THALES
Building a future we can all trust

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

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)
- …

THALES

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

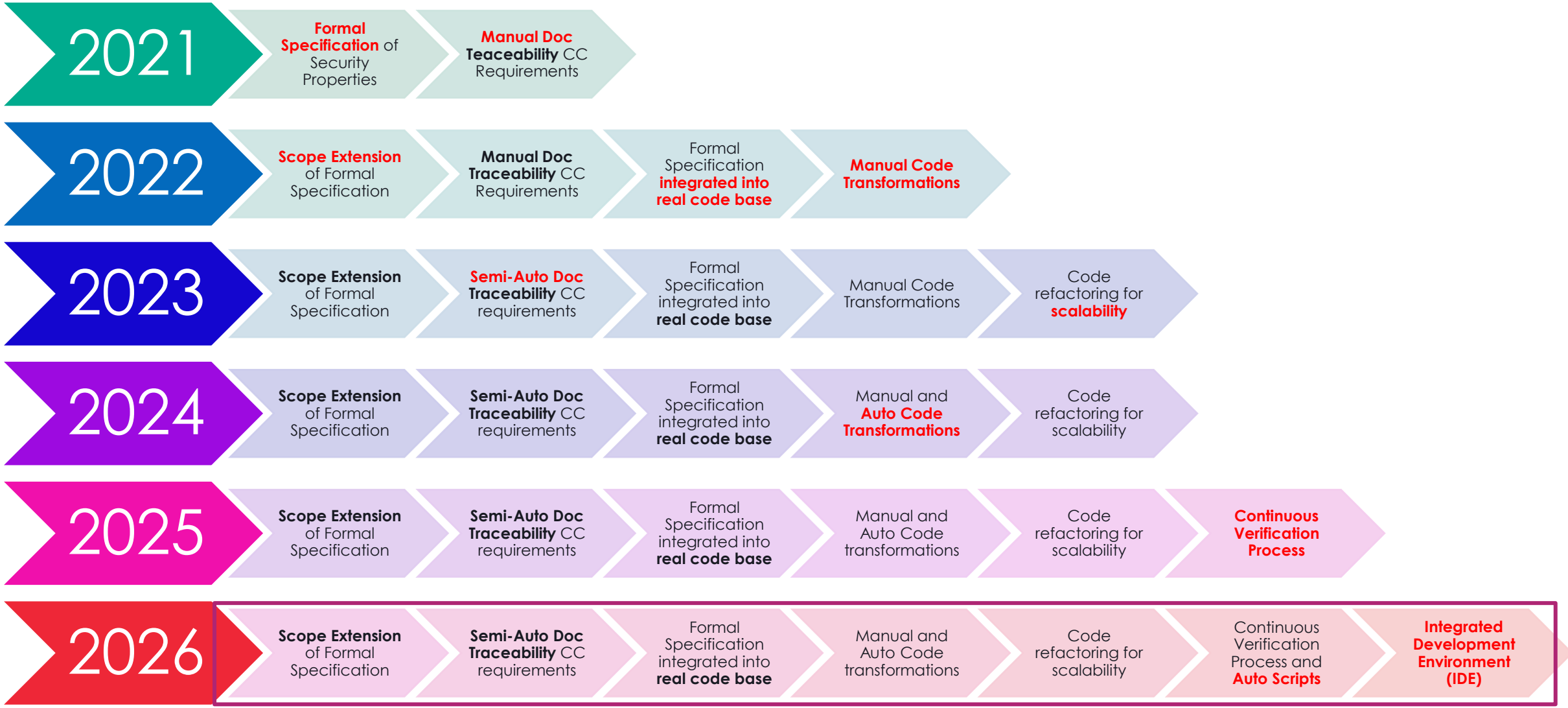
> Issues and points of attention

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

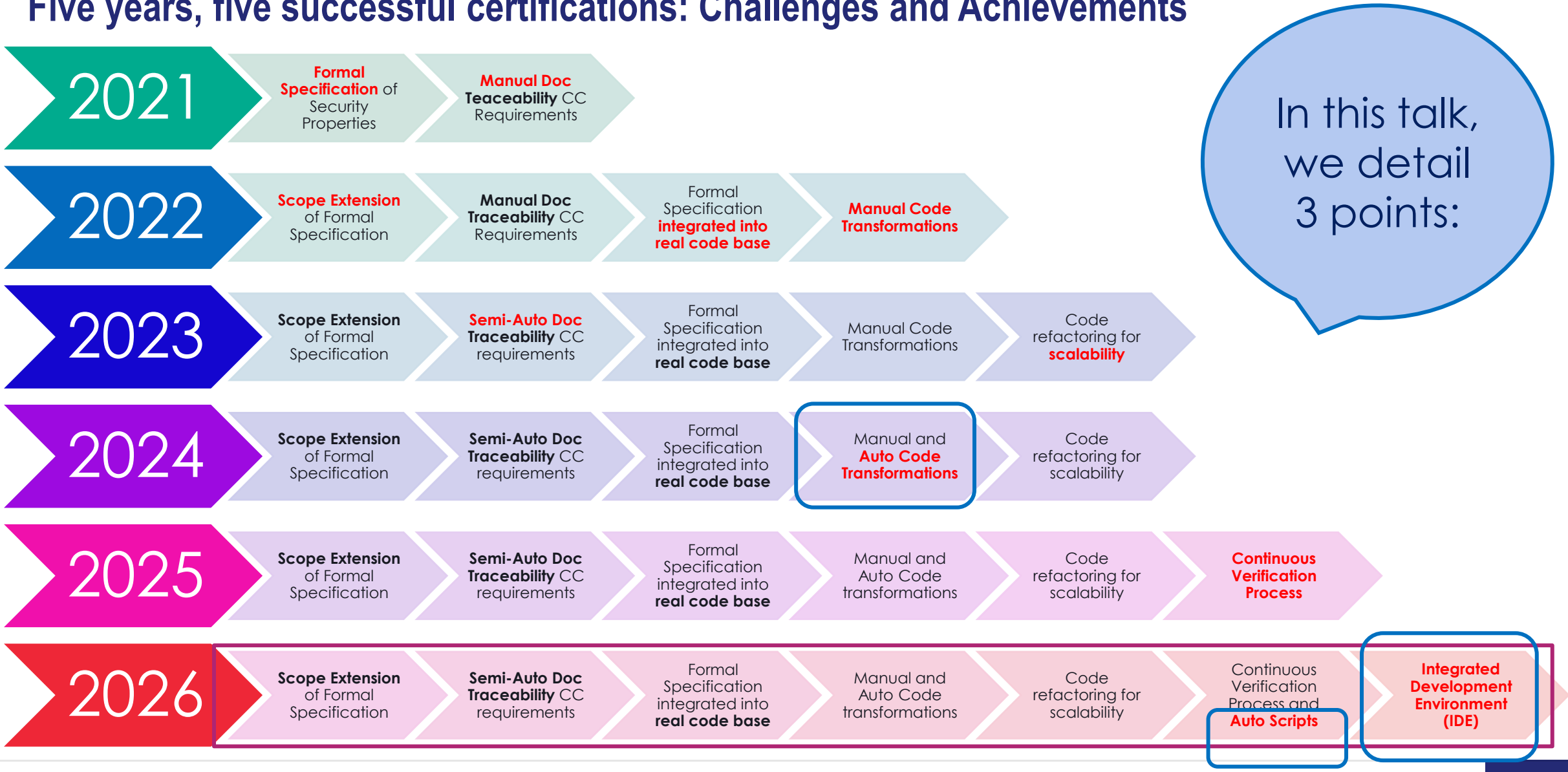
- **FM 2021:** A.Djoudi, N.Kosmatov "Formal Verification of a JavaCard Virtual Machine with Frama-C"
- **ERTS 2022:** A.Djoudi, M.Hána, N.Kosmatov, M.Kříženecký, F.Ohayon, P.Mouy, A.Fontaine and D.Féliot "A Bottom-Up Formal Verification Approach for Common Criteria Certification: Application to JavaCard 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 MAVE1.2	
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



In this talk, we detail 3 points:

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;
*/
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){


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

     k++;
    }
    return 1;
}
```

function contract in ACSL

loop contract in ACSL

```
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];

/*@ 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;
}
```


Global security property


function contract in ACSL


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

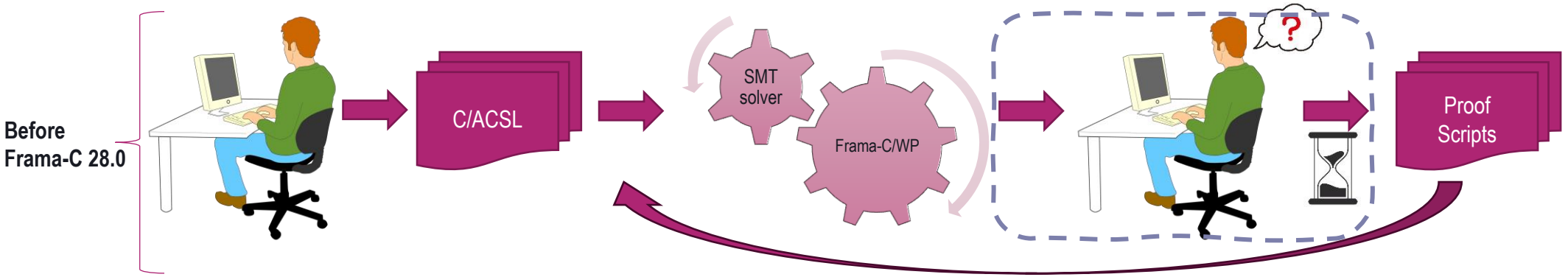
> Solution: Automatic code transformations for heterogeneous pointer casts with Uncast

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

> Uncast transformations are formally proved to be correct

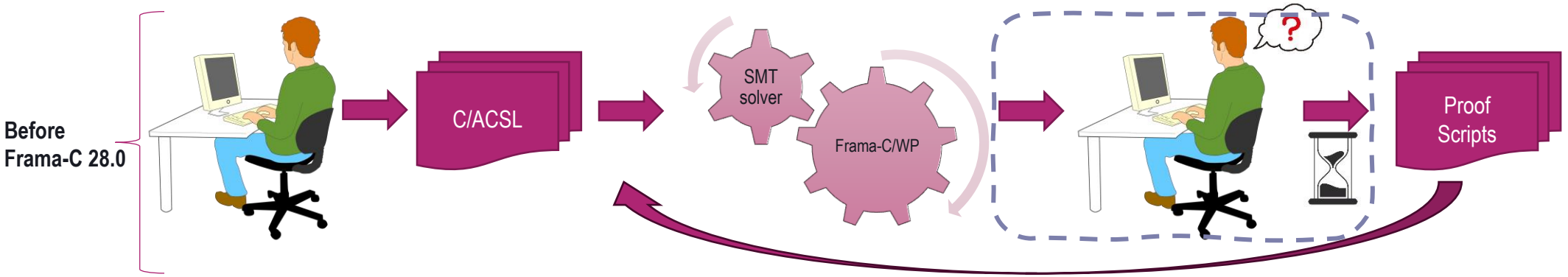
Manual vs. Automatic Proof Script Creation

Motivation: manual proof script creation is hard

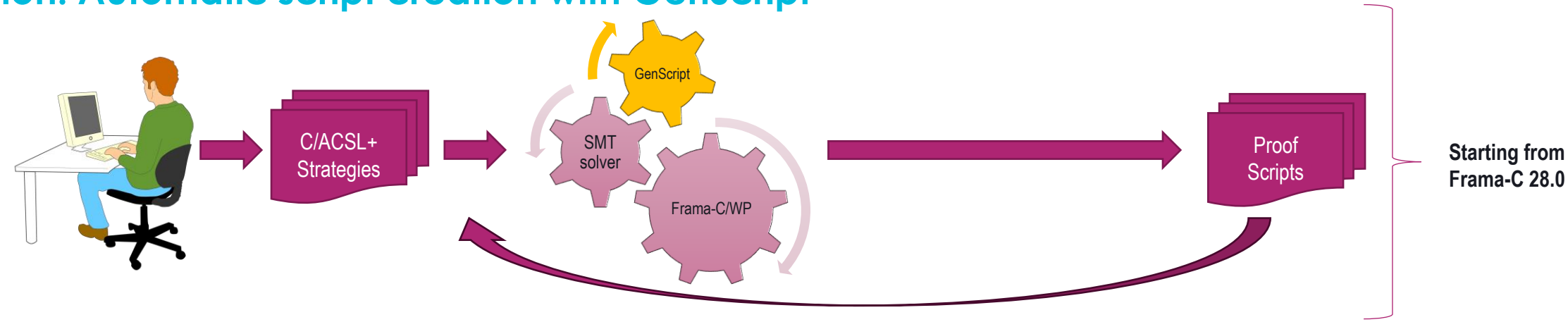


Manual vs. Automatic Proof Script Creation

Motivation: manual proof script creation is hard



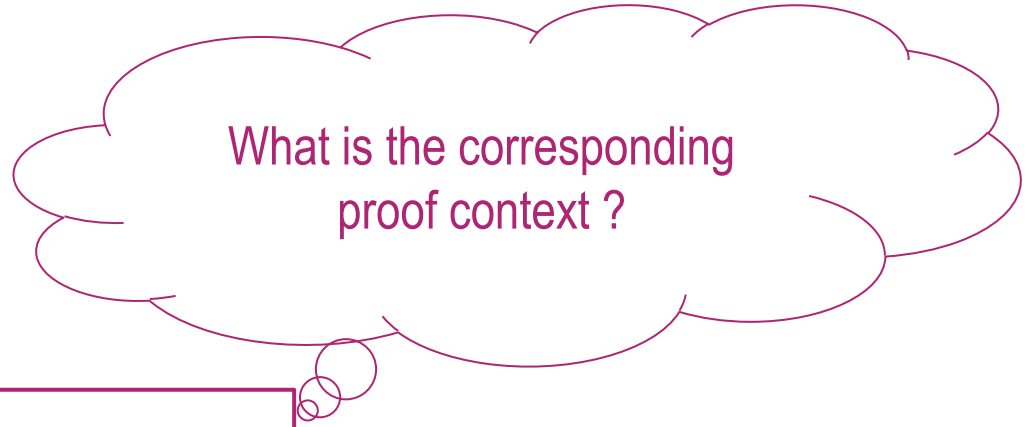
Solution: Automatic script creation with GenScript



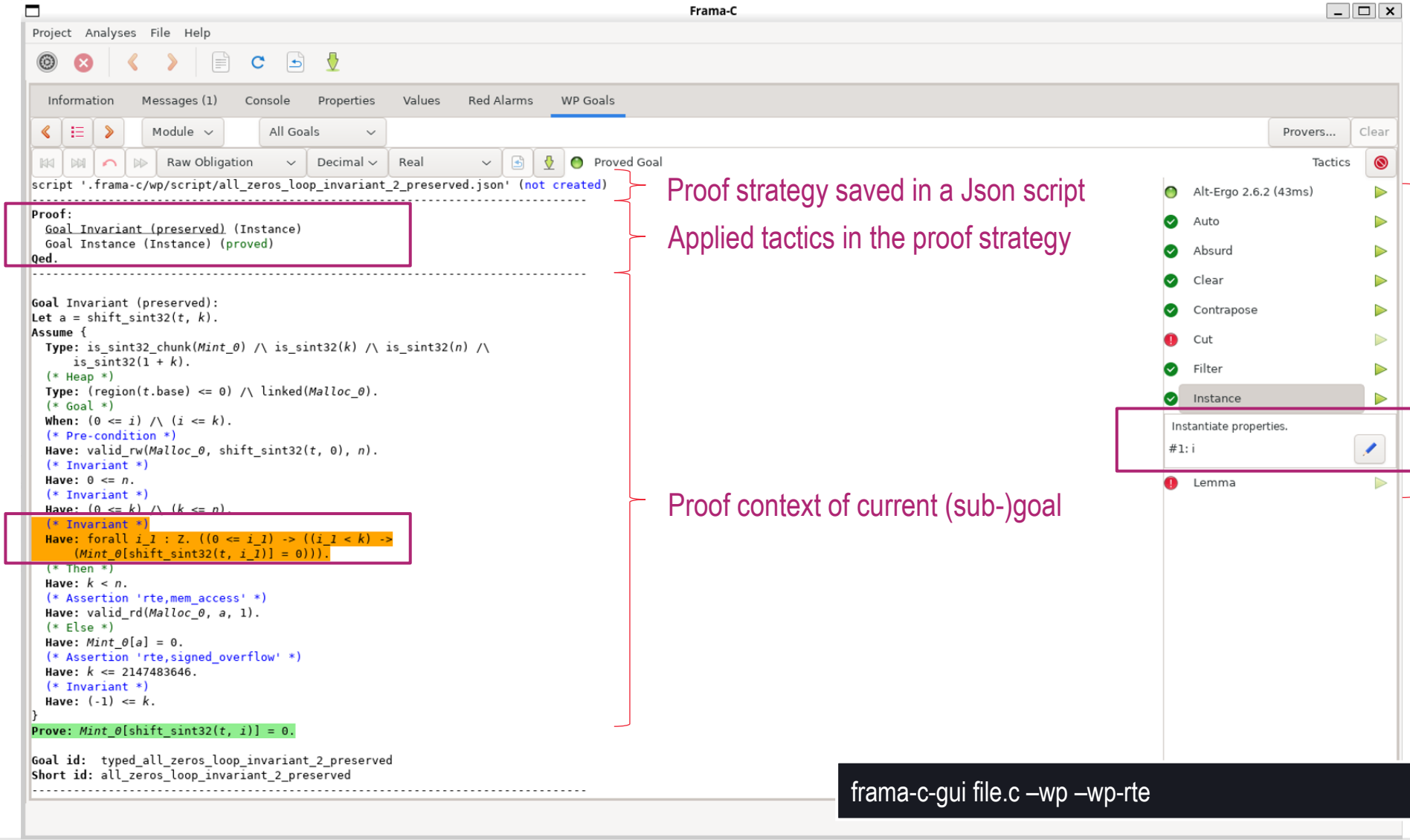
Working Example: Assume we have a complex goal

```
/*@ 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;
}
```



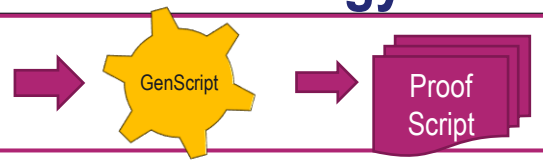
Working Example: Manually created interactive proof script with proof tactics



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: ∀ Z j; 0 ≤ j < k ⇒ 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

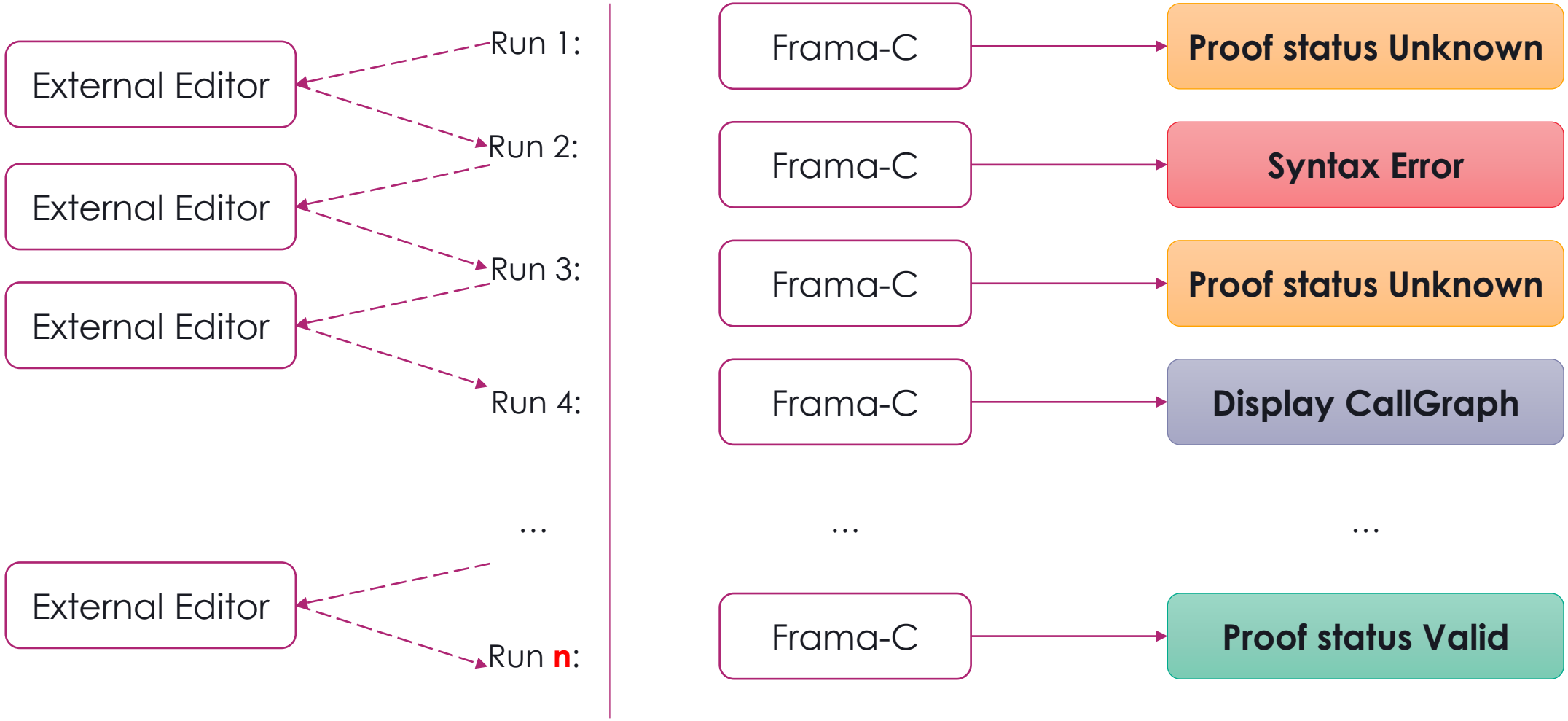
Breaking news: now 100% of proof scripts are generated automatically

Automation of Proof Script Creation

Tactic	filter	overflow	split	modmask	unfold	shift	bittestrange	cut	range	instance	ratio	
All occurr. 2023/fc27.1	179	238	1167	82	1323	68	6	130	2	178	4.6	without auto scripts
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	with auto scripts
Scripts 2025/fc30.0	219	102	531	48	367	58	8	72	63	124	1.7	
Auto level 2025/fc30.0												

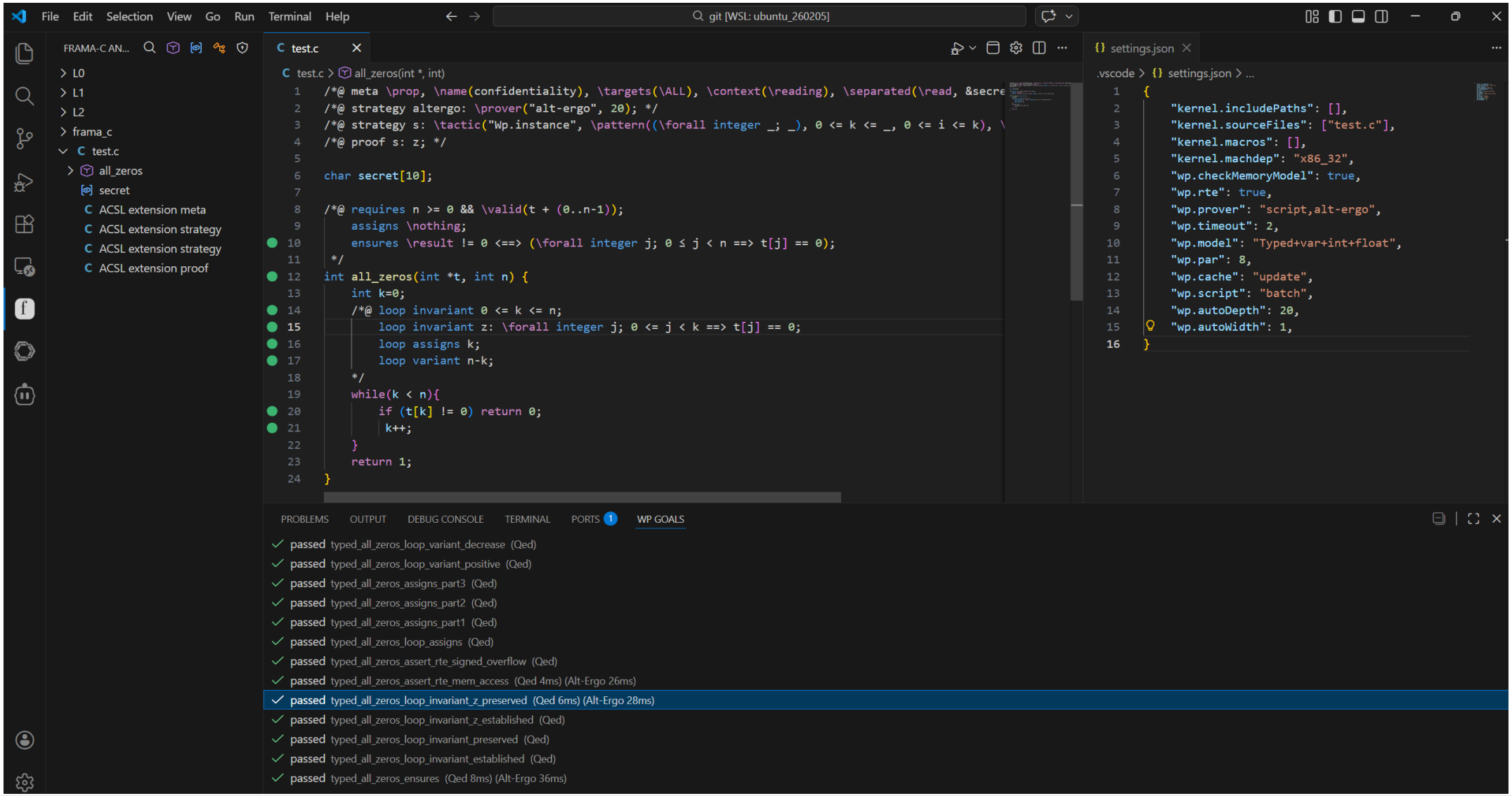
- **unfold** and **split** tactics are the most used in proof scripts
- **instance** tactic is relatively the most complicated to apply when relevant
- **instance** tactic will be available in next Frama-C release (33.0)
- In average, 1.7 tactics distinct tactics are used in a proof script
- **Multiple 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

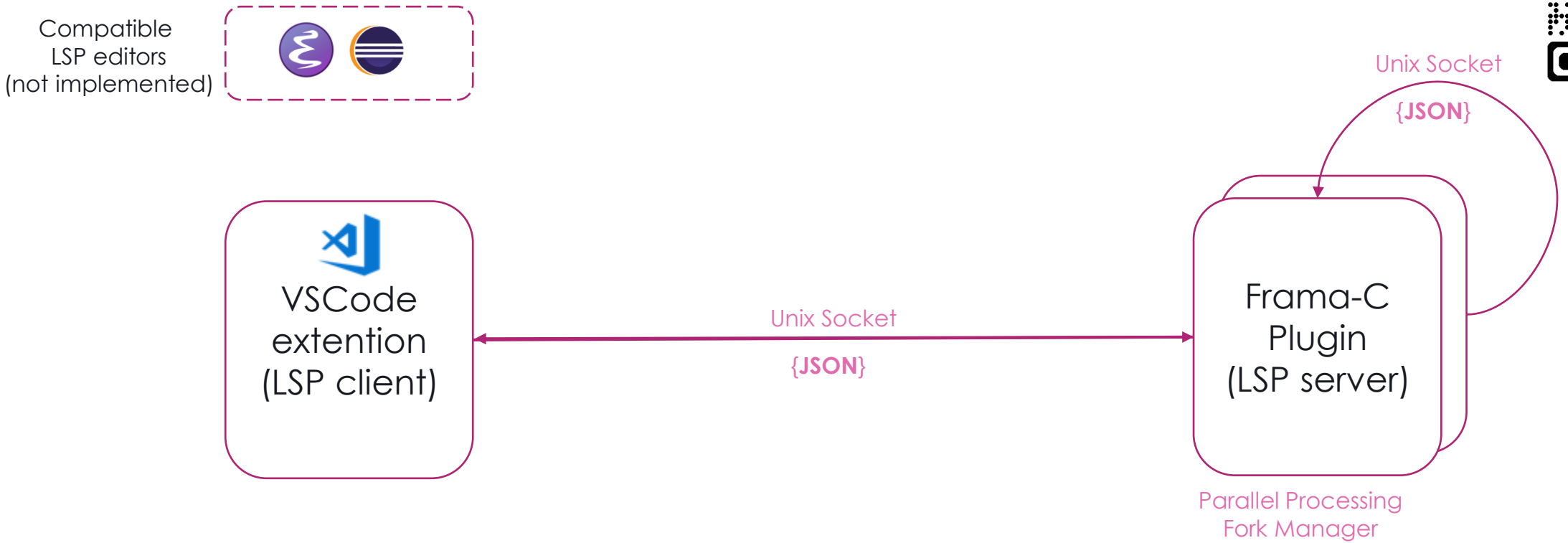


n is often big for complex proof attempts with WP

Solution: VSCode Extension for Frama-C/WP



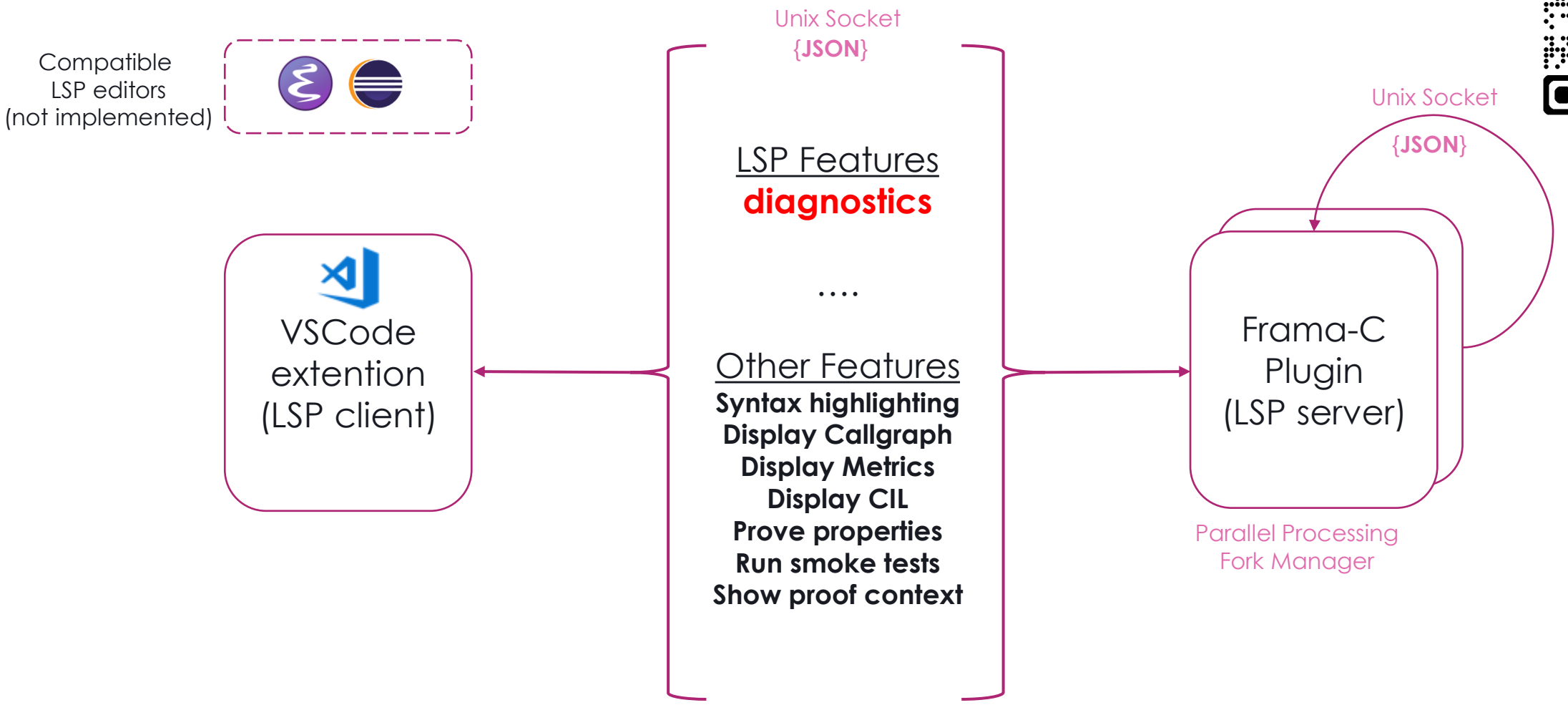
LSP-based VSCode extension: Simplified workflow



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

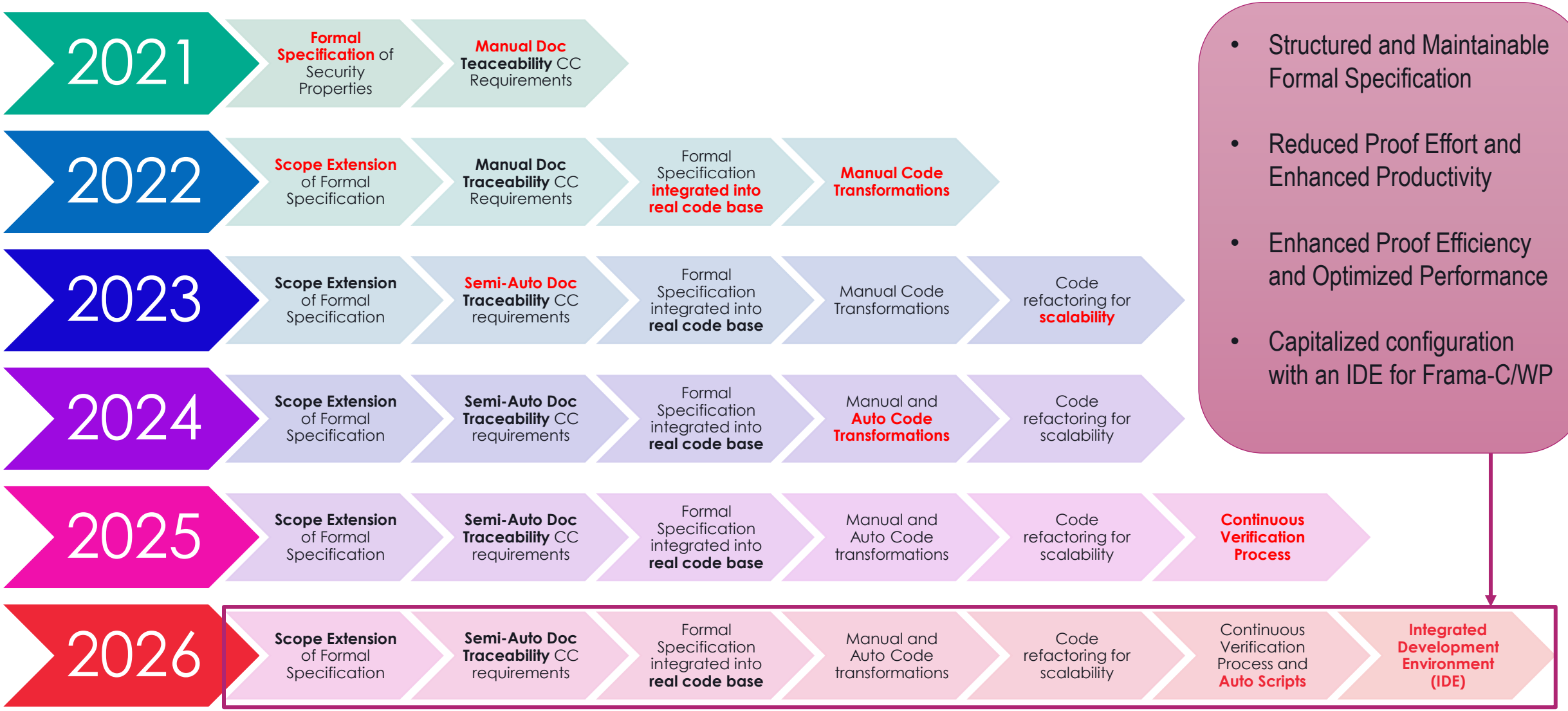
Available in open-source https://github.com/ThalesGroup/frama-c-lsp

LSP-based VSCode extension: Simplified workflow



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

Five years, five successful certifications: Main Achievements



- Structured and Maintainable Formal Specification
- Reduced Proof Effort and Enhanced Productivity
- Enhanced Proof Efficiency and Optimized Performance
- Capitalized configuration with an IDE for Frama-C/WP