Adaptive Variables for Declarative UAV Planning

John Henry Burns, Xiaozhou Liang, Yu David Liu
Department of Computer Science
SUNY Binghamton
Binghamton, NY, USA
{jburns11,xliang24,davidl}@binghamton.edu

ABSTRACT

Unmanned Aerial Vehicles (UAVs) are an important subset of autonomous robotics, offering unique opportunities in domains like merchandise delivery, geographical survey, and disaster recovery. The planning layer of UAVs is made up of high-level directives that instruct the system on how to achieve the plan's goals. UAVs execute their plans in the physical environment, and thus the plans must adapt to changes in the dynamic context. In this paper, we present a simple programming abstraction, adaptive variables, to declaratively define adaptation for UAV flight plans in a dynamic context. Building on top of a declarative language for expressing UAV flight plans, adaptive variables can change during a UAV flight based on predicates over physical data. We implement adaptive variable for Paparazzi and demonstrate its usefulness in adaptive UAV planning with the NPS Simulator.

CCS CONCEPTS

• **Software and its engineering → Domain specific languages**; • **Computer systems organization → External interfaces for robotics**.

KEYWORDS

Unmanned Aerial Vehicles, Flight Plans, Adaptability, Declarative Languages

ACM Reference Format:
John Henry Burns, Xiaozhou Liang, Yu David Liu. 2020. Adaptive Variables for Declarative UAV Planning. In *12th International Workshop on Context-Oriented Programming and Advanced Modularity (COP'20), July 21, 2020, Virtual Event, USA.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3422584.3422763

1 INTRODUCTION

Within the field of autonomous mobile robotics, Unmanned Aerial Vehicles (UAVs) are an emerging platform which greatly enriches the application domain of robotics. Like other autonomous mobile systems, the behavior of UAVs is highly dependent on their *environmental* and *system resource* context. First, UAVs must adapt to environmental changes, such as wind conditions, and potential possibilities of collision. Second, UAVs must adjust their behavior based on the availability of system resources, such as battery level and communication channels. A top priority of UAV design is to build *adaptive systems* that respond to the rapidly changing context.

Existing solutions to UAV programming follow two routes. First, as UAVs can be treated as a form of embedded systems, they can be programmed with low-level languages such as C, or paired with C-based frameworks such as ROS [19]. These solutions can flexibly support adaptive UAV behaviors, but the lack of UAV-specific programming abstractions makes high-level intentions buried in embedded code and complicates program reasoning. Another route is to use high-level declarative frameworks, such as the Paparazzi's Flight Plans [10]. Declarative programming simplifies UAV behavior specification and promotes program reasoning for this family of cyber-physical systems. Both goals are important for UAV software development. However, existing declarative frameworks do not naturally support the variability and dynamism that UAVs require to navigate under complex contexts.

In this paper we take the declarative approach for adaptive UAV planning. We describe a simple and intuitive extension to Paparazzi's Flight Plans, with the focus on its support of adaptability: how the UAVs respond to the potentially rapidly changing context. The centerpiece of our design is the *adaptive variable*, a programming abstraction that allows the state of the program to change upon the satisfaction of its guarding condition. The guarding condition is a predicate over program states that may change as the environment or system resource context changes. As a result, when a UAV carries out its flight plan and flies over a 3D space, the guarding condition may become satisfied on the fly, leading to the change of the adaptive variable and ultimately the flight behavior of the UAV itself.

Imagine the use scenario of a UAV-based geographical survey (GS) for example. GS is a common flight task where a UAV flies over a large (often rectangle) area through the zig-zag pattern. In Paparazzi's existing design, the *resolution* of the survey, i.e., the density of the zig-zag pattern, must be statically set by the programmer as a constant. However, operating in a realistic environment makes battery consumption dynamic. As most commercial UAVs operate with a battery life of less than 60 minutes [9], it is essential for the GS flight plan to be aware of battery level in order to ensure task completion and that the UAV can return home. With adaptive variables, we are able to support *energy aware* GS: the battery level can serve as a guarding condition to determine the resolution of the GS, captured as an adaptive variable.

Our solution, albeit simple, is the first step toward a direction with a distinct philosophy: there is great benefit for UAV software systems to overlay low-level code with a high-level *declarative and adaptive* language design. Broadly, this philosophy is well aligned

COP'20, July 21, 2020, Virtual Event, USA

ACM ISBN 978-1-4503-8144-4/20/07…$15.00
https://doi.org/10.1145/3422584.3422763

with Context-Oriented Programming (COP) [13] in that high-level programming abstractions can improve the safety and maintainability of software systems while retaining adaptiveness and expressiveness. Through this lens, our preliminary design is an instance of declarative COP in an emerging domain.

The rest of this paper is structured as follows. Section 2 will present some background information on UAV planning that is required to understand our implementation and examples. Section 3 will present our design in the context of two motivating examples. In Section 4, we will discuss the implementation details. Section 5 discusses related work, followed by a discussion on conclusion and future work in Section 6.

2 BACKGROUND

```
1  <flight_plan>
2      <waypoints>
3          <waypoint name="p1" x="0" y="100"/>
4          <waypoint name="p2" x="500" y="0"/>
5      </waypoints>
6      <blocks>
7          <block name="Takeoff">
8              ...
9          </block>
10         <block name="Survey">
11             <survey_rectangle wp1="p1" wp2="p2" grid="5"
12             orientation="NS"/>
13         </block>
14         <block name="Land">
15             ...
16         </block>
17     </blocks>
18 </flight_plan>
```

Listing 1: An Example of a Simple Paparazzi Flight Plan

We build our adaptive variables on top of Paparazzi, a widely used open-source software and hardware ecosystem. The majority of Paparazzi's source code is written in C, including both autopilot code to be deployed on the UAVs, and a ground station solution. As UAV hardware components are diverse, Paparazzi supports a highly configurable compilation process to allow for the generation and deployment of autopilot code.

On top of the C code base, Paparazzi supports a high-level XML-based domain-specific definition for flight plans. This high-level plan will eventually guide the compilation process to generate an autopilot control loop that represents the planning layer of the UAV software stack.

As shown in Listing 1, each `flight_plan` consists of two parts: a list of `waypoints` and a sequence of `blocks`. Each waypoint defines a position in 3D space — identified by the `x`, `y`, and an optional altitude `alt` tags — and can be associated with a `name`. For example, in Listing 1, two waypoints have been defined: `p1` and `p2`. Each block defines a segment of flight objective. In the example, 3 blocks have been defined: `Takeoff`, `Survey` and `Land`. It is important to note that the order of blocks represents the sequential control flow of the program — the UAV will first takeoff, perform a survey, and finally land. Once the task defined by a block is finished, the UAV continues to execute the next block.

Surveying is a fundamental task of UAVs. The general idea is to have the UAV sweep back and forth over a predefined area to achieve sufficient coverage. The most standard form of survey covers a rectangular area, whose diagonal corners are defined by XML tags `wp1` and `wp2` at line 11, i.e. waypoints `p1` and `p2` in our example. Each survey consists of sequence of *sweeps*, each of which is defined as a flight path connecting two waypoints on opposite sides of the rectangle. The direction of the sweep is indicated by the `orientation` tag at line 12, where `NS` means the sweep follows a North-South direction. The resolution of the survey, i.e., the distance between two adjacent sweeps, is indicated by the `grid` tag at line 11, e.g., 5 meters in the example.

3 ADAPTIVE VARIABLES

```
1 <block name="adaptive_variable_design">
2     <command tag_1=adaptive_var_1 tag_2=adaptive_var_2 ... />
3     <adaptation guard=condition_1 var=adaptive_var_1 value=
        assignment_expression_1/>
4     <adaptation guard=condition_2 var=adaptive_var_2 value=
        assignment_expression_2/>
5     ...
6 </block>
```

Listing 2: A Flight Plan Block With Adaptive Variables (Terminals and Non-Terminals are Represented by `TrueType` Font and *italic* Font Respectively)

In this section, we describe the programming abstraction of adaptive variables. We introduce its syntax and informal semantics in Section 3.1, with two motivating use scenarios detailed in Sections 3.2 and 3.3.

3.1 Syntax and Semantics

We support adaptive variables with a new clause called `adaptation` as shown in Listing 2. The `adaptation` clause can be associated with three tags: `guard`, `var`, and `value`. The `guard` tag specifies a predicate where the adaptation will be triggered if the predicate becomes true. The `var` tag represents the name of the adaptive variable whose state will change upon adaptation. The `value` tag contains the new value for the adaptive variable upon the satisfaction of the guarding variable. The clause is block-scoped, and each block may contain many adaptations. For example, Listing 2 includes two adaptation clauses defining adaptive variables *adaptive_var_1* and *adaptive_var_2*.

The reason that the state change in adaptive variables has an affect on flight behavior is due to their participation in built-in Paparazzi commands. Paparazzi contains 46 commands which are linked to C functions whose arguments can be provided by the tag associated with the command. For example, the command in Listing 3 is `survey_rectangle`. Tags `wp1`, `wp2`, and `grid` are tags associated with the arguments of a C function that implements the survey functionality. A more general form can be found at line 2 in Listing 2. Here, any change of *adaptive_var_1* or *adaptive_var_2* will result in an adaption of *command*.

There are two noteworthy aspects in the run-time semantics of a flight plan. First, the evaluation of different blocks follows a sequential semantics: the first block encountered in the lexical order will be executed first, followed by the execution of the second block, and so on. Second, within each block, a *control loop* is at work: the run-time system "executes" the non-adaptation commands sequentially included in the block in *stages*, where the execution of such a command forms a stage. The semantics of each command execution

is event-based. More precisely, the lower-level C function that mirrors each higher-level command is indeed an event handler, which will be periodically triggered based on a pre-determined timer set by the UAV system. A stage will terminate when the command it embodies is "completed", a command-specific condition. For example, the `survey_rectangle` command terminates when the sweep of a geographical survey is completed. As this is the last command in the block, the control loop for the block is also terminated.

As adaptive variables are block-scoped, their semantics is more related to their integration within the control loop. The guarding condition of each adaptive variable is evaluated periodically, at the same rate as the control loop. When the guarding condition is met, the variable is updated. The event handler associated with each command is always called with the latest states of the adaptive variables. Their execution is *stage-less*, in that their evaluation will be applied to all stages defined in the block. In Section 4, we will revisit the semantic behavior when the implementation and translation of adaptive variables is discussed.

Multiple adaptive variables can appear in the same block. We also allow multiple adaptation clauses for the same adaptive variable to appear in the same block. At the beginning of each period of the control loop at run time, the guarding conditions of all adaptation clauses enclosed in the block will be evaluated, and the adaptive variables will be updated accordingly. The order of evaluation follows the lexical order in which the adaptation clauses appear in the block.

We allow adaptive variables to be defined dependently, i.e., one adaptive variable (say, `x`) appears in the guarding condition of another (say, `y`). The semantics of their updates is simple, following the lexical order of evaluation: if the adaptation clause of `x` appears before that of `y`, and variable `x` is adapted in an iteration of the control loop, variable `y` will be adapted based on the updated value of `x`; otherwise if the adaptation clause of `x` appears after that of `y`, then variable `y` will be adapted based on value of `x` from the previous iteration.

Guarding conditions can be arbitrary C Boolean expressions. In particular, we also allow user-defined functions in the guarding condition, a feature also supported in Paparazzi flight plan. Physically, these functions can be defined in a Paparazzi sub-directory called `sw/airborne/modules`.

Let us now describe the new construct through two examples.

3.2 Battery-Aware Geographical Survey

```
1 <block name="Battery Aware Survey">
2     <set var="sweep" value="5"/>
3     <survey_rectangle wp1="p1" wp2="p2" grid="sweep" orientation="
        NS"/>
4     <adaptation guard="12.3 > electrical.vsupply" var="sweep"
        value="10"/>
5     <adaptation guard="12.2 > electrical.vsupply" var="sweep"
        value="15"/>
6     <adaptation guard="12.0 > electrical.vsupply" var="sweep"
        value="20"/>
7 </block>
```

Listing 3: A Flight Plan Block For Battery-Aware GS

Our first example can be viewed as an instance of COP where system resources serve as the context. Listing 3 demonstrates flight plan block for battery-aware survey. The overall goal is to adjust the resolution of the survey based on the remaining battery level of the UAV. Line 2 firstly initializes a variable `sweep` to 5, using a pre-defined Paparazzi command named `set`. The variable serves as an adaptive variable for the survey rectangle clause, specifically its argument `grid`. The UAV will then begin its survey between the given waypoints `p1` and `p2`. During the course of the survey, the battery level `electrical.vsupply` will be checked and compared against the given values in the adaptation's conditions defined by the `guard` tags. Once a guard is satisfied, the `sweep` variable will be set to the corresponding value defined by the `value` tag, and the survey will continue with this new resolution.

Adjusting the `grid` variable, i.e., the distance between sweeps, will change the resolution of the survey. A shorter sweep width also means that there are more legs, defined as a flying task from the north most edge of the rectangle to the south most edge, or vice versa. This will take more time and energy from the UAV, but can be considered as a more precise solution. By applying our adaptive variables to the `grid` argument of our survey, we can adjust how precisely we complete our task. On the high level, the design here adapts the resolution of our flying task to our remaining resources, an instance of energy-aware flight planing. With this programming idiom, the UAVs have more likelihood to complete its task before the battery depletes.

3.3 Wind-Adaptive Circle Navigation

```
1  <block name="Wind Adaptive Circle">
2      <set var="var_throttle" value="0.80"/>
3
4      <circle wp="HOME" throttle="var_throttle" pitch="-15" vmode="
         throttle"/>
5
6      <!--STRONG WIND WITH-->
7      <adaptation guard="wind_speed() > 4.5 && (0.25 >
8      (abs(wind_dir-stateGetHorizontalSpeedDir_i()) ||
9        (abs(wind_dir-stateGetHorizontalSpeedDir_i())>1.75)))"
10     var="var_throttle"  value="0.5"/>
11
12     <!--STRONG WIND AGAINST-->
13     <adaptation guard="wind_speed() > 4.5 && (1.25 >
14     (abs(wind_dir-stateGetHorizontalSpeedDir_i()) &&
15       (abs(wind_dir-stateGetHorizontalSpeedDir_i())>0.75)))"
16     var="var_throttle" value="0.9"/>
17
18     <!--WEAK WIND WITH-->
19     <adaptation guard="wind_speed() < 4.5 && (0.25 >
20     (abs(wind_dir-stateGetHorizontalSpeedDir_i()) ||
21       (abs(wind_dir-stateGetHorizontalSpeedDir_i())>1.75)))"
22     var="var_throttle"  value="0.75"/>
23
24     <!--WEAK WIND AGAINST-->
25     <adaptation guard="wind_speed() < 4.5 && (1.25 >
26     (abs(wind_dir-stateGetHorizontalSpeedDir_i()) &&
27       (abs(wind_dir-stateGetHorizontalSpeedDir_i())>0.75)))"
28     var="var_throttle" value="0.85"/>
29 </block>
```

Listing 4: A Flight Plan Block Wind-Adaptive Circle Navigation

Our Second example can be viewed as an instance of COP where environmental conditions serve as the context. Listing 4 demonstrates a flight plan block for a wind-adaptive circle navigation. At the beginning of the flight plan block, the command for the UAV to follow a `circle` routine. Line 2 firstly initializes `var_throttle` to 0.80. It serves as the adaptive variable for the `circle` clause,

specifically the argument `throttle`. In UAV system design, throttle refers to the percentage of power given to the motors, where 100% indicates full motor speed. The main command for the block is a `circle` routine. Its tag `wp` represents the center of the circle; tag `pitch` indicates the orientation of the UAV; and tag `vmode` is a constant whose value `throttle` indicates that the `throttle` tag will be used to control the trajectory of the UAV.

Our adaptations are set up to detect environmental factors including the wind strength represented by `wind_speed()` and the relative direction of wind represented by the difference expression `abs(wind_dir-stateGetHorizontalSpeedDir_i())`. By being able to detect these wind features, we can better adjust our throttle to stay on course.

In circle navigation, the desired trajectory should cover a sequence of implicitly defined waypoints at a relatively constant ground speed. What the wind condition can introduce is a discrepancy between the air speed and the ground speed. If the UAV's throttle were to remain unchanged in the presence of wind, the UAV will experience different ground speeds at the various waypoints of the circle navigation. Therefore, to maintain a constant ground speed, we set up 4 adaptations in Listing 4 to adjust the variable `var_throttle` accordingly. In the adaptation at line 7, for example, if we have a strong wind traveling in the same direction of our trajectory, then we should be able to decrease our throttle, hence airspeed, while maintaining the same speed over ground.

4 IMPLEMENTATION AND PRELIMINARY EVALUATION

4.1 Compilation Process

Figure 1 shows the general compilation process in Paparazzi from flight plan to binary deployed on the UAV. The flight plan generator takes the XML-based flight plan file as the input, and transforms it into a C file `AutoNav.c`. This C program is linked to the required firmware files, and cross-compiled to an architecture-specific binary. This process is the general Paparazzi groundwork that our adaptive variables are built on. Specifically, by extending the flight plan generator and changing certain features of the underlying implementation, we can support adaptive variables in our flight plan language. The flight plan generator by Paparazzi is originally written in OCaml, which is also our development language for the extension. Our development and evaluation have been performed with Paparazzi version 5.15_devel.

After the autopilot C code is generated, the executable will be compiled together with application modules, communication protocols, and hardware drivers. These software components are "libraries" for the autopilot software. As a result, our flight plan can refer to any global variables or predefined functions in these software components. For example, `electrical.vsupply` in Listing 4 is a global variable defined as a hardware driver, in a C file `sw/airborne/modules/sensors/bat_voltage_ardrone2.c`. Paparazzi has an extensible design for adding new modules, allowing these libraries to be extended.

4.2 Code Generation For Adaptive Variables

```
1  void auto_nav() {
2    ...
3    switch(nav_block) {
4      ...
5      Block(N)
6        switch(nav_stage) {
7          ...
8          Stage(M)
9            ...
10       }
11       ...
12   }
13 }
```

Listing 5: Autopilot Control Loop

```
1  void auto_nav() {
2    ...
3    switch(nav_block) {
4      Block(1) //Takeoff
5        switch(nav_stage) {
6          ...
7        }
8      Block(2) //Dynamic Survey
9        if ((12.300000>electrical.vsupply)) {sweep = 10;}
10       if ((12.200000>electrical.vsupply)) {sweep = 15;}
11       if ((12.000000>electrical.vsupply)) {sweep = 20;}
12       switch(nav_stage) {
13         Stage(0)
14           sweep = 5;
15           NextStageAndBreak();
16         Stage(1)
17           NavSurveyRectangleInit(4, 5, sweep, NS);
18           NextStageAndBreak();
19         Stage(2)
20           if (NavSurveyRectangle1(4, 5, sweep)) {break;}
21           else {NextStageAndBreak();}
22           break;
23         default : break
24       }
25     Block(3) //Land
26       switch(nav_stage) {
27         ...
28       }
29   }
30 }
```

Listing 6: Autopilot Control Loop with Battery-Aware GS

Paparazzi compiles the flight plan file into an autopilot control loop, written in C. As is shown in Listing 5, the autopilot control loop takes the form of a periodically called function named `auto_nav`. The function consists of twin-tiered nested switch statements. Each case of the outer switch statement is mapped to a flight plan block that we described in Section 2. Each case of the inner switch statement is mapped to a stage within the block.

The logic specific to adaptive variables is generated at the level of the inner switch. By placing it at this level, we assure that our adaptations will only be considered when the enclosing block is executed. We place the guarding condition check before the inner switch statement, ensuring that adaptation will start before every stage of a block. Consider the adaptations presented in the rectangle survey example presented in Section 3. These adaptations should only be considered during a survey, and should not be considered by the logic inside of our takeoff and landing blocks. As shown in Listing 6, the adaptation clauses cannot be reached when the outer switch case is not `Block(2)`.

```
1  #define InitStage() nav_init_stage();
2  #define NextStageAndBreak() { nav_stage++; InitStage(); break; }
```

Listing 7: Flight Plan Utility Functions

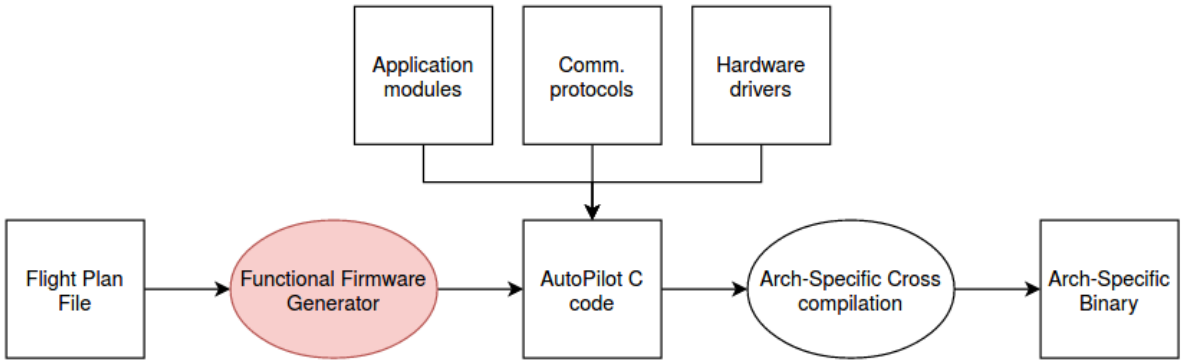

Figure 1: Autopilot Compilation (Our efforts focus on the colored components)

It is important to observe that the `auto_nav` function is periodically called. To allow for stage transitions, a Paparazzi utility function names `NextStageAndBreak` will be called, whose definition can be found in Listing 7. To determine whether the stage should be completed, each command is associated with a termination condition. In the case of `survey_rectangle` the termination condition is `NavSurveyRectangle()`. If the function returns false, the current stage is complete, and the autopilot control loop will transition to the next stage. This can be seen on lines 20 and 21 in Listing 6.

Let us now take a closer look at the generated code. Recall that adaptive variables are stage-less: note that the generated code for their updates appears immediately before the inner switch. As a result, the guarding conditions will be checked at every period, and the variables will be updated accordingly at the same rate if the guarding conditions are true. The generated code for Stage 0 results from the `set` command in line 2 of Listing 4. In Paparazzi, the rectangle survey is split into two separate stages, for initialization (stage 1) and for surveying itself (stage 2). Function call `NavSurveyRectangleInit(4,5,sweep,NS)` initializes the survey with the initial setting for sweep in stage 1. In this stage, there is a function call `NextStageAndBreak`, and as a result, the code in stage 1 is only executed once in the periodic control loop. Stage 2, on the other hand, will be called periodically until the survey is completed, when `NavSurveyRectangle1` returns false. Function `NavSurveyRectangle1` is a variant of the existing Paparazzi survey function `NavSurveyRectangle`, which only takes the first two arguments of the former. As Paparazzi does not support adaptive survey, its original API does not include a dynamic sweep width as an argument, which we have extended.

In Paparazzi's run-time design the `auto_nav` function is exclusively executed by a single thread. Since adaptive variables only appear in blocks, they are exclusively read/written by one thread. There are no race conditions.

Figure 2: The Simulation of Paparazzi's GS

Figure 3: The Simulation of Energy-Aware GS (Two battery-aware adaptations occur at the red circles.)

4.3 Simulation and In-Flight Preparation

We have used the New Paparazzi Simulator (NPS) to evaluate our design of adaptive variables, using the example provided in Section 3 as our test case. NPS can simulate the flight path given a flight plan. For example, in Figure 2, the simulation shows the trajectory of

a rectangle survey, where S1 and S2 are the bounding waypoints. Observe that in paparazzi's build-in rectangle survey support, the grid size must be constant. As a result, the trajectory of the UAV has equal distance between a pair of adjacent sweeps.

Figure 3 demonstrates the simulation results from our energy-aware survey. The two small red circles placed on the UAV's path are locations where adaptations happen. Observe that the difference in sweep density before and after each adaptation. This result shows that our UAV is capable of adjusting the behavior of survey based on the remaining battery levels. NPS does not support a build-in battery simulation. In this experiment, we emulate the behavior of the battery by linearly decreasing the supply voltage.

We have taken the preparation steps in testing our feature on a real-world platform. We have reconfigured a Paparazzi AR drone 2, successfully re-rooting it with Paparazzi autopilot. The UAV is installed with a GPS (Ublox NEO-6M), required for autonomous navigation. We have been able to confirm the operational condition of our UAV and its components, together with a successful compilation and deployment of our target code.

4.4 Open Source Development

The original implementation of Paparazzi's rectangle survey required the user to statically set the sweep width while initializing the survey, and did not support adjustments to its parameters during the survey. Instead, to mimic dynamic adjustment, a Paparazzi flight plan programmer would have to stop the survey, reinitialize it with the new sweep width, and restart.

A sub-component of our adaptive variable support is to enable the sweep width (the `grid` tag) to associate with a variable. With this design, the sweep width can be updated in the middle of a survey without the need to stop-reinitialize-restart. This implementation, which can be viewed as adaptive variables but without language support, has been accepted into the main Paparazzi Repository available at GitHub [1].

Our language support for adaptive variables are not currently included in the main branch of the paparazzi repository, it can also be found on GitHub [2].

5 RELATED WORK

In main stream of autonomous robot programming, the most commonly used programming language is C, often with extensions. Simmons and Apfelbaum [20] describe a higher-level language for describing robotic tasks built as extensions to C. ROS [19] is a robotics middle-ware with a programming interface. ROS is graph-based where individual robotic tasks are represented as nodes that may communicate with each other. ROS is equipped with a higher-level configuration language to help generate glue code for this graph-based model. Vanthienen et al. [23] proposes a high-level domain-specific language for configuring and coordinating tasks in a constraint-based model. Their high-level language is built on top of Lua [14]. Meld [4] is a logic programming language to enable global programming for an ensemble of robots. A survey on programming challenges for robots can be found in [25]. Among these efforts, our design is more aligned with the higher-level languages with a focus on adaptation support.

Adaptation is a central focus for Context-Oriented Programming (COP) [13]. Layers [2][3][7][16][17][22] are a powerful programming abstraction to capture the unit of behavioral variation. Through the dynamic activation of layers, the program may adapt to new behavior. Context traits [12] supports run-time adaptation via dynamic traits composition. COP is known to be useful in many system domains such as fault-tolerant distributed systems [11] and wireless sensor networks [1]. Adaptive variables are a language design to apply the principle of COP to UAV planning. As the behavior of UAVs is context-driven, we believe that COP language features can be beneficial for defining context adaptation. Relative to existing COP features, adaptive variables are simple, working with a higher-level declarative programming model.

FRP [8] is an influential functional language to support reactive programming, with libraries and extensions specific for robotic development. In FRP-like languages, change propagation is managed implicitly, i.e., when the value of a signal is changed, all values that are dependent on this signal, based on the data-flow latent in the program, will be automatically updated as well. Reactive features have also been introduced into COP languages [15] [18], including support for embedded systems [24]. Our language allows for dependencies between adaptive variables, but we do not support reaction semantics among them. Instead dependent adaptive variables are explicitly represented in the declarative program, and the changes are propagated explicitly in the next iteration of the control loop. Within the case studies we have conducted for UAV flight planning, our current design appears to be sufficient. In general, it is interesting future work to investigate the need of reactive features in the presence of dependent adaptive variables for UAV planning.

Adaptation is also studied in the context of energy-aware programming languages. In Eon [21], the data flow may be dynamically adapted based on the energy level. In Energy Types [6], the `mswitch` construct allows the program to adjust its behavior based on the energy modes. Eco [26] introduces mode cases which allows adaptive values to be modeled as first-class citizens. In Ent [5], a combination of static typing and dynamic typing enables principled energy-adaptive behavior. Eon is built on top of a data flow programming model, and the rest of the existing work are built on general-purpose object-oriented languages. Our adaptation support is based on declarative languages.

6 CONCLUSIONS AND FUTURE WORK

Like other autonomous mobile systems, a top priority of UAVs is to respond to a rapidly changing context. Declarative programming simplifies UAV behavior specification and promotes program reasoning, but does not naturally support the variability and dynamism that UAVs require to navigate complex contexts. We have described a declarative language for UAV planning, with a focus on its support of adaptability through using adaptive variables, and evaluated its usefulness through motivating use scenarios.

In this paper we focus on adaptive variables, an abstraction for declarative and adaptive *state* changes. In the future we also plan to support declarative and adaptive *behavior* changes. In our

[1] https://github.com/paparazzi/paparazzi
[2] https://github.com/jburns11/paparazzi_adaptions

current design, adaptive behavior change can be supported when the adaptive variable is a function pointer, but we think a design in this flavor would weaken its declarative nature. To mirror our support for variables, we are interested in designing an abstraction called *adaptive blocks*, where flight plan blocks may be guarded by a condition just as adaptive variables are. Alternative behaviors can thus be supported through multiple adaptive blocks guarded by different conditions, which we briefly illustrate in Listing 8. If the `wind_speed` is below 4.5 meters per second, a `Survey` will be performed. Otherwise, the UAV will `Circle` around waypoint `p1`.

```
<flight_plan>
    <blocks>
        <adaptive_block name="Survey" guard="wind_speed() < 4.5">
            <survey_rectangle wp1="p1" wp2="p2" grid="sweep"
            orientation="NS"/>
        </adaptive_block>
        <adaptive_block name="Circle" guard="wind_speed() >= 4.5">
            <circle wp="p1" radius="10"/>
        </block>
    </blocks>
</flight_plan>
```

Listing 8: A Proposed Adaptive Block

Acknowledgments We thank anonymous reviewers for their useful suggestions. We thank Anthony Canino, Hector Garcia de Marina, and Gautier Hattenberger for their help with Paparazzi. This work is sponsored by US National Science Foundation (NSF) award CNS-1823260.

REFERENCES

[1] M. Afanasov, L. Mottola, and C. Ghezzi. 2014. Context-Oriented Programming for Adaptive Wireless Sensor Network Software. In *2014 IEEE International Conference on Distributed Computing in Sensor Systems*. 233–240.

[2] Malte Appeltauer, R. Hirschfeld, Michael Haupt, and H. Masuhara. 2011. ContextJ: Context-oriented programming with Java. *Information and Media Technologies* 6 (06 2011), 399–419. https://doi.org/10.11185/imt.6.399

[3] Malte Appeltauer, Robert Hirschfeld, Hidehiko Masuhara, Michael Haupt, and Kazunori Kawauchi. 2010. Event-Specific Software Composition in Context-Oriented Programming. 50–65. https://doi.org/10.1007/978-3-642-14046-4_4

[4] M. P. Ashley-Rollman, S. C. Goldstein, P. Lee, T. C. Mowry, and Padmanabhan Pillai. 2007. Meld: A declarative approach to programming ensembles. In *2007 IEEE/RSJ International Conference on Intelligent Robots and Systems*. 2794–2800.

[5] Anthony Canino and Yu David Liu. 2017. Proactive and adaptive energy-aware programming with mixed typechecking. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017*. 217–232.

[6] Michael Cohen, Haitao Steve Zhu, Senem Ezgi Emgin, and Yu David Liu. 2012. Energy Types. In *OOPSLA '12*.

[7] Pascal Costanza and Robert Hirschfeld. 2005. Language constructs for context-oriented programming: An overview of ContextL. *Proceedings of the Dynamic Languages Symposium*. https://doi.org/10.1145/1146841.1146842

[8] Conal Elliott and Paul Hudak. 1997. Functional Reactive Animation. In *Proceedings of the Second ACM SIGPLAN International Conference on Functional Programming* (Amsterdam, The Netherlands) *(ICFP '97)*. 263–273.

[9] J. N. Fadila and P. M. N. S. A. Basid. 2019. Solar Cell-Powered UAVs for Marathon Flights as a Geographic Data Retrieval Tool. In *2019 International Conference on Electrical Engineering and Computer Science (ICECOS)*. 227–230.

[10] Balazs Gati. 2013. Open source autopilot for academic research - The Paparazzi system. *2013 American Control Conference* (2013). https://doi.org/10.1109/acc.2013.6580045

[11] Carlo Ghezzi, Matteo Pradella, and Guido Salvaneschi. 2010. Programming Language Support to Context-Aware Adaptation: A Case-Study with Erlang. In *Proceedings of the 2010 ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems* (Cape Town, South Africa) *(SEAMS '10)*. Association for Computing Machinery, New York, NY, USA, 59–68. https://doi.org/10.1145/1808984.1808991

[12] Sebastián González, Kim Mens, Marius Colacioiu, and Walter Cazzola. 2013. Context Traits: Dynamic Behaviour Adaptation through Run-Time Trait Recomposition. In *Proceedings of the 12th Annual International Conference on Aspect-Oriented Software Development* (Fukuoka, Japan) *(AOSD '13)*. Association for Computing Machinery, New York, NY, USA, 209–220. https://doi.org/10.1145/2451436.2451461

[13] Robert Hirschfeld, Pascal Costanza, and Oscar Nierstrasz. 2008. Context-Oriented Programming. *Journal of Object Technology, March-April 2008, ETH Zurich* 7, 3 (2008), 125–151.

[14] Roberto Ierusalimschy, Luiz Henrique de Figueiredo, and Waldemar Celes Filho. 1996. Lua—An Extensible Extension Language. *Software: Practice and Experience* 26, 6 (1996), 635–652. https://doi.org/10.1002/(SICI)1097-024X(199606)26:6<635::AID-SPE26>3.0.CO;2-P

[15] Hiroaki Inoue and Atsushi Igarashi. 2016. A Library-Based Approach to Context-Dependent Computation with Reactive Values: Suppressing Reactions of Context-Dependent Functions Using Dynamic Binding. In *Companion Proceedings of the 15th International Conference on Modularity* (Málaga, Spain) *(MODULARITY Companion 2016)*. Association for Computing Machinery, New York, NY, USA, 50–54. https://doi.org/10.1145/2892664.2892669

[16] Tetsuo Kamina, Tomoyuki Aotani, and Hidehiko Masuhara. 2011. EventCJ: A context-oriented programming language with declarative event-based context transition. *Proceedings of the 10th International Conference on Aspect-Oriented Software Development, AOSD.11*, 253–264. https://doi.org/10.1145/1960275.1960305

[17] Tetsuo Kamina, Tomoyuki Aotani, and Hidehiko Masuhara. 2012. Bridging Real-World Contexts and Units of Behavioral Variations by Composite Layers. In *Proceedings of the International Workshop on Context-Oriented Programming* (Beijing, China) *(COP '12)*. Association for Computing Machinery, New York, NY, USA, Article 4, 6 pages. https://doi.org/10.1145/2307436.2307440

[18] Tetsuo Kamina, Tomoyuki Aotani, and Hidehiko Masuhara. 2017. Push-based reactive layer activation in context-oriented programming. 17–21. https://doi.org/10.1145/3117802.3117805

[19] Morgan Quigley, Ken Conley, Brian P. Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler, and Andrew Y. Ng. 2009. ROS: an open-source Robot Operating System. In *ICRA Workshop on Open Source Software*.

[20] R. Simmons and D. Apfelbaum. 1998. A task description language for robot control. In *Proceedings. 1998 IEEE/RSJ International Conference on Intelligent Robots and Systems. Innovations in Theory, Practice and Applications (Cat. No.98CH36190)*, Vol. 3. 1931–1937 vol.3.

[21] Jacob Sorber, Alexander Kostadinov, Matthew Garber, Matthew Brennan, Mark D. Corner, and Emery D. Berger. 2007. Eon: A Language and Runtime System for Perpetual Systems. In *Proceedings of the 5th International Conference on Embedded Networked Sensor Systems* (Sydney, Australia) *(SenSys '07)*. 161–174. https://doi.org/10.1145/1322263.1322279

[22] Matthias Springer, Hidehiko Masuhara, and Robert Hirschfeld. 2016. Classes as Layers: Rewriting Design Patterns with COP: Alternative Implementations of Decorator, Observer, and Visitor. In *Proceedings of the 8th International Workshop on Context-Oriented Programming* (Rome, Italy) *(COP'16)*. Association for Computing Machinery, New York, NY, USA, 21–26. https://doi.org/10.1145/2951965.2951968

[23] D. Vanthienen, M. Klotzbucher, J. De Schutter, T. De Laet, and H. Bruyninckx. 2013. Rapid application development of constrained-based task modelling and execution using domain specific languages. In *2013 IEEE/RSJ International Conference on Intelligent Robots and Systems*. 1860–1866.

[24] Takuo Watanabe. 2018. A Simple Context-Oriented Programming Extension to an FRP Language for Small-Scale Embedded Systems. In *Proceedings of the 10th International Workshop on Context-Oriented Programming: Advanced Modularity for Run-Time Composition* (Amsterdam, Netherlands) *(COP '18)*. Association for Computing Machinery, New York, NY, USA, 23–30. https://doi.org/10.1145/3242921.3242925

[25] S. Yang, X. Mao, B. Ge, and S. Yang. 2015. The Roadmap and Challenges of Robot Programming Languages. In *2015 IEEE International Conference on Systems, Man, and Cybernetics*. 328–333.

[26] Haitao Steve Zhu, Chaoren Lin, and Yu David Liu. 2015. A Programming Model for Sustainable Software. In *ICSE'15*. 767–777.