Universidade Federal do Amazonas

Faculdade de Tecnologia

Programa de Pós-Graduação em Engenharia Elétrica

Formal Verification Applied to Attitude Control Software of Unmanned Aerial Vehicles

Lennon Corrêa Chaves

Manaus – Amazonas

Fevereiro de 2018

Lennon Corrêa Chaves

Formal Verification Applied to Attitude Control Software of Unmanned Aerial Vehicles

Dissertação apresentada ao Programa de Pós-Graduação em Engenharia Elétrica, como requisito parcial para obtenção do Título de Mestre em Engenharia Elétrica. Área de concentração: Automação e Controle.

Orientador: Prof. Dr. Lucas Carvalho Cordeiro

Coorientador: Prof. Dr. Eddie Batista de Lima Filho

Lennon Corrêa Chaves

Formal Verification Applied to Attitude Control Software of Unmanned Aerial Vehicles

Manaus – Amazonas

Fevereiro de 2018

Ficha Catalográfica

Ficha catalográfica elaborada automaticamente de acordo com os dados fornecidos pelo(a) autor(a).

Chaves, Lennon Corrêa
C512f Formal Verification Applied to Attitude Control Software of Unmanned Aerial Vehicles / Lennon Corrêa Chaves. 2018
94 f.: il. color; 31 cm.

Orientador: Lucas Carvalho Cordeiro
Coorientador: Eddie Batista de Lima Filho
Dissertação (Mestrado em Engenharia Elétrica) - Universidade Federal do Amazonas.

1. Unmanned Aerial Vehicle. 2. Symbolic Model Checking. 3. Fixed-Point Digital Controllers. 4. Embedded Systems. I. Cordeiro, Lucas Carvalho II. Universidade Federal do Amazonas III. Título

To my parents...

Acknowledgements

I would like to thank my supervisor Dr. Cordeiro for his support, guidance, encouragement and friendship during the last two years. I believe that his support helped me to become a better professional and researcher through the M.Sc. dissertation. You have set an example of excellence as a researcher, mentor, instructor, and role model. Thanks for giving me the opportunity to be part of your research group.

I would also like to thank my co-supervisor Dr. Lima Filho for being a professor who encouraged me always to develop my work with high quality, giving me excellents feedbacks to always improve as researcher.

Many thanks to my friends and colleagues from UFAM to help me with their contributions in my research and classes, and for always encouraged me to not give up during hard times and do all my best in any situation. Thank you Anne, Alay, Myke and Felipe.

I would also like to thank my friends Lauana and Mauricio, for being the friends that I needed to have in my life. Thanks for the fun moments and for inspiring me to do my best.

I would especially like to thank my family for the love, support, and constant encouragement I have gotten over the years. In particular, I would like to thank my parents for their support in my education and because they never stopped to believe in me. I dedicate this dissertation to them.

"I solemnly swear that I am up to no good."

J.K. Rowling, Harry Potter and The Prisoner of Azkaban

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 failures in UAV attitude control software used in aerial surveillance, which are hardly found by simulation and testing tools, can be easily identified by DSVerifier.

Keywords: Unmanned Aerial Vehicle, Symbolic Model Checking, Fixed-Point Digital Controllers, Embedded Systems.

Resumo

Durante as últimas décadas, técnicas de verificação de modelos tem sido utilizadas para melhorar a confiabilidade de sistemas, no que diz respeito a veículos aéreos não-tripulados (VANTs). Contudo, existem poucos esforços focados em aplicar esses métodos ao controle de sistemas, especialmente os relativos à investigação de erros de implementação de baixo nível, os quais estão relacionados a controladores digitais e compatibilidade de hardware. O presente trabalho aborda os problemas mencionados e propõe a aplicação de uma ferramenta de verificação limitada de modelos, conhecida como Digital System Verifier (DSVerifier) ou Verificador de Sistemas Digitais, à verificação de implementação de sistemas digiais, com o objetivo de investigar problemas em controladores digitais projetados para sistemas de atitude em VANTs. Apresenta-se uma metodologia de verificação para procurar por erros de implementação relacionados a efeitos de tamanho de palavra finita (*i*.e, estouros aritméticos e ciclos-limites), em controladores de atitude de VANTs, juntamente com sua avaliação, o que visa garantir a corretude desses sistemas. Resultados experimentais mostram que falhas encontradas em software de controle de atitude de VANTs usados em vigilância aérea, as quais são dificilmente encontradas pro simulação e ferramentas de teste, podem ser facilmente identificadas pelo DSVerifier.

Palavras-Chave: Veículos Aéreos Não Tripulados, Verificação Símbolica de Modelos, Controladores de Ponto-Fixo, Sistemas Embarcados.

Contents

List of Figures

List of Tables

List of Algorithms

Acronyms

ANSI-C - ***A**merican **N**ational **S**tandards **I**nstitute **C** Programming Language*

BMC - ***B**ounded **M**odel **C**hecking*

CBMC - ***C** **B**ounded **M**odel **C**hecker*

DFI - ***D**irect **F**orm **I***

DFII - ***D**irect **F**orm **II***

DSP - ***D**igital **S**ignal **P**rocessor*

DSVerifier - ***D**igital-**S**ystems **V**erifier*

DSValidator - ***D**igital-**S**ystems **V**alidator*

DSSynth - ***D**igital-**S**ystems **S**ynhtesizer*

ESBMC - ***E**fficient **S**MT-Based Context-**B**ounded **M**odel **C**hecker*

FPGA - ***F**ield **P**rogramable **G**ate **A**rray*

FWL - ***F**inite **W**ord **L**ength*

GUI - ***G**raphical **U**ser **I**nterface*

LCO - ***L**imit **C**ycle **O**scilations*

LTI - ***L**inear **T**ime-**I**nvariant*

SAT - *Boolean **Sat**isfiability*

SMT - ***S**atisfatibility **M**odule **T**heories*

SSA - ***S**ingle **S**tatic **A**ssignment*

SV-COMP - ***I**nternational **C**ompetition on **S**oftware **V**erification*

TDFII - ***T**ransposed **D**irect **F**orm **II***

VC - ***V**erification **C**ondition*

WCET - ***W**orst-**C**ase **E**xecution **T**ime*

Chapter 1

Introduction

During the last decades, the unmanned aerial vehicle (UAV) approach has been used in various military and civil applications [2], such as armed attacks, training targets, aerial surveillance, journalism, and entertainment [2]. More recently, autonomous UAVs have gone through a notable development, due to the current evolution of embedded systems.

Applications based on autonomous UAVs are becoming very usual, both in military and civil missions, in order to avoid fatal risks regarding human lives [2]. For instance, after the cataclysmic earthquake and tsunami that struck Japan, in 2011, a nuclear leakage occurred and prevented human access to the affected areas. In order to monitor those regions, Japan employed small UAVs, whose low-altitude flight and maneuverability allowed scientists to register accurate live readings of radiation levels [3].

Since the 90's decade, formal methods have been applied to improve automation systems. In that sense, hybrid systems (HS) and cyber-physical systems (CPS), which are the two important parts of the actual industrial evolution trends, are usually represented by hybrid automata, for which several formal verification methods have been proposed. Alur *et al.* [4, 5] present the earliest application of model checking for timed automata using the Timed Computation Tree Logic (TCTL) as an extension of CTL model checking for real-time systems. Those initiatives inspired the development of formal methods and model checking tools for verifying timed automata and for representing any control system by finite state machines; notable model checking tools include UPPAAL [6] and HyTech [7], which support cyber-physical and hybrid systems and are also able to improve their reliability [8].

Formal verification has been applied to avionics embedded software, since the 2000s, due to safety and reliability requirements [9]. Different tools (*e.g.*, SPIN [10], SMV [11], and NuSMV [12]) were used for developing and validating flight control software, such as the NASA's missions Mars science laboratory [13] and deep space 1 [14], the flight control system FCS 5000 [15], and the military aircraft A-7 [16].

Most previous studies concentrate on safety regarding real-time requirements [8]; 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 specialized in implementation aspects.

The main challenges regarding verification of UAV digital controllers are related to how finite word-length (FWL) effects influence their 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 implementation. If an overflow is not avoided, an UAV controller might then perform wrong 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 [17]. Indeed, in terms of verification, detection of overflow and limit-cycle is computationally expensive, due to the presence of finite word-length (FWL) effects, which thus requires additional processing time and memory [18, 19].

1.1 Problem Description

During the development of digital controllers for UAV systems implemented in fixed-point arithmetic, designers must take into account all characteristics of the underlying hardware. In this matter, it is necessary to balance between:

- **Cost:** A device (with a high number of bits) can raise the value of a final product (if produced on large scale);

- **Accuracy:** The higher the number of bits, the lower the number of rounds, and, consequently, the accuracy of numerical values will be better.

- **Performance:** If a few bits are available for fractional parts, problems related to the discrete domain could manifest as overflows, limit cycles, loss of stability, and minimum-phase.

Based on those factors, this dissertation aims to seek answers to the following research questions:

- **RQ1:** How digital controllers that are designed for UAV attitude systems are susceptible to violations, according to fixed-point representations and realizations?
- **RQ2:** Are verification results using BMC approaches sound and can their overflow and limit cycle violations be reproduced and also validated by external tools (*e.g.*, MATLAB [20])?

1.2 Objectives

The overall objective of this dissertation is employ the Bounded Model Checking (BMC) techniques as an alternative tool during design and verification of UAV digital controllers, by assisting control engineers with an efficient approach, which brings more confidence and less effort than traditional simulation tools (*e.g.*, MATLAB [20] and LabVIEW [21]), mainly the ones that require manual intervention.

The specific objectives are listed below:

- Research and improve the existing arithmetic overflow and limit cycle checks for the BMC-based verifier;
- Investigate the impact of overflow verification, according to the saturate and wrap-around modes;
- Investigate the counterexamples by developing an automatic validation tool to reproduce all counterexamples generated by DSVerifier;
- Apply the proposed methodology to the verification of digital controllers, considering UAV Benchmarks (using the traditional and synthesis approaches) for different realization forms (*e.g.,* DFI, DFII and TDFII) and with different implementations for fixed-point arithmetic.

1.3 Contributions

The main contribution of the present study is to introduce a verification methodology based on DSVerifier and also its theoretical basis, which aims to investigate finite word-length (FWL) effects in digital controllers developed for UAV systems. In addition, this work describes for the first time (i) the verification applied to closed-loop systems described in transfer-function in order to check limit-cycle oscillations and quantization error, (ii) the automatic validation of counterexamples applied for digital controllers, in order to reproduce the verification results, and (iii) the overflow verification regarding saturation and wrap-around modes.

1.4 Organization of the Dissertation

The remainder of this work is organized as follows. Chapter 2 describes fundamental concepts about digital controllers, along with implementation aspects, and also theoretical fundamentals about bounded model checkers, fixed-point arithmetics, and how the UAV models could be designed.

In Chapter 3, the proposed verification methodology for UAVs is presented, which includes a detailed explanation about the verification of each property checked by DSVerifier for UAV systems: overflow for saturate and wrap-around mode, and limit-cycle oscillations.

Chapter 4, in turn, tackles the performed verification experiments and discusses the obtained results for overflow and limit cycle properties. In addition, this chapter presents discussions about limit-cycle and overflow effects in closed-loop dynamics and efficiency. Specifically, section 4.5 describes how those same outcomes were reproduced an validated.

Finally, chapter 5 concludes this dissertation proposes future research topics, and includes an activity planning regarding this research.

Chapter 2

Background

2.1 Digital Controllers

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) [22], but one of the most common approaches is the difference equation, which can be described as

$$y(n) = -\sum_{k=1}^{N} a_k y(n-k) + \sum_{k=0}^{M} b_k x(n-k), \tag{2.1}$$

where $y(n)$ and $x(n)$ are the n-th output and input samples, respectively, a_k and b_k are multiplier coefficients, and N is the order for output equation, and M is the order for input equation, in controller structures [22]. Through z-transform, equation (2.1) can be converted into a transfer function

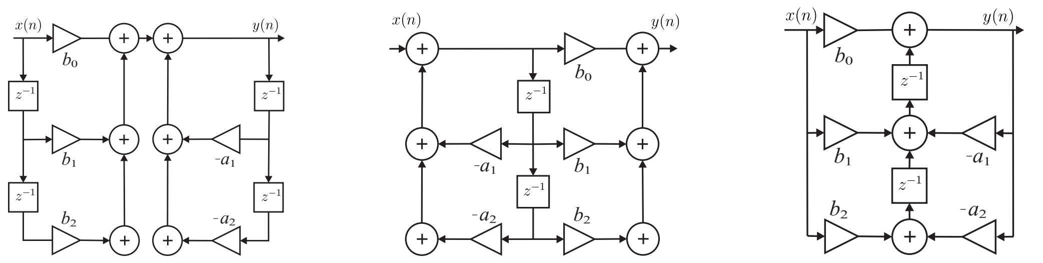
$$H(z) = \frac{b_0 + b_1 z^{-1} + ... + b_M z^{-M}}{1 + a_1 z^{-1} + ... + a_N z^{-N}}, \tag{2.2}$$

where z and z^{-1} are known as the forward-shift and backward-shift operators, respectively. In addition, the roots of the numerator are called zeros of $H(z)$ and the roots of the denominator are called poles of $H(z)$ [22, 23]. For this dissertation will explore the transfer function representation.

2.1.1 Direct-Form Realizations

Direct realizations use the coefficients in equation (2.1) in its implementation. The advantage of this implementation is that state variables are derivations of delayed inputs and outputs, using the shift operator. Nonetheless, direct-form implementations make controllers extremely sensitive to numerical errors, which is strongly evident in fixed-point implementations. As a result, that may specially harm system stability and performance.

Different direct forms (*e.g.*, DFI, DFII, and TDFII) present different numerical performances, given that those realizations implement the same controller, but with different operations numbers and orders. Figures 2.1a, 2.1b, and 2.1c show three different direct representations, where gains a_k and b_k are the coefficients and z^{-1} represents the shift operations in equations (2.1) and (2.2), respectively. Figure 2.1 shows three different direct representations: Direct Form I (DFI), Direct Form II (DFII), and Transposed Direct Form II (TDFII), in parts 2.1a, 2.1b, and 2.1c, respectively. Gains a_k and b_k represent controller coefficients, while z^{-1} describes shift operations, as shown in equations (2.1) and (2.2). Further details about digital-system representations can be found in control and digital signal processing literature [22–24].



(a) Direct form I.

(b) Direct form II.

(c) Direct transposed form II.

Figure 2.1: Direct realizations for digital controllers.

2.2 Fixed-Point Representation

2.2.1 Fixed-Point Arithmetics

Let $\langle I, F \rangle$ denote a fixed-point format and $\mathscr{F}_{\langle I,F \rangle}(x)$ denote a real number x represented in a fixed-point domain, with I bits representing the integer part and F bits representing the decimal one. The smallest number c_m that can be represented in such a domain is $c_m = 2^{-F}$ and any mathematical operations performed at $\mathscr{F}_{\langle I,F \rangle}(x)$ will introduce errors, for which an upper bound can be given [25].

Arithmetic operations with fixed-point variables are different from the ones with real numbers, once there are some non-linear phenomenons, *e.g.*, overflows and round-offs, the radix must be aligned [25]. Here, we treat fixed-point operators for sums, multiplications, subtractions, and divisions.

2.2.2 Fixed-Point Implementation in UAVs Controllers

Typically, UAVs are driven by digital controllers, which are implementations of difference equations, which generally run on microcontrollers and whose main goal is to make a plant (*e.g.*, a UAV system) follow a desired behavior, based on errors 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) conversion and zero-order hold (ZOH) devices [22], and then delivered to UAV actuators.

Direct realizations employ the coefficients shown in equation (2.1) 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 become 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 the executed operations [26].

Fig. 2.2 shows an algorithm for Direct Form I controllers, which can be implemented in C language (as shown in Fig. 2.3) 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. 2.3, 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 on each output, for a Direct Form I controller.

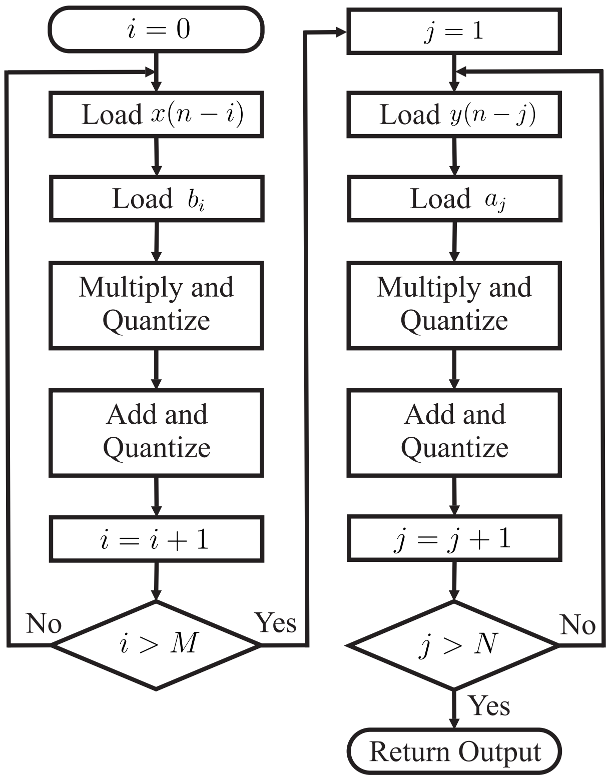


Figure 2.2: An algorithm for direct form I controllers.

Various other digital controller realizations may also be found in the literature, *e.g.*, state-space representations [22], through δ [26] and ρ [27] operators, and lattices [28].

2.3 Problems Related to Fixed-Point Implementations

Implementations of digital controllers are subject to finite word-length (FWL) effects, which are amplified in a fixed-point processor. FWL effects are related to small imprecision or functional problems, such as instability. The most common error sources are round-offs and quantization [29]. the latter occurs during analog-to-digital conversions, which consist of approximating analog signal values to quantized ones. This process generates a rounding error, whose maximum value is 2^{-F-1}, where F is the number of bits in the fractional part.

A realistic model of a FWL system must include the quantization of every numerical value, including each arithmetic result (sums and products), input signals, and system coefficients; the latter are narrowly related to the system's dynamics. Those accumulated errors might affect the position of poles and zeros of digital controllers (mainly in direct forms) and make them lose stability or minimum phase characteristics. In the control literature, this is called

```c
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);
}
```

Figure 2.3: C code fragment of a direct form I controller.

controller's fragility [23].

In the design phase, control engineers avoid poles and zero positions, which might be fatally affected by FWL effects (*i.e.*, they seek positions slightly away from the unitary circle); however, sometimes, it cannot be done easily or simply it may not be desired. An example is given by resonant controllers, in which poles must be positioned next to the stability border. The fixed-point arithmetic influence in those controllers is studied by Harnefors and Peretz *et al.* [30, 31].

A particular fixed-point representation $< I, F >$, where I is the number of bits of the integer part and F is the number of bits of the fractional one, can only represent values in the range from $2^{I-1} - 2^{-F}$ to -2^{I-1}. An overflow occurs when an addition or multiplication operation returns a result outside the range of representable values. A microprocessor generally handles overflow via wrap-around (*i.e.*, it allows numerical representation wrapping) or saturation (*i.e.*, it holds the maximum representation), and Round-off or overflow errors might lead to periodic oscillations called limit cycles. There are books that explain completely the fixed-point theory and operations, as in Granas and Dugundji [32]. Additionally, problems related to FWL effects on fixed-point format can be extensively found in literature [23, 24, 29, 33].

2.3.1 Round-off Effect and Poles and Zeros Sensitivity

The problems related to the precision of digital controllers are well-known [23, 34–37]. There are, essentially, two alternatives to handle them: the first one is a simple but expensive technique that consists of increasing the word-length, using more expensive and improved hardware, with the drawback of increasing the product price; the second one consists of optimizing and minimizing the FWL effects via deep knowledge and understanding of the phenomena related to round-off effects with the drawback of it being a quite laborious task. Some authors give an overview of the main techniques to minimize the performance degradation in digital filters, controllers, and identifiers, because of quantizations and effects of round-offs, and how to achieve optimal realization [23, 38, 39].

Round-off effects may be especially harming when coefficients are affected, because they might displace poles and zeros and consequently change the desired system's behavior [35]. Although optimal FWL techniques, improved hardware, and special representation (with larger number of bits for coefficients) have been used, poles and zeros sensitivity to FWL still presents a challenge. There are several tools and techniques to handle model uncertainty and perturbation (*e.g.*, $H\infty$ control), producing optimal and robust controllers; however, Keel and Bhattacharyya showed that these controllers may still present an undesired property [40]: fragility or sensitivity to implementation aspects (*e.g.*, the FWL representation of the digital controllers). The fragility of digital systems may be reduced by generating only small output errors, or may be very harmful by affecting a system's stability. To minimize controller fragility, some techniques, such as the nonfragile control, were proposed [39].

2.3.2 Arithmetic Overflow

The arithmetic overflow is a well-known problem in digital controllers. Overflow may occur in any node of a digital controller realization, when an operation result exceeds the limited range of a processor's word-length, resulting in undesirable nonlinearities at the output. In particular, those events may be avoided with a correct scaling of input signals. Jackson *et al.* showed that if the computation result of a digital controller, implemented with two-complement arithmetic, does not exceed the numerical range, then intermediate steps of the computation may be allowed to overflow any number of times, without causing an erroneous accumulation [33]: overflow distortions because of sums can be recovered by subsequent additions. Overflow can be avoided if one ensures that it will not occur in the final node, which can be done by scaling the multiplier inputs. The necessary and sufficient condition to avoid overflow, for any input signal, is to ensure that an output will be bounded by v_m, which represents the bound of a processor's representative range.

This condition is rarely used in practice as it generally results in a very stringent scaling, which may be harmful to the accuracy of the output signal, once the word-length is finite. There are previous studies that present efforts to optimize the number of bits used in the fixed-point format [41–49] or to present more relaxed norms [24, 50, 51] to minimize round-off effects. In any case, the accuracy is directly related to the dynamical behavior and to the stability of a system, which might present fragility because of the FWL effects and round-offs. Some researchers focus their efforts on the design phase, developing controllers with an adequate performance after FWL effects [38, 39, 52–54], and other authors investigate different implementation forms [53, 55].

2.3.3 Limit Cycle

Limit cycle oscillations (LCO) in digital controllers are defined by the presence of oscillations occurring in the output, even when the input sequence is a constant value [23]. Those oscillations may be very harmful to a control system, because of the signal-to-noise ratio of a controller's output, degrading control actions and causing damages to physical plants (especially in mechanical systems), harming surround products, and increasing material losses [56].

Some authors have studied the consequences of limit cycles. Peterchev and Sanders show that the presence of limit cycle oscillations in pulse-width modulation (PWM) power converters can increase the energy waste and decrease the lifespan of electronic devices [57]. The same problem was also studied for resonant controllers by Peretz and Ben-Yaakov [31].

LCOs may be classified as granular or overflow limit cycles. Granular limit cycles are autonomous oscillations, originating from quantization performed in the least significant bits [24]. The absence of overflow limit cycles may be assured by preventing overflows or treating them through saturation.

2.3.4 Sampling effect

The sample time is an important factor to be considered in a digital controller design. Obviously, the choice of slow sampling implies difficulties in the signal reconstruction, increasing reconstruction errors. The minimum criterion is the Shannon's theorem below:

Theorem 2.1 *Any signal $y(t)$ that presents frequency components in the interval $\left[-\frac{\omega_s}{2}, \frac{\omega_s}{2}\right]$ can be exactly reconstructed, from the sample values $y(kT)$, where T is the sampling period and $\omega_s = \frac{2\pi}{T}$,*

$$y(t) = \sum_{k=-\infty}^{\infty} y(kT) \frac{\sin\left(\frac{\omega_s}{2}(t-kT)\right)}{\frac{\omega_s}{2}(t-kT)}. \tag{2.3}$$

This theorem is not practical and the sum in equation (2.3) is difficult to be evaluated and may accumulate large errors after signal reconstruction. For this specific reason, polynomial interpolation are more commonly used in signal reconstruction (*e.g.*, zero-order hold) and a superior sample rate is necessary, which is extended to almost ten times the bandwidth,in order to avoid loss of information [37]. Furthermore, with the use of delta operators, small sample times can approximate the convergence region from the continuous one [34].

However, the use of excessively high sampling might result in undesirable performance, leading to large numerical errors due to FWL effects. In a real-time system, which may consist of other tasks beyond control actions, high sample rates will require more processing resources. Therefore, the use of high sample rates may be insufficient to execute tasks and may degrade a system's performance. Recommended sample rates are between 10 and 50 times of the bandwidth [37].

2.4 Satisfiability Modulo Theories

The Satisfiability Module Theories (SMT) decide the satisfiability of first-order formulae using a combination of different background theories and thus generalizes propositional satisfiability by supporting uninterpreted functions, linear and non-linear arithmetic, bit-vectors, tuples, arrays, and other decidable first-order theories.

In general, $\Sigma - theory$ is a collection of sentences about the signature Σ. Given a theory T and a quantifier-free formula φ, we say that φ is T-satisfiable if $T \cup \{ \varphi \}$ is satisfiable. In other words, we can say that the theory T is defined as a class of structures and φ is a satisfiable modulo if exists a structure M in T that satisfies φ (*i.e.*, $M \models \varphi$) [58].

The SMT solvers as Z3 [59] and Boolector [60] support different types of theories, and their performance may vary according to their implementation.

Theory	Example
Equality	$x1 = x2 \land \neg(x2 = x3) \Rightarrow \neg(x1 = x3)$
Linear Arithmetic	$(4y_1 + 3y_2 \geq 4) \lor (y_2 - 3y_3 \leq 3)$
Bit-vectors	$(b >> i)\&1 = 1$
Arrays	$store(a, j, 2) \Rightarrow a[j] = 2$
Combined Theories	$(j \leq k \land a[j] = 2) \Rightarrow a[i] < 3$

Table 2.1: Example of supported theories.

The Table 2.1 shows some of the theories that are supported by the SMT solvers used in this dissertation. The equality theory allows verification of equality and inequality between predicates, using operators $(=)$ $(\leq)$ $(<)$. The theory of linear arithmetic is only responsible for handling arithmetic functions (addition, subtraction, multiplication, and division) between variables and arithmetic constants. The theory about bit-vectors allows operations *bit* to *bit* regarding different architectures (*e.g.*, 32 and 64 *bits*) and is described by some operators as: and (&), or $(|)$, or-exclusive $(\oplus)$, complement $(\sim)$, shift-right $(\gg)$, and shift-left $(\ll)$. In addition, the array theory allows the manipulation of operators such as *select* and *store*.

In addition, the formulas generated in boolean satisfiability, which consist of propositional variables only, can assume $true$ or $false$ values and accept the usual logical connectives (*e.g.*, $\lor$, $\land$). The first-order formulas are formed by logical connectors, variables, quantifiers, functions, and predicate symbols [58]. In general, the satisfiability modulo theories have been exploited in several applications [61], with better results (if compared to boolean satisfiability), including the support for different decision procedures [58, 59].

2.5 Bounded Model Checking

The formal verification of software and systems is an important task to ensure that a model model meets the desired requirements. However, for robust and complex systems, this procedure becomes hard. Since model verification techniques do not require proof (methods algorithms rather than deductions), they have gained an important place in the verification paradigm, once the systems become more complex (*e.g.*, cyberphysical), which require short development cycles and high confidence level [62].

A prominent model-checking technique, commonly used for embedded software verification, since the beginning of the 2000s decade, is Bounded Model Checking (BMC). The SMT-based BMC was successfully applied to verify single- and multi-threaded programs [63]; however, the application of BMC to ensure correctness of discrete-time systems, considering the FWL effects (*i.e.*, verifying system robustness related to implementation aspects) is somewhat recent [26, 64, 65]. The basic idea behind BMC is to check the negation of a given property at a given depth. The graphic representation of how BMC works is presented in Fig. 2.4.

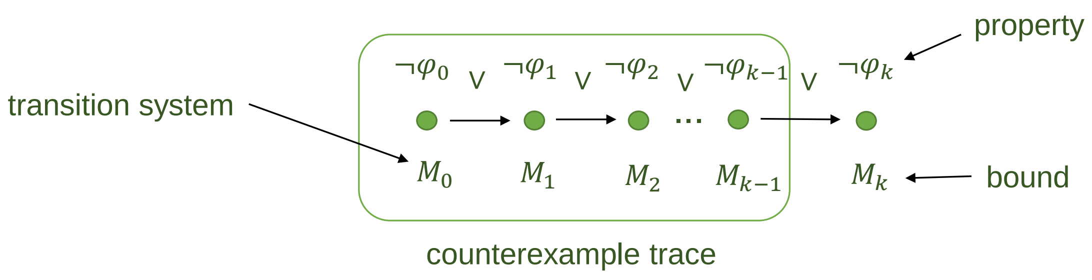

property
$\neg\varphi_0 \vee \neg\varphi_1 \vee \neg\varphi_2 \vee \neg\varphi_{k-1} \vee \neg\varphi_k$
transition system
M_0 M_1 M_2 M_{k-1} M_k
bound
counterexample trace

Figure 2.4: Representation of a transition system.

For a better understading about bounded model checking methodology exposed in Fig. 2.4, we have the following definition:

Definition 2.1 *Given a transition system M, a property ϕ, and a bound k; BMC unrolls the system k times and translates it into a verification condition (VC) ψ, which is satisfiable if and only if ϕ has a counterexample of depth less than or equal to k. SMT solvers, such as Z3 [59] and Boolector [60], can be used to check whether ψ is satisfiable.*

In BMC of digital controllers, the bound k limits the number of loop iterations and recursive calls, *i.e.*, the number of considered input samples. This way, such an approach bounds the state space that needs to be explored, during verification, and is able to find many errors in digital controllers [66]. BMC thus generates verification conditions (VCs) that reflect the exact path a statement goes through during execution, the context in which a given controller function is called, and the bit-accurate representation of arithmetic expressions. BMC tools are, however, susceptible to time and memory exhaustion, when verifying programs with loops whose bounds are too large or cannot be statically determined.

2.6 UAV Control Systems

2.6.1 Modeling UAV Attitude Dynamics

In terms of modelling, the dynamic UAV model described in this work presents some specific characteristics: (1) the associated vehicle has a rigid and symmetrical structure, (2) the mass center and the origin of the coordinate system coincide, (3) propellers are rigid, (4) thrust and drag forces are proportional to the square of the velocity, for the propellants, and (5) gyroscopic effects related to propellants and (6) influences of soil or other surface are disregarded.

In addition, the body fixed-frame *B* and the earth fixed-frame *E* are illustrated in Fig. 2.5. *B* represents the angular movements pitch (θ), roll (ϕ), and yaw (ψ), while *E* describes quadrotor translational movements in a three-dimensional space. The angular dynamics of the mentioned quadrotor's hover control are given by equation (2.4) [67].

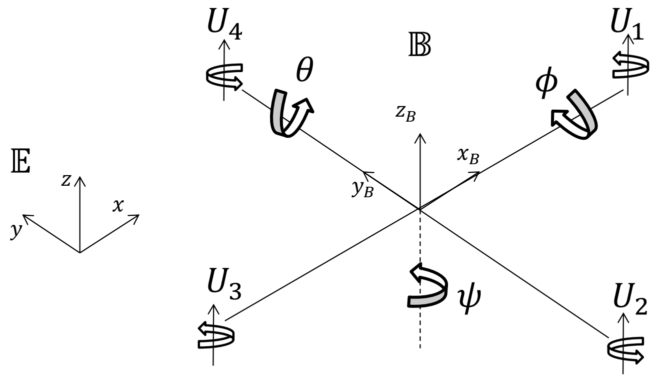


Figure 2.5: Reference system for a quadcopter model [1].

$$\begin{cases} \ddot{\phi} = \frac{I_{yy}-I_{zz}}{I_{xx}}\ \dot{\theta}\dot{\psi} + \frac{u_x}{I_{xx}} \\ \ddot{\theta} = \frac{I_{zz}-I_{xx}}{I_{yy}}\ \dot{\psi}\dot{\phi} + \frac{u_y}{I_{yy}} \\ \ddot{\psi} = \frac{I_{xx}-I_{yy}}{I_{zz}}\ \dot{\phi}\dot{\theta} + \frac{u_z}{I_{zz}} \end{cases} \tag{2.4}$$

The Fig. 2.6 shows the quadcopter used in this dissertation.

Figure 2.6: Unmanned aerial vehicle.

Nonetheless, as the physical parameters described in equation (2.4) require knowledge of I_{xx}, I_{yy}, I_{zz} (empirical parameters obtained by system identification technique for pitch, roll and yam angles, respectively), which are unknown, empirical models obtained from black-box system identification are used when designing controllers. Indeed, their dynamics approximate a real response, as described in equation (2.5), which represent roll (ϕ) and pitch (θ) angles' dynamics, while (2.6) expresses the yaw (ψ) angle's dynamics.

As a result, the physical plants determined in equations (2.5) and (2.6) are obtained based on a computational tool for systems identification available in MATLAB [20]. The respective model consists in an autoregressive with Exogenous Inputs (ARX) structure with two poles, one zero, and no delay.

$$G_1(z) = \frac{-0.06875z^2}{z^2 - 1.696z + 0.7089}. \tag{2.5}$$

$$G_2(z) = \frac{-0.04785z^2}{z^2 - 1.789z + 0.8014}. \tag{2.6}$$

2.6.2 Control Strategies in UAVs

Fig. 2.7 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 [67]. Attitude control is the basis of UAV control systems, since they are responsible for translating movement references into desired motor speeds, which are fed to motor controllers.

The present work tackles only attitude controllers. In summary, they must work properly, which means that bugs related to control software implementations should be identified and corrected; otherwise, they may cause serious problems, which normally lead to degraded performance or stability loss.

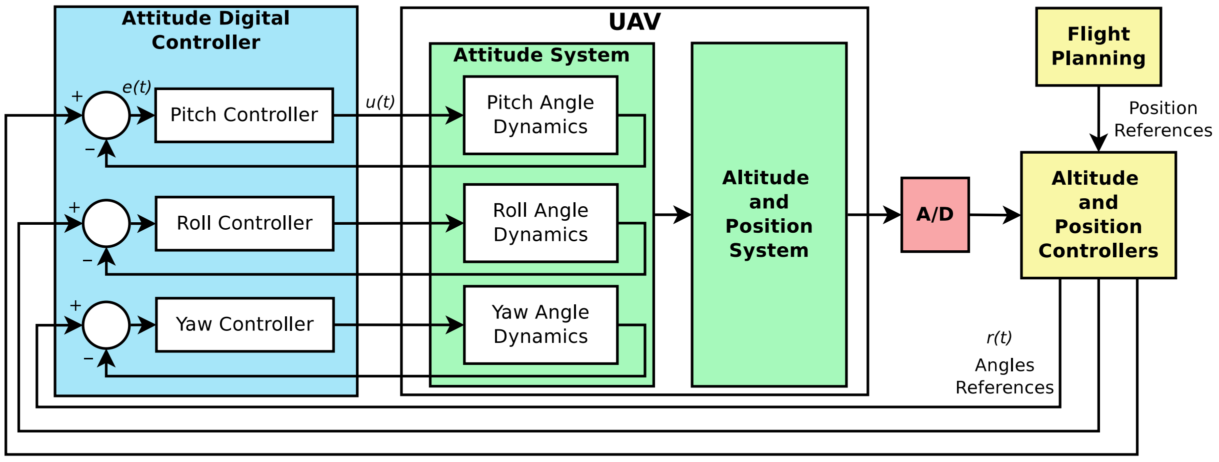


Figure 2.7: A typical digital control system for UAVs.

One of the best-known control strategies available in literature is the proportional-integral-derivative (PID) control, which is shown in Fig. 2.7. In that approach, the controller output $u(t)$ represents a response to the obtained error $e(t)$, with respect to 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 two of those: the PD and PI controllers, where the integrative and derivative actions are null, respectively. In general, a continuous PID controller has its response represented by

$$u(t) = K_P e(t) + K_D \frac{\mathrm{d}e(t)}{\mathrm{d}t} + K_I \int_0^t e(t)\mathrm{d}t, \tag{2.7}$$

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, which can be done by model-based (exact) (*e.g.*, pole assignment [22] and counter-example guided inductive synthesis [68]) or empirical (*e.g.*, Ziegler-Nichols tuning) methods. Here, PID controllers for attitude UAV control were designed using the empirical Ziegler-Nichols tuning, which is a popular and attractive approach, due to the simplicity and predictability of its results [22]. Thus, a mathematical model for UAV attitude angles was simulated, in Simulink [20], and a proportional closed-loop gain was progressively enhanced, until stability was reached. In that case, a constant oscillation appeared on its output and K_P, K_D, and K_I were then computed from that same stability threshold and the associated oscillation period, using conventional and classical formulas [22]. A PID controller, behaving as in equation (2.7), can be represented by a continuous transfer function

$$C(s) = \frac{K_D s^2 + K_P s + K_I}{s}; \tag{2.8}$$

however, it is still analog. A respective digital controller can then be obtained by converting the latter into digital form, with a chosen sample time T, which is known as discretization by approximation. Discretization by approximation consists in approximating a discrete-time transfer function (*e.g.*, $Hd(z)$) by evaluating its respective continuous-time one $Hc(s)$. Different approximations could be adopted by a designer, such as the Euler's forward difference (EF), where $s \approx \frac{z-1}{T}$ is considered, the backward difference (BF), which uses $s \approx \frac{z-1}{Tz}$, and the Tustin's (bilinear) approximation, which is based on $s \approx \frac{2}{T}\frac{z-1}{z+1}$ [22].

2.6.3 Synthesizing UAV attitude controllers with CEGIS

In order to synthesize controllers for a UAV system, given a continuous plant, a tool known as DSSynth [68] can be used, which is a program synthesizer that implements counter-example guided inductive synthesis (CEGIS) [69]. Given a plant model for roll (ϕ), pitch (θ), and yaw (ψ) angle dynamics, which is expressed by an ANSI-C syntax as input, 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 calculating a group of controller parameters to be synthesized, based on the Jury's criteria [70]. 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 a UAV, which is used as input for the CEGIS engine. The resulting intermediate C code model contains a specification υ for the property of interest (*i.e.*, robust stability) and is passed to the CEGIS module of CBMC [71], where a controller is marked as the input variable to synthesize.

CEGIS employs an iterative counterexample-guided refinement process and reports a successful synthesis result if it generates a controller that is safe, regarding υ. In particular, the C code model guarantees that a synthesized solution is complete and sound, w.r.t. the stability property υ, since it does not depend on system inputs and outputs. In case of stability, υ consists of a number of assumptions on polynomial coefficients, following the Jury's criteria.

For instance, by employing the controller-synthesis methodology described by Abate *et al.* [68], using DSSynth, the stabilizing controller in equation (2.9) was synthesized for the attitude dynamics module [72] described in equation (2.5), with a sampling time of $0.002s$.

$$H(z) = \frac{-0.39154052734375z^2 - 0.7646636962890625z}{0.8602752685546875z^2 + 0.52484130859375z} \tag{2.9}$$

.

2.7 Digital-System Verifier

In this study, DSVerifier [73] is used, which is a BMC tool for digital systems that employs CBMC [71] and ESBMC [66] 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. 2.8, 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. 2.8 represents the standard format used by DSVerifier [73] for characterizing digital systems. In particular, the second-order system described in Fig. 2.8 represents a controller for an AC motor plant [74].

```
#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
};
```

Figure 2.8: A digital-system input file for DSVerifier.

This file is directly sent to a C parser module and then follows the normal verification flow of CBMC [71] or ESBMC [66]. Fig. 2.9 shows an example of C code 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. 2.8), `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 representation 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 [75]. If they find a property violation, a counterexample is generated; otherwise, the evaluated implementation can then be embedded into a microcontroller. Finally, further details on DSVerifier are provided by Ismail *et al.* [73][1].

[1]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`

```
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);
}
```

Figure 2.9: C code fragment of a DFI controller, which was modified by DSVerifier.

2.7.1 Verifying Stability

Stability is a basic requirement when designing digital systems. In particular, digital controllers are updated every sampling period and one has to ensure that systems will be stable during its execution, that is, if all its poles are in the interior region of the unitary circle of z-plane (*i.e.*, the poles must have the module less than one) [22].

In previous studies [65, 76], stability verification using Schur Decomposition was used and implemented in the ESBMC tool, using the Eigen Library [77]. Such a method, however, involves many matrix operations that make it computationally expensive. In the present study, another method to check for stability was chosen: the Jury's Stability criteria [78], as it is computationally less expensive than the Schur's Decomposition and does not require the use of an external library. The main advantage of the Jury's algorithm can be easily noticed through its complexity, which is $O(n^2)$, whereas the complexity of the previous stability verification, based on the Schur's decomposition, is $O(n^3)$ [77].

The Jury's algorithm can be used for a given polynomial of the form

$$F(z) = a_n z^n + a_{n-1} z^{n-1} + ... a_1 z + a_0 = 0, a_n > 0, \tag{2.10}$$

where a_n until a_0 represent the digital controller denominator coefficients. In particular, such coefficients are distributed in a Jury's table using the format shown in Table 2.2.

Considering the Jury's table as a matrix m with dimensions $[2n-1][n]$, where n is the number of coefficients, one can notice:

a) The first line of matrix m has the digital controller's denominator coefficients;

b) Even-numbered lines have inverse order, when compared with of their previous lines (*i.e.*, despising final zeros);

row	z^n	z^{n-1}	...	z^{n-k}	...	z^1	z^0
1	a_n	a_{n-1}	...	a_{n-k}	...	a_1	a_0
2	a_0	a_1	...	a_k	...	a_{n-1}	a_n
3	b_0	b_1	...	b_{n-k}	...	b_{n-1}	0
4	b_{n-1}	b_{n-2}	...	b_k	...	b_0	0
5	c_0	c_1	...	...	c_{n-2}	0	0
6	c_{n-2}	c_{n-3}	...	...	c_0	0	0
...	...	...	...	...	...	...	...
$2n-1$	r_0	0	0	0	0	0	0

Table 2.2: Digital controller denominator coefficients distributed in a Jury's table.

c) b_0 is in line 3 of column 1 and its value is $b_0 = m[3-2][1] - (m[3-2][p]/m[3-2][1]) * m[3-1][1]$. Generalizing b_j and c_j leads to $m[i][j] = m[i-2][j] - (m[i-2][p]/m[i-2][1]) * m[i-1][j]$, where p is the last nonzero column number for line $i-2$;

d) Line $2n-1$ has only one nonzero element in the first column.

With the Jury's table filled in, it is necessary to check stability using the following two definitions:

Definition 2.2 *If $m[1][1]$ is positive, then F(z) will be stable iff all first elements, in odd lines (i.e., $m[3][1]$, $m[5][1]$, ..., $m[2n-1][1]$), are positive too.*

Definition 2.3 *If $m[1][1]$ is negative, then F(z) will be stable iff first elements, in odd lines (i.e., $m[3][1]$, $m[5][1]$, ..., $m[2n-1][1]$), are negative too.*

2.7.2 Verifying Minimum-Phase

The invertibility of a linear time-invariant system is intimately related to the characteristics of the phase spectral function of the system [29]. The minimum-phase property of Finite Impulse Response (FIR) systems carries over to Infinite Impulse Response (IIR) systems that have rational system functions [24]. Specifically, an IIR system with system function

$$H(z) = \frac{B(z)}{A(z)} \tag{2.11}$$

is called minimum-phase if all its zeros are stable, which means that all zeros are inside the unit circle [29, 78]. This discussion bring us to an important point that should be emphasized, where a stable pole-zero system that is minimum phase has a stable inverse which is also minimum phase. The inverse system has the system function

$$H^{-1}(z) = \frac{A(z)}{B(z)} \tag{2.12}$$

Hence the minimum-phase property of $H(z)$ ensures the stability of the inverse system $H^{-1}(z)$ and the stability of $H(z)$ implies the minimum-phase property of $H^{-1}(z)$. Conceptually, as already mentioned, a minimum-phase system has all its poles and zeros inside the unitary circle [29, 78]. Indeed, minimum-phase is a desirable property in digital controllers, given that, in a closed-loop system, zeros of its feedback-part become poles in the general system model (*i.e.*, a digital control system with a nonminimum-phase controller is potentially unstable).

In summary, minimum-phase systems are less prone to closed loop instability. The verification engine for this particular property is similar to the stability one and also uses the Jury's stability test; however, it analyzes numerator coefficients to check minimal phase.

2.7.3 Verifying Overflow

When dealing with UAV digital controllers, we need to check overflow to avoid incorrect system behavior, which is difficult to detected without computational tools, as they usually occur at run-time during quantization processes. In the present study, assertions are coded in the quantizer block, and the verification engine is configured to use nondeterministic inputs in a specified range, in order to detect overflows in a digital controller, for a given fixed-point word-length. For any addition or multiplication result, during controller operation, if there exists a value that exceeds the range representable by the fixed-point, an assert statement detects that as an arithmetic overflow. As a consequence, a literal $l_{signed\_overflow}$ is generated, with the goal of representing the validity of each addition and multiplication operation, according to the constraint

$$l_{signed\_overflow} \Leftrightarrow (FP \geq MIN) \wedge (FP \leq MAX), \tag{2.13}$$

where *FP* is the fixed-point approximation, for the result of adders and multipliers, and *MIN* and *MAX* are the minimum and maximum values that are representable for the given fixed-point bit format, respectively. Therefore, in the 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 an overflow has occurred. In addition, an arithmetic overflow can be solved by saturation or wrap-around.

Definition 2.4 ***(Saturation)*** *Saturation occurs when values outside a bit range are represented by minimum or maximum values.*

Although it is easy to finding those limits, it would be difficult to know which input led to saturation.

Definition 2.5 ***(Wrap-around)*** *Wrapping around consists in attributing minimum values when the maximum limit is reached and vice-versa [79].*

2.7.4 Verifying Limit Cycle

Limit cycle is defined by the presence of oscillations in the output, even when the input sequence is constant. LCO can be classified as granular or overflow.

Definition 2.6 ***(Granular LCO)*** *Granular limit cycles are autonomous oscillations due to round-offs in the least significant bits [24].*

Definition 2.7 ***(Overflow LCO)*** *Overflow limit cycles appear when an operation results in overflow using the wrap-around mode [24].*

In order to verify the presence of limit cycles, in a particular fixed-point controller realization, the quantizer block routine is configured by setting a flag variable on it to enable wrap-around on overflow, which means that the verification engine is not expected to detect overflow failures, as in the previous case. Additionally, the controller is configured to use a zero or constant input signal and a nondeterministic initial state, for previous outputs. The controller execution is then unrolled, for a bounded number of entries, and an assert statement is added to detect a failure, if a set of previous outputs states (that repeats during the zero-input response) is found.

One can note that this method is slightly different from the one presented by Cox [64, 79], which aims at finding a limit cycle by comparing input and output windows, within a bounded number of steps.

The constant input limit cycle occurrence is represented by a literal l_{LCO}, with the goal of determining whether a set of previous outputs states is found, according to the constraint

$$\begin{gathered} l_{LCO} \iff \exists n,k \in \mathbb{N} | x_m = 0 \implies \exists y_{k+i} = y_{k+n+i}, \\ \forall i \in \{0,1,2,...,n\}, m \in \{k,k+1,k+2,...,k+2n\}. \end{gathered} \tag{2.14}$$

Limit cycle absence is then verified by checking $\neg l_{LCO}$, that is, there exists no execution of the digital controller on which a set of previous outputs states is found.

2.7.5 Verifying Quantization Error

In the present work, the impairments present on a digital controller output, due to coefficient rounding and arithmetic-operation results, are considered. The quantization error E, due to the rounding of a number represented with F-bit precision, is

$$-2^{-F-1} \leq E \leq 2^{-F-1}. \tag{2.15}$$

One may notice that the accuracy on the computation of digital controllers is limited by the word-length specified in the digital system realization. Coefficient rounding is highly affected by the number of precision bits during design phase, *i.e.*, the higher the precision, the better is the fixed-point approximation of a floating-point number. Coefficient-rounding modifications cause variations on a controller's output, which can also be observed in time domain.

Based on that, the verification of output error bound (defined by the designer) is proposed. As a result, the output of a digital controller, implemented with reduced-word-length fixed-point representation, is compared to a reference output of the same structures, implemented using floating-point arithmetic. The verification engine then applies non-deterministic inputs to two different implementations (*i.e.*, fixed- and floating-point) for comparison purposes and check whether the difference between those implementation outputs cannot be greater than the value specified by the designer E_d. The verification condition for this property is

$$l_{error} \iff \left| y_{fxp} - y_{float} \right| \leq E_d, \tag{2.16}$$

where y_{fxp} is the output value in fixed-point implementation, y_{float} is the respective floating-point output, and E_d is the acceptable error value defined by a digital controller designer. Thus, the verification engine checks for the satisfiability of equation (3.8) and when a counterexample is found, it indicates that the output error is higher than the acceptable error for a digital controller implementation.

2.8 Related Work

The use of autonomous UAVs has led researchers to develop model checking applications for UAV software. Groza *et al.* [80] used the Hybrid Logics Model Checker, in order to ensure mission accomplishment requirements related to obstacle detection and avoidance. Kim [81, 82] proposed a methodology for building and checking specifications with SAT-based model checking, which are related to UAV mission goals, such as assurance of reachability of designed locations and surveillance of predetermined areas, which are typically represented by linear-time temporal logic expressions for safety, reachability, coverage, sequencing, and avoidance. Based on that methodology, Humphrey employed the SPIN model checker to verify autonomous UAV mission planing [83], human-automation collaboration UAV missions [84], and cooperative control of autonomous UAV units [85]. In addition, Zutshi *et al.* [86] employed the symbolic execution in order to falsify safety properties of hybrid systems that model a software system controlling a physical plant (for closed-loop systems).

Formal methods were introduced in NASA's flight-mission testing-processes by Pingree *et al.* [14], for the Deep Space 1 mission, and Groce *et al.* [13], for the Curiosity rover mars mission. In a recent work [87], Groce *et al.* tackled various verification methods and tools employed in that project, including model checking. In particular, tools based on abstraction and BMC techniques were evaluated; however, they were regarded as difficult practical implementations. Finally, the SPIN model checker was employed to perform model-driven verification, regarding NASA's flight-mission UAVs.

All aforementioned studies have a common concern about high-level specifications, generally related to flight planning and navigation. By contrast, the present work focuses on low-level aspects, by handling hardware-level implementation issues. 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 Kripke-based structures [88], 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 that change with respective outputs (closed-loop).

A promising approach to capture continuous dynamic behavior and also discrete state transitions is the hybrid automata representation. Lerda *et al.* [89] adopted that, which merges dynamic responses obtained through MATLAB and Simulink and the Java Pathfinder model checker, in order to detect errors in controller designs. Nonetheless, even employing the MATLAB and Simulink environment, their work do not consider FWL effects in controller dynamics, which may fatally affect closed-loop systems' dynamics and damage their performance and stability. Regarding that, the present work does not take into account closed-loop specifications and UAV dynamics, but instead investigates digital controllers for UAV systems, with respect to FWL problems and stringent hardware constraints.

One may also notice that software engineering techniques typically disregard platforms on which (embedded) system software operates. As a consequence, the proposed approach represents an important contribution towards the verification of low-level implementation aspects, in order to check errors caused by FWL effects. Indeed, it considers hardware implementation parameters and FWL effects related to digital controllers used in UAV systems, which was not tackled by previous approaches [13, 80–84, 87–89].

The work introduced by Abate *et al.* [68] presents a method for synthesizing stable controllers that are suitable to continuous plants given as transfer functions, which exploits bit-accurate verification of software implemented in digital microcontrollers [70, 73]. The mentioned authors developed a tool called DSSynth, which is able to synthesize digital controllers that are algorithmically and numerically sound, w.r.t. stability. DSSynth marks the first use of counterexample-guided inductive synthesis [69] to synthesize digital controllers, considering physical plants with uncertain models and FWL effects; however, low-level implementation errors (*e.g.*, limit cycles) are not further investigated.

Recently, Bessa *et al.* [26] 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) [73]. 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) [66] and the C Bounded Model Checker (CBMC) [71]. It was developed to verify digital-system implementations and is suitable for investigating finite word-length (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 dissertation is the first to investigate finite word-length (FWL) effects in UAVs. In particular, it includes overflow and limit-cycle oscillation (LCO) on the output of physical plants (in this case, UAVs), considering feedback control, where prior work [26] only tackled such effects at the output 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 [73] 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.

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 [90, 91] 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.* [90] 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 [91] also discussed a comparison about testing and model checking tools, in order to prove that model checking presents reliable results, when looking for bugs in programs. During the respective experiments, which were performed with tools for random fuzz testing and model checking, they showed that software model checkers are competitive for finding bugs, the model checking approach is also mature enough to be used in practice, given that it even outperforms the bug-finding capabilities of state-of-the-art testing tools, and BMC techniques are able to find more bugs in programs and are also faster than state-of-the-art software testing tools. Finally, previous studies mentioned by Lipka [90] and Dirk [91] had already shown that model checking could be considered for practical applications, which is the case for the work described here.

Nonetheless, other FWL effects, such as overflow and LCO, are mostly neglected by current studies. Recently, Ismail *et al.* [73] and Bessa *et al.* [26] 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 methodology presented here was integrated into the DSVerifier tool [73]. Nonetheless, there are other verification tools that provide similar features, such as Astrée [92], PolySpace [93], and Simulink Design Verifier (SDV) [94]. 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 parts for further review. In addition, PolySpace and SDV can be integrated into Matlab, Simulink, and Stateflow, but there is no study that reports their use for checking low-level properties, in digital systems. As a consequence, our proposed approach, which is supported by an automated verification tool (DSVerifier), provides distinctive qualities that make it interesting and highly suitable for application to UAVs, to the detriment of the other mentioned schemes and tools, which would still require considerable work. The Table 2.3 shows the comparison between the features available in each tool.

Table 2.3: Comparison between verification tools.

Tool	**Astrée**	**Polyspace**	**SDV**	**DSVerifier**
Supports FWL Effects?	yes	no	yes	yes
Includes Transfer-Function Format?	yes	yes	yes	yes
Processes C/C++ programs?	yes	yes	no	yes
Prepared for Digital Controllers?	no	yes	yes	yes
Investigates Low level problems (*e.g.*, overflow, LCO)?	yes	no	no	yes
Is it available (free) for any user?	no	no	no	yes

Chapter 3

Verifying Attitude Control Software in UAVs

3.1 The Proposed Verification Methodology

This section describes the proposed verification methodology implemented in DSVerifier, in order to automatically check the presence of oscillations and overflows, in attitude controllers employed in UAVs, as show in Fig. 3.1. The addressed devices are commonly used in aerial surveillance, besides being typically implemented in architectures operating with fixed-point arithmetic. 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 Section 2.2.2). Then, they were converted into digital format, with different methods [1] and sample times. A numerical fixed-point format is then selected in *step* 2, which, in this work, is performed as suggested by Carletta *et al.* [95].

In *step* 3, all implementations are checked for DFI, DFII, and TDFII, in order to provide comparison data among different realization forms, and hardware model and overflow mode (saturate or wrap-around) properties are then defined in *step* 4. Finally, overflow and limit cycle events are verified, with 10 (non-deterministic) inputs, which regarding previous studies [26,

[1](*e.g.,* Forward Euler, Tustin, Backward Euler)

96], a bound of $k = 10$ has shown to be enough to detect errors in digital controllers, as already mentioned in Section 2.5 when we discuss about proof by mathematical induction.

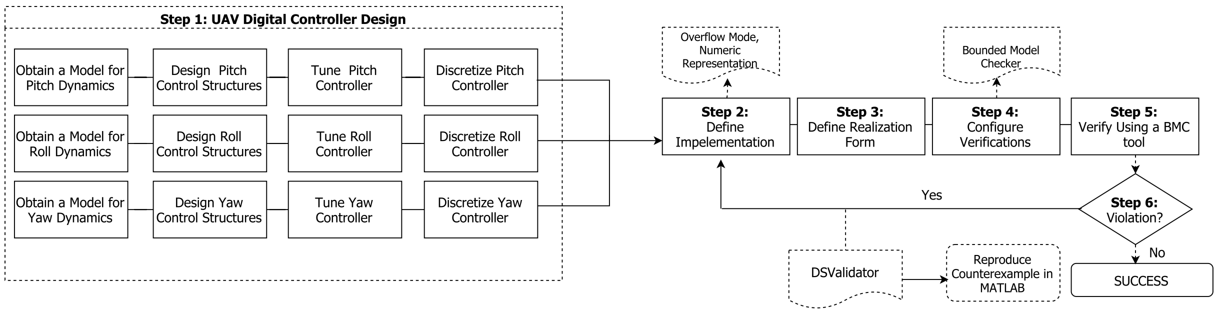

Step 1: UAV Digital Controller Design
Obtain a Model for Pitch Dynamics
Design Pitch Control Structures
Tune Pitch Controller
Discretize Pitch Controller
Obtain a Model for Roll Dynamics
Design Roll Control Structures
Tune Roll Controller
Discretize Roll Controller
Obtain a Model for Yaw Dynamics
Design Yaw Control Structures
Tune Yaw Controller
Discretize Yaw Controller
Overflow Mode, Numeric Representation
Step 2: Define Impelementation
Step 3: Define Realization Form
Bounded Model Checker
Step 4: Configure Verifications
Step 5: Verify Using a BMC tool
Yes
Step 6: Violation?
No
DSValidator
Reproduce Counterexample in MATLAB
SUCCESS

Figure 3.1: The proposed methodology for digital-system verification.

In summary, when using DSVerifier, a digital-system engineer must define UAV system parameters (*step* 1), implementation characteristics (*steps* 2 and 3), and verification settings (*step* 4). In particular, the overflow mode (*step* 4) can be defined as saturate or wrap-around, which then affects all computations and quantization operations. By default, in limit cycle verifications, the overflow mode is set to wrap-around in order to avoid overflows during the outputs computation. UAV attitude-controller verification finally occurs in *step* 5, where the underlying model-checker is employed. 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 [60], or a Boolean Satisfiability (SAT) one for CBMC, called MiniSAT [97]. Furthermore, DSVerifier provides verification results in *step* 6, which can be classified as *successful*, if there is no property violation, up to a bound $k = 10$, or *failed*, if it indicates some violation, along with a counterexample, which contains inputs and states that lead to the associated failure.

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. Other implementation options (*i.e.*, realizations and representations) can then be evaluated with that same counterexample, in order to avoid the related errors and find a suitable digital controller implementation.

Obviously, re-implementation procedures can be faster when performed by experienced engineers and, regarding this 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 [26]. Additionally, some trade-offs have to be taken into account, when performing modifications related to the 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 one;
- Decreasing the number of bits of fractional parts leads to an increase in quantization noise and, consequently, signal-to-noise ratio (SNR) problems [24];
- 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 [24, 26];
- 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 (overflow) result affects the system [33, 98];
- DFII, in turn, needs verification after each equivalent adder (input and output) [33, 98];
- If saturation arithmetic is employed, when an overflow occurs, not only the final result but also intermediate operations have the potential to affect the system output, even if DFI and TDFII are employed [33, 98]. 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 the resulting SNR [24];
- Scaling can also prevent overflow LCOs.

When performing verification with DSVerifier [73], 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. For instance, consider the following digital controller (equation (3.1)) representing an UAV controller and its ANSI-C program represented in Fig. 3.2 as follows:

$$H(z) = \frac{0.0096z - 0.009}{0.002z} \tag{3.1}$$

Performing verification with DSVerifier [73], in order to check the overflow violation by sat-

```
#include <dsverifier.h>

digital_system ds = {
    .b = { 0.0096, −0.009 },
    .b_size = 2,
    .a = { 0.02, 0.0 },
    .a_size = 2,
    .sample_time = 0.02
};

implementation impl = {
  .int_bits = 3,
  .frac_bits =   13,
  .max =   1.0,
  .min =   −1.0
  };
```

Figure 3.2: ANSI-C program describing the controller defined in equation 3.1.

uration, with direct-form II (DFII) realization, and with 3-bits as integer-part and 13-bits as fractional-part, an overflow in the digital controller described in equation (3.1) is detected by DSVerifier. The Fig. 3.3 shows graphically an overflow by saturation in DFII realization form. The maximum word representation for this fixed-point implementation is 3.999, and the minimum representation allowed is −4.000. However, when a digital-system engineer modifies the implementation to use the direct-form I (DFI) realization form (See Fig. 3.4), the system is free from overflow violation, which means that the choose of a diferent realization form impacts

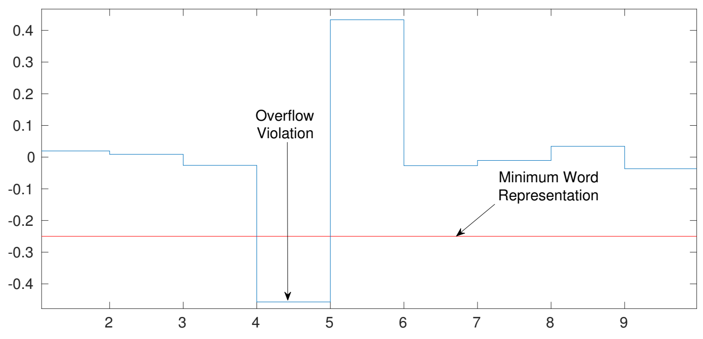


Figure 3.3: Overflow for the digital controller described in equation (3.1).

directly the presence of arithmetic overflows in digital controllers. The overflow verification

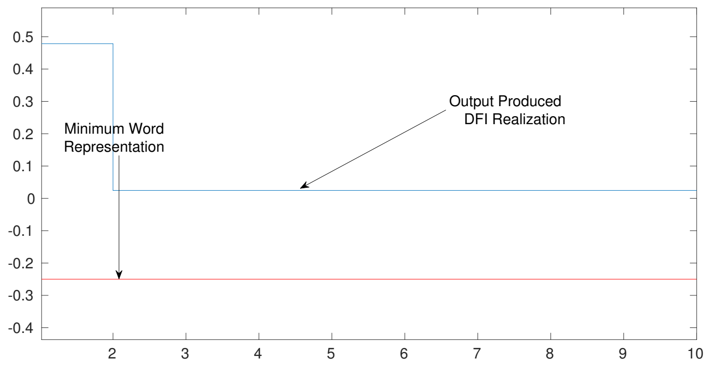


Figure 3.4: Digital controller defined in equation (3.1) implemented with DFI realization.

scheme employed here is the same used in previous studies [26]; however, the current LCO verification presents some enhancements, which are explained in the next section. LCO verification is indeed a novelty and especially important for UAV attitude-control.

3.1.1 The DSVerifier Counterexample Format

For our current work, DSVerifier exploits counterexamples provided by verifiers [66, 99]; if there is a property violation, then the verifier provides a counterexample, which contains inputs and initial states that lead the digital system to a failure state. Fig. 3.5 shows an example of the present counterexample format related to an overflow limit-cycle violation for the digital system represented by equation (3.2):

$$H(z) = \frac{2002 - 4000z^{-1} + 1998z^{-2}}{1 - z^{-2}}. \tag{3.2}$$

```
Property = LIMIT_CYCLE
Numerator  = { 2002,  −4000,  1998 }
Denominator  = { 1,  0,  −1 }
X_Size = 10
Sample_Time = 0.001
Implementation = <13,3>
Numerator (fixed−point) = { 2002, −4000, 1998 }
Denominator (fixed−point) = { 1, 0, −1 }
Realization = DFI
Dynamical_Range = { −1, 1 }
Initial_States = { −0.875, 0, −1 }
Inputs = { 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5}
Outputs = { 0, −1, 0, −1, 0, −1, 0, −1, 0, −1}
```

Figure 3.5: Proposed counterexample format example.

The proposed counterexample format shown in Fig. 3.5 describes the violated property (represented by a *string*), transfer function numerator and denominator (represented by *fixed-point numbers*), bound (represented by an *integer*), sample time (represented by a *fixed-point number*), implementation aspects (integer and fractional bits represented by an *integer*), realization form (represented by a *string*), dynamical range (represented by an *integer*), initial states, inputs, and outputs (which are represented by *fixed-point numbers*). In particular, the counterexample provides the needed data to reproduce a given property violation via simulation in MATLAB.

3.2 Verifying Closed-Loop Systems

The proposed methodology for verifying closed-loop digital control systems is described as follows. The plant model must be represented by a parametric uncertain model, *i.e.*, plant intervals. Suppose that the digital controller $C(z)$ and the plant model $P(z)$ are given as

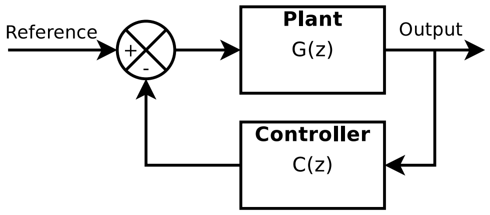
$$C(z) = \frac{\beta_0 + \beta_1 z^{-1} + ... + \beta_{M_C} z^{-M_C}}{\alpha_0 + \alpha_1 z^{-1} + ... + \alpha_{N_C} z^{-N_C}} \qquad P(z) = \frac{b_0 + b_1 z^{-1} + ... + b_{M_G} z^{-M_G}}{a_0 + a_1 z^{-1} + ... + a_{N_G} z^{-N_G}} \tag{3.3}$$

Consider the transfer functions $C(z)$ and $P(z)$ be arranged in vectors c_0 and p_0, respectively. For each $C(z)$ implementation, there is a function $FWL[\cdot] : \mathbb{R}^{N_C+M_C+2} \rightarrow Q[\mathbb{R}^{N_C+M_C+2}]$, which applies the FWL effects to a digital-system, where $Q[\mathbb{R}]$ represents the quantized set of representable real numbers in the chosen implementation format. The quantized controller parameters vector (c) is generated via $FWL[c_0]$. The uncertainty plant parameters (p) can be expressed as

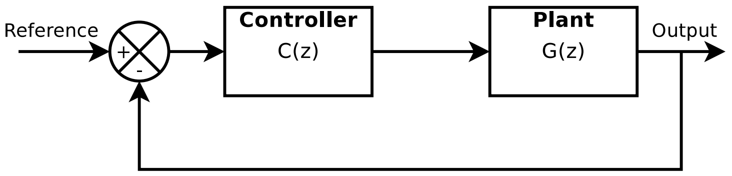
$$p = p_0 + \Delta p = \begin{bmatrix} b_0 + \Delta b_0 \; b_1 + \Delta b_1 \; ... \; b_{M_G} + \Delta b_{M_G} \\ a_0 + \Delta a_0 \; a_1 + \Delta a_1 \; ... \; a_{N_G} + \Delta a_{N_G} \end{bmatrix}, \tag{3.4}$$

where Δp represents the uncertainties for each plant coefficient. The polynomial set of possible values of p is denoted by $\mathfrak{P}$.

In addition, DSVerifier supports two closed-loop configurations: feedback (See Fig. 3.6a), *i.e.*, a digital controller is connected through a feedback path, and series (See Fig. 3.6b), where a controller is located at a forward path. In the DSVerifier command-line version, loop configuration is chosen with `-connection-mode <connection_name>`, where `<connection_name>` can be represented by `SERIES` or `FEEDBACK`.



(a) Feedback configuration.



(b) Series configuration.

Figure 3.6: Closed-loop configurations supported by DSVerifier.

3.2.1 Stability verification

A discrete-time linear time invariant system is considered asymptotic stable if its poles lie inside the unit circle, *i.e.*, a circle placed at the origin of a complex plane with unitary radius [78]. Consequently, if a discrete-time linear system is asymptotic stable, then it is considered bounded-input and bounded-output (BIBO) stable, *i.e.*, given an arbitrary bounded input, the output is also bounded. Furthermore, a discrete-time system is considered internally stable if all its internal states are bounded for all initial conditions and all bounded signals injected in it, *i.e.*, if all its components are stable [78].

Lemma 3.1 *A feedback digital control system represented by* $C(z) = \frac{N_C(z)}{D_C(z)}$ *and* $P(z) = \frac{N_P(z)}{D_P(z)}$ *transfer functions, which represent the controller and plant, respectively, as the one shown in Fig. 3.6b, is internally stable if and only if:*

– *the roots of the characteristic polynomial* $S(z)$ *are inside the open unit circle, where*

$$S(z) = N_C(z)N_P(z) + D_C(z)D_P(z);$$

– *the direct loop product* $\frac{N_C(z)}{D_C(z)} \cdot \frac{N_P(z)}{D_P(z)}$ *has no pole-zero cancellation on or outside the unit circle.*

Based on Lemma 3.1, DSVerifier checks stability for closed-loop systems according to Algorithm 3.1. Firstly, DSVerifier applies FWL effects on a controller's numerator and denominator, then it builds a non-deterministic model to represent the plant family $\mathfrak{P}$ and, finally, applies the Jury's criteria [78] to determine stability regarding $S(z)$.

Algorithm 3.1: Closed-loop stability verification

```
Data: $N_C(z)$, $N_P(z)$, $D_C(z)$, $D_P(z)$, implementation settings, and plant's parametric deviations $\Delta p\%$.
Result: SUCCESS for stable systems or FAILED for unstable systems, along with a counterexample.
begin
    Formulate an FWL effect function $\mathscr{FWL}[\cdot]$
    Construct the plant interval set $\mathfrak{P}$, where $\hat{N}_P(z) \in \mathfrak{P}$ and $\hat{D}_P(z) \in \mathfrak{P}$
    Obtain $FWL[N_C(z)]$ and $FWL[D_C(z)]$
    Check $\neg\phi_{stability}$ for $S(z) = \mathscr{FWL}[N_C(z)] \cdot \hat{N}_P(z) + \mathscr{FWL}[D_C(z)] \cdot \hat{D}_P(z)$
    if $\neg\phi_{stability}$ is satisfiable then
        return FAILED and a counterexample (i.e., unstable)
    end
    else
        return SUCCESS (i.e., stable)
    end
end
```

Precisely, the stability verification is encoded as a verification condition (VC) $\psi_k = \bigwedge_{i=0}^{k} \neg\phi_{stability}(s_i)$ that is satisfiable if, in a given state s_i, some system's poles (*i.e.*, eigenvalues) has magnitude less than 1.

3.2.2 Limit-cycle oscillation verification

The presence of oscillation in a system's output, even though a input sequence is constant, is considered as LCO. In DSVerifier, LCO verification is performed in a system's general equation $H(z)$, which is computed from plant and controller transfer functions in series configuration, as

$$H(z) = \frac{P(z)}{(1+C(z)\cdot P(z))} = \frac{\frac{N_P(z)}{D_P(z)}}{1+\frac{N_C(z)}{D_C(z)}\cdot\frac{N_P(z)}{D_P(z)}} = \frac{N_H(z)}{D_H(z)}, \tag{3.5}$$

and in feedback configuration, as

$$H(z) = \frac{C(z)\cdot P(z)}{(1+C(z)\cdot P(z))} = \frac{\frac{N_C(z)}{D_C(z)}\cdot\frac{N_P(z)}{D_P(z)}}{1+\frac{N_C(z)}{D_C(z)}\cdot\frac{N_P(z)}{D_P(z)}} = \frac{N_H(z)}{D_H(z)}. \tag{3.6}$$

Basically, DSVerifier checks the presence of persistent oscillation in an output, given a constant input signal, which is illustrated in Algorithm 3.2.

Algorithm 3.2: Closed-loop limit cycle verification

```
Data: $H(z)$ and its outputs up to $k$-depth.
Result: SUCCESS for the absence of LCOs, otherwise FAILED along with a counterexample.
begin
    Formulate a FWL effect function $FWL[\cdot]$
    Construct the plant interval set $\mathfrak{P}$, where $\hat{N}_P(z) \in \mathfrak{P}$ and $\hat{D}_P(z) \in \mathfrak{P}$
    Obtain $FWL[N_C(z)]$ and $FWL[D_C(z)]$
    Compute $H(z)$ according to series or feedback configuration (Cf. equations (3.5) and (3.6)
    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)
    end
end
```

Firstly, the quantizer block routine is configured to enable wrap-around and DSVerifier selects the last output as a reference and searches the same value among previous elements, in order to compute the length of a time window for (potential) LCO. If a time window is detected and the elements inside that are repeated, then DSVerifier confirms the presence of LCO. Precisely, LCO verification is encoded as a VC that is satisfiable *iff* there is any window (with non-deterministic size) of output samples, which is repeated from any sample until a bound k (the same used by the BMC algorithm). It is worth noticing that LCO verification can also be performed for non-deterministic inputs and states, which was impossible with the previous versions of DSVerifier.

3.2.3 Quantization error verification

Floating-point representations provide better approximation of rational numbers, when compared with fixed-point ones; however, many practical implementations of digital controllers are designed with the latter [26]. Additionally, using floating-point arithmetic in BMC frameworks leads to higher verification time and memory consumption [100]. As a consequence, DSVerifier supports only fixed-point representation, using bit-vector and rational arithmetic.

In such a context, precision in a digital controller's operation is limited by its word length, which is specified in a digital system's realization. Furthermore, FWL computations may lead to rounding and truncation errors, which change pole and zero positions and modify the associated frequency response. Consequently, such changes cause variations that can also be observed in time domain. Precisely, the quantization error E_d, due to round-off errors in numerical operations with a precision of F bits, is given by

$$-2^{-F-1} \leq E_d \leq 2^{-F-1}. \tag{3.7}$$

Hence, DSVerifier applies non-deterministic inputs to two different implementations (*i.e.*, with and without FWL effects) and compares results from both of them, in order to check whether differences regarding their outputs are inside a tolerable bound. Therefore, the VC for this property is given as

$$l_{error} \iff \left| y_{fxp} - y_{float} \right| < e_b, \tag{3.8}$$

where y_{fxp} is the output value from the fixed-point implementation (*i.e.*, with FWL effects), y_{float} is the output value from the reference floating-point implementation (*i.e.*, without FWL effects), and e_b is the acceptable error value defined by a designer. In summary, DSVerifier compares the output signal of two closed-loop systems, *i.e.*, with and without FWL effects, and then checks whether E_d is inside a tolerable bound, as described in Algorithm 3.3.

Algorithm 3.3: Closed-loop quantization error verification

```
Data: The controller $C(z)$, the plant $P(z)$, and $e_b$ as an acceptable error value.
Result: SUCCESS if the quantization error is lower than $e_b$, otherwise FAILED along with a
        counterexample.
begin
    Formulate a FWL effect function $FWL[\cdot]$
    Construct the plant interval set $\mathfrak{P}$, where $\hat{N}_P(z) \in \mathfrak{P}$ and $\hat{D}_P(z) \in \mathfrak{P}$
    Obtain $FWL[N_C(z)]$ and $FWL[D_C(z)]$
    Compute $H_{fxp}(z)$ according to series or feedback configuration (Cf. equations (3.5) and (3.6) in
      fixed-point arithmetic
    Compute $H_{float}(z)$ according to series or feedback configuration (Cf. equations (3.5) and (3.6) in in
      floating-point arithmetic
    Calculate outputs from $H_{fxp}(z)$ (i.e., $y_{fxp}(k)$)
    Calculate outputs from $H_{float}(z)$ (i.e., $y_{float}(k)$)
    Compute the difference between the fixed- and floating-point outputs, i.e., $E_d = y_{fxp}(k) - y_{float}(k)$
    if $E_d < e_b$ then
        return SUCCESS (i.e., quantization error is within a tolerable bound)
        else
            return FAILED and a counterexample (i.e., high quantization error)
        end
    end
end
```

3.3 Verifying Limit Cycle in UAV digital controllers

Limit cycle oscillations (LCO) 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[101].

Some authors have already studied the consequences of limit cycles. For instance, Peterchev and Sanders showed that the presence of limit cycle oscillations, in pulse-width modulation power converters, can increase energy waste and decrease lifespan of electronic devices [57], which was also studied for resonant controllers, by Peretz and Ben-Yaakov [31]. Additionally, according to Pedroza *et al.* [17], the LCO effect would surpass the limiting safe flight boundaries of an aircraft [101] and, as a consequence, could potentially lead to structural damage and catastrophes. The LCO verification scheme employed here extends previous DSVerifier versions [26, 73], where only zero-input LCO events were detected, by comparing past and current states. Indeed, DSVerifier searches for the repetition of an 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 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 the system 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 dynamic range that is provided. 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, in order to enable 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

$$\begin{gathered} l_{LCO} \iff \exists n,k \in \mathbb{N}, \exists c\mathbb{R} | x_m = c \implies \exists y_{k+i} = y_{k+n+i}, \\ \forall i \in \{0,1,2,...,n\}, m \in \{k,k+1,k+2,...,k+2n\}, \end{gathered} \tag{3.9}$$

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

3.3.1 Illustrative Example

In order to explain the DSVerifier-aided verification methodology, the following second-order controller from an UAV controller [1] is used:

$$H(z) = \frac{1.5610 - 1.485z^{-1}}{1 - 0.9z^{-1}}. \tag{3.10}$$

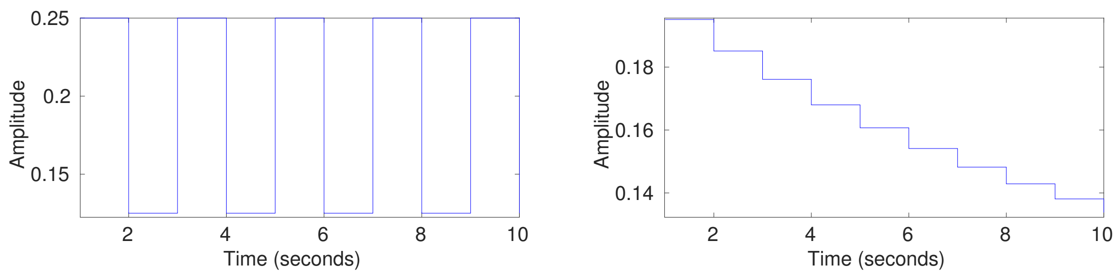
Indeed, a transfer function definition corresponds to the first step of the proposed methodology, which is shown in Fig. 3.1. The second step is the choice of the FWL representation, whose fixed-point parameters are computed according to the method described by Carletta *et al.* [95] and considering an 8-bits hardware architecture. It allows the calculation of a (sufficient) number of bits to avoid overflow, using

$$j = \lceil \log_2(\|h\|_1 \cdot \|x\|_\infty) \rceil + 1, \tag{3.11}$$

where $||h||_1$ is the l_1-norm of the system 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 equation (3.11) is enough to prevent overflow in the system 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 [33].

Using equation (3.11) for the system in equation (3.10), 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 the system 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. 2.8, and define the desired DFII realization, in the third step. In Step 4, a timeout of 1 hour and a bound of 10 cycles are set, given that limit cycle occurrences need to be verified[2]. After a few seconds, the verification process is concluded and a failure



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

Figure 3.7: Responses to a constant input equal to 0.125, w.r.t. equation (3.10) with format $\langle 4,4 \rangle$.

(Step 6) is indicated. A persistent oscillation in the system 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. 3.7a, with amplitude between 0.25 and 0.125. As a consequence, a designer should go back to Step 2, in order to avoid the limit cycle reported by DSVerifier, through a simple realization change. For instance, DFI was able to fix the controller's implementation, which would lead to a successful verification (Step 6), as can be seen in Fig. 3.7b.

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

3.4 Verifying Overflow in 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

$$l_{signed\_overflow} \Leftrightarrow (FP \geq MIN) \wedge (FP \leq MAX), \tag{3.12}$$

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 [24, 29].

Algorithm 3.4 describes how DSVerifier [73] 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 to avoid overflow. 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.

Nonetheless, this approach does not consider wrap-around and saturation effects during the output computation. The proposed overflow detection algorithm is implemented in DSVerifier, where each computation is iteratively checked, by applying saturation or wrap-around, according to the maximum bounded number of entries k. The overflow mode is defined by users, while the constant input signal $x(n)$ and initial states are determined using non-deterministic

Algorithm 3.4: 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 $FWL[\cdot]$;
    Obtain $FWL[N_C(z)]$ and $FWL[D_C(z)]$;
    Compute $H(z) = \frac{FWL[N_C(z)]}{FWL[D_C(z)]}$;
    Obtain the outputs $(y(n))$ from $H(z)$;
    Obtain the $MIN$ and $MAX$ representation given $I$-integer bits and $F$-fractional bits;
    $MIN \leftarrow -2^I$;
    $MAX \leftarrow 2^I - 2^{-F}$;
    for $i \leftarrow 0$ 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
```

values, according to the dynamic range that is provided.

Indeed, overflow occurrences in wrap-around mode are detected according to the algorithm described in Algorithm 3.5. Finally, overflow occurrences in saturate mode are detected according to the algorithm described in Algorithm 3.6.

Algorithm 3.5: Wrap-around overflow algorithm

```
Data: value to be evaluated in wrap-around algorithm, maximum as the maximum word
      representation, and minimum as the minimum word representation.
Result: value after the wrap-around algorithm.
begin
    Compute the range of the word-length to be evaluated
    range = maximum + minimum - 1;
    Check if there is an arithmetic overflow in value
    if value < minimum then
        $value \mathrel{+}= range * ((minimum - value) / range + 1)$;
    end
    return minimum + (value - minimum) % range;
end
```

In those algorithms, *value* represents each output that is computed according to the system realization, *maximum* represents the maximum word representation, and *minimum* represents the minimum word representation.

Algorithm 3.6: Saturate overflow algorithm

```
Data: value to be evaluated in the saturate algorithm, maximum as the maximum word
      representation, and minimum as the minimum word representation.
Result: Saturation of the value.
begin
    Check if there is an arithmetic overflow in value
    if value < minimum then
        return minimum;
    end
    else
        return maximum;
    end
end
```

3.4.1 Illustrative Example

In order to explain the proposed DSVerifier-aided verification methodology, the following second-order controller [1] is used:

$$H(z) = \frac{60z - 50}{z} \tag{3.13}$$

In particular, for DFI and TDFII realization forms with wrap-around mode and according to the Jackson's rule [33], 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. 2.1b) 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.

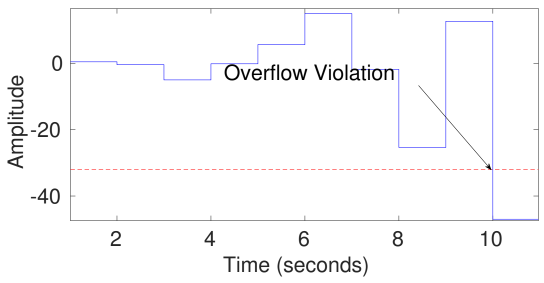


Figure 3.8: Overflow in a second-order digital controller.

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 equation (3.13), 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. Regarding Fig. 2.1a, the mentioned overflow violation occurred during a multiplication by b_0. Finally, Table 3.1 shows values computed for each output from equation (3.13).

Table 3.1: 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

Chapter 4

Experimental Evaluation

This section is split into five parts. Section 4.1 presents all benchmarks adopted for the evaluation of the UAV attitude-angle control software, while in Section 4.2 the objectives of the experiments are described. Then, the employed setup is described in Section 4.3, while Section 4.4 shows the performed experiments, through result evaluation and performance comparison. Additionally, in Section 4.5, the counterexamples generated by the mentioned experiments are reproduced and automatically validated, using the DSValidator tool [102]. It is worth noticing that an exhaustive-search algorithm, specifically designed to check the absence of limit cycles [103, 104], was implemented and applied during the proposed experiments, in order to confirm the results provided by DSVerifier.

4.1 UAV Control System Description and Benchmarks

UAV modeling is a hard task, given that such kind of system presents many nonlinearities and complex structures. Generally, the respective 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.* [1]. 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. 2.5 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. 4.1 shows the controlling structure of the PD/PD, PD/PI, PD/P and PID/PD strategies, while Fig. 4.2 shows the specific approach adopted for the PID one.

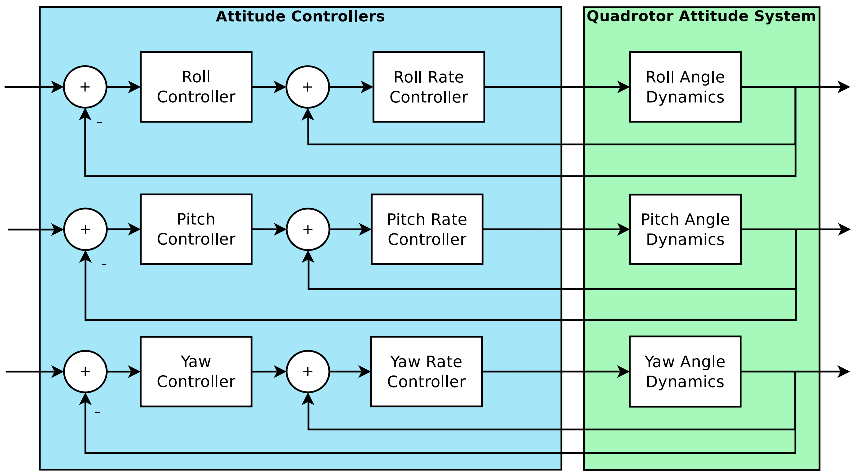


Figure 4.1: Attitude control system with combined structure.

4.1.1 Digital Controllers for UAV Attitude Angles

In the control strategy shown in Fig. 4.1, 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. 4.2 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

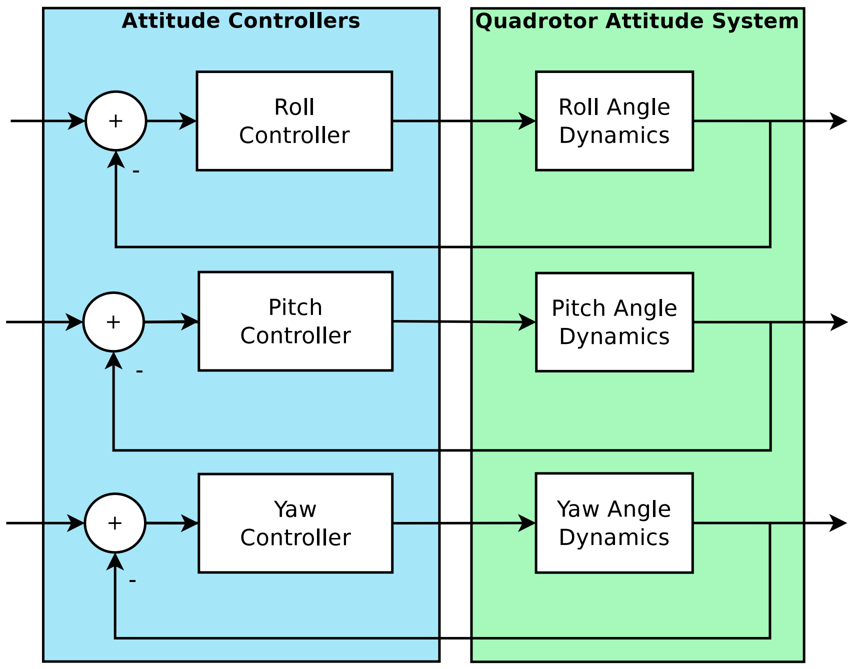


Figure 4.2: Attitude control system with only one PID controller, for each angle.

the roll, pitch, and yaw PID controllers, respectively. In some 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.* [72]. 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 dissertation present order less or equal to 2, which is common in the digital-controller area. Although higher order controllers are not used in this dissertation, our benchmarks are representative of UAV attitude control systems, since they are indeed extracted from real UAV systems.

Table 4.1 shows the association of each controller, regarding its function (Figs. 4.1 and 4.2), while Table 4.2 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.* [95], which suggests 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 more than that. 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 4.1: Controller distribution for the adopted control strategies.

Control Func. and 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 4.2: Digital controllers for the evaluated quadrotor attitude system.

Controller ID	**Tuning Gains**			**Discretization Method**	**Sample Time (ms)**	**Discrete Transfer Function**
	K_P	K_D	K_I			
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. (equation 2.9)

4.2 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 [33, 98] 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 [33, 98] 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. 2.1 (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, the result of the output adder may be incorrect, as previously mentioned.

For all tested implementations, 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)?

4.3 Experimental Setup

In the present work, DSVerifier v2.0.3 was used to check controllers described in Sec. 4.1.1. The ones in Table 4.2 were verified with 3 different numerical formats (with at least the number of integer bits suggested by Carletta *et al.* [95]) 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. CBMC5.5 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).

4.4 Experimental Results

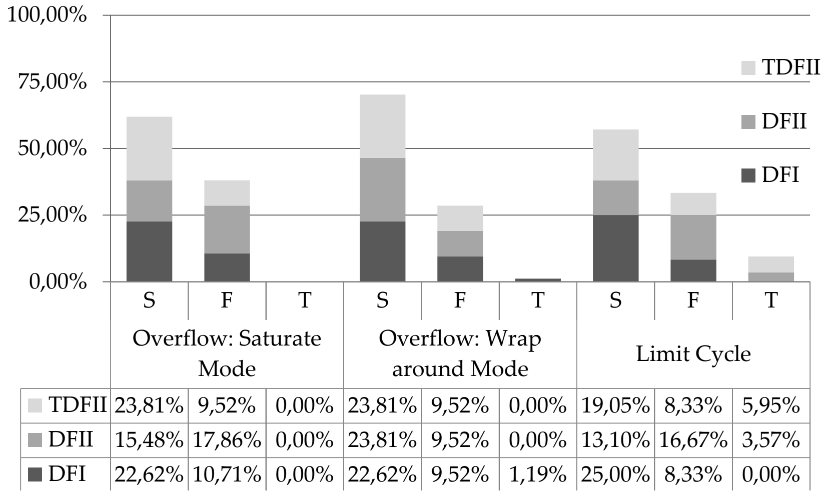
Table 4.3 shows the obtained verification results, where *VT* denotes the verification time, in seconds, *VR* represents the verification result, *S* means success, that is, DSVerifier did not find a failure up to $k = 10$, *F* means failed, that is, DSVerifier found a property violation and then returned 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.

Fig. 4.3 summarizes the obtained verification results, for each realization form, which show that 29% of the 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% of the controller implementations, which means that an overflow violation was detected in intermediate nodes, during verification procedures. In LCO verification, 33% of the controller implementations failed, while 57% of them presented successful verification and 10% indicated timeout. The larger times in LCO verification procedures are explained by the more complex algorithm employed, with non-deterministic initial states, (constant) inputs, and oscillation periods. Despite that,

Table 4.3: Verification results for the digital controllers used in the mentioned quadrotor attitude system

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

LCO verification procedures were concluded for 90% of the chosen benchmarks, while overflow ones were concluded for 99% and 100% of them, in wrap-around and saturate mode, respectively.



	Overflow: Saturate Mode			Overflow: Wrap around Mode			Limit Cycle		
	S	F	T	S	F	T	S	F	T
▨ TDFII	23,81%	9,52%	0,00%	23,81%	9,52%	0,00%	19,05%	8,33%	5,95%
▨ DFII	15,48%	17,86%	0,00%	23,81%	9,52%	0,00%	13,10%	16,67%	3,57%
■ DFI	22,62%	10,71%	0,00%	22,62%	9,52%	1,19%	25,00%	8,33%	0,00%

Figure 4.3: Summary of the obtained verification results, per realization form, for the evaluated controllers.

4.4.1 Overflow Occurrence Discussion

Particularly, digital controller C_5 presented overflow in every (possible) implementation, realization, and overflow mode (saturate and wrap-around). That happened because when we compute the bits using equation (3.11) suggested at least 17 bits [95], but the UAV architecture used for the experiments supports only 16 bits, as mentioned earlier. In the failed results, DSVerifier also returned counterexamples, which can be easily reproduced through DSValidator,*i.e.*, a MATLAB toolbox designed to automatically reproduce counterexamples generated by DSVerifier [102]. The maximum fixed-point implementation for our UAV architecture is 16-bits, and for the controller C_5 the implementation are distributed as $< 10,6 >$, $< 12,4 >$ and $< 13,3 >$, and the outputs produced by the computation of any realization form for controller C_5 exceeds the maximum/minimum representation for each fixed-point implementation.

The time spent in overflow verification is relatively low, with a maximum value of 670 and 436 seconds, in saturate and wrap-around modes, respectively, which happened for successful verification procedures for controller C_4, using TDFII realization. Lower verification times, if compared with previous studies [26], 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. The time spent in verification for different realization can be explained due to each realization has different number of multipliers or adders, and it could increase the complexity of computation.

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, regarding specifically saturate mode, by changing realization forms. As an example, overflow occurs for 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 [95]. Fig. 4.4 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 controller output; red dashed lines denote representation limits. One may notice that, in the first sample, the output is slightly greater than the numerical representation limit; however, in Fig. 4.5, the output does not contain overflow violation, using the same controller specified for C_1 and fixed-point format $\langle 10,6 \rangle$.

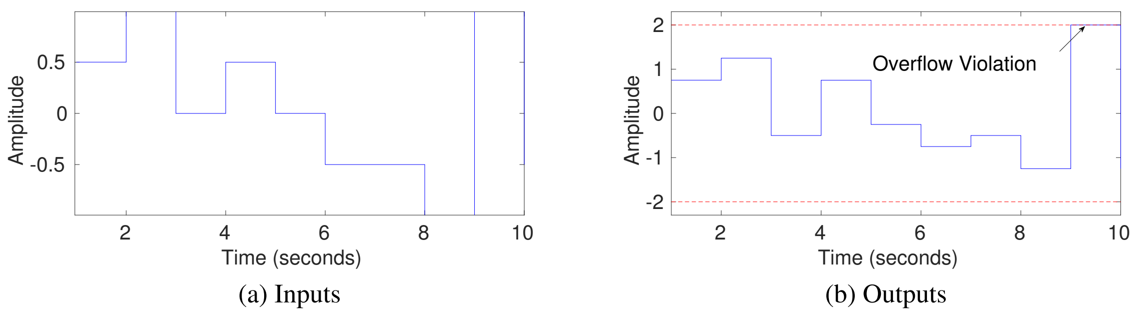


Figure 4.4: Arithmetic overflow occurrence in controller C_1, with DFI realization and a format containing 2 bits in integer part and 14 bits in fractional one.

Carletta's Issue

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.* [95] 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.* [53]. For instance, when the impulse

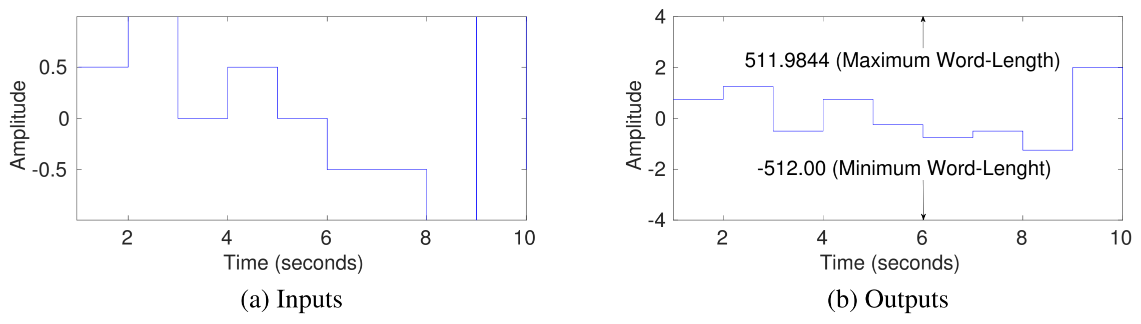


Figure 4.5: Absence of arithmetic overflow in Controller C_1, with DFI realization and a format containing 10 bits in integer part and 6 bits in fractional one.

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, this behavior was noticed when the maximum output value is a power of 2, due to the way it is computed by the logarithm, in equation (3.11).

Although the mentioned finding revealed a flawed dimensioning procedure, it further proved that the proposed methodology is sound and reliable. As a 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 part. As previously presented, the methodology described by Carletta [95] is shown in equation (4.1), as follows:

$$j = \lceil \log_2(\|h\|_1 \cdot \|x\|_\infty) \rceil + 1, \tag{4.1}$$

However, this solution is not asymmetric for two's complement representation. In order to solve this issue, the algorithm purposed by Volkova *et al.* [53] considers a second term to compute a correct dimensioning, as follows:

$$j = \lceil \log_2(\|h\|_1 \cdot \|x\|_\infty) - \log_2(1 - 2^{(1-w)}) \rceil, \tag{4.2}$$

where,

$w = i + f$, with i-integer bits and f-fractional bits.

Additionally, some controllers presented overflow, even with formats with many integer bits (the second and third ones), in saturate mode. That was expected, given that intermediate operations are checked in saturate mode and the adopted dimensioning procedure only takes into account system outputs.

Overflow's conclusion

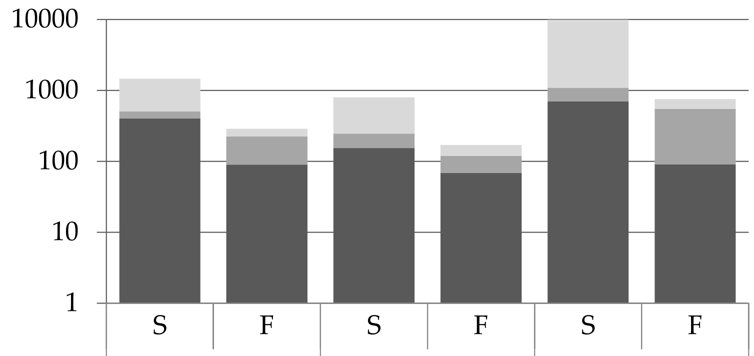
Finally, one may notice that C_4 presented the highest verification time, which occurred due to the high order associated to its implementation. Verification times typically increase with controller complexity, given that direct-form implementations need two-nested loops to generate a controller order. 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.

4.4.2 Limit Cycle Occurrence Discussion

An example of LCO occurrence was noticed (see Table 4.3) 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. 4.7a. 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).

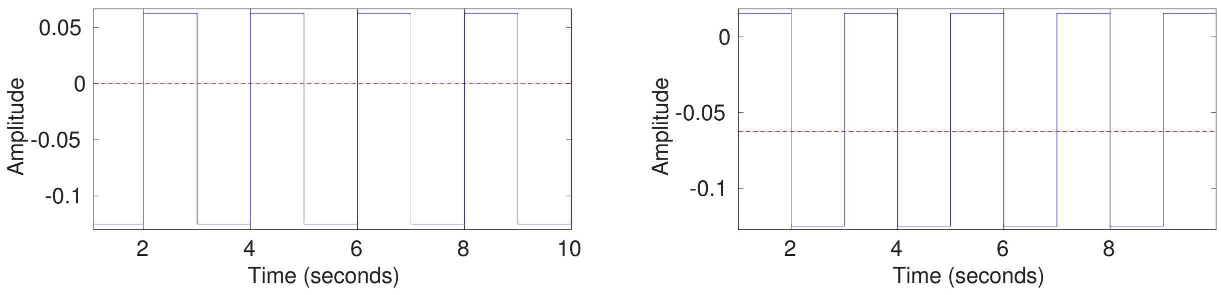
If the same controller is implemented in DFI format, LCO events are also detected by DSVerifier. In particular, DSVerifier found that initial states -0.109375, 0.015625, and -0.1250 and an input sequence -0.0625 lead to the mentioned LCO. Fig. 4.7b shows an LCO occurrence in C_5, with DFI realization, 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 4.4.1, so the LCO occurrences illustrated in Figs. 4.7a and 4.7b 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 com-



	Overflow: Saturate Mode		Overflow: Wrap around Mode		Limit Cycle	
	S	F	S	F	S	F
▢ TDFII	955	63	556	50	8845	203
▨ DFII	103	135	92	51	388	462
■ DFI	405	90	154	69	704	91

Figure 4.6: Verification-time results for the digital controllers in the mentioned quadrotor attitude system.



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

Figure 4.7: LCO occurrence in C_5, with different realizations and format $\langle 13, 3 \rangle$.

puting cost. In fact, LCO verifications tend to take longer, due to their algorithmic complexity, *i.e.*, a search for persistent oscillations on the system 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 algorithm is interrupted when a failure is found.

4.4.3 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 $< 8, 8 >$ and direct-form realizations (*i.e.*, DFI, DFII and TDFII). As a 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. 4.8, which shows granular LCO for a DFII realization form.

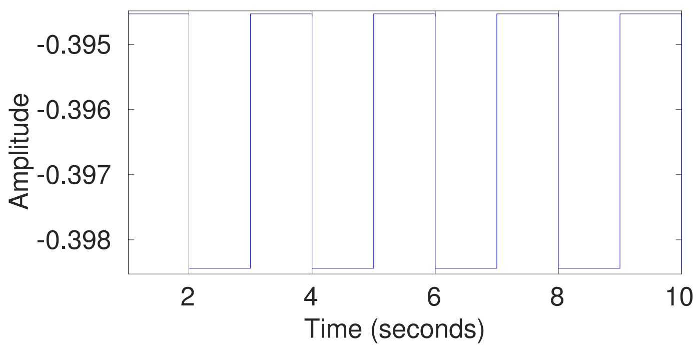


Figure 4.8: Granular LCO for the synthesized controller C_{10}, in DFII realization.

In spite of the synthesized controller C_{10} is stable, a granular LCO was found. The result obtained here about the granular LCO is very interesting due to the stability found by DSSynth is not completely safe [68, 105]. In fact, if we reproduce the stability checking using MATLAB, we can validate the stability of C_{10}, and we can conclude that indeed the controller C_{10} is stable. Possibly, employing an algorithm to exhaustively detect the absence of LCO [103, 104] could be integrated into DSSynth [68, 105] in order to synthesize stable and LCO-free digital controllers.

4.4.4 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 equation (4.3), which represents roll (ϕ) and pitch (θ) angle dynamics.

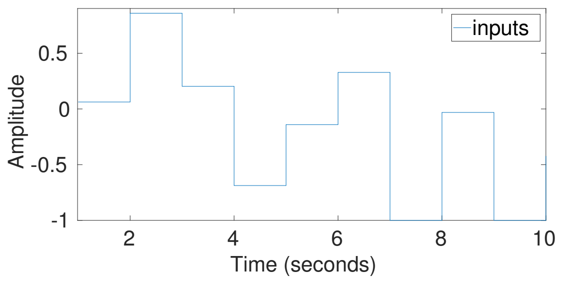
$$G(z) = \frac{-0.06875z^2}{z^2 - 1.696z + 0.7089}. \tag{4.3}$$

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. 4.9. Fig. 4.9(a) shows effects in the plant defined by equation (4.3), Fig. 4.9(b) presents the inputs produced by DSVerifier, which were extracted from the counterexample related to the controller C_5, and, finally, Fig. 4.9(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 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.

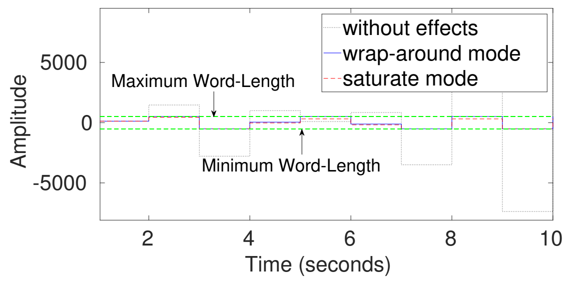
A second simulation was performed using the angle rate controller C_6, in DFI realization and with format $\langle 10,6 \rangle$, in order to investigate LCO effects regarding roll (ϕ) and pitch (θ) angles' behavior. In particular, C_6 presented granular LCO. Fig. 4.10 shows, in part (a), the digital 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 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.

4.4.5 Verification Efficiency Discussion

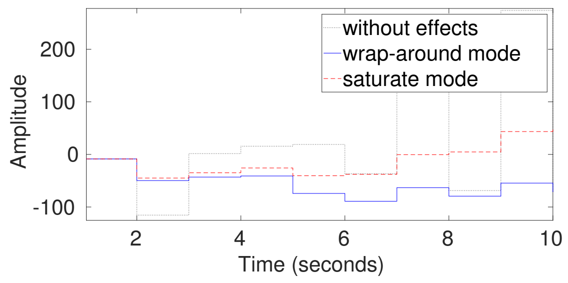
It is very important to elaborate on verification efficiency. The mean time (disregarding timeouts) spent for verifying a controller is around $22s$ ($\sigma = 76s$) for overflow verification, in saturate mode, $13s$ ($\sigma = 48s$), in wrap-around mode, and $146s$ ($\sigma = 342s$), for LCO verification.



(a) Effects in roll and pitch angles, whose dynamics is defined by equation (4.3).



(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.

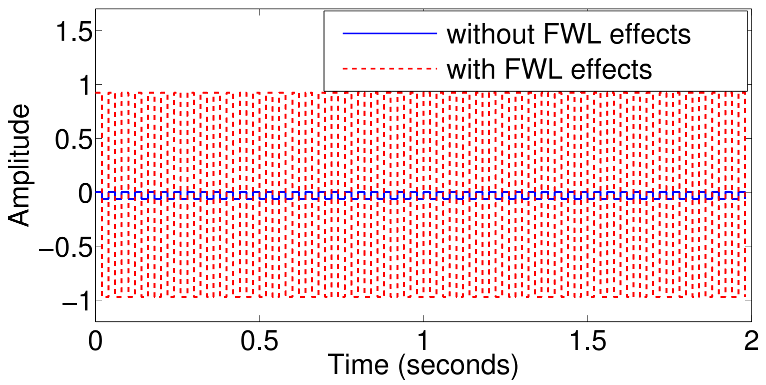


(c) Overflow closed-loop effects reproducibility in DSValidator.

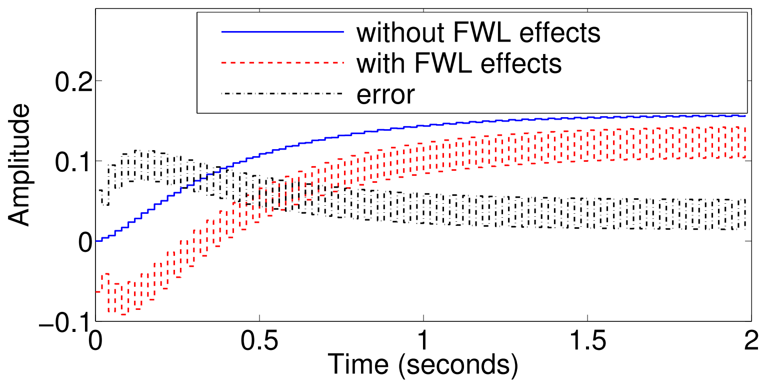
Figure 4.9: Overflow effects in closed-loop dynamics.

One may notice that the high standard deviations regarding verification times indicate that time spent in a successful verification is much shorter 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 maximum values 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



(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 equation (4.3).

Figure 4.10: LCO effects in closed-loop dynamics of pitch/roll angles.

compared with previous studies [26], 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. 4.6, 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 represents 10% of our benchmarks.

4.5 On the validation of DSVerifier results

All verification results provided by DSVerifier v2.0.3 were validated with DSValidator v1.0.1 [102], 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 details about DSValidator features and usage are described in the dissertation's appendix C.

4.5.1 Automated Counterexample Reproducibility

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 typical MATLAB features (*e.g.*, operational functions, rounding functions, logical functions, conditional functions, loops and structs), and as a consequence, it is also suitable for investigating digital controller and filter behavior, considering the 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 4.4, 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 us to check the property violation in graphical mode, using MATLAB, which makes re-implementation phases easier, in the proposed DSVerifier-aided verification methodology (see Fig. 3.1). Indeed, it's very important to notice that DSValidator returns a `.MAT`-file that represents the digital system with its implementation (*e.g.*, realization, fixed-point format, inputs). By combining this implementation extracted from the counterexample, the re-design of the digital system is more practical, due to the fact that a control engineer is then able not only to implement the same digital system with a different realization or fixed-point format, but he can also check with DSValidator if the violation is still occurring, *i.e.,* through graphs, property-verification simulation, and also result validation. due to this reason, DSValidator is a strong tool to support the verification performed by the DSVerifier *v*2.0.

In order to ensure the absence of LCOs detected by DSVerifier, an algorithm proposed by Bauer [103, 104] 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

Table 4.4: 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	26	1	0.057538 s

(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.

Note that the automated validation of all counterexamples took less than 1 second. We consider such times short enough to be of practical use. The present results also show that all counterexamples (except one) generated by the underlying verifier, considering FWL effects and different realizations forms (*i.e.*, DFI, DFII and TDFII), are sound and reliable, since DSVerifier is able to simulate the underlying digital system in MATLAB and then reproduce the respective counterexample. Nonetheless, for the limit cycle property, there is one counterexample that was not reproduced in DSVerifier. Previously, DSVerifier did not take into account overflow in intermediate operations to compute the system's output using the DFII realization form. Indeed, this bug was confirmed and fixed by the DSVerifier's developer via a github commit.[1]

[1]https://github.com/ssvlab/dsverifier/commit/88e857bdbc74a7ce3c74d327e2a1e7a246fa48cc

Chapter 5

Conclusions

This work described an SMT-based BMC verification methodology, 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 differs from previous approaches, once it investigates FWL effects over digital-controller implementations. Overflow and LCO were investigated in 10 different digital controllers, through 84 different implementations (realizations and representations).

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 a 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. However, this combination of realization and numerical representation is performed by the control engineer, and as a future work could be implemented as an algorithm to permute different realizations and fixed-point representations until the right implementation could be found. In addition, a flawed dimensioning procedure (available in literature) was identified and the resulting violations were detected by DSVerifier, which reinforces its effectiveness and applicability.

According to the provided verification results, regarding overflow verification using wrap-around mode, 30% of the adopted controller implementations presented violations, which represents 24 elements, while in saturate mode 40% of the controller implementations presented violations, which represents 32 elements. For LCO validation, 30% that of controllers implementations are susceptible to LCO, while the other 70% present absence of LCO. Finally, all mentioned results were reproduced with DSValidator, which shows that DSVerifier is sound and reliable and can also be incorporated into digital-controller design processes, in order to obtain more robust implementations.

In addition, the method presented by Carletta *et al.* [95] is not really accurate enough and it 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.* [53]. The algorithm purposed by Carletta is working in some cases, except for some ones where $||h||_1.||x||_\infty$ is just strictly below 2^{bits}, but close enough to make the fixed-point format chosen inadequate.

5.1 Future Work

As future work, this study will be extended to altitude and position UAV models and will also support closed-loop verification using k-induction techniques [106, 107], which considers system dynamics and how it is affected by FWL effects. Further studies will also investigate the consequences of attitude controller fragility in an UAV mission, considering every control software module.

Additionally, other digital-controller representations can be included, *e.g.*, state-space based forms [22]. In this respect, further studies could include a method to automatically fix controller implementations using fault localization methods [108, 109], where an optimal instance can be found, considering hardware constraints. Finally, the latter can also be integrated into DSSynth [110], which would greatly improve its results and consequently provide LCO- and overflow-free implementations.

Finally, DSVerifier could include the verification of nonlinear systems [111], and also include other properties for digital filters verification (*e.g.,* magnitude and phase [24, 29]). In addition, DSVerifier could be integrated with DSValidator [102] and DSSynth [110], in order to synthesize stable controllers (DSSynth), verified by a bounded model checking tool (DSVerifier), and also validated by a reproducibility tool (DSValidator), which could be a strategy to design reliable digital controllers, and free from errors related to FWL effects.

References

[1] BOUABDALLAH, S.; MURRIERI, P.; SIEGWART, R. Design and control of an indoor micro quadrotor. In: *IEEE Conference Robotics and Automation*. 2004. v. 5, p. 4393–4398 Vol.5. ISSN 1050-4729.

[2] SADEGHZADEH, I.; ZHANG, Y. A review on fault-tolerant control for unmanned aerial vehicles (UAVs). *Infotech@ Aerospace*, Springer Netherlands, v. 73, n. 1-4, p. 535–555, 2011. ISSN 0921-0296.

[3] LI, K. *Fukushima Radiation Disaster News: Japan Uses Drone to Check Rad Levels at Nuclear Site*. `http://www.latinpost.com/articles/6442/20140127/fukushima-radiation-disaster-news-japan-uses-drone-check-rad-levels.htm`. Accessed: 01-05-2018.

[4] ALUR, R.; COURCOUBETIS, C.; DILL, D. Model-checking in dense real-time. *Information and Computation*, v. 104, n. 1, p. 2–34, 1993.

[5] ALUR, R.; COURCOUBETIS, C.; DILL, D. Model-checking for real-time systems. In: *ACM/IEEE Symposium on Logic in Computer Science*. 1990. p. 414–425.

[6] DAVID, A.; BEHRMANN, G.; LARSEN, K.; YI, W. A tool architecture for the next generation of UPPAAL. *Lecture Notes in Computer Science (LNCS)*, v. 2757, p. 352–366, 2003.

[7] HENZINGER, T.; HO, P.-H.; WONG-TOI, H. Hytech: the next generation. In: *IEEE Real-Time Systems Symposium*. 1995. p. 56–65.

[8] ALUR, R. Formal verification of hybrid systems. In: *International Conference on Embedded Software (EMSOFT)*. 2011. p. 273–278.

[9] CÁMARA, P. de la; CASTRO, J. R.; GALLARDO, M.; MERINO, P. Verification support for arinc-653-based avionics software. *Software: Testing, Verification and Reliability*, v. 21, n. 4, p. 267–298, 2011.

[10] HOLZMANN, G. J. The model checker SPIN. *IEEE Transactions on Software Engineering*, v. 23, n. 5, p. 279–295, May 1997. ISSN 0098-5589.

[11] MCMILLAN, K. L. *Symbolic Model Checking*. Norwell, MA, USA: Kluwer Academic Publishers, 1993. ISBN 0792393805.

[12] CIMATTI, A.; CLARKE, E.; GIUNCHIGLIA, F.; ROVERI, M. NUSMV: a new symbolic model checker. *International Journal on Software Tools for Technology Transfer (STTT)*, v. 2, n. 4, p. 410–425, 2000. ISSN 1433-2779.

[13] GROCE, A.; HAVELUND, K.; SMITH, M. From scripts to specifications: the evolution of a flight software testing effort. In: ACM. *International Conference on Software Engineering-Volume 2*. 2010. p. 129–138.

[14] PINGREE, P. J.; MIKK, E.; HOLZMANN, G. J.; SMITH, M. H.; DAMS, D. Validation of mission critical software design and implementation using model checking [spacecraft]. In: *Digital Avionics Systems Conference*. 2002. v. 1, p. 6A4–1–6A4–12 vol.1.

[15] WHALEN, M.; COFER, D.; MILLER, S.; KROGH, B. H.; STORM, W. Integration of formal analysis into a model-based software development process. In: *International Workshop on Formal Methods for Industrial Critical Systems*. 2008. (Lecture Notes in Computer Science (LNCS), v. 4916), p. 68–84. ISBN 3-540-79706-8, 978-3-540-79706-7.

[16] SREEMANI, T.; ATLEE, J. M. Feasibility of model checking software requirements: a case study. In: *Eleventh Annual Conference on Computer Assurance*. 1996. p. 77–88.

[17] PEDROZA, N. R.; MACKUNIS, W.; GOLUBEV, V. V. Robust nonlinear regulation of limit cycle oscillations in uavs using synthetic jet actuators. *Robotics*, v. 3, n. 4, p. 330, 2014. ISSN 2218-6581. Disponível em: <http://www.mdpi.com/2218-6581/3/4/330>.

[18] WU, J.; LI, G.; CHEN, S.; CHU, J. A μ-based optimal finite-word-length controller design. *Automatica*, v. 44, n. 12, p. 3093–3099, 2008.

[19] WU, J.; LI, G.; CHEN, S.; CHU, J. Robust finite word length controller design. *Automatica*, v. 45, n. 12, p. 2850–2856, 2009.

[20] MATHWORKS, I. S. *MATLAB and SIMULINK: Student Version*. : MathWorks, 2004. (MATLAB & SIMULINK: Student Version). ISBN 9780975578735.

[21] GW, J. *Labview graphical programming: Practical applications in instrumentation and control*. : McGraw-Hill School Education Group, 1997.

[22] ÅSTRÖM, K.; WITTENMARK, B. *Computer-controlled systems: theory and design*. : Prentice Hall, 1997. (Prentice Hall information and system sciences series). ISBN 9780133148992.

[23] ISTEPANIAN, R.; WHIDBORNE, J. *Digital Controller Implementation and Fragility: A Modern Perspective*. : Springer London, 2001. (Advances in Industrial Control). ISBN 9781852333904.

[24] DINIZ, P.; NETTO, S.; SILVA, E. D. *Digital Signal Processing: System Analysis and Design*. New York, NY, USA: Cambridge University Press, 2002. ISBN 0521781752.

[25] PADGETT, W.; ANDERSON, D. *Fixed-Point Signal Processing*. : Morgan & Claypool, 2009. (Synthesis lectures on signal processing). ISBN 9781598292589.

[26] BESSA, I.; ISMAIL, H.; CORDEIRO, L.; FILHO, J. Verification of fixed-point digital controllers using direct and delta forms realizations. *Design Automation for Embedded Systems*, 2016.

[27] LI, G.; ZHAO, Z. On the generalized dfiit structure and its state-space realization in digital filter implementation. *IEEE Transactions on Circuits and Systems I: Regular Papers*, v. 51, n. 4, p. 769–778, April 2004. ISSN 1549-8328.

[28] GAZSI, L. Explicit formulas for lattice wave digital filters. *IEEE Transactions on Circuits and Systems*, v. 32, n. 1, p. 68–88, Jan 1985. ISSN 0098-4094.

[29] PROAKIS, J.; MANOLAKIS, D. *Digital signal processing: principles, algorithms, and applications*. : Prentice Hall, 1996. (Prentice-Hall International editions). ISBN 9780133737622.

[30] HARNEFORS, L. Implementation of Resonant Controllers and Filters in Fixed-Point Arithmetic. *IEEE Transactions on Industrial Electronics*, v. 56, n. 4, p. 1273–1281, 2009. ISSN 0278-0046.

[31] PERETZ, M.; BEN-YAAKOV, S. Digital control of resonant converters: Resolution effects on limit cycles. *IEEE Transactions on Power Electronics*, v. 25, n. 6, p. 1652–1661, June 2010. ISSN 0885-8993.

[32] GRANAS, A.; DUGUNDJI, J. *Fixed Point Theory*. : Springer, 2003. (Monographs in Mathematics). ISBN 9780387001739.

[33] JACKSON, L.; KAISER, J.; MCDONALD, H. An approach to the implementation of digital filters. *IEEE Trans. Audio Electroacoust*, v. 34, n. 3, p. 413–421, 1968.

[34] MIDDLETON, R.; GOODWIN, G. Improved finite word length characteristics in digital control using delta operators. *IEEE Transactions on Automatic Control*, v. 31, n. 11, p. 1015–1021, November 1986. ISSN 0018-9286.

[35] LI, G. On pole and zero sensitivity of linear systems. *IEEE Transactions on Circuits and Systems I: Fundamental Theory and Applications*, v. 44, n. 7, p. 583–590, Jul 1997. ISSN 1057-7122.

[36] LI, G. On the structure of digital controllers with finite word length consideration. *IEEE Transactions on Automatic Control*, v. 43, n. 5, p. 689–693, May 1998. ISSN 0018-9286.

[37] MIDDLETON, R. H.; GOODWIN, G. C. *Digital Control and Estimation: A Unified Approach*. : Prentice Hall Professional Technical Reference, 1990. ISBN 0132116650.

[38] GEVERS, M.; LI, G. *Parametrizations in control, estimation and filtering problems: accuracy aspects*. : Springer Science & Business Media, 2012.

[39] YANG, G.; GUO, X.; CHE, W.; GUAN, W. *Linear Systems: Non-Fragile Control and Filtering*. : Taylor & Francis, 2013. ISBN 9781466580350.

[40] KEEL, L.; BHATTACHARYYA, S. Robust, fragile, or optimal? *IEEE Transactions on Automatic Control*, v. 42, n. 8, p. 1098–1105, 1997. ISSN 0018-9286.

[41] SUNG, W.; KUM, K.-I. Simulation-based word-length optimization method for fixed-point digital signal processing systems. *IEEE Transactions on Signal Processing*, v. 43, n. 12, p. 3087–3090, Dec 1995. ISSN 1053-587X.

[42] LI, G.; ANDERSON, B. D. O.; GEVERS, M.; PERKINS, J. E. Optimal fwl design of state-space digital systems with weighted sensitivity minimization and sparseness consideration. *IEEE Transactions on Circuits and Systems I: Fundamental Theory and Applications*, v. 39, n. 5, p. 365–377, May 1992. ISSN 1057-7122.

[43] LOPEZ, B.; HILAIRE, T.; DIDIER, L.-S. Formatting bits to better implement signal processing algorithms. In: *4th international Conference on Pervasive and Embedded Computing and Communication Systems (PECCS)*. Lisbon, Portugal: ScitePress, 2014. p. 104–111. Disponível em: <http://hal.upmc.fr/hal-01076049>.

[44] MENARD, D.; ROCHER, R.; SENTIEYS, O.; SIMON, N.; DIDIER, L. S.; HILAIRE, T.; LOPEZ, B.; GOUBAULT, E.; PUTOT, S.; VEDRINE, F.; NAJAHI, A.; REVY, G.; FANGAIN, L.; SAMOYEAU, C.; LEMONNIER, F.; CLIENTI, C. Design of fixed-point embedded systems (defis) french anr project. In: *Proceedings of the 2012 Conference on Design and Architectures for Signal and Image Processing*. 2012. p. 1–2.

[45] PARASHAR, K. N.; MENARD, D.; SENTIEYS, O. A polynomial time algorithm for solving the word-length optimization problem. In: *2013 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. 2013. p. 638–645. ISSN 1092-3152.

[46] SARBISHEI, O.; RADECKA, K.; ZILIC, Z. Analytical optimization of bit-widths in fixed-point lti systems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, v. 31, n. 3, p. 343–355, March 2012. ISSN 0278-0070.

[47] MENARD, D.; ROCHER, R.; SENTIEYS, O. Analytical fixed-point accuracy evaluation in linear time-invariant systems. *IEEE Transactions on Circuits and Systems I: Regular Papers*, v. 55, n. 10, p. 3197–3208, Nov 2008. ISSN 1549-8328.

[48] SKAF, J.; BOYD, S. P. Filter design with low complexity coefficients. *IEEE Transactions on Signal Processing*, v. 56, n. 7, p. 3162–3169, July 2008. ISSN 1053-587X.

[49] ISTEPANIAN, R. S. H.; WHIDBORNE, J. F. Multi-objective design of finite word-length controller structures. In: *Proceedings of the 1999 Congress on Evolutionary Computation-CEC99 (Cat. No. 99TH8406)*. 1999. v. 1, p. 68 Vol. 1.

[50] BALAKRISHNAN, V.; BOYD, S. On computing the worst-case peak gain of linear systems. In: *[1992] Proceedings of the 31st IEEE Conference on Decision and Control*. 1992. p. 2191–2192 vol.2.

[51] VIASSOLO, D. E. Realizations for fixed-point implementation of controllers and filters with peak-to-peak scaling. In: *Proceedings of the 2003 American Control Conference, 2003.* 2003. v. 1, p. 83–88 vol.1. ISSN 0743-1619.

[52] THIELE, L. On the sensitivity of linear state-space systems. *IEEE Transactions on Circuits and Systems*, v. 33, n. 5, p. 502–510, May 1986. ISSN 0098-4094.

[53] VOLKOVA, A.; HILAIRE, T.; LAUTER, C. 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*. 2015. p. 737–741.

[54] LI, T.; ZHENG, W. X. New stability criterion for fixed-point state-space digital filters with generalized overflow arithmetic. *IEEE Transactions on Circuits and Systems II: Express Briefs*, v. 59, n. 7, p. 443–447, July 2012. ISSN 1549-7747.

[55] FENG, Y.; CHEVREL, P.; HILAIRE, T. Generalised modal realisation as a practical and efficient tool for FWL implementation. *International Journal of Control*, Taylor & Francis, v. 84, n. 1, p. 66–77, fev. 2011. Disponível em: <https://hal.archives-ouvertes.fr/hal-00564076>.

[56] GOLNARAGHI, F.; KUO, B. C. *Automatic Control Systems*. 9th. ed. : Wiley Publishing, 2009. ISBN 0470048964, 9780470048962.

[57] PETERCHEV, A.; SANDERS, S. Quantization resolution and limit cycling in digitally controlled pwm converters. *IEEE Transactions on Power Electronics*, v. 18, n. 1, p. 301–308, Jan 2003. ISSN 0885-8993.

[58] MOURA, L.; BJORNER, N. Satisfiability Modulo Theories: An Appetizer. In: *Formal Methods: Foundations and Applications*. : Springer-Verlag, 2009. p. 23–36. ISBN 978-3-642-10451-0.

[59] MOURA, L. D.; BJØRNER, N. Z3: An Efficient SMT Solver. In: *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. Berlin, Heidelberg: Springer-Verlag, 2008. (TACAS'08/ETAPS'08), p. 337–340. ISBN 3-540-78799-2, 978-3-540-78799-0.

[60] BRUMMAYER, R.; BIERE, A. Boolector: An Efficient SMT Solver for Bit-Vectors and Arrays. In: *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. 2009. (Lecture Notes in Computer Science (LNCS), v. 5505), p. 174–177.

[61] BJORNER, N.; MOURA, L. *Z310: Applications, Enablers, Challenges and Directions*.

[62] LEE, E. Cyber physical systems: Design challenges. In: *IEEE International Symposium on Real-Time Computing (ISORC)*. 2008. p. 363–369.

[63] BIERE, A.; CIMATTI, A.; CLARKE, E. M.; ZHU, Y. Symbolic model checking without bdds. In: *Tools and Algorithms for Construction and Analysis of Systems, 5th International Conference, TACAS '99, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS'99, Amsterdam, The Netherlands, March 22-28, 1999, Proceedings*. 1999. (Lecture Notes in Computer Science, v. 1579), p. 193–207.

[64] COX, A.; SANKARANARAYANAN, S.; CHANG, B.-Y. E. A bit too precise? Verification of quantized digital filters. *International Journal on Software Tools for Technology Transfer*, Springer Berlin Heidelberg, v. 16, n. 2, p. 175–190, 2014. ISSN 1433-2779.

[65] ABREU, R. B.; CORDEIRO, L. C.; FILHO, E. B. L. Verifying Fixed-Point Digital Filters using SMT-Based Bounded Model Checking. In: *Brazilian Symposium on Telecommunications (SBrT)*. 2013.

[66] CORDEIRO, L.; FISCHER, B.; MARQUES-SILVA, J. SMT-Based Bounded Model Checking for Embedded ANSI-C Software. *IEEE Transaction on Software Engineering*, v. 38, n. 4, p. 957–974, 2012. ISSN 0098-5589.

[67] MELLINGER, D.; MICHAEL, N.; KUMAR, V. Trajectory generation and control for precise aggressive maneuvers with quadrotors. *Int. J. Rob. Res.*, Sage Publications, Inc., Thousand Oaks, CA, USA, v. 31, n. 5, p. 664–674, abr. 2012. ISSN 0278-3649. Disponível em: <http://dx.doi.org/10.1177/0278364911434236>.

[68] ABATE, A.; BESSA, I.; CATTARUZZA, D.; LUCAS, C.; DAVID, C.; KESSELI, P.; KROENING, D. Sound and automated synthesis of digital stabilizing controllers for continuous plants. In: *20th ACM International Conference on Hybrid Systems: Computation and Control (HSCC)*. 2017. p. 1–10.

[69] SOLAR-LEZAMA, A. Program sketching. *International Journal on Software Tools for Technology Transfer (STTT)*, v. 15, n. 5-6, p. 475–495, 2013.

[70] BESSA, I.; ISMAIL, H.; PALHARES, R.; CORDEIRO, L.; FILHO, J. E. C. Formal non-fragile stability verification of digital control systems with uncertainty. *IEEE Transactions on Computers*, v. 66, n. 3, 2017.

[71] KROENING, D.; TAUTSCHNIG, M. CBMC - C Bounded Model Checker (competition contribution). In: *International Conference on Tools and Algorithms for the Construction and Analysis of Systems TACAS*. 2014. v. 8413, p. 389–391.

[72] FRUTUOSO, A.; FILHO, J. ao E.; HENRIQUE, A. Discrete robust controller for quadrotors stability. In: *International Congress of Mechanical Engineering COBEM*. 2015.

[73] ISMAIL, H.; BESSA, I.; CORDEIRO, L. C.; FILHO, E. B. de L.; FILHO, J. E. C. Dsverifier: A bounded model checking tool for digital systems. In: *Model Checking Software - 22nd International Symposium, SPIN 2015, Stellenbosch, South Africa, August 24-26, 2015, Proceedings*. 2015. (Lecture Notes in Computer Science, v. 9232), p. 126–131.

[74] OGATA, K. *Discrete-Time Control Systems*. : Prentice-Hall International, 1995. (Prentice Hall International editions). ISBN 9780133286427.

[75] BEYER, D. Reliable and reproducible competition results with benchexec and witnesses (report on sv-comp 2016). *Lecture Notes in Computer Science (LNCS)*, v. 9636, p. 887–904, 2016.

[76] BESSA, I.; ABREU, R.; FILHO, J.; CORDEIRO, L. SMT-based bounded model checking of fixed-point digital controllers. In: *Annual Conference of the IEEE Industrial Electronics Society (IECON)*. 2014. p. 295–301.

[77] GUENNEBAUD, G. *Eigen: a C++ Linear Algebra Lirary*. 2011.

[78] FADALI, S.; VISIOLI, A. *Digital Control Engineering: Analysis and Design*. : Elsevier/Academic Press, 2009. (Electronics & Electrical, v. 303). ISBN 9780123744982.

[79] COX, A.; SANKARANARAYANAN, S.; CHANG, B.-Y. E. A Bit Too Precise? Bounded Verification of Quantized Digital Filters. Springer Berlin Heidelberg, v. 7214, p. 33–47, 2012.

[80] GROZA, A.; LETIA, I. A.; GORON, A.; ZAPOROJAN, S. A formal approach for identifying assurance deficits in unmanned aerial vehicle software. In: *Progress in Systems Engineering*. : Springer, 2015. p. 233–239.

[81] KIM, B. Verification and validation of UAV mission planning for human automation collaboration. In: *IIE Annual Conference and Expo*. 2014.

[82] KIM, B. U.; HUMPHREY, L. R. Satisfiability checking of ltl specifications for verifiable UAV mission planning. In: *Aerospace Sciences Meeting*. : American Institute of Aeronautics and Astronautics, 2014.

[83] HUMPHREY, L. Model checking UAV mission plan. In: *AIAA Modeling and Simulation Technologies Conference*. 2012.

[84] HUMPHREY, L.; PATZEK, M. Model checking human-automation UAV mission plans. In: *AIAA Guidance, Navigation, and Control (GNC) Conference*. 2013.

[85] HUMPHREY, L. R. Model checking for verification in UAV cooperative control applications. In: *Recent Advances in Research on Unmanned Aerial Vehicles*. : Springer, 2013. p. 69–117.

[86] ZUTSHI, A.; SANKARANARAYANAN, S.; DESHMUKH, J. V.; KAPINSKI, J.; JIN, X. Falsification of safety properties for closed loop control systems. In: *Proceedings of the 18th International Conference on Hybrid Systems: Computation and Control*. New York,

NY, USA: ACM, 2015. (HSCC '15), p. 299–300. ISBN 978-1-4503-3433-4. Disponível em: <http://doi.acm.org/10.1145/2728606.2728648>.

[87] GROCE, A.; HAVELUND, K.; HOLZMANN, G.; JOSHI, R.; XU, R.-G. Establishing flight software reliability: Testing, model checking, constraint-solving, monitoring and learning. *Annals of Mathematics and Artificial Intelligence*, Springer, v. 70, n. 4, p. 315–349, 2014.

[88] SIRIGINEEDI, G.; TSOURDOS, A.; WHITE, B. A.; ŻBIKOWSKI, R. Kripke modelling and verification of temporal specifications of a multiple uav system. *Annals of Mathematics and Artificial Intelligence*, Kluwer Academic Publishers, Hingham, MA, USA, v. 63, n. 1, p. 31–52, set. 2011. ISSN 1012-2443.

[89] LERDA, F.; KAPINSKI, J.; MAKA, H.; CLARKE, E.; KROGH, B. Model checking in-the-loop: Finding counterexamples by systematic simulation. In: *American Control Conference*. 2008. p. 2734–2740. ISSN 0743-1619.

[90] LIPKA, R.; PASKA, M.; POTUZAK, T. Simulation testing and model checking: A case study comparing these approaches. In: *Software Engineering for Resilient Systems - 6th International Workshop, SERENE 2014, Budapest, Hungary, October 15-16, 2014. Proceedings*. 2014. p. 116–130.

[91] BEYER, D.; LEMBERGER, T. 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, HVC 2017, Haifa, Israel, November 13-15, 2017, Proceedings*. 2017. (LNCS, v. 10629), p. 99–114.

[92] MONNIAUX, D. Compositional analysis of floating-point linear numerical filters. In: *Computer Aided-Verification*. 2005, (Lecture Notes in Computer Science (LNCS), v. 3576). p. 199–212. ISBN 978-3-540-27231-1.

[93] MUNIER, P. *Polyspace*. 2013. 113-142 p. (Static Analysis of Software: The Abstract Interpretation).

[94] HAMON, G. Simulink design verifier – applying automated formal methods to simulink and stateflow. In: *Third Workshop on Automated Formal Methods*. 2008. p. 1–2.

[95] CARLETTA, J.; VEILLETTE, R.; KRACH, F.; FANG, Z. Determining appropriate precisions for signals in fixed-point IIR filters. In: *Design Automation Conference*. 2003. p. 656–661.

[96] ABREU, R. B.; GADELHA, M. Y. R.; CORDEIRO, L. C.; FILHO, E. B. de L.; SILVA, W. S. da. Bounded model checking for fixed-point digital filters. *Journal of the Brazilian Computer Society*, v. 22, n. 1, p. 20, 2016. ISSN 1678-4804.

[97] EÉN, N.; SÖRENSSON, N. An extensible sat-solver. In: SPRINGER. *Theory and applications of satisfiability testing*. 2004. p. 502–518.

[98] DATTORRO, J. The implementation of recursive digital filters for high-fidelity audio. *J Audio Eng Soc*, v. 36, n. 11, p. 851–878, 1988.

[99] CLARKE, E.; KROENING, D.; LERDA, F. A tool for checking ansi-c programs. In: ____. *Tools and Algorithms for the Construction and Analysis of Systems: 10th International Conference, TACAS 2004, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2004, Barcelona, Spain, March 29 - April 2, 2004. Proceedings*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004. p. 168–176.

[100] DUGGIRALA, P. S.; VISWANATHAN, M. Analyzing real time linear control systems using software verification. In: *2015 IEEE Real-Time Systems Symposium*. 2015. p. 216–226. ISSN 1052-8725.

[101] RUBILLO, C.; MARZOCCA, P.; BOLLT, E. M. Active aeroelastic control of lifting surfaces via jet reaction limiter control. *I. J. Bifurcation and Chaos*, v. 16, n. 9, p. 2559–2574, 2006.

[102] CHAVES, L.; BESSA I., C. L.; KROENING, D. Dsvalidator: An automated counterexample reproducibility tool for digital systems. In: *21st ACM International Conference on Hybrid Systems: Computation and Control (HSCC)*. 2018. p. 1–6.

[103] BAUER, P.; LECLERC, L.-J. A computer-aided test for the absence of limit cycles in fixed-point digital filters. *IEEE Trans. Signal Processing*, v. 39, n. 11, p. 2400–2410, Nov 1991. ISSN 1053-587X.

[104] PREMARATNE, K.; KULASEKERE, E.; BAUER, P.; LECLERC, L.-J. An exhaustive search algorithm for checking limit cycle behavior of digital filters. *IEEE Trans. Signal Processing*, v. 44, n. 10, p. 2405–2412, Oct 1996. ISSN 1053-587X.

[105] ABATE, A.; BESSA, I.; CATTARUZZA, D.; CORDEIRO, L. C.; DAVID, C.; KESSELI, P.; KROENING, D.; POLGREEN, E. Automated formal synthesis of digital controllers for state-space physical plants. In: *Computer Aided Verification - 29th International Conference, CAV 2017, Heidelberg, Germany, July 24-28, 2017, Proceedings, Part I*. 2017. (Lecture Notes in Computer Science, v. 10426), p. 462–482.

[106] ROCHA, H.; ISMAIL, H.; CORDEIRO, L. C.; BARRETO, R. S. Model checking embedded C software using k-induction and invariants. In: *2015 Brazilian Symposium on Computing Systems Engineering, SBESC 2015, Foz do Iguacu, Brazil, November 3-6, 2015*. 2015. p. 90–95.

[107] GADELHA, M. Y. R.; ISMAIL, H. I.; CORDEIRO, L. C. Handling loops in bounded model checking of C programs via k-induction. *International Journal on Software Tools for Technology Transfer (STTT)*, v. 19, n. 1, p. 97–114, 2017.

[108] ALVES, E. H. da S.; CORDEIRO, L. C.; FILHO, E. B. de L. Fault localization in multi-threaded C programs using bounded model checking. In: *2015 Brazilian Symposium on Computing Systems Engineering, SBESC 2015, Foz do Iguacu, Brazil, November 3-6, 2015*. [s.n.], 2015. p. 96–101. Disponível em: <https://doi.org/10.1109/SBESC.2015.25>.

[109] ALVES, E. H. da S.; CORDEIRO, L. C.; FILHO, E. B. de L. A method to localize faults in concurrent C programs. *Journal of Systems and Software*, v. 132, p. 336–352, 2017.

[110] ABATE, A.; BESSA, I.; CATTARUZZA, D.; CHAVES, L.; CORDEIRO, L. C.; DAVID, C.; KESSELI, P.; KROENING, D.; POLGREEN, E. DSSynth: An Automated Digital Controller Synthesis Tool for Physical Plants. In: *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 2017.

[111] ISIDORI, A. *Nonlinear Control Systems*. 3rd. ed. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 1995. ISBN 3540199160.

[112] CORDEIRO, L. C.; FISCHER, B. Verifying multi-threaded software using smt-based context-bounded model checking. In: *Proceedings of the 33rd International Conference on*

Software Engineering, ICSE 2011, Waikiki, Honolulu , HI, USA, May 21-28, 2011. 2011. p. 331–340.

[113] CORDEIRO, L. C.; MORSE, J.; NICOLE, D. A.; FISCHER, B. Context-bounded model checking with ESBMC 1.17 - (competition contribution). In: *Tools and Algorithms for the Construction and Analysis of Systems - 18th International Conference, TACAS 2012, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2012, Tallinn, Estonia, March 24 - April 1, 2012. Proceedings*. 2012. (Lecture Notes in Computer Science, v. 7214), p. 534–537.

[114] MORSE, J.; CORDEIRO, L. C.; NICOLE, D. A.; FISCHER, B. Handling unbounded loops with ESBMC 1.20 - (competition contribution). In: *Tools and Algorithms for the Construction and Analysis of Systems - 19th International Conference, TACAS 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings*. 2013. (Lecture Notes in Computer Science, v. 7795), p. 619–622.

[115] MORSE, J.; RAMALHO, M.; CORDEIRO, L.; NICOLE, D.; FISCHER, B. ESBMC 1.22 - (Competition Contribution). In: *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. 2014. p. 405–407.

[116] WINTERSTEIGER, C. *Compiling GOTO-programs*. `http://www.cprover.org/goto-cc`. Accessed: 08-27-2017.

[117] ROCHA, H.; BARRETO, R. S.; CORDEIRO, L. C.; NETO, A. D. Understanding programming bugs in ANSI-C software using bounded model checking counter-examples. In: *Integrated Formal Methods - 9th International Conference, IFM 2012, Pisa, Italy, June 18-21, 2012. Proceedings*. 2012. (Lecture Notes in Computer Science, v. 7321), p. 128–142.

Appendix A

Bounded Model Checking Tools

A.1 Efficient SMT-Based Context-Bounded Model Checker (ESBMC)

The ESBMC [66, 112–115] is an SMT-based context-bounded model checker for C/C++ programs. In ESBMC [66, 115], the associated problem is formulated by constructing the logical formula:

$$\psi_k = I(S_0) \wedge \bigvee_{i=0}^{k} \bigwedge_{j=0}^{i-1} \gamma(s_j, s_{j+1}) \wedge \overline{\phi(s_i)} \tag{A.1}$$

where ϕ is a property (*e.g.*, overflow and limit cycle), S_0 is a set of initial states of M, and $\gamma(s_j, s_{j+1})$ is the transition relation of M between time steps j and $j+1$. Hence, $I(S_0) \wedge \bigwedge_{j=0}^{i-1} \gamma(s_j, s_{j+1})$ represents the executions of a transition system M of length i. The above verification condition (VC), ψ, can be satisfied if, and only if, for some $i \leq k$ there exists a reachable state at time step i in which ϕ is violated. If the logical formula (A.1) is satisfiable (i.e., returns *true*), then the SMT solver provides a satisfying assignment, from which the values of the digital controller's variables can be extracted to construct a counterexample.

It is important to note that this approach can be used only to find property violations up to a bound k. In order to prove properties, we need to compute the completeness threshold (CT), which can be smaller than or equal to the maximum number of loop-iterations occurring in a program. However, computing CT to stop the BMC procedure and to conclude that no counterexample can be found is as hard as model checking. Moreover, complex programs involve large data-paths and complex expressions. Consequently, even if we knew CT, the resulting formulae would quickly become too hard to solve and require too much memory to build. In practice, we can thus only ensure that the property holds in M up to a given bound k. In ESBMC, the focus is on embedded software verification, since it has characteristics that make it attractive for BMC, *e.g.*, dynamic memory allocations and recursion are highly discouraged, which make the limitations of bounded model checking less stringent.

A.2 C Bounded Model Checker (CBMC)

CBMC (*C Bounded Model Checker*) [99] implements BMC for ANSI-C/C++ programs using SAT solvers. It can process C/C++ code using the goto-cc [116] tool, which compiles the C/C++ code into equivalent GOTO-programs using a gcc-compliant style. Alternatively, CBMC uses its own internal parser based on Flex/Bison, in order to process C/C++ files and build an abstract syntax tree (AST). The typechecker annotates this AST with types and generates a symbol table. CBMC's IRep class then converts the annotated AST and the C/C++ GOTO-programs into an internal language-independent format used by the remaining phase of the front-end.

CBMC derives VCs using two recursive functions that compute *assumptions* or *constraints* (*i.e.*, variable assignments) and *properties* (*i.e.*, safety conditions and user-defined assertions). CBMC's VC generator (VCG) automatically generates safety conditions that check for arithmetic overflow and underflow, array bounds violations and null-pointer dereferences. Both functions accumulate control flow predicates to each program point and use that to guard both constraints and properties, so that they properly reflect a program's semantics.

Although CBMC implements several state-of-the-art techniques for propositional BMC, it still has the following limitations: (*i*) large data-paths involving complex expressions lead to large propositional formulae, (*ii*) high-level information is lost when verification conditions are converted into propositional logic, and (*iii*) encoding size increases with array sizes used in a program.

A.3 The BMC Counterexample

CBMC [99] and ESBMC [66] construct counterexamples whether a property violation is found. A counterexamples is a trace that shows that a given property does not hold in the model. Counterexamples allow the user: (i) to analyze a failure, (ii) to understand an error, and (iii) to correct either the respective specification or model, in this case, from the property and the program that has been analyzed [117].

Definition A.1 ***(Counterexample)*** *A counterexample for a reachability property ϕ is a sequence of states $s_0, s_1, \ldots, s_k$ with $s_0 \in S_0$ (initial state), $s_k \in S$ (bad state) and $\gamma(s_i, s_{i+1}) \in R$ for $0 \leq i < k$, that refutes ϕ.*

Appendix B

DSVerifier Toolbox

A MATLAB toolbox was developed, with the goal of checking occurrences of design errors typically found in fixed-point digital systems, considering finite word-length effects. In particular, the present toolbox works as a front-end to a recently introduced verification tool, known as Digital-System Verifier (DSVerifier), and checks overflow, limit cycle, quantization, stability, and minimum phase errors in digital systems represented by transfer-function and state-space equations. It provides a command-line version with simplified access to specific functionality and a graphical-user interface, which was developed as a MATLAB application. The resulting toolbox enables application of verification to real-world systems by control engineers.

B.1 DSVerifier Toolbox Architecture

The verification methodology for the toolbox is based on DSVerifier and can be split into four main steps, as illustrated in Fig. B.1.

B.2 DSVerifier Toolbox Features

DSVerifier's features can be described as follows:

a) **Digital-system representation:** DSVerifier handles digital systems represented by transfer-function and state-space representations (cf. step 1 of Fig. B.1).

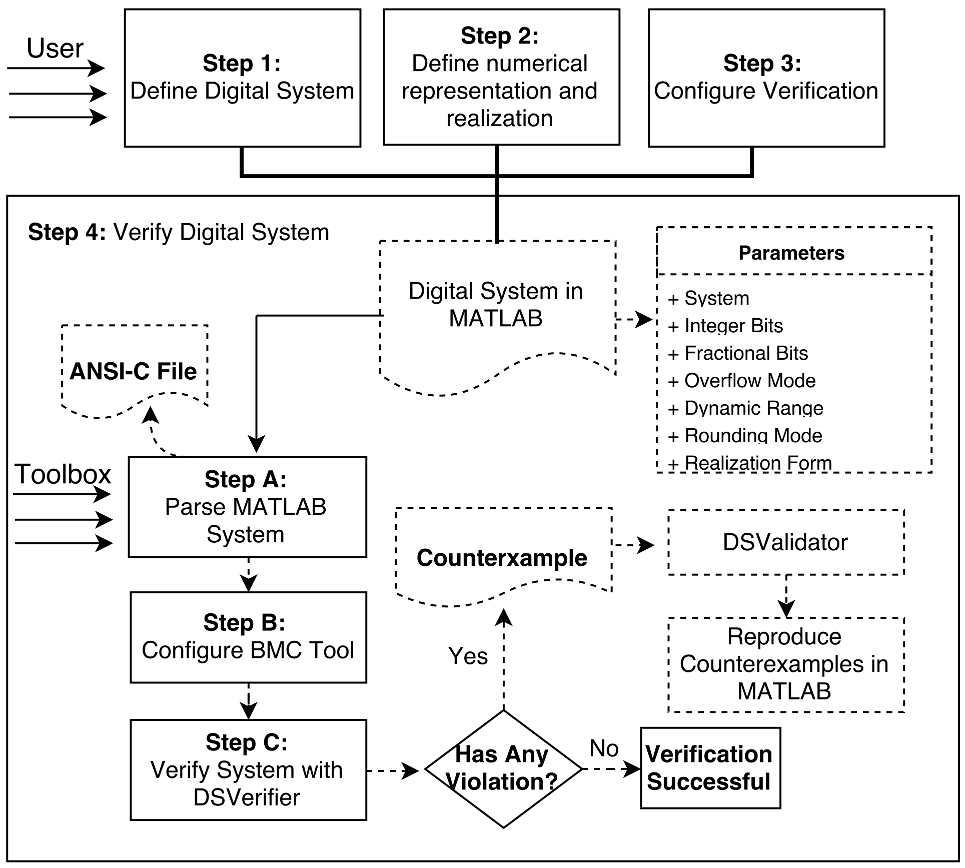

User
Step 1: Define Digital System
Step 2: Define numerical representation and realization
Step 3: Configure Verification
Step 4: Verify Digital System
Digital System in MATLAB
Parameters
+ System
+ Integer Bits
+ Fractional Bits
+ Overflow Mode
+ Dynamic Range
+ Rounding Mode
+ Realization Form
ANSI-C File
Toolbox
Step A: Parse MATLAB System
Step B: Configure BMC Tool
Step C: Verify System with DSVerifier
Counterexample
DSValidator
Reproduce Counterexamples in MATLAB
Yes
Has Any Violation?
No
Verification Successful

Figure B.1: DSVerifier's verification methodology.

b) **Realization:** DSVerifier performs the verification of direct-form I (DFI), direct-form II (DFII) and transposed direct-form II (TDFII), and also delta direct-form I (DDFI), delta direct-form II (DDFII) and delta transposed direct-form II (TDDFII) (cf. step 2 of Fig. B.1).

c) **Open-loop systems:** DSVerifier verifies, for transfer-function representation, stability, overflow, minimum phase, limit-cycle and quantization error, while in state-space representation, it verifies stability, quantization error, observability and controllability properties (cf. step 3 of Fig. B.1).

d) **Closed-loop systems:** DSVerifier verifies stability, limit-cycle and quantization error in transfer-function representation, while for state-space systems, all properties mentioned for open-loop systems are checked, via state feedback matrix (cf. step 3 of Fig. B.1).

e) **BMC tools:** DSVerifier handles the verification for digital-systems using CBMC or ESBMC as back-end, to perform the symbolic verification (cf. step 3 of Fig. B.1).

Table B.1: DSVerifier's commands and parameters used during verification procedures.

Verification Command	system	intbits	fracbits	max	min	bound	cmode	error
verifyStability	x	x	x	x	x			
verifyOverflow	x	x	x	x	x	x		
verifyError	x	x	x	x	x	x		x
verifyMinimumPhase	x	x	x	x	x			
verifyLimitCycle	x	x	x	x	x	x		
verifyClosedStability	x	x	x	x	x		x	
verifyClosedQuantizationError	x	x	x	x	x	x	x	x
verifyClosedLimitCycle	x	x	x	x	x	x	x	
verifyStateSpaceStabiltiy	x	x	x					
verifyStateSpaceControllability	x	x	x					
verifyStateSpaceObservability	x	x	x					
verifyStateSpaceQuantizationError	x	x	x			x		x

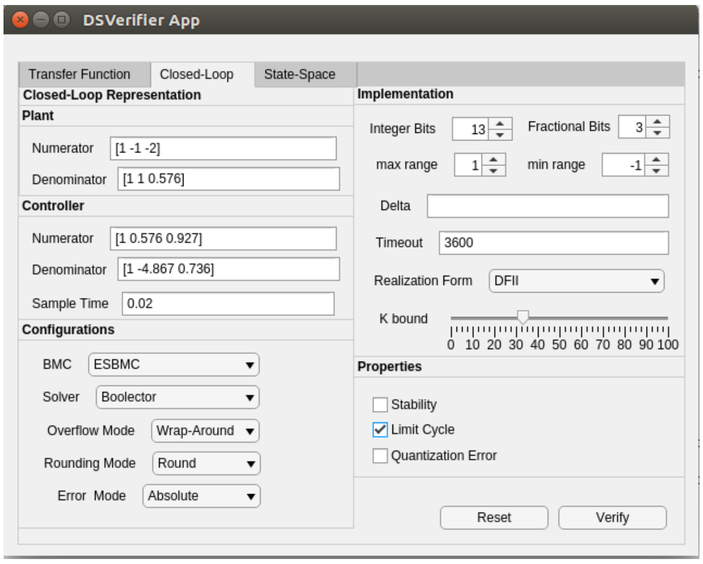
B.3 DSVerifier Toolbox Usage

In order to explain the DSVerifier's workflow, the following second-order controller represented by a transfer-function $H(z) = \frac{B(z)}{A(Z)}$ for an A/C motor plant is used

$$H(z) = \frac{z^3 - 2.819z^2 + 2.6370z - 0.8187}{z^3 - 1.97z^2 + 1.033z - 0.06068}. \tag{B.1}$$

B.3.1 Command Line Version

Users must provide a digital system described as a MATLAB system using a `tf` (for transfer-function) or an `ss` (for state-space) command (cf. step 1 of Fig. B.1). DSVerifier is then invoked to check the digital system representation and the desired property (cf. steps 2 and 3 of Fig. B.1). Table B.1 summarizes the DSVerifier's commands that perform the proposed automatic verification and the required parameters for each property (cf. step 4 of Fig. B.1). In Table B.1, *system* represents the digital system in transfer-function or state-space format, *intbits* is the integer part, *fracbits* is the fractional part, *max* and *min* are the maximum and minimum dynamic range, respectively, *bound* is the *k* bound to be employed during verification, *cmode* is the connection mode, for closed-loop systems in transfer-function (series or feedback), and *error* is the maximum possible value in the quantization error check. Additionally, optional parameters can be included, such as overflow mode, rounding mode, BMC tool, solver, quan-



Closed-Loop Representation

Plant

Numerator [1 -1 -2]

Denominator [1 1 0.576]

Controller

Numerator [1 0.576 0.927]

Denominator [1 -4.867 0.736]

Sample Time 0.02

Configurations

BMC ESBMC

Solver Boolector

Overflow Mode Wrap-Around

Rounding Mode Round

Error Mode Absolute

Implementation

Integer Bits 13 Fractional Bits 3

max range 1 min range -1

Delta

Timeout 3600

Realization Form DFII

K bound 0 10 20 30 40 50 60 70 80 90 100

Properties

☐ Stability

☑ Limit Cycle

☐ Quantization Error

Figure B.2: GUI application for transfer-function verification, in closed-loop format.

tization error mode and delta coefficient (for delta realization).[1] All available functions w.r.t. DSVerifier have been exhaustively tested and experimental results are available online.[2]

B.3.2 MATLAB Application Version

A graphical user interface application was developed (Fig. B.2), in order to favor digital-system verification in MATLAB, besides improving usability and, consequently, attracting more digital-system engineers. Users can provide all required parameters for digital-system verification: specification, target implementation and properties to be checked.

B.3.3 Illustrative Example

To illustrate the DSVerifier's usage, Fig. B.3 shows the stability verification for the digital system specified in Eq. B.1, where "num" and "den" represent the numerator $A(z)$ and denominator $B(z)$, resp., using a dynamic range $[-1,1]$ and a fixed-point format $<2,13>$, *i.e.*, 2 bits for the integer part and 13 for the fractional one.

If the fixed-point format is changed to $<12,3>$, for the same system described in Eq. B.1, the verification indicates a failure, *i.e.*, a digital system is unstable, as shown in

[1] All functions implemented in DSVerifier are detailed in the Toolbox's Documentation.

[2] http://www.dsverifier.org/benchmarks

```
>> num = [1.0000 −2.8190 2.6370 −0.8187];
>> den = [1.0000 −1.9700 1.0330 −0.0607];
>> system = tf(num,den,0.001);
>> verifyStability(system,2,13,1,−1);
>> VERIFICATION SUCCESSFUL
```

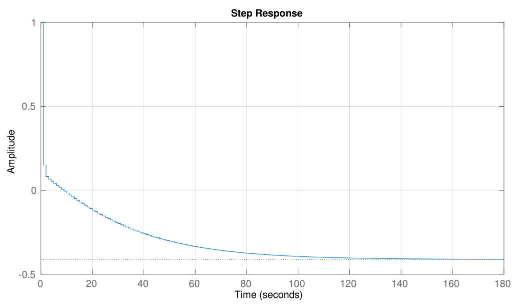
Figure B.3: Verifying stability for Eq. B.1 in MATLAB, with a fixed-point format $< 2, 13 >$.

Fig. B.4, which indicates that DSVerifier can correctly verify digital systems with different implementations.

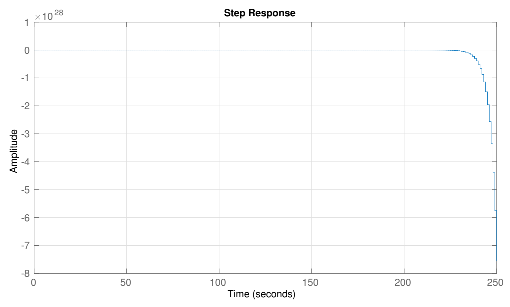
```
>> verifyStability(system,12,3,1,−1);
>> VERIFICATION FAILED
```

Figure B.4: Verifying stability for Eq. B.1 in MATLAB, with a fixed-point format $< 12, 3 >$.

After verifying that the adopted digital system is unstable with format $< 12, 3 >$, the respective failed verification result can be confirmed by reproducing the counterexample generated by DSVerifier.



(a) Successful verification using the fixed-point format $<$ $2, 13 >$.



(b) Failed verification using the fixed-point format $<$ $12, 3 >$.

Figure B.5: Step response for Eq. B.1.

The stability for the digital systems described above could be indeed observed through the associated step response for both cases, as shown in Fig. B.5. In subfigure B.5(a), the step response shows that the digital system is stable, while in B.5(b) it is unstable.

Appendix C

DSValidator Toolbox

An automated counterexample reproducibility tool based on MATLAB is presented, called DSVerifier, with the goal of reproducing counterexamples that refute specific properties related to digital systems. We exploit counterexamples generated by the Digital System Verifier (DSVerifier), which is a model checking tool based on satisfiability modulo theories for digital systems. DSVerifier reproduces the execution of a digital system, relating its input with the counterexample, in order to establish trust in a verification result.The resulting toolbox leverages the potential of combining different verification tools for validating digital systems via an exchangeable counterexample format.

C.1 Proposed Counterexample Format

DSVerifier exploits counterexamples provided by verifiers; if there is a property violation, then the verifier provides a counterexample, which contains inputs and initial states that lead the digital system to a failure state. Fig. C.1 shows an example of the present counterexample format related to an overflow LCO violation for the digital system represented by Eq. (C.1):

$$H(z) = \frac{2002 - 4000z^{-1} + 1998z^{-2}}{1 - z^{-2}}. \tag{C.1}$$

The proposed counterexample format shown in Fig. C.1 describes the violated property (represented by a *string*), transfer function numerator and denominator (represented by *fixed-point numbers*), bound (represented by an *integer*), sample time (represented by a *fixed-point number*), implementation aspects (integer and fractional bits represented by an *integer*), realiza-

```
Property = LIMIT_CYCLE
Numerator  = { 2002,  −4000,  1998 }
Denominator  = { 1,  0,  −1 }
X_Size = 10
Sample_Time = 0.001
Implementation = <13,3>
Numerator (fixed−point) = { 2002, −4000, 1998 }
Denominator (fixed−point) = { 1, 0, −1 }
Realization = DFI
Dynamical_Range = { −1, 1 }
Initial_States = { −0.875, 0, −1 }
Inputs = { 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5}
Outputs = { 0, −1, 0, −1, 0, −1, 0, −1, 0, −1}
```

Figure C.1: Proposed counterexample format example.

tion form (represented by a *string*), dynamical range (represented by an *integer*), initial states, inputs, and outputs (which are represented by *fixed-point numbers*). In particular, the counterexample provides the needed data to reproduce a given property violation via simulation in MATLAB.

C.2 Automated Counterexample Validation

There are five steps to automatically perform the automated counterexample validation in DSVerifier. In *step* (1), DSVerifier obtains the counterexample and then uses a shell script

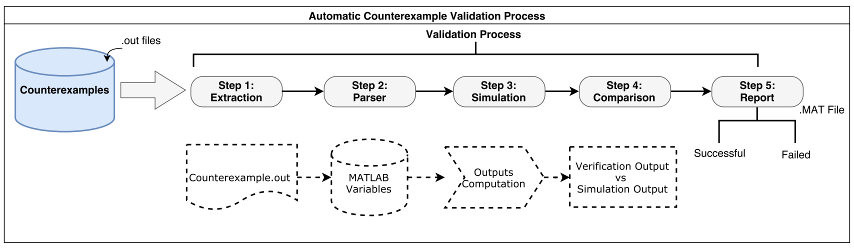


Figure C.2: Automatic counterexample validation process.

to extract the data related to the digital system, *i.e.*, property, transfer function numerator and denominator, fixed-point representation, *k*-bound, sample time, implementation aspects, realization form, dynamical range, initial states, inputs, and outputs. In *step* (2), DSVerifier converts all counterexample attributes into variables that can be manipulated in MATLAB. In *step* (3), DSVerifier simulates the counterexample (violation) for the failed property, which is derived from the counterexample by providing concrete, lower-level details needed to simulate the digital system in MATLAB. In this specific *step*, all FWL effects are applied to the digital system, and computations to perform the outputs are produced, according to the realization form and property, as previously mentioned. In *step* (4), DSVerifier compares the result between the output provided by the verifier and that simulated by MATLAB. Finally, in *step* (5), DSVerifier stores the extracted counterexample in a `.MAT` file and then reports its reproducibility.

C.3 DSValidator Features

DSVerifier's features can be described as follows:[1]

- **Macro Functions:** functions to reproduce the validation steps (*e.g.*, parsing, simulation, comparison, and report).
- **Validation Functions:** check and validate a violated property (*e.g.*, overflow, limit-cycle, stability, and minimum-phase).
- **Realizations:** reproduce realizations forms to validate overflow and limit-cycle (for direct and delta forms).
- **Numerical Functions:** perform the quantization process; select rounding mode and overflow mode (wrap-around and saturate); fixed-point operations (*e.g.*, sum, subtraction, multiplication, division); and delta operator.
- **Graphic Functions:** plot the graphical representation of overflow to show each output exceeding the supported word-length limits; limit-cycle to represent the system's output oscillations; and poles/zeros to show stability and minimum-phase with (or without) FWL effects inside a unitary circle.

C.4 DSValidator Result

The DSVerifier result is structured with counterexample data composed by attributes and classes as shown in Fig. C.3. The attributes are defined in the `.MAT` file with the following structure: *counterexample* that represents the counterexample identification; *digital system* that

[1]Functions implemented in DSVerifier are described in the Toolbox Documentation.

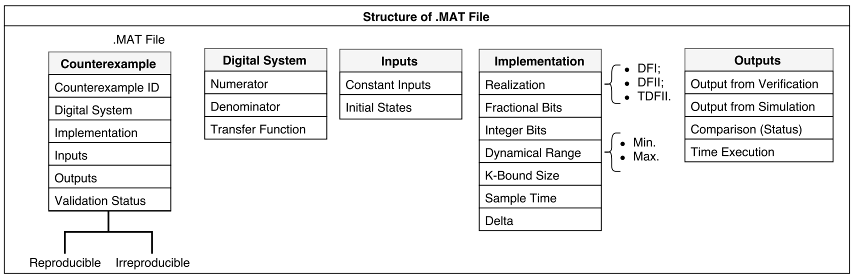


Figure C.3: Structure of the `.MAT` file for representing counterexamples.

represents the numerator, denominator, and transfer function representation; *inputs* that represent the input vector and initial states; *implementation* that represents the integer and fractional bits, dynamical ranges, delta operator, sample time, bound, and realization form; *outputs* report the verification and simulation results, execution time in MATLAB, and comparison status, where it reports whether the counterexample is reproducible or not. Importantly, all execution times are actually CPU times, *i.e.*, only the elapsed time periods spent in the allocated CPUs, which is measured with the `times` system call (POSIX system).

C.5 DSValidator Usage

DSVerifier can be called via command line in MATLAB as:

```
validation(path, property, ovmode, rmode, filename)
```

where `path` is the directory with all counterexamples; `property` is defined as: **m** for minimum phase; **s** for stability; **o** for overflow; and **lc** for limit cycle; `ovmode` represents the overflow mode: **wrap** for wrap-around mode (default) and **saturate** for saturation mode; `rmode` represents the rounding mode, which can be **round** (default) and **floor**; `filename` represents the `.MAT` filename, which is generated after the validation process; by default, the `.MAT` file is named as `digital_system`.

```
Running Automatic Validation...
Counterexamples (CE) Validation Report...
CE 1 time: 0.081929 status: reproducible
CE 2 time: 0.013996 status: reproducible
CE 3 time: 0.009488 status: reproducible
General Report:
Total Counterexamples Reproducible: 3
Total Counterexamples Irreproducible: 0
Total Counterexamples: 3
Total Execution Time: 0.10541
```

Figure C.4: Counterexample reproducibility report.

After executing the `validation` command, DSVerifier prints statistics about the counterexamples validation. Fig. C.4 shows a report about the digital system represented in Eq. (C.1) for realizations DFI, DFII, and TDFII.

Appendix D

Publications

D.1 Related to the Research

Chaves, L., Ismail, H., Monteiro, F., Bessa, I., Cordeiro, L., Lima Filho, E. DSVerifier v2.0: A BMC Tool to Verify Fragility in Digital Systems with Uncertainties. In Science of Computer Programming, pp. 1-27, Elsevier, 2018. **(Submitted)**

Chaves, L., Bessa, I., Ismail, H., Frutuoso, A., Cordeiro, L., Lima Filho, E. DSVerifier-Aided Verification Applied to Attitude Control Software in Unmanned Aerial Vehicles. In IEEE Transactions on Reliability, pp. 1-15, 2018. **(In Rebuttal)**

Chaves, L., Bessa, I., Cordeiro, L., Kroening, D. DSValidator: An automated counterexample reproducibility tool for digital systems. In 21st ACM International Conference on Hybrid Systems: Computation and Control, 2018. **(Published)**

Chaves, L., Bessa, I., Cordeiro, L., Kroening, D., Lima Filho, E. Verifying Digital Systems with MATLAB. In 26th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA), pp. 388-391, 2017. **(Published)**

D.2 Contribution in Another Research

Abate, A., Bessa, I., Cattaruzza, D., **Chaves, L.**, Cordeiro, L., David, C., Kesseli, P., Kroening, D., Polgreen, E. DSSynth: An Automated Digital Controller Synthesis Tool for Physical Plants. In 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE), pp. 1-6, 2017. **(Published)**