

# A Verified UAV Flight Plan Generator

FormaliSE 2023

---

B. Pollien<sup>1</sup>, C. Garion<sup>1</sup>, G. Hattenberger<sup>2</sup>, P. Roux<sup>3</sup>, X. Thirioux<sup>1</sup>

May 2023

<sup>1</sup>ISAE-SUPAERO, <sup>2</sup>ENAC and <sup>3</sup>ONERA

Paparazzi is an autopilot for micro-drones

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

Complete drone control system:

- Control software part,
- Design of some hardware components,
- Support for ground and aerial vehicles,
- Support for simultaneous control of several drones,
- User can define their own mission using **flight plans**.



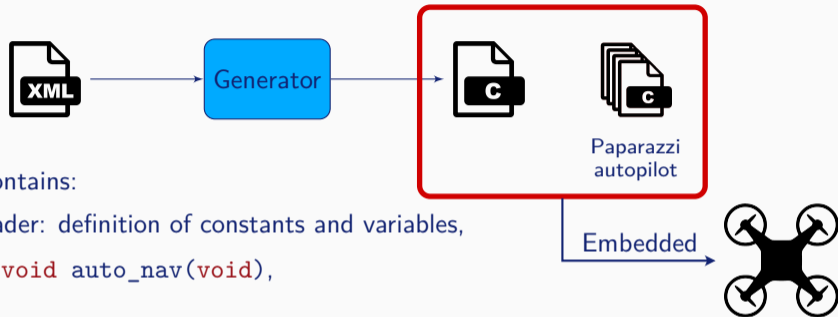
The **flight plan** (FP)

- 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)`,
- Auxiliary functions:  
pre\_call\_block, post\_call\_block and forbidden\_deroute.

⇒ **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.

⇒ **Certified Compilation problem**

## Solutions to similar problems

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

**Coq** is a proof assistant

- Developed by Inria,
- Based on the Gallina language.

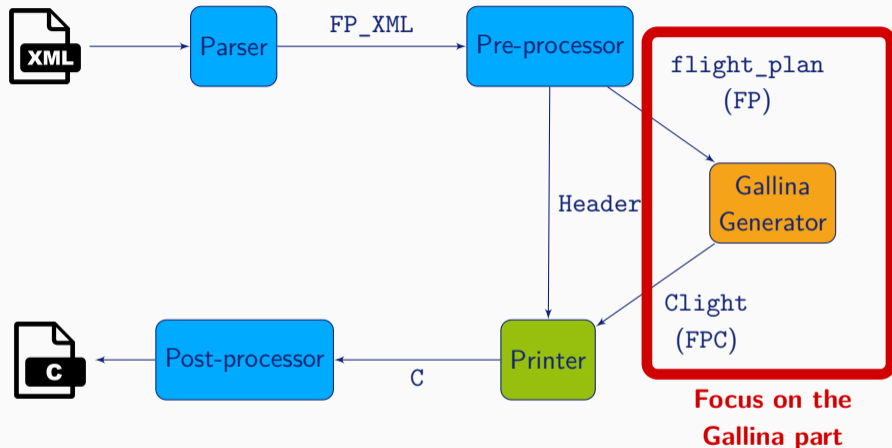
Software for writing and verifying formal proofs

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

**Our solution:** New flight plan generator developed and verified in Coq.



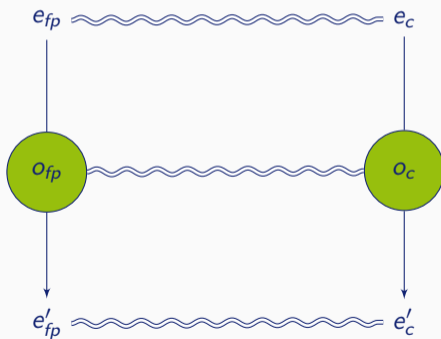
# New Verified Flight Plan Generator (VFPG)



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

**Post-processing:** Produces a compilable C code.

# Overview of the semantics preservation proof



FP semantics

FPC semantics



# 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

## 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()<br>!GPSFixValid() ↑ true        |
| 2    | 0             | !GPSFixValid() ↑ true                         |
| 3    | 0             | !GPSFixValid() ↑ true                         |
| ⋮    | ⋮             | ⋮   |
| 9    | 0             | !GPSFixValid() ↑ false<br>home = GPSPosHere() |
| 10   | 1             | StartMotors()                                 |
| 11   | 1             | TakeOffDone() ↑ false                         |
| 12   | 1             | TakeOffDone() ↑ false                         |
| ⋮    | ⋮             | ⋮   |
| 20   | 1             | TakeOffDone() ↑ true<br>Deroute → 10          |
| 21   | 10            | ...   |
| ⋮    | ⋮             | ⋮   |

# Generator

---

# Generator Function

**Definition** `generate_flight_plan`:

`flight_plan`  $\rightarrow$  `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

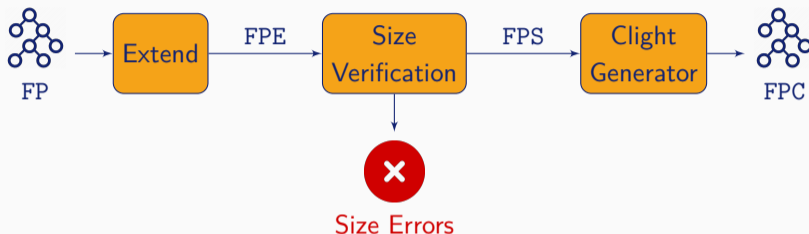
## Example of a flight plan:

```
{| 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



## Extended 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

---



# Generic Big Step Semantics for Flight Plans

## 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;  
}.
```

Instanciation of the semantics:

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

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.**

# FP Semantics: Abstraction of the Drone Environment

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.
```

# Bisimulation relation

**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 L1 → env L2 → Prop)
    (forward_simulation: fp_simulation FP1 FP2 match_envs).
  (backward_simulation: fp_simulation FP2 FP1 match_envs).
```

# Correctness theorem of the generator

## Theorem (`bisim_fp_fpc`)

$\forall$  fp prog warnings,  
generator fp = CODE (prog, warnings)  
 $\rightarrow$  bisimulation (semantics\_fp fp) (semantics\_fpc prog).

This theorem states that the generator preserves the semantics.

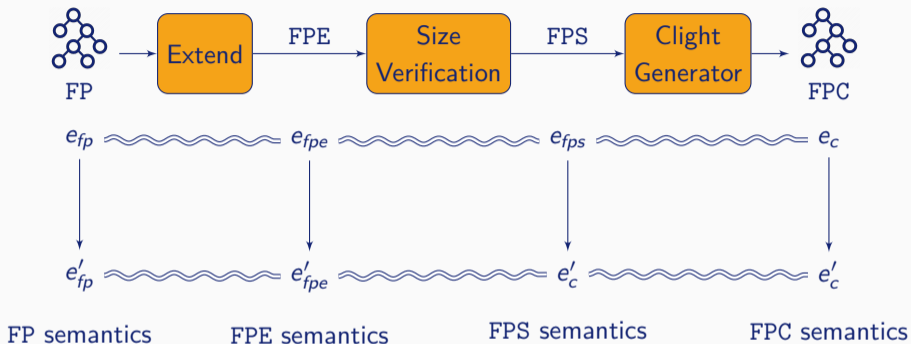
## Forward simulation

FP behaviours is simulated by the Clight code.

## Backward simulation

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

# Verification of the bisim\_fp\_fpc



**Lemma** compose\_bisimulations:

- ∀ FP1 FP2 FP3, bisimulation FP1 FP2
  - bisimulation FP2 FP3
  - bisimulation FP1 FP3.

Interpretation of the **arbitrary C code**.

⇒ **Parameter** eval:  $\text{fp\_env} \rightarrow \text{cond} \rightarrow (\text{bool} * \text{fp\_env})$ .

⇒ New axioms to convert arbitrary C code into traces.

Axiom stating that the function **create\_ident** is injective.

Classical Coq standard library:

*excluded middle, proof irrelevance and functional extensionality.*

An axiom to prove that the Clight semantics is deterministic.

*not mandatory, will be removed in future work.*

## **Lessons Learned & Conclusion**

---



## Development methodology

- Constrained by the previous generator:  
*Input language, C code generated...*
- Split the proof in 3 independent parts.
- Forced clarification of the semantics:
  - Unexpected behaviour (ex: *RETURN* after a *DEROUTE*),
  - Bug (ex: the FP contains more than 256 blocks/stages).

## Technical remarks

- 1.3k loc of OCaml and 17k loc of Coq (12% of working code).
- Verification functions produce dependent type.  
⇒ Avoid axioms, improves confidence in preprocessing.
- Using Clight has some downside.

## Summary:

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

## Perspectives:

- Remove axiom to prove that Clight semantics is deterministic,
- Verify properties on the flight plan language,
- Integrate the new generator in Paparazzi toolchain,
- Reduce the number of pre-processing steps,
- Generalize the generator.

This work is supported by the Defense Innovation Agency (AID) of the French Ministry of Defense  
(research project CONCORDE N 2019 65 0090004707501)