Paparazzi UAV Flight Plan Generator Verified with Coq

AID 2022

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

June 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

   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.

Example: Potential Execution of a Flight Plan

Flight Plan:

⋮

```
<block name="Wait GPS">
    <call_once fun="NavKillThrottle()"/>
    <while cond="!GpsFixValid()"/>
</block>
<block name="Start Engine">
    <call_once fun="NavResurrect()"/>
    <attitude pitch="0" roll="0" throttle="0"
              until="FALSE"/>
</block>
<block name="Takeoff">
    <exception
        cond="stateGetPositionEnu_f().z > 2.0"
        deroute="Standby"/>
    <call_once fun="NavSetWaypointHere(WP_CLIMB)"/>
    <stay vmode="climb" climb="nav_climb_vspeed"
          wp="CLIMB"/>
</block>
```

⋮

Results of `auto_nav`:

($T : 0 \times p$, Block: `Wait GPS`):
`NavKillThrottle()`
`GpsFixValid()`$\Uparrow$`false`.

($T : 1 \times p$, Block: `Wait GPS`):
`GpsFixValid()`$\Uparrow$`false`.

($T : 2 \times p$, Block: `Wait GPS`):
`GpsFixValid()`$\Uparrow$`false`.

⋮

($T : n \times p$, Block: `Wait GPS`):
`GpsFixValid()`$\Uparrow$`true`.

($T : (n+1) \times p$, Block: `Start Engine`):
`NavResurrect()`
`NavAttitude(0, 0, 0)`

($T : (n+2) \times p$, Block: `Takeoff`):

⋮

$\Longrightarrow$ **Possible risks 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

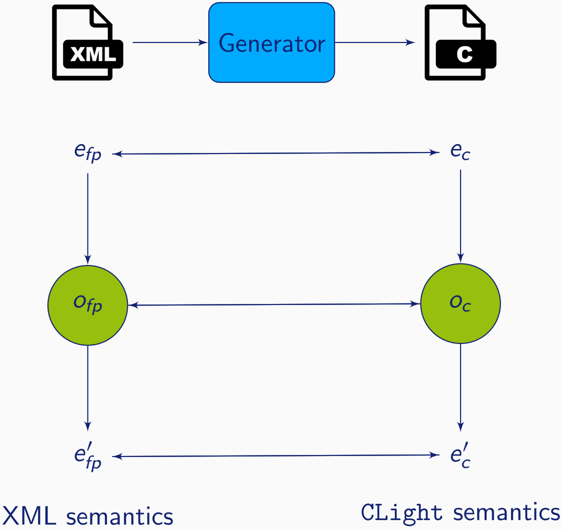
- Developed by Inria,
- Based on Gallina language.

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.

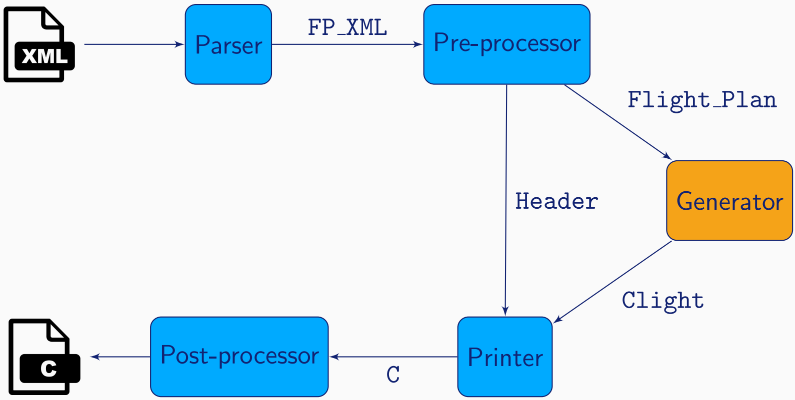
Our solution: New flight plan generator developed and verified in Coq.

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 flight plan:

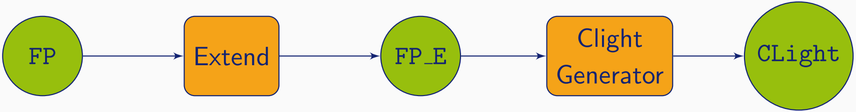
```
⋮

<blocks>
    <block name="b0">
        <stage s0/>
        <stage s1/>
    </block>
</blocks>
⋮
```

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)
  }
}
```

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 have a structure similar to the C code generated.

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

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'.

$$e \overset{\text{fp}}{\hookrightarrow}_o e'$$

- o are the generated ouputs, i.e. all the extern C code called

As the function is defined in Coq it terminates.

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`). We note:

$$m1 \Downarrow^{\mathtt{s}}_{(\mathtt{out},\mathtt{t})} m2$$

Preservation of the Semantics

Verification of the Preservation of the Semantics

Suppose we have:

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

Theorem: Preservation of the Semantics

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

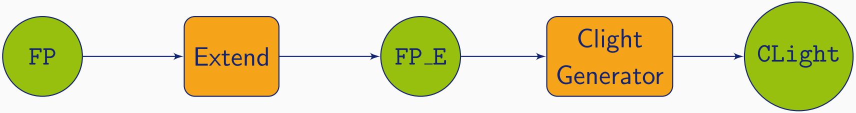
$$prog = \texttt{generate\_flight\_plan}\ fp$$

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

$$\rightarrow e \overset{\text{fp}}{\hookrightarrow}_o e'$$

$$\rightarrow \exists m'\ T, \quad m \Downarrow^{prog}_{(Out\_normal, T)} m' \quad \bigwedge e' \overset{env}{\sim} m' \quad \bigwedge o \overset{output}{\sim} T$$

Flight Plan Extension



Verification of the generator:

- Verification of the Extend pass : **DONE**
- Verification of the Clight generation pass: **To be done**

Conclusion

Conclusion

Summary:

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

Perspectives:

- Verification of the generation pass,
- 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

Execution of abstracted functions

Axiom: Execution to Trace

$$
\begin{aligned}
&\forall f\ \ m\ m'\ out\ \ T,\\
&\quad m \Downarrow^{(SCALL\ \ f)}_{(out, T)} m'\\
&\quad \rightarrow\ \ m = m'\\
&\qquad\quad \wedge\ out = \texttt{Out\_normal} \wedge T = [SYS\_CALL\ f]
\end{aligned}
$$