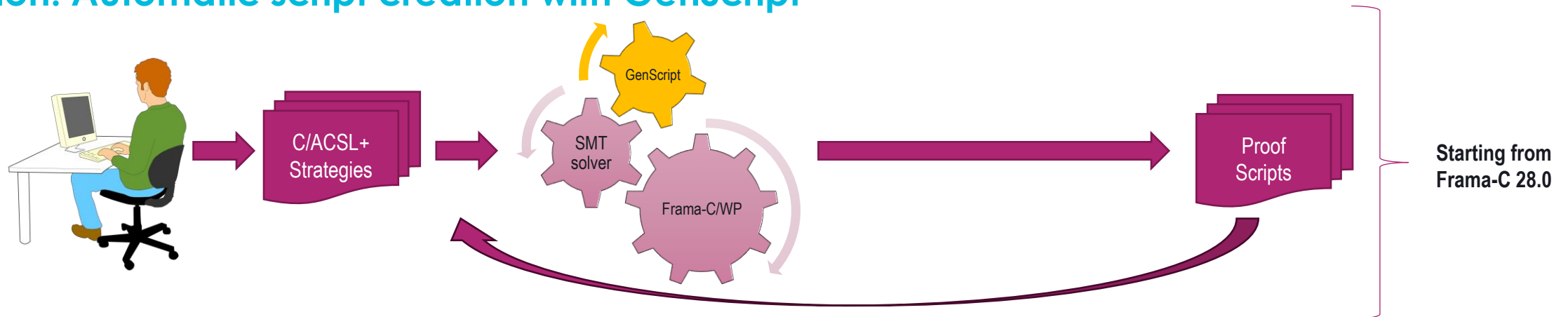


# Manual vs. Automatic Proof Script Creation

## Motivation: manual proof script creation is hard



## Solution: Automatic script creation with GenScript



## Working Example: Assume we have a complex goal

```
/*@ meta \prop, \name(confidentiality), \targets(\ALL), \context(\reading), \separated(\read, &secret[0..9]); */
```

```
char secret[10];
```

```
/*@ requires n ≥ 0 ∧ \valid(t + (0..n-1));
```

```
terminates \true;
```

```
exits \false;
```

```
ensures \result != 0 ⇔ (∀ Z j; 0 ≤ j < n ⇒ t[j] == 0);
```

```
assigns \nothing;
```

```
*/
```

```
int all_zeros(int *t, int n) {
```

```
int k=0;
```

```
/*@ loop invariant 0 ≤ k ≤ n;
```

```
loop invariant ∀ Z j; 0 ≤ j < k ⇒ t[j] == 0;
```

```
loop assigns k;
```

```
loop variant n-k;
```

```
*/
```

```
while(k < n){
```

```
/*@ assert rte: mem_access: \valid_read(t + k); */
```

```
/*@ assert confidentiality: meta: \separated(t + k, &secret[0 .. 9]); */
```

```
if (t[k] != 0) return 0;
```

```
/*@ assert rte: signed_overflow: k + 1 ≤ 2147483647; */
```

```
k++;
```

```
}
```

```
return 1;
```

```
}
```

What is the corresponding proof context ?

# Working Example: Manually created interactive proof script with proof tactics

The screenshot displays the Frama-C GUI interface. The main window shows a proof script for a goal named 'Invariant (preserved)'. The script includes several 'Have' statements and a 'Prove' statement. A 'Tactics' panel on the right lists various tactics such as 'Alt-Ergo', 'Auto', 'Absurd', 'Clear', 'Contrapose', 'Cut', 'Filter', 'Instance', and 'Lemma'. A 'Provers...' button is also visible. The script content is as follows:

```
script '.frama-c/wp/script/all_zeros_loop_invariant_2_preserved.json' (not created)

Proof:
Goal Invariant (preserved)
Goal Instance (Instance) (proved)
Qed.

Goal Invariant (preserved):
Let a = shift_sint32(t, k).
Assume {
  Type: is_sint32_chunk(Mint_0) /\ is_sint32(k) /\ is_sint32(n) /\
  is_sint32(1 + k).
  (* Heap *)
  Type: (region(t.base) <= 0) /\ linked(Malloc_0).
  (* Goal *)
  When: (0 <= i) /\ (i <= k).
  (* Pre-condition *)
  Have: valid_rw(Malloc_0, shift_sint32(t, 0), n).
  (* Invariant *)
  Have: 0 <= n.
  (* Invariant *)
  Have: (0 <= k) /\ (k <= n).
  (* Invariant *)
  Have: forall i_I : Z. ((0 <= i_I) -> ((i_I < k) ->
    (Mint_0[shift_sint32(t, i_I)] = 0))).
  (* Then *)
  Have: k < n.
  (* Assertion 'rte_mem_access' *)
  Have: valid_rd(Malloc_0, a, 1).
  (* Else *)
  Have: Mint_0[a] = 0.
  (* Assertion 'rte_signed_overflow' *)
  Have: k <= 2147483646.
  (* Invariant *)
  Have: (-1) <= k.
}
Prove: Mint_0[shift_sint32(t, i)] = 0.

Goal id: typed_all_zeros_loop_invariant_2_preserved
Short id: all_zeros_loop_invariant_2_preserved
```

Annotations in the image point to specific parts of the script and the tactics panel:

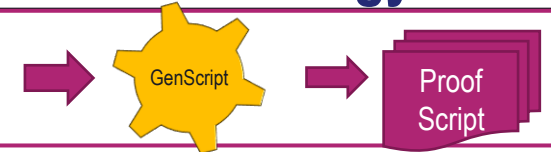
- A red box highlights the 'Proof:' section of the script, with a label 'Proof strategy saved in a Json script'.
- A red box highlights the 'Have: forall i\_I : Z. ((0 <= i\_I) -> ((i\_I < k) -> (Mint\_0[shift\_sint32(t, i\_I)] = 0))).' line, with a label 'Applied tactics in the proof strategy'.
- A red box highlights the 'Tactics' panel, with a label 'Available tactics'.
- A red box highlights the 'Instantiate properties. #1: i' entry in the tactics panel, with a label 'Proof context of current (sub-)goal'.

The command line at the bottom of the window is: `frama-c-gui file.c -wp -wp-rte`

# Example with GenScript: Automatic proof script generation from a user-provided strategy

```
/*@ strategy altergo: \prover("alt-ergo", 20); */
/*@ strategy s: \tactic("Wp.instance", \pattern((\forall integer _; _), 0 <= k <= _, 0 <= i <= k), \param("P1", i), \children(altergo)), altergo; */
/*@ proof s: z; */
/*@ requires ...
    terminates ...
    exits ...
    ensures ...
    assigns ...
*/
int all_zeros(int *t, int n) {
    int k=0;
    /*@ loop invariant ...
        loop invariant z:  $\forall Z j; 0 \leq j < k \Rightarrow t[j] == 0;$ 
        loop assigns ...
        loop variant ...
    */
    while(k < n){
        /*@ assert ...; */

        if (t[k] != 0) return 0;
        /*@ assert ...; */
        k++;
    }
    return 1;
}
```



```
frama-c file.c -wp -wp-rte -wp-prover tip -wp-script update
```

## Proof metrics over years

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

- ▶ Continuous **scope extensions**
- ▶ **Refactoring of redundant ACSL annotations** with macros for spec maintainability
- ▶ **Well designed metaproperties** not affected by scope extensions
- ▶ **Promising automation rate** of proof script generation
- ▶ Script generation time is incomparable with **numerous days of corresponding human effort**
- ▶ **Local assertions instead of Lemmas** to avoid proof efficiency issues

## Proof metrics over years

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

- ▶ Continuous **scope extensions**
- ▶ **Refactoring of redundant ACSL annotations** with macros for spec maintainability
- ▶ **Well designed metaproperties** not affected by scope extensions
- ▶ **Promising automation rate** of proof script generation
- ▶ Script generation time is incomparable with **numerous days of corresponding human effort**
- ▶ **Local assertions instead of Lemmas** to avoid proof efficiency issues

Breaking news:  
now 100% of  
proof scripts  
are generated  
automatically

# Automation of Proof Script Creation

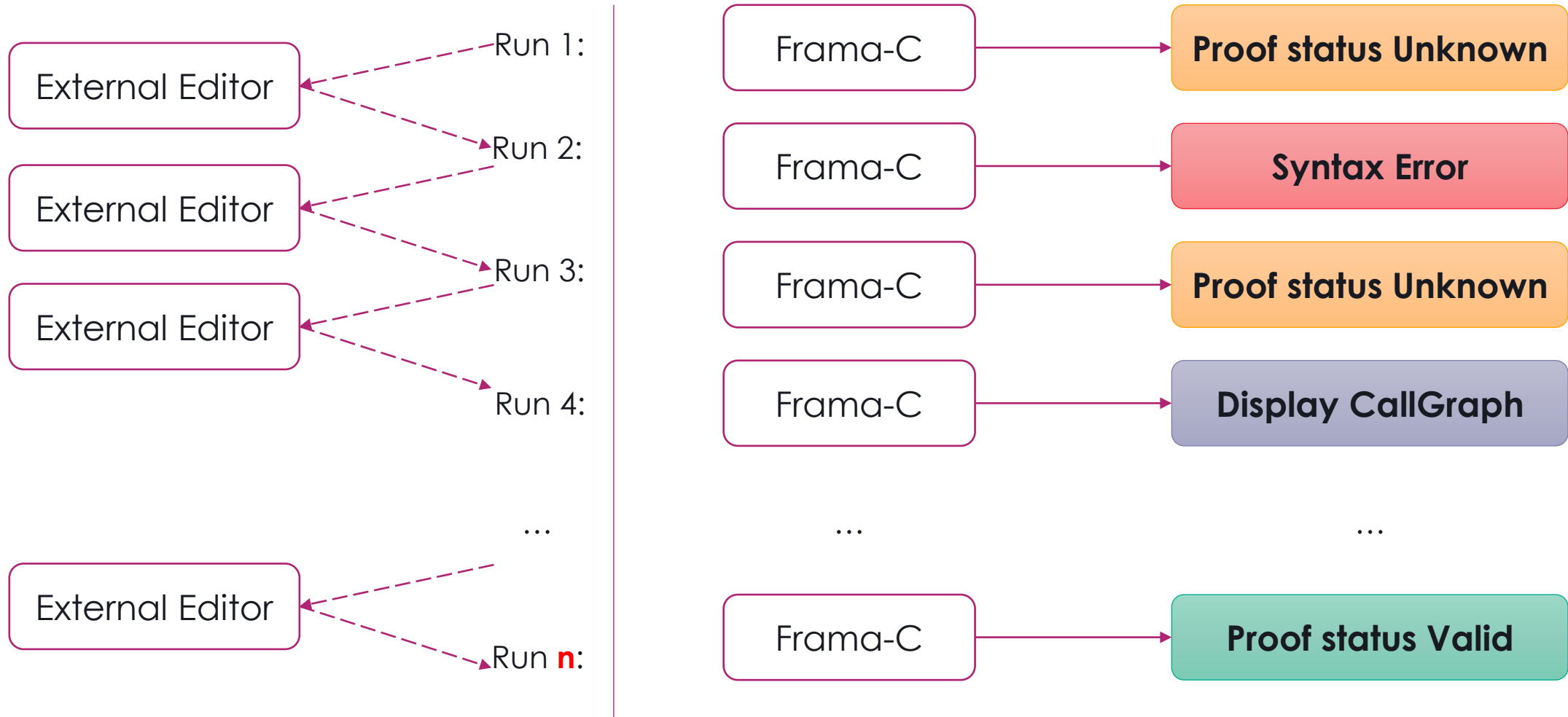
Tactic	filter	overflow	split	modmask	unfold	shift	bittestrange	cut	range	instance	ratio
All occur. 2023/fc27.1	179	238	1167	82	1323	68	6	130	2	178	4.6
Scripts 2023/fc27.1	86	128	452	35	315	43	6	63	2	105	1.7
All occur. 2025/fc30.0	535	998	2043	90	2341	215	36	104	191	210	7.2
Scripts 2025/fc30.0	219	102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

without auto scripts

with auto scripts

- ▶ **unfold** and **split** tactics are the most used in proof scripts
- ▶ **instance** tactic is relatively the most complicated to apply when relevant
- ▶ **instance** tactic will be available in next Frama-C release (33.0)
- ▶ In average, 1.7 distinct tactics are used in a proof script
- ▶ **Multiple applications of the same tactic** in a proof script occur more frequently in automatically generated scripts
- ▶ Future version of automatic script generation would allow to **better control the application of each tactic**

# Motivation: Integrated Development Environment for Frama-C/WP needed



**n is often big for complex proof attempts with WP**

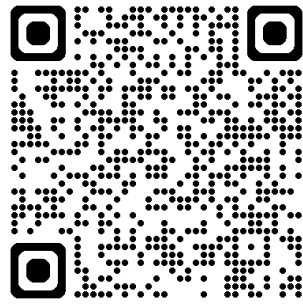
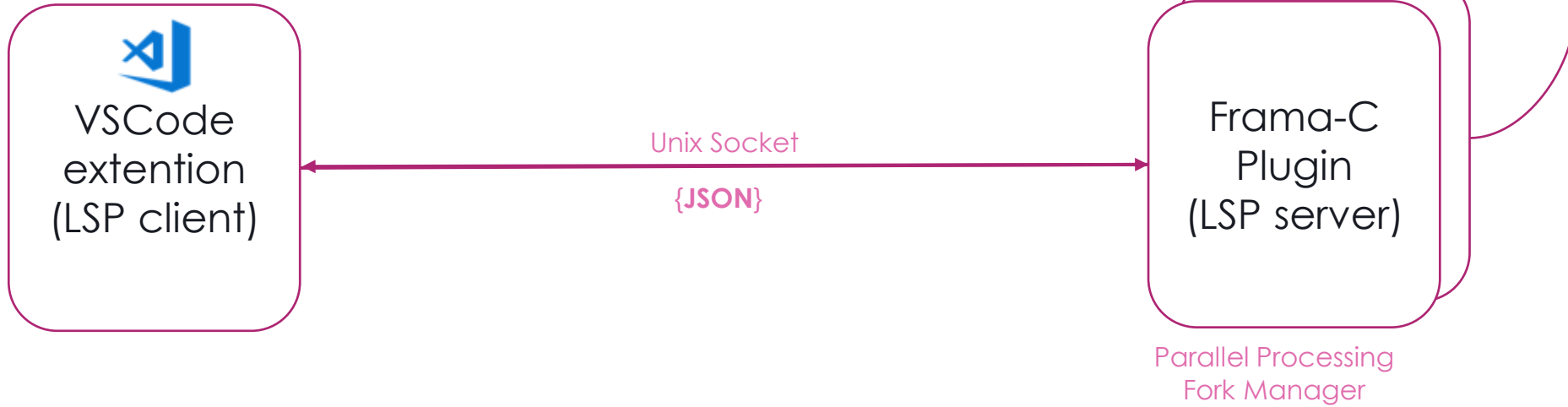
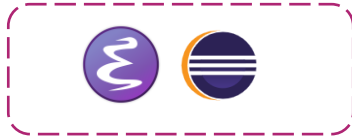
# Solution: VSCode Extension for Frama-C/WP

The screenshot displays the VSCode interface with the following components:

- Editor (Left):** Shows the source code for `test.c`. The code includes annotations for meta-properties, strategies, and proof goals. For example, line 10: `/*@ requires n >= 0 && \valid(t + (0..n-1));`
- Editor (Right):** Shows the `settings.json` file with configuration for the Frama-C extension, including `"kernel.includePaths": [],` and `"wp.prover": "script,alt-ergo",`
- Bottom Panel:** Displays the 'WP GOALS' section, listing 15 verified goals with their respective provers and execution times. The top goal is highlighted: `passed typed_all_zeros_loop_invariant_z_preserved (Qed 6ms) (Alt-Ergo 28ms)`

# LSP-based VSCode extension: Simplified workflow

Compatible LSP editors (not implemented)



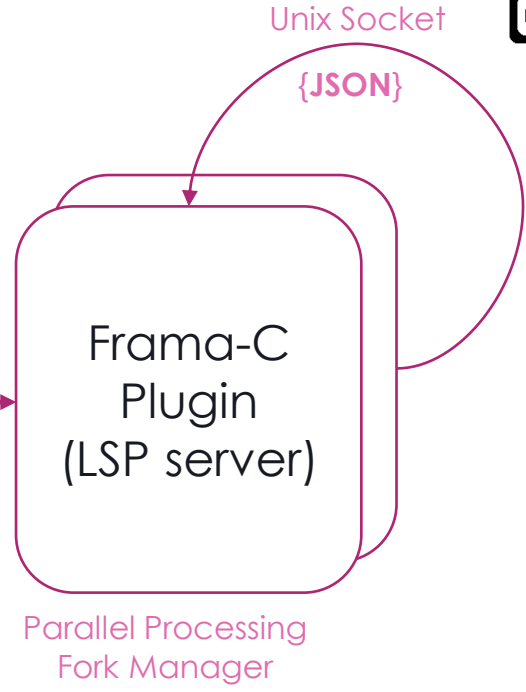
GitHub

**LSP: Language Server Protocol** <https://microsoft.github.io/language-server-protocol/specifications/lsp/3.17/specification/>

**Available in open-source** <https://github.com/ThalesGroup/frama-c-lsp>

# LSP-based VSCode extension: Simplified workflow

Compatible LSP editors (not implemented)



GitHub

Available in open-source <https://github.com/ThalesGroup/frama-c-lsp>

# Five years, five successful certifications: Main Achievements

