

FORMAL VERIFICATION OF AN UAV AUTOPILOT

STATIC ANALYSIS AND VERIFIED CODE GENERATION

Baptiste POLLIEN¹

Thesis supervised by
Christophe GARION¹, Gautier HATTENBERGER², Pierre Roux³ and Xavier THIRIOUX¹

November 16, 2023

Critical systems

Systems which must be **highly reliable** and where any bugs can be **costly** or **life-endangering**.

They can be found in several domains:



Space



Avionics



Automotive



Medical



Nuclear



Autonomous Drone

INTRODUCTION

DEVELOPMENT OF A SYSTEM: 3 MAIN STEPS

Specification

Definition of:

- functional needs,
- guarantees required (safety and security),
- constraints (time, consumption...).

INTRODUCTION

DEVELOPMENT OF A SYSTEM: 3 MAIN STEPS

Specification

Definition of:

- functional needs,
- guarantees required (safety and security),
- constraints (time, consumption...).

Implementation

Refinement of the specification into a system.

INTRODUCTION

DEVELOPMENT OF A SYSTEM: 3 MAIN STEPS

Specification

Definition of:

- functional needs,
- guarantees required (safety and security),
- constraints (time, consumption...).

Implementation

Refinement of the specification into a system.

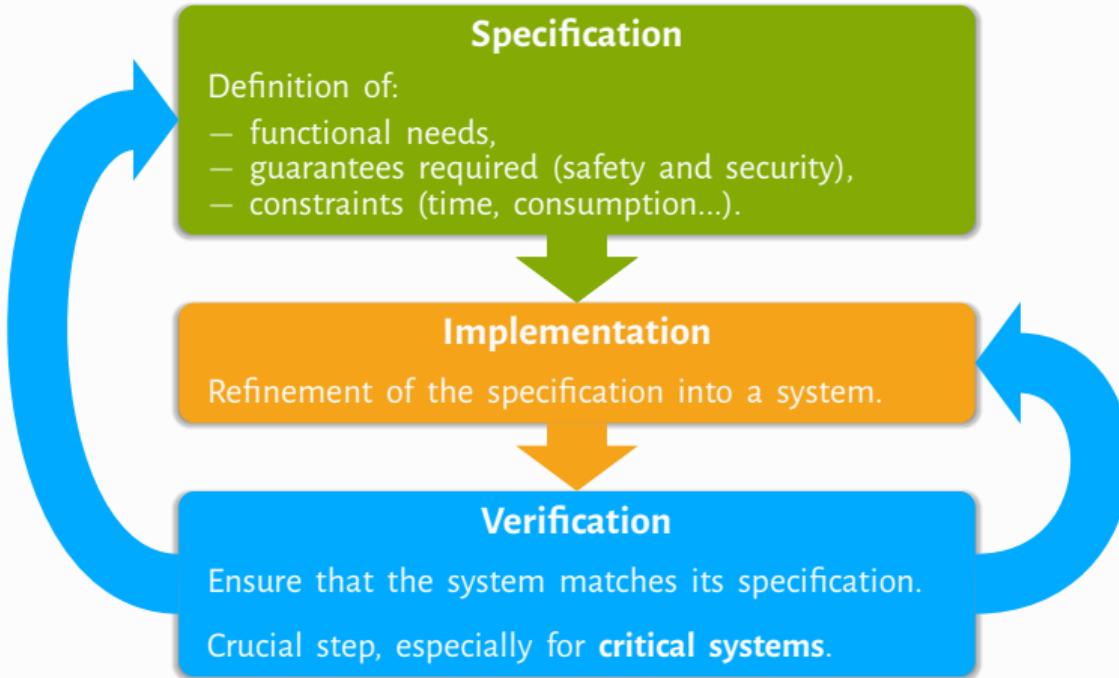
Verification

Ensure that the system matches its specification.

Crucial step, especially for **critical systems**.

INTRODUCTION

DEVELOPMENT OF A SYSTEM: 3 MAIN STEPS



Traditional verification and validation techniques:

- ▶ Code review,
- ▶ Tests.

Traditional verification and validation techniques:

- ▶ Code review,
- ▶ Tests.

“Program testing can be a very effective way to show the presence of bugs, but it is hopelessly inadequate for showing their absence.”, Edsger Dijkstra.

Traditional verification and validation techniques:

- ▶ Code review,
- ▶ Tests.

"Program testing can be a very effective way to show the presence of bugs, but it is hopelessly inadequate for showing their absence.", Edsger Dijkstra.

How to be more confident in the absence of errors?

Traditional verification and validation techniques:

- ▶ Code review,
- ▶ Tests.

“Program testing can be a very effective way to show the presence of bugs, but it is hopelessly inadequate for showing their absence.”, Edsger Dijkstra.

How to be more confident in the absence of errors?

⇒ A solution is to use **Formal Methods**.

Formal methods

- ▶ Verification techniques and tools based on **mathematical models and proofs**,
- ▶ Offer **stronger guarantees** than test.
- ▶ Examples: abstract interpretation, deductive methods, model-checking.

Formal methods

- ▶ Verification techniques and tools based on **mathematical models and proofs**,
- ▶ Offer **stronger guarantees** than test.
- ▶ Examples: abstract interpretation, deductive methods, model-checking.

Industrial use:

- ▶ Used in several domains: aerospace, automotive, medical, cybersecurity, etc.
- ▶ Recommended in avionics with DO-178C and DO-333 standards.

Formal methods

- ▶ Verification techniques and tools based on **mathematical models and proofs**,
- ▶ Offer **stronger guarantees** than test.
- ▶ Examples: abstract interpretation, deductive methods, model-checking.

Industrial use:

- ▶ Used in several domains: aerospace, automotive, medical, cybersecurity, etc.
- ▶ Recommended in avionics with DO-178C and DO-333 standards.

Limitations

- ▶ Verification tools not always scalable on large projects,
- ▶ Applied by engineers not trained in formal methods.

FORMAL VERIFICATION OF AN UAV AUTOPILOT

STATIC ANALYSIS AND VERIFIED CODE GENERATION

Goals of this thesis:

- ▶ Review verification processes using formal tools,
- ▶ Apply them on critical components,
- ▶ Ensure that these processes can be used on existing projects.

FORMAL VERIFICATION OF AN UAV AUTOPILOT

STATIC ANALYSIS AND VERIFIED CODE GENERATION

Goals of this thesis:

- ▶ Review verification processes using formal tools,
- ▶ Apply them on critical components,
- ▶ Ensure that these processes can be used on existing projects.

Thesis realised in the context of the **Concorde Project**.

Concorde Project

- ▶ Research project supported by Defense Innovation Agency (DIA).

Goals: Propose methods for the analysis and design towards the certification of future drones systems and their operations.

⇒ Apply the verification processes on **critical components** of a **drone autopilot**.

FORMAL VERIFICATION OF AN UAV AUTOPILOT

STATIC ANALYSIS AND VERIFIED CODE GENERATION

Goals of this thesis:

- ▶ Review verification processes using formal tools,
- ▶ Apply them on critical components,
- ▶ Ensure that these processes can be used on existing projects.

Thesis realised in the context of the **Concorde Project**.

Concorde Project

- ▶ Research project supported by Defense Innovation Agency (DIA).

Goals: Propose methods for the analysis and design towards the certification of future drones systems and their operations.

⇒ Apply the verification processes on **critical components** of a **drone autopilot**.

Case study: Paparazzi UAV autopilot, developed at ENAC.

INTRODUCTION

PAPARAZZI AS A FORMAL METHOD SUBJECT OF STUDY?

Paparazzi is a good candidate for testing if formal methods are usable/efficient as

- ▶ The autopilot has been developed:
 - ▶ without verification purpose,
 - ▶ by good programmers,
 - ▶ using classic C idioms in the code (pointers, etc).
- ▶ The code base is sizable ($\sim 350,000$ lines of code).

INTRODUCTION

PAPARAZZI AS A FORMAL METHOD SUBJECT OF STUDY?

Paparazzi is a good candidate for testing if formal methods are usable/efficient as

- ▶ The autopilot has been developed:
 - ▶ without verification purpose,
 - ▶ by good programmers,
 - ▶ using classic C idioms in the code (pointers, etc).
- ▶ The code base is sizable ($\sim 350,000$ lines of code).

This thesis focuses on 2 critical components:

- ▶ A mathematical library used by the control system.
 ⇒ Verified using **static code analysis**.
- ▶ A flight plan generator producing embedded C code.
 ⇒ Verified using **code generation verification techniques**.

TABLE OF CONTENTS

Introduction

Paparazzi

Static Code analysis using Frama-C

Verified Compiler in Coq

Conclusion

TABLE OF CONTENTS

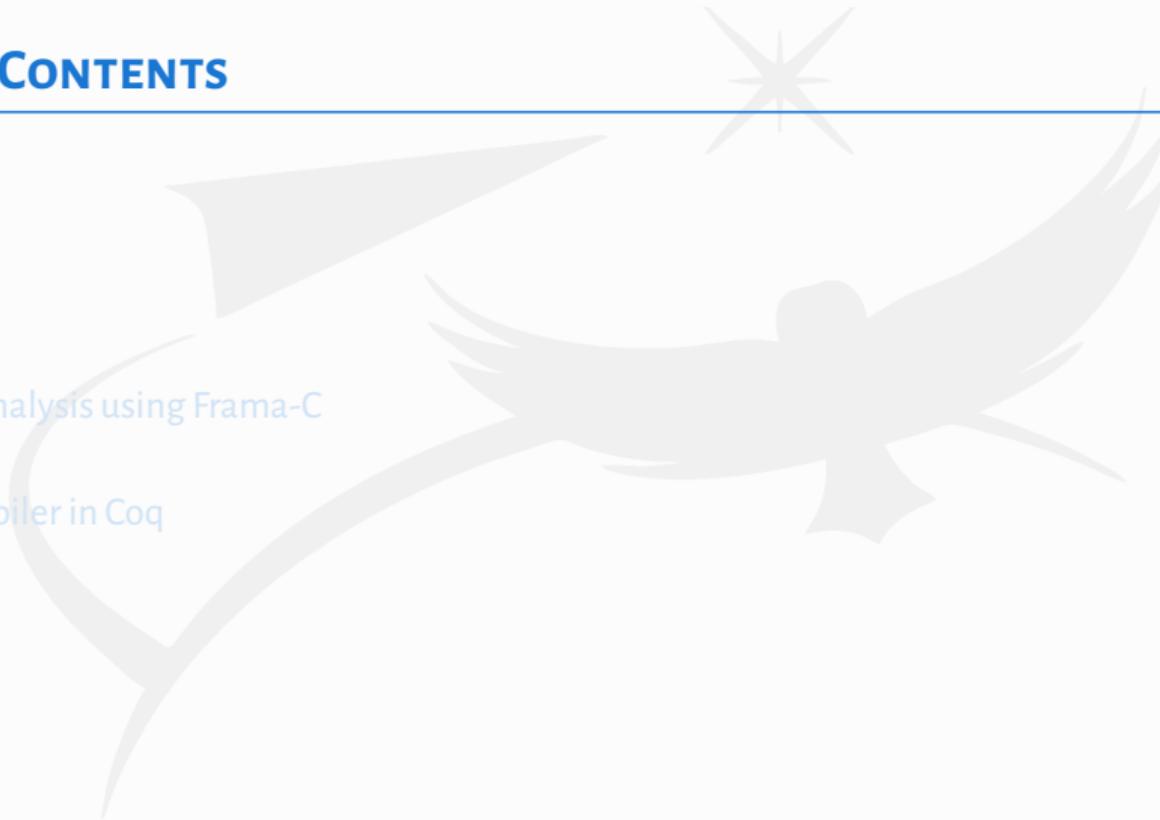
Introduction

Paparazzi

Static Code analysis using Frama-C

Verified Compiler in Coq

Conclusion



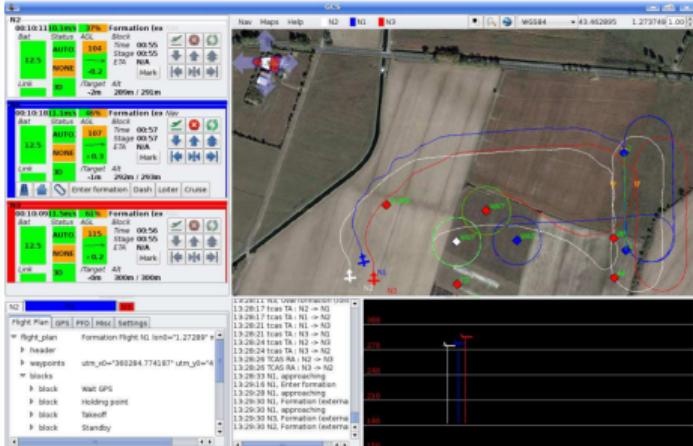


Paparazzi is an autopilot for micro-drones

- ▶ Developed at ENAC since 2003,
- ▶ Open-Source under GPL license.

Complete UAV control system:

- ▶ Control embedded software part,
- ▶ Design of some hardware components,
- ▶ Support for ground and aerial vehicles,
- ▶ Support for simultaneous control of several drones.



Paparazzi GCS connected to 3 drones¹

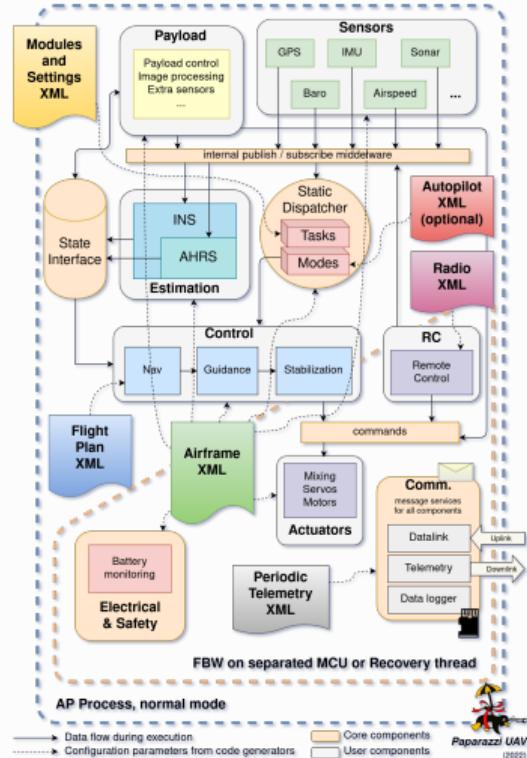
¹Pascal Brisset and Gautier Hattenberger. "Multi-UAV control with the Paparazzi system". In: HUMOUS 2008. Brest, France

PAPARAZZI

FLIGHT SYSTEM ARCHITECTURE



8

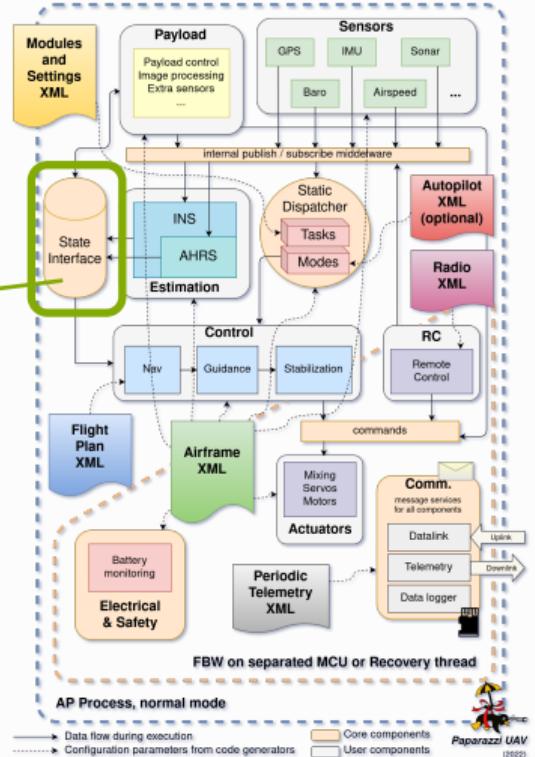




State Interface

Black board interface:

- Collects data from sensors.
- Converts automatically the data between different representations, provided by a **mathematical library**.

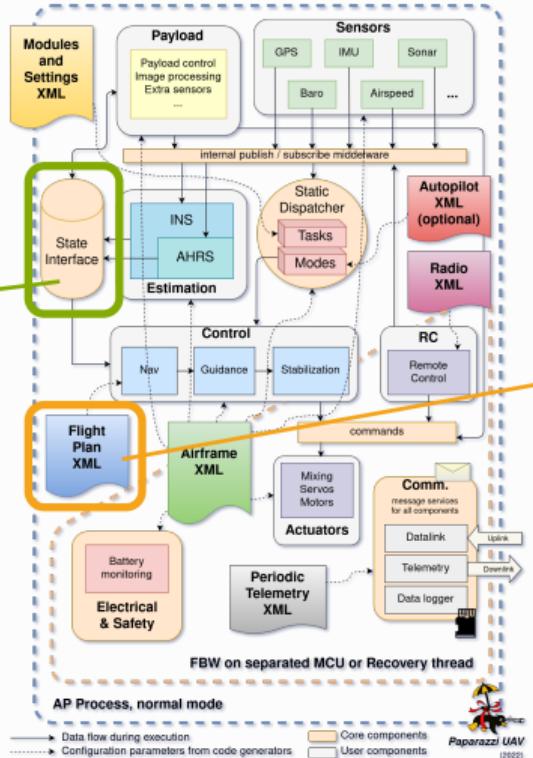




State Interface

Black board interface:

- Collects data from sensors.
- Converts automatically the data between different representations, provided by a **mathematical library**.



Flight Plan

- Defines the behaviour of the drone once launched.
- **Flight Plan Generator** that converts XML flight plans into embedded C code.

TABLE OF CONTENTS

Introduction

Paparazzi

Static Code analysis using Frama-C

Mathematical Library

Frama-C

Absence of runtime errors

Functional verification

Verified Compiler in Coq

Conclusion



MATHEMATICAL LIBRARY



pprz_algebra: mathematical algebra library coded in C ($\sim 3\ 200$ lines of code)

Library used for UAV state representations, in particular **attitude and speed representations**.



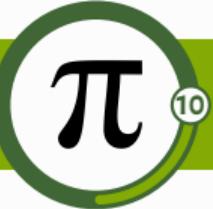
pprz_algebra: mathematical algebra library coded in C ($\sim 3\ 200$ lines of code)

Library used for UAV state representations, in particular **attitude and speed representations**.

The library contains:

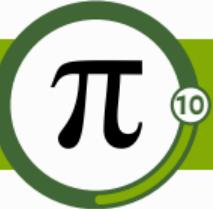
- ▶ The definition of a representation of vectors,
- ▶ Different representations of vector rotations,
rotation matrices, Euler angles, quaternions.
- ▶ Elementary operations,
ex: addition of vectors, computation of the rotation of a vector, normalisation of a quaternion, etc
- ▶ Conversion functions between these different representations.

Note: Each representation/function has a fixed point (int) and floating-point versions
(for float and double).



Data produced is used by the navigation system.

⇒ Any bug can lead to the crash of the program or produce invalid data.

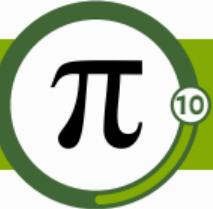


Data produced is used by the navigation system.

⇒ Any bug can lead to the crash of the program or produce invalid data.

Existing C verification tools:

- ▶ **CBMC**, a model checker for C programs.
- ▶ **VST**, a set of tools and methods for the formal verification of C software.
- ▶ **Frama-C**, a workbench implementing several verification methods for C code.
- ▶ ...



Data produced is used by the navigation system.

⇒ Any bug can lead to the crash of the program or produce invalid data.

Existing C verification tools:

- ▶ **CBMC**, a model checker for C programs.
- ▶ **VST**, a set of tools and methods for the formal verification of C software.
- ▶ **Frama-C**, a workbench implementing several verification methods for C code.
- ▶ ...

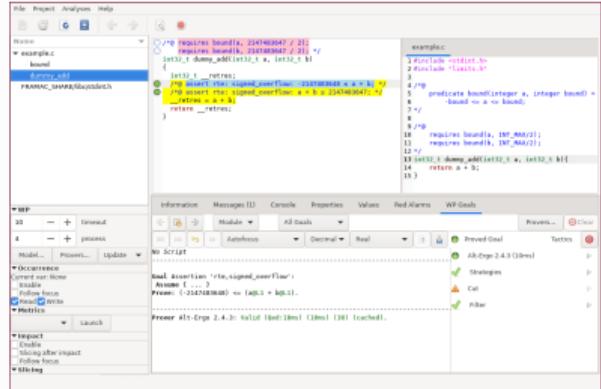
Our objective: Ensure the correctness of the library using Frama-C, **without modifying the code**.

Frama-C



Frama-C is a C code analysis tool

- ▶ Mainly developed by CEA,
 - ▶ Modular, which supports different analysis methods
ex: static analysis with EVA or dynamic analysis with E-ACSL.



Frama-C GUI

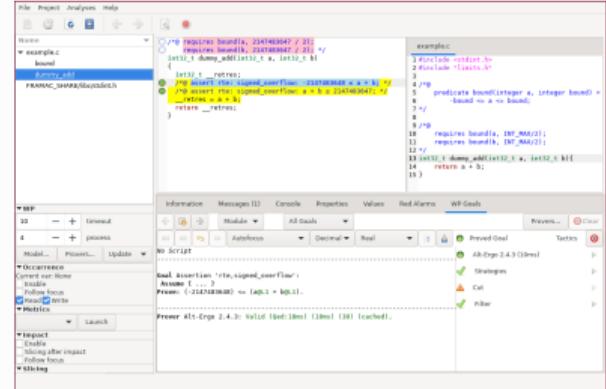
Frama-C is a C code analysis tool

- ▶ Mainly developed by CEA,
- ▶ Modular, which supports different analysis methods
ex: static analysis with EVA or dynamic analysis with E-ACSL.

Verification process of a C program using Frama-C:

1. Code specification with **ACSL** (*ANSI C Specification Language*),
2. Generation of the abstract syntax tree of the analysed code,
3. Analysis of the tree by the plugins
⇒ Verify whether the specification is respected.

Note: the tree analysis can be performed by several plugins.



Frama-C GUI



RTE (RunTime Errors):

- ▶ Adds assertions in the code,
- ▶ Allows to verify the absence of runtime errors
ex: division by 0, overflows ...



RTE (*RunTime Errors*):

- ▶ Adds assertions in the code,
- ▶ Allows to verify the absence of runtime errors
ex: division by 0, overflows ...

WP (*Weakest Precondition*)

- ▶ Implements weakest precondition calculus,
- ▶ Interfaced with Why3 to verify goals with automatic provers (Alt-Ergo, Z3, CVC4).

RTE (*RunTime Errors*):

- ▶ Adds assertions in the code,
- ▶ Allows to verify the absence of runtime errors
ex: division by 0, overflows ...

WP (*Weakest Precondition*)

- ▶ Implements weakest precondition calculus,
- ▶ Interfaced with Why3 to verify goals with automatic provers (Alt-Ergo, Z3, CVC4).

EVA (*Evolved Value Analysis*)

- ▶ Based on static analysis by abstract interpretation methods,
- ▶ Computes domains of values for each variable in the program.



ABSENCE OF RUNTIME ERRORS



There are different types of runtime errors in C:

- ▶ Dereferencing an invalid pointer,
- ▶ Division by 0,
- ▶ Overflows,
- ▶ Non finite float value,
- ▶ ...



There are different types of runtime errors in C:

- ▶ Dereferencing an invalid pointer,
- ▶ Division by 0,
- ▶ Overflows,
- ▶ Non finite float value,
- ▶ ...

Goal: Determine the “minimal” contracts for the functions of the library in order to guarantee the absence of runtime errors.

There are different types of runtime errors in C:

- ▶ Dereferencing an invalid pointer,
- ▶ Division by 0,
- ▶ Overflows,
- ▶ Non finite float value,
- ▶ ...

Goal: Determine the “minimal” contracts for the functions of the library in order to guarantee the absence of runtime errors.

Process :

- ▶ Analyse the code with Frama-C using RTE and WP plugins.
- ▶ Deduce the missing information in contracts.

Analysis of the instruction:

```
c->x = a->x * b->x;
```

Analysis of the instruction:

```
c->x = a->x * b->x;
```

Frama-C finds 2 potential errors!



Analysis of the instruction:

```
c->x = a->x * b->x;
```

Frama-C finds 2 potential errors!

- ▶ Pointers might not be valid.

```
• /*@ assert rte: mem_access: \valid(&c->x); */  
• /*@ assert rte: mem_access: \valid_read(&a->x); */  
• /*@ assert rte: mem_access: \valid_read(&b->x); */
```



Analysis of the instruction:

```
c->x = a->x * b->x;
```

Frama-C finds 2 potential errors!

- ▶ Pointers might not be valid.

- `/*@ assert rte: mem_access: \valid(&c->x); */`
- `/*@ assert rte: mem_access: \valid_read(&a->x); */`
- `/*@ assert rte: mem_access: \valid_read(&b->x); */`

⇒ Require the validity of pointers as a precondition.

Analysis of the instruction:

```
c->x = a->x * b->x;
```

Frama-C finds 2 potential errors!

- ▶ Pointers might not be valid.

- `/*@ assert rte: mem_access: \valid(&c->x); */`
- `/*@ assert rte: mem_access: \valid_read(&a->x); */`
- `/*@ assert rte: mem_access: \valid_read(&b->x); */`

⇒ Require the validity of pointers as a precondition.

- ▶ The values are not bounded.

- `/*@ assert rte: signed_overflow: -2147483648 ≤ a->x * b->x; */`
- `/*@ assert rte: signed_overflow: a->x * b->x ≤ 2147483647; */`

⇒ Determine bounds which guarantee the absence of overflows.

EXAMPLE OF FINAL CONTRACT

THE FUNCTION INT32_QUAT_COMP



```
#define SQRT_INT_MAX4 23170 // 23170 = SQRT(INT_MAX/4)

/*@
  requires \valid(a2c);
  requires \valid_read(a2b);
  requires \valid_read(b2c);
  requires \separated(a2c, a2b) && \separated(a2c, b2c);
  requires bound_Int32Quat(a2b, SQRT_INT_MAX4);
  requires bound_Int32Quat(b2c, SQRT_INT_MAX4);
  assigns *a2c;
*/
void int32_quat_comp(struct Int32Quat *a2c,
                      struct Int32Quat *a2b,
                      struct Int32Quat *b2c)
```

EVA and WP had to be associated to verify the absence of RTE.

- ▶ WP is overloaded when accessing values by reference,
 - ▶ EVA cannot verify loop variants and invariants.
- ⇒ The same problem has been raised in the thesis of V. Todorov².

²Vassil Todorov. "Automotive embedded software design using formal methods". PhD Thesis. Université Paris-Saclay, Dec. 2020

EVA and WP had to be associated to verify the absence of RTE.

- ▶ WP is overloaded when accessing values by reference,
 - ▶ EVA cannot verify loop variants and invariants.
- ⇒ The same problem has been raised in the thesis of V. Todorov².

The **real arithmetic model** (real in the mathematical sense) has been used to verify floating-point version of the functions.

The **real** model guarantees :

- ▶ The absence of division by 0,
- ▶ The lack of dereference of invalid pointers.

But the absence of overflows and rounding errors are not verified.

²Vassil Todorov. "Automotive embedded software design using formal methods". PhD Thesis. Université Paris-Saclay, Dec. 2020



FUNCTIONAL VERIFICATION



Functional verification

Offer guarantees on the behavior or the result of a function.

Example: Functional properties for square root function

```
/*@ requires x >= 0;
   ensures \result >= 0;
   ensures \result * \result == \old(x);
   assigns \nothing;
*/
float sqrt(float x);
```



Functional verification

Offer guarantees on the behavior or the result of a function.

Example: Functional properties for square root function

```
/*@ requires x >= 0;
   ensures \result >= 0;
   ensures \result * \result == \old(x);
   assigns \nothing;
*/
float sqrt(float x);
```

Using the real model:

- ▶ Offers no functional guarantee during execution.
- ▶ Used to verify that the code is correct in a mathematical sense.

How to specify the functional properties?

π

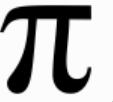
18

HOW TO SPECIFY THE FUNCTIONAL PROPERTIES?



Functional properties must be expressed in ACSL logic.

How TO SPECIFY THE FUNCTIONAL PROPERTIES?



Functional properties must be expressed in ACSL logic.

First, it is necessary to define:

- ▶ **Types,**

ex: RealVect3, RealRMat, RealQuat.

- ▶ **Elementary functions,**

ex: addition of vectors, rotation of a vector...

- ▶ **Conversion functions** between representations,

ex: Definition of the function rmat_of_quat: $\mathbb{H} \rightarrow M_{3,3}(\mathbb{R})$,

```
/*@
```

```
logic RealRMat l_RMat_of_FloatQuat(struct FloatQuat *q) =  
[...]
```

```
*/
```

Functional properties must be expressed in ACSL logic.

First, it is necessary to define:

- ▶ **Types**,
ex: RealVect3, RealRMat, RealQuat.

- ▶ **Elementary functions**,
ex: addition of vectors, rotation of a vector...

- ▶ **Conversion functions** between representations,
ex: Definition of the function rmat_of_quat: $\mathbb{H} \rightarrow M_{3,3}(\mathbb{R})$,

```
/*@  
 logic RealRMat l_RMat_of_FloatQuat(struct FloatQuat *q) =  
 [...]  
 */
```

- ▶ **Lemmas...**

Lemmas: Verify that the mathematical definitions are correct.

Ex: The conversion function produces the same rotation,



Lemmas: Verify that the mathematical definitions are correct.

Ex: The conversion function produces the same rotation,

- ▶ Mathematically,

$$\forall q \in \mathbb{H}, \forall v \in \mathbb{R}^3, q(0, v)q^* = (0, \text{rmat_of_quat}(q).v)$$

Lemmas: Verify that the mathematical definitions are correct.

Ex: The conversion function produces the same rotation,

- ▶ Mathematically,

$$\forall q \in \mathbb{H}, \forall v \in \mathbb{R}^3, q(0, v)q^* = (0, \text{rmat_of_quat}(q).v)$$

Finally, the functional properties are expressed in the form of predicates:

- ▶ M is a rotation matrix: $M \cdot M^t = I \wedge \det M = 1$
- ▶ ...



Example: Specification of the function float_rmat_of_quat.

```
/*@
  requires ...
  ensures rotation_matrix(l_RMat_of_FloatRMat(rm));
  ensures l_RMat_of_FloatRMat(rm) == l_RMat_of_FloatQuat(q);
*/
void float_rmat_of_quat(struct FloatRMat *rm, struct FloatQuat *q)
```



Example: Specification of the function float_rmat_of_quat.

```
/*@  
    requires ...  
    ensures rotation_matrix(l_RMat_of_FloatRMat(rm));  
    ensures l_RMat_of_FloatRMat(rm) == l_RMat_of_FloatQuat(q);  
*/  
void float_rmat_of_quat(struct FloatRMat *rm, struct FloatQuat *q)
```

Functional properties specified and verified in some float function contracts.

- ▶ Contracts and lemmas mainly verified **automatically** with solvers.
- ▶ Some lemmas had to be proven **manually** with Coq (~9% of the lemmas).
⇒ Approximately 2,600 lines of ACSL annotations and 200 lines of Coq for 3,200 lines of code.

TABLE OF CONTENTS

Introduction

Paparazzi

Static Code analysis using Frama-C

Verified Compiler in Coq

Flight Plan Generator

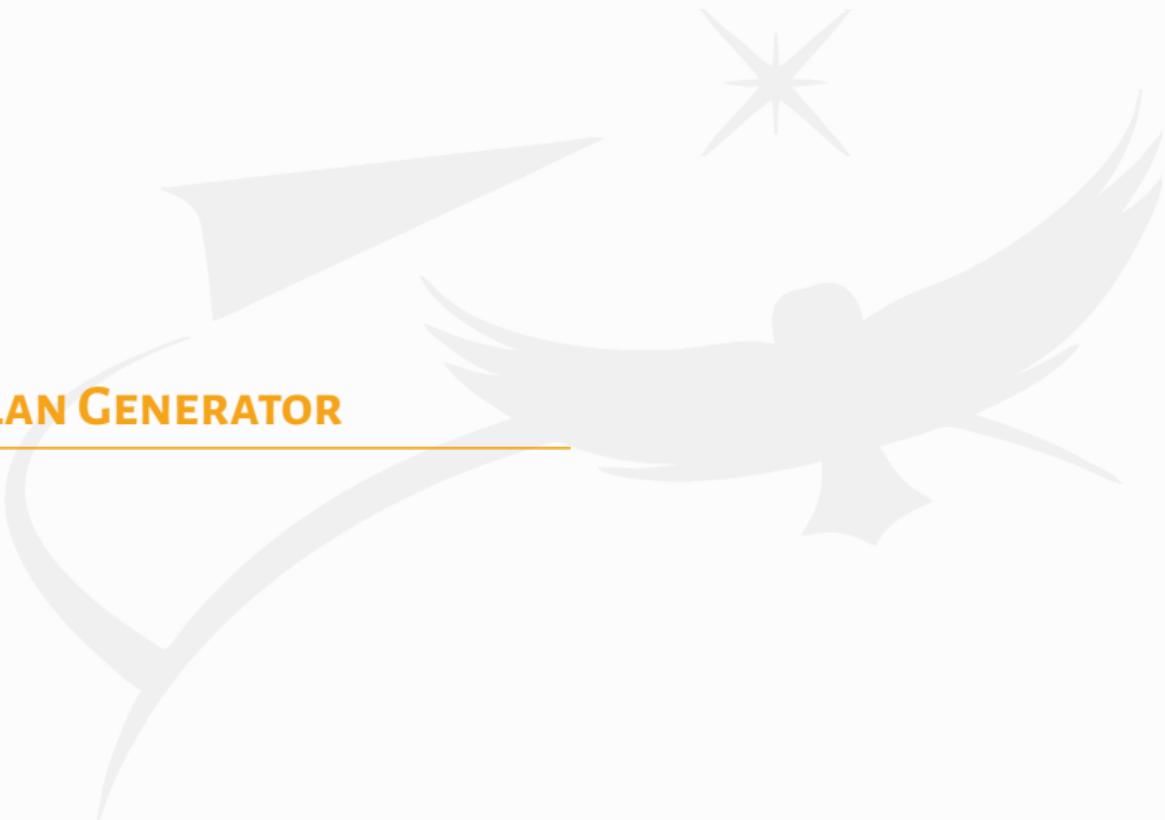
Coq

Flight Plan Language

Generator

Verification of the Generator

Conclusion



FLIGHT PLAN GENERATOR



The **flight plan** (FP)

- ▶ describes how the drone might behave when launched,
- ▶ is defined in a XML configuration file.



The **flight plan** (FP)

- ▶ describes how the drone might behave when launched,
- ▶ is defined in a XML configuration file.

Example:

1. Wait until the GPS connection is set,
2. Take off,
3. Do a circle around a specific GPS position.
4. If battery is less than 20%: Go *home* and *land*.

PRESENTATION OF THE GENERATOR

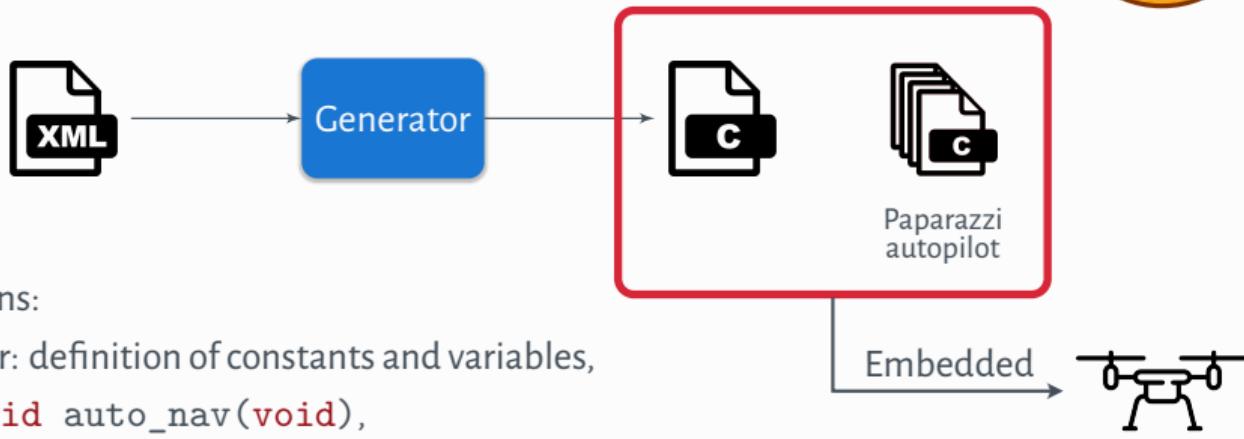




The **generated C file** contains:

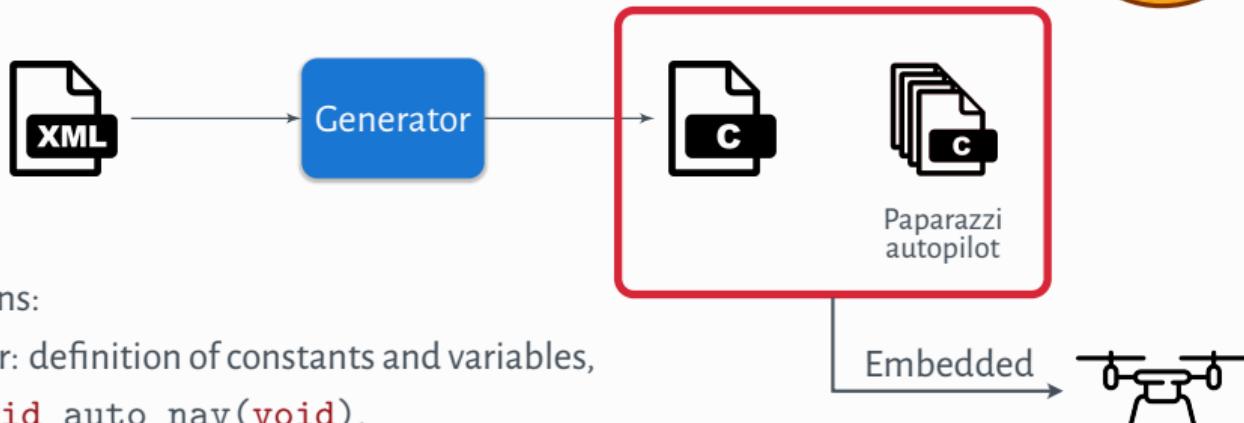
- ▶ The Flight Plan Header: definition of constants and variables,
- ▶ The main function: `void auto_nav(void)`,

PRESENTATION OF THE GENERATOR



The **generated C file** contains:

- ▶ The Flight Plan Header: definition of constants and variables,
 - ▶ The main function: `void auto_nav(void)`,
- ⇒ Compiled with the autopilot and embedded on the drone.



The **generated C file** contains:

- ▶ The Flight Plan Header: definition of constants and variables,
 - ▶ The main function: `void auto_nav(void)`,
- ⇒ Compiled with the autopilot and embedded on the drone.

Function `auto_nav`:

- ▶ Called at 20 Hz,
- ▶ Sets navigation parameters for actuators.



Problems:

- ▶ The behaviour of flight plans is not formally defined.
- ▶ Does the `auto_nav` function always terminate?
- ▶ Generator is a complex software that generates embedded code.



Problems:

- ▶ The behaviour of flight plans is not formally defined.
- ▶ Does the `auto_nav` function always terminate?
- ▶ Generator is a complex software that generates embedded code.

⇒ **Certified Compilation problem**



Problems:

- ▶ The behaviour of flight plans is not formally defined.
- ▶ Does the auto_nav function always terminate?
- ▶ Generator is a complex software that generates embedded code.

⇒ **Certified Compilation problem**

Solutions to similar problems

- ▶ CompCert: C compiler proved in Coq.
- ▶ Vélus: Lustre compiler proved in Coq.
- ▶ ...



Problems:

- ▶ The behaviour of flight plans is not formally defined.
- ▶ Does the `auto_nav` function always terminate?
- ▶ Generator is a complex software that generates embedded code.

⇒ **Certified Compilation problem**

Solutions to similar problems

- ▶ CompCert: C compiler proved in Coq.
- ▶ Vélus: Lustre compiler proved in Coq.
- ▶ ...

Our objective: Develop a new verified flight plan generator in Coq.

CoQ





Coq is a proof assistant

- ▶ Development supported by Inria,
- ▶ Based on the Gallina language.

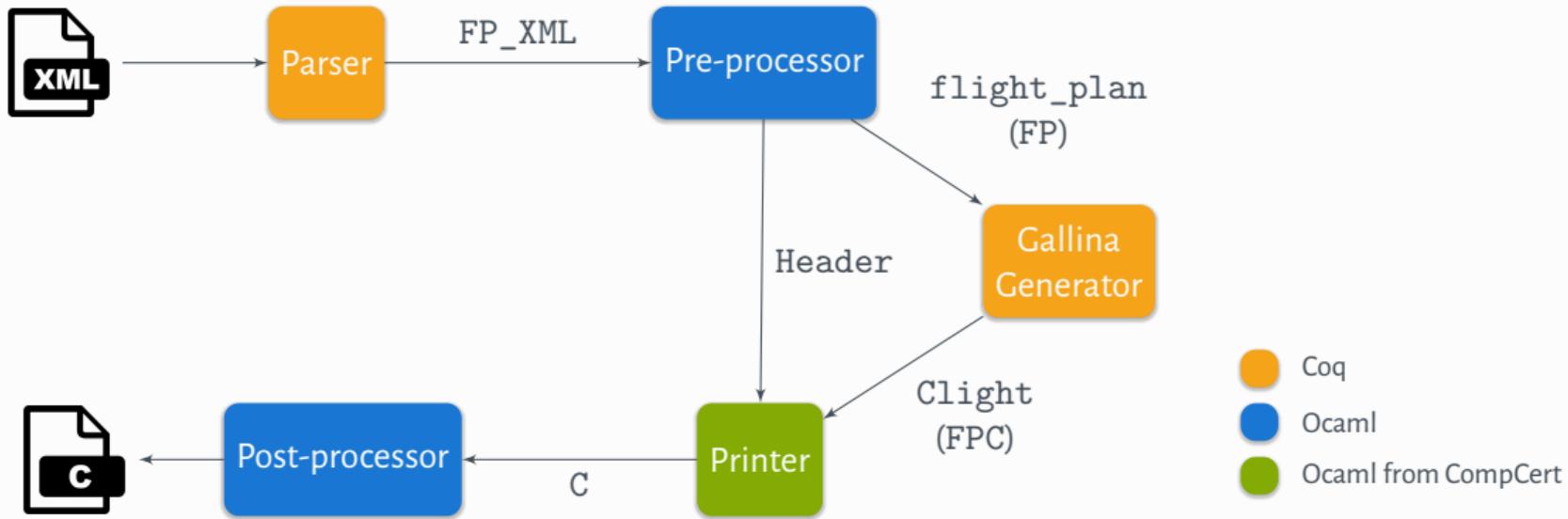
Software for writing and verifying formal proofs

- ▶ Proofs of mathematical theorems,
- ▶ Proofs of properties on programs.
⇒ Coq code can be extracted into OCaml code with guarantees.

The screenshot shows the CoqIDE interface. On the left, there is a code editor window titled "scratch" containing Gallina code for defining the power function and Fermat's Last Theorem. The right side of the interface shows a proof state with a subgoal: "1 subgoal" followed by the theorem statement "forall x y z n : nat, x^n + y^n = z^n -> n <= 2". Below the main windows, there are tabs for "Messages", "Errors", and "Jobs". A status bar at the bottom indicates "Ready, proving Fermat", "Line: 13 Char: 1", "Coq is ready", and "0 / 0".

CoqIDE, a vintage GUI for Coq

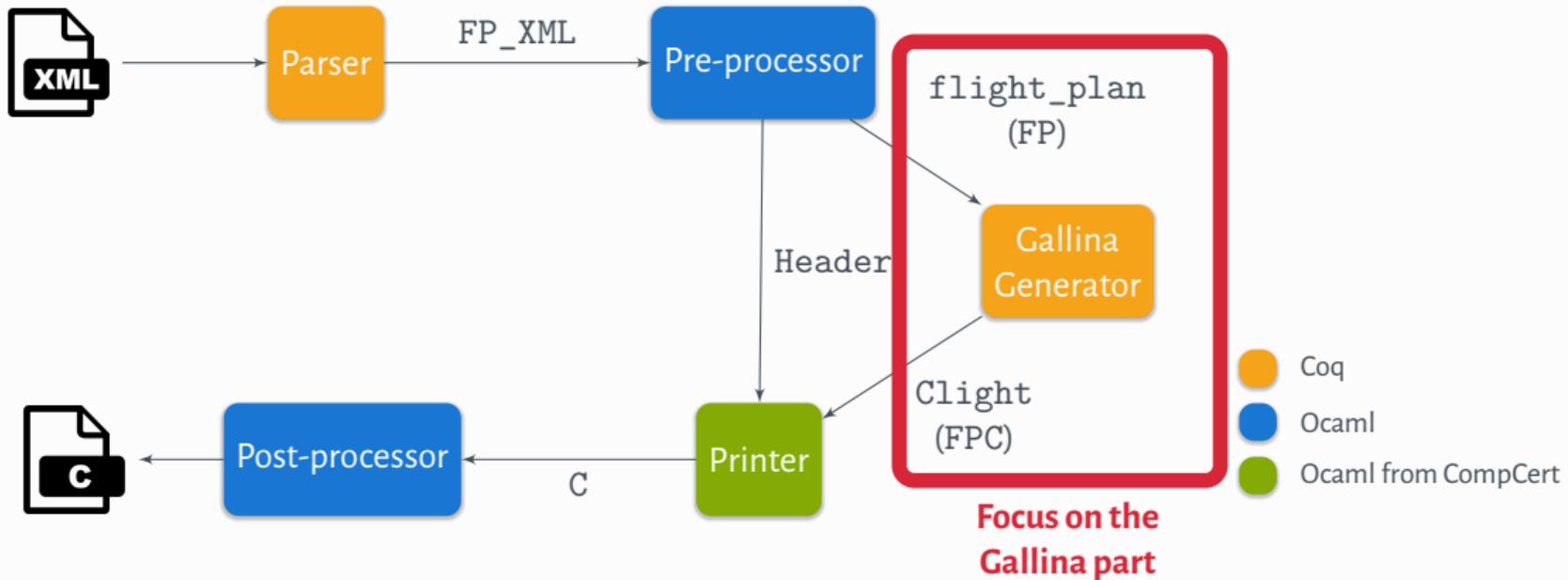
THE NEW VERIFIED FLIGHT PLAN GENERATOR (VFPG)



Pre-processing: Manages included files, converts block names into indices...

Post-processing: Produces a compilable C code.

THE NEW VERIFIED FLIGHT PLAN GENERATOR (VFPG)



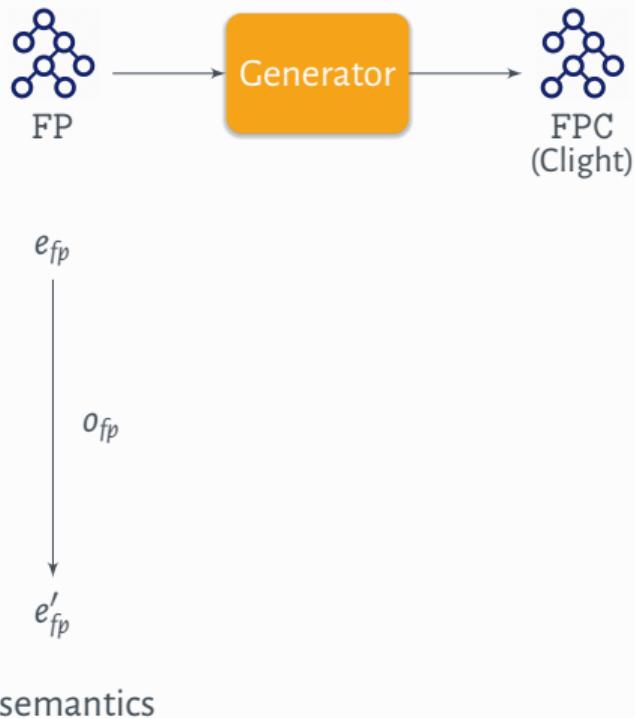
Pre-processing: Manages included files, converts block names into indices...

Post-processing: Produces a compilable C code.

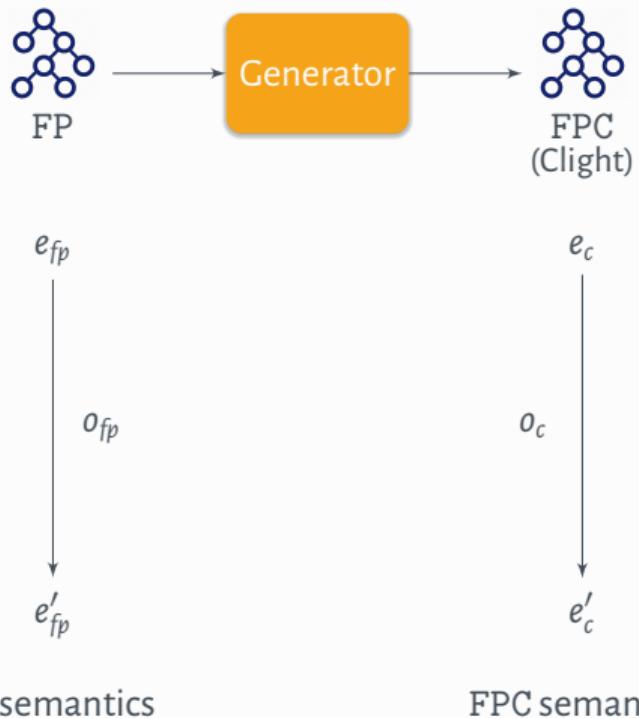
OVERVIEW OF THE SEMANTICS PRESERVATION PROOF



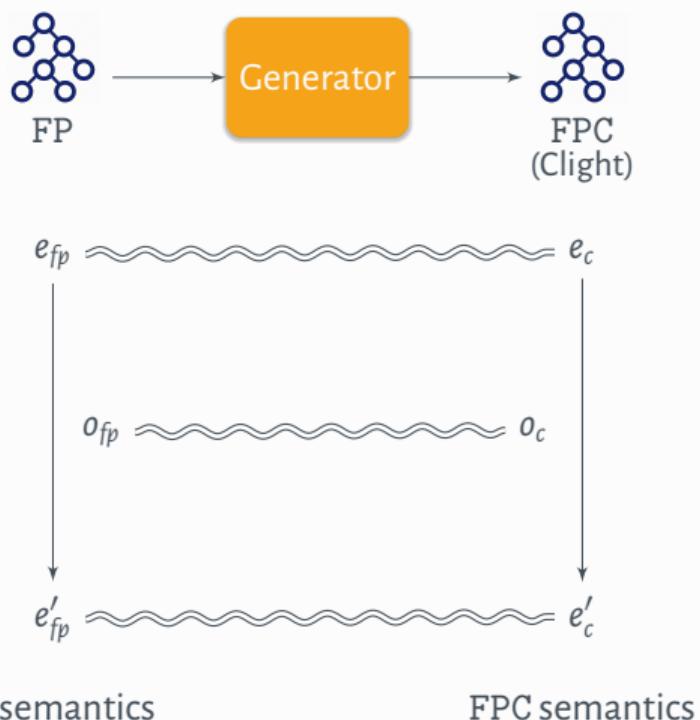
OVERVIEW OF THE SEMANTICS PRESERVATION PROOF



OVERVIEW OF THE SEMANTICS PRESERVATION PROOF



OVERVIEW OF THE SEMANTICS PRESERVATION PROOF





FLIGHT PLAN LANGUAGE

FLIGHT PLAN STRUCTURE IN GALLINA



```
Record flight_plan :={
    blocks: list fp_block
    excpts: list fp_exception;
    fb_deroutes: list fp_fb_deroute; (* New feature *)
}
```

```
Record fp_block := {
    id: block_id;
    excpts: list fp_exception;
    stages: list fp_stage;
}.
```

```
Inductive fp_stage :=
| WHILE (cond: c_cond) (body: list fp_stage)
| SET (var: var_name) (value: c_value)
| CALL (fun: c_code)
| DEROUTE (idb: block_id)
| RETURN (reset: bool)
| NAV (nav_mode: fp_nav_mode) (init: bool).
```

```
Record fp_exception :={
    cond: c_cond;
    id: block_id;
    exec: option c_code;
}.
```

```
Record fp_fb_deroute :={
    from: block_id;
    to: block_id;
    only_when: option c_cond;
}.
```

EXAMPLE: POTENTIAL EXECUTION OF A FLIGHT PLAN



28

Flight Plan:

```
{|  excpts: [ ],
    fb_deroutes: [ ],
    blocks: [
        {|  id: 0, excpts: [ ],
            stages: [
                CALL "InitSensors()";
                WHILE "!GPSFixValid()" [];
                SET "home" "GPSPosHere()"]
            |};
        {|  id: 1, excpts: [ ],
            stages: [
                NAV (TakeOff params) true;
                DEROUTE 10]
            |};
        ...  {|  id: 10, ... |} ...
    ]
}|}
```

EXAMPLE: POTENTIAL EXECUTION OF A FLIGHT PLAN



28

Flight Plan:

```
{| excpts: [ ],
  fb_deroutes: [ ],
  blocks: [
    {| id: 0, excpts: [ ],
      stages: [
        CALL "InitSensors()";
        WHILE "!GPSFixValid()" [];
        SET "home" "GPSPosHere()"]
      };
    {| id: 1, excpts: [ ],
      stages: [
        NAV (TakeOff params) true;
        DEROUTE 10]
      };
    ... {| id: 10, ... |} ...
  ]
}|}
```

Results of auto_nav:

Call	Current Block	Code Executed
1	0	InitSensors() !GPSFixValid() ↑ true

EXAMPLE: POTENTIAL EXECUTION OF A FLIGHT PLAN



28

Flight Plan:

```
{| excpts: [ ],
  fb_deroutes: [ ],
  blocks: [
    {| id: 0, excpts: [ ],
      stages: [
        CALL "InitSensors()";
        WHILE "!GPSFixValid()" [];
        SET "home" "GPSPosHere()"
      ];
    };
    {| id: 1, excpts: [ ],
      stages: [
        NAV (TakeOff params) true;
        DEROUTE 10]
    };
    ... {| id: 10, ... |} ...
  ]
}|}
```

Results of auto_nav:

Call	Current Block	Code Executed
1	0	InitSensors() !GPSFixValid() ↑ true
2	0	!GPSFixValid() ↑ true

EXAMPLE: POTENTIAL EXECUTION OF A FLIGHT PLAN



28

Flight Plan:

```
{| excpts: [ ],
  fb_deroutes: [ ],
  blocks: [
    {| id: 0, excpts: [ ],
      stages: [
        CALL "InitSensors()";
        WHILE "!GPSFixValid()" [];
        SET "home" "GPSPosHere()"]
      }];
    {| id: 1, excpts: [ ],
      stages: [
        NAV (TakeOff params) true;
        DEROUTE 10]
      ];
    ... {| id: 10, ... |} ...
  ]
}|}
```

Results of auto_nav:

Call	Current Block	Code Executed
1	0	InitSensors() !GPSFixValid() ↑ true
2	0	!GPSFixValid() ↑ true
3	0	!GPSFixValid() ↑ true
:	:	:

EXAMPLE: POTENTIAL EXECUTION OF A FLIGHT PLAN



28

Flight Plan:

```
{| excpts: [ ],
  fb_deroutes: [ ],
  blocks: [
    {| id: 0, excpts: [ ],
      stages: [
        CALL "InitSensors()";
        WHILE "!GPSFixValid()" [];
        SET "home" "GPSPosHere()"
      ];
    };
    {| id: 1, excpts: [ ],
      stages: [
        NAV (TakeOff params) true;
        DEROUTE 10]
    };
    ... {| id: 10, ... |} ...
  ]
}|}
```

Results of auto_nav:

Call	Current Block	Code Executed
1	0	InitSensors() !GPSFixValid() ↑ true
2	0	!GPSFixValid() ↑ true
3	0	!GPSFixValid() ↑ true
:	:	:
9	0	!GPSFixValid() ↑ false home = GPSPosHere()

EXAMPLE: POTENTIAL EXECUTION OF A FLIGHT PLAN



28

Flight Plan:

```
{| excpts: [ ],
  fb_deroutes: [ ],
  blocks: [
    {| id: 0, excpts: [ ],
      stages: [
        CALL "InitSensors()";
        WHILE "!GPSFixValid()" [];
        SET "home" "GPSPosHere()"
      ];
    };
    {| id: 1, excpts: [ ],
      stages: [
        NAV (TakeOff params) true;
        DEROUTE 10]
    };
    ... {| id: 10, ... |} ...
  ]
}|}
```

Results of auto_nav:

Call	Current Block	Code Executed
1	0	InitSensors() !GPSFixValid() ↑ true
2	0	!GPSFixValid() ↑ true
3	0	!GPSFixValid() ↑ true
:	:	:
9	0	!GPSFixValid() ↑ false home = GPSPosHere()
10	1	StartMotors()

EXAMPLE: POTENTIAL EXECUTION OF A FLIGHT PLAN



28

Flight Plan:

```
{| excpts: [ ],
  fb_deroutes: [ ],
  blocks: [
    {| id: 0, excpts: [ ],
      stages: [
        CALL "InitSensors()";
        WHILE "!GPSFixValid()" [];
        SET "home" "GPSPosHere()"
      ];
    };
    {| id: 1, excpts: [ ],
      stages: [
        NAV (TakeOff params) true;
        DEROUTE 10]
    };
    ... {| id: 10, ... |} ...
  ]
}|}
```

Results of auto_nav:

Call	Current Block	Code Executed
1	0	InitSensors() !GPSFixValid() ↑ true
2	0	!GPSFixValid() ↑ true
3	0	!GPSFixValid() ↑ true
:	:	:
9	0	!GPSFixValid() ↑ false home = GPSPosHere()
10	1	StartMotors()
11	1	TakeOffDone() ↑ false

EXAMPLE: POTENTIAL EXECUTION OF A FLIGHT PLAN



28

Flight Plan:

```
{| excpts: [ ],
  fb_deroutes: [ ],
  blocks: [
    {| id: 0, excpts: [ ],
      stages: [
        CALL "InitSensors()";
        WHILE "!GPSFixValid()" [];
        SET "home" "GPSPosHere()"
      ];
    };
    {| id: 1, excpts: [ ],
      stages: [
        NAV (TakeOff params) true;
        DEROUTE 10]
    };
    ... {| id: 10, ... |} ...
  ]
}|}
```

Results of auto_nav:

Call	Current Block	Code Executed
1	0	InitSensors() !GPSFixValid() ↑ true
2	0	!GPSFixValid() ↑ true
3	0	!GPSFixValid() ↑ true
:	:	:
9	0	!GPSFixValid() ↑ false home = GPSPosHere()
10	1	StartMotors()
11	1	TakeOffDone() ↑ false
12	1	TakeOffDone() ↑ false
:	:	:

EXAMPLE: POTENTIAL EXECUTION OF A FLIGHT PLAN



28

Flight Plan:

```
{| excpts: [ ],
  fb_deroutes: [ ],
  blocks: [
    {| id: 0, excpts: [ ],
      stages: [
        CALL "InitSensors()";
        WHILE "!GPSFixValid()" [];
        SET "home" "GPSPosHere()"
      ];
    };
    {| id: 1, excpts: [ ],
      stages: [
        NAV (TakeOff params) true;
        DEROUTE 10]
    };
    ... {| id: 10, ... |} ...
  ]
}|}
```

Results of auto_nav:

Call	Current Block	Code Executed
1	0	InitSensors() !GPSFixValid() ↑ true
2	0	!GPSFixValid() ↑ true
3	0	!GPSFixValid() ↑ true
:	:	:
9	0	!GPSFixValid() ↑ false home = GPSPosHere()
10	1	StartMotors()
11	1	TakeOffDone() ↑ false
12	1	TakeOffDone() ↑ false
:	:	:
20	1	TakeOffDone() ↑ true Deroute → 10

EXAMPLE: POTENTIAL EXECUTION OF A FLIGHT PLAN



28

Flight Plan:

```
{| excpts: [ ],
  fb_deroutes: [ ],
  blocks: [
    {| id: 0, excpts: [ ],
      stages: [
        CALL "InitSensors()";
        WHILE "!GPSFixValid()" [];
        SET "home" "GPSPosHere()"
      ];
    };
    {| id: 1, excpts: [ ],
      stages: [
        NAV (TakeOff params) true;
        DEROUTE 10]
    };
    ... {| id: 10, ... |} ...
  ]
}|}
```

Results of auto_nav:

Call	Current Block	Code Executed
1	0	InitSensors() !GPSFixValid() ↑ true
2	0	!GPSFixValid() ↑ true
3	0	!GPSFixValid() ↑ true
:	:	:
9	0	!GPSFixValid() ↑ false home = GPSPosHere()
10	1	StartMotors()
11	1	TakeOffDone() ↑ false
12	1	TakeOffDone() ↑ false
:	:	:
20	1	TakeOffDone() ↑ true Deroute → 10
21	10	...
:	:	:



GENERATOR



Definition `generate_flight_plan: flight_plan → res_generator`

Inputs:

- ▶ Flight plan to convert.

Outputs:

- ▶ **Variant** `res_generator :=`
 - | `CODE(res: Clight.program* list err_msg)`
 - | `ERROR(errs: list err_msg).`
- ▶ Warnings and errors currently produced during the generation.
 - ▶ detect when there is a possible deroute that is forbidden,
 - ▶ detect when the flight plan has an incorrect size.

EXAMPLE OF GENERATED C CODE



30

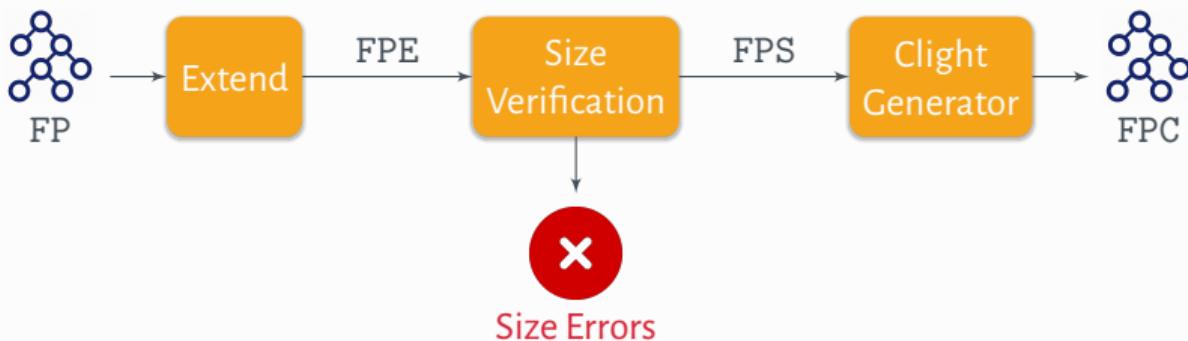
Example of a flight plan:

```
{| excpts: [],
  fb_deroutes: [],
  blocks: [
    {| id: 0,
      excpts: [],
      stages: [
        CALL "func1()";
        CALL "func2()"
      ]
    }
  ]
}
```

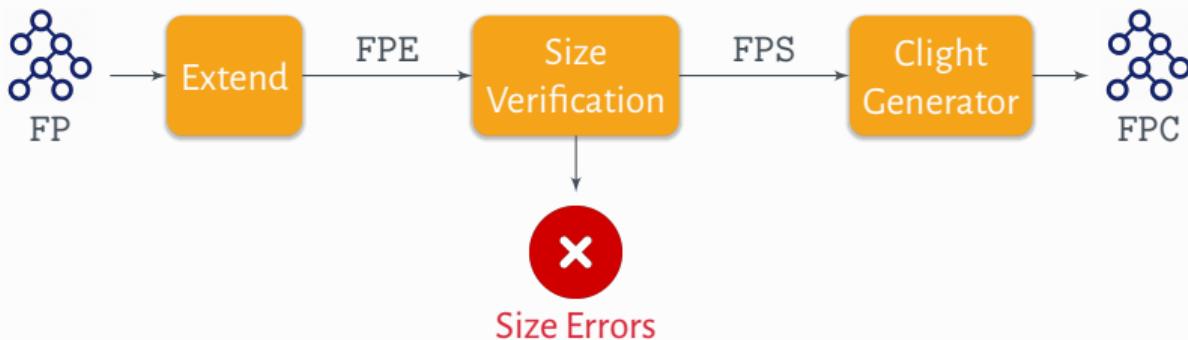
C code generated:

```
static inline void auto_nav(void) {
  switch (get_nav_block()) {
    case 0: // Block 0
      switch (get_nav_stage()) {
        case 0: // Stage 0
          func1();
        case 1: // Stage 1
          func2();
        default:
        case 3: // Default Stage
          NextBlock();
          break;
      }
      break;
    case 1: // Default Block
      GEN_DEFAULT_C_CODE()
  }
}
```

STEPS OF GENERATE_FLIGHT_PLAN FUNCTION



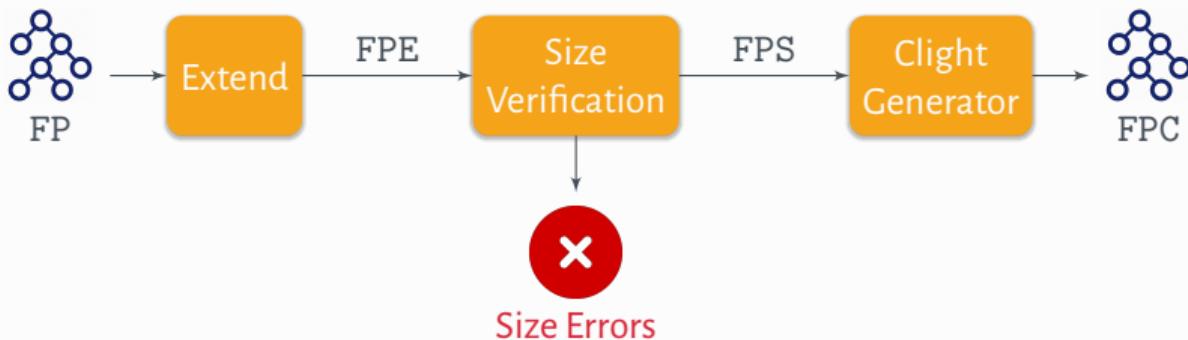
STEPS OF GENERATE_FLIGHT_PLAN FUNCTION



Extend Flight Plan:

- ▶ Index stages,
- ▶ Split NAV into NAV_INIT and NAV,
- ▶ Flatten stages contained in a WHILE stage.

STEPS OF GENERATE_FLIGHT_PLAN FUNCTION

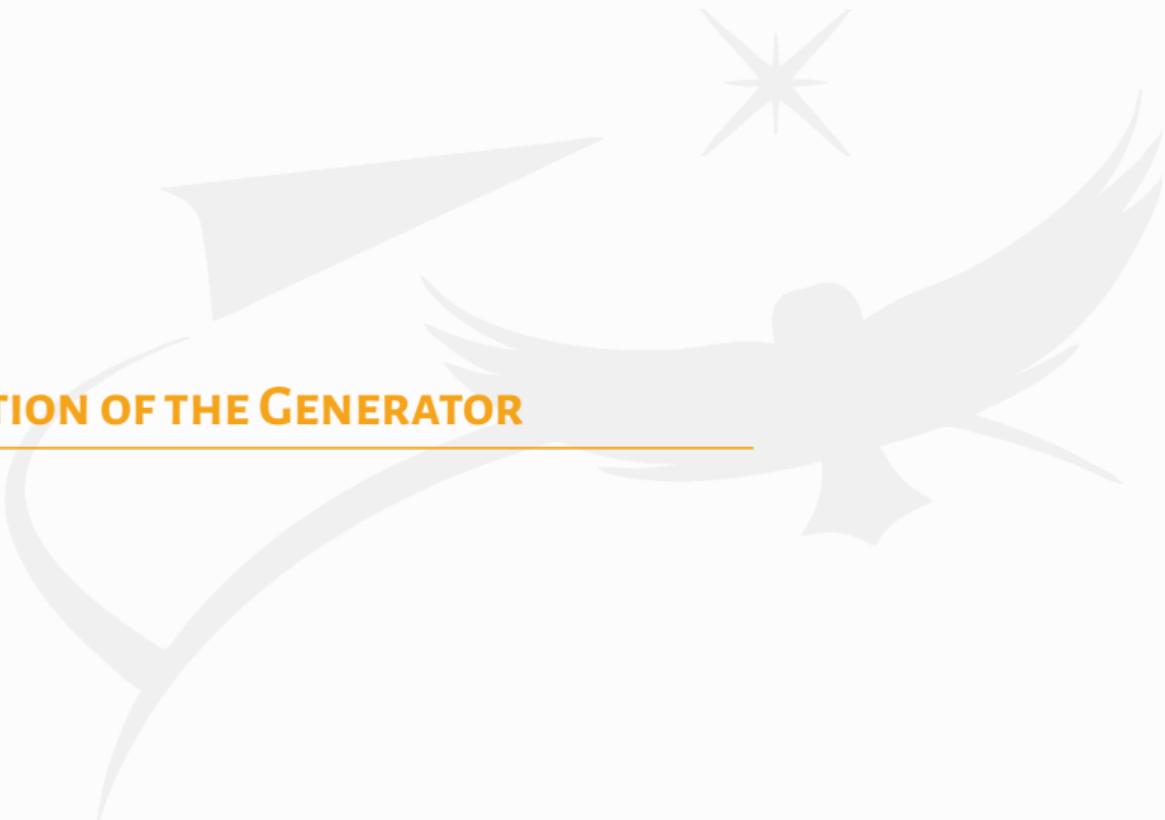


Extend Flight Plan:

- ▶ Index stages,
- ▶ Split NAV into `NAV_INIT` and `NAV`,
- ▶ Flatten stages contained in a `WHILE` stage.

Size verification:

- ▶ Check block indexing,
- ▶ Check that numbers of blocks and stages are less than 256,
- ▶ Check that `block_id` fields are 8 bits values.



VERIFICATION OF THE GENERATOR



Definition (fp_semantics)

A generic definition for the flight plan semantics.



Definition (fp_semantics)

A generic definition for the flight plan semantics.

```
Record fp_semantics: Type := FP_Semantics_gen {
  (** Environment for the semantics *)
  env: Type;
  (** Properties stating if an env is an initial environment *)
  initial_env: env → Prop;
  (** Properties stating the execution of the auto_nav function *)
  step: env → env → Prop;
}.
```



Definition (fp_semantics)

A generic definition for the flight plan semantics.

```
Record fp_semantics: Type := FP_Semantics_gen {
  (** Environment for the semantics *)
  env: Type;
  (** Properties stating if an env is an initial environment *)
  initial_env: env → Prop;
  (** Properties stating the execution of the auto_nav function *)
  step: env → env → Prop;
}.
```

Instantiation of the semantics:

- ▶ **FP semantics**: semantics_fp,
- ▶ **FPC semantics**: semantics_fpc,
- ▶ **FPE semantics**: semantics_fpe,
- ▶ **FPS semantics**: semantics_fps.

DEFINITION OF THE FP SEMANTICS

A DENOTATIONAL SEMANTICS



33

Environment: **Definition** $\text{fp_env} := (\text{fp_state} * \text{fp_trace})$.

- ▶ **fp_state**: the memory storing the execution **state** of the flight plan,
- ▶ **fp_trace**: the memory that can be modified by flight plan **external functions**.

DEFINITION OF THE FP SEMANTICS

A DENOTATIONAL SEMANTICS



33

Environment: Definition $\text{fp_env} := (\text{fp_state} * \text{fp_trace})$.

- ▶ fp_state the memory storing the execution **state** of the flight plan,
- ▶ fp_trace : the memory that can be modified by flight plan **external functions**.

Initial environment property noted $\text{initial_env } e$



Environment: Definition $\text{fp_env} := (\text{fp_state} * \text{fp_trace})$.

- ▶ fp_state the memory storing the execution **state** of the flight plan,
- ▶ fp_trace : the memory that can be modified by flight plan **external functions**.

Initial environment property noted $\text{initial_env } e$

Step property noted $e \xrightarrow{\text{FP}} e'$.

- ▶ Defined as a **function** for early validation purposes: $e \xrightarrow{\text{FP}} e' := \text{step } e = e'$



Environment: Definition $\text{fp_env} := (\text{fp_state} * \text{fp_trace})$.

- ▶ fp_state the memory storing the execution **state** of the flight plan,
- ▶ fp_trace : the memory that can be modified by flight plan **external functions**.

Initial environment property noted $\text{initial_env } e$

Step property noted $e \xrightarrow{\text{FP}} e'$.

- ▶ Defined as a **function** for early validation purposes: $e \xrightarrow{\text{FP}} e' := \text{step } e = e'$
- ▶ Interpretation of arbitrary C code.
 - ▶ **Hypothesis:** Arbitrary C code **terminates** and **does not modify** the FP state.
 - ▶ **Parameter eval:** $\text{fp_env} \rightarrow \text{cond} \rightarrow (\text{bool} * \text{fp_env})$.

DEFINITION OF THE FP SEMANTICS

EXAMPLE OF SOME INFERENCE RULES



34

- Inference rules for the **WHILE** stage.

$$e.\text{stages} = \text{WHILE } (\text{cond}, \text{body}) :: s \quad \text{eval cond} = (\text{true}, e')$$

$$e' \{ \text{stages} := \text{body} ++ e.\text{stages} \} = e''$$

$$e \xrightarrow{\text{FP}} e''$$

$$e.\text{stages} = \text{WHILE } (\text{cond}, \text{body}) :: s \quad \text{eval cond} = (\text{false}, e')$$

$$e' \{ \text{stages} := s \} \xrightarrow{\text{FP}} e''$$

$$e \xrightarrow{\text{FP}} e''$$

- Inference rules for the **NAV** stage.

$$e.\text{stages} = \text{NAV } (\text{mode}, \text{true}) :: s \quad e(\text{init_nav_code mode}) = e'$$

$$e' \{ \text{stages} := \text{NAV } (\text{mode}, \text{false}) :: s \} = e''$$

$$e \xrightarrow{\text{FP}} e''$$

$$e.\text{stages} = \text{NAV } (\text{mode}, \text{false}) :: s$$

$$e \xrightarrow[\text{nav}]{\text{FP}} e'$$

$$e \xrightarrow{\text{FP}} e'$$

BISIMULATION RELATION



fp_simulation describes if FP2 can simulate every behaviour of FP1.



fp_simulation describes if FP2 can simulate every behaviour of FP1.

```
Record fp_simulation(FP1 FP2: fp_semantics)
  (match_envs: env FP1 → env FP2 → Prop): Prop := {
  match_initial_envs:
    ∀ (e1: FP1.env), initial_env e1 →
      ∃ (e2: FP2.env), initial_env e2 ∧ match_envs e1 e2;
  match_step:
    ∀ (e1 e1': FP1.env), step e1 e1' →
      ∀ (e2: FP2.env), match_envs e1 e2 →
        ∃ (e2': FP2.env), step e2 e2' ∧ match_envs e1' e2';
}.
}
```



fp_simulation describes if FP2 can simulate every behaviour of FP1.

```
Record fp_simulation (FP1 FP2: fp_semantics)
  (match_envs: env FP1 → env FP2 → Prop): Prop := {
  match_initial_envs:
    ∀ (e1: FP1.env), initial_env e1 →
      ∃ (e2: FP2.env), initial_env e2 ∧ match_envs e1 e2;
  match_step:
    ∀ (e1 e1': FP1.env), step e1 e1' →
      ∀ (e2: FP2.env), match_envs e1 e2 →
        ∃ (e2': FP2.env), step e2 e2' ∧ match_envs e1' e2';
}.
}
```

Definition of a **bisimulation** relation between 2 semantics.

```
Inductive bisimulation (FP1 FP2: fp_semantics): Prop :=
  Bisimulation (match_envs: env FP1 → env FP2 → Prop)
    (forward_simulation: fp_simulation FP1 FP2 match_envs).
    (backward_simulation: fp_simulation FP2 FP1 match_envs).
```



Theorem (bisim_fp_fpc)

$$\begin{aligned} & \forall \text{fp prog warnings}, \\ & \quad \text{generator fp} = \text{CODE}(\text{prog, warnings}) \\ & \quad \rightarrow \text{bisimulation}(\text{semantics_fp fp})(\text{semantics_fpc prog}). \end{aligned}$$

This theorem states that the generator **preserves the semantics**.



Theorem (bisim_fp_fpc)

$$\forall \text{fp prog warnings}, \\ \text{generator fp} = \text{CODE}(\text{prog, warnings}) \\ \rightarrow \text{bisimulation}(\text{semantics_fp fp})(\text{semantics_fpc prog}).$$

This theorem states that the generator **preserves the semantics**.

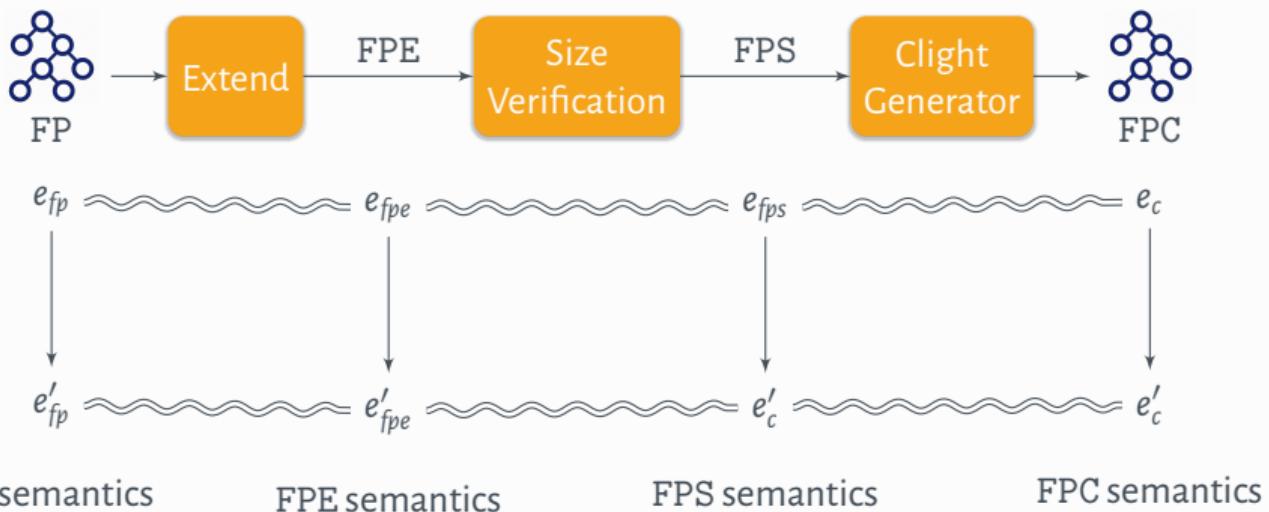
Forward simulation

FP behaviour is simulated by the Clight code.

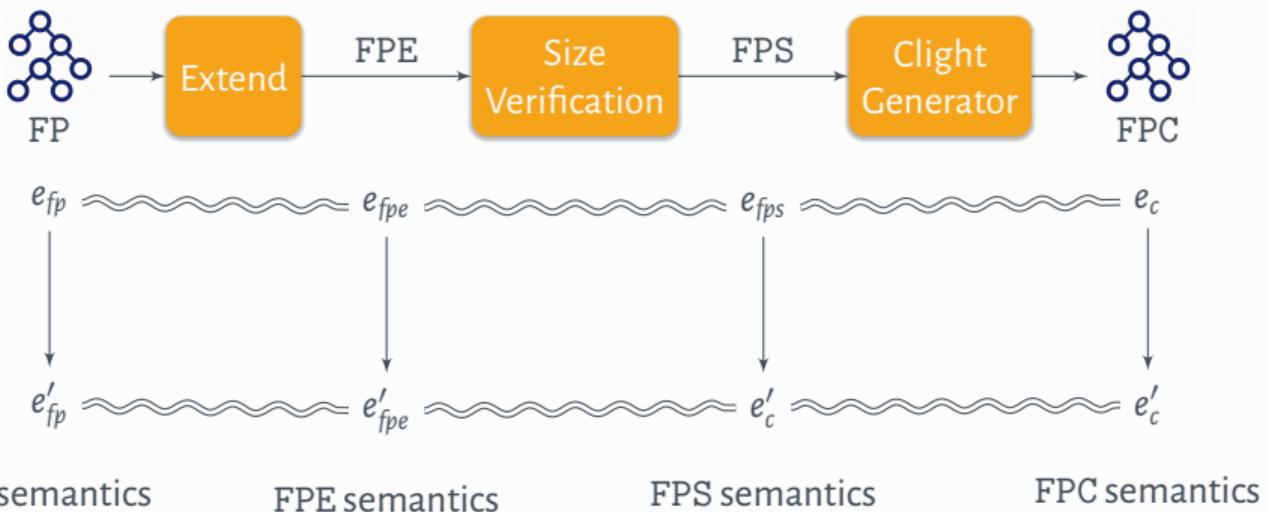
Backward simulation

Every possible execution of the Clight code is described by the FP semantics.

VERIFICATION OF THE CORRECTNESS THEOREM



VERIFICATION OF THE CORRECTNESS THEOREM



Lemma `compose_bisimulations`:

- ▽ $\text{FP1 } \text{FP2 } \text{FP3, bisimulation } \text{FP1 } \text{FP2}$
 - $\text{bisimulation } \text{FP2 } \text{FP3}$
 - $\text{bisimulation } \text{FP1 } \text{FP3.}$



Interpretation of the **arbitrary C code**.

Hypothesis: Arbitrary C code **terminates** and **does not modify** the FP state.



Interpretation of the **arbitrary C code**.

Hypothesis: Arbitrary C code **terminates** and **does not modify** the FP state.

New axioms to extend the **operational** Clight semantics.

⇒ These axioms convert arbitrary C code into traces.

Note

These axioms can be **improved by modifying the generation** of the arbitrary C code.



Interpretation of the **arbitrary C code**.

Hypothesis: Arbitrary C code **terminates** and **does not modify** the FP state.

New axioms to extend the **operational** Clight semantics.

⇒ These axioms convert arbitrary C code into traces.

Note

These axioms can be **improved by modifying the generation** of the arbitrary C code.

Classic logic axioms from Coq standard library:
excluded middle, proof irrelevance and functional extensionality.



Constrained by the previous generator:

Input language, C code generated...

Split the proof in 3 independent parts.

Verification functions produce dependent type.

⇒ **Avoid axioms, improves confidence in preprocessing.**

Forced clarification of the semantics:

- ▶ **Unexpected behaviour** (ex: *RETURN after a DEROUTE*),
- ▶ **Bug** (ex: *the FP contains more than 256 blocks/stages*).



Constrained by the previous generator:

Input language, C code generated...

Split the proof in 3 independent parts.

Verification functions produce dependent type.

⇒ **Avoid axioms, improves confidence in preprocessing.**

Forced clarification of the semantics:

- ▶ **Unexpected behaviour** (ex: *RETURN after a DEROUTE*),
- ▶ **Bug** (ex: *the FP contains more than 256 blocks/stages*).

⇒ **2,100 loc of OCaml and 20,000 loc of Coq (14% of functional code).**

TABLE OF CONTENTS

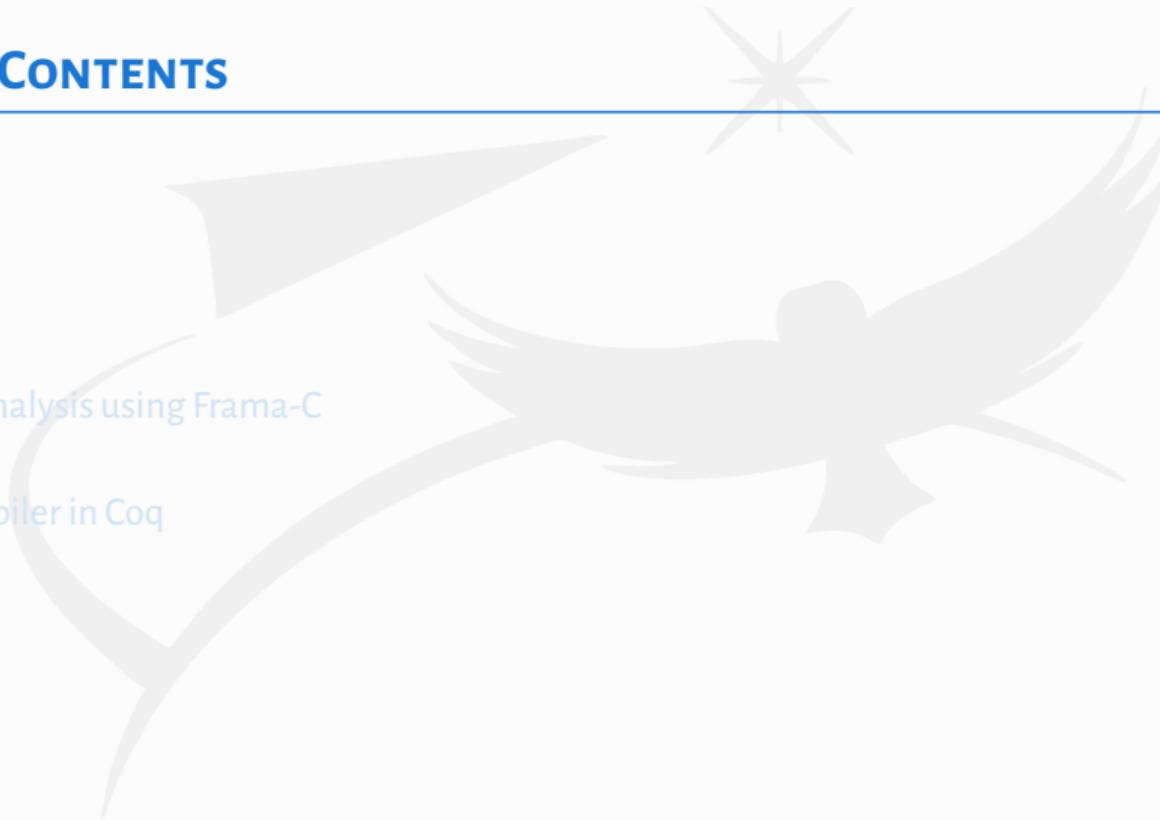
Introduction

Paparazzi

Static Code analysis using Frama-C

Verified Compiler in Coq

Conclusion



CONCLUSION

THE VERIFICATION OF 2 PAPARAZZI CRITICAL COMPONENTS



The C **mathematical library** verified using **Frama-C**

- ▶ Verification of the absence of runtime errors in the library,
- ▶ Verification of functional properties on some floating-point functions.

CONCLUSION

THE VERIFICATION OF 2 PAPARAZZI CRITICAL COMPONENTS



The C **mathematical library** verified using **Frama-C**

- ▶ Verification of the absence of runtime errors in the library,
- ▶ Verification of functional properties on some floating-point functions.

A **flight plan generator** verified using **Coq**

- ▶ Development of a new generator in Coq with new features,
- ▶ Formalisation of the flight plan semantics,
- ▶ Verification of the preservation of the semantics.

CONCLUSION

THE VERIFICATION OF 2 PAPARAZZI CRITICAL COMPONENTS



The C **mathematical library** verified using **Frama-C**

- ▶ Verification of the absence of runtime errors in the library,
- ▶ Verification of functional properties on some floating-point functions.

A **flight plan generator** verified using **Coq**

- ▶ Development of a new generator in Coq with new features,
- ▶ Formalisation of the flight plan semantics,
- ▶ Verification of the preservation of the semantics.

Limitation of these verification processes

- ▶ Verifications based on hypotheses,
- ▶ Require an expertise in formal methods,
- ▶ High cost in terms of maintainability.

Continue the verification of the **mathematical library**:

- ▶ Verification of calls to library functions,
- ▶ Verifying the floating-point library without the `real` model,

Continue the verification of the **mathematical library**:

- ▶ Verification of calls to library functions,
- ▶ Verifying the floating-point library without the `real` model,

Improve the **new flight plan generator**:

- ▶ Verify new properties on the flight plan language,
- ▶ Reduce the number of pre-processing steps,
- ▶ Generalise the generator.

Continue the verification of the **mathematical library**:

- ▶ Verification of calls to library functions,
- ▶ Verifying the floating-point library without the `real` model,

Improve the **new flight plan generator**:

- ▶ Verify new properties on the flight plan language,
- ▶ Reduce the number of pre-processing steps,
- ▶ Generalise the generator.

Verify the **Paparazzi autopilot generator**, similarly to VFPG.

Continue the verification of the **mathematical library**:

- ▶ Verification of calls to library functions,
- ▶ Verifying the floating-point library without the `real` model,

Improve the **new flight plan generator**:

- ▶ Verify new properties on the flight plan language,
- ▶ Reduce the number of pre-processing steps,
- ▶ Generalise the generator.

Verify the **Paparazzi autopilot generator**, similarly to VFPG.

Use **model checking** approaches:

- ▶ Verify critical C code using model checking tools such as CBMC,
- ▶ Verify design of hardware components.

Formal verification of an UAV autopilot

Static analysis and Verified Code Generation

Case study: Paparazzi

Project publicly available

- ▶ **Verified library:** gitlab.isae-supaero.fr/b.pollien/paparazzi-frama-c
- ▶ **VFPG:** gitlab.isae-supaero.fr/b.pollien/vfpg

Publications

- ▶ Technical report:
 - ▶ Formal verification for autopilots: preliminary state of the start
 - ▶ A gentle introduction to C code verification using the Frama-C platform
- ▶ Verification of some parts of Paparazzi mathematical library
Publications: AFADL 2021, FMICS 2021
- ▶ Development of a verified flight plan generator
Publications: FormaliSE 2023



The **drone environment** can be modelled in a variety of ways.

From the point of view of the flight plan execution, the **global drone environment** can be split into 2 distinct elements:

- ▶ the memory storing the execution **state** of the flight plan,
- ▶ the memory that can be modified by flight plan **external functions**.

Remark

External functions can be:

- ▶ complex functions that corresponds to navigation stages,
- ▶ arbitrary C code contained in the flight plan.

⇒ **It is not possible to represent the effect of their execution.**

⇒ **We assume that these 2 memory regions are strictly disjoint.**



The FP semantics will use `fp_env`, an abstraction of the drone environment.

```
Record fp_env := {  
    state: fp_state;  
    trace: fp_trace;  
}.
```

`fp_state` represents an abstraction of the current state of the flight plan.

```
Record fp_state := {  
    idb: block_id; stages: list fp_stages; (* Current position *)  
    lidb: block_id; lstages: list fp_stages; (* Last position *)  
}
```

A **position** is a couple of a block ID and the remaining stages to execute.

`fp_trace` represents the history of external functions execution.

```
Variant fp_event := COND (cond * bool) | C_CODE (c: c_code).  
Definition fp_trace := list fp_event.
```



Contracts of the trigonometric functions from the libc do not provide mathematical results.

⇒ Extend the contracts.

ex: *Extension of the contract for the function sinf.*

```
/*@ requires finite_arg: \is_finite(x);
   assigns \result \from x;
   ensures finite_result: \is_finite(\result);
   ensures result_domain: -1. <= \result <= 1. ;
   ensures result_value: \result == \sin(x);
*/
extern float sinf(float x);
```

Some lemmas could not be proved by the SMT solvers.

⇒ Enable interactive mode of Frama-C to use Coq.

- ▶ Lemma to verify the correctness of the function `quat_of_rmat`

$\forall R \in SO_3(\mathbb{R}), \forall q \in \mathbb{H},$

$$\|q\| > 0 \wedge Tr(R) > 0$$

$$\rightarrow (R = \text{rmat_of_quat}(q) \leftrightarrow q = \text{quat_of_rmat}(R))$$

- ▶ Lemma used to verify that `rmat_of_euler` compute rotation matrix:

$\forall a, b, c \in \mathbb{R},$

$$\sin(a)^2 \cos(b)^2$$

$$+ (\sin(a) \sin(b) \cos(c) - \sin(c) \cos(a))^2$$

$$+ (\cos(c) \cos(a) + \sin(a) \sin(b) \sin(c))^2 = 1$$