A Verified UAV Flight Plan Generator

Baptiste Pollien
ISAE-SUPAERO
University of Toulouse
Toulouse, France
baptiste.pollien@isae-supaero.fr

Christophe Garion
ISAE-SUPAERO
University of Toulouse
Toulouse, France
christophe.garion@isae-supaero.fr

Gautier Hattenberger
ENAC
University of Toulouse
Toulouse, France
gautier.hattenberger@enac.fr

Pierre Roux
ONERA
Toulouse, France
pierre.roux@onera.fr

Xavier Thirioux
ISAE-SUPAERO
University of Toulouse
Toulouse, France
xavier.thirioux@isae-supaero.fr

***Abstract*—`FPL` is a domain specific language used to specify complex drone missions for the Paparazzi open-source autopilot. `FPL` missions are compiled into C code that is directly embedded into the autopilot code. The `FPL` to C code generator, currently written in OCaml, is therefore a critical component when addressing the drone safety. This paper presents the formal verification of the `FPL` compilation process. First, we have developed in Coq a new three-pass code generator, targeting the Clight intermediate language from the CompCert suite. We have then formally defined an operational semantics for `FPL`. Finally, we have proved a bisimulation relation between `FPL` semantics and Clight semantics. In the course of the formalization and verification process, we have also unveiled several problems in the original Paparazzi code generator.**

***Keywords*—Code Generation, Compilation, Mechanized proof, Operational semantics**

I. Introduction

Paparazzi [1] is an open-source autopilot under GPL license developed at ENAC[1] since 2003. Paparazzi offers a variety of customization settings in order to support different types of drones or hardware, but also to define specific missions. These missions, or *flight plans*, are expressed using an XML-based domain specific language, denoted by `FPL` in this paper.

Flight plans are complex missions and `FPL` offers common imperative features to define them, such as mutable variables, exceptions or loops. `FPL` also offers some specific navigation primitives like “go to a position” or “do a circle around a location”. For instance, the following mission can be described in `FPL`: “when the drone is started, it first needs to initiate its sensors and wait for the GPS connection to be established. Then the drone should take off and circle around a certain GPS position to acquire data. During the flight, if the battery level is below 20%, the drone must automatically go back to the `Home` position and land.”

Paparazzi currently offers a code generator that takes as input an `FPL` flight plan and generates a C file that must be compiled together with the autopilot to be embedded in the drone. The flight plan thus cannot be changed during a flight. However, the drone operator can interact with the flight plan and change its execution order. The generated C code file is mainly composed of a step function, named `auto_nav`, which is called periodically by the autopilot to compute the next steps of the flight plan to be executed.

The generator is written in OCaml and Paparazzi users may not entirely trust the generated code. In particular, we may wonder if a) the function computing the next steps to be executed in the flight plan always terminates and b) the generated C code behaves as described by the flight plan. These questions are crucial as the produced C code is intented to be directly embedded in the drone.

Verifying that the embedded C code is correct, i.e. that it behaves as prescribed by the flight plan, can be seen as a *compiler verification* problem: we must prove that the compiler translating `FPL` flight plans into C code guarantees the correctness of the embedded code. Even if compiler verification is a known problem [2], [3], some advances have recently been done using proof assistants, particularly the Coq proof assistant. Coq allows one to write programs in the Gallina functional programming language, to formally prove properties on such programs using powerful tactics, and to extract Gallina programs into OCaml programs that are semantically equivalent [4]–[6]. The main steps of the verification of a compiler with Coq may be summarized as follows: first, express the semantics of both the source and the target languages in Gallina, then write the compiler in Gallina and finally prove a semantics preservation theorem establishing that the produced code has the same behavior as the source code according to their respective semantics. The CompCert C compiler [7] and the Velus Lustre compiler [8], [9] are recent *tours de force* showing that this approach can be applied to large subsets of real programming languages. Notice that there is also an on-going work [10] using Coq to verify a compiler for Esterel, an imperative synchronous language, closer to `FPL` than the previous ones.

This paper presents an extension of `FPL` bringing new

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

[1]The French Civil Aviation University

features requested by Paparazzi users, a formal semantics for this extended language and the development of a new verified compiler from FPL to C code following the previously described approach. The extension of FPL is conservative and does not invalidate flight plans defined with the original DSL.The syntax and semantics of FPL are expressed in Gallina and we use as target language Clight, the Gallina structure used in CompCert to represent C code, providing us an already defined C semantics and a correct C pretty printer. We then write a three-pass compiler in Gallina and formally prove that the compiler preserves FPL semantics and that the step function (i.e. a call to the auto_nav function) always terminates. This new compiler produces C code similar to the one produced by the current compiler in order to be fully compatible with Paparazzi global architecture and APIs. The differences between the C code generated with the new generator and the previous one are minor and are mainly due to limitations of the Clight syntax (see Section III-A5).

The paper is organised as follows: Section II introduces FPL with its new functionalities, gives a glimpse of its semantics through the execution of an example and then defines its formal semantics. Section III presents the architecture of the new three-pass compiler and describes the sketch of the preservation proof of the compiler as well as choices made to simplify its writing. Section IV presents the lessons learned and the problems faced during the development of the generator. Finally, Section V concludes and gives some perspectives. All source files are publicly available at https://gitlab.isae-supaero.fr/b.pollien/vfpg/-/tree/formalise-2023.

II. FPL: Flight Plan Language

Paparazzi uses a Domain Specific Language (DSL) to describe flight plans, with an XML concrete syntax. FPL is a conservative extension of this language that offers new features such as a protection mechanism against unwanted behaviors, either specified in the flight plan or by the drone operator.

A FPL file[2] is divided into two sections: the *flight plan header* and the *core* part. The *flight plan header* contains definitions and meta-information. It is composed of several sections: the header (arbitrary C code added in the header of the generated C file), waypoints (a list of constant points defined using GPS positions or relative coordinates), sectors defined using lists of waypoints, and local variables that can be used by the flight plan[3].

The *core* is the main part of a flight plan and defines the different parts of the mission. In the following, when talking about a flight plan, we will only refer to its *core* part. Section II-A presents the syntax of FPL and an informal description of its semantics. Section II-B provides an intuitive example of a flight plan execution. Section II-C presents the new features we added in the language. Section II-D introduces FPL formal semantics. Finally, Section II-E presents the structure of the generated C code.

[2]The project contains several flight plan examples such as tests/regression-tests/full_example.xml for instance.

[3]The value of these variables can be updated by operators during a flight.

A. Syntax and informal semantics of FPL

```
flight_plan ::= {| excpts : list fp_exception, blocks : list fp_block |}

fp_exception ::= {|                    fp_stage ::=
  cond : c_cond,                         WHILE (cond: c_cond)
  id : block_id,                                (body: list fp_stage)
  exec : option c_code                   | SET (var: var_name)
|}                                             (value: c_value)
fp_block ::= {|                          | CALL (fun: c_code)
  id : block_id,                         | DEROUTE (idb: block_id)
  stages : list fp_stage,                | RETURN (reset: bool)
  excpts : list fp_exception             | NAV (mode: fp_nav_mode)
|}                                             (init: bool)
```

Fig. 1. Flight Plan (FP) Gallina structure representing FPL.

Figure 1 presents the FP Gallina structure that corresponds to the FPL grammar[4]. We denote by *c_code* any valid C code, *c_value* any C expression, *c_cond* C code that can be evaluated as a boolean expression and *var_name* the name of a C variable. The parsing of FPL into FP is done by a pre-processor that is presented in section III-A1.

A flight plan is composed of at least one block and possibly exceptions. A block describes a part of the mission (e.g. "take off" or "initialize sensors") and is composed of a unique id, potential local exceptions and a list of atomic instructions called *stages*. The **WHILE** stage is a classic imperative loop. The **SET** stage assigns value to var. The **CALL** stage executes the C code fun. The **DEROUTE** stage changes the block being currently executed to block idb (the position in the block and the id of the block before the deroute are memorized as the *last* position). The **RETURN** stage returns to the block that was executed prior to the last deroute stage or exception. Its reset parameter allows the user to choose whether the execution should start at the beginning of the *last* block or at its latest stage reached when the deroute or exception occurred. Finally, the **NAV** stage executes the navigation code corresponding to the primitive (e.g. GO to a position or do a CIRCLE) depending on the value taken by the parameter mode[5].

Exceptions constitute a protection mechanism to avoid unwanted behaviours. An exception is raised when its condition cond is evaluated to true. The execution of the flight plan then proceeds to the block id and, if specified, the code exec is called. For example, if the drone has less than 20% of battery left, the flight plan must execute the block that lands the drone at the *Home* position. An exception can be global and will be tested at every call to the auto_nav function or local to a block and will be tested only when the block is executed.

Finally, notice that a flight plan may contain arbitrary C code calls, for example, in conditions or in **CALL** stages. We will discuss the implication of the presence of such code in section II-D1.

[4]Available in the FlightPlan.v file.

[5]This parameter is defined by the sum *fp_nav_mode* type where each constructor correspond to a navigation primitive, see the FPNavigationMode.v file.

Example of a flight plan:

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

A possible execution of the `auto_nav` function:

Call	Current Block	Code Executed

1	0	`InitSensors()`
		`!GPSFixValid()` ⇑ `true`

2 → 8	0	`!GPSFixValid()` ⇑ `true`

9	0	`!GPSFixValid()` ⇑ `false`
		`home = GPSPosHere()`

10	1	`StartMotors()`
11 → 19	1	`TakeOffDone()` ⇑ `false`
20	1	`TakeOffDone()` ⇑ `true` `Deroute` → 10
21	10	...
⋮	⋮	⋮

Fig. 2. Example of the execution of a flight plan.

B. A flight plan execution example

Execution of flight plans is similar to execution of programs written in typical synchronous languages such as Lustre [11] or SCADE [12] that are commonly used in embedded code. The execution of synchronous code is composed of an initialisation phase followed by periodic calls to a step function. The initialisation phase, for flight plan execution, sets the environment in order to start the execution at block 0. Executing the flight plan then consists in calling regularly the `auto_nav` function. The first call to `auto_nav` executes the first block of the plan and stages inside the block are to be executed. There are 2 types of stages: *continue* stages and *break* stages. The forms of the stages are defined statically and implicitly by the semantics. Executing a block consists in executing its stages sequentially in their definition order as long as they are *continue* stages. The execution of the `auto_nav` function terminates when a *break* stage is executed. In this case, the execution of the flight plan pauses and is resumed by the next call to `auto_nav`.

Figure 2 presents an example of a flight plan and one of its possible executions. This flight plan contains no exceptions and several blocks of which only 3 are represented. Let us present briefly the execution of the `auto_nav` function on this example. The first block (block 0) is entered and its first stage is executed. The **CALL** stage is a *continue* stage: its execution simply calls the C code it contains and the next stage is executed. The next stage is a **WHILE** stage, therefore the condition of the loop, i.e. "`!GPSFixValid()`", must be evaluated. Assume that this evaluation returns `true`. In this case, the **WHILE** stage is a *break* stage: it ends the execution of the `auto_nav` function. The flight plan will continue its execution at the next call to `auto_nav` during which the execution of the flight plan is resumed where it was left, i.e. inside the **WHILE** loop. As the loop body is empty in this example, there is no stage to execute and the condition of the loop is reevaluated.

The execution of `auto_nav` continues to evaluate the loop condition until it is evaluated to `false`. Assume that the loop condition evaluates to `false` in call 9. In this case, the **WHILE** stage is considered as *continue*. The last stage **SET** is therefore executed and as it is a continue stage, its execution terminates the execution of the block, stopping `auto_nav` execution and the next block (numbered 1 here) will be executed at next iteration.

The **NAV** stage execution is translated into a call to a C function that sets navigation parameters depending on the navigation primitive used. These parameters will be used by the autopilot and translated into orders for the motors. In this example, the **NAV** stage runs the navigation primitive `TakeOff`. This stage requires an initialisation step as the `init` parameter is set to `true`. There is thus some specific code that will be called at the first execution of the stage (here `StartMotors` that does not appear in the flight plan but is specified in Paparazzi autopilot and added by the generator). Then, the C code corresponding to the second part of the `TakeOff` primitive is a function `TakeOffDone` that returns a boolean value depending on whether the drone has taken off or not. If the `TakeOffDone` function returns `false`, then the stage is a *break* stage, otherwise **NAV** is considered as a *continue* stage. Notice that the navigation primitive `TakeOff` and the C functions called have been forged for this example. The real C code generated may correspond to several calls to functions that may have non-explicit names. This code is generated from functions found in the `FPNavigationModeGen.v` file.

Finally, when the drone has taken off (call 20), `auto_nav` execution continues and the **DEROUTE** stage is executed. This stage is a *break* stage and the next call to `auto_nav` will execute the corresponding derouted block (block 10 here).

The grammar presented in figure 1 and used in the previous example defines a simplified syntax of `FPL`. For instance, the **CALL** stage has several more parameters: it is possible to specify that the code must be executed until it returns `true` or that the stage must break after the execution. We omit such details to keep the key elements of `FPL` and give an intuitive presentation of its syntax and semantics. The full definition of `FPL` can be found in the `flight_plan.dtd` file.

Blocks and stages are normally executed one after the other in their definition order, but **DEROUTE** statements or raised exceptions can change the currently executed block. Note that a human operator can also manually change the current block during the flight plan execution on Paparazzi control console.

C. New features

`FPL` brings two new features to the language currently used in Paparazzi. First, we add another protection mechanism called *forbidden deroute*. Forbidden deroutes should be specified by the user to prevent the execution of "dangerous" block changes such as jumping directly to a block where the drone motors are cut off from a block where the drone is airborne. A flight plan may contain any number of forbidden deroutes. The syntax of flight plans is extended as follows:

$fp\_fb\_deroute ::= \{|$
  `from` : $block\_id$
  `to` : $block\_id$
  `only_when` : option c_cond
$|\}$

$flight\_plan ::= \{|$
  `fb_drtes` : list $fp\_fb\_deroute$
  `excpts` : list $exception$
  `blocks` : list fp_block
$|\}$

A forbidden deroute describes a deroute from a block `from` to a block `to` that must be either unconditionally forbidden or watched by a `only_when` condition[6].

The second feature we added is a Paparazzi user request about the execution of C code when entering or leaving a block. We add the possibility for users to specify `on_enter` and `on_exit` code for every block. This code is executed when entering (resp. exiting) the block. Like other flight plan details, `on_enter` and `on_exit` are not presented in `FPL` formal syntax to focus on the most interesting elements of `FPL`. Notice however that `on_enter` and `on_exit` are actually handled in our verified implementation.

D. Semantics

Flight plans describe how the drone should behave when flying autonomously. In this section, we define a semantics for `FP` describing system evolution and observable events that happen during execution. The system modeled as an execution state and observable events is introduced in Section II-D1. Section II-D2 presents some relevant inference rules of the semantics. These definitions will then be used in Section III to verify the generator. The Gallina definition of the semantics can be found in the `FPBigStep.v` Coq file.

1) States and traces: A semantics generally defines how a system evolves during its execution from an initial state s to a final state s', but also describes the interactions with the outside world. These external operations are modeled by *traces* or *outputs*. For instance, an imperative programming language often uses, as a state, an abstraction of the computer memory. The corresponding semantics describes how the memory changes during the program execution. Traces thus can be sequences of accesses to an external memory or messages received and sent over a network.

Definition 1 (`FP` state):

[6]Since conditions are pieces of arbitrary C code seen as an abstract type *c_cond*, we use an option type to represent unconditional deroute with `None`, in order to optimize code generation.

$fp\_state ::= \{|$
  `idb`: $block\_id$, `stages`: list fp_stage,
  `lidb`: $block\_id$, `lstages`: list fp_stage
$|\}$

We abstract the real memory state of the flight plan by focusing on key elements as shown in Definition 1: an environment contains the current (resp. previous) position in the flight plan, represented by the current block `idb` and remaining stages `stages` to be executed within this block (resp. `lidb` and `lstages`). The previous position is saved when a deroute or an exception occurs.

When executing a flight plan, several C functions specific to each type of drone (fixedwing, rover, rotorcraft etc) are called. For instance, the `NavCircleWaypoint` function setting the navigation parameters to perform a circle has different implementations for fixedwing or rotocraft drones. The execution of the flight plan can also execute arbitrary C code defined by the user. The semantics will therefore produce outputs that represent calls to such C code that are considered as *external calls*.

Definition 2 (Traces):

$c\_exec ::= \textbf{COND}\ (c\_cond * bool) \mid \textbf{C\_CODE}\ c\_code$
$fp\_trace ::= list\ c\_exec$

A single trace is a value of *c_exec*, i.e. the evaluation of a condition or arbitrary C code. The *bool* field records the value of the condition *c_cond*.

The execution of C code represented in the trace may modify the memory environment of the drone. Specific C functions like the navigation functions are defined outside the flight plan code, therefore their execution does not modify the state of the flight plan. However, arbitrary C code may be introduced by users and we must assume that such code does not modify the state of the flight plan as discussed in section III-C1. The drone memory environment can thus be decomposed into two disjoint parts: a) the memory space abstracted by *fp_state* b) the memory space that can be modified when executing C code calls represented by the fp_trace.

Definition 3 (`FP` environment):

$fp\_env ::= (fp\_state \times fp\_trace)$

An environment contains the current state of the flight plan and a history of the external operations that occurred.

The result of the evaluation of a condition is required to define the semantics, but since such conditions are arbitrary C code that cannot be interpreted, we assume the existence of an $evalc$ function specified as follows.

Parameter 1 (Evalc Function):

$evalc : fp\_env \rightarrow c\_cond \rightarrow (bool \times fp\_env)$

$evalc\ e\ c$ returns a couple $(b,\ e')$ such that evaluating the C code c returns the boolean b. The trace of the environment e is updated by appending $COND\ (c,\ b)$ to it, yielding e'.

The trace history contained in *fp_env* is essential for evaluating conditions. Let us consider an example: evaluating the condition "`Battery() < 80`" twice may produce different results as the drone battery is emptying during flight. But as we chose to not specify its real outside environment, considering a trace history allows us to support such cases and to represent C function calls with side effects.

2) Inference rules describing flight plan semantics: We have defined an initial *fp_env* noted e_0 and a *step* function representing a usual big-step semantics evaluation function [13], [14]. The state e_0 contains an empty trace and refers to the first block of the flight plan. Executing the flight plan then consists in calling regularly the `auto_nav` function. The *step* function represents the execution of a call to `auto_nav`. We emphasize the fact that during this execution, the flight plan is not run to completion, but current stage and/or current block are modified and C code may be called. The resulting environment will be ready for another execution step. In the following, in addition to each Notation definition, we provide its Gallina name and a clickable link to its definition in the source code.

Notation 1 (Step function, `step`*):*
The *step* function describes the execution of a call to the navigation function:

$$step : flight\_plan \rightarrow fp\_env \rightarrow fp\_env$$

$step\ fp\ e =\ e'$ noted $e \underset{fp}{\hookrightarrow} e'$ states that e' is the resulting environment after the execution of the flight plan *fp* starting from the environment e. The definition of the function using this notation is presented in the Figure 3[7]. In the following, we will manipulate interchangeably *fp_env* as one variable or a couple of *fp_state* and *fp_trace* (see Definition 3).

The execution of a flight plan can be decomposed into two phases. First, all exceptions are checked and if one of them is raised, then the environment is derouted to a safe block, otherwise the remaining stages are executed. We therefore first present the semantics to manage exceptions, to deroute to a block and then to execute stages.

Notation 2 (Exceptions semantics, `exception`*):*
$e \xhookrightarrow[fp]{\text{exceptions}} e' \Downarrow res$ holds iff the test of the *fp* exceptions starting from e terminates in environment e' and res is `true` iff a global or local exception has been raised.

Notation 3 (Deroute semantics, `goto_block`*):*
$e \xhookrightarrow[(fp,id)]{\text{goto\_block}} e'$ holds iff the deroute from environment e to block id generates a new environment e'. The new environment e' memorizes, as the previous position, the current position of e and its current position points to block id iff the deroute is not forbidden. If the deroute is forbidden, the new environment points to the same position as the environment e with an updated *fp_trace* if some conditions of the forbidden deroutes have been evaluated.

Notation 4 (Stages execution semantics, `run_step`*):*
$(s,\ t) \xhookrightarrow[fp]{\text{stages}} e'$ holds iff the execution of *fp* starting from the environment $(s,\ t)$ with the list s.`stages` remaining to be executed terminates in environment e'. If there are no remaining stages, the execution continues to the next block for the next call to the $step$ function.

We explain in the following the inference rules presented on Figure 3. We will denote by $var\{f := v\}$ a record similar to var but with value v for field f and $id_{db}(fp)$ the default block index of the flight plan *fp* which is the last block of the flight plan.

Inference rule (IR-1) describes what happens when an exception is raised and rule (IR-2) describes what happens when no exception is raised.

Inference rule (IR-3) describes what happens when there is no stage to execute but the current block is not the last block of the flight plan. In that case, the next block to be executed is the one following the current block in the flight plan. Otherwise, the current block becomes the default block and we stay there (c.f. rule (IR-4)).

The most interesting part of the semantics concerns execution of stages. When there are remaining stages to be executed in the current environment, they are executed in their definition order until a *break* stage is executed. We focus on the presentation of the differences for stages **SET**, **WHILE** and **NAV** that are representative of all stages, therefore without loss of generality.

The **SET** stage (rule (IR-5)) is a *continue* stage that simply assigns a value to a variable. The assigned value is arbitrary C code and cannot be analyzed. The assignment is thus added in the trace.

The **WHILE** stage executes a list of stages while a condition holds. The condition is evaluated and if it holds, the body of the loop is added at the beginning of the stages list to be executed and the execution is stopped and the final environment is updated (rule (IR-6)). The **WHILE** stage is kept in the stages list in order to evaluate the condition and possibly iterate the loop after having executed its body. When the loop condition is `false`, the **WHILE** stage is consumed and the execution continues (rule (IR-7)).

The navigation stage is designed to encapsulate the behaviour of all navigation primitives (`CIRCLE`, `GO`, etc). Some of them require an initialisation step, described by the $init$ parameter in `FP`. We note $e \xhookrightarrow[(fp,mode)]{\text{nav}} e'$ the semantics representing the execution of the **NAV** stage with parameter $mode$ from the state e to e'. We note $init\_code\ mode$ the initial code to be executed for mode $mode$.

When $init$ is set to `false`, the navigation code is directly executed as presented in rule (IR-8). In the other case, an initialisation step is required first, before executing the navigation code. Rule (IR-9) describes this behaviour: $init\_code$ is executed but the navigation stage is not consumed and is modified by setting its $init$ parameter to `false`.

The other inference rules of the semantics are similar and can be found inside the `FPBigStep.v` file that contains the Gallina definitions.

E. Generated C code

Figure 4 presents the C code generated from a simple `FP` flight plan with one block and two **CALL** stages. We denote by

[7]Notice that in the Coq development, the semantics is implemented as a computable function (see `FPBigStep.v`) but we present it as a set of inference rules for readability.

$$\frac{e \xhookrightarrow[fp]{\text{exceptions}} e' \Downarrow \texttt{true}}{e \xhookrightarrow[fp]{} e'} \text{(IR-1)} \qquad \frac{e \xhookrightarrow[fp]{\text{exceptions}} e' \Downarrow \texttt{false} \quad e' \xhookrightarrow[fp]{\text{stages}} e''}{e \xhookrightarrow[fp]{} e''} \text{(IR-2)}$$

$$\frac{s.\texttt{stages} = [\,] \quad s.\texttt{idb} < id_{db}(fp) \quad (s,\ t) \xhookrightarrow[(fp, s.\texttt{idb}+1)]{\text{goto\_block}} e'}{(s,\ t) \xhookrightarrow[fp]{\text{stages}} e'} \text{(IR-3)}$$

$$\frac{s.\texttt{stages} = [\,] \quad s.\texttt{idb} \geq id_{db}(fp) \quad (s,\ t) \xhookrightarrow[(fp, id_{db}(fp))]{\text{goto\_block}} e'}{(s,\ t) \xhookrightarrow[fp]{\text{stages}} e'} \text{(IR-4)}$$

$$\frac{s.\texttt{stages} = \textbf{SET}\ (var, value) :: stages' \quad s' = s\{\texttt{stages} := stages'\} \quad (s',\ t +\!\!+ [\textbf{C\_CODE}\ (var = value)]) \xhookrightarrow[fp]{\text{stages}} e'}{(s,\ t) \xhookrightarrow[fp]{\text{stages}} e'} \text{(IR-5)}$$

$$\frac{s.\texttt{stages} = \textbf{WHILE}\ (cond, body) :: stages' \quad evalc\ (s,\ t)\ cond\ =\ (\texttt{true}, (s',\ t')) \quad s'' = s'\{\texttt{stages} := body +\!\!+ s.\texttt{stages}\}}{(s,\ t) \xhookrightarrow[fp]{\text{stages}} (s'',\ t')} \text{(IR-6)}$$

$$\frac{s.\texttt{stages} = \textbf{WHILE}\ (cond, body) :: stages' \quad evalc\ (s,\ t)\ cond\ =\ (\texttt{false}, (s',\ t')) \quad (s'\{\texttt{stages} := stages'\},\ t') \xhookrightarrow[fp]{\text{stages}} e''}{(s,\ t) \xhookrightarrow[fp]{\text{stages}} e''} \text{(IR-7)}$$

$$\frac{s.\texttt{stages} = \textbf{NAV}\ (mode, \texttt{false}) :: stages' \quad (s\{\texttt{stages} := stages'\},\ t) \xhookrightarrow[(fp, mode)]{\text{nav}} e''}{(s,\ t) \xhookrightarrow[fp]{\text{stages}} e''} \text{(IR-8)}$$

$$\frac{s.\texttt{stages} = \textbf{NAV}\ (mode, \texttt{true}) :: stages' \quad s' = e\{\texttt{stages} := \textbf{NAV}\ (mode, \texttt{false}) :: stages'\}}{(s,\ t) \xhookrightarrow[fp]{\text{stages}} (s',\ t +\!\!+ [init\_code\ mode])} \text{(IR-9)}$$

Fig. 3. Inference rules for FP.

GEN_DEFAULT_C_CODE the code generated for the default block added by the pre-processing[8] at the end of the list of blocks. The get_nav_block()/get_nav_stage() functions return the value of the nav_block/nav_stage variables, i.e. the ids of the current block/stage being executed.

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

```
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()
  }
}
```

Fig. 4. The C auto_nav function generated from an simple FPL file

The auto_nav function is mainly composed of a switch statement in which every case statement corresponds to the treatment of a block. Each block treatment also consists in a switch statement called *stage switch*. Each case of a stage switch corresponds to the execution of one stage. The previous example shows two **CALL** stages which are *continue* stages, therefore the generated case statements do not contain a break statement: if stage 0 is executed, when the function func1 has been executed then the function func2 will be called. Otherwise, *break* stages have a break statement and the stage switch will be exited when encountering such a stage, ending auto_nav execution. It is worth noting that the structure of the C code generated for the stages of a block may be different from the FP structure. In the example presented on Figure 4, every C case statement corresponds to a FP stage, but there are stages that may produce several case statements. For instance, navigation stages may require an initialisation step, depending on the parameter init, that is added as an extra case statement in the generated code.

[8]The pre-processing is one of the steps for the code generation that will be detailed in Section III-A.

III. Verified Generator

This Section presents our new generator and how we verified it. Section III-A presents the new architecture of the verified generator. Section III-B presents a sketch of the semantics preservation theorem that has been proved. Finally, we discuss in Section III-C the hypotheses made and the axioms used to verify the generator.

A. Code generator architecture

Figure 5 presents the code generator architecture. First, the FPL input file is parsed by a pre-processor written in OCaml and some tranformations are performed to generate the FP Gallina structure representing the flight plan. The core of the generator, written in Gallina, then translates FP into Clight code which is then printed using an OCaml post-processor to obtain the final C code file to be embedded in the autopilot. The FP to Clight generator is split into three passes:

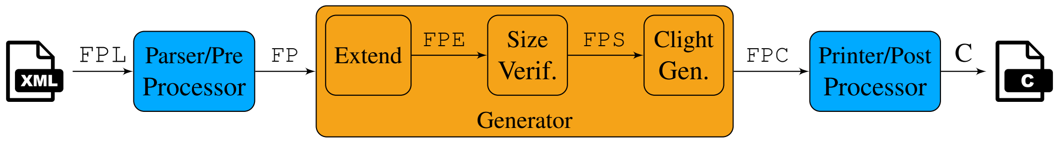

Fig. 5. Architecture of the generator.

an *extension* pass, a *size verification* pass and a *generation* pass. The extension pass transforms the FP structure into an *extended* version FPE closer to the C code to be generated. The *size verification* pass performs some size verification on the FPE structure and may return errors. If the FPE is well-sized, this pass returns a structure called FPS. The generation pass finally converts FPS into Clight code.

We should ideally verify that the whole generator is correct. Unfortunately, some operations such as reading and writing files are difficult to implement in Gallina, therefore the generator frontend and backend are currently written in OCaml. However, we are confident that these pre-processing and post-processing phases are correct as they only perform minor transformations. The code for these two phases can be found in the OCaml files preproc.ml and postproc.ml.

1) Pre-processor: The first role of the pre-processor is to convert the FPL input file into a FP Gallina structure.

The flight plan header is pre-processed with simple transformations, e.g. coordinates are converted into different formats (UTM, LLA etc). The main transformations happen on the core part of the flight plan: a safety block is added and macro stages are expanded. The safety block is a block that sends the drone to land at its take-off position. It is added at the end of the flight plan in order to be called only when the original flight plan has been completed. Macro stages are "syntactic sugar" that are translated to atomic stages: **FOR** loops are converted to **WHILE** loops and **PATH** stages that take a list of waypoints to be followed by the drone are replaced by a list of **NAV** primitive GO.

The final transformation indexes the blocks initially referenced in FPL using names. The pre-processor verifies that every block reference corresponds to a position of a defined block in the FPL file (starting at index 0) and replaces each block name with its index. The pre-processor then generates a C file header composed of definitions (constants for the waypoints coordinates, block names, security height, etc) collected during the pre-processing phase.

2) Extension pass: This pass transforms FP structures into FPE, an extended flight plan which is an intermediate representation closer to the final C code structure[9]. As presented in Section II-E, the FP structure contains stages that correspond to several case statements in the generated C code, such as the **NAV** stage that requires an initialisation stage. Some constructs like **WHILE** use lists of nested stages to represent their bodies, however the *stage switch* only occurs once. Moreover, all stages are referenced with indexes in the generated C code but FP does not contain such indices and its semantics proceeds by maintaining a sequence of remaining stages to be executed. FPE has been defined to fill this gap between FP and C code and the differences between FP and FPE only concern stages.

[9]The Coq definitions are available in the FlightPlanExtended.v file.

The extension pass is realized by the extend_stages_-default function numbering the stages and linearizing the list of stages. For instance, the **WHILE** stage will be unfolded by appending the loop body to the main list of stages. New stages will be also added like $\textbf{NAV\_INIT}_e$ that corresponds to the case where a **NAV** requires an initialisation stage or $\textbf{END\_WHILE}_e$ added after the loop body used to generate a *switch case* in the generated C code that will restart the loop.

3) Size verification pass: During the pre-processing phase, flight plans that are syntactically incorrect are rejected, but no guarantee is obtained on the block numbering process at the Coq level. Moreover, the generated C code aims to be compiled and executed on drones with limited resources. In order to optimise memory usage, integer variables like nav_-block that contains the current block id are coded on 8 bits, therefore limiting the number of blocks and stages to 256. We thus defined a property verified_fp_e, noted H_{ws}, that is satisfied if a FPE is correct: the flight plan does not contain more than 256 blocks, each block is well-numbered and does not contain more than 256 stages, as well as others properties not detailed here.

The size verification pass, realized with the function size_verification, either returns error messages if the flight plan is detected as incorrect or a FPS, a well-sized flight plan. FPS is the subset of FPE flight plans that respect the property H_{ws}. The FPS structure also carries a proof of H_{ws} which will be used to prove the semantics preservation theorem presented in Section III-B1.

The size restriction has been added after finding an issue in the old generator as it did not perform size verification and thus can produce a flight plan with more than 256 blocks while using 8 bits variables to store the current block id. Some blocks cannot therefore be accessed and the computation of the next block id may create an overflow. This problem was not raised before because the size upper bound is not reached in real use cases. Flight plans are in general short as they describe missions that can be realized by small drones with limited batteries.

4) Clight generation function: The final Gallina generation function is a global_definition function taking a FPS flight plan as a parameter and returning a Clight program, noted FPC, containing a global variable nb_block representing the number of blocks of the flight plan and some functions:

`forbidden_deroute` testing if a deroute between two blocks is forbidden or not, `auto_nav` and some auxiliary functions that will not be detailed in this paper.

The functions `extend_stages_default`, `size_verification` and `global_definition` are regrouped inside the `generate_flight_plan` function. This function takes a `FP` flight plan and produces the final Clight code or possible warning[10] and error messages, which are mainly produced during the *size verification* pass.

5) Post-processor: The post-processor produces a compilable C code file from the generated Clight structure. The first part of the file is the header generated by the pre-processor. The second part contains the functions produced by the generator. Notice that we want to keep full compatibility with Paparazzi and therefore be as close as possible to the C code generated by the previous and unverified Paparazzi code generator. Indeed, Paparazzi users may rely on the generated code structure, incidentally or not, to develop their own components. The flight plan is also not run in isolation and accesses a number of global variables in Paparazzi.

The post-processor uses the CompCert `PrintClight` module that translates the Gallina Clight structure into pseudo C code. Unfortunately, this module has mainly been developed for debugging purposes and the produced code cannot be compiled. For instance, when a Clight variable identifier is printed, the $ symbol is added in front of the identifier. The post-processor makes a final pass to convert the printed C code into a compilable one.

The generated C code is very similar to the code generated by the old OCaml generator. The only noticeable differences are due to Clight limitations. For instance, the boolean operator (`&&`) does not exist in Clight and must be replaced by conditional statements. Similarly, expressions like `fun1(fun2())` cannot be defined in Clight and must be split into two statements, using a temporary variable to store the result of calling `fun2()`.

B. Verification of the generator

The verification of the generator is divided into two parts: we first prove that the generator behaves correctly and preserves semantics and then prove that the code generated for the `auto_nav` function always terminates.

1) Semantics preservation: Let us first present the abstract semantics of a flight plan.

Definition 4 (Abstract semantics):

```
fp_semantics ::= {|
  flight_plan_type:     $Type$,
  env:                  $Type$,
  flight_plan:          flight_plan_type,
  init_env:             env $\rightarrow$ $Prop$,
  step:                 env $\rightarrow$ env $\rightarrow$ $Prop$
|}
```

`fp_semantics` describes the semantics of a flight plan: the execution begins with an initial environment e_0 (i.e. `init_env` e_0 is satisfied) and every call to the `auto_nav` function is represented by the `step` predicate. A `fp_semantics` structure has been defined for `FP`, `FPE`, `FPS` and `FPC`. The step function for `FPC` corresponds to the execution of a call to `auto_nav` using the Clight semantics [15]. The environment for `FPC` semantics is composed of a list of Clight traces and a memory environment that contains at least the same information than the *fp_state*. In the following, we will denote by "`fp_sem` fp" the `fp_semantics` for the flight plan fp and "`fpc_sem` cfp" the `fp_semantics` for the Clight program cfp.

The idea of semantics preservation consists in "executing" through the corresponding formal semantics both the source program and the generated program starting from two equivalent environments and to verify that both executions terminate in equivalent states. In order to do so, we define a matching relation $\overset{\text{env}}{\sim}$ between source and target environments. Two environments are matched through the relation $\overset{\text{env}}{\sim}$ if they represent the same flight plan environment.

We formalise our verification problem as a standard *bisimulation* problem, i.e. we exhibit a forward simulation and a backward simulation between both semantics.

Definition 5 (Simulation):

```
simulation $FP1$ $FP2$ ($\overset{\text{env}}{\sim}$) ::= {|
  match_initial_envs:
    $\forall e_1$, $FP1$.init_env $e_1$
    $\rightarrow \exists e_2$, $FP2$.init_env $e_2 \wedge e_1 \overset{\text{env}}{\sim} e_2$,
  match_step:
    $\forall e_1 e_1'$, $FP1$.step $e_1$ $e_1'$
    $\rightarrow \forall e_2$, $e_1 \overset{\text{env}}{\sim} e_2$
    $\rightarrow \exists e_2'$, $FP2$.step $e_2$ $e_2' \wedge e_1' \overset{\text{env}}{\sim} e_2'$
|}
```

Definition 6 (Bisimulation):

```
bisimulation $FP1$ $FP2$ ($\overset{\text{env}}{\sim}$) ::= {|
  forward_sim: simulation $FP1$ $FP2$ ($\overset{\text{env}}{\sim}$),
  backward_sim: simulation $FP2$ $FP1$ ($\overset{\text{env}}{\sim}$)
|}
```

The definitions of `fp_semantics` and `bisimulation` can be found in the `FPBigStepGeneric.v` file.

Finally, we define the semantics preservation theorem for the flight plan generator, i.e. a bisimulation between the semantics of `FP` and `FPC`. We note $\overset{\text{cenv}}{\sim}$ the matching relation between *fp_env* and the environment of the `FPC` semantics.

Theorem 1 (Semantics Preservation):

$\forall fp\ cfp$, $cfp = generate\_flight\_plan\ fp$
  $\rightarrow$ `bisimulation` (`fp_sem` fp) (`fpc_sem` cfp) ($\overset{\text{cenv}}{\sim}$)

This theorem states that if the C code *cfp* can be generated from the flight plan *fp*, then there is a bisimulation between *fp* and *cfp*. The forward simulation property states that any execution of *fp* can be simulated by the C code generated: no behavior described by the semantics is lost during the generation. Reciprocally, the backward simulation property states that any Clight execution of the `auto_nav` function corresponds to a behavior of the `FP` semantics: all executions are allowed by the semantics.

The verification of Theorem 1 in Coq has been divided into smaller proofs by proving bisimulation for the different passes of the generator, i.e. between `FP-FPE` (see `VerifFPToFPE.v`), `FPE-FPS` (see `VerifFPEToFPS.v`)

[10]Warnings will be discussed in Section V.

and FPS-FPC (see VerifFPSToFPC.v). These proofs were finally combined using the bisimulation composition property. The global proof of the theorem 1 can be found in the Coq file VerifFPToFPC.v.

2) Termination: We want to ensure that the autopilot can call the auto_nav function without the risk of being blocked because of an infinite loop. Using Theorem 1, the termination proof of the C code is equivalent to ensure that the FP *step* function terminates (see Section II-D). Since Coq does not allow to define non-terminating functions, the definition of the *step* semantics function of FP in Gallina proves the termination of the auto_nav function. The implementation of this function can be found in the Coq file FPBigStep.v.

C. Verification hypotheses

The proof of the semantics preservation theorem is based on a modeling of our system and some hypotheses. The goal of this section is to present these choices in order to give confidence in the new verified generator. Section III-C1 summarizes the model used to define the main theorem. Section III-C2 presents all axioms defined to verify this theorem.

1) Models of the system: The drone autopilot is a complex system evolving in a real outside environment. The auto_-nav function interacts with the drone autopilot and indirectly with the outside environment. An ideal property we may want to prove is that the drone will always behave as defined in the flight plan. Unfortunately, proving this property would require to represent concretely the interaction between the flight plan, the drone autopilot, and the outside environment which is complex and time-consuming. Instead, we proved that the execution of the auto_nav function will interact with the drone autopilot as defined in the flight plan without any further guarantees that the autopilot will behave correctly. As presented in Section II-D1, we model the drone environment by two elements: *fp_state* that represents the internal states of the flight plan and *fp_trace* that represents the interactions between the flight plan and the autopilot. These interactions can be calls to functions of the autopilot but also to arbitrary C code defined by the user (for example the code executed in **CALL** stages). We thus assume that they will not modify the internal state of the flight plan.

We also add another hypothesis for proving the termination of the auto_nav function. As the execution of the flight plan depends on the execution of arbitrary C code that cannot be verified *a priori*, the termination of the auto_nav function holds under the condition that arbitrary C code eventually terminates, which we consider established by other means[11].

2) Axioms: We use a parameter function called create_-ident that takes a string and produces a Clight ident during the code generation pass. Clight idents are used in the Clight syntax to describe variable or function names. This function is a Coq Parameter, i.e. it is defined in OCaml and linked during the extraction. It has to be defined in OCaml as the corresponding string is stored in a hashtable. The PrintClight function uses the hashtable to print the corresponding name of variables and functions. As this function is defined outside Coq, it is not possible to prove properties on it. We thus add an axiom stating that the create_ident function is injective which is reasonable knowing the OCaml implementation. The declaration of the function and the axiom can be found in the ClightGeneration.v file.

[11]The WCET analysis presented in [16] shows that on a real-world example, the function auto_nav always terminates in less time than its call period.

The step property of FPC semantics executes symbolically the Clight code of the auto_nav function from an initial memory state and terminates in a final memory state. During program execution, the Clight semantics produces a list of trace that corresponds to the *fp_trace* of fp_env. The list of trace must thus be generated by the execution of the arbitrary C code of the flight plan. In the Clight semantics, the traces are only generated for statements executing calls of external or built-in functions. However, arbitrary C code in Paparazzi is not restrained to these constructions. For instance, such code may appear in the condition of a conditional statement. Moreover, there is no way to use results of the $evalc$ function in the Clight semantics. We thus define 4 axioms that are available in the ClightLemmas.v and CommonFPVerification.v files. Roughly speaking, these axioms can be seen as an extension of the Clight semantics that takes into account the model defined in the previous section by mainly adding two cases: 1) the call to a void function produces a trace, 2) the call to a function with a boolean result will return a value using $evalc$ and such a call also produces a trace.

Using these axioms, we do not need to use the external call mechanism of Clight. We thus define another axiom stating that the execution of external code from the same state always produces the same trace (see the FPBigStepClight.v file). This axiom allows to prove that the Clight semantics is deterministic. For the proof of Theorem 1, we first proved the forward simulation and then we used the property that Clight is deterministic in order to prove the backward simulation.

Finally, we use some classical axioms from Coq standard library such as excluded middle, proof irrelevance (two proofs are equal if they prove the same property) and functional extensionality (equality of functions is pointwise equality). These axioms are particularly necessary when manipulating dependent types, see Section IV-B.

IV. Lessons learned

During the development of the project, we have learned several lessons we think valuable for those wanting to prove a compiler with Coq. Section IV-A presents some feedback on development process in Coq and Section IV-B presents some technical remarks about using Coq, Clight and MathComp.

A. Development methodology

The verification of a compiler using a proof assistant like Coq is a well-tested and tried technique. Some of the projects using it are mature enough to be used in critical software development (see CompCert [7] for instance). Even if its complexity cannot be compared to projects like CompCert or Velus, the FPL code generator is not a toy example.

The context of the Paparazzi autopilot has constrained our formal development in several ways. First, the FPL input language was fixed and already in use, so we were obliged to keep its quirks[12] and propose only conservative extensions, instead of a global redesign that could have been helpful in order to ease the verification effort. Second, we were also tied to a specific C code structure that users may rely upon. Third, we needed as soon as possible to devise an executable semantics and present it to the original FPL designers for validation and feedback, which led us to a simple FP semantics without a formal model of external C code.

Developing such a project with Gallina forced us to a deep clarification of some rather tedious semantic details and incidentally unveiled a number of issues in the original compiler. For instance, writing down the proof allowed us to find a bug when there are more than 256 blocks or stages. Also, in the original semantics of FPL based on the previous generator, we found that the **DEROUTE** stage has an unexpected behaviour: when it is executed, the current position is stored in `last_block` and `last_stage`, and the current block becomes the derouted block. Therefore, when there is a **RETURN** stage executed in the derouted block, the execution resumes at the stored position, so the same **DEROUTE** stage is executed again and the program enters an infinite loop. This issue was only found during the formalization of the semantics as the **RETURN** stage is rarely used by Paparazzi users and when it is, it is only to resume the plan execution after an exception.

Finally, verification of the generator by ensuring semantics preservation is really time-consuming and on-the-fly changes must be avoided as much as possible. Indeed, the code generator was split into three independent passes so that the proof effort for each pass is manageable, but also so that at least the different passes are unlikely to change, whatever the development hazards of the other passes.

B. Technical remarks

In order to define the syntax of FP and FPE, we naturally wanted to reuse Coq types and libraries, such as *list*. We thus defined the flight plan as presented in section II-A but we faced a problem when performing induction over the nested recursive *fp_stage* type. The induction principle generated automatically by Coq does not take into account the interplay between *fp_stage* and *list*. We thus had to define our own induction principle: proving a property P for a *fp_stage* requires proving P for all the different nested stages.

In Section II-D, for the sake of readability, the semantics of FP is given through a relational presentation with inference rules, but we decided to implement the semantics in Coq as a *step* function for early validation purposes. An Ocaml interpreter is then obtained by mere extraction of the Coq definition. Having a function also makes it possible to automate and simplify proofs by using normalization by evaluation instead of manually applying many rewrite rules. Another important point is that we wanted to avoid gratuitously assuming new axioms about correctness of non-Gallina phases, such as preprocessing. Therefore we used dependent types (i.e. a base type with some property) associated with verification functions that inject a structure from the base type if its correctness property holds, or return an error otherwise. For instance, we defined the type of natural numbers representable in 8 bits for FPS and also the type of well-sized and well-numbered flight plans (cf. H_{ws} property in Section III-A3).

We used Clight as the output language because it has many advantages. First, it provides a trustworthy semantics for C. However, Clight semantics for external calls does not correspond to our model. We thus have to define our own Clight semantics for arbitrary C code, as presented in Section III-C2. Also, our modelisation choices naturally split the flight plan memory states from the drone states (represented by the trace of events), thus we avoid dealing with separation logic.

Second, CompCert offers the `clightgen` tool that takes a C file and converts it into a Clight structure. This tool can help the user to understand Clight and its differences with C, by translating existing C programs. For instance, the `&&` boolean operator and some other constructions do not exist in Clight (see Section II-E).

CompCert also offers two semantics: a small-step one and a big-step one. On the one hand, the big-step semantics is the natural semantics which can describe the run-to-completion model of a program. This semantics is easier to work with, but unfortunately, it does not define the behavior of all Clight statements such as `goto`, which is used for loops. On the other hand, the small-step semantics describes the behavior of all Clight statements. However, the small-step semantics only describes one step of execution and the remaining code to execute is stored in a continuation. The execution of a whole program then corresponds to a succession of small steps which can be tedious for verifying semantics preservation, as our own semantics is direct and does not involve continuations.

It is also possible to produce C code with the `PrintClight` module of CompCert. However, as presented in section II-E, the module does not produce compilable C code and some transformations have to be performed.

In addition to CompCert implementation, we also used MathComp, a Coq library of formalized mathematics. We used some MathComp lemmas about arithmetics on natural numbers but we mainly used `seq`, a library about lists that provides a lot of useful functions and lemmas like the `drop` function that removes the first elements of a list. We also use SSReflect, a proof language that allows to simplify proofs compared to standard Coq proof tactics. The main problem we faced using MathComp concerns program extraction. Indeed, the MathComp library is not designed to be extracted into Ocaml code as it depends on constructions that are not supported. We thus have to manually specify the only functions that need to be extracted from MathComp.

[12]For instance, the *fp_state* only stores one previous position. If we execute two **DEROUTE** then two **RETURN** stages, then the first return rolls back from the second deroute as expected, but the second return does nothing.

V. Conclusion

This paper presents the development and the formal verification of a new flight plan generator for the Paparazzi autopilot. This generator is implemented in Coq as a three-pass compiler, translating drone missions described in the `FPL` language into C code. This new generator is fully compatible with the previous unproved generator but supports new features such as forbidden deroutes and detects erroneous flight plans which were previously unnoticed. During the early development stages, we developed some tests showing that the C code generated by the two generators behave similarly on several examples. We have defined a Gallina structure representing `FPL` programs with its operational semantics and then proved that the semantics of `FPL` programs is preserved when generating Clight code. The project is composed of 1,3k loc of OCaml for the pre- and post-processing steps and almost 17k loc of Coq with only 12% of working code (the rest are lemmas, definitions and proofs).

As future work, we want to reduce as much as possible the number of unverified transformations, such as the OCaml pre-processing phase. For instance, we might use `menhir` with the `-coq` option to get a formally verified parser. We will also consider supporting new types of error and warning messages. We are currently specifying and implementing warning messages when forbidden deroutes are blocking. We want to warn the users when there is a `deroute` (or potential exceptions raised) to an explicitly forbidden block in the flight plan. When the `deroute` stage is executed, depending on its condition, the change of block may be forbidden and the flight plan may be stuck in a deadlock.

Moreover, we want to simplify the use of the generator. Currently, the generator is a distinct project from the Paparazzi UAV autopilot project. We plan to integrate the verified generator directly into Paparazzi build process to ease its adoption by Paparazzi users. We should also consider connecting our generator to CompCert to have a fully verified compiler that translates `FPL` to assembly code.

Finally, we want to modularize the generator by separating the navigation modes from the flight plan. The idea is to offer users a generic interface to specify their own navigation mode. They will thus have to define the semantics and the code generation function for their navigation modes. If they prove that the semantics is preserved for their navigation modes, then they have a personnalised C code generator. It will then be possible to support easily new autopilots or to use the generator in another context that uses synchronous programs such as finite-state machine.

References

[1] G. Hattenberger, M. Bronz, and M. Gorraz, "Using the Paparazzi UAV System for Scientific Research," in *IMAV 2014, International Micro Air Vehicle Conference and Competition 2014*, Delft, Netherlands, Aug. 2014, pp. pp 247–252. [Online]. Available: https://hal-enac.archives-ouvertes.fr/hal-01059642

[2] M. A. Dave, "Compiler verification: a bibliography," *ACM SIGSOFT Software Engineering Notes*, vol. 28, 2003.

[3] J. Mac Carthy and J. Painter, "Correctness of a compiler for arithmetic expressions," in *Proceedings of Symposia in Applied Mathematics*, ser. Mathematical aspects of Computer Science 1, 1967. [Online]. Available: http://jmc.stanford.edu/articles/mcpain.html

[4] P. Letouzey, "Programmation fonctionnelle certifiée: l'extraction de programmes dans l'assistant Coq," Thèse de Doctorat, Université Paris-Sud, Jul. 2004. [Online]. Available: http://www.lri.fr/~letouzey/download/these_letouzey.ps.gz

[5] M. Sozeau, S. Boulier, Y. Forster, N. Tabareau, and T. Winterhalter, "Coq Coq Correct! Verification of Type Checking and Erasure for Coq, in Coq," *Proceedings of the ACM on Programming Languages*, pp. 1–28, Jan. 2020. [Online]. Available: https://hal.archives-ouvertes.fr/hal-02380196

[6] A. Anand, A. W. Appel, G. Morrisett, Z. Paraskevopoulou, R. Pollack, O. S. Bélanger, M. Sozeau, and M. Z. Weaver, "Certicoq : A verified compiler for coq," 2016.

[7] X. Leroy, S. Blazy, D. Kästner, B. Schommer, M. Pister, and C. Ferdinand, "CompCert - A Formally Verified Optimizing Compiler," in *ERTS 2016: Embedded Real Time Software and Systems, 8th European Congress*. Toulouse, France: SEE, Jan. 2016. [Online]. Available: https://hal.inria.fr/hal-01238879

[8] T. Bourke, L. Brun, and M. Pouzet, "Mechanized semantics and verified compilation for a dataflow synchronous language with reset," in *Proceedings of the 47th ACM SIGPLAN Symposium on Principles of Programming Languages*, ser. POPL'20, vol. 4. New Orleans, LA, USA: ACM, Jan. 2020, p. 29.

[9] L. Brun, "Mechanized semantics and verified compilation for a dataflow synchronous language with reset," Theses, Université Paris sciences et lettres, Jul. 2020. [Online]. Available: https://tel.archives-ouvertes.fr/tel-03068862

[10] G. Berry and L. Rieg, "Towards coq-verified esterel semantics and compiling," *CoRR*, vol. abs/1909.12582, 2019. [Online]. Available: http://arxiv.org/abs/1909.12582

[11] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud, "The synchronous dataflow programming language lustre," *Proceedings of the IEEE*, vol. 79, no. 9, pp. 1305–1320, September 1991.

[12] G. Berry, "Scade: Synchronous design and validation of embedded control software," in *Next Generation Design and Verification Methodologies for Distributed Embedded Control Systems*, S. Ramesh and P. Sampath, Eds. Dordrecht: Springer Netherlands, 2007, pp. 19–33.

[13] H. R. Nielson and F. Nielson, *Semantics with Applications: An Appetizer*, ser. Undergraduate Topics in Computer Science. Springer, 2007. [Online]. Available: https://doi.org/10.1007/978-1-84628-692-6

[14] G. Winskel, *The formal semantics of programming languages – an introduction*. MIT Press, 1993.

[15] S. Blazy and X. Leroy, "Mechanized semantics for the Clight subset of the C language," *Journal of Automated Reasoning*, vol. 43, no. 3, pp. 263–288, 2009. [Online]. Available: https://hal.inria.fr/inria-00352524

[16] G. Hattenberger, F. Bonneval, M. Ladeira, E. Grolleau, and Y. Ouhammou, "Micro-drone autopilot architecture for efficient static scheduling," in *13^{th} International Micro Air Vehicle Conference*, G. de Croon and C. D. Wagter, Eds., Delft, the Netherlands, Sep 2022, pp. 175–182, paper no. IMAV2022-21. [Online]. Available: http://www.imavs.org/papers/2022/21.pdf