DSVerifier-Aided Verification Applied to Attitude Control Software in Unmanned Aerial Vehicles

DOI:
10.1109/TR.2018.2873260

Document Version
Accepted author manuscript

DSVerifier-Aided Verification Applied to Attitude Control Software in Unmanned Aerial Vehicles

Lennon Chaves, Iury Bessa, Hussama Ismail, Adriano Frutuoso, Lucas Cordeiro, Eddie de Lima Filho

***Abstract*—During the last decades, model checking techniques have been applied to improve overall system reliability, in unmanned aerial vehicle (UAV) approaches. Nonetheless, there is little effort focused on applying those methods to the control-system domain, especially when it comes to the investigation of low-level implementation errors, which are related to digital controllers and hardware compatibility. The present study addresses the mentioned problems and proposes the application of a bounded model checking tool, named as Digital System Verifier (DSVerifier), to the verification of digital-system implementation issues, in order to investigate problems that emerge in digital controllers designed for UAV attitude systems. A verification methodology to search for implementation errors related to finite word-length effects (*e.g.*, arithmetic overflows and limit cycles), in UAV attitude controllers, is presented, along with its evaluation, which aims to ensure correct-by-design systems. Experimental results show that low-level failures in UAV attitude control software used in aerial surveillance are identified by DSVerifier, which can also be used for developing sound and correct implementations, through its integration into development processes. Finally, given that the proposed approach handles C code and takes into account hardware specifications, it is suitable for verifying final controller implementations, which is a more practical scenario.**

***Index Terms*—Unmanned Aerial Vehicle, Symbolic Model Checking, Fixed-Point Digital Controllers, Formal Verification, Embedded Systems.**

I. Introduction

DURING the last decades, the unmanned aerial vehicle (UAV) approach has been used in various military and civil applications, such as armed attacks, training targets, aerial surveillance, journalism, and entertainment. More recently, autonomous UAVs have gone through a notable development, due to the current evolution of embedded systems [1].

Autonomous UAVs typically demand improved intelligence and reliability, in order to ensure mission accomplishment and reduce costs related to crashes and malfunctioning [2]. Thus, the development of robust UAV systems is leading to an increasingly interest in both academy and industry [3]: techniques related to fault detection and diagnosis (FDD), fault tolerant control (FTC) [4], and formal verification via model checking are currently being applied to UAVs [5].

In particular, the FDD and FTC communities concentrate their efforts on fast detection and hardware-failure isolation, which aim to maintain system functionality during faulty conditions and eliminate crashes, while the model checking community aims to ensure correct implementation. As a result, model checking techniques are able to ultimately improve safety and reliability of UAV applications. For instance, Tafazoli [6] studied failure causes of on-orbit spacecrafts and showed that 6% of them are related to software, which sometimes leads to the loss of entire systems (*e.g.*, the fatal failure in Mars Climate Orbiter mission [7]).

Since the 90's decade, formal methods have been applied to improve automation systems. In that sense, Alur *et al.* [8] presented the earliest application of model-checking tools for timed automata, using timed computation tree logic (TCTL). Since then, this kind of work has inspired the development of some formal methods and model checking tools for real-time and automation systems, such as UPPAAL [9] and HyTech [10], which support cyber-physical and hybrid systems and are also able to improve their reliability [11].

Formal verification has been applied to avionics embedded software, since the 2000s, due to safety and reliability requirements [12]. Different tools (*e.g.*, SPIN [13], SMV [14], and NuSMV [15]) were used for developing and validating flight control software, such as the NASA's missions Mars Science Laboratory [16] and Deep Space 1 [17], the flight control system FCS 5000 [18], and the military aircraft A-7 [19].

Most previous studies concentrate on safety regarding real-time requirements [11], [20], [21]; however, there are a few that discuss low-level properties and even less related to implementation aspects and hardware features, such as the finite word-length (FWL) problem. That occurs because such properties typically take into account complex system dynamics and require verification tools with some knowledge about the underlying hardware and specialized in implementation aspects [22], [23]. As a result, the main challenges for those verification tools rely on checking how FWL effects influence digital-controller performance and make them susceptible to errors related to overflows (limited by the maximum and minimum representations) and limit-cycles (due to overflows and also round-offs), which are problems related to fixed-point implementations. In summary, comprehensive verification procedures should not focus only on high-level specifications, but also include low-level aspects.

As a consequence, system models, based on non-linear arithmetic, are necessary, when trying to identify conditions that lead to overflow and limit-cycle events, and it is interesting to use non-deterministic inputs, in order to explore large state spaces, which thus make verification conditions really hard to be checked. Those considerations make the mentioned challenges even more difficult, due to the need for system knowledge and behavior investigation. Indeed, if an overflow is not avoided, an UAV controller might then perform wrong

L. Chaves is with Federal University of Amazonas, Brazil. I. Bessa is with Department of Electricity, Federal University of Amazonas, Brazil. H. Ismail and E. Lima Filho are with Graduate Program in Electrical Engineering, Federal University of Amazonas, Brazil. A. Frutuoso is with Federal Institute of Amazonas, Brazil. L. Cordeiro is with School of Computer Science, University of Manchester. E-mails: {lennonchaves, iurybessa, hussamaibrahim}@ufam.edu.br, {adriano.frutuoso}@ifam.edu.br, {lucas.cordeiro}@manchester.ac.uk, {eddie_batista}@yahoo.com.br

operations, which could impair performance and navigation/-positioning of UAV systems. In addition, limit cycles might surpass the limiting safe flight boundaries of aircrafts and potentially lead to structural damage and catastrophes [24].

Recently, Bessa *et al.* [25] investigated FWL effects in digital controllers. Properties related to overflow, limit cycle, time constraint, stability, and minimum phase were verified, in different software realizations (delta- and direct-forms) and implementations (*e.g.*, number of bits), using the Digital System Verifier (DSVerifier) [26]. DSVerifier is a bounded model checking (BMC) framework for digital systems that uses other state-of-the-art tools, such as the Efficient SMT-based Context-Bounded Model Checker (ESBMC) [27] and the C Bounded Model Checker (CBMC) [28]. It was developed to verify digital-system implementations and is suitable for investigating FWL performance and hardware compatibility, thus considering implementation aspects. In addition, DSVerifier is able to verify digital filters, digital controllers, and closed-loop control systems. Following that same line of research, this article is the first to investigate FWL effects in UAVs. In particular, it includes overflow and limit-cycle oscillation (LCO) on outputs of UAVs attitude controllers and their effects on physical plants UAVs dynamics, considering fragility of feedback controllers, where prior work [25] only tackled such effects at outputs of digital controllers. Furthermore, the LCO and overflow verification engines were substantially improved, in order to generalize detection for any constant input and different overflow modes (*i.e.*, saturate and wrap-around). By contrast, DSVerifier v1.0 [26] was only able to detect zero-input LCO and in wrap-around overflow mode. Finally, UAV applications present complex elements (multiple control loops and different control configurations), which are tightly coupled to each other (attitude dynamics, angle variations, and position information) and impose an ever increasing level of difficulty to existing verification methodologies.

Contributions. The main contribution of the present study is the introduction of a verification methodology based on DSVerifier and also its theoretical foundations, which aims to investigate FWL effects in digital controllers developed for UAV systems. In particular, this research proposes a DSVerifier-aided verification methodology for UAV attitude control software, considering implementation aspects. As one may notice, DSVerifier is an approach that addresses computations and operations occurring inside a digital controller based on fixed-point arithmetic and, harnessing on that, our methodology is able to detect issues related to arithmetic overflow in wrap-around and saturation modes and also check round-offs errors, which could generate LCOs, *i.e.*, undesirables errors that UAV digital controllers are susceptible to. Experimental results show that implementation-level failures, whose effects would probably degrade overall system performance, can be detected. The proposed methodology substantially extends that presented by Ismail *et al.* [26]: specifically, its application to UAV attitude control is an important step towards safe and reliable avionics systems, which is crucial for navigation and positioning systems. Lastly, our proposed methodology was evaluated using two different verification techniques, which are incremental BMC and k-induction, and we have also compared it to a state-of-the-art fuzzing technique – American Fuzzy Lop (AFL)[1]. Finally, we have observed that AFL was unable to find any property violation in our UAV benchmarks given the time limits, while incremental BMC was able to find a substantial amount of property violations under the same constraints.

Availability of Data and Tools. The presented experimental results are based on a real quadcopter attitude control system for aerial surveillance [29]. All benchmarks, tools, and results of this evaluation are available on a supplementary web page.[2]

Outline. Section II presents related studies. Section III, in turn, describes fundamental concepts about digital controllers, along with implementation aspects. In section IV, the proposed verification methodology for UAVs is presented. Section VI tackles the performed verification experiments and discusses the obtained results; it also shows how those same outcomes were reproduced an validated. Finally, section VII concludes this work and proposes future research topics.

II. Related Work

The focus of the present work is to introduce a methodology for checking problems related to FWL effects in UAV software. As a consequence, the available literature regarding that is presented and discussed, in order to clarify the importance of our main contributions.

A. Formal Verification of Avionics Software

The use of autonomous UAVs has led researchers to develop model checking applications for UAV software with different purposes, *e.g.*, obstacle detection and avoidance [30], ensure the reachability of mission plan for single- [31], [32] and multi-UAVs [5], [33], and evaluate the reliability of fault protection software [17].

All aforementioned studies have a common concern about high-level specifications, generally related to flight planning and navigation. Furthermore, those related studies disregard the dynamics of motion controllers in UAV systems, *i.e.*, they consider that a given UAV behavior is totally described by finite state machines, which only represent transitions and relationships between static tasks, without any computation of input/output models for controllers and physical plants. As a consequence, such structures are able to capture some intuition about the underlying system, but do not cover the complexity and myriad of entanglements regarding all involved elements, mainly when those change with respective outputs.

In a recent work [34], Groce *et al.* tackled various verification methods and tools employed in that project, including model checking tools based on abstraction and BMC techniques. This particular work considers low level issues associated to those devices, *e.g.*, wear leveling and effects of long erase times of NAND flash blocks.

The present work also focuses on low-level aspects, by handling hardware-level implementations of attitude control systems. One may also notice that software engineering techniques typically disregard platforms on which (embedded) system software operates the proposed approach becomes an important contribution towards verification of low-level implementation aspects, in order to check errors caused by FWL effects.

[1] http://lcamtuf.coredump.cx/afl/

[2] http://dsverifier.org/

B. Verification and Validation tools

Ngoc and Ogawa proposed a tool named *C ANAlyzer* (CANA) for statically analyzing fixed-point errors, such as overflow (due to the finite integer part of a representation) and roundoff (due to the finite fraction part of that), in C programs, via model checking [35]. That strategy includes two main steps: a new range representation is proposed to estimate overflow and roundoff errors and their analysis problems are encoded as weighted model checking problems. Nonetheless, CANA presents limitations regarding detection of overflow and roundoff errors in digital controllers, since it does not take into account their implementation and typical realization aspects (*e.g.*, direct and delta forms).

UPPAAL [9] and HyTech [10] are two state-of-the-art verifiers capable of checking reachability properties related to cyber-physical and hybrid systems. In particular, UPPAAL is an automated tool to model, validate, and verify cyber-physical and hybrid systems, which are actually modeled as networks of timed automata extended with specific data types (*e.g.*, bounded integers and arrays), in order to specify additional system's constraints (*e.g.*, timing). HyTech is also another automated tool to analyze, validate, and verify cyber-physical and hybrid systems, which allows system specifications as collections of automata with discrete and continuous components and uses a symbolic model verifier to check temporal requirements of the system. Nonetheless, given the current knowledge in system verification, UPPAAL and HyTech do not tackle FWL effects when verifying cyber-physical and hybrid systems and, in particular, they have not been applied to the verification of low-level control software in UAVs yet.

SMV [14] and NuSMV [15] are two symbolic verifiers based on binary decision diagrams (BDDs); actually, the latter is a reimplementation and extension of SMV, which also supports SAT solvers. The system properties can be expressed in a wide range of temporal properties such as Computation Tree Logic (CTL), Real-Time CTL, Linear Temporal Logic (LTL), and Property Specification Language (PSL). SMV and NuSMV also construct counterexamples whether a property violation is found as other typical model checkers [27], [28].

Particularly, software testing has been the standard technique for identifying software bugs during many years.

Currently, there are studies that compare software testing and model checking techniques [36], [37] and show that model checking is an effective technique, since it is able to find more bugs in software than testing. The work of Lipka *et al.* [36] examined a simple software consisting of several components, in order to compare two different tools: the first one is SimCo, which is a framework for the simulation-based testing of software components, and the second one is Java Pathfinder (JPF), which is a software model checking tool for verifying correctness of components' behaviors. As a result, they showed that JPF was able to detect more bugs than SimCo, since JPF covers all possible scenarios and SimCo was not originally designed for some of them (*e.g.*, it does not search the entire state space).

Dirk and Lemberger [37] also discussed a comparison about testing and model checking tools, in order to prove that model checking presents reliable results, when searching for bugs in programs. During the respective experiments, which were performed with tools for random fuzz testing and model checking, the authors showed that software model checkers are competitive for finding bugs; they are also mature enough to be used in practice, given that they even outperform bug-finding capabilities of state-of-the-art testing tools. Lastly, the authors conclude that BMC techniques are able to find more bugs in programs and are also faster than state-of-the-art software testing tools.

C. Fragility verification of control software

Problems related to low-level implementation aspects (*e.g.*, control software and its compatibility with physical platforms, or simply numerical issues) are called control software fragility and may cause fatal and expensive faults too. A promising approach to capture continuous dynamic behavior and also discrete state transitions is the hybrid automata representation, as proposed by Lerda *et al.* [38] which merge dynamic responses obtained through MATLAB/Simulink and the Java Pathfinder model checker, in order to detect errors in controller designs without considering FWL effect, as studied in this work.

Only a few studies investigated those problems and most tackled them by proposing i) improved and resilient implementations [39]–[43], ii) analysis and formal verification of FWL effects [22], [25], [26], [44]–[48], and iii) non-fragile design [49]–[51] and formal synthesis [52]–[54] of digital control systems. In particular, Anta *et al.* [22] and Hilaire *et al.* [45] investigated round-off effects in fixed-point implementations of digital controllers, Park *et al.* [46], [47] employed SMT solvers and convex optimization to check the input-output equivalence between control system design and extracted models of control system implementations, in order to ensure design control specifications, and Feron *et al.* [44] and Bessa *et al.* [48] employed formal methods to ensure stability of control systems, considering software implementations of digital controllers.

Nonetheless, other FWL effects, such as overflow and LCO, are mostly neglected by current studies. Recently, Ismail *et al.* [26] and Bessa *et al.* [25] employed DSVerifier to check zero-input LCO and overflow occurrences in fixed-point implementations of digital controllers and filters, which is also performed in the present work; however, the current algorithms are more comprehensive, allowing detection of granular LCO for any constant input and overflow with and without two's complement arithmetics. Furthermore, it focuses on UAV attitude control systems, which are more complex than those used in previous studies, where FWL violations were propagated for different sub-systems of a control loop.

The work introduced by Abate *et al.* [52] presents a method for synthesizing stable controllers, which are suitable to continuous plants given as transfer functions, by exploiting bit-accurate verification of software implemented in micro-controllers [26]. The mentioned authors developed a tool called DSSynth, which marks the first use of counterexample-guided inductive synthesis (CEGIS) [55] to synthesize digital controllers, considering physical plants with uncertain models and FWL effects; however, low-level implementation errors (*e.g.*, limit cycles) were not investigated.

The methodology presented here was integrated into the DSVerifier tool [26]. Nonetheless, there are other verification tools that provide similar features, such as Astrée [56], PolySpace [57], and Simulink Design Verifier (SDV) [58]. Although Astrée works on preprocessed C code, it tackles only digital filters and is focused on verifying overflow and register dimensioning, which means that it is not prepared to handle digital controllers and physical plants. SVD is focused on block level (Simulink) and needs substantial work regarding requirement expression and its respective encoding. Finally, PolySpace is more software oriented and generically handles potential run-time errors, while also leaves code fragments for further review.

III. Preliminaries

A. Fixed-Point Arithmetics

Let $\langle I, F\rangle$ denote a fixed-point format and $\mathcal{F}_{\langle I,F\rangle}(x)$ denote a real number x represented in fixed-point domain, with I bits representing the integer part and F bits representing the decimal one. The smallest absolute number c_m that can be represented in such a domain is $c_m = 2^{-F}$ and any mathematical operations performed at $\mathcal{F}_{\langle I,F\rangle}(x)$ will introduce errors, for which an upper bound can be given [59].

Arithmetic operations with fixed-point variables are different from the ones with real numbers, since there are some non-linear phenomenons, *e.g.*, overflows and round-offs, and the radix must be aligned [59]. Here, we treat fixed-point operators for sums, multiplications, subtractions, and divisions as `fxp_add`, `fxp_mult`, `fxp_sub`, and `fxp_div`, respectively.

B. Fixed-Point Implementation in UAVs

Typically, UAVs are driven by digital controllers, which are implementations of difference equations that generally run on microcontrollers and whose main goal is to make a plant (*e.g.*, an UAV system) follow a desired behavior, based on error regarding reference and output signals. Nonetheless, signals provided by UAV sensors (*e.g.*, accelerometers and gyroscopes) are actually analog and then must be converted into digital form, by analog-to-digital (A/D) converting devices. Besides, microcontrollers run control routines and produce discrete control signals, which have to be converted into analog form through digital-to-analog (D/A) converters and zero-order hold (ZOH) devices [60] and then delivered to UAV actuators.

A digital controller is a linear time-invariant (LTI) discrete-time system, which deals with discrete numerical signals and whose implementation is a program executed by a microprocessor. There are many mathematical representations employed for controllers (*e.g.*, transfer functions, state equations, and difference equations) [60], but one of the most common approaches is through transfer functions, which can be described as

$$H(z) = \frac{b_0 + b_1 z^{-1} + ... + b_M z^{-M}}{1 + a_1 z^{-1} + ... + a_N z^{-N}}, \qquad (1)$$

where z and z^{-1} are known as the forward-shift and backward-shift operators, respectively. There are many ways to implement digital controllers and their realization structures heavily influence their performance. Different realizations are studied in the available literature [60], but the present work considers only direct forms, due to their simplicity and usability [60].

Direct realizations employ exact coefficients of transfer function (1), *i.e.* $a_1, \ldots, a_N$ and $b_0, \ldots, b_M$ and their main advantage is that they only deal with delayed input and output versions; however, they also make controllers extremely sensitive to numerical errors, which becomes evident in fixed-point implementations and may severely harm system stability and performance. Different direct forms may present distinct numerical performances, given that those realizations implement the same controller, but with different organizations regarding arithmetic operations [25].

Fig. 1 shows an algorithm for Direct Form I controllers, which can be implemented through C programming language (as shown in Fig. 2) and verified by the supported BMC tools present in DSVerifier. In a C Program, fixed-point variables are implemented as integer variables, with implicit power-of-2 scaling factors. As illustrated in Fig. 2, functions `fxp_add`, `fxp_mult`, `fxp_div`, and `fxp_sub` take two input arguments and return the respective addition, multiplication, division, and subtraction results, in `fxp32_t` format, which is internally defined in DSVerifier as `int32_t`. Addition and multiplication blocks also include quantization effects and consider the fixed-point representation used by a given system. Besides, function `fxp_quantize` provides quantization effects in each output, for a Direct Form I controller.

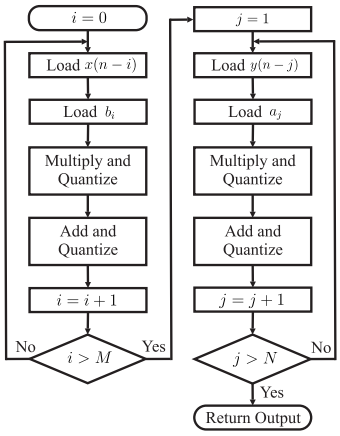


Fig. 1. An algorithm for Direct Form I controllers.

Figure 3 shows three different direct representations: Direct Form I (DFI), Direct Form II (DFII), and Transposed Direct Form II (TDFII), in parts 3a, 3b, and 3c, respectively. The gains a_i and b_i represent controller coefficients, while z^{-1} describes shift operations, as shown in (1). Further details about digital-system representations can be found in control and digital signal processing literature [60], [61].

C. Problems Related to Fixed-Point Implementations

Real implementations of digital controllers are subject to FWL effects, which are of paramount importance in fixed-point processors. Such events, which are due to quantization, are related to round-offs in operation results, which may cause accuracy loss and parametric truncation that ultimately result in functional problems, such as instability.

```
fxp_t fxp_direct_form_1(fxp_t y[], fxp_t x[],
fxp_t a[], fxp_t b[], int Na, int Nb) {
  fxp_t *a_ptr, *y_ptr, *b_ptr, *x_ptr;
  fxp_t sum = 0;
  a_ptr = &a[1];
  y_ptr = &y[Na - 1];
  b_ptr = &b[0];
  x_ptr = &x[Nb - 1];
  int i, j;
  for (i = 0; i < Nb; i++) {
    sum = fxp_add(sum, fxp_mult(*b_ptr++, *x_ptr--));
  }
  for (j = 1; j < Na; j++) {
    sum = fxp_sub(sum, fxp_mult(*a_ptr++, *y_ptr--));
  }
  sum = fxp_div(sum,a[0]);
  return fxp_quantize(sum);
}
```

Fig. 2. C code fragment of a Direct Form I controller.

Quantization occurs during A/D conversion, which approximates analog values to discrete ones and generates rounding errors, whose maximum value is 2^{-b-1}, where b is the number of bits in the fractional part of the chosen representation. A fixed-point representation $\langle I, F\rangle$ can only represent values in the range from -2^{I-1} to $2^{I-1} - 2^{-F}$ [61].

Overflow violations occur when addition or multiplication operations return results outside a given range of representable values, regarding a specific fixed-point format. This way, a microprocessor generally handles overflow through wrap-around (*i.e.*, it allows numerical representation wrapping) or saturation (*i.e.*, it holds the maximum representation). Besides, such errors and round-offs may lead to periodic persistent oscillations in an output, or LCOs, and are not related to instability, but instead to FWL aspects. In addition, verification modules normally handle overflow in the following ways: detection as failure, wrap around, or saturation to a maximum/minimum value.

LCO events, in digital controllers, are defined by the presence of oscillations in their outputs, even with constant input sequences, and are classified as overflow or granular [61]. The former appear when an operation results in overflow and wrap-around. The latter, in turn, are autonomous oscillations, which originate from quantization in the least significant bits [61]. On the one hand, the absence of overflow LCO may be assured by preventing overflow or handling it through saturation, in which the maximum (or minimum) value is held. On the other hand, granular LCO could be eliminated from a system's output through different filter structures or magnitude truncation (zero-input LCO) [61].

IV. DSVerifier-Aided Verification Applied to Attitude Control Software in UAVs

A. Digital-System Verifier (DSVerifier)

In this study, DSVerifier [26] is used, which is a BMC tool for digital systems that employs CBMC [28] and ESBMC [27] as back-ends. Indeed, DSVerifier implements a front-end for those BMC tools, in order to provide support for digital system design and verification, and performs three main procedures: initialization, validation, and instrumentation. When it receives a digital-system specification, the first step is to initialize its internal parameters for quantization, that is, it computes the maximum and minimum representable numbers for the chosen FWL format. Then, during validation, it checks whether all required parameters, for the verification procedure, were correctly provided. In the last step, explicit calls to its verification engine (for the evaluated properties) are added, using specific functions available in CBMC and ESBMC (*e.g.*, `assume` and `assert`), with the goal of checking property violations.

Once those three procedures are performed, an ANSI-C file provided by a user, as shown in Fig. 4, can then be verified. Such a file contains a digital-system description (struct `ds`), *i.e.*, the numerator (`ds.b` = $\{1.561, -1.485\}$) and also the denominator (`ds.a` = $\{1.0, -0.9\}$) of its transfer function, along with implementation-specific data (struct `impl`), such as number of bits in the integer (`impl.int_bits = 3`) and precision (`impl.frac_bits = 5`) parts, input range (`impl.min = -3` and `impl.max = 3`), and scaling factor (`impl.scale = 10`).

Indeed, the input file described in Fig. 4 represents the standard format used by DSVerifier [26] for characterizing digital systems. In particular, the second-order system (*i.e.*, a system that contains two poles and whose transfer-function denominator order is two) described in Fig. 4 represents a controller for an AC motor plant.

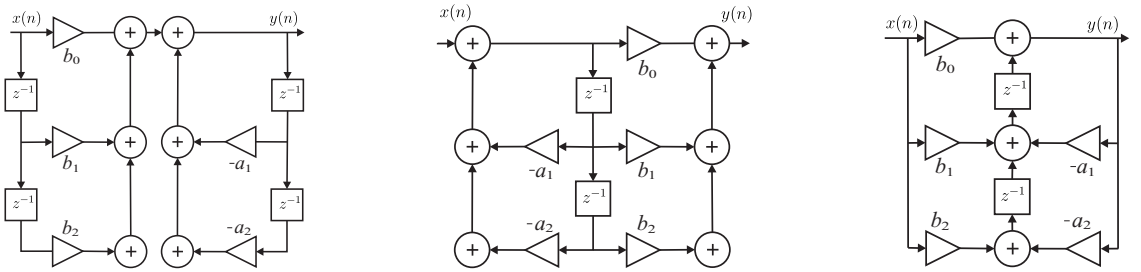
This file is directly sent to a C parser module and then follows the normal verification flow of CBMC [28] or ESBMC [27]. Fig. 5 shows an example of a C code fragment automatically produced by DSVerifier, which computes a DFI structure, includes *assert* and *assume* statements, and is later verified by the chosen back-end, in order to check overflow violations. In particular, `__DSVERIFIER_assume` limits non-deterministic values to the dynamic range defined in $impl$ (as shown in Fig. 4), `shiftL` gets values from the vector with inputs $x(n)$ (determined with non-deterministic values) and permutes them to the left, in order to employ all necessary values for computing $y(n)$, `fxp_direct_form_1()` is the DFI controller implementation, and, finally, `__DSVERIFIER_assert` represents a given property to be checked (in the present case, overflow), using the maximum and minimum representations defined by `fwl_max` and `fwl_min`, respectively.

In the present work, CBMC and ESBMC are employed for reasoning about bit-vector programs, using SAT and SMT solvers, respectively [62]. If they find a property violation, then a counterexample is generated; otherwise, the evaluated implementation is safe, w.r.t. a set of given properties, up to the bound k, which can be later embedded into a microcontroller. Further details on DSVerifier are provided by Ismail *et al.* [26].[3]

In order to prove that our controllers are safe for any depth k, we have applied a state-of-the-art k-induction algorithm to both falsify and prove properties in digital controllers [63]. One may notice that the first version of the k-induction algorithm was originally proposed by Sheeran *et al.* [64], which consists of two cases:

- *Base Case ($B(k)$):* it is a standard BMC procedure that is satisfiable if and only if it has a counterexample of length k or less.

[3]One may also notice that users can even access documentation, benchmarks, and publications about DSVerifier, which are available on its website http://www.dsverifier.org



(a) Direct Form I. (b) Direct Form II. (c) Direct Transposed Form II.

Fig. 3. Direct realizations for digital controllers.

```
#include <dsverifier.h>
digital_system ds = {
    .a = { 1.0, -0.9 },      /* denominator */
    .a_size = 2,             /* denominator length */
    .b = { 1.561, -1.485 }, /* numerator */
    .b_size = 2              /* numerator length */
};
implementation impl = {
    .int_bits = 3, /*  integer bits */
    .frac_bits = 5,/*  precision bits */
    .min = -3.0,   /*  minimum input */
    .max = 3.0,    /*  maximum input */
    .scale = 10
};
```

Fig. 4. A digital-system input file for DSVerifier.

```
fxp_t xaux[ds.b_size]; fxp_t yaux[ds.a_size];
for (int i = 0; i < k; ++i) {
      x[i] = nondet_int();
      __DSVERIFIER_assume(x[i] >= impl.min &&
      x[i] <= impl.max);
      shiftL(x[i], xaux, ds.b_size);
      y[i] = fxp_direct_form_1(yaux, xaux,
      ds.a, ds.b, ds.a_size, ds.b_size);
      shiftL(x[i], xaux, ds.b_size);
      __DSVERIFIER_assert(y[i] >= fwl_min &&
      y[i] <= fwl_max);
 }
```

Fig. 5. C code fragment of a DFI controller, which was produced by DSVerifier.

- *Inductive Step ($I(k)$):* it checks that if a safety property holds in the first k steps, it also holds for $k+1$ steps.

Nonetheless, ESBMC implements an efficient version of this k-induction algorithm, by adding one additional step:[4]

- *Forward Condition ($F(k)$):* it checks whether all program states were reachable for the current k.

Hence, through the combination of $B(k)$, $F(k)$, and $I(k)$, the k-induction algorithm in ESBMC, when verifying a program P at a given k, is:

$$k_{ind}(P,k) = \begin{cases} P \text{ contains a bug}, & \text{if } B(k) \\ P \text{ is correct}, & \text{if } \neg B(k) \wedge [F(k) \vee I(k)] \\ k_{ind}(P,k+1), & \text{otherwise.} \end{cases} \quad (2)$$

The ESBMC k-induction algorithm is applied through an iterative deepening scheme, which allows BMC to be used for proving (partial[5]) correctness, without fully unwinding loops. Furthermore, the incremental nature of the algorithm implies that it always finds the smallest k, aiming to either prove correctness or find a property violation.

B. The DSVerifier-aided Verification Method

This section describes the main steps of our verification method supported by DSVerifier, in order to automatically check the presence of LCO and overflow in attitude controllers employed in UAVs, as shown in Fig. 6. In step 1, UAV attitude controllers are designed through four tasks, for each angle dynamics (pitch, roll, and yaw): angle-dynamics modeling, selection and design of associated structures, coefficient tuning, and controller discretization. In particular, in this work, PID controllers were designed with Ziegler-Nichols tuning, which represents a classical control design approach, and second order structures were designed with the CEGIS-based approach via DSSynth (see subsection III-B). Then, they were converted into digital format, with different methods and sample times (some use the Euler's methods and others the bilinear transformation) [60]. Indeed, DSVerifier requires a digital system described as a transfer-function and encoded in an ANSI-C file, with all its multiplier coefficients. A numerical fixed-point format is then chosen in step 2, which, in this work, is performed as suggested by Carletta *et al.* [40]. Fixed-point formats consider I-bits in their integer parts and F-bits in their fractional ones, which must also be described in an ANSI-C file, as can be seen in Fig. 4. In step 3, all implementations can be tested for DFI, DFII, and TDFII, in order to provide comparison data among different realization forms and hardware model. Then, verification parameters, *e.g.*, properties, overflow mode, timeout, and memory usage, are defined in step 4. One may notice that the respective overflow mode is selected in this step, which can be saturation or wrap-around and is an important definition for overflow verification, since it directly influences a system's output, according to the chosen realization form (DFI, DFII and TDFII). Finally, overflow and LCO events are verified, with non-deterministic inputs, in order to detect violations in digital controllers.

In summary, when using DSVerifier, a digital-system engineer must define the target UAV's system parameters (step 1) as represented by a transfer-function, implementation characteristics (steps 2 and 3), and verification settings (step 4). In particular, the overflow mode (step 4) can be defined as

[4]In particular, ESBMC represents the most prominent k-induction algorithm as reported in SV-COMP 2018: https://sv-comp.sosy-lab.org/2018/results/results-verified/.

[5]There is no proof of termination.

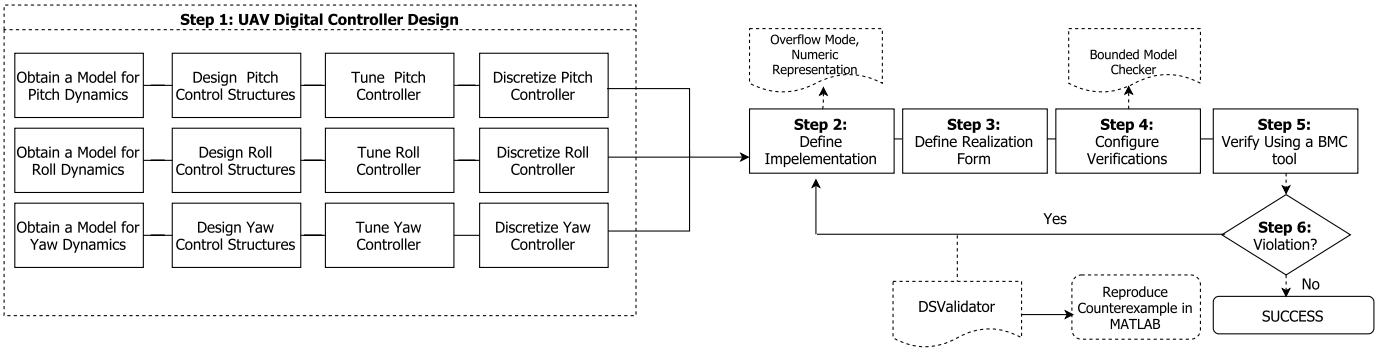


Fig. 6. The proposed methodology for UAV digital-system verification.

saturate or wrap-around, which then affects all computations and quantization operations. By default, in LCO verification, the overflow mode is set to wrap-around, in order to avoid saturation, which would then impair such a check. The intended UAV attitude-controller verification finally occurs in step 5, where an underlying model-checker is employed. Furthermore, DSVerifier provides verification results in step 6, which can be classified as "successful", if there is no property violation, up to a bound k, or "failed", if it indicates some violation along with a counterexample, which contains inputs and states that lead to the associated failure.

It is worth noticing that the actual digital-system verification only occurs in step 5, with the selected BMC tool and using two possible different solvers: an SMT one for ESBMC, called Boolector [65], or a Boolean Satisfiability (SAT) one for CBMC, called MiniSAT [66].

In step 6, DSVerifier checks violations in digital-controller implementations, considering the desired property. In particular, if the current verification fails, DSVerifier shows a counterexample with inputs, which can lead to the violated property. Finally, other implementation options (*i.e.*, realizations and representations) can then be evaluated, in order to avoid the related errors and thus find a suitable digital controller implementation.

C. Fixing Digital-Controller Implementations

DSVerifier does not automatically fix a digital controller implementation, if a property violation is found; however, it does provide a counterexample showing that a given property does not hold in a model. This counterexample then allows users *(i)* to analyze a failure, *(ii)* to understand an error, and *(iii)* to correct either the respective specification or the model, *i.e.*, the property and the controller that have been analyzed [67]. One may notice that this step requires manual intervention, unless automated synthesis procedures can be employed to repair a failure [52]. In addition, re-implementation procedures can be faster, when performed by experienced engineers, and, for this specific work and considering a specific hardware choice (*i.e.*, UAV attitude controller), tuning only three parameters (*i.e.*, number of bits, realization, and scaling) is enough to fix the majority of implementation-related problems usually found in those scenarios [25]. Additionally, some trade-offs have to be taken into account, when performing modifications related to a representation format:

- Increasing the number of bits of integer parts should fix overflow; however, if the maximum value for a given hardware platform is achieved, then it may be necessary to decrease the number of bits of the fractional ones;
- Decreasing the number of bits of fractional parts leads to an increase in quantization noise and, consequently, signal-to-noise ratio (SNR) problems [61];
- LCOs are very difficult to prevent. In particular, if a specific controller implementation presents LCO, another implementation of the same controller (changing the number of fractional bits) may not suffer from the same problem. Reciprocally, an implementation free from granular LCO may then present that same problem, if its number of fractional bits is changed [25], [61];
- Operations can be executed faster if less bits are employed (mainly in field-programmable gate arrays).

Regarding the effect of changing realizations, for the direct forms addressed in this work, it is important to notice that:

- DFI and TDFII present the same performance issues, regarding overflows in two-complement architecture with wrap-around, because only the final operation affects a system using those realizations [68], [69];
- DFII, in turn, needs verification after each equivalent adder (input and output) [68], [69];
- If saturation arithmetic is employed, when overflow occurs, not only the final result but also intermediate operations have the potential to affect a system's output, even if DFI and TDFII are employed [68], [69]. It means that all system-node operations have to be evaluated, during an overflow verification, and violations may occur for a controller implementation, in a specific realization form, and disappear when employing another one.

Finally, for the scaling effect:

- An appropriate scaling factor can prevent overflows, in stable systems, but such a result usually requires large attenuation, which can affect resulting SNR figures [61];
- Scaling can also prevent overflow LCOs.

When performing verification with DSVerifier [26], with the goal of checking LCO or overflow violations, a digital-system engineer is also able to analyze the impact of implementation aspects (number of bits in integer and fractional parts), realization forms (*i.e.*, direct forms), and scaling factors, in order to design robust digital systems, because the mentioned tool is able to verify them, with respect to those trade-offs.

Note that the overflow verification scheme employed here extends our previous studies, by considering both wrap-around and saturation modes [25]. Additionally, the current LCO verification presents some enhancements, which are explained in the next subsection. LCO verification is indeed a novelty and especially important for UAV attitude-control.

D. Overflow Verification for UAV digital controllers

When dealing with UAV digital controllers, we need to take care about overflow. In the present study, assertions are encoded into the quantizer block and the verification engine is configured to use nondeterministic inputs in a specified range, in order to detect overflow, for a given fixed-point word-length. For any arithmetic computation, if there exists a value that exceeds the representable range, an assert statement detects that as arithmetic overflow. As a consequence, a literal $l_{signed\_overflow}$ is generated, with the goal of representing the validity of each addition, subtraction, division, and multiplication operation, according to the constraint

$$l_{signed\_overflow} \Leftrightarrow (FP \geq MIN) \wedge (FP \leq MAX), \quad (3)$$

where *FP* is the fixed-point approximation, for the result of arithmetic computations, and *MIN* and *MAX* are the minimum and maximum values, which are representable for a given fixed-point bit format, respectively. Therefore, in overflow verification, an expression of a fixed-point type can not be out of the range provided by a fixed-point bit format. If this condition is violated, then overflow has occurred. In addition, arithmetic overflow events can be solved by saturation or wrap-around.

Algorithm 1 describes how DSVerifier [26] performs overflow verification. Firstly, it formulates an FWL-effects function and obtains the numerator and also the denominator of a digital controller, with those effects. Then, DSVerifier computes a transfer-function with FWL effects, retrieves its outputs, according to the employed realization form (*e.g.* ,DFI, DFII, or TDFII), and stores the respective results in a vector $y(n)$. After that, the maximum and minimum word-representations are verified, based on I-integer bits and F-fractional ones. Finally, it checks if values stored in $y(n)$ are inside the allowed range, according to the maximum and minimum representations. If a sample is outside that range, then DSVerifier returns "failed" together with a counterexample; otherwise, if no violation is found, it returns "successful" up to the given depth k.

1) Illustrative Example: In order to explain the proposed DSVerifier-aided verification methodology, the following second-order controller is used.

$$H(z) = \frac{60z - 50}{z} \quad (4)$$

In particular, for DFI and TDFII realization forms with wrap-around mode and according to the Jackson's rule [69], a system's output will not be affected by overflows in intermediate operations; however, in DFII realization form, if overflow occurs in the input adder (as can be seen in Fig. 3b) and that is not avoided, then the mentioned system's output can be incorrectly computed. Besides, in saturate mode, any overflow in intermediate operations will also affect its output. Indeed, DSVerifier can identify violations in intermediate nodes and if any problem is detected, then it automatically concludes the current verification and generates a counterexample with related inputs and outputs. For instance, operation for controller 4, with fixed-point format $\langle 6, 10\rangle$ and a DFI realization led to a violation in an intermediate node, when computing $y(10) = -46.9922$. In particular, an overflow violation in saturate mode occurs due to the minimum representable value, which is -32.00, during a multiplication by b_0 (see Fig. 3a). Finally, Table I shows values computed for each output from Eq. 4.

Algorithm 1: Overflow verification

Data: $N_C(z)$ as the controller numerator, $D_C(z)$ as the controller denominator and its output up to depth k.
Result: SUCCESS for the absence of overflows up to the depth k; otherwise, FAILED along with a counterexample.

```
begin
    Formulate a FWL effect function FWL[·];
    Obtain FWL[N_C(z)] and FWL[D_C(z)];
    Compute H(z) = FWL[N_C(z)] / FWL[D_C(z)];
    Obtain the outputs (y(n)) from H(z);
    Obtain the MIN and MAX representation given I-integer bits
      and F-fractional bits;
    MIN ← −2^I;
    MAX ← 2^I − 2^−F;
    for i ← 0 to n by 1 do
        if y(i) < MIN and y(i) > MAX then
            return FAILED and a counterexample (i.e., presence
              of overflow);
        end
    end
    return SUCCESS (i.e., free from overflows up to the depth k);
end
```

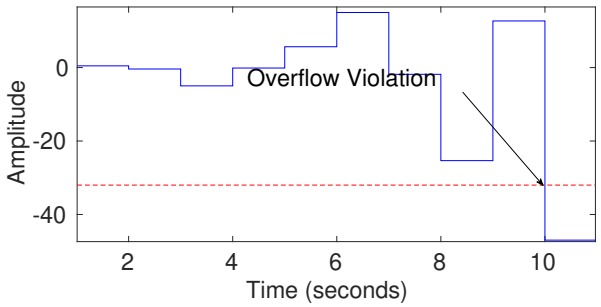


Fig. 7. Overflow in a second-order digital controller.

E. Limit Cycle Verification for UAV digital controllers

LCOs may be very harmful to digital control systems, given that they degrade control actions, cause damage to physical plants, harm surround products, and increase material losses. In particular, the presence of LCO violations, in UAVs, is related to flutter behavior in UAV wings [70].

The LCO verification scheme employed here extends previous DSVerifier versions [25], [26], where only zero-input LCO events were detected, by comparing past and current states. Indeed, now DSVerifier searches for the repetition of any output sequence caused by any non-deterministic constant input, with non-deterministic initial states, which allows verification for many other attitude-angle references (not only the zero one). In summary, this is a more realistic approach, once the

TABLE I
REPRODUCING AN OVERFLOW VIOLATION IN SATURATE MODE.

n	1	2	3	4	5	6	7	8	9	10
$y(n)$	0.46875	-0.390625	-4.980	-0.126	5.67	15.00	-1.83	-25.34	12.69	-46.9922

reference of an attitude system is variable and coupled to the device position dynamics: for each different target position, different attitude-angle references are generated.

The proposed LCO detection algorithm is implemented in DSVerifier, where a system's output computation is iteratively checked, according to the maximum bounded number of entries k. The latter is defined by users, while the constant input signal $x(n)$ and initial states are determined using non-deterministic values, according to the provided dynamic range. In order to verify the presence of LCO, in a particular digital controller realization, the quantizer block routine is configured by setting a flag variable on it, with the goal of enabling wrap around on overflow, which then avoids overflow detection. According to a specific realization, the LCO algorithm execution is then unrolled, for a bounded number of entries k, and an assert statement is added to detect a failure, if a set of previous outputs states (that repeat during a constant-input response) is found.

LCO occurrences are represented by a literal l_{LCO}, with the goal of determining whether a set of previous outputs is found, according to the constraint

$$\begin{aligned} l_{LCO} \iff \exists n, k \in \mathbb{N}, \exists c\mathbb{R} | x_m = c \implies \exists y_{k+i} = y_{k+n+i}, \\ \forall i \in \{0, 1, 2, ..., n\}, m \in \{k, k+1, k+2, ..., k+2n\}, \end{aligned} \tag{5}$$

where x_k and y_k are the k-th input and output samples, respectively. LCO absence is then verified by checking $\neg l_{LCO}$, that is, if there exists no execution where a set of previous outputs is found.

Algorithm 2 describes the steps employed for detecting LCO, as described in Eq. 5. Firstly, DSVerifier formulates an FWL-effects function that obtains a controller's numerator and denominator with those effects. Then, a new transfer function with FWL effects is obtained and its outputs are computed, based on the employed realization forms (*e.g.,* DFI, DFII and TDFII). After that, the algorithm selects the last outputs as reference and searches the same values in the previous elements, in order to compute the time window length for LCO. Finally, if a window has been found, then the algorithm verifies if the elements inside that are repeating. Whenever it occurs, LCO presence is identified; otherwise, DSVerifier returns "successful" for LCO verification, up to depth k.

Algorithm 2: Limit cycle verification

Data: $N_C(z)$ as the controller numerator, $D_C(z)$ as the controller denominator and its outputs up to k-depth.
Result: SUCCESS for the absence of LCOs up to the depth k; otherwise FAILED along with a counterexample.

```
begin
    Formulate an FWL effect function FWL[·];
    Obtain FWL[N_C(z)] and FWL[D_C(z)];
    Compute H(z) = FWL[N_C(z)] / FWL[D_C(z)];
    Obtain the last output from H(z), as reference;
    Check the presence of a time window;
    if size of time window is bigger than one then
        Check whether elements inside that time window are
          repeated;;
        if all elements are repeated then
            return FAILED and a counterexample (i.e., presence
              of LCO);
        end
    end
    else
        return SUCCESS (i.e., LCO-free up to the depth k);
    end
end
```

F. Illustrative Example

In order to explain the proposed DSVerifier-aided verification methodology, the second-order controller

$$H(z) = \frac{1.5610 - 1.485z^{-1}}{1 - 0.9z^{-1}} \tag{6}$$

is used.

Indeed, a transfer function definition corresponds to the first step of the proposed methodology, which is shown in Fig. 6. The second step is the choice of the FWL representation, whose fixed-point parameters are computed according to the method described by Carletta *et al.* [40] and considering an 8-bits hardware architecture. It allows the calculation of a (sufficient) number of bits to avoid overflow, using

$$j = \lceil \log_2(\|h\|_1 \cdot \|x\|_\infty) \rceil + 1, \tag{7}$$

where $\|h\|_1$ is the l_1-norm of a system's impulse response $H(z)$ and $\|x\|_\infty$ is the l_∞-norm of input k, that is, the maximum value that can be assumed by $x(k)$. Indeed, Carletta *et al.* claim that (7) is enough to prevent overflow in a system's output, which is true when two-complement is employed in wrap-around mode (the chosen mode), with DFI and TDFII, and is known as the Jackson's rule [69].

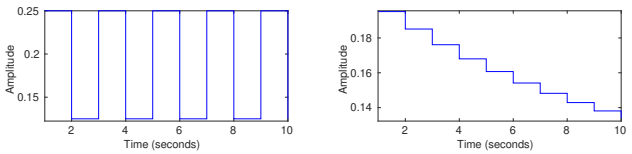
Using (7) for the system in (6), where $\|x\|_\infty = 3$, due to its dynamic range, and $\|h\|_1 \approx 1.9$, one may find that 4 bits are sufficient for its integer part. As a result, the representation $\langle 4, 4 \rangle$ is suggested, with a resulting range between -8 and 7.9375. In addition, it is worth noticing that the maximum value of a system's output is perfectly known, through $\|h\|_1$.

According to the DSVerifier's configuration, users must provide specifications in a ANSI-C file, as shown in Fig. 4, and define the desired DFII realization, in the third step. Then, a timeout of 1 hour and a bound of 10 cycles are set, given that limit cycle occurrences need to be verified.[6]

After a few seconds, the verification process is concluded and a failure (Step 6) is indicated. A persistent oscillation in this system's output is reported, for a constant input $x(k) = 0.125$ and an initial state $y(-1) = -2.875$. The resulting oscillation can be seen in Fig. 8a, with amplitude between 0.25 and 0.125. As a consequence, a designer should go back to Step 2 to avoid the limit cycle reported by DSVerifier, through

[6]The DSVerifier is invoked through command line as follows: `dsverifier filename.c --realization DFII --property LIMIT_CYCLE --x-size 10 --timeout 3600 --bmc CBMC`

a simple realization change. For instance, DFI could fix the controller's implementation, which would lead to a successful verification (Step 6), as can be seen in Fig. 8b.



(a) LCO occurrence for DFII. (b) Solving LCO Violation using DFI.

Fig. 8. Output of controller (6) implemented with format $\langle 4, 4\rangle$ for a constant input equal to 0.125.

V. UAV CONTROL SYSTEMS AND BENCHMARKS

A. Modeling UAV Attitude Dynamics

In terms of modeling, the body fixed-frame B and the earth fixed-frame E are illustrated in Fig. 9. B represents the angular movements (inertial reference system) pitch (θ), roll (ϕ), and yaw (ψ), while E describes quadrotor translational movements in a three-dimensional space.

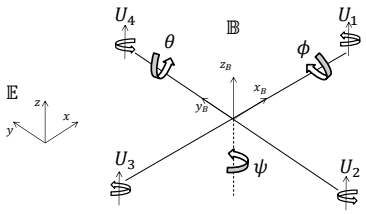


Fig. 9. Reference system for a quadcopter model [71].

As a result, the physical plant described by (8) was obtained, based on a computational tool for systems identification available in MATLAB [72]. The respective model consists in an autoregressive with Exogenous Inputs (ARX) structure and presenting two poles and one zero.

$$G_1(z) = \frac{-0.06875z^2}{z^2 - 1.696z + 0.7089}. \tag{8}$$

B. Control Strategies in UAVs

Fig. 10 shows a typical digital control-system for UAVs, which can be divided into attitude, altitude, and position controls. Typically, a high-level controller provides coordinates that contain reference values regarding position and altitude; however, they are coupled to the attitude dynamics and depend on angle variations. This way, position and altitude controllers generate references to the attitude control system, which then drives UAV motors. The attitude of a quadrotor consists in its orientation w.r.t. an inertial reference system, which is described by the Euler angles: pitch, roll, and yaw [73]. The present work tackles only attitude controllers.

One of the best-known control strategies available in literature is the proportional-integral-derivative (PID) control, which is shown in Fig. 10. In that approach, the controller output $u(t)$ represents a response to the obtained error $e(t)$, with respect to

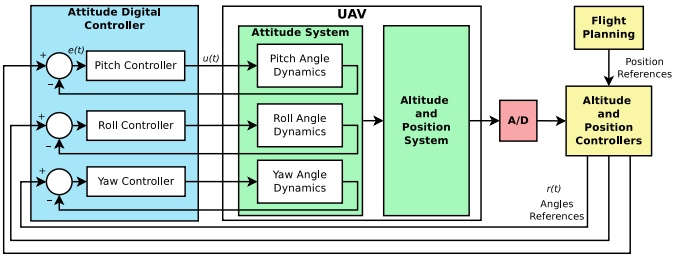


Fig. 10. A typical digital control system for UAVs.

a reference $r(t)$ and measured sensor signals, which is proportional to the error itself (P), its derivative (D), and also its integral (I). Additionally, a controller might contain only some of those: proportional-derivative (PD), proportional-integral (PI), and proportional (P) controllers, where integrative, derivative, and both actions are null, respectively. In general, a continuous PID controller has its response represented by

$$u(t) = K_P e(t) + K_D \frac{\mathrm{d}e(t)}{\mathrm{d}t} + K_I \int_0^t e(t)\mathrm{d}t, \tag{9}$$

where K_P, K_D, and K_I, are the proportional, derivative, and integrative gains, respectively.

The effort necessary to design a PID controller may be reduced to merely tuning its gains, *i.e.*, K_P, K_D, and K_I. Here, PID controllers for attitude UAV control were designed using empirical Ziegler-Nichols tuning [60] and CounterExample Guided Inductive Synthesis (CEGIS) [52]. A PID controller structured as (9) can then be represented by a continuous transfer function

$$C(s) = \frac{K_D s^2 + K_P s + K_I}{s}. \tag{10}$$

1) Synthesizing UAV attitude controllers with CEGIS: In order to synthesize controllers for a UAV system, given a continuous plant, a tool named as DSSynth [52] can be used, which is a program synthesizer that implements CEGIS for synthesizing digital controllers [55]. Given a plant model for roll (ϕ), pitch (θ), and yaw (ψ) angle dynamics, which is expressed in ANSI-C syntax, DSSynth constructs a non-deterministic model to represent that plant family, *i.e.*, it addresses plant variations as interval sets and formulates a function using implementation details, with the goal of computing a group of controller parameters to be synthesized. Then, DSSynth synthesizes the respective controller coefficients for a given implementation specification, *i.e.*, numerical representation and realization form, and, finally, it builds intermediate C code representing a digital system for an UAV, which is used as input for the CEGIS engine.

For instance, by employing the controller-synthesis methodology described by Abate *et al.* [52], using DSSynth, the stabilizing controller in (11) was synthesized for the attitude dynamics module [29] described in (8), with a sampling time of $0.002s$.

$$H(z) = \frac{-0.39154052734375z^2 - 0.7646636962890625z}{0.8602752685546875z^2 + 0.52484130859375z} \tag{11}$$

C. Description of Benchmarks

UAV modeling is a hard task, given that such a kind of system presents many nonlinearities and complex structures. Generally, its control system is reasonably sectioned, with high interdependence among attitude, altitude, and position.

The present experiments were performed on a quadcopter system, whose model was described by Bouabdallah *et al.* [71]. Such an investigation focuses on the attitude control system, *i.e.*, the control of angular movement, through adjustment of the pitch (θ), roll (ϕ), and yaw (ψ) angles. Indeed, an attitude control system is the basis for quadcopter stabilization and its reliability is needed for a correct operation of attitude and position control systems. As mentioned before, Fig. 9 shows a quadcopter's attitude angles (θ, ϕ, and ψ) and the associated cartesian-position (x, y, and z) references.

Any attitude control system aims to provide stabilization, along with reference tracking. Five strategies are employed here: combined PD/PD, combined PID/PD, combined PD/P, combined PD/PI, and PID control. The first strategy consists of two control loops for each angle, *i.e.*, one for angular velocity and another for the orientation angle itself: on each loop, a PD controller is employed. The second one is very similar to the first, but the angle-control loop does employ a PID controller. The third one, in turn, employs a PD controller for angle control and a P rate controller, while the fourth one occurs when the angle control loop employs both PI and PD controllers. Finally, the last strategy uses only one loop with a PID controller. Fig. 11 shows the controlling structure of the PD/PD, PD/PI, PD/P and PID/PD strategies, while Fig. 12 shows the specific approach adopted for the PID one.

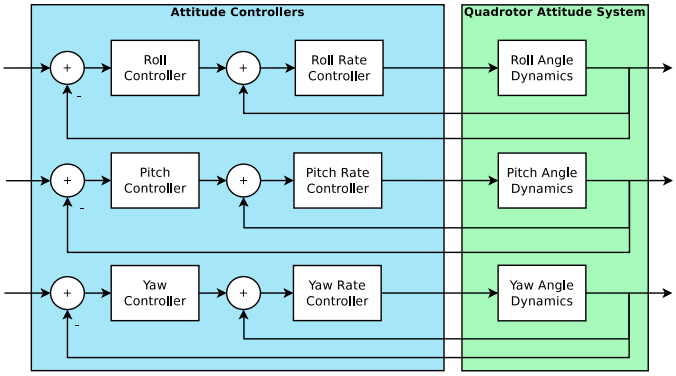


Fig. 11. Attitude control system with combined structure.

In the control strategy shown in Fig. 11, two controllers are employed for each angle, where the inner one is used for stabilizing angular rate, *i.e.*, roll rate ($\dot{\phi}$), pitch rate ($\dot{\theta}$), and yaw rate ($\dot{\psi}$) controllers, by computing the control torque around the x (u_x), y (u_y), and z (u_z) axes, respectively. The outer controllers (roll, yaw, and pitch) are used for stabilization and reference tracking of attitude angles (ϕ,θ, and ψ), by computing angular rate references ($\dot{\phi}_{ref}$, $\dot{\theta}_{ref}$, and $\dot{\psi}_{ref}$), which are provided to the inner control system.

By contrast, Fig. 12 shows a control strategy that employs a single controller for each attitude angle, which thus directly computes the control torques u_x, u_y, and u_z, by means of the roll, pitch, and yaw PID controllers, respectively. In some

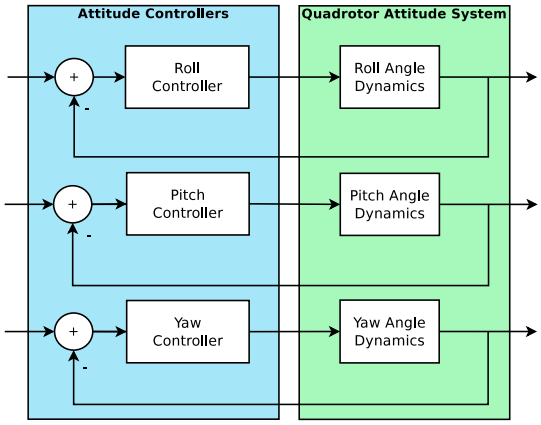


Fig. 12. Attitude control system with only one PID controller, for each angle.

specific cases, the same controller can be employed for different angles, especially roll and pitch, whose normal behavior is usually identical. In summary, ten different controllers were designed for the proposed control strategies, which were tuned in continuous time and then converted into digital format, with different sample times and methods.

In addition, this work evaluates transfer functions of real digital controllers, which were employed for UAV attitude control by Frutuoso *et al.* [29]. The dynamic models related to the attitude angles were obtained via an identification process based on the least-squares algorithm. All controllers studied in this paper present order less or equal to 2, which is common in the digital-control area. Although higher order controllers are not used in this paper, our benchmarks are representative of UAV attitude control systems, since they are indeed extracted from real UAV systems.

Table II shows the association of each controller, regarding its function (Figs. 11 and 12), while Table III describes all designed controllers, with their tuning gains in continuous time, transformation methods, and sample times. The chosen number of bits, associated to each implementation, is based on the methodology presented by Carletta *et al.* [40], who suggested a computation based on the impulse response sum. In particular, C_5 does not employ the mentioned methodology, because the current UAV architecture supports only 16 bits and it would require at least 17 bits. As a consequence, C_5 can be seen as an example of design failure, in such a way that the impact of FWL effects, on the implementation of fixed-point digital controllers, are promptly detected and analyzed.

TABLE II
CONTROLLER DISTRIBUTION FOR THE ADOPTED CONTROL STRATEGIES.

Control Strategy	PD/PD Control	PID/PD Control	PID Control	PD/PI Control	PD/P Control	DSSynth Controller
Roll	C_2	C_4	C_5	C_7	C_9	C_{10}
Roll Rate	C_1	C_1	-	C_6	C_8	-
Pitch	C_2	C_4	C_5	C_7	C_9	C_{10}
Pitch Rate	C_1	C_1	-	C_6	C_8	-
Yaw	C_3	C_4	C_5	C_7	C_9	-
Yaw Rate	C_1	C_1	-	C_6	C_8	-

TABLE III
DIGITAL CONTROLLERS FOR THE EVALUATED QUADROTOR ATTITUDE SYSTEM.

Controller ID	Tuning Gains K_P	K_D	K_I	Discretization Method	Sample Time (ms)	Discrete Transfer Function
C_1	1	0.01	-	Forward Euler (FE)	20	$\frac{1.5z-0.5}{z}$
C_2	10	1	-	Forward Euler (FE)	20	$\frac{60z-50}{z}$
C_3	10	2	-	Forward Euler (FE)	20	$\frac{110z-100}{z}$
C_4	10	2.5	0.5	Forward Euler (FE)	20	$\frac{135z^2-260z+125}{z^2-z}$
C_5	2	1	0.1	Tustin (Bilinear)	1	$\frac{2002z^2-4000z+1998}{z^2-z}$
C_6	5.5	0.0465	-	Tustin (Bilinear)	20	$\frac{0.93z-0.87}{z+1}$
C_7	0.4	-	50	Tustin (Bilinear)	20	$\frac{0.1z-0.09998}{z-1}$
C_8	0.3	0.009	-	Backward Euler (BE)	2	$\frac{0.0096z-0.009}{0.002z}$
C_9	0.1	-	-	Backward Euler (BE)	2	$\frac{0.1z-0.1}{z-1}$
C_{10}	-	-	-	Contoller Synthesized	2	Cf. (11)

VI. EXPERIMENTAL EVALUATION

A. Experimental Objectives

In order to verify the impact of saturation effects in intermediate operations, regarding different realization forms (*i.e.*, DFI, DFII, and TDFII), overflow-checking experiments were executed, considering both saturate and wrap-around modes. In saturate mode, an overflow violation can be detected in intermediate nodes, during intermediate operations (*i.e.*, sums and multiplications) of a realization form; otherwise, in wrap-around mode, overflow detections take into account only system outputs, since previous studies [68], [69] showed that if a digital system, implemented with 2's complement arithmetic and using DFI or TDFII, does not present overflow on its final result, it will not be affected by overflow in intermediate operations. Such a behavior is a direct consequence of the Jackson's rule [68], [69] and is extensively used in digital systems, in order to simplify designs and minimize FWL effects, given that all quantizers are configured to wrap-around. In particular, for DFII, overflow detection must also be checked during a specific intermediate operation, as will be explained in the next paragraph.

According to parts (a) and (c) of Fig. 3 (DFI and TDFII realizations, respectively), multiplier outputs are directly connected to equivalent adders (disregarding delays), which means that the Jackson's rule is valid for those realizations forms. By contrast, part (b) (*i.e.*, Direct Form II realization) shows two equivalent adders (input and output) connected through a multiplier (b_0), which also means that the Jackson's rule is still valid for each equivalent element; however, the output of the input adder must not overflow. If that is not avoided, then the result of the output adder may be incorrect, as previously mentioned.

For the implementations tested in this work, signal inputs lie between -1 and 1, that is, a sensor's (gyroscope) output bound in normal conditions, which means that inputs employed during verification of LCO and overflow violations also fall within such a range.

In summary, our experimental evaluation aims to answer two research questions:

- **RQ1:** How digital controllers designed for UAV attitude systems are susceptible to violations, according to fixed-point representations and realizations?
- **RQ2:** Are verification results using BMC sound and can their overflow and limit cycle violations be reproduced and also validated by external tools (*e.g., MATLAB)?*

B. Experimental Setup

In the present work, DSVerifier v2.0.3 was used to check controllers described in Sec. V-C. The ones in Table III were verified with 3 different numerical formats (with at least the number of integer bits suggested by Carletta *et al.* [40]) and 3 different realizations (DFI, DFII, and TDFII), for each one. C_{10}, in turn, was synthesized with DSSynth and verified with only one format, but using the same 3 different realizations. As a result, there are 84 different verification tasks for each evaluated property (overflow and LCO), which aim to investigate the importance of realization forms and numerical formats, regarding FWL performance.

The present experiments were executed on an otherwise idle computer with the following configuration: Intel Core i7$-$2600 3.40 GHz processor, 24 GB of RAM, and Ubuntu 64-bits OS. CBMC v5.5 was employed and the maximum timeout was set to 3600s. All presented execution times are CPU times, *i.e.*, only time periods spent in allocated CPUs, which were measured with the `times` system call (POSIX system).

C. Experimental Results

Table IV shows the obtained verification results, where *VT* denotes the verification time, in seconds, *VR* represents the verification result, *S* means success, that is, DSVerifier did not find a failure up to $k = 10$, *F* means failed, that is, DSVerifier found a property violation and then returned a counterexample, and, finally, *T* means timeout, that is, DSVerifier exceeded the maximum verification time. All digital controllers were verified with implementations on an ATMEGA328, which is based on a 16-bits processor driven by a 16 MHz clock.

The larger times regarding LCO verification procedures, as shown in Table IV, are explained by the high-complexity algorithm employed for that, with non-deterministic initial states, (constant) inputs, and variable oscillation periods.

Fig. 13 summarizes the obtained verification results, for each realization form, which show that 28,6% of our controller implementations presented overflow, when checked by DSVerifier in wrap-around overflow mode, *i.e.*, an overflow violation was detected only in system outputs. In saturate mode, verification procedures failed for 38,1% of our controller implementations, which means that an overflow violation was detected in intermediate nodes, during verification procedures. In LCO verification, 34,5% of our controller implementations failed, while 56% of them presented successful verification and 9,5% led to timeout. Despite that, LCO verification procedures were concluded for 90,5% of the chosen benchmarks, while overflow ones were concluded for 99% and 100% of them, in wrap-around and saturate mode, respectively.

TABLE IV
VERIFICATION RESULTS FOR THE DIGITAL CONTROLLERS USED IN THE MENTIONED QUADROTOR ATTITUDE SYSTEM. *VR*: VERIFICATION RESULT (*S* - SUCCESSFUL, *F* - FAILED, OR *T* - TIMEOUT), *VT*: VERIFICATION TIME (IN SECONDS), FORMAT $\langle k, l\rangle$, WHERE *k* IS THE INTEGER BITS, AND *l* IS THE FRACTIONAL BITS.

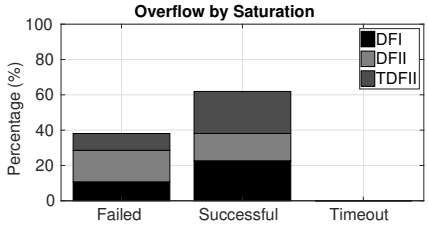
ID	Format $\langle k, l\rangle$	Overflow - Saturate Mode						Overflow - Wrap-Around Mode						Limit Cycle					
		DFI		DFII		TDFII		DFI		DFII		TDFII		DFI		DFII		TDFII	
		VR	VT	VR	VT	VR	VT	VR	VT	VR	VT	VR	VT	VR	VT	VR	VT	VR	VT
C_1	$\langle 2,14\rangle$	F	5	F	10	F	6	F	12	F	10	F	8	S	35	S	52	S	367
	$\langle 4,12\rangle$	S	5	S	4	S	4	S	3	S	4	S	3	S	21	S	30	S	576
	$\langle 6,10\rangle$	S	4	S	5	S	4	S	4	S	3	S	5	S	19	S	23	S	197
C_2	$\langle 6,10\rangle$	F	8	F	7	F	8	F	10	F	9	F	8	S	33	S	32	S	510
	$\langle 8,8\rangle$	S	5	S	4	S	5	S	4	S	4	S	4	S	14	S	22	S	150
	$\langle 10,6\rangle$	S	4	S	5	S	4	S	3	S	4	S	4	S	9	S	11	S	57
C_3	$\langle 7,9\rangle$	F	8	F	6	F	8	F	11	F	9	F	8	S	22	S	33	S	338
	$\langle 9,7\rangle$	S	4	S	5	S	3	S	4	S	3	S	4	S	11	S	17	S	113
	$\langle 11,5\rangle$	S	5	S	5	S	5	S	3	S	3	S	4	S	9	S	10	S	30
C_4	$\langle 8,8\rangle$	F	9	F	7	F	8	F	14	F	12	F	12	S	35	T	3600	T	3600
	$\langle 10,6\rangle$	F	15	F	10	S	670	T	3600	S	14	S	436	S	42	T	3600	T	3600
	$\langle 11,5\rangle$	S	236	F	13	S	97	S	86	S	11	S	36	S	54	T	3600	S	1823
C_5	$\langle 10,6\rangle$	F	8	F	7	F	7	F	6	F	5	F	5	F	21	F	17	F	91
	$\langle 12,4\rangle$	F	8	F	10	F	6	F	5	F	6	F	4	F	16	F	33	F	22
	$\langle 13,3\rangle$	F	11	F	8	F	12	F	6	F	5	F	5	F	13	F	18	F	50
C_6	$\langle 4,12\rangle$	F	18	F	12	F	14	F	5	F	5	F	8	F	15	F	14	F	13
	$\langle 8,8\rangle$	S	15	S	15	S	19	S	5	S	5	S	6	F	16	F	12	F	15
	$\langle 10,6\rangle$	S	16	S	14	S	16	S	4	S	4	S	4	F	10	F	12	F	12
C_7	$\langle 4,12\rangle$	S	17	F	15	S	24	S	7	S	5	S	10	S	86	F	29	T	3600
	$\langle 8,8\rangle$	S	14	S	12	S	18	S	5	S	5	S	9	S	33	F	57	S	661
	$\langle 10,6\rangle$	S	13	S	12	S	17	S	5	S	5	S	4	S	30	F	98	S	248
C_8	$\langle 3,13\rangle$	S	7	F	10	S	7	S	2	S	2	S	2	S	44	S	76	T	3600
	$\langle 4,12\rangle$	S	7	F	8	S	5	S	1	S	2	S	2	S	47	S	55	S	2092
	$\langle 5,11\rangle$	S	6	F	8	S	5	S	1	S	2	S	1	S	29	S	27	S	662
C_9	$\langle 4,12\rangle$	S	18	F	14	S	15	S	7	S	6	S	10	S	73	F	37	T	3600
	$\langle 8,8\rangle$	S	15	S	12	S	19	S	5	S	5	S	8	S	32	F	69	S	787
	$\langle 10,6\rangle$	S	14	S	10	S	18	S	5	S	5	S	4	S	26	F	66	S	234
C_{10}	$\langle 8,8\rangle$	S	31	S	16	S	43	S	33	S	23	S	30	F	47	F	279	F	95

1) Overflow Occurrence Discussion: Particularly, the digital controller C_5 presented overflow in every (possible) implementation, realization, and overflow mode (saturate and wrap-around). That happened because (7) suggested at least 17 bits [40], but the UAV architecture used for the experiments supports only 16 bits.

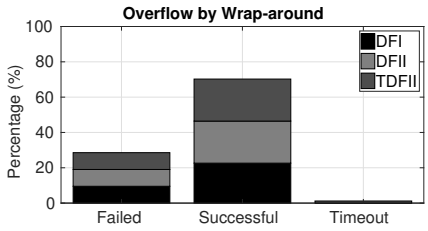
There was only one unfinished overflow verification due to timeout, when checking in wrap-around mode: a task started for controller C_4 with DFI realization and fixed-point format $\langle 10, 6\rangle$. Indeed, the latter can be explained by the high order of C_4, which requires more operations for computing system outputs. By contrast, in saturate mode, its verification failed in 15 seconds, with the same system specification.

Overflows may be avoided by changing bit format implementations or, specifically regarding saturate mode, by changing realization forms. As an example, overflow occurs for the digital controller C_1, in all realization forms, when it is implemented in fixed-point format $\langle 2, 14\rangle$. For that particular numerical format, that happens when the output $y(t)$ is less than -2 or greater than 1.999, according to Carletta's rule [40]. Fig. 14 shows an overflow failure, for this C_1 implementation, in which the graph on the left illustrates the input sequence provided by DSVerifier and the graph on the right corresponds to the controllers's output, *i.e.*, the controlling torque (u_x, u_y, or u_z) produced by the rate controller. In addition, red dashed lines denote representation limits, indicating when controlling UAV torques suffer from saturation, for this specific implementation. One may notice that, in the first sample, the output is slightly greater than the numerical representation limit; however, in Fig. 15, the output torque does not contain overflow violation, using the same controller specified for C_1 and fixed-point format $\langle 10, 6\rangle$.

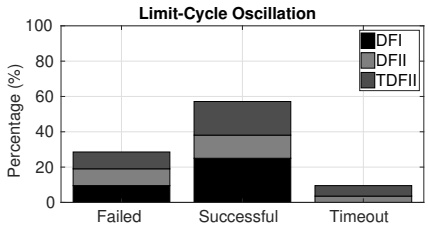
It is worth noticing that there were overflow occurrences in DFI and TDFII realizations with the first format (the one that follows exactly what was computed) of many controllers, in wrap-around mode, which contradicts what is presented by the jackson's rule. As a consequence, a deeper investigation was conducted and an interesting behavior was noticed: the method presented by Carletta *et al.* [40] does not guarantee complete absence of overflow events, in certain cases, which occurs because it does not considers the asymmetry of two's complement representations, as shown by Volkova *et al.* [43]. For instance, when the impulse response summation ($\|h\|_1$) is 2 and the maximum input is bounded by -1 and 1 (bounds included), which means that the output range lies between -2 and 2, the mentioned method returns a format with 2 bits. Nonetheless, the resulting two-complement range provided by such a representation will allow values between -2 and 1, which is not enough. Indeed, such a behavior was noticed when the maximum output value is a power of 2, due to the



(a) Overflow by Saturation.

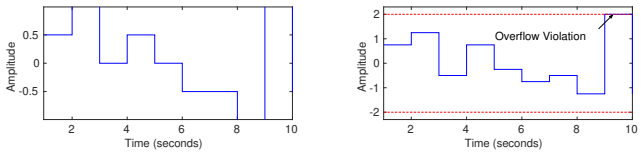


(b) Overflow by Wrap-around.



(c) Limit-cycle Oscillation

Fig. 13. Summary of the obtained verification results, per realization form, for the evaluated controllers.

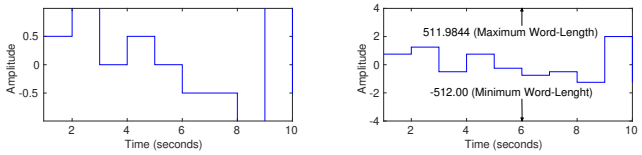


(a) Inputs (b) Outputs

Fig. 14. Arithmetic overflow occurrence in controller C_1, with DFI realization and a format containing 2 bits in the integer part and 14 bits in the fractional one.

way it is computed by the logarithm in (7).

Although the mentioned finding revealed a flawed dimensioning procedure, it further proved that the proposed methodology is sound and reliable. As future work, a correct dimensioning procedure can be employed (or even developed), which would allow the choice of a correct number of bits for the integer and fractional part.



(a) Inputs (b) Outputs

Fig. 15. Absence of arithmetic overflow in controller C_1, with DFI realization and a format containing 10 bits in the integer part and 6 bits in the fractional one.

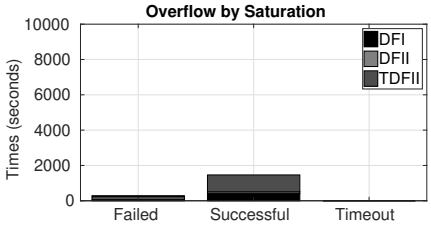
Additionally, some controllers presented overflow, even with formats with many integer bits (the second and third ones), in saturate mode. Examples of such a behavior are controllers C_4 and C_8, which presented overflow even when using numerical formats with more integer bits, i.e., $\langle 11, 5\rangle$ and $\langle 5, 11\rangle$, respectively. That was expected, given that intermediate operations are checked in saturate mode and the adopted dimensioning procedure only takes into account system outputs.

Finally, one may notice that C_4 presented the highest verification time, which occurred due to the high order associated to its implementation. Verification times typically increase with controller complexity, given that direct-form implementations need two-nested loops to generate a controller order. It is worth noticing that C_5 is also a second-order system, but it presents low verification times. Indeed, that is an expected result, once verification procedures for such a controller found errors in all implementations, since property refutation is typically faster than property correctness.

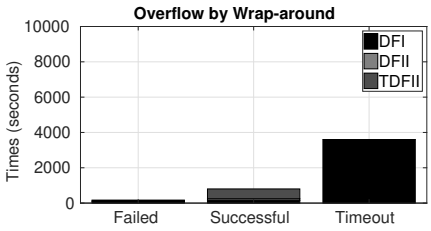
2) Limit Cycle Occurrence Discussion: An example of LCO occurrence was noticed (see Table IV) for the attitude angle controller C_5 in TDFII realization and with format $\langle 13, 3\rangle$. DSVerifier was able to find a violation for initial states -0.125, -0.0625 and 0.000, with a constant zero-input, as shown in Fig. 17a. The red dashed line represents the input sequence and the blue continuous one the controller's response. Besides, one may also notice that the same figure indicates an oscillation on C_5's output (the controlling torque u_x, u_y, or u_z) between -0.125 and 0.0625, *i.e.*, the violation indicates that the controller might produce oscillating torques, when it should maintain the UAV movement (*e.g.*, in hovering state).

If the same controller is implemented in DFI format, LCO events are also detected by DSVerifier. In particular, DSVerifier found that initial states -0.109375, 0.015625, and -0.1250 and an input sequence -0.0625 lead to the mentioned LCO. Fig. 17b shows an LCO occurrence in C_5, with DFI realization, which indicates an output (torque) oscillation between -0.1250 and 0.015625, *i.e.*, the attitude angle controller C_5 might produce torque oscillations during a pitch, roll, or yaw command, which is performed for any UAV displacement. Indeed, this same controller also presents overflow, as shown in section VI-C1, so the LCO occurrences illustrated in Figs. 17a and 17b are expected, given that an overflow event can generate LCO on system outputs, which is known as overflow LCO. By contrast, C_7 and C_9 present LCO in DFII realizations with no overflow, *i.e.*, granular LCO occurrences were identified.

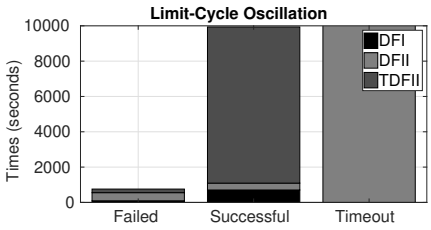
In LCO verification, C_4 implementations took a reasonable amount of time. C_4 is a second order system, which means that many non-deterministic initial states are considered and there are more mathematical operations, which consequently increase the model checking computing cost. In fact, LCO verifications tend to take longer, due to their algorithmic complexity, *i.e.*, a search for persistent oscillations in a system's output, based on combinations of non-deterministic constant input, initial states, and oscillation window size. It is worth noticing that verification times for C_5, which is also a second-order system, are much shorter than what is obtained with C_4. That happens because failed verifications are generally faster than successful ones. In fact, the proposed verification



(a) Overflow by Saturation.

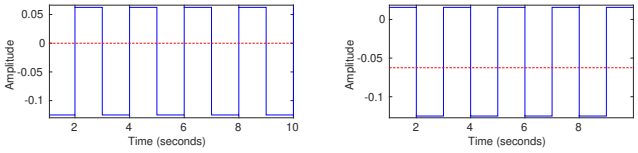


(b) Overflow by Wrap-around.



(c) Limit-cycle Oscillation

Fig. 16. Verification-time results for the digital controllers in the mentioned quadrotor attitude system.



(a) TDFII realization. (b) DFI realization.

Fig. 17. LCO occurrence in the control torque produced by C_5, with different realizations and format $\langle 13, 3 \rangle$.

algorithm is interrupted when a failure is found.

3) Evaluation of Verification Bounds: In order to evaluate our DSVerifier-based approach with different k-bounds, we have considered verification of overflow and LCOs properties with $k = 5, 20, 30$. We have also evaluated the k-induction proof rule to compare both verification techniques: bounded and unbounded model checking, with the same experimental setup previously configured for $k = 10$. Verification results for different k-bound values are illustrated in Figs. 18-20.

For $k = 5$, the respective results are described in Tables V-VII on the first column. If those are compared with the ones obtained for $k = 10$, which are shown in Tables V-VII on the second column, one may notice a lower number of timeout events, regarding LCO and overflow by wrap-around checks, which happens because $k = 5$ presents a reduced state space to DSVerifier. The number of successful results for overflow checks is also higher, when compared with what was obtained for $k = 10$, given that some property violations were not found. Nonetheless, it is interesting to notice that the number

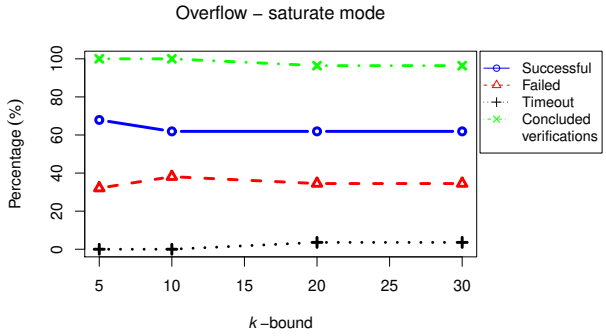


Fig. 18. Experimental results for overflow verification in saturate mode, for different k-bound values.

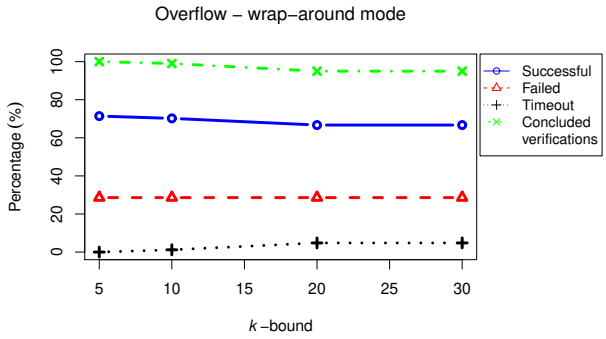


Fig. 19. Experimental results for overflow verification in wrap-around mode, for different k-bound values.

of failed LCO verification procedures, for $k = 5$, was higher, because some of them were not interrupted by timeout. Indeed, one could consider the adoption of a lower k, when checking LCO, in order to ensure a correct system behavior.

For $k = 20, 30$, the obtained results were the same, as can be seen in the Tables V-VII on the third and the last columns; however, the number of timeout events was much higher for all verification procedures, *e.g.*, four times higher for overflow in wrap-around mode. No limit-cycle check was even completed, because $100,0\%$ of our set of experiments produced timeouts for that. Indeed, such a result is not surprising, given the

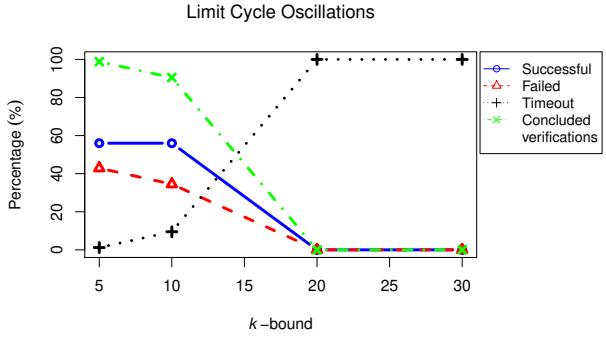


Fig. 20. Experimental results for LCO verification, with different k-bound values.

TABLE V
EXPERIMENTAL RESULTS FOR OVERFLOW BY SATURATE.

k-bounds	$k = 5$	$k = 10$	$k = 20$	$k = 30$
Successful	67,9%	61,9%	61,9%	61,9%
Failed	32,1%	38,1%	34,5%	34,5%
Timeout	0,0%	0,0%	3,6%	3,6%
Concluded Verification	100,0%	100,0%	96,4%	96,4%

complexity of LCO verification in digital controllers.

TABLE VI
EXPERIMENTAL RESULTS FOR OVERFLOW BY WRAP-AROUND.

k-bounds	$k = 5$	$k = 10$	$k = 20$	$k = 30$
Successful	71,4%	70,2%	66,7%	66,7%
Failed	28,6%	28,6%	28,6%	28,6%
Timeout	0,0%	1,2%	4,8%	4,8%
Concluded Verification	100%	99%	95%	95%

TABLE VII
EXPERIMENTAL RESULTS FOR LIMIT-CYCLE OSCILLATION.

k-bounds	$k = 5$	$k = 10$	$k = 20$	$k = 30$
Successful	56,0%	56,0%	0,0%	0,0%
Failed	42,9%	34,5%	0,0%	0,0%
Timeout	1,2%	9,5%	100,0%	100,0%
Concluded Verification	98,8%	90,5%	0,0%	0,0%

In addition, some results changed from $k = 10$ to $k = 20, 30$. In particular, for $k = 10$, we found overflow in implementations with formats $\langle 10, 6\rangle$ and $\langle 11, 5\rangle$, when DFII was used for verifying controller C_4; however, for $k = 20, 30$, with the same configuration for controller C_4, DSVerifier returned that those implementations did not present overflow due to timeout events. We suspect that the search strategy employed by the chosen SAT solver did not find property violations, within the given time limit, due to the larger state space that arose from $k = 20, 30$, if compared with the one from $k = 10$. In summary, DSVerifier was able to find more bugs with a lower number of inputs ($k = 10$) and, when we employed $k = 20, 30$, timeout events increased considerably as illustrated on Fig. 20. Figs. 18-20 show that the violation detection rates (failed results) do not increase for k-bounds values larger than 10 in any verification category, as well as the concluded verification rates. Indeed, we noted that for the results illustrated in Fig.20 (LCO verification), no new failures were detected by DSVerifier; as a consequence of a k-bound higher than 10, DSVerifier produces more timeouts. Based on what was presented, one may conclude that $k = 10$ is an interesting choice for our set of benchmarks and it provides a good trade-off between behaviour check and evaluation time.

When using k-induction, whose results are shown in Table VIII, the numbers of violations regarding overflow in saturate and wrap-around mode were the same; however, no successful case was ever found, which is consistent with the clear increase in the number of associated timeouts. Indeed, due to the lack of inductive invariants, the inductive step of the k-induction proof rule was unable to prove correctness of any property. The identification of inductive invariants is an active research area in the verification community [74]–[77] and we have not exploited that branch of research yet, regarding control software [77] verification. Additionally, no LCO verification was completed.

TABLE VIII
EXPERIMENTAL RESULTS USING k-INDUCTION

Property Evaluated	Overf. Saturate	Overf. Wrap-around	LCO
Successful	0,0%	0,0%	0,0%
Failed	28,6%	28,6%	0,0%
Timeout	71,4%	71,4%	100,0%
Concluded Verification	29%	29%	0,0%

We have also noticed that the k-induction algorithm detected overflow violations only in implementations where overflow was already expected, due to wrong estimation regarding Carletta's criterion [40] and independing on overflow mode, as can also be seen in Table VIII (the mentioned results are equal). For instance, controllers C_1, C_2, C_3, C_4, C_5 and C_6 had already presented overflow in the first check round, which means we would find the same bugs in those controllers, with k-induction.

For our set of benchmarks and associated experiments, with different k-bounds, it is possible to conclude that as k increases, the number of timeouts also gets higher. In particular, for $k = 10$, DSVerifier was able to find more failures with less timeouts, while for $k = 20, 30$, the number of timeouts increased considerably. Indeed, *k*-induction is used here as an iterative deepening style, starting with $k = 1$ and then incrementally increasing k to either find a property violation or prove partial correctness (without actually fully unwinding loops). It means that, in order to prove correctness, the inductive step of our *k*-induction proof rule is independent on controller order, but the base case still needs to fully unwind loops to expose bugs. One may notice further that *k*-induction is a computationally expensive approach, because three checks are performed for each k, *i.e.*, base case, forward condition, and inductive step (cf. Section IV-A), where the latter is the most expensive one. As a result, given that we search deeper in a system's state-space, we will eventually reach some property violation (if they really exist), prove partial correctness, or exhaust time and memory resources; however, since we are unable to produce inductive invariants to instrument our UAV software, our *k*-induction proof rule does not seem to be an effective approach to check properties that depend on system inputs [7], given that a substantial amount of time is spent in the inductive step.

Nonetheless, our *k*-induction experiments were inconclusive, since this algorithm was unable to prove correctness for all reachable states of a controller (*i.e.*, the procedure did not terminate, possibly due to large state space search). Indeed, the employed *k*-induction algorithm was able to find the same violations (with the respective counterexamples) as with the plain BMC procedure $B(k)$; however, it consumed more time and memory when trying to prove correctness. There are verification tools (*e.g.*, Impara [78]) that implement

[7]There are other properties in control software, such as stability and minimum-phase, which do not depend on the system inputs [26].

the interpolation and SAT-based model checking approach described by McMillan [79], but we have observed that it does not lead to better results, when compared with those presented by the *k*-induction algorithm, as previously reported in the international software verification competition (SV-COMP) [62]. We could further investigate a "Property-directed Reachability" (or IC3) procedure [80] for safety verification of digital controllers, but we have not found any software tool that is publicly available for verifying safety properties in full C programs. Besides, implementing such an algorithm would demand time and effort and, as a consequence, we decided to leave it as future work.

4) Synthesizing Digital Controllers and Analysing the Impact of LCO and Overflow Effects: DSVerifier was used to find overflow (in saturate and wrap-around modes) and LCO violations in the synthesized controller C_{10}, using fixed-point representation $\langle 8, 8\rangle$ and direct-form realizations (*i.e.*, DFI, DFII and TDFII). As result, the proposed algorithm found no overflow violations; however, in all employed direct forms, DSVerifier found LCO on system outputs, which means that DSVerifier can detect granular LCOs for closed-loop systems, even though a stable synthesized-controller is used. The outputs produced by DSVerifier, in one of the counterexamples obtained for C_{10}, are graphically represented in Fig. 21, which shows granular LCO for a DFI realization form, with constant inputs $x = 0.0$ and initial states $y_{-1} = -0.99609375$ and $y_0 = 0.00390625$.

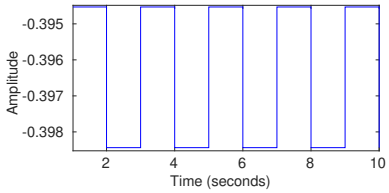
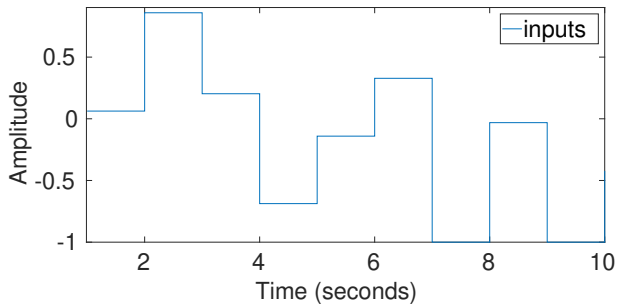


Fig. 21. Granular LCO for the synthesized attitude angle controller C_{10}, in DFI realization and with format $\langle 8, 8\rangle$.

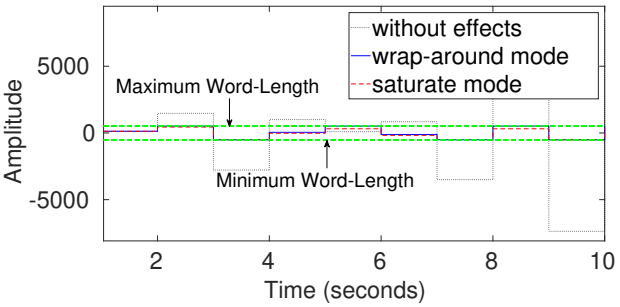
5) LCO and Overflow Effects in Closed-loop Dynamics: In order to investigate overflow effects in closed-loop dynamics, attitude dynamics should be simulated, considering FWL effects and the system output provided by DSVerifier. The plant employed to analyze impacts regarding overflow and LCO effects is described in (12), which represents roll (ϕ) and pitch (θ) angle dynamics.

$$G(z) = \frac{-0.06875z^2}{z^2 - 1.696z + 0.7089}. \tag{12}$$

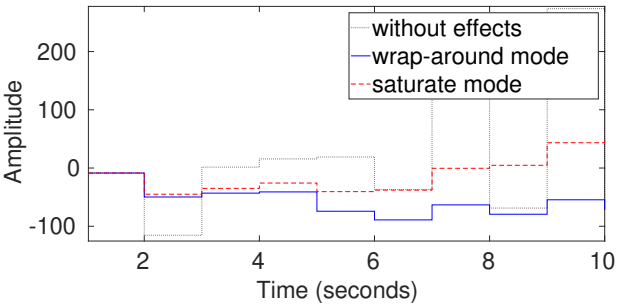
A simulation using C_5 in DFI realization and with format $\langle 10, 6\rangle$ was performed. The roll and pitch angles behavior, after overflow in C_5, is illustrated in Fig. 22. Fig. 22(a) shows effects in the plant defined by Eq. (12), Fig. 22(b) presents the inputs produced by DSVerifier, which were extracted from the counterexample related to the controller C_5, and, finally, Fig. 22(c) illustrates closed-loop overflow reproducibility in DSValidator, considering three different scenarios: (i) ideal operation, *i.e.*, without overflow and FWL effects, (ii) with overflows handled by wrapping-around, and (iii) with overflows handled by saturating. The black dotted signal represents the output, disregarding FWL effects and overflows, and the



(a) Effects in roll and pitch angles, whose dynamics is defined by Eq. (12).



(b) Torques around the x and y (u_x and u_y) axes produced by the roll/pitch controller C_5 and extracted from an overflow counterexample provided by DSVerifier.

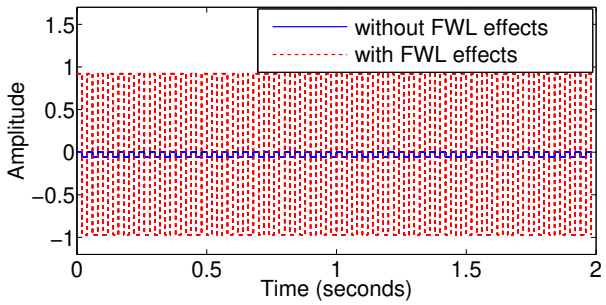


(c) Overflow closed-loop effects reproducibility in DSValidator.

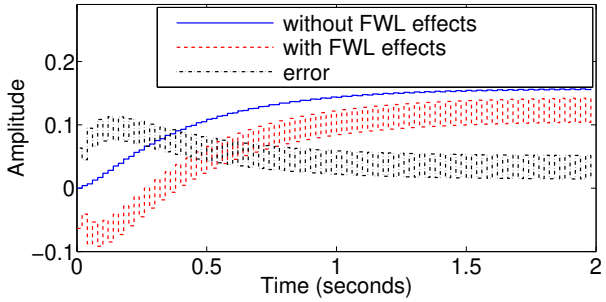
Fig. 22. Overflow effects in closed-loop dynamics.

red dashed and blue continuous lines represent the output after overflow events handled by saturation and wrap-around, respectively. One may notice that such an output behavior is affected by overflow occurrences and the related discrepancy tends to be larger for the wrap-around mode, when compared with the saturation one.

A second simulation was performed using the angle rate controller C_6, in DFI realization and with format $\langle 10, 6\rangle$, in order to investigate LCO effects regarding roll (ϕ) and pitch (θ) angles' behavior. In particular, C_6 presented granular LCO. Fig. 23 shows, in part (a), the digital controller's output with and without the FWL effects due to the fixed-point format and, in part (b), the effect of the LCO violation is observed in the pitch/roll dynamics. The output without FWL effects (blue) is compared with the one presenting them (red), whose difference (error in closed-loop output due to FWL effects) is relevant during transient time and still remains after steady state (black). One may notice that LCO on a controller's output produces oscillating torques around x and y axes



(a) LCO in the torques around the x and y (u_x and u_y) axis produced by C_6, for DFI realization and format $\langle 10, 6\rangle$.



(b) Effects in the roll and pitch closed-loop behavior described in Eq.(12).

Fig. 23. LCO effects in closed-loop dynamics of pitch/roll angles.

and, consequently, roll and pitch angles have the potential to present strong oscillation when they should be constant. Such oscillations can hinder the action of eventual position control and even lead to instability.

6) Verification Efficiency Discussion: It is important to elaborate on verification efficiency. The mean time (disregarding timeouts) spent for verifying a controller is around $22s$ ($\sigma = 76s$) for overflow verification in saturate mode, $13s$ ($\sigma = 48s$) in wrap-around mode, and $146s$ ($\sigma = 342s$) for LCO verification.

One may notice that high standard deviations regarding verification times indicate that time spent in a successful verification is much longer than what is necessary to find a violation, *i.e.*, time spent to achieve a failure result after a model checking procedure. In general, the mean figure of the latter is $288s$, for overflow verification in saturate mode, and $170s$, in wrap-around mode. Regarding LCO verification, the mean time spent to find a violation is $756s$.

The time spent in overflow verification is relatively low, with a maximum value of $670s$ and $436s$, in wrap-around and saturate modes, respectively, which happened for successful verification procedures for controller C_4, using TDFII realization. Lower verification times, if compared with previous studies [25], are justified by enhancements in the employed verifiers and solvers. Besides, successful verification results typically take longer than failed ones, since BMC techniques stop searching when a violation is found.

In general, LCO verification times take longer than overflow ones, as can be noticed in Fig. 16, due to the fact that the former is considerably more complex and considers all possible initial states, constant inputs, and oscillation periods. In addition, LCO verification presented more timeout events, which represent 10% of our benchmarks.

Finally, at this point, it is interesting to address the first research question raised in section VI-A, *i.e.*, **RQ1**. In summary, fixed-point representations change coefficient values, which then modify internal operations performed by digital controllers. As a consequence, computations result in different values, which then modify system behavior. Besides, those differences depend on other parameters, such as realization form, saturation mode, and numerical format. Among the possible final manifestations of such deviations, two were tackled here: LCO and overflow events. Particularly, in our experiments, overflow occurrences are more frequent in saturate mode, but they can also be avoided, depending on the chosen realization form and numerical format, as well as LCOs. Overflow events also occurred for wrap-around mode, which cannot be avoided by changing realization, but only by choosing suitable numerical formats. Closed-loop simulations showed that overflow and LCO events, in digital attitude controllers, produced undesired behavior in attitude dynamics, although they are usually disregarded during usual controller-synthesis techniques [81]. As a consequence, when the presence of those conditions is not evaluated, fragile systems may be designed.

D. On the Validation of DSVerifier Results

All verification results provided by DSVerifier v2.0.3 were validated with DSValidator v1.0.1 [67], in order to demonstrate its reliability and soundness. Particularly, the latter is a complement and an additional support to DSVerifier, which is employed to reproduce its counterexamples.

The main purpose of DSValidator is to automatically check whether a given counterexample, provided by DSVerifier, is reproducible or not. Indeed, it is able to reproduce counterexamples generated by DSVerifier, using MATLAB features, and, as a consequence, it is also suitable for investigating digital controller and filter behaviors, considering implementation and FWL aspects.

DSValidator takes into account implementation aspects, overflow mode (saturate or wrap-around), and rounding approach (floor and round). Currently, DSValidator performs counterexample reproducibility for stability, minimum-phase, limit cycle, and overflow occurrences.

According to Table IX, DSVerifier produced 27 LCO and 56 overflow counterexamples, the latter being divided into 32 in saturate and 24 in wrap-around mode. DSValidator confirmed the DSVerifier's results, *i.e.*, all counterexamples were reproduced by DSValidator, which suggests that DSVerifier is sound and reliable. Furthermore, DSValidator allows violation verification in graphical mode, using MATLAB, which makes re-implementation phases easier, in the proposed DSVerifier-aided verification methodology (see Fig. 6).

TABLE IX
REPRODUCIBILITY RESULTS FOR THE MENTIONED QUADROTOR ATTITUDE SYSTEM.

Property Evaluated	Tests Successful	Tests Failed	Execution Time
Overflow: Saturate	32	0	0.050703 s
Overflow: Wrap-around	24	0	0.037437 s
LCO	27	0	0.057538 s

In order to ensure the absence of LCOs detected by DSVerifier, an algorithm proposed by Bauer [82], [83] was employed. Such a scheme searches exhaustively for the absence of limit cycle and is applicable to all direct form realizations, besides being independent on quantization and digital controller order. Therefore, the Bauer's method decides about asymptotic stability of (linearly stable) digital systems, by employing an exhaustive search method. If it detects that a digital system is asymptotic stable, then the latter is free from LCO; otherwise, it is susceptible to those problems. This way, controllers that present no LCO occurrences, according to Bauer's algorithm, are the same with successful verification in DSVerifier.

Regarding **RQ2**, given that all verification results provided by DSVerifier v2.0.3 were reproduced and validated by an external tool based on exhaustive search, *i.e.*, DSValidator, we can regard its findings as sound and reliable. Furthermore, comparisons with different bound values and verification strategies demonstrate that $k = 10$, when using the BMC technique, is enough for successfully detecting most occurrences of the violations considered here, thus providing a reasonable trade-off between verification correctness and time.

E. Comparison to Testing/Fuzzing Techniques

We have also compared our verification method with another approach. Indeed, we have employed the American Fuzzing Loop (AFL) [37] tool, in order to find property violations in our benchmarks. That choice was made because AFL is one of the most efficient fuzzers currently available in literature, as recently reported by Beyer *et al.* [37]. AFL is able to find software bugs by providing invalid, unexpected, or random data as inputs to a given program and, in particular, it monitored our UAV benchmarks to find failing built-in code assertions. In addition, we have also contacted the developers of AFL, in order to get the most suitable set of parameters to evaluate our benchmarks.

In summary, experiments using AFL took around 243 hours (*i.e.,* approximately 10 days) and we used the same experimental configuration[8] adopted for the earlier tests with DSVerifier, *i.e.*, we employed benchmarks for all realization forms, regarding overflow and limit-cycle properties, and with 3600 seconds as time limit. As a result, AFL was unable to find any (single) property violation in our UAV benchmarks. Given that DSVerifier found a substantial amount of violations under the same time limit, we can conclude that BMC outperforms AFL for finding bugs in UAV control software benchmarks. As also reported by other researchers [37], it is a real challenge for fuzzers to produce inputs that can exercise all program paths in a program, when some of them depend on complex conditions. Finally, our experimental results also confirm such findings.

VII. Conclusions

This paper described an SMT-based BMC verification method, which is supported with the aid of a tool named DSVerifier, with the goal of verifying low-level properties of digital controllers. FWL effects are considered, in order to investigate LCO and overflow occurrences, in UAV digital attitude controllers. The need for reliability and autonomy, in UAV systems, has already motivated other researchers regarding the application of formal methods; however, the present work is the first one to investigate FWL effects over software implementations of UAV attitude digital controllers. In particular, overflow and LCO were investigated in 10 different digital controllers, through 84 different implementations (realizations and representations). We have also compared our approach using incremental BMC and *k*-induction with another state-of-the-art fuzzing technique.

The present experimental results show that digital controllers might present failures after implementation, depending on the chosen numerical format and realization. In particular, DSVerifier was able to identify LCO and overflow in digital controllers designed with the Ziegler-Nichols tuning and also with the CEGIS-based approach, via DSSynth. The resulting simulations showed that failures due to FWL effects caused significant changes in the behavior of UAV attitude angles. Indeed, the present method based on DSVerifier is repeated until a sound and non-fragile implementation is found. Consequently, a suitable combination of realization and numerical representation can be identified, regarding a digital attitude controller designed for a specific hardware platform. In addition, a flawed dimensioning procedure (available in the literature) was identified and the resulting violations were detected by DSVerifier, which reinforces its effectiveness and applicability. Finally, we have also employed AFL to find property violations in our UAV benchmarks; however, this fuzzing tool was unable to identify any single property violation in our UAV attitude control software benchmarks, while BMC produced effective results regarding such violations.

As future work, this study will be extended to altitude and position UAV controllers and will also support closed-loop verification, which considers system dynamics and how it is affected by FWL effects. Further studies will also investigate the consequences of controllers fragility in an UAV mission, considering every control software module. Additionally, other digital-controller representations can be included, *e.g.*, state-space based forms [60]. Regarding that, we intend to develop a method to automatically fix controller implementations, where an optimal instance can be found, considering hardware constraints. Finally, the latter can also be integrated into DSSynth, which would improve its results and consequently provide LCO- and overflow-free implementations.

[8]The command-line to run AFL as indicated by its developer is: `./afl-fuzz -f filename.c -i input/ -o output/ dsverifier filename.c --realization REALIZATION --property PROPERTY --x-size 10 --timeout 3600 --bmc CBMC`

References

[1] G. Cai, B. M. Chen, and T. H. Lee, "An overview on development of miniature unmanned rotorcraft systems," *Frontiers of Electrical and Electronic Engineering in China*, vol. 5, no. 1, pp. 1–14, Mar 2010.

[2] K. W. Williams, "Human factors implications of unmanned aircraft accidents: Flight-control problems," DTIC Document, Tech. Rep., 2006.

[3] B. T. Clough, "Unmanned aerial vehicles: autonomous control challenges, a researcher's perspective," *Journal of Aerospace Computing, Information, and Communication*, vol. 2, no. 8, pp. 327–347, 2005.

[4] I. Sadeghzadeh and Y. Zhang, "A review on fault-tolerant control for unmanned aerial vehicles (UAVs)," *Infotech@ Aerospace*, vol. 73, no. 1-4, pp. 535–555, 2011.

[5] L. Humphrey, "Model checking UAV mission plan," in *AIAA Modeling and Simulation Technologies Conference*, 2012.

[6] M. Tafazoli, "A study of on-orbit spacecraft failures," *Acta Astronaut.*, vol. 64, no. 23, pp. 195 – 205, 2009.

[7] B. J. Sauser, R. R. Reilly, and A. J. Shenhar, "Why projects fail? how contingency theory can provide new insights - a comparative analysis of NASAs mars climate orbiter loss," *Int. J. Project Manage.*, vol. 27, no. 7, pp. 665 – 679, 2009.

[8] R. Alur, C. Courcoubetis, and D. Dill, "Model-checking for real-time systems," in *LICS*, 1990, pp. 414–425.

[9] A. David, G. Behrmann, K. Larsen, and W. Yi, "A tool architecture for the next generation of UPPAAL," *LNCS*, vol. 2757, pp. 352–366, 2003.

[10] T. Henzinger, P.-H. Ho, and H. Wong-Toi, "Hytech: the next generation," in *IEEE Real-Time Systems Symposium*, 1995, pp. 56–65.

[11] R. Alur, "Formal verification of hybrid systems," in *EMSOFT*, 2011, pp. 273–278.

[12] P. de la Cámara, J. R. Castro, M. Gallardo, and P. Merino, "Verification support for arinc-653-based avionics software," *Softw. Test., Verif. Reliab.*, vol. 21, no. 4, pp. 267–298, 2011.

[13] G. J. Holzmann, "The model checker SPIN," *IEEE Trans. Software Eng.*, vol. 23, no. 5, pp. 279–295, May 1997.

[14] K. L. McMillan, *Symbolic Model Checking*. Norwell, MA, USA: Kluwer Academic Publishers, 1993.

[15] A. Cimatti, E. Clarke, F. Giunchiglia, and M. Roveri, "NUSMV: a new symbolic model checker," *STTT*, vol. 2, no. 4, pp. 410–425, 2000.

[16] A. Groce, K. Havelund, and M. Smith, "From scripts to specifications: the evolution of a flight software testing effort," in *ICSE-Volume 2*. ACM, 2010, pp. 129–138.

[17] P. J. Pingree, E. Mikk, G. J. Holzmann, M. H. Smith, and D. Dams, "Validation of mission critical software design and implementation using model checking [spacecraft]," in *Digital Avionics Systems Conference*, vol. 1, Oct 2002, pp. 6A4–1–6A4–12 vol.1.

[18] M. Whalen, D. Cofer, S. Miller, B. H. Krogh, and W. Storm, "Integration of formal analysis into a model-based software development process," in *FMICS*, ser. LNCS, vol. 4916, 2008, pp. 68–84.

[19] T. Sreemani and J. M. Atlee, "Feasibility of model checking software requirements: a case study," in *Eleventh Annual Conference on Computer Assurance*, Jun 1996, pp. 77–88.

[20] S. Li and P. Pettersson, "Verification and controller synthesis for resource-constrained real-time systems: Case study of an autonomous truck," in *15th IEEE International Conference on Emerging Technologies and Factory Automation*. IEEE, 2010, pp. 1–8.

[21] A. Aminifar, P. Tabuada, P. Eles, and Z. Peng, "Self-triggered controllers and hard real-time guarantees," in *Design, Automation & Test in Europe Conference & Exhibition*. IEEE, 2016, pp. 636–641.

[22] A. Anta, R. Majumdar, I. Saha, and P. Tabuada, "Automatic verification of control system implementations," in *Proceedings of the Tenth ACM International Conference on Embedded Software*, ser. EMSOFT '10, 2010, pp. 9–18.

[23] M. Shahbakhti, J. Li, and J. K. Hedrick, "Early model-based verification of automotive control system implementation," in *American Control Conference*. IEEE, 2012, pp. 3587–3592.

[24] N. R. Pedroza, W. MacKunis, and V. V. Golubev, "Robust nonlinear regulation of limit cycle oscillations in uavs using synthetic jet actuators," *Robotics*, vol. 3, no. 4, p. 330, 2014.

[25] I. Bessa, H. Ismail, L. Cordeiro, and J. Filho, "Verification of fixed-point digital controllers using direct and delta forms realizations," *Design Autom. for Emb. Sys.*, vol. 20, no. 2, pp. 95–126, 2016.

[26] H. Ismail, I. B. L. Cordeiro, E. B. L. Filho, and J. ao Edgar Chaves Filho, "DSVerifier: A bounded model checking tool for digital systems," in *SPIN*, ser. LNCS, vol. 9232, 2015, pp. 126–131.

[27] L. Cordeiro, B. Fischer, and J. Marques-Silva, "SMT-Based Bounded Model Checking for Embedded ANSI-C Software," *IEEE Transaction on Software Engineering*, vol. 38, no. 4, pp. 957–974, 2012.

[28] D. Kroening and M. Tautschnig, "CBMC - C Bounded Model Checker (competition contribution)," in *TACAS*, vol. 8413, 2014, pp. 389–391.

[29] A. Frutuoso, J. ao Edgar Filho, and A. Henrique, "Discrete robust controller for quadrotors stability," in *COBEM*, 2015.

[30] A. Groza, I. A. Letia, A. Goron, and S. Zaporojan, "A formal approach for identifying assurance deficits in unmanned aerial vehicle software," in *Progress in Systems Engineering*. Springer, 2015, pp. 233–239.

[31] G. Holzmann, A. Telephone, and T. Company, *Design and Validation of Computer Protocols*, ser. Prentice-Hall software series. Prentice Hall, 1991.

[32] B. Kim, "Verification and validation of UAV mission planning for human automation collaboration," in *IIE Annual Conference and Expo*, 2014.

[33] G. Sirigineedi, A. Tsourdos, B. A. White, and R. Żbikowski, "Kripke modelling and verification of temporal specifications of a multiple uav system," *Annals of Mathematics and Artificial Intelligence*, vol. 63, no. 1, pp. 31–52, Sep. 2011.

[34] A. Groce, K. Havelund, G. Holzmann, R. Joshi, and R.-G. Xu, "Establishing flight software reliability: Testing, model checking, constraint-solving, monitoring and learning," *Annals of Mathematics and Artificial Intelligence*, vol. 70, no. 4, pp. 315–349, 2014.

[35] D. T. B. Ngoc and M. Ogawa, "Overflow and roundoff error analysis via model checking." in *SEFM*, D. V. Hung and P. Krishnan, Eds. IEEE Computer Society, 2009, pp. 105–114.

[36] R. Lipka, M. Paska, and T. Potuzak, "Simulation testing and model checking: A case study comparing these approaches," in *6th International Workshop on Software Engineering for Resilient Systems*, 2014, pp. 116–130.

[37] D. Beyer and T. Lemberger, "Software verification: Testing vs. model checking - A comparative evaluation of the state of the art," in *Hardware and Software: Verification and Testing - 13th International Haifa Verification Conference*, ser. LNCS, vol. 10629, 2017, pp. 99–114.

[38] F. Lerda, J. Kapinski, H. Maka, E. Clarke, and B. Krogh, "Model checking in-the-loop: Finding counterexamples by systematic simulation," in *American Control Conference*, June 2008, pp. 2734–2740.

[39] R. Sakthivel, C. Wang, S. Santra, and B. Kaviarasan, "Non-fragile reliable sampled-data controller for nonlinear switched time-varying systems," *Nonlinear Analysis: Hybrid Systems*, vol. 27, pp. 62–76, 2018.

[40] J. Carletta, R. Veillette, F. Krach, and Z. Fang, "Determining appropriate precisions for signals in fixed-point IIR filters," in *Design Automation Conference*, 2003, pp. 656–661.

[41] T. Hilaire, P. Chevrel, and J. F. Whidborne, "Finite wordlength controller realisations using the specialised implicit form," *Intl. Journal of Control*, vol. 83, no. 2, pp. 330–346, 2010.

[42] T. Hilaire, P. Chevrel, and J. Whidborne, "Low parametric closed-loop sensitivity realizations using fixed-point and floating-point arithmetic," in *Proc. European Control Conference (ECC'07)*, 2007, pp. 4707–4714.

[43] A. Volkova, T. Hilaire, and C. Lauter, "Determining fixed-point formats for a digital filter implementation using the worst-case peak gain measure," in *2015 49th Asilomar Conference on Signals, Systems and Computers*, Nov 2015, pp. 737–741.

[44] E. Feron, "From control systems to control software," *IEEE Control Systems*, vol. 30, no. 6, pp. 50–71, Dec 2010.

[45] T. Hilaire, D. Menard, and O. Sentieys, "Bit accurate roundoff noise analysis of fixed-point linear controllers," in *Computer-Aided Control Systems, 2008. CACSD 2008. IEEE International Conference on*, Sept 2008, pp. 607–612.

[46] J. Park, M. Pajic, O. Sokolsky, and I. Lee, "Automatic verification of finite precision implementations of linear controllers," *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 10205 LNCS, pp. 153–169, 2017.

[47] J. Park, M. Pajic, I. Lee, and O. Sokolsky, "Scalable verification of linear controller software," *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 9636, pp. 662–679, 2016.

[48] I. Bessa, H. Ismail, R. Palhares, L. Cordeiro, and J. E. C. Filho, "Formal non-fragile stability verification of digital control systems with uncertainty," *IEEE Transactions on Computers*, vol. 66, no. 3, 2017.

[49] M. Soliman, "Robust non-fragile power system stabilizer," *International Journal of Electrical Power and Energy Systems*, vol. 64, pp. 626–634, 2015.

[50] G. Yang and D. Ye, *Reliable Control and Filtering of Linear Systems with Adaptive Mechanisms*, ser. Automation and Control Engineering. CRC Press, 2010.

[51] M.-T. Ho, A. Datta, and S. Bhattacharyya, "Robust and non-fragile PID controller design," *International Journal of Robust and Nonlinear Control*, vol. 11, no. 7, pp. 681–708, 2001.

[52] A. Abate, I. Bessa, D. Cattaruzza, C. Lucas, C. David, P. Kesseli, and D. Kroening, "Sound and automated synthesis of digital stabilizing controllers for continuous plants," in *20th ACM International Conference on Hybrid Systems: Computation and Control (HSCC)*, 2017, pp. 1–10.

[53] A. Abate, I. Bessa, D. Cattaruzza, L. Cordeiro, C. David, P. Kesseli, D. Kroening, and E. Polgreen, "Automated Formal Synthesis of Digital Controllers for State-Space Physical Plants," in *CAV*, ser. LNCS, vol. 10426, 2017, pp. 462–482.

[54] G. Reissig, A. Weber, and M. Rungger, "Feedback refinement relations for the synthesis of symbolic controllers," *IEEE Transactions on Automatic Control*, vol. 62, no. 4, pp. 1781–1796, April 2017.

[55] A. Solar-Lezama, "Program sketching," *STTT*, vol. 15, no. 5-6, pp. 475–495, 2013.

[56] D. Monniaux, "Compositional analysis of floating-point linear numerical filters," in *CAV*, ser. LNCS, 2005, vol. 3576, pp. 199–212.

[57] P. Munier, *Polyspace*. Wiley-ISTE, 2013.

[58] G. Hamon, "Simulink design verifier – applying automated formal methods to simulink and stateflow," in *Third Workshop on Automated Formal Methods*, 2008, pp. 1–2.

[59] W. Padgett and D. Anderson, *Fixed-Point Signal Processing*, ser. Synthesis lectures on signal processing. Morgan & Claypool, 2009.
[60] K. Åström and B. Wittenmark, *Computer-controlled systems: theory and design*, ser. Prentice Hall information and system sciences series. Prentice Hall, 1997.
[61] P. Diniz, S. Netto, and E. D. Silva, *Digital Signal Processing: System Analysis and Design*. New York, NY, USA: Cambridge University Press, 2002.
[62] D. Beyer, "Reliable and reproducible competition results with benchexec and witnesses (report on sv-comp 2016)," *LNCS*, vol. 9636, pp. 887–904, 2016.
[63] M. Y. R. Gadelha, H. I. Ismail, and L. C. Cordeiro, "Handling loops in bounded model checking of C programs via k-induction," *STTT*, vol. 19, no. 1, pp. 97–114, 2017.
[64] M. Sheeran, S. Singh, and G. Stålmarck, "Checking safety properties using induction and a sat-solver," in *Formal Methods in Computer-Aided Design, Third International Conference, FMCAD 2000, Austin, Texas, USA, November 1-3, 2000, Proceedings*, ser. Lecture Notes in Computer Science, vol. 1954. Springer, 2000, pp. 108–125.
[65] R. Brummayer and A. Biere, "Boolector: An Efficient SMT Solver for Bit-Vectors and Arrays," in *TACAS*, ser. LNCS, vol. 5505, 2009, pp. 174–177.
[66] N. Eén and N. Sörensson, "An extensible sat-solver," in *Theory and applications of satisfiability testing*. Springer, 2004, pp. 502–518.
[67] L. Chaves, I. Bessa, L. C. Cordeiro, and D. Kroening, "Dsvalidator: An automated counterexample reproducibility tool for digital systems," in *21st International Conference on Hybrid Systems: Computation and Control*. ACM, 2018, pp. 253–258.
[68] J. Dattorro, "The implementation of recursive digital filters for high-fidelity audio," *J Audio Eng Soc*, vol. 36, no. 11, pp. 851–878, 1988.
[69] L. Jackson, J. Kaiser, and H. McDonald, "An approach to the implementation of digital filters," *IEEE Trans. Audio Electroacoust*, vol. 34, no. 3, pp. 413–421, 1968.
[70] C. Rubillo, P. Marzocca, and E. M. Bollt, "Active aeroelastic control of lifting surfaces via jet reaction limiter control," *I. J. Bifurcation and Chaos*, vol. 16, no. 9, pp. 2559–2574, 2006.
[71] S. Bouabdallah, P. Murrieri, and R. Siegwart, "Design and control of an indoor micro quadrotor," in *ICRA*, vol. 5, April 2004, pp. 4393–4398 Vol.5.
[72] I. S. MathWorks, *MATLAB and SIMULINK: Student Version*, ser. MATLAB & SIMULINK: Student Version. MathWorks, 2004.
[73] D. Mellinger, N. Michael, and V. Kumar, "Trajectory generation and control for precise aggressive maneuvers with quadrotors," *Int. J. Rob. Res.*, vol. 31, no. 5, pp. 664–674, Apr. 2012.
[74] P. Garg, C. Löding, P. Madhusudan, and D. Neider, "ICE: A robust framework for learning invariants," in *26th International Conference on Computer Aided Verification*, ser. Lecture Notes in Computer Science, vol. 8559. Springer, 2014, pp. 69–87.
[75] P. Garg, D. Neider, P. Madhusudan, and D. Roth, "Learning invariants using decision trees and implication counterexamples," in *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM, 2016, pp. 499–512.
[76] D. Neider, P. Garg, P. Madhusudan, S. Saha, and D. Park, "Invariant synthesis for incomplete verification engines," in *24th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, ser. Lecture Notes in Computer Science, vol. 10805. Springer, 2018, pp. 232–250.
[77] W. Rocha, H. Rocha, H. Ismail, L. C. Cordeiro, and B. Fischer, "Depthk: A k-induction verifier based on invariant inference for C programs - (competition contribution)," in *23rd International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, ser. Lecture Notes in Computer Science, vol. 10206, 2017, pp. 360–364.
[78] B. Wachter, D. Kroening, and J. Ouaknine, "Verifying multi-threaded software with impact," in *Formal Methods in Computer-Aided Design*. IEEE, 2013, pp. 210–217.
[79] K. L. McMillan, "Interpolation and sat-based model checking," in *15th International Conference on Computer Aided Verification*, ser. Lecture Notes in Computer Science, vol. 2725, 2003, pp. 1–13.
[80] Z. Hassan, A. R. Bradley, and F. Somenzi, "Better generalization in IC3," in *Formal Methods in Computer-Aided Design*. IEEE, 2013, pp. 157–164.
[81] G. Yang, X. Guo, W. Che, and W. Guan, *Linear Systems: Non-Fragile Control and Filtering*. Taylor & Francis, 2013.
[82] P. Bauer and L.-J. Leclerc, "A computer-aided test for the absence of limit cycles in fixed-point digital filters," *IEEE Trans. Signal Processing*, vol. 39, no. 11, pp. 2400–2410, Nov 1991.
[83] K. Premaratne, E. Kulasekere, P. Bauer, and L.-J. Leclerc, "An exhaustive search algorithm for checking limit cycle behavior of digital filters," *IEEE Trans. Signal Processing*, vol. 44, no. 10, pp. 2405–2412, Oct 1996.

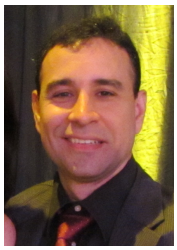
Lennon Chaves received the B.Sc. degree in computer engineering and the M.Sc. degree in electrical engineering from the Federal University of Amazonas, in 2015 and 2018, respectively. He has also studied at Nokia Teaching Institute, which is a technical high school, where he started his career in computer science. Currently, he holds a software engineer position at CESAR, an institute of innovation in Brazil. His interests are focused on the field of model checking, automated testing and software engineering.

Iury V. Bessa received the B.Sc. and the M.Sc. degrees in electrical engineering from the Federal University of Amazonas, Brazil, in 2014 and 2015, respectively. He is currently a Ph.D student at the Federal University of Minas Gerais. Since 2015, he has been assistant professor at the Department of Electricity, Federal University of Amazonas, Brazil. His areas of interest include computational intelligence, fault detection and diagnosis, fault tolerant control, formal verification and synthesis of control systems.

Hussama Ismail holds a B.Sc. degree in Computer Engineering from the Foundation Center for Analysis, Research, and Technological Innovation (FUCAPI, Brazil) and a M.Sc. degree in Electrical Engineering from the Federal University of Amazonas (UFAM, Brazil). His current research interests include formal methods, bounded model checking, compilers, multicore platforms, and embedded systems. He has been working with software development since 2011. Furthermore, Hussama has professional experience in software development using Shell Script, ANSI-C, C++, Java, PHP, and JavaScript. In addition, he has participated in many projects including embedded systems at STMicroelectronics, mobile platform applications (Android) at Paulo Feitoza Foundation (FPF Tech), web projects at FUCAPI, and hardware tests of devices at Jose Rocha Sergio Cardoso Institute (iTJRSC).

Adriano Frutuoso Adriano Bruno dos Santos Frutuoso received his B.Sc. and M.Sc. degrees in Electrical Engineering from the Federal University of Amazonas, Brazil, in 2012 and 2015, respectively. His experience is in the area of automatic control systems and in automation of industrial processes. Currently, he is an assistant professor at the Federal Institute of Education, Science and Technology of Amazonas.

Lucas C. Cordeiro received his Ph.D. degree in Computer Science in 2011 from the University of Southampton, UK. Currently, he is a Senior Lecturer in the School of Computer Science at the University of Manchester, UK. He is also a collaborator in the Postgraduate Program in Electrical Engineering and Informatics at the Federal University of Amazonas, Brazil. His work focuses on software model checking, automated testing, program synthesis, and embedded & cyber-physical systems.



Eddie B. de Lima Filho Eddie Batista de Lima Filho received and his M. Sc. and D. Sc. degrees in Electrical Engineering from the Universidade Federal do Rio de Janeiro (COPPE/UFRJ), Rio de Janeiro, RJ, Brazil, in 2004 and 2008, respectively. Since 2016, he has been with TPV, Manaus, AM, Brazil, working with digital TV, signal processing, and embedded systems. His research interests include digital signal processing, channel/source coding, video/image processing, and cognitive radio.