Paparazzi UAV Flight Plan Generator Verified with Coq

Concorde Seminare 2022

C. Garion[1], G. Hattenberger[2], B. Pollien[1], P. Roux[3], X. Thirioux[1]

April 2022

[1]ISAE-SUPAERO, [2]ENAC and [3]ONERA

Presentation of Paparazzi

Paparazzi is an autopilot for micro-drones

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

Complete drone control system:

- Offers the control software part,
- Also offers some designs of hardware components,
- Supports for ground and aerial vehicles,
- Supports for simultaneous control of several drones.

Flight Plan

The flight plan (FP)

- describes how the drone might behave when it is launch,
- 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*.

Remark: The user can interact with flight plan during a flight.

Presentation of the Generator

The **C file generated** contains:

- Flight Plan Header: definition of constantes and variables,
- The main function `void auto_nav(void)`,
- Other auxilary functions:
  `pre_call_block`, `post_call_block` and `forbidden_deroute`.

$\Longrightarrow$ Compiled with the autopilot and embedded on the drone.

Function `auto_nav`:

- Called at 20 Hz,
- Sets navigation parameters.

XML File Describing the Flight Plan

Flight plan architecture:

1. Header
2. Waypoints
3. Sectors
4. Modules
5. Includes

(1–5: Flight Plan Header)

6. Blocks := list of Block
   Block := list of Stage
7. Exceptions
8. Forbidden Deroutes (*New*)

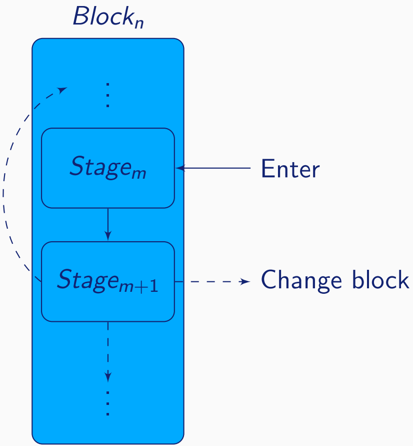
Stages supported:

- While
- Set
- Call
- Deroute
- Return
- Nav: Go, Circle, Stay, Survey Rectangle, Oval, Home...
- *Path, For, Call_Once*

Remark: The flight plan can contain arbitrary C code.

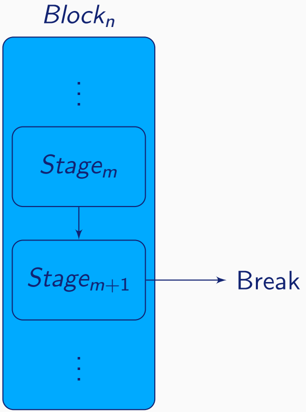
Continue vs Break Stage

Continue stage:



Ex: Call_Once "`NavStartDetectGround()`"

Break stage:



Ex: Go "`WP1`"

$\Longrightarrow$ **Risk of an infinite loop**

Motivation

Problems:

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

$\Longrightarrow$ **Compilation problem**

Solution to similar problems

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

Coq

Coq is a proof assistant

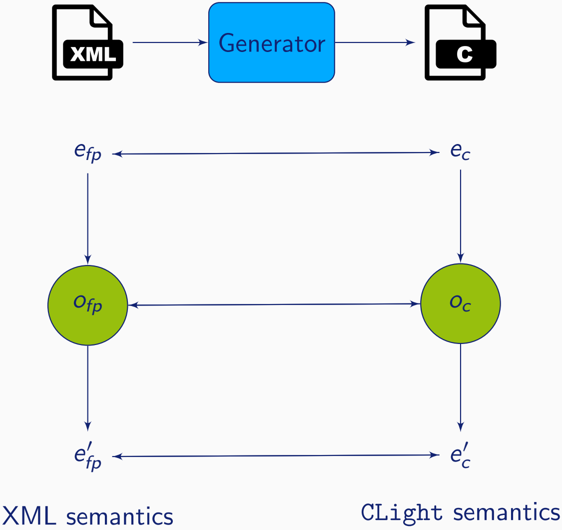
- Developed by Inria,
- Based on Gallina language.

Software for writing and verifying formal proofs

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

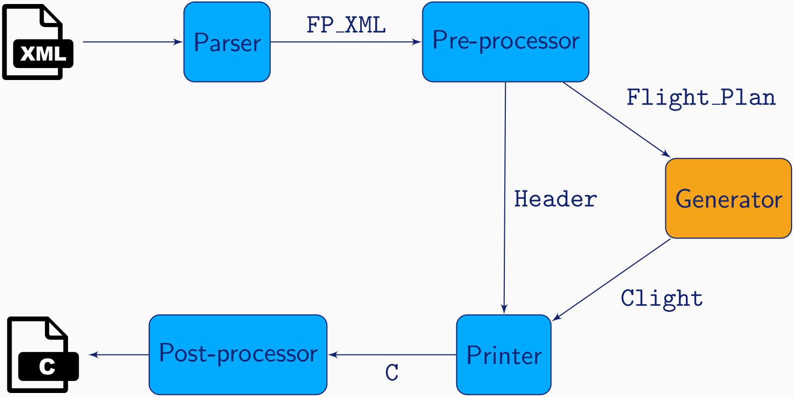
Our solution: New flight plan generator developed and verified in Coq.

Process to Develop a Verified Generator



Generator

VFPG (Verified Flight Plan Generator)



Pre-Processing

Pre-processing: several transformations are performed on the flight plan

- Manage included files that contain processus
- Update, convert and verify the coordinates
- Add a safety home block
- Process `paths`
- Process `for` loops and compute the list of local variables
- Index the blocks

Flight Plan Structure in Coq

```
Inductive fp_stage :=
  WHILE (params: fp_params_while)
        (block: list fp_stage)
  | SET (params: fp_params_set)
  | CALL (params: fp_params_call)
  | DEROUTE (params: fp_params_deroute)
  | RETURN (params: fp_params_return)
  | NAV (nav_mode: fp_navigation_mode)
        (init: bool).

Definition fp_block :=
  (* Index of the block *)
  nat
  (* List of local exceptions *)
  * fp_exceptions
  (* Parameters of the block *)
  * fp_params_block
  (* List of stage *)
  * list fp_stage
```

```
Definition flight_plan :=
    (* List of deroutes forbidden *)
    fp_forbidden_deroutes
    (* List of exceptions *)
    * fp_exceptions
    (* List of block *)
    * list fp_block.
```

Generator Function

`generate_flight_plan:`

```
flight_plan -> list gdef -> (Clight * list err_msg)
```

Inputs:

- Flight plan to convert,
- List of local variables.

Ouputs:

- Clight program generated
- List of warnings and errors found during the generation.
  For now: detect if there is a possible deroute that is forbidden.

Example of C Code Generated

Example of a Coq flight plan:

```
Variable s0, s1: fp_stage.
Variable le1, e: fp_exceptions.
Variable p1: fp_params_block;
Variable fbd1: fp_forbidden_deroutes.

Definition b0: fp_block :=
  (0, le1, p1, [s0, s1])

Definition fp: flight_plan :=
  (fbd1, e, [b0])
```

C code generated:

```
static inline void auto_nav(void) {
  switch (get_nav_block()) {
    case 0: // Block b0
      set_nav_block(0);
      switch (get_nav_stage()) {
        case 0: // Stage s0
          set_nav_stage(0);
          C_CODE(s0)
        case 1: // Stage s1
          set_nav_stage(1);
          C_CODE(s1)
        default:
        case 3: // Default Stage
          set_nav_stage(3);
          NextBlock();
          break;
        }
      break;
    case 1: // Default Block
      C_CODE(DEFAULT_BLOCK)
  }
}
```

Semantics of the Flight Plan

Abstraction of the External Drone Environment

The drone environment of the flight plan is too complex.

$\Longrightarrow$ *fp_env* represents an abstraction of the current state of the flight plan.

```
Definition fp_env := block_id * list fp_stage (* Current position *)
                   * block_id  * list fp_stage. (* Last position *)
```

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

Abstraction of the External Drone Functions

- Execution of navigation stages corresponds to complex function call.
- The flight plan can potentially contain arbitrary C code.

$\Longrightarrow$ The semantics will generate a trace for these calls.

```
Variant c_exec := COND (c: cond) | C_CODE (c: c_code) | SKIP.
Definition outputs := list c_exec.
```

- We also need the result of the evaluation of conditions.

$\Longrightarrow$ Definition of the function `eval`

```
Parameter eval: time → cond → (bool * time).
```

Evaluates a condition at a time t and produce a boolean result at a time $t' > t$.

Big Step Semantics of the Flight Plan

Big Step Function

Represents the execution of the `auto_nav` function starting from a state e and finishing in a state e'.

$$(t, e) \overset{\text{fp}}{\hookrightarrow}_o (t', e')$$

- t, t' are time instants as the drone evolve continuously
- o are the generated ouputs, i.e. all the extern C code called

As the function is defined in Coq it terminates.

Example of semantics inferance rules for CALL stage[1]

Evaluation of the c code returns true:

$$\frac{s = \texttt{CALL}\ c \qquad eval\ \ t\ \ c = (true, t')}{(t, s\ \ ::\ \ stages) \overset{\text{fp}}{\hookrightarrow}_{[COND\ \ c]} (t', s\ \ ::\ \ stages)}$$

Evaluation of the c code returns false:

$$\frac{s = \texttt{CALL}\ c \qquad eval\ \ t\ \ c = (false, t') \qquad (t', stages) \overset{\text{fp}}{\hookrightarrow}_{o} (t'', stages')}{(t, s\ \ ::\ \ stages) \overset{\text{fp}}{\hookrightarrow}_{COND\ \ c::o} (t'', stages')}$$

[1] To simplify the notation, the *fp_env* only contains a list of remaining stages and we consider that there is no change of block

Semantics of Clight defined in CompCert

```
Variable ge: genv. (* Global environment: symbols and functions *)
Variable e: env. (* Local environments: map variables to location. *)
Variable le1, le2: temp_env. (* Temp env: maps local temporaries to values. *)
Variable m1, m2: mem. (* Memories: maps adresses to values. *)
Variable s: statement.
Variable T: trace. (* List of event (load, store, syscall) *)
Variable out: outcome. (* Break, continue, return or normal*)
```

```
exec_stmt ge e le1 m1 s T le2 m2 out.
```

`exec_stmt` is a Coq proposition that describes the execution of the statement `s` in the environment (`ge`, `e`).

$$(\mathtt{le1}, \mathtt{m1}) \Downarrow^{(\mathtt{s},\mathtt{ge},\mathtt{e})}_{(\mathtt{out},\mathtt{T})} (\mathtt{le2}, \mathtt{m2})$$

Example: C vs Printed Clight.

```
int a = fun1(fun2(10));
```

```
int a; int tmp;
tmp = fun2(10);
fun1(tmp);
```

Execution of abstracted functions

Axiom: Execution to Trace

$$\forall ge\ e\ f\ m\ m'\ out\ T,$$
$$(m) \Downarrow^{(SCALL\ f, ge, e)}_{(out, T)} (m')$$
$$\rightarrow\ m = m'$$
$$\land\ out = \texttt{Out\_normal} \land T = [SYS\_CALL\ f]$$

Preservation of the Semantics

Common Flight Plan Code

The generated C code depends on C functions and variables that can be abstracted and defined in the file `common_flight_plan.c`.

$\Longrightarrow$ These functions are converted into `CommonFP.v` using `clightgen`.

`CommonFP.v` contains the CLight definitions of

- Some global variables (`private_nav_stage, private_nav_block`)
- Some functions (`get_nav_stage, get_nav_block, NextBlock`)

All these definitions form ge_{fp}, the common global environment.

Similarly, e_{fp} is common for all flight plans as the number of local variables for the function `auto_nav` is fixed.

Verification of the Preservation of the Semantics

Suppose we have:

- $\overset{env}{\sim}$ an equivalence relation between `fp_env` and (`temp_env` * `mem`).
- $\overset{output}{\sim}$ an equivalence relation betweeen `output` and `trace`.

Theorem: Preservation of the Semantics

$$\forall fp\ prog\ \ e\ \ m\ \ e'\ \ t\ \ t'\ \ o\ ,$$

$$prog = \texttt{generate\_flight\_plan}\ fp$$

$$\rightarrow e \overset{env}{\sim} m$$

$$\rightarrow (t, e) \overset{\mathrm{fp}}{\hookrightarrow}_o (t', e')$$

$$\rightarrow \exists m'\ \ T,$$

$$(m) \Downarrow^{(prog, ge_{fp}, e_{fp})}_{(Out\_normal, T)} (m')$$

$$\bigwedge e' \overset{env}{\sim} m' \bigwedge o \overset{output}{\sim} T$$

Conclusion

Conclusion

Summary:

- Development of the generator in Coq,
- Formalisation of the flight plan semantics,
- Add new features.

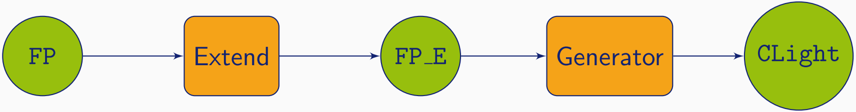
Perspectives:

- Verification of the preservation of the semantics,
- Reduce the number of steps in pre-processing,
- Verify new properties.

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

Thank you

Flight Plan Extension



Extended Flight Plan:

- Numerotation of the stage,
- Split `NAV` into `NAV_INIT` and `NAV`,
- Inline all stage contained in the `WHILE` (stage `END_WHILE` is then added).

$\Longrightarrow$ Allow to define a environments close to the CLight semantics.