Verifying Fragility in Digital Systems with Uncertainties using DSVerifier $v2.0$

Lennon C. Chaves[1,2], Hussama I. Ismail[1], Iury V. Bessa[1], Lucas C. Cordeiro[2], and Eddie B. de Lima Filho[3]

[1] *Faculty of Technology, Federal University of Amazonas, Brazil*
[2] *School of Computer Science, University of Manchester, United Kingdom*
[3] *TPV Technology, Brazil*

Abstract

Control-system robustness verification with respect to implementation aspects lacks automated verification approaches for checking stability and performance of uncertain control systems, when considering finite word-length (FWL) effects. Here we describe and evaluate novel verification procedures for digital systems with uncertainties, based on software model checking and satisfiability modulo theories, named as DSVerifier $v2.0$, which is able to check robust stability of closed-loop control systems with respect to FWL effects. In particular, we describe our verification algorithms to check for limit-cycle oscillations (LCOs), output quantization error, and robust non-fragile stability on common closed-loop associations of digital control systems (*i.e.*, series and feedback). DSVerifier $v2.0$ model checks new properties of closed-loop systems (*e.g.*, LCO), including stability and output quantization error for uncertain plant models, and considers unknown parameters and FWL effects. Experimental results over a large set of benchmarks show that 35%, 34%, and 41% of success can be reached for stability, LCO, and output quantization error verification procedures, respectively, for a set of 396 closed-loop control system implementations and realizations.

Keywords: fixed-point digital controllers, formal methods, bounded model checking, system reliability, uncertainty

1. Introduction

The current control theory provides construction of reliable systems, by offering mathematical guarantees about stability and desired performance of closed-loop systems, where plant states or outputs are fed back and compared to a reference signal, which guides control objectives [1]. In such a context, robustness is a typical control system desirable property that denotes its capability to ensure stability and acceptable performance, with respect to uncertainties, *i.e.*, unknown parameters and exogenous perturbations[2].

Feedback control systems usually seek to guarantee robustness for closed-loop architectures; however, digital-controller implementations through electronic systems, such as microcontrollers, microprocessors, and specific circuitry, commonly face unavoidable variations and disturbances [3] and might be subject to problems caused by architecture restrictions, such as finite word-length (FWL) effects (*i.e.*, round-offs and truncation), which have the potential to make them fragile. Regarding that, Keel and Bhattacharyya [4] showed that even robust and optimal controllers might be fragile and therefore could not hold stability, due to FWL effects. Fragility is a control system's sensitivity to extremely small perturbations that are caused by imprecisions in implementations, *e.g.*, round-offs in digital controllers' coefficients due to FWL formats [4]. Thus, a controller that is designed for a specific purpose is considered fragile when it fails to achieve that, due to implementation issues. Finally, non-fragile control [5] is the sub-area of control theory dedicated to study techniques for designing non-fragile controllers.

Robust control, in turn, deals with disturbance signals and dynamic perturbations, the latter being related to mismatches between mathematical model and real system. For instance, the work developed by Zhao *et al.* [3] investigated stability regarding continuous-time uncertain systems and provided precise mathematical modeling for a specific class of them, with a novel type of Lyapunov function. In addition, Sakthivel *et al.* [6] tackled time-delay systems subject to actuator faults and disturbances and developed a design approach, which includes sufficient conditions under uncertainties, modeled through an optimization problem. Although robust control is widely investigated in the literature [2–4, 6], its practical applications, while taking into account target implementation architectures and respective restrictions, is not commonly considered and constitutes a new research branch.

Indeed, platform restrictions and uncertainties, if not properly tackled, can cumulate and thus lead to incorrect behavior and system instability. As a consequence, the verification and control theory communities lack a formal framework able to automatically perform that, which could be integrated into design phases and even guide them, with the goal of creating correct-by-construction systems.

The fragility problem is hardly predicted in the control design step or detected during tests and simulations, which can cause several losses during operation. Non-fragile control techniques [5, 7, 8] and specialized controller realizations [1, 9] are usually employed to design safe controllers and implementations, with respect to FWL effects. Nonetheless, there are only a few tools to indicate fragility and detect violation of control specifications (*e.g.*, stability), when considering implementation issues [10–13]. In fact, those formal verification tools consider FWL effects to ensure correctness of digital controller designs; however, a direct comparison with them is difficult, due to some differences and difficulties, as further discussed in Section 2.

Some model checking tools are able to verify systems represented by timed automata, *e.g.*, UPPAAL [14]; however, they consider mainly high-level properties during system verification. Only a few studies employ model checking tools for low-level specification of controllers (*e.g.*, stability and transient behavior). As an example, SAHVY [13] simulates system execution, by solving ordinary differential equations (represented by Taylor models) for a range of initial states, and performs bounded model checking (BMC) based on satisfiability modulo theories (SMT) [15], in order to verify safety properties expressed by computational tree logic formulae [16]. Nonetheless, SAHVY does not consider FWL effects in digital control system implementations and important design aspects, such as fragility and robustness. In addition, Ismail *et al.* proposed the Digital-System Verifier (DSVerifier $v1.0$) [11] to find FWL problems in digital controllers and filters (*e.g.*, overflows, limit-cycle oscillations, and stability loss); however, it does not consider the consequences associated to closed-loop systems. Regarding the latter, they are typically represented as hybrid systems, *i.e.*, controllers are digital and plants are physical continuous systems, whose interaction must be considered, under the influence of FWL effects.

These prior studies are the main source of inspiration for the current work, which tackles both fragility verification and uncertain models, in such a way that realization aspects are considered along with variations in plant models. As a consequence, verification and design procedures can now rely on a broad and extensible tool, which is able to scale on closed-loop control systems. Indeed, while previous studies either consider mathematical conditions for operation under uncertainties [3, 6] or provide verification regarding implementation aspects [11, 13, 14], the proposed approach, which was implemented in DSVerifier $v2.0$, provides a formal framework that checks both in conjunct, while evaluating merit figures specific to digital systems, such as stability, limit-cycle oscillations, and output quantization error.

Given the current knowledge in control system verification, DSVerifier $v2.0$[1] Nonetheless, note that MATLAB [17] has two toolboxes for similar problems: Robust Control Toolbox (RCT) and Fixed-Point Designer (FPD). The first allows tuning and analysis of impacts regarding plant model uncertainties on control systems (no implementation aspects), while our work allows verification and validation of closed-loop systems, when considering FWL effects. Additionally, the second and our work do not overlap, since the former is an analysis and design tool, while the latter is a verification one. Indeed, FPD does not support closed-loop system verification and uncertain hybrid system verification. Furthermore, LCO verification in DSVerifier $v2.0$ is more comprehensive than that of FPD, since it can verify any system represented by a transfer-function and it is also able to find LCO for any constant input, while FPD can only indicate zero-input LCO for second-order systems [18]. Another important contribution of this work is its novel approach for verifying controller fragility. Traditionally, the control-systems community considers the latter as uncertainties in a controller model, by representing it as an inexact model. By contrast, our approach allows the computation of FWL effects in digital controllers, by obtaining an exact model of a digital controller implementation and a plant model with nondeterministic coefficients related to uncertainties.

Finally, DSVerifier can easily scale on control-system verification, given that it is able to analyze any structure represented by transfer functions (TFs) of single-input single-output (SISO) systems. Indeed, architecture restrictions and uncertainties are both considered as effects on TF coefficients of digital controllers and plants, respectively. In addition, DSVerifier is based on bounded model checking [16], which means that a maximum depth for system unrolling must be defined and properties are checked until that. As a consequence, system complexity directly relates to memory and processing demands, which may result in resource exhaustion. In summary, completeness could be achieved by computing a completeness threshold [19], which can be smaller than or equal to the maximum number of loop-iterations occurring in the control software; however, that may result in inability to provide property checking, due to high

[1]Our tool is available at: `http://dsverifier.org/` is the only verification tool that checks robust non-fragile stability and limit-cycle oscilations (LCOs), in closed-loop systems, and it can be employed to validate implementations of digital controllers designed through different techniques, including the non-fragile one.

resource demand, which inherently leads to a trade-off between system unrolling and state-space search exploration.

1.1. *Improvements since DSVerifier v1.0*

We extended the previous work of Ismail *et al.* [11] (*i.e.*, DSVerifier $v1.0$) to enable closed-loop system verification in uncertain systems. In summary, the improvements since DSVerifier $v1.0$[2] are:

- **Closed-loop System Verification -** DSVerifier $v2.0$ checks stability of closed-loop systems, under FWL implementation effects in digital controllers. It considers both plant and controller transfer function models, while plant models can also contain uncertainties.
- **Stability and LCO -** DSVerifier $v2.0$ checks stability and occurrence of LCO in closed-loop systems, by using two loop configurations: series and feedback. Additionally, its LCO verification is split into two categories: zero input LCO (previously supported) and LCO verification for non-deterministic inputs and states.
- **Output Quantization Error -** DSVerifier $v2.0$ computes the output of a closed-loop control system, considers round-off and FWL effects, and compares it with an ideal response (*i.e.*, without FWL effects), in order to check whether the output error is inside tolerable bounds.
- **Support for CBMC -** DSVerifier $v2.0$ now supports two efficient model-checking tools as back-end modules: ESBMC [16] (previously supported) and CBMC [20].
- **Support for New SAT/SMT Solvers -** DSVerifier $v2.0$ now supports Yices [21], MathSAT [22], CVC4 [23] by means of ESBMC, and MiniSat [24] by means of CBMC, in addition to Boolector [25] and Z3 [26] (both previously supported) by ESBMC.

Although some improvements over DSVerifier $v1.0$ might not sound as a major scientific contribution, they are particularly relevant, from a practical perspective. Specifically, they allow us to use off-the-shelf software model checkers to verify a large set of properties in a variety of digital control systems, by using SAT/SMT solvers. One may notice that SMT solvers apply different algebraic reduction rules and contextual simplification and they also use different SAT solvers as back-end (after bit-blasting), which implement different search heuristics. That means a particular SMT solver might perform better than others, for a specific verification problem. Providing such an alternative, *i.e.*, selection of different SAT/SMT solvers, can wide tool application regarding real-world problems and also contribute to the SAT/SMT community, with new problems and benchmarks. As a result, the nature of our contribution is more experimental rather than theoretical, since we add novel features to our formal verification tool (DSVerifier), describe details of its implementation, and provide an extensive experimental evaluation to demonstrate its feasibility for control engineers.

[2]DSVerifier $v1.0$ is available at: `http://dsverifier.org/downloads`, and the software code is available at `https://github.com/ssvlab/dsverifier`.

Regarding SMT back-ends, ESBMC provides a superior alternative to CBMC, which generates SMT formulae in a file and externally calls solvers, whereas ESBMC uses a solvers' native APIs. In [16], we explain the difference in performance, when using both approaches (*i.e.*, API and file interfaces). Additionally, the SMT back-end of CBMC is unable to support full ANSI-C, as recently reported in our previous work [27].

Lastly, different model checkers provide different verification strategies, counterexample format, and verification results. Although it is not a big deal to support a new software model checker, they usually consume a considerable implementation effort, in order to ensure that we exploit the full capabilities of each verifier. In particular, such a task should not be underestimated, since each verifier has its own characteristics and data format. As an example, a lot of effort has been devoted in the International Competition on Software Verification for establishing a standard format for counterexamples and invariants produced by different verifiers, in order to make it easy for a new verifier to use the existing benchmarking infrastructure [28].

1.2. Preliminaries

A transfer function representation of a digital system model $G(z)$ is expressed as a ratio of two polynomials in descending powers of z, *i.e.*, the numerator $B_G(z)$ and the denominator $A_G(z)$ in

$$G(z) := \frac{B_G(z)}{A_G(z)} := \frac{b_0 + b_1 z^{-1} + ... + b_{M_G} z^{-M_G}}{a_0 + a_1 z^{-1} + ... + a_{N_G} z^{-N_G}}, \tag{1}$$

where the subscript G in $B_G(z)$ and $A_G(z)$ indicate the system they describe (*i.e.*, G), and M_G and N_G represent numerator and denominator orders, respectively, related to system G.

A general vectorial notion is employed to represent a polynomial, *e.g.*, an L-th order polynomial $V_\lambda(z) := v_0 + v_1 z^{-1} + ... + v_{L_\lambda} z^{L_\lambda}$, related to system λ, is represented by vector $\vec{V}_\lambda = \begin{bmatrix} v_0 & v_1 & \dots & v_{L_\lambda} \end{bmatrix}$. Let $C(z)$ be a digital controller transfer function implemented with the fixed-point format $\langle I, F \rangle$ (*i.e., I* bits representing the integer part and F bits representing the fractional one), which could be signed and with a sign bit included in its integer part, such that $\vec{A}_C$ and $\vec{B}_C$ are their nominal denominator and numerator vectors and $\hat{\vec{A}}_C$ and $\hat{\vec{B}}_C$ are their correspondent in the FWL domain defined by $\langle I, F \rangle$. Note that when a specific format $\langle I, F \rangle$ is chosen, it is applied to all controller coefficients, irrespective of their values. As a consequence, in final implementations, some care must be taken regarding the chosen representation, in order to keep coefficient critical-information intact. Indeed, this work also intended to show FWL effects through different formats (with 8, 16, and 32 bits), as carried out for the experiments described in Section 5, and how they affect a digital-system's behavior, which is then anticipated by our verification framework. There is also a function $\mathcal{FWL}_{\langle I,F \rangle}[\cdot] : \mathbb{R}^n \to \mathbb{R}^n_{\langle I,F \rangle}$, where $\mathbb{R}^n_{\langle I,F \rangle}$ is the set of real numbers that are representable with fixed-point format $\langle I, F \rangle$, which computes the representation of a polynomial in the FWL domain, *i.e.*, $\hat{\vec{A}}_C := \mathcal{FWL}[\vec{A}_C]$ and $\hat{\vec{B}}_C := \mathcal{FWL}[\vec{B}_C]$.

Similarly to $C(z)$, let $P(z)$ be a nominal plant transfer function and $P_\delta(z)$ a plant transfer function with uncertainties, whose denominators and numerators vectors are $\vec{B}_P$, $\vec{B}_{P_\delta}$, $\vec{A}_P$, and $\vec{A}_{P_\delta}$, which are related as

$$\vec{A}_{P_\delta} = \vec{A}_P + \Delta\vec{p}_a\% \tag{2}$$

and

$$\vec{B}_{P_\delta} = \vec{B}_P + \Delta\vec{p}_b\%, \tag{3}$$

where $\Delta\vec{p}_a\%$ and $\Delta\vec{p}_b\%$ represent variations on numerator and denominator coefficients, due to model uncertainties. Thus, the set of all possible plant models, given parametric deviations (*i.e.*, plant family), is denoted by $\mathfrak{P}$.

1.3. Modelling FWL effects on digital-controller implementations

From $C(z)$ and $\langle I, F\rangle$, a model $\hat{C}(z) \triangleq \frac{\hat{B}_C(z)}{\hat{A}_C(z)}$ that represents only coefficient round-off is obtained, which is still a linear time-invariant system that may be represented by a transfer function. The latter is related to a difference equation implemented in hardware, through direct-form representations, which are directly supported by DSVerifier. For instance, one may consider the FWL second-order approximated transfer function

$$\hat{C}(z) = \frac{\hat{b}_0 + \hat{b}_1 z^{-1} + \hat{b}_2 z^{-2}}{1 + \hat{a}_1 z^{-1} + \hat{a}_2 z^{-2}}, \tag{4}$$

which can be represented by the difference equation

$$y(k) = -\hat{a}_1 y(n-1) - \hat{a}_2 y(n-2) + \hat{b}_0 x(k) + \hat{b}_1 x(n-1) + \hat{b}_2 x(n-2). \tag{5}$$

If the plant model is a continuous-time system, the discrete-time model in transfer-function or difference equation must be obtained via discretization. Among the methods available in the literature [9], we considered the sample-and-hold (ZOH) processes in complex systems [29], which models the exact effect of sampling and digital-to-analog conversion (DAC) interpolation over plants.

Assumption 1. *The sample-and-hold effects of the analog-to-digital conversion (ADC) module and the presence of ZOH for the DAC are synchronized, i.e., there is no delay between sampling a plant's output, at the ADC, and updating the DAC accordingly. Indeed, the DAC's interpolation is an ideal ZOH process.*

Assumption 2. *[9] Given a synchronized ZOH input and a sample-and-hold output on a plant, with a sample time T satisfying the Nyquist criterion, the discrete pulse transfer function $G(z,T)$ is an exact z-domain representation of $G(s)$, which can be computed through*

$$G(z,T) = (1 - z^{-1})\mathcal{Z}\left\{\mathcal{L}^{-1}\left\{\frac{G(s)}{s}\right\}_{t=kT}\right\}. \tag{6}$$

Software implementations of (5) usually contain basic arithmetic operations, *i.e.*, additions, subtractions, and multiplications, whose computation are also subject to FWL effects, such as round-off and overflow, which are already considered by DSVerifier. For the sake of simplicity, it is assumed that hardware numeric-representations are performed through two's complement and, if final operation results are representable, then overflow in intermediate results do not affect system outputs [30].

Assumption 3. *It is assumed that, in two's complement representations, the number of bits available for operations is equal to the number of bits for coefficients and only final operation results affect a system's output, i.e., if a final result is representable, then overflow in intermediate computations should not be flagged as violations [30].*

There are many ways to implement (5) depending on the desired realization structure for the target system. The commonly known structures are Direct Form I (DFI), Direct Form II (DFII), and Transposed Direct Form II (TDFII), where z^{-1} is defined as the backward-shift operator, that is, a unit delay. In order to illustrate this process, one may consider that (5) is implemented in Direct Form I, as illustrated in Fig. 1, whose algorithm implementation is shown in Fig 2. The latter can be implemented in the ANSI-C programming language (as shown in Fig. 3) and verified by the supported BMC tools present in DSVerifier. In a ANSI-C program, fixed-point variables are implemented as integer variables, with implicit power-of-2 scaling factors. As illustrated in Fig. 3, functions `fxp_add`, `fxp_mult`, and `fxp_sub` take two input arguments and return the respective addition, multiplication, and subtraction results, in `fxp32_t` format, which is internally defined in DSVerifier as `int32_t`. Besides, those blocks also include quantization effects and consider the fixed-point representation used by a given system, while function `fxp_quantize` provides quantization effects in each output, for a Direct Form I controller.

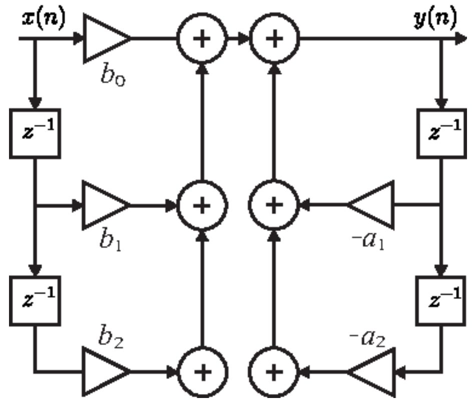


Figure 1: Direct form I realization of $\hat{C}(z)$.

Similarly, DSVerifier $v2.0$ also implements the filter functions in Direct Form II (DFII) and Transposed Direct Form II (TDFII), using C language and fixed-point library. Fixed-point functions as `fxp_add`, `fxp_mult`, `fxp_sub` and `fxp_quantize` are also implemented in both structures as illustrated in Fig. 3, according to previous works from the DSVerifier [11, 31, 32] which presented with details theses realization structures.

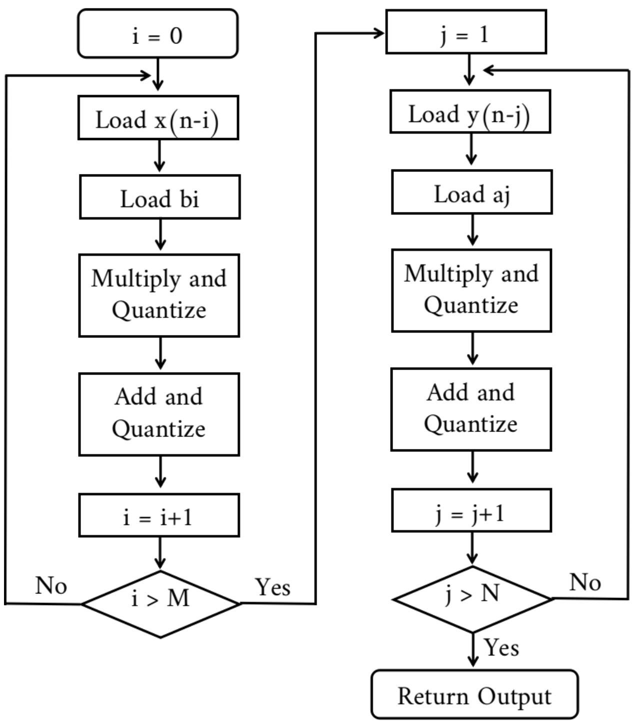


Figure 2: Flowchart for Direct Form I realizations.

Remark 1. *In the literature, it is shown that round-off effects may also be modeled as Gaussian noise in a system's output [33], i.e., measurement noise. Indeed, our stability result ensures internal stability, i.e., a system will be still stable for measurement noises, if the Jury's criteria are met.*

2. Related Work

Although formal methods provide applicability to check high-level specifications in all sorts of cyber-physical systems (CPS) [34], there is little effort regarding application of model checking for verifying different control goals, which are related to robust stability, robust performance, and non-fragility. In addition, relevant studies [12, 35–37] about performance and safety verification of closed-loop systems (as described below) propose verification methods based on symbolic execution of plant models.

Closed-Loop Symbolic Execution (CLSE) [36] performs a bounded-time symbolic execution of a plant's dynamics, which is represented by ordinary difference equations (ODEs) combined with concolic execution of controller software. Additionally, robustness analysis is also performed [36], where plant-state deviation is computed through sensor signals (*i.e.*, measurement noise). In contrast

```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--));
}
return fxp_quantize(sum);
}
```

Figure 3: C code fragment of a Direct Form I representation of $\hat{C}(z)$.

to Majumdar *et al.* [36], DSVerifier does not investigate robustness regarding measurement noises; however, it does perform robustness verification with respect to parametric uncertainties and investigate fragility, *i.e.*, robustness with respect to implementation issues. In particular, Zutshi *et al.* [35] employed numerical simulation of plant model and control software implementation, in order to build abstractions of state and input spaces, which then allows falsification of desired properties.

In the last decades, symbolic verification of closed-loop systems presented important advances; however, there are a few related model checking approaches for verifying closed-loop systems. One promising approach is Costan [12], which checks stability of closed-loop systems on embedded ANSI-C code controller. It compares the Simulink implementation [17] of a control system with code generated by MathWorks' Fixed-Point Advisor and Real-Time Workshop [38]. A notable feature of Costan is its error calculation through static analysis in controller code, when unrolling bounded loops, where deviations are compared with a pre-computed error bound. If any violation is found, then Costan provides a concrete test input that leads to such a failure. By contrast, DSVerifier computes quantization effects and checks stability in a closed-loop function for a plant family $\mathfrak{P}$, without handling the usual stability concept proposed by Keel and Bhattacharyya [39], who computed stability margins for measuring fragility. Such a behavior makes DSVerifier's stability verification slower than Costan; however, it provides improved accuracy, which is suitable for correct-by-design approaches [40]. Unfortunately, it seems that Costan is no longer maintained and its currently available version is obsolete, *i.e.*, it does not compile with current operating systems' libraries, which impairs experimental evaluation procedures. Rungger and Tabuada [37] established a background on robustness of CPSs and hybrid systems based on hybrid automata representations, by providing symbolic models for the robustness property that can be used to verify and synthesize robust closed-loop hybrid systems, with respect to external disturbances.

Sample And Hold Verification (SAHVY) [13] simulates system execution, by solving ODEs represented by Taylor models. It performs SMT-based BMC within a range of initial states and checks safety properties expressed by computation tree logic (CTL) formulae. Indeed, its verification engine is similar to that of DSVerifier $v2.0$; however, it is limited to hybrid systems with ZOH sampling and does not take into account FWL effects. Our work, differently, neither does not tackle external disturbances nor uses the robustness modeling provided in [37]; however, it is able to consider simultaneously FWL effects in digital controllers and parametric uncertainties that are not considered by Rungger and Tabuada [37].

Barnat *et al.* [41, 42], in turn, presented an approach that uses Simulink diagrams to open up new possibilities towards verification properties beyond standard stability tests, for first-order systems; however, it is still under development and there are limitations related to the theorem's proof (Why3 [41, 42]). In fact, Why3 can solve problems of previous studies related to state-space explosion [41], but it is not fully automatic, *i.e.*, users have to manually change parameters, in order to produce new proofs. Additionally, there is no counterexample and error trace generation and its verification is done over Simulink models (which contrasts to our study).

Finally, the studies introduced by Abate *et al.* [43–45] describe a method called Digital System Synthesizer (DSSynth), which synthesizes stable controllers for continuous plants given as transfer functions and exploits bit-accurate verification of software implemented in digital microcontrollers [11, 32]. DSSynth marks the first use of counterexample-guided inductive synthesis [46] for synthesizing digital controllers, while considering physical plants with uncertain models and FWL effects; however, low-level implementation errors (*e.g.*, LCOs) are not further investigated in those studies. In fact, our experimental evaluation shows the DSVerifier $v2.0$'s precision to detect LCO (*cf.* Section 5) in controllers synthesized by DSSynth.

Even though transfer functions can describe a huge amount of real-world systems, a drawback of DSVerifier $v2.0$ is that such a representation is still limited and it is not widely used by the aforementioned tools; however, support to state-space systems is under development [47]. Additionally, DSVerifier $v2.0$ presents some advantages over many formal verification tools available in the literature [12, 13, 36], *e.g.*, bit-precise verification, counterexamples for failures, and automated verification procedures.

3. Finite Word-Length Effects (FWL)

Finite word-length (FWL) effects are related to differences in coefficient values, due to representations used in real implementations. During the last decades, various researchers have studied FWL effects and digital controller and filter fragility [29, 48]. Some researchers focused their efforts on the design phase, by developing non-fragile design techniques [5, 49]; others, in turn, investigated improved realizations, FWL formats with adequate performance under FWL effects [50–53], and formal verification and synthesis of digital control systems, with respect to FWL effects [12, 32, 43, 44, 54].

Usually, when designing digital systems, such as digital controllers, traditional approaches [9] compute elements through mathematical models, which are encoded in computer applications and toolboxes [17]. Indeed, those descriptions are often created in floating-point arithmetic, which provide lower approximation errors for rational numbers; however, in order to reduce cost through cheaper processing units and systems, fixed-point representations may be employed, which then present higher error magnitude [31]. More specifically, floating-point representations are able to support wider amplitude ranges, with gaps between adjacent numbers that are not uniformly spaced, large errors for large numbers, and small errors for small numbers, while fixed-point ones present more restricted ranges and constant gaps, no matter a number's magnitude [55]. As a consequence, whenever design procedures are performed with floating-point representations and real systems are implemented with fixed-point ones, wrong operation may be notice in the latter.

In fact, deviation from a designed behavior occurs due to quantization and cumulated errors caused by round-off. For instance, mere quantization error directly affects locations of poles and zeros, which may be moved to the external part of the unit circle, and round-off cumulate through operations usually result in wrong or oscillating output, which may incorrectly activate or control further stages. As a consequence, our study focuses on investigating FWL effects and tackles the following properties: stability, limit-cycle oscillations, and output error. The first is only related to quantization, while the others are also due to cumulated error.

3.1. Stability

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 [1]. 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 [1].

Lemma 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 controller and plant, respectively, as shown in Figs. 7b and 7a, is internally stable if and only if:*

- *the roots of its 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, i.e.,* $\frac{N_C(z)}{D_C(z)} \cdot \frac{N_P(z)}{D_P(z)}$ *in series (cf. Fig. 7b) and* $\frac{N_P(z)}{D_P(z)}$ *in feedback configuration, has no pole-zero cancellation on or outside the unit circle.*

As a consequence, given that stability depends on poles and those are roots of denominators of transfers functions, they are directly affected by coefficient quantization, *i.e.*, their locations may be changed when fixed-point arithmetic is employed. Finally, if that change exceeds boundaries of the unit circle, systems may become unstable.

3.2. Limit-Cycle Oscillations

Limit-cycle oscillations in digital systems are defined by the presence of oscillations occurring in their outputs, even when their input sequences are composed by constant values [29], and may be classified as granular or overflow limit cycles. Granular LCOs are autonomous oscillations, originating from quantization performed in the least significant bits [56], while overflow LCOs take place after overflow and wrap-around events. In addition, even a non-zero constant output resulting from a constant input equal to zero is a limit-cycle effect [57]. Indeed, absence of overflow LCOs, in digital controllers, may be assured by preventing overflows or treating them via saturation, when the maximum (or minimum) value achieved is held; however, it may not be enough to ensure absence of persistent oscillation, in closed-loop systems.

In addition, different implementations of the same controller may present different behaviors regarding LCO, *i.e.*, one may present it and the other may not. For instance, that usually happens when the number of allocated fractional bits or a chosen scaling factor is different. As a consequence, even if a design is correctly performed and should mitigate LCO by construction, different bit fixed-point formats may or may not result in such a behavior.

3.3. Output Quantization Error

Floating-point representations provide better approximation of rational numbers, when compared with fixed-point ones with the same number of bits. Multiple-precision floating-point arithmetic can further represent rational numbers, whose precision digits are bounded by the available memory of a system [58], and practical software packages do exist to implement that type of arithmetic (e.g., MPFR[3] and MPFI[4]); however, many practical implementations of digital controllers are designed with fixed-precision arithmetic [31]. Additionally, using floating-point arithmetic in BMC leads to higher verification time and memory consumption [59]. Indeed, both CBMC and ESBMC, used as back-end model checkers in DSVerifier, support floating-point arithmetic and, in particular, the IEEE floating-point standard (IEEE 754-2008) [55]. As reported in our previous work [60], ESBMC represents the most efficient verifier for C programs that contain floating-point arithmetic; however, the model produced by DSVerifier and corresponding verification conditions are hard to be solved by both verifiers. As a consequence, DSVerifier $v2.0$ currently focuses on fixed-point representation only, with 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. A common representation, which is also used here, employs digits separated by a decimal point, where the ones to the left are the integer part and the remaining ones, to the right, are the fractional part, while using two's complement. As a

[3]https://www.mpfr.org/
[4]https://directory.fsf.org/wiki/MPFI

consequence, a real number R represented by a format $\langle I, F\rangle$ can be written as

$$R = -b_{I-1}2^{I-1} + \sum_{i=I-2}^{-F} b_i 2^i \tag{7}$$

and the output quantization error E_d, due to round-off errors when rounding to nearest [52], is given by

$$-2^{-F-1} \leq E_d \leq 2^{-F-1}. \tag{8}$$

In addition, when truncation and von Neummann rounding are performed, those are given by

$$0 \leq E_d < 2^{-F} \tag{9}$$

and

$$-2^{-F} < E_d < 2^{-F}, \tag{10}$$

respectively. The simplest rounding procedure is truncation, which works by dropping some least significant bits. Round to nearest modes provide smaller error and differ in the manner numbers half-way from two rounded ones are treated [52, 55], while von Neumann rounding aims to obtain unbiased error.

As a consequence, outputs in closed-loop systems suffer from round-off, which vary with rounding modes and are fed back to their inputs. Indeed, such errors may cumulate and result in incorrect computations, which ultimately result in wrong behaviors. In addition, in our case, the employed rounding mode is the one specified in Eq. (8). Finally, those differences in output samples could be monitored and even evaluated, in order to check if they lie within acceptable bounds.

4. Automated Verification Methodology for Fragility

DSVerifier $v2.0$'s verification flow is split into two major processes as illustrated in Fig. 4: Steps 1 to 5 are carried out by users and Steps A to D are automatically performed by DSVerifier $v2.0$. Importantly, Steps 1 to 5 result in an ANSI-C file (see Fig. 5) that contains vector representations for transfer functions corresponding to digital controller and plant models, which is then used as input for Steps $A - D$ (*cf.* Section 4.6). In addition, implementation details for a digital controller must be provided, *e.g.*, number of bits used for fractional and integer parts of fixed-point calculations, realization, input signal range, and sample time. In Step 1, users provide inputs p_0 representing a plant model, $\Delta\vec{p}_a\%$ and $\Delta\vec{p}_b\%$ through *.a_uncertainty and .b_uncertainty, respectively, which are related to their respective components of a plant model (i.e., .a_uncertainty*[0] *to .a*[0]*, .a_uncertainty*[1] *to .a*[1]*, .b_uncertainty*[0] *to .b*[0]*, and so on). They define the percentual of uncertanty (which by default is zero) to be taken into account by DSVerifier* $v2.0$*, during model generation with uncertanties.* Furthermore, sizes of numerator and denominator polynomials (*i.e.*, parameters *a_size* and *b_size*) must be provided, since typical software verifiers have difficulty in handling variable-length arrays (VLAs).[5] In Step 2, a digital controller and

[5]C99 introduced VLAs but C11 made them an optional feature.

also a control loop must be designed, with any preferred method (*e.g.*, pole assignment) and configuration (*e.g.*, series or feedback). A controller's numerical representation is then chosen in Step 3 and, in Step 4, one realization form is defined, from three different direct representations: Direct Form I (DFI), Direct Form II (DFII), and Transposed Direct Form II (TDFII). Finally, in Step 5, users configure verification parameters, *e.g.*, verification time, properties, and BMC tool. Thus, Steps 1 to 5 result in ANSI-C code that should be used as input to DSVerifier $v2.0$, whose verification engine automatically checks property ϕ (*e.g.*, stability, LCO, or output quantization error).

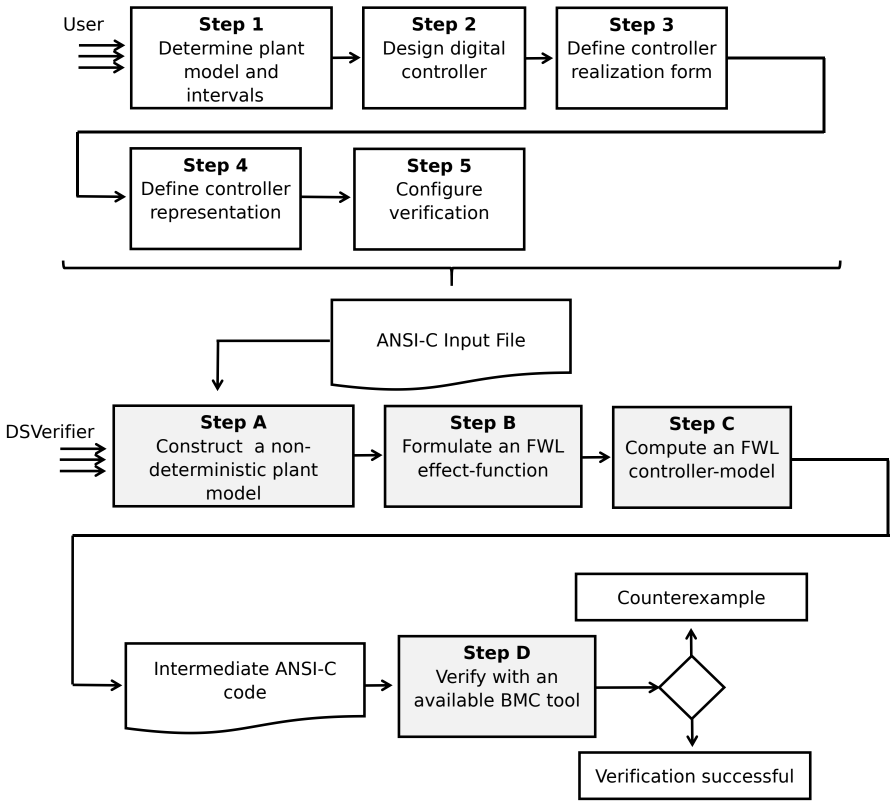

User
Step 1
Determine plant model and intervals
Step 2
Design digital controller
Step 3
Define controller realization form
Step 4
Define controller representation
Step 5
Configure verification
ANSI-C Input File
DSVerifier
Step A
Construct a non-deterministic plant model
Step B
Formulate an FWL effect-function
Step C
Compute an FWL controller-model
Counterexample
Intermediate ANSI-C code
Step D
Verify with an available BMC tool
Verification successful

Figure 4: DSVerifier $v2.0$'s verification flow.

In Step A, DSVerifier $v2.0$ builds a non-deterministic model for a plant family $\mathfrak{P}$, using p_0, $\Delta\vec{p}_a\%$, and $\Delta\vec{p}_b\%$. Then, it formulates $FWL[\cdot]$, in Step B, using implementation details provided in Steps 2 and 3, and computes $FWL[c_0]$, in Step C. Thus, DSVerifier $v2.0$ builds an intermediate ANSI-C code for a given digital system implementation and makes that an input for a checker, as indicated in Step D.

Definition 1. Non-deterministic approach *is a representation of all possible values of a given variable and is limited here by the dynamic range (minimum and maximum values) defined in the data structure "impl" (as shown in Fig. 5,*

```
#include <dsverifier.h>

digital_system controller = {
  .b = {  0.0039062  , 0.00097656 },
  .b_uncertainty = {  0.005  , 0.005 },
  .b_size =  2,
  .a = {  0.31348  , -0.00097656 },
  .a_uncertainty = {  0.005  , 0.005 },
  .a_size =  2,
  .sample_time = 2.000000e-01
};

implementation impl = {
  .int_bits =  6,
  .frac_bits =   2,
  .max =  1.000000,
  .min =  -1.000000
  };

digital_system plant = {
  .b = {  0  , 0.00097541 },
  .b_uncertainty = {  0.005  , 0.005 },
  .b_size =  2,
  .a = {  1  , -0.9512 },
  .a_size =  2,
  .a_uncertainty = {  0.005  , 0.005 }
  };
```

Figure 5: A digital-system input file for DSVerifier *v*2.0.

lines 13-18*).*

Note that this intermediate ANSI-C model contains three main modules: digital-controller code to be embedded into a microprocessor, plant model code, which simulates plant model dynamics with uncertainties, and model-checking directives, *i.e.*, `asserts` and `assumes`, which control the verification flow.

Fig. 6 shows an example of ANSI-C code[6] automatically produced by DSVerifier *v*2.0, which computes, with (*fxp_direct_form_1*) and without fixed-point FWL effects (*double_direct_form_1*), outputs for a Direct Form I (DFI) implementation structure [32] and also includes `assume` (`__DSVERIFIER_assume`) and `assert` (`__DSVERIFIER_assert`) statements, which are used for controlling system input range and checking output quantization error violations (through the chosen back-end), respectively. Indeed, the former limits non-deterministic values, within the dynamic range defined by *impl.min* and *impl.max* (shown in Fig. 5), which are applied to the digital controller input, and the latter checks if deviation between the output with (`y_qtz`) and without FWL effects (`y_double`) is greater than an admissible value provided by a user (`max_error`). It is worth noticing that computations are internally performed in DSVerifier, by using fixed-point arithmetic in $\langle I, F \rangle$ (*fxp_direct_form_1*) or floating-point representations *double_direct_form_1*. Finally, `shiftL` gets values $x(k)$ (determined with non-deterministic values) and permutes them to the left, in order to compute $y(k)$, and `fxp_direct_form_1()` is the DFI controller implementation.

On the one hand, digital controller's coefficients are quantized values and all its operations use fixed-point arithmetic (*i.e.*, additions, multiplications, subtractions and divisions). On the other hand, numerator and denominator coefficients for a plant model are not quantized. Indeed, those are represented

[6]The DSVerifier *v*2.0 code is available at `https://github.com/ssvlab/dsverifier`

```
nondet_constant_input = nondet_double();

__DSVERIFIER_assume((nondet_constant_input >= impl.min) &&
                    (nondet_constant_input <= impl.max));

for (int i = 0; i < k; ++i) {
    x_qtz[i] = nondet_constant_input;
    x_double[i] = nondet_constant_input;
    shiftL(x_qtz[i], xaux_qtz, ds.b_size);
    shiftL(x_double[i], xaux_double, ds.b_size);
    y_qtz[i] = fxp_direct_form_1(yaux_qtz, xaux_qtz,
        ds.a, ds.b, ds.a_size, ds.b_size);
    y_double[i] = double_direct_form_1(yaux_double, xaux_double,
        ds.a, ds.b, ds.a_size, ds.b_size);
    shiftL(x_qtz[i], xaux_qtz, ds.b_size);
    shiftL(x_double[i], xaux_double, ds.b_size);
    absolute_error = y_double[i] - fxp_to_double(y_qtz[i]);
    __DSVERIFIER_assert((absolute_error < (max_error)) &&
                        (absolute_error > (-max_error)));
}
```

Figure 6: Intermediate ANSI-C code fragment of a DFI controller, which was modified by DSVerifier $v2.0$.

with maximum precision, based on double-precision variables, and treated as non-deterministic variables, to support model uncertainties. Nonetheless, computer representations will always present limited precision, even for double variables. In general, double-precision variables are enough for our verification engines; however, a more comprehensive analysis may be achieved in further studies by using interval arithmetic, as done by Abate *et al.* [44]. The directive `assume` bounds non-deterministic variables, *i.e.*, inputs and plant uncertain coefficients. For instance, if a polynomial $-0.06875z^2$ has a coefficient -0.06875 (*i.e.*, a_0) with 5% of uncertainty (*i.e.*, $\Delta\vec{p_a}\%$), it will be internally represented by the non-deterministic interval $[-0.06875-\Delta\vec{p_a}\%(0.06875), -0.06875+\Delta\vec{p_a}\%(0.06875)] \Rightarrow [-0.0721875, -0.0653125]$.

Finally, in Step D, translation of intermediate ANSI-C code into SMT formulae is performed by a back-end model-checking tool (*e.g.*, CBMC [20] or ESBMC [16]). Here, DSVerifier $v2.0$ checks a given property ϕ (*e.g.*, stability, LCO, or output quantization error) with respect to a closed-loop system, which is composed by $FWL[c_0]$ and every p in $\mathfrak{P}$ (*cf.* Section 1.2). If any property violation is found, then DSVerifier $v2.0$ reports a counterexample, which contains system inputs or parametric deviations that lead to a failure. A successful verification result is reported *iff* a system is safe up to a bound k, with respect to ϕ.

In particular, stability verification is the only one that is complete, since it does not depend on system outputs and inputs (*i.e.*, no bound k for loop unwinding is defined) [32]. Furthermore, DSVerifier $v2.0$ using ESBMC as backend is able to check digital systems through proof by mathematical induction, which combines a state-of-the-art k-induction proof rule [61] with invariants [62]; however, that algorithm must be further extended, as a new direction for future work, in order to infer invariants that are inductive w.r.t. quantization and LCO properties, since invariance can not determine induction of a non-inductive assertion [63].

4.1. Loop configurations

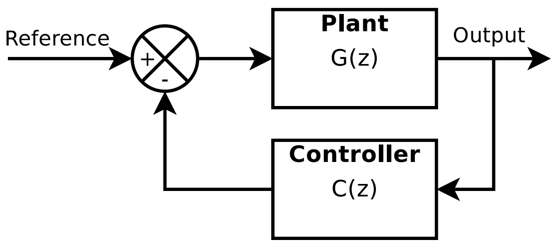
DSVerifier $v2.0$ supports two closed-loop configurations: feedback as

$$H(z) = \frac{C(z) \cdot G(z)}{(1 + C(z) \cdot G(z))} = \frac{\frac{N_C(z)}{D_C(z)} \cdot \frac{N_G(z)}{D_G(z)}}{1 + \frac{N_C(z)}{D_C(z)} \cdot \frac{N_G(z)}{D_G(z)}} = \frac{N_H(z)}{D_H(z)}, \tag{11}$$

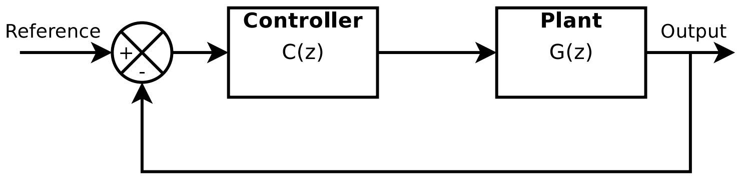
where a digital controller is connected through a feedback path (see Fig. 7a), and series as

$$H(z) = \frac{G(z)}{(1 + C(z) \cdot G(z))} = \frac{\frac{N_G(z)}{D_G(z)}}{1 + \frac{N_C(z)}{D_C(z)} \cdot \frac{N_G(z)}{D_G(z)}} = \frac{N_H(z)}{D_H(z)}, \tag{12}$$

where a controller is located at a forward path (see Fig. 7b). In the DSVerifier $v2.0$'s 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 7: Closed-loop configurations supported by DSVerifier $v2.0$.

4.2. Stability verification

As already mentioned, system stability may be influenced by FWL effects. That being said, it would be interesting to check pole location during system design, as a consequence of using fixed-point formats as final implementation. Based on Lemma 1, DSVerifier $v2.0$ is able to check stability for closed-loop systems, according to Algorithm 1. Firstly, DSVerifier $v2.0$ applies FWL effects on a controller's numerator and denominator, then it builds a non-deterministic model to represent plant family $\mathfrak{P}$ and, finally, applies the Jury's criteria [1] to determine stability regarding $S(z)$.

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 greater than 1.

```
Algorithm 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.
1  begin
2      Formulate an FWL effect function $\mathcal{FWL}[\cdot]$
3      Construct the plant interval set $\mathfrak{P}$, where $\hat{N}_P(z) \in \mathfrak{P}$ and $\hat{D}_P(z) \in \mathfrak{P}$
4      Obtain $FWL[N_C(z)]$ and $FWL[D_C(z)]$
5      Check $\neg\phi_{stability}$ for $S(z) = \mathcal{FWL}[N_C(z)] \cdot \hat{N}_P(z) + \mathcal{FWL}[D_C(z)] \cdot \hat{D}_P(z)$
6      if $\neg\phi_{stability}$ is satisfiable then
7          return FAILED and a counterexample (i.e., unstable)
8      end
9      else
10         return SUCCESS (i.e., stable)
11     end
12 end
```

4.3. Limit-cycle oscillation verification

LCO may severely compromise system behavior and operation, due to associated oscillations; however, its presence may be checked, if such repetitions are identified and characterized, and even avoided, if different approaches are employed (*e.g.*, realization and coefficient format).

As a toy example regarding LCO verification, which is supposed to present such an effect for illustrative purposes, a single-pole system, described by difference equation

$$y(n) = -a\,y(n-1) + x(n), \tag{13}$$

is adopted. Here, such a filter is also modeled using 2 bits for the integer part and 4 bits for the fractional one (as in the previous case), but with a zero input signal. If the verification engine is executed for the implemented model, then it finds a particular initial condition leading that system to a limit cycle. In Table 1, the resulting system response, for that particular condition, is presented, through columns y_2 and y_{10}, in binary and decimal formats, respectively. Due to the adopted rounding procedure (*cf.* Eq. (8)), which was applied to the fractional part of the fixed-point number, one can notice, in Table 1 and for $a = 0.5$, that the resulting output starts repeating after $n = 2$. Similarly, for $a = -0.5$, the same output keeps in a nonzero steady-state value, instead of decaying towards zero.

Table 1: Identification of Limit-cycle oscillations in the adopted toy example.

$a = 0.5_{10} = 0.1000_2$			$a = -0.5_{10} = 1.1000_2$		
n	y_2	y_{10}	n	y_2	y_{10}
-1	0.0010	0.125	-1	0.0010	0.125
0	1.0001	-0.0625	0	0.0001	0.0625
1	0.0001	0.0625	1	0.0001	0.0625
2	1.0001	-0.0625	2	0.0001	0.0625
3	0.0001	0.0625	3	0.0001	0.0625

In DSVerifier $v2.0$, LCO verification is performed in a system's general equation $H(z)$, which is computed from plant and controller transfer functions in series configuration (Eq. 12) or feedback configuration (Eq. 11). Basically, DSVerifier $v2.0$ checks the presence of persistent oscillation in an output, given a constant input signal, which is illustrated in Algorithm 2.

Algorithm 2: 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.
1  begin
2      Formulate a FWL effect function $FWL[\cdot]$
3      Construct the plant interval set $\mathfrak{P}$, where $\hat{N}_P(z) \in \mathfrak{P}$ and $\hat{D}_P(z) \in \mathfrak{P}$
4      Obtain $FWL[N_C(z)]$ and $FWL[D_C(z)]$
5      Compute $H(z)$ according to feedback or series configuration (cf. Eqs. (11) or (12),
         respectively)
6      Obtain the last output from $H(z)$, as reference
7      Check the presence of a time window
8      if size of time window is bigger than one with non-zero constant input or bigger
         than zero with zero input then
9          Check whether elements inside that time window are repeated;
10         if all elements are repeated then
11             return FAILED and a counterexample (i.e., presence of LCO)
12         end
13     end
14     else
15         return SUCCESS (i.e., LCO-free)
16     end
17 end
```

Firstly, the quantizer block routine is configured to enable wrap-around. Then, DSVerifier $v2.0$ 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. In summary, the last output is compared with the previous ones, with the goal of finding an equal element. If that happens, within a distance of w samples, a possible time window is flagged, which is encoded in line 7 of Algorithm 2. If the employed input is zero and w is greater or equal to one or the employed input is non-zero and w is greater than one (see line 8 of the same Algorithm), there is limit-cycle occurrence; otherwise, there is not. If the former happens, each element between the reference output and the first equal sample is compared with its respective pair w samples away and, if that is successful for all of them, which is performed in lines 9 and 10, DSVerifier $v2.0$ confirms presence of LCO. Precisely, our 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), *i.e.*, $w < k$. One may notice that the proposed LCO verification can also be performed for non-deterministic inputs and states, which was impossible with the previous versions of DSVerifier.

4.4. Quantization error verification

Output round-off errors may be checked, if an expected behavior is compared with an obtained one. Indeed, given that designs are often performed in floating-point and real implementations in fixed-point arithmetic, a possible verification approach would be to compare both and compute the resulting deviation.

Based on that, DSVerifier $v2.0$ is able to apply 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

$$l_{error} \iff |y_{fxp} - y_{float}| < e_b, \tag{14}$$

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.*, with greatly reduced FWL effects), and e_b is the acceptable error value defined by a designer. In summary, DSVerifier $v2.0$ 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.

Algorithm 3: Output quantization error verification

```
Data: Controller $C(z)$, plant $P(z)$, and $e_b$ as an acceptable error value.
Result: SUCCESS if the output quantization error is lower than $e_b$, otherwise
        FAILED along with a counterexample.
1  begin
2      Formulate a FWL effect function $FWL[\cdot]$
3      Construct the plant interval set $\mathfrak{P}$, where $\hat{N}_P(z) \in \mathfrak{P}$ and $\hat{D}_P(z) \in \mathfrak{P}$
4      Obtain $FWL[N_C(z)]$ and $FWL[D_C(z)]$
5      Compute $H_{fxp}(z)$ according to feedback or series configuration (cf. Eqs. (11)
         or (12)), i.e., a transfer function in fixed-point arithmetic
6      Compute $H_{float}(z)$ according to feedback or series configuration (cf. Eqs. (11)
         or (12)), i.e., a transfer function in floating-point arithmetic
7      Calculate outputs from $H_{fxp}(z)$ (i.e., $y_{fxp}(k)$)
8      Calculate outputs from $H_{float}(z)$ (i.e., $y_{float}(k)$)
9      Compute the difference between the fixed- and floating-point outputs, i.e.,
         $E_d = y_{fxp}(k) - y_{float}(k)$
10     if $E_d$ ¡ $e_b$ then
11         return SUCCESS (i.e., output quantization error is within a tolerable bound)
12         else
13             return FAILED and a counterexample (i.e., high output quantization
                 error)
14         end
15     end
16 end
```

4.5. Structured Uncertainties Description Example

DSVerifier v2.0 supports only structured uncertainties. This version does not support the specification of unstructured uncertainties. Find below an example of specification of structured uncertainties via DSVerifier for a cruise control system, whose mechanical schematic is illustrated in Fig. 4.5. The nominal continuous time transfer function $G(s)$ can be expressed as follows:

$$G(s) = \frac{1}{ms^2 + bs}. \tag{15}$$

Consider that the parameter mass (m), and damping ratio (b) are uncertain, such that $m \in [1, 2]$ kg, and $b \in [0.18, 0.22]$ N $\cdot$ s/m. The ZOH discretization is

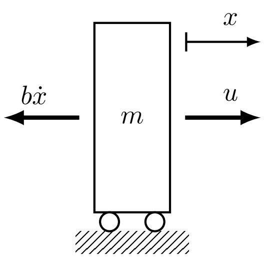


Figure 8: A spring-mass-damping system.

obtained by:

$$G(z) = (1 - z^1)\mathcal{Z}\left\{\mathcal{L}^{-1}\left\{\frac{G(s)}{s}\right\}\right\}, \tag{16}$$

where $\mathcal{Z}\left\{\cdot\right\}$ is the z transform and $\mathcal{Z}\left\{\cdot\right\}$ is the inverse Laplace transform. Thus, the discrete model of Eq. (15) with sample time $T = 0.5$ s is:

$$G(z) = \frac{\left(T - \frac{m}{b} + \frac{m}{b}e^{-\frac{T\cdot b}{m}}\right)z + \frac{m}{b} - e^{\frac{-T\cdot b}{m}}\left(\frac{b}{m} + T\right)}{bz^2 - b\left(1 + e^{-\frac{T\cdot b}{m}}\right)z + b\cdot e^{-\frac{T\cdot b}{m}}}. \tag{17}$$

Substituting, the parameters and their interval, it is obtained the following interval system:

$$G(z) = \frac{b_1 z + b_2}{a_0 z^2 + a_1 z + a_2}. \tag{18}$$

$$b_1 \in [0.011, 0.027],\ b_2 \in [3.9, 10.547],$$

$$a_0 \in [0.18, 0.22],\ a_1 \in [-0.428, 0.345],\ a_2 \in [0.164, 0.209]$$

Based on these intervals, the vectors $\vec{A}$, $\vec{B}$, $\Delta\vec{p}_a\%$, and $\Delta\vec{p}_b\%$ can be computed, such that the elements of $\vec{A}$ and $\vec{B}$ are the mid point of the above intervals, and the elements of $\Delta\vec{p}_a\%$ and $\Delta\vec{p}_b\%$ are the percentage of deviation from midpoint to bounds of intervals. Then, the following vectors are obtained

$$\vec{A} = [0.2\ \ -0.386\ 0.186]$$

$$\vec{B} = [0.019\ 7.224]$$

$$\Delta\vec{p}_a\% = [10\ 10.834\ 11.729]$$

$$\Delta\vec{p}_b\% = [41.05\ 46.005]$$

With these parameters an ANSI-C input file may be written according to Figure 5.

4.6. Illustrative Example

The methodology applied in this example follows the verification flow shown in Fig 4. Consider the plant model given by Eq. (19), which represents the pitch angle dynamics of an unmanned aerial vehicle (UAV) quadcopter system [64], and the digital controller given by Eq. (20), which was synthesised by DSSynth [44].

$$P(z) = \frac{N_P(z)}{D_P(z)} = \frac{-0.06875z^2}{z^2 - 1.696z + 0.7089}. \tag{19}$$

$$C(z) = \frac{N_C(z)}{D_C(z)} = \frac{-0.9983^2 + 0.09587z + 0.1926}{z^2 + 0.5665z + 0.75}. \quad (20)$$

The general equation $H(z)$ that represents the closed-loop system derived from (19) and (20), using feedback configuration, is described by

$$H(z) = \frac{N_H(z)}{D_H(z)} = \frac{0.06863z^4 - 0.006591z^3 - 0.01324z^2}{1.069z^4 - 1.136z^3 + 0.4849z^2 - 0.8704z + 0.5317}. \quad (21)$$

As mentioned, representations regarding digital controller and plant are needed. Therefore, by considering a fixed-point implementation $\langle 8, 8\rangle$, which corresponds to 8 bits for both integer and fractional parts, the resulting ANSI-C file is shown in Fig. 9, with plant uncertainty of 0.5% (*i.e.*, $\Delta\vec{p}_a\% = \Delta\vec{p}_b\% = 0.005$).

```
#include <dsverifier.h>

digital_system controller = {
  .b = {  -0.9983 , 0.09587, 0.1926 },
  .b_size =  3,
  .a = {  1, 0.5665, 0.75 },
  .a_size =  3,
  .sample_time = 2.000000e-01
};

implementation impl = {
  .int_bits =  8,
  .frac_bits =   8,
  .max =  1.000000,
  .min =  -1.000000,
  .max_error = 0.005
  };

digital_system plant = {
  .b = {  -0.06875 },
  .b_uncertainty = {  0.005 },
  .b_size =  1,
  .a = {  1, -1.696, 0.7089},
  .a_uncertainty = {  0.005 , 0.005, 0.005 },
  .a_size =  3,
  };
```

Figure 9: Closed-loop system from Eqs. (19) and (20), described as an ANSI-C file.

In order to check stability with the mentioned file, DSVerifier $v2.0$ must be executed using the command line

```
dsverifier <file>.c --k-size <bound> --property
STABILITY_CLOSED_LOOP --CONNECTION-MODE feedback,
```

where `<file>.c` is the ANSI-C file and <*bound*> is the maximum loop unrolling (which is set to 10, as default). By doing so, DSVerifier $v2.0$ reports that the system shown in Fig. 9 is stable. In order to validate and reproduce closed-loop system stability, one can obtain the associated step response using MATLAB, with command `dstep`, and then observe, in graph shown in Fig. 10, that the system is, in fact, stable.

If DSVerifier is used to check LCO occurrence in a closed-loop system, the digital system described in Fig. 9 might use DFI. By combining realization and fixed-point implementation, we could invoke LCO verification with

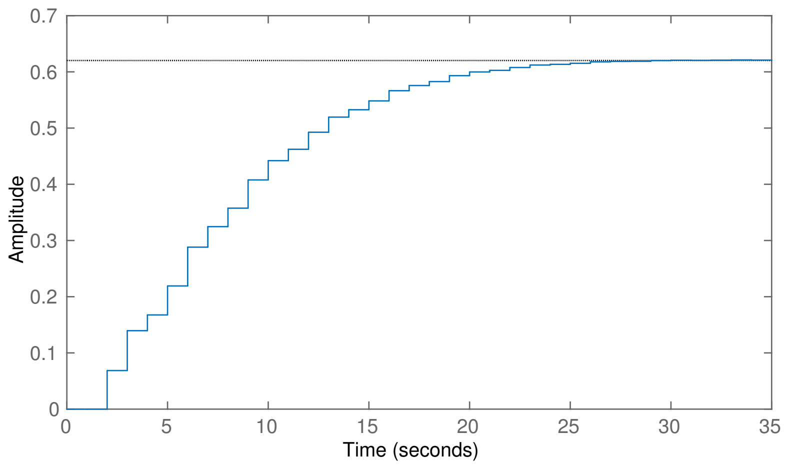


Figure 10: Step response for Eq. (19), which describes a stable UAV quadcopter system.

```
    dsverifier <file>.c --k-size <bound> --REALIZATION DFI
--CONNECTION-MODE FEEDBACK --property LIMIT_CYCLE_CLOSED_LOOP.
```

Then, DSVerifier would inform that this system presents LCO for initial states $y_{-2} = -0.99609375$, $y_{-1} = 0.0078125$, and $y_0 = 0.01171875$ and associated constant inputs formed with $x(k) = -0.015625$, as described in Table 2.

n	$x(k)$	$y(k)$
1	-0.015625	-0.00390625
2	-0.015625	0.0078125
3	-0.015625	0.01171875
4	-0.015625	-0.00390625
5	-0.015625	0.0078125
6	-0.015625	0.01171875
7	-0.015625	-0.00390625
8	-0.015625	0.0078125
9	-0.015625	0.01171875
10	-0.015625	-0.00390625

Table 2: Counterexample for LCO verification, regarding the system in Fig. 9.

In addition, initial states and constant inputs are generated as non-deterministic values, by DSVerifier, and LCO is graphically represented in Fig. 11. Finally, in order to check output quantization error, the digital closed-loop system in Fig. 9 can be used with a different configuration, in order to better understand how FWL effects are able to impact a digital system implementation. For this illustrative example, we used a DFI realization, with 2-bit in its integer part, 14-bit in its fractional one, and maximum error 0.005. Bu combining realization and fixed-point implementation, we can invoke output quantization error verification with

```
    dsverifier <file>.c --k-size <bound> --REALIZATION DFI
             --CONNECTION-MODE FEEDBACK --property
                 QUANTIZATION_ERROR_CLOSED_LOOP.
```

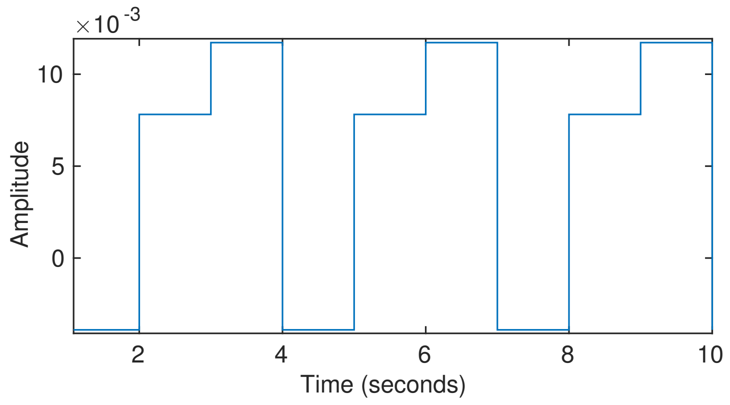


Figure 11: LCO event detected for the closed-loop system in Fig. 9.

As a consequence, DSVerifier returns output quantization error violation. Regarding the associated counterexample, DSVerifier reveals its inputs and outputs, as described in Table 3, where y_{float} represents outputs in floating-point and y_{fxp} in fixed-point arithmetic.

n	$x(k)$	$y_{fxp}(k)$	$y_{float}(k)$	$error$
1	-0,648437500000000	0,647335156250000	0,647368907928467	-0,000033751678467
2	-0,992858886718750	0,562289957470703	0,562372747837799	-0,000082790367096
3	0,997070312500000	-2,01948836503313	-2,01952749508780	0,000039130054670
4	0,315063476562500	0,312159331313241	0,312026103111179	0,000133228202062
5	-0,274230957031250	1,83378365467574	1,83386485931182	0,000133228202062
6	0,0626220703125000	-1,30108284791635	-1,30097064528742	-0,000112202628930
7	-0,553649902343750	-0,132378914595784	-0,132510583417806	0,000131668822022
8	0,138854980468750	0,871168458658317	0,871170006324547	-0,000001547666230
9	-0,346618652343750	-0,141524289462200	-0,141367224477051	-0,000157064985149
10	-0,298034667968750	-0,282161685942112	-0,282230138194609	**-0,009931547747503**

Table 3: Counterexample for output quantization error verification regarding the system in Fig. 9, with modified representation.

Moreover, inputs $x(k)$ are generated as non-deterministic values by DSVerifier and the error signal identified in its outputs is graphically represented in Fig. 12.

5. Experimental Evaluation

This section is split into five parts. Firstly, in Section 5.1, we present all benchmarks adopted for evaluating DSVerifier $v2.0$, then we describe the main goals of our experiments in Section 5.2. Further, we describe the employed setup in Section 5.3 and discuss experimental results through a performance comparison, in Section 5.4. Finally, in Section 5.5, we apply DSValidator [65] to reproduce and automatically validate the counterexamples generated for each experiment.

Availability of Data and Tools. All benchmarks, tools, and results for

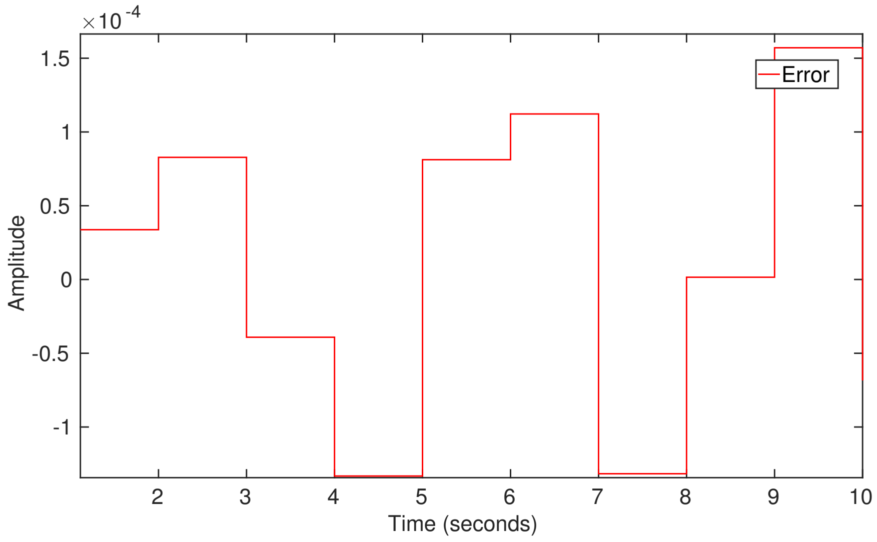


Figure 12: Output quantization error event detected for the closed-loop system represented by Eq. 21.

this evaluation are available on a supplementary web page[7].

5.1. Benchmark Description

Our experimental evaluation consists of a set of fourteen closed-loop systems, shown in Table 10, ranging from first up to eighth order [2, 32, 43]. The first benchmark, which is represented by controller C_1 and plant G_1, with $0.2ms$ of sample time, uses a discrete model for a cruise-control system of a car and accounts for rolling friction, aerodynamic drag, and gravitational disturbance force [66]. The second one, which is represented by controller C_2 and plant G_2, with a sample time of $2ms$, describes a discrete model of a DC motor [67]. The third one, which is represented by controller C_3 and plant G_3, with sample time $0.01s$, represents a discrete model of a DC servo-motor velocity dynamics [68]. The fourth benchmark, which is represented by controller C_4 and plant G_4, with a sample time of $0.02s$, contains a well-studied discrete non-minimal phase model that normally provides additional difficulties, when designing stable controllers [69]. The fifth benchmark, which is represented by controller C_5 and plant G_5, with a sample time of $2ms$, describes a discrete model for a helicopter longitudinal motion [70]. The sixth one, which is represented by controller C_6 and plant G_6, with a sample time of $2ms$, contains a discrete model for the well-known inverted pendulum that describes a pendulum dynamics with its center of mass above its pivot point [70]. The seventh benchmark, which is represented by controller C_7 and plant G_7, with a sample time of $0.001s$, uses a discrete model for satellite attitude dynamics that requires attitude control for orientation of antennas and sensors w.r.t. Earth [70]. The eighth benchmark, which is represented by controller C_8 and plant G_8, with a sample time of $0.001s$, considers a discrete model for a simple spring-mass damper plant [71].

[7]http://dsverifier.org/

The ninth benchmark, which is represented by controller C_9 and plant G_9, with a sample time of $2ms$, in turn, contains a magnetic suspension discrete model that describes the dynamics of a mass that levitates with support only of a magnetic field [70]. The tenth one, which is represented by controller C_{10} and plant G_{10}, with a sample time of $2ms$, contains a computer tape-driver discrete model that describes a system able to read and write data from a storage device [70]. One may notice that all digital controllers mentioned so far were obtained with DSSynth [45]. Finally, the last four benchmarks, which are represented by controllers C_{11}, C_{12}, C_{13}, and C_{14}, and plants G_{11}, G_{12}, G_{13}, and G_{14}, respectively, consist of digital systems extracted from Keel *et al.* [2] and Bessa *et al.* [32].

For all benchmarks, input signal ranges lie between -1 and 1, when verifying LCO and quantization-error properties. Among the discretization methods available in literature [70], we considered the sample-and-hold processes for complex systems, *i.e.*, the discrete-time plant models in Table 10 were obtained by computing discrete-pulse transfer functions from original continuous models.

5.2. Objectives

DSVerifier $v2.0$ checks properties of closed-loop control systems, *i.e.*, stability, output quantization error, and LCO. In summary, our experimental evaluation aims to answer two research questions:

RQ1 **(performance)** Is our BMC tool able to check violations related to stability, LCO, and output quantization error in closed-loop systems with uncertainty, in a reasonable amount of time?

RQ2 **(sanity check)** Is the proposed verification sound and can its counterexample reproducibility be confirmed by an external tool?

5.3. Experimental Setup

The present study employed DSVerifier $v2.0$ to check the fourteen closed-loop control systems described in Section 5.1. The related experiments were based on 3 different implementations (*i.e.*, 8-, 16-, and 32-bit) and 3 different realization forms (*i.e.*, Direct-Form I, Direct-Form II, and Transposed Direct-Form II) [56]. In addition, we verified each benchmark regarding uncertainties of 0%, 0.5%, 1.5% and 5%, against 3 properties: stability, output quantization error, and LCO. In summary, we performed 924 experiments with DSVerifier $v2.0$, with CBMC $v5.8$ [20] as the back-end model checker and MiniSAT [24] as the back-end solver.

The present experiments were executed on an otherwise idle computer with Intel Core i7 $-$ 2600 3.40 GHz processor and 24 GB of random access memory, running Ubuntu 64-bit OS. 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), while the execution-time limit was set to $3600s$.

It is worth noticing that all computations are performed in true fixed-point arithmetic, through format $\langle I, F\rangle$, which includes coefficients, operands, and operation results. Firstly, we convert coefficients to fixed-point format and then all following operations are also performed in fixed-point, until outputs are found.

5.4. General Results and Discussion

In order to answer **RQ1**, we have carried out experiments based on our set of benchmarks (*cf.* Section 5.1), according to the setup description presented in Section 5.3. In general, uncertainty bounds depend on specific applications and on uncertain physical parameters of plants (*e.g.*, masses, lengths, viscosity, and stiffness). As a consequence, the realistic uncertainty bounds used in our experiments were carefully chosen, in order to properly evaluate the DSVerifier $v2.0$'s effectiveness. Regarding the stability property, we have 168 closed-loop system implementations, and DSVerifier $v2.0$ returned that 58 of them are stable, while 110 are unstable (see Fig. 13).

LCO and output quantization error properties have been verified only in stable closed-loop system implementations[8] with uncertainties of 0%, 0.5% and 1.5%. Indeed, we avoided higher percentages of uncertainty on those experiments (*e.g.*, >5%), since they dramatically increase associated state spaces, which typically leads to longer verification times, which then makes our approach susceptible to timeouts. If a verification procedure takes a long time to find a solution, a timeout could be reached, our verification would not finish, and, as a consequence, results associated to an employed uncertainty level might not be conclusive. In addition, we have further performed experiments only on stable implementations, since unstable ones are inherently susceptible to LCO and output quantization error.

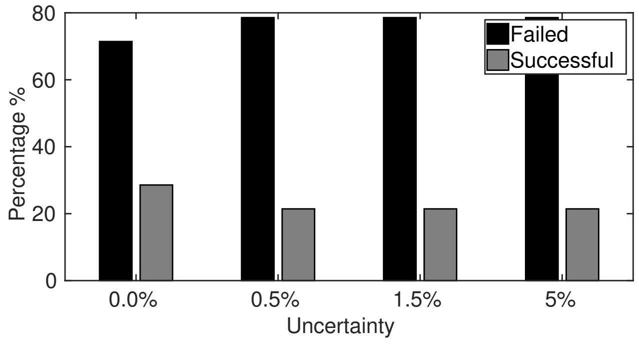
Regarding systems implemented with a precision of 8 bits, we have verified 10 stable implementations with 3 different realizations, *i.e.*, 30 verifications. For the ones implemented with 16 bits, we have checked 16 stable implementations with 3 different realizations, *i.e.*, 48 verifications. Finally, for those implemented with 32 bits, we have evaluated 21 stable implementations with 3 different realizations, *i.e.*, 63 verifications. In summary, we have verified 114 benchmarks for output quantization error and LCO, which led to 228 experiments. Regarding all chosen properties, we have checked a total of 396 closed-loop system implementations, with DSVerifier $v2.0$.

In general, we have obtained that 35% of our controllers are stable, while 65% are unstable. Among our stable controllers, we have checked that 66% of the chosen implementations presented LCO, 48% output quantization error, and 11% timed out during verification. The highest times in LCO and output quantization error verification procedures are explained by the inherent complexity of their associated algorithms, with non-deterministic initial states, (constant) inputs, and oscillation periods. Despite that, output quantization error verification procedures were concluded for 91% of the chosen benchmarks, while LCO and stability ones were concluded for all of them.

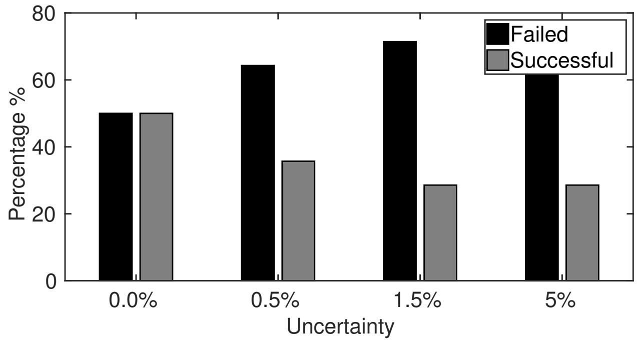
5.4.1. Stability Occurrence Discussion

For the stability verification (see Fig. 13), 110 (65%) implementations failed (*i.e.*, unstable closed-loop systems). In particular, 8 and 16-bit implementations produced more than 50% of unstable systems; importantly, the same systems turned from failure to success when implemented in 32 bits of precision. Here, we can clearly see the impact of FWL effects, according to the number of bits used in a specific implementation. In addition, if implementations are combined

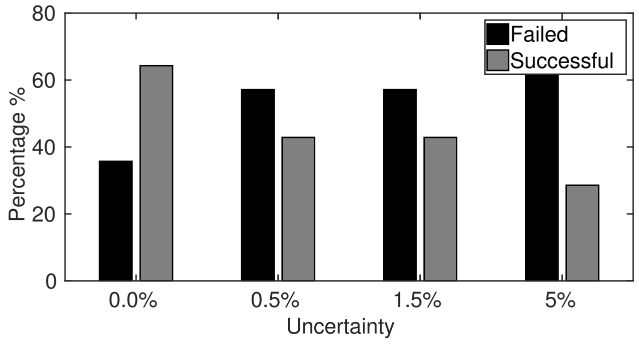
[8]All stable benchmarks are listed at `http://ssvlab.hussama.io/dsverifier/benchmarks/jss-benchmarks/`



(a) 8 bits.



(b) 16 bits.



(c) 32 bits.

Figure 13: Stability verification results, whereas (13a), (13b), and (13c) correspond to experiments run on digital systems with precisions of 8, 16, and 32 bits, respectively.

with an uncertainty of 5%, failures are higher when compared with uncertainties of 0%, 0.5%, and 1.5%, which states that the disturbance related to uncertainties heavily influence stability of a closed-loop system.

Furthermore, one may notice, in Fig. 13, that more than 70% of the controllers implemented with 8 bits are unstable, for each uncertainty. For the ones implemented with 16-bit precision, experimental results show that the number of stable controllers increases. Finally, regarding 32-bit implementations, at least

50% of the associated controllers are stable for uncertainties of 0.0%, 0.5%, and 1.5%. Therefore, one may conclude that when the number of bits is increased, the number of stable systems increases as well, due to better precision.

According to the experimental results, the closed-loop system (H_4), which is composed by controller C_4 (Eq. (22)) and plant G_4 (Eq. (23)), presented different verification results for different levels of uncertainty, *i.e.*, with 0%, it was reported as stable; however, with 0.5%, the resulting one was reported as unstable. Regarding such a system,

$$C_4 = \frac{b_3z^3 + b_2z^2 + b_1z + b_0}{a_3z^3 + a_2z^2 + a_1z + a_0}, \tag{22}$$

where $b_3 = -0.580535888671875$, $a_3 = 0.7188720703125$, $b_2 = 0.919769287109375$, $a_2 = -0.38751220703125$, $b_1 = 0.11871337890625$, $a_1 = -0.415924072265625$, $b_0 = -0.951934814453125$, and $a_0 = 0.437286376953125$, and

$$G_4 = \frac{-0.01285z^2 + 0.02582z - 0.01293}{z^3 - 2.99z^2 + 2.983z - 0.9929}. \tag{23}$$

Based on a fixed-point implementation $\langle 3, 5\rangle$, with 3 bits in its integer part and 5 in its fractional one, DSVerifier *v*2.0 returns *stable*, when considering an uncertainty of 0% (see Fig. 14); however, it returns *unstable*, for an uncertainty of 0.5% (see Fig. 15), which means poles of that system are placed on the outside part of the unitary circle. Indeed, if one plots a zeros and poles map of H_4, in order to check stability and considering each uncertainty, it becomes clear that the results found in the experiments are reproducible (the *stable* one is shown in Fig. 14 and the *unstable* one in Fig. 15).

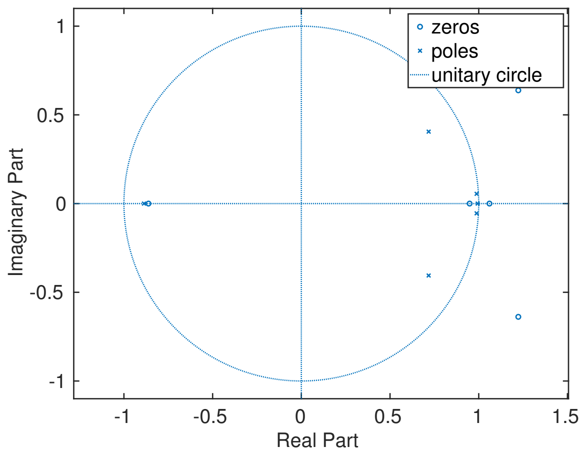


Figure 14: Zeros and Poles Map of the closed-loop system H_4, with 0% of uncertainty.

As a consequence, one could say, as general conclusion, that a good way of dealing with uncertainty is to use as many bits as possible, in any digital-controller fixed-point implementation.

5.4.2. LCO Occurrence Discussion

Regarding the LCO experiments (see Fig. 16), only 39 implementations did not present LCO (*i.e.*, 34%), according to DSVerifier *v*2.0. In fact, Fig. 16

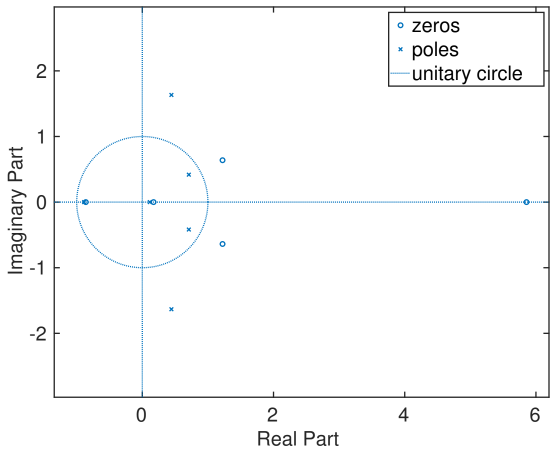
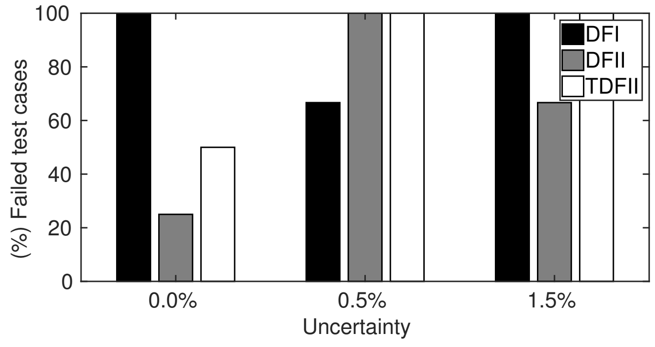
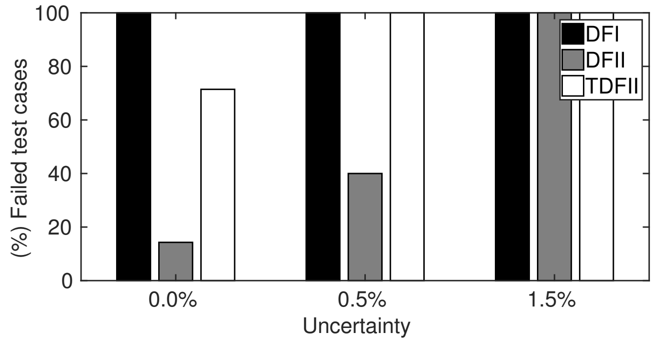


Figure 15: Zeros and Poles Maps of the closed-loop system H_4, with 0.5% of uncertainty.

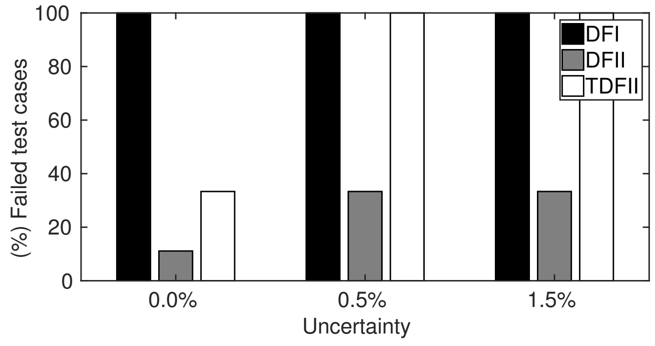
summarizes the obtained verification results for the LCO property, which show that 76% of our controllers presented LCO, when using 8-bit implementations. In particular, for DFII realizations, more than 50% of our controllers did not present LCO, which means that, for our set of benchmarks, DFII realizations presented better results, when compared with DFI and TDFII ones, in order to avoid LCO occurrence in closed-loop systems. Indeed, DFI and TDFII realizations present less nodes to check any overflow than that of DFII realization, which represents less quantization operations performed during computations. As a consequence, DFII realization needs to handle with overflows in more than one node, which could be by saturation or wrap-around mode. If an overflow is detected during the computation for DFI and TDFII realizations, the output is automatically influenced by this overflow, while DFII realization performs one more step to avoid overflow during the computation (by saturation or wrap-around). In our experiments, the overflow is avoided by employing wrap-around mode. As a result, for our set of benchmarks (which is very specific for our study), DFII realization presented less LCO occurrences in some closed-loop systems than that of DFI and TDFII realizations, and then, the results for DFII realization are better than that of DFI and TDFII realizations. When we used 16-bit implementations, our results showed that 72% of our controllers failed for the LCO property and we have also noticed that the ones not presenting LCO, in 8-bit forms, are the same in 16-bit ones, which means that 8 bits would be enough for them. Finally, for 32-bit implementations, 70% of our controllers failed for the LCO property and the number of correct DFII realizations increased,when compared with elements designed with 8 bits (more controllers did not present LCO). In particular, we noticed that when changing from DFI to DFII (or TDFII), LCO occurrences were not identified in some controllers. In addition, when we configured our verification procedures with uncertainty of 1.5%, for 16-bit implementations, all controllers presented LCO, according to DSVerifier $v2.0$. We also noticed that controllers H_2 and H_9 that did not present LCO, with 8 bits, are the same as those that did not present LCO in 16-bit and 32-bit implementations, which means that, for our set of benchmarks, the implemented controllers are appropriate to avoid LCO; however, those results are not transposable for all realization forms, because they are very specific for DFII and TDFII realizations, no matter the adopted uncertainty level.



(a) 8 bits.



(b) 16 bits.



(c) 32 bits.

Figure 16: LCO verification results.

In particular, we have noticed that the closed-loop system H_2 presented LCO for a 8-bit format and DFI realization, with initial states $y_{-2} = -0.9921875$, $y_{-1} = -0.9921875$, and $y_0 = -0.21875$, while, for DFII and TDFII, it did not present LCO. One may notice that the LCO occurrence detected for closed-loop system H_2 is classified as a granular one, because the difference between the maximum (*i.e.*, -0.171875) and minimum (*i.e.*, -0.1875) amplitudes is only in fractional parts, and also due to the constant input, which was 0.375. Fig. 17 shows the LCO occurrence in closed-loop system H_2. As already mentioned, the value computed for the constant input was 0.375, which was obtained with a non-deterministic approach. Finally, the same closed-loop system (H_2) implemented in DFII realization form and under the same input is LCO-free, as can be seen in Fig. 18.

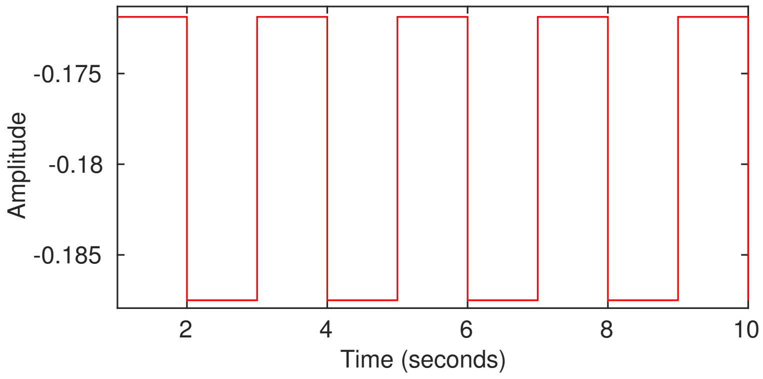


Figure 17: Closed-loop system H_2 with LCO violation in DFI realization.

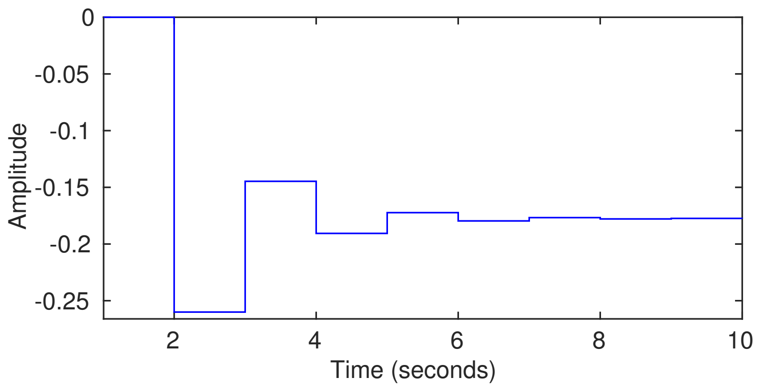


Figure 18: Closed-loop system H_2 without LCO in DFII realization.

In LCO verification, we also noticed that the chosen implementations took a reasonable amount of time. Some closed-loop systems are eighth-order ones, which means that many non-deterministic initial states are considered and there are more arithmetic operations, which consequently increases the model checking procedures' computational cost. In fact, LCO verifications tend to take longer than stability ones, due to their algorithmic complexity, *i.e.*, a search for persistent oscillations in a system's output, based on combinations of non-deterministic constant input, initial states, and oscillation window size. As a conclusion, for our set of benchmarks, we have checked that the appropriate implementation should use DFII realization and 32-bit implementations, in order to avoid LCO.

5.4.3. Output Quantization Error Occurrence Discussion

For the quantization error verification (see Fig. 19), we obtained that 47 implementations (*i.e.*, 41%) did not present quantization error, 13 timed out (*i.e.*, 11%), and 54 (*i.e.*, 48%) failed. In fact, Fig. 19 summarizes the obtained verification results for the output quantization error property, which shows that 100% of our controllers did not presented quantization error for DFII realization, with all bits implementation (*i.e.*, 8-bit, 16-bit, and 32-bit) and regarding all uncertainty levels, which means that, for our set of benchmarks, the DFII realization is the suitable one, in order to avoid output quantization error. In addition, when we increased the number of bits from 8 to 16 and 32, our set of benchmarks were more susceptible to timeouts, which represented 11% of them. The maximum allowed error (E_d) adopted for our set of experiments was defined as 0.05, which was chosen according to usual admissible errors in real systems.

Assuming closed-loop system H_2, *i.e.*, controller C_2, and plant G_2, as represented in Eqs. (24) and (25), respectively, which are given as

$$C_2 = \frac{-0.3466796875z + 0.015625}{0.5z^2 + 0.19921875z} \tag{24}$$

and

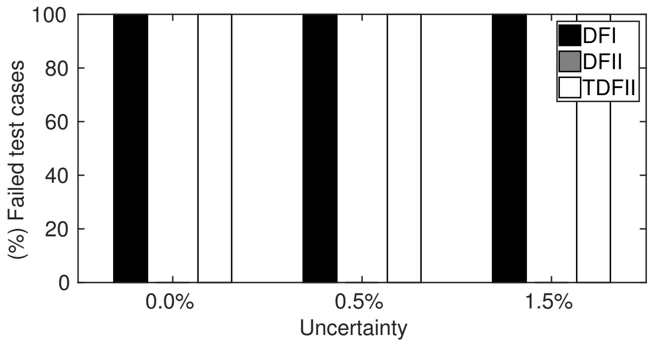
$$G_2 = \frac{0.1898z + 1.8027e^{-4}}{z^2 - 0.9012z - 1.0006e^{-16}} \tag{25}$$

and were implemented with 8 bits (*i.e.*, 1-bit for its integer part and 7-bit for its fractional one) and 0% of uncertainty, we were able to notice that, across different realizations (*i.e.*, DFI and TDFII), the same closed-loop system presented output quantization error violations. In DFII, H_2 presented no output quantization error violation, which means that implementing it with a DFII realization makes output quantization error effects not significantly detectable, according to our adopted bounds and experiments. For that specific realization (DFII), we noticed that its structure is the most suitable approach, for our set of benchmarks; however, other studies in literature concluded that there are also fewer output quantization error occurrences for other structures, such as cascade and parallel ones [72]. For our set of benchmarks, which is based on real system controllers, we have found that DFII realization could be employed as a base structure for usual implementations, while possible bit formats would then be explored.

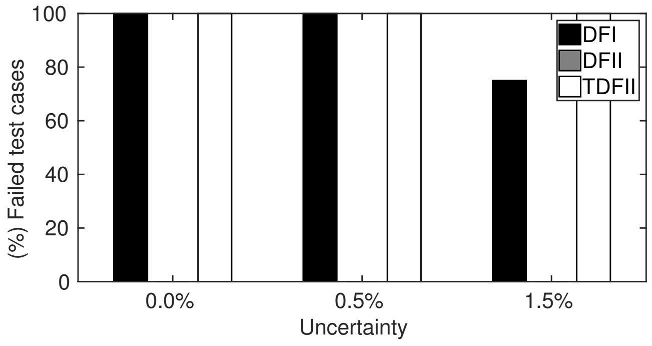
Table 4 shows the output from the output quantization error verification for the mentioned experiment, using DFI, while Table 5 shows that in TDFII realization. As can be seen for DFI and TDFII, there is presence of quantization error in the produced outputs, which means that $y_{fxp}(k)$-$y_{float}(k)$ is larger than the maximum error allowed (E_d), *i.e.*, 0.05. The produced error is represented in Fig. 20, for DFI, and in Fig. 21, for TDFII.

In our experiments, as already mentioned, the maximum allowed error E_d was defined as 0.05. In practice, it heavily depends on applications; in particular, on its specification. In fact, in Table 4 (*i.e.*, results for DFI realization), detection occurred when $n = 2$, which produced error larger than E_d.

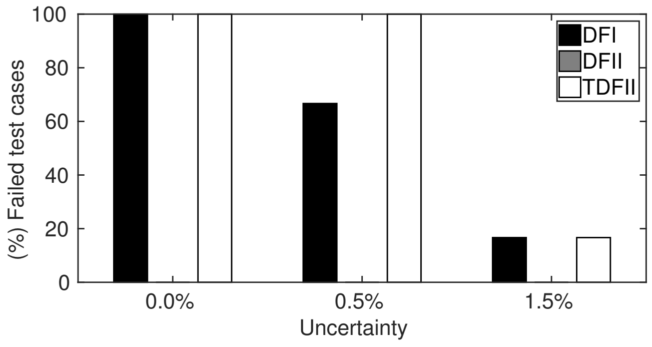
In addition, in Table 5 (*i.e.*, realization results with TDFII), detection occurred when $n = 4$, which produced error larger than E_d. In fact, as already mentioned, when using DFII realization form, closed-loop system H_2 does not



(a) 8 bits.



(b) 16 bits.



(c) 32 bits.

Figure 19: Output quantization error verification results.

contain quantization error, as can be seen in Table 6, with fixed- and floating-point arithmetic.

Finally, for our set of benchmarks, we can conclude that the appropriate implementation to be used is the DFII realization, in order to avoid output quantization error.

n	$x(k)$	$y_{fxp}(k)$	$y_{float}(k)$	$error$
1	-0,718750000000000	0	0	0
2	-1	0,505371093750000	0,498352500000000	**0,07001859375000002**
3	-0,0937500000000000	0,483253479003906	0,472335492400000	0,0109179866039063
4	-0,523437500000000	-0,154102921485901	-0,154444853591856	0,000341932105955123
5	0,703125000000000	0,425308758392930	0,421537944965139	0,00377081342779090
6	0,648437500000000	-0,676878421247238	-0,671833750666910	-0,00504467058032820
7	-0,242187500000000	-0,169554327637798	-0,159942529134276	-0,00961179850352123
8	0,976562500000000	0,256783917046015	0,251914298183261	0,00486961886275361
9	-0,992187500000000	-0,794520084783600	-0,785050467343139	-0,00946961744046093
10	1	1,03850882218109	1,03125621133320	0,00725261084789342

Table 4: Output samples from the output quantization error verification for the second benchmark, in DFI realization.

n	$x(k)$	$y_{fxp}(k)$	$y_{float}(k)$	$error$
1	-0,195312500000000	0	0	0
2	0,00781250000000000	0,137329101562500	0,135421875000000	0,00190722656250000
3	-1	-0,0652408599853516 -	0,0654778825000000	0,000237022514648438
4	0,00781250000000000	0,728853851556778	0,719693148128300	**0,09016070342847797**
5	-0,882812500000000	-0,321451699826866	-0,323421412940240	0,00196971311337346
6	0	0,746538749932370	0,741215043396909	0,00532370653546055
7	0,546875000000000	-0,319204589817332	-0,322917612516065	0,00371302269873258
8	-0,945312500000000	-0,259832191477605	-0,250517956469099	-0,00931423500850548
9	0,984375000000000	0,783259645108439	0,772348093325548	0,0109115517828915
10	0	-1,02764048637048	-1,01980163992963	-0,00783884644085298

Table 5: Output samples from the output quantization error verification for the second benchmark, in TDFII realization.

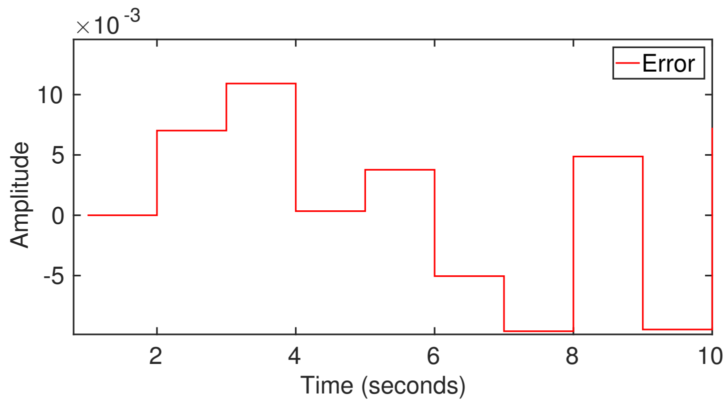


Figure 20: Closed-loop system H_2 with output quantization error in DFI realization.

5.4.4. *Verification Efficiency Discussion*

It is important to elaborate on verification efficiency. The mean time (disregarding timeouts) spent for verifying a closed-loop system is around 5.5 hours ($\sigma = 2.1h$) for stability, 13.5 hours ($\sigma = 2.2h$) for LCO, and 14.3 hours ($\sigma = 5.3h$) for output quantization error.

One may notice that high standard deviation regarding verification times indicate that the time spent in a successful verification is much longer than what is necessary to find a violation, *i.e.*, the time spent to achieve a failure result with

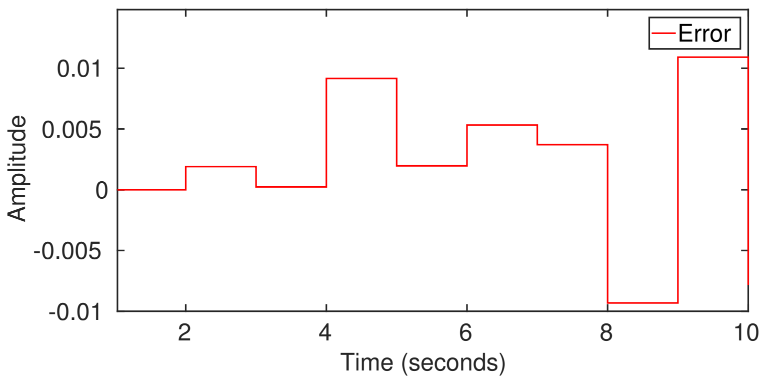


Figure 21: Closed-loop system H_2 with output quantization error in TDFII realization.

n	$x(k)$	$y_{fxp}(k)$	$y_{float}(k)$	$error$
1	-0,00781250000000000	0	0	0
2	-0,500000000000000	0,00549316406250000	0,00541687500000000	7,62890625000002e-05
3	-0,0156250000000000	0,349172592163086	0,344277559700000	0,00489503246308592
4	-0,882812500000000	-0,141034215688705	-0,141965200886868	0,000930985198162565
5	-0,109375000000000	0,675330748315901	0,668183208391364	0,00714753992453698
6	-0,773437500000000	-0,214484667310899	-0,217982558176455	0,00349789086555624
7	-0,335937500000000	0,624188346605820	0,619705626729827	0,00448271987599314
8	1	-0,0317874400803984	-0,0381598067892321	0,00637236670883379
9	-1	-0,701206078093594	-0,688653653457898	-0,0125524246356961
10	-0,796875000000000	1,00828362425531	0,998997161683765	0,00928646257154542

Table 6: Output samples from the output quantization error verification for the second benchmark, in DFII realization.

a model checking procedure, which was already expected, since CBMC [20] needs to explore all paths in C code, in order to conclude that there is no violation. In general, the total time spent to find all violations, in our set of benchmarks, was 21.9 hours, for stability verification, and 40.4 hours, for LCO verification. Regarding output quantization error verification, the total time spent to find all violations, in our benchmarks, was 42.8 hours.

In general, LCO and output quantization error verification times take longer than stability ones, as shown in Table 8, due to the fact that the output quantization error and LCO algorithms are much more complex and consider all possible initial states, constant inputs, and oscillation periods. It is worth noticing that some of the benchmarks employed in our verification procedures present orders greater than eight; indeed, it is the first time that DSVerifier works with verification of such high-order systems [11, 31, 32]. In addition, the output quantization error verification presented more timeout events, which represent 11% of our benchmarks, due to the complexity of the associated algorithm and the high-order systems used in our benchmarks.

Moreover, C code used during our experiments had a size of 13573 lines, and we have also recorded the size of SAT/SMT formulae for each benchmark, during our experiments, as can be seen in Table 7. In particular, our experiments showed that size for SAT/SMT formulae increases if uncertainty is considered in verification procedures of closed-loop systems.

System	Without Uncertainty	With Uncertainty
H_1	1277244 variables, 4321669 clauses	1727643 variables, 8266646 clauses
H_2	1722172 variables, 5241322 clauses	2360240 variables, 11329867 clauses
H_3	3253587 variables, 6125808 clauses	5338240 variables, 26248761 clauses
H_4	1844814 variables, 7139887 clauses	2895533 variables, 14736433 clauses
H_5	2472803 variables, 5962070 clauses	3195781 variables, 14218084 clauses
H_6	1856178 variables, 5665784 clauses	2376520 variables, 11391401 clauses
H_7	1434076 variables, 5099746 clauses	2627499 variables, 13020009 clauses
H_8	1630051 variables, 4656029 clauses	2218405 variables, 10479693 clauses
H_9	2764042 variables, 5148166 clauses	3472051 variables, 15715523 clauses
H_{10}	1179154 variables, 4403589 clauses	1714830 variables, 8254551 clauses
H_{11}	1434491 variables, 5240815 clauses	1985506 variables, 9825689 clauses
H_{12}	1831898 variables, 6647088 clauses	2608707 variables, 13128360 clauses
H_{13}	2637406 variables, 8339858 clauses	3257443 variables, 16232626 clauses
H_{14}	3219849 variables, 8736102 clauses	4037904 variables, 20407908 clauses

Table 7: Size of SAT/SMT formulae for each employed benchmark.

One may notice that, although software model checking has been experiencing significant progress in the last two decades, one major bottleneck for its practical applications remains being scalability. In particular, BMC is a promising approach to check digital control systems [32], but its application for refuting properties in large instances is still limited by its resource requirements [31]. That happens when BMC techniques unwind all loops, up to their given maximum bound or completeness threshold [73], which is typically infeasible when checking some realistic control systems. In this study, we have proposed an encoding approach able to be efficiently handled by underlying SMT solvers, *e.g.*, use of Jury's Criteria for stability check, which does not depend on bound k, and fixed-point arithmetic for computation modeling. We have also investigated the application of k-induction and abstract interpretation techniques, in combination with BMC procedures in previous work [74]; however, we were still unable to scale our verification engine to larger instances. Nonetheless, in our experiments, an unwinding bound (k) of 10 was enough for finding most property violations. In particular, this value was empirically determined, by considering different orders and realization forms (*e.g.*, direct and delta) of digital controllers. Although this approach is an under-approximation, we have not encountered any problems in our benchmarks.

In order to prove that our controllers are safe for any depth k, we have applied a state-of-the-art k-induction algorithm to both falsify and prove safety properties in digital controllers; however, our experiments were inconclusive, since this k-induction algorithm was unable to prove safety for all reachable states of the controllers, *i.e.*, that procedure did not terminate, possibly due to large a state-space exploration. Indeed, the employed k-induction algorithm was able to find the same property violations (with the respective counterexamples) as with plain BMC procedure; however, it tends to consume more time and memory. There are verification tools (*e.g.*, Impara [75]) that implement the interpolation and SAT-based model checking approach described by McMillan [76], but as we have observed over the last years, in the international software verification competition (SV-COMP), that algorithm does not seem to produce better results, when compared with the k-induction approach. We were able to further investigate a "property-based reachability" (or IC3) procedure for safety verification of digital controllers, but we have not found any software tool that is publicly available for verifying safety properties in full C programs, via IC3.

Uncertainty/Property	Stability	Limit-Cycle	Quantization Error
0.0%	2,3h	16,0h	8,7h
0.5%	6,1h	12,7h	14,9h
1.5%	6,4h	11,8h	19,2h
5.0%	7,1h	-	-

Table 8: Mean-time results for verification of each uncertainty level.

5.5. On the Validation of DSVerifier's Results

In order to answer RQ2, we have performed validation regarding results produced by DSVerifier $v2.0$, through reproduction of the counterexamples generated for each failed verification and confirmation of final results. The main purpose of the employed tool, names as DSValidator [65], is to automatically check whether a given counterexample, provided by DSVerifier, is reproducible or irreproducible. Indeed, it is able to reproduce counterexamples generated by DSVerifier, by using typical MATLAB features. As a consequence, it is also suitable for investigating digital system behavior, when considering implementation and FWL aspects. Thus, DSValidator supports automatic validation of results generated by DSVerifier. In addition, it takes into account implementation aspects, overflow mode (*i.e.*, saturate or wrap-around), and rounding approach (*i.e.*, floor or round). Currently, DSValidator is able to perform counterexample reproducibility for stability, minimum-phase, LCO, output quantization error, and overflow occurrences.

In DSValidator, when we employ the counterexample to reproduce the violation that has been found by DSVerifier, we do not undo the discretization on the closed-loop system. In fact, we just take the closed-loop system that was previously discretized, employ the initial states and the inputs provided by the DSVerifier counterexample, and then apply all quantizations and fixed-point operations to compute the outputs by running the scripts inside MATLAB (DSValidator). If the outputs produced via simulation in MATLAB (DSValidator) are the same outputs produced by that of DSVerifier, then we can confirm that the result found by DSVerifier is indeed reliable and reproducible. Note that, during this procedure to compute the output in DValidator, we perform the same algorithm employed in DSVerifier, i.e., we apply the same fixed-point representations, realization form, coefficients, and quantization procedure.

Property Evaluated	**Reproducible**	**Irreproducible**	**Execution Time**
Stability	110	0	0.50703 s
limit cycle	75	0	0.74359 s
Quantization Error	54	0	0.82934 s

Table 9: Reproducibility results for our set of benchmarks.

According to Table 9, DSVerifier produced 110 stability, 54 output quantization error, and 75 LCO counterexamples. DSValidator was able to reproduce all DSVerifier's counterexamples, which suggests that the latter is sound and reliable. Nonetheless, output quantization error and LCO present high-complexity counterexamples, due to the enormous amount of states, which are generated to reach a given violation, in closed-loop control systems. Even so, DSValidator is able to quickly reproduce those counterexamples, *i.e.*, in less than one second, since it just replays them with actual inputs and states over actual systems. In addition, controllers that do not present LCO or quantization error, when evaluated by DSVerifier, except for the ones present in the assembled test set, whose results were validated by DSValidator, are not undoubtedly free from violations, due to the depth that we have to employ for any verification, *i.e.*, $k = 10$.

5.6. Threats to validity

Benchmark selection. We reported an assessment of our approaches, over a diverse set of real-world benchmarks. Nevertheless, that set of benchmarks is limited within the scope of this paper and the obtained performance may not be generalized to other groups.

Fixed-point implementation and realization. Our experiments were composed by different implementations (*i.e.,* 8, 16 and 32 bits) and realizations (*i.e.,* DFI, DFII and TDFII), against four different uncertainty levels (*i.e.,* 0%, 0.5%, 1.5% and 5%). The purpose of this set of closed-loop system implementations was to emphasize that DSVerifier $v2.0$ is able to check digital systems with different fixed-point formats and realizations, while considering uncertainty. In addition, with our results, we have been able to check how those three variables influence closed-loop system performance, regarding sensibility to FWL effects and violations related to LCO, output quantization error, and stability.

Noise-free model. DSVerifier does not consider process or sensor noise in its verification model. Nonetheless, noise-rejection ability is a consequence given by Lemma 1fadali, as demonstrated by Fadali [1]. Furthermore, the effect of noise in the output signal's dynamics can be investigated through DSVerifier, by checking the noise sensitivity transfer function [70].

Numerical aspects. One may notice that our experiments performed verification with plant discrete-models, in order to investigate occurrence of violations, in closed-loop systems. In general, we use high precision for plants, *i.e.*, floating-point arithmetic; however, FWL effects can still influence them, due to the finite representation models designed for computers. Due to that limitation, if there are small errors during computations, which were caused by FWL effects, our engine does not consider them. Further work includes use of interval arithmetic [77, 78], in order to reduce numerical issues.

Correctness of our models. The idea of encoding properties of digital control systems into C programs has already been discussed in our previous work [31, 32], *i.e.*, how to convert realization forms into C code, and correctness of such C models is actually a major issue. Consequently, the usefulness of our approach relies on the fact that our C models approximate original behaviours of digital control systems. In that sense, all developed C models were manually verified and exhaustively compared with original digital control systems, in order to ensure the same behaviour. One may notice further that behaviors of digital control systems are actually represented in C code, by using realization forms [31] and native C functions (*e.g.*, *log*, *exp*, and *assert*). The soundness proof for those native C functions, which are already supported by ESBMC, can be found in Cordeiro *et al.* [16]. Although further proofs regarding soundness of C models could be carried out, it represents a hard task, due to unbounded memory usage (*e.g.*, we do not know, in advance, the number of samples that should be provided to a given digital system).

6. Conclusions

DSVerifier $v2.0$ included novel verification methods w.r.t. its previous release, in order to allow engineers to perform closed-loop system verification [11]. In particular, DSVerifier $v2.0$ is now able to consider hardware implementation aspects during verification of fundamental properties of digital control systems, which consists of digital controller and plant modeled by an uncertain discrete transfer function. In this respect, DSVerifier $v2.0$ is able to check stability and occurrence of LCO in closed-loop systems, by using two loop configurations: series and feedback. It is also able to compute the output of a closed-loop control system, while considering round-off and FWL effects, and compare that with an near-ideal response (*i.e.*, with floating-point arithmetic), in order to check whether output quantization error is within tolerable bounds. Lastly, DSVerifier $v2.0$ also uses state-of-the-art model checkers as its back-end, whose efficiency and effectiveness were confirmed in recent competitions [28, 79]. Our experimental evaluation suggests that DSVerifier $v2.0$ can be considered as an automated and reliable verification tool for improving digital control system design, while considering both fragility and robustness aspects, which was not true for previous verification approaches.

In addition, we were able to verify, with DSVerifier $v2.0$, real-world closed-loop systems with high-order, regarding different realizations, implementations, and uncertainty levels. Indeed, for our set of benchmarks, we were able to evaluate closed-loop systems properties as stability (34.5% stable and 65.5% unstable), output quantization error (41% are quantization-error free, 11% timed out, and 48% failed), and LCO (34% are LCO-free and 66% failed). Verification of closed-loop systems were not previously supported by DSVerifier $v1.0$ [11], and now DSVerifier $v2.0$ is able to not only verify previous properties supported for open-loop systems, but also for closed-loop ones, while considering uncertainty.

Our experimental results also showed that, when we implement a closed-loop system in DFII realization, output quantization error occurrence is minimized, which means that DFII could be employed as a default structure, in order to avoid quantization error effects. Additionally, we were able to check that even for unstable closed-loop systems, their implementations are still susceptible to FWL effects, *i.e.*, they produce round-offs (limit-cycles) and output quantization error violation. Finally, greater number of bits is also desirable for any representation, because it helps mitigate FWL effects.

In future work, DSVerifier will verify non-fragile and robust performance and support a wide range of dynamic systems, in addition to linear and SISO ones (*e.g.*, multiple-input multiple-output and non-linear systems), as well as other types of representation (*e.g.*, state space) and realization forms (*e.g.*, Rho-DFIIT realization form [80]). In addition, reliability of controllers obtained via non-fragile techniques will also be investigated. Thus, the proposed formal verification techniques will be applied to ensure correctness of fault diagnosis and fault tolerant control system design. Note that other features related to processing entities and implementation strategies could influence stability, such as cache and pipeline structures, which could be encoded as properties to be checked by DSVerifier and tackled in a more generic evaluation regarding worst-case execution time (WCET) analysis, in addition to FWL effects, but with the goal of checking closed-loop behavior maintenance. As a result, DSVerifier would be able to check processing capabilities, along with implementation strategies, which could lead to a generic framework for system verification and evaluation. Future versions of DSVerifier will support the linear fractional transform framework framework [81], in order to obtain a standard representation of control-loop configurations and uncertainty. Finally, we will also add a fixed-point format check to DSVerifier, with the goal of instantly suggesting representations suitable to a given system's coefficient and their inherent dynamic range, which has the potential to shorten the verification effort.

References

[1] S. Fadali, A. Visioli, Digital Control Engineering: Analysis and Design, Elsevier/Academic Press, 2012.

[2] S. Bhattacharyya, H. Chapellat, L. Keel, Robust Control: The Parametric Approach, Prentice Hall PTR, 1995. `doi:10.1016/S1474-6670(17)45891-5`.

[3] X. Zhao, L. Zhang, P. Shi, H. R. Karimi, Robust control of continuous-time systems with state-dependent uncertainties and its application to electronic circuits, IEEE Transactions on Industrial Electronics 61 (8) (2014) 4161–4170. `doi:10.1109/TIE.2013.2286568`.

[4] L. H. Keel, S. P. Bhattacharyya, Robust, fragile, or optimal?, IEEE Transactions on Automatic Control 42 (8) (1997) 1098–1105. `doi:10.1109/9.618239`.

[5] G. Yang, X. Guo, W. Che, W. Guan, Linear Systems: Non-Fragile Control and Filtering, Taylor & Francis, 2013.

[6] R. Sakthivel, R. Sakthivel, P. Selvaraj, H. R. Karimi, Delay-dependent fault-tolerant controller for time-delay systems with randomly occurring uncertainties, International Journal of Robust and Nonlinear Control 27 (18) (2017) 1–17. `doi:10.1002/rnc.3849`.

[7] M. Mahmoud, Resilient Control of Uncertain Dynamical Systems, Springer, 2004. `doi:10.1007/b95075`.

[8] G. Yang, D. Ye, Reliable Control and Filtering of Linear Systems with Adaptive Mechanisms, CRC Press, 2010.

[9] K. Åström, B. Wittenmark, Computer-controlled systems: theory and design, Prentice Hall, 1997.

[10] P. Tabuada, S. Caliskan, M. Rungger, R. Majumdar, Towards robustness for cyber-physical systems, IEEE Trans. on Automatic Control 59 (12) (2014) 3151–3163. `doi:10.1109/TAC.2014.2351632`.

[11] H. Ismail, I. Bessa, L. C. Cordeiro, E. B. de Lima Filho, J. E. C. Filho, Dsverifier: A bounded model checking tool for digital systems, in: Model Checking Software - 22^{nd} International Symposium, SPIN 2015, Stellenbosch, South Africa, August 24-26, 2015, Proceedings, Springer, Cham, 2015, pp. 126–131. `doi:10.1007/978-3-319-23404-5_9`.

[12] A. Anta, R. Majumdar, I. Saha, P. Tabuada, Automatic verification of control system implementations, in: Proceedings of the 10^{th} ACM International Conference on Embedded Software, ACM, New York, NY, USA, 2010, pp. 9–18. `doi:10.1145/1879021.1879024`.

[13] G. Simko, E. K. Jackson, A bounded model checking tool for periodic sample-hold systems, in: Proceedings of the 17^{th} International Conference on Hybrid Systems: Computation and Control, ACM, New York, NY, USA, 2014, pp. 157–162. `doi:10.1145/2562059.2562134`.

[14] A. David, G. Behrmann, K. Larsen, W. Yi, A tool architecture for the next generation of UPPAAL, Lecture Notes in Computer Science 2757 (2003) 352–366. `doi:10.1007/978-3-540-40007-3\_22`.

[15] L. De Moura, N. Bjørner, Satisfiability modulo theories: Introduction and applications, Commun. ACM 54 (9) (2011) 69–77. `doi:10.1145/1995376.1995394`.

[16] L. Cordeiro, B. Fischer, J. Marques-Silva, SMT-Based Bounded Model Checking for Embedded ANSI-C Software, IEEE Transaction on Software Engineering 38 (4) (2012) 957–974. `doi:10.1109/ASE.2009.63`.

[17] T. A. Davis, MATLAB Primer, CRC Press, 2010.

[18] M. Inc., Fixed-point designer, `http://www.mathworks.com/help/fixedpoint/index.html`, accessed: 2017-11-10 (2017).

[19] D. Kroening, J. Ouaknine, O. Strichman, T. Wahl, J. Worrell, Linear completeness thresholds for bounded model checking, in: Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings, Vol. 6806 of Lecture Notes in Computer Science, Springer, 2011, pp. 557–572.

[20] D. Kroening, M. Tautschnig, CBMC - C bounded model checker - (competition contribution), in: Tools and Algorithms for the Construction and Analysis of Systems - 20^{th} International Conference, TACAS 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014. Proceedings, Springer, Berlin, Heidelberg, 2014, pp. 389–391. `doi:10.1007/978-3-642-54862-8_26`.

[21] B. Dutertre, Yices 2.2, in: Computer Aided Verification - 26^{th} International Conference, CAV 2014 Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings, Springer, Cham, 2014, pp. 737–744. `doi:10.1007/978-3-319-08867-9_49`.

[22] A. Cimatti, A. Griggio, B. J. Schaafsma, R. Sebastiani, The mathsat5 SMT solver, in: Tools and Algorithms for the Construction and Analysis of Systems - 19^{th} 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, Springer, Berlin, Heidelberg, 2013, pp. 93–107. `doi:10.1007/978-3-642-36742-7_7`.

[23] C. Barrett, C. L. Conway, M. Deters, L. Hadarean, D. Jovanović, T. King, A. Reynolds, C. Tinelli, Cvc4, in: Proceedings of the 23rd International Conference on Computer Aided Verification, Springer-Verlag, Berlin, Heidelberg, 2011, pp. 171–177. doi:10.1007/978-3-642-22110-1_14.

[24] N. Eén, N. Sörensson, An extensible sat-solver, in: Theory and Applications of Satisfiability Testing, 6th International Conference, SAT 2003. Santa Margherita Ligure, Italy, May 5-8, 2003 Selected Revised Papers, Springer, Berlin, Heidelberg, 2003, pp. 502–518. doi:10.1007/978-3-540-24605-3_37.

[25] R. Brummayer, A. Biere, Boolector: An efficient smt solver for bit-vectors and arrays, in: Proceedings of the 15th International Conference on Tools and Algorithms for the Construction and Analysis of Systems: Held As Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009, Springer-Verlag, Berlin, Heidelberg, 2009, pp. 174–177. doi:10.1007/978-3-642-00768-2_16.

[26] L. De Moura, N. Bjørner, Z3: An Efficient SMT Solver, in: TACAS, Springer-Verlag, Berlin, Heidelberg, 2008, pp. 337–340. doi:10.1007/978-3-540-78800-3_24.

[27] H. F. Albuquerque, R. F. Araujo, I. V. de Bessa, L. C. Cordeiro, E. B. de Lima Filho, Optce: A counterexample-guided inductive optimization solver, in: SBMF, Vol. 10623 of LNCS, 2017, pp. 125–141. doi:10.1007/978-3-319-70848-5_9.

[28] D. Beyer, Software verification and verifiable witnesses, in: Proceedings of the 21st International Conference on Tools and Algorithms for the Construction and Analysis of Systems - Volume 9035, Springer-Verlag, New York, NY, USA, 2015, pp. 401–416. doi:10.1007/978-3-662-46681-0_31.

[29] R. Istepanian, J. Whidborne, Digital Controller Implementation and Fragility: A Modern Perspective, Advances in Industrial Control, Springer London, 2001.

[30] L. Jackson, J. Kaiser, H. McDonald, An approach to the implementation of digital filters, IEEE Trans. Audio Electroacoust 34 (3) (1968) 413–421.

[31] I. V. Bessa, H. I. Ismail, L. C. Cordeiro, J. E. Filho, Verification of fixed-point digital controllers using direct and delta forms realizations, Des. Autom. Embedded Syst. 20 (2) (2016) 95–126. doi:10.1007/s10617-016-9173-5.

[32] I. Bessa, H. Ismail, R. M. Palhares, L. C. Cordeiro, J. E. C. Filho, Formal non-fragile stability verification of digital control systems with uncertainty, IEEE Trans. Computers 66 (3) (2017) 545–552. doi:10.1109/TC.2016.2601328.

[33] B. Widrow, I. Kollár, Quantization Noise: Roundoff Error in Digital Computation, Signal Processing, Control, and Communications, Cambridge, 2008.

[34] R. Alur, Principles of Cyber-Physical Systems, The MIT Press, 2015.

[35] A. Zutshi, S. Sankaranarayanan, J. V. Deshmukh, X. Jin, Symbolic-numeric reachability analysis of closed-loop control software, in: Proceedings of the 19^{th} International Conference on Hybrid Systems: Computation and Control, ACM, New York, NY, USA, 2016, pp. 135–144. doi:10.1145/2883817.2883819.

[36] R. Majumdar, I. Saha, K. Shashidhar, Z. Wang, CLSE: Closed-loop symbolic execution, Proc. 4^{th} Int'l Symposium on NASA Formal Methods, LNCS 7226 (2012) 356–370. doi:10.1007/978-3-642-28891-3_33.

[37] M. Rungger, P. Tabuada, A notion of robustness for cyber-physical systems, IEEE Transactions on Automatic Control 61 (8) (2016) 2108–2123. doi:10.1109/TAC.2015.2492438.

[38] B. Chou, T. Erkkinen, Converting models from floating point to fixed point for production code generation, Matlab Digest, accessed: 2018-03-21 (2008).

[39] L. Keel, S. Bhattacharyya, Stability margins and digital implementation of controllers, in: Proc. American Control Conference, Springer, London, 1998, pp. 2852–2856. doi:10.1109/ACC.1998.688377.

[40] K. Vorobyov, P. Krishnan, Comparing model checking and static program analysis: A case study in error detection approaches, in: Proc. Int'l Work. on Systems Software Verification, IEEE, Vancouver, Canada, 2010, pp. 1–7.

[41] P. Bauch, V. Havel, J. Barnat, Accelerating temporal verification of simulink diagrams using satisfiability modulo theories, Software Quality Journal 24 (1) (2014) 37–63. doi:10.1007/s11219-014-9259-x.

[42] J. Barnat, P. Bauch, V. Havel, Temporal verification of simulink diagrams, in: Proc. Int'l Symp. on High-Assurance Systems Engineering, IEEE, Danvers, MA, 2014, pp. 81–88. doi:10.1109/HASE.2014.20.

[43] A. Abate, I. Bessa, D. Cattaruzza, L. C. Cordeiro, C. David, P. Kesseli, D. Kroening, E. Polgreen, Automated formal synthesis of digital controllers for state-space physical plants, in: Computer Aided Verification - 29^{th} International Conference, CAV 2017, Heidelberg, Germany, July 24-28, 2017, Proceedings, Part I, Springer, Cham, 2017, pp. 462–482. doi:10.1007/978-3-319-63387-9_23.

[44] A. Abate, I. Bessa, D. Cattaruzza, L. C. Cordeiro, C. David, P. Kesseli, D. Kroening, Sound and automated synthesis of digital stabilizing controllers for continuous plants, in: Proceedings of the 20^{th} International Conference on Hybrid Systems: Computation and Control, HSCC 2017, Pittsburgh, PA, USA, April 18-20, 2017, ACM, New York, NY, USA, 2017, pp. 197–206. doi:10.1145/3049797.3049802.

[45] A. Abate, I. Bessa, D. Cattaruzza, L. Chaves, L. C. Cordeiro, C. David, P. Kesseli, D. Kroening, E. Polgreen, Dssynth: an automated digital controller synthesis tool for physical plants, in: Proceedings of the 32^{nd} IEEE/ACM International Conference on Automated Software Engineering, ASE 2017, Urbana, IL, USA, October 30 - November 03, 2017, IEEE, Danvers, MA, 2017, pp. 919–924. doi:10.1109/ASE.2017.8115705.

[46] A. Solar-Lezama, Program sketching, STTT 15 (5-6) (2013) 475–495. doi:10.1007/s10009-012-0249-7.

[47] F. R. Monteiro, Bounded model checking of state-space digital systems: The impact of finite word-length effects on the implementation of fixed-point digital controllers based on state-space modeling, in: Proceedings of the 2016 24^{th} ACM SIGSOFT International Symposium on Foundations of Software Engineering, ACM, New York, NY, USA, 2016, pp. 1151–1153. doi:10.1145/2950290.2983979.

[48] M. Gevers, G. Li, Parametrizations in control, estimation, and filtering problems: accuracy aspects, Communications and control engineering series, Springer-Verlag, 1993.

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

[50] T. Hilaire, P. Chevrel, J. F. Whidborne, Finite wordlength controller realisations using the specialised implicit form, International Journal of Control 83 (2) (2010) 330–346. doi:10.1080/00207170903160747.

[51] R. Sakthivel, C. Wang, S. Santra, B. Kaviarasan, Non-fragile reliable sampled-data controller for nonlinear switched time-varying systems, Nonlinear Analysis: Hybrid Systems 27 (2018) 62–76. doi:10.1016/j.nahs.2017.08.005.

[52] T. Hilaire, A. Volkova, Error analysis methods for the fixed-point implementation of linear systems, in: IEEE Workshop on Signal Pro- cessing Systems (SiPS), IEEE, Pacific Grove, CA United States, 2017, pp. 1–1. doi:10.1109/ACSSC.2015.7421231.

[53] A. Volkova, T. Hilaire, C. Lauter, Determining fixed-point formats for a digital filter implementation using the worst-case peak gain measure, in: 2015 49^{th} Asilomar Conference on Signals, Systems and Computers, IEEE, Pacific Grove, CA United States, 2015, pp. 737–741. doi:10.1109/ACSSC.2015.7421231.

[54] E. Feron, From control systems to control software, IEEE Control Systems 30 (6) (2010) 50–71. doi:10.1109/MCS.2010.938196.

[55] IEEE Computer Society, IEEE standard for floating-point arithmetic, IEEE Std 754-2008, Aug 2008.

[56] P. Diniz, S. Netto, E. D. Silva, Digital Signal Processing: System Analysis and Design, Cambridge University Press, 2002. doi:10.1017/CBO9780511781667.

[57] W. T. Padgett, D. V. Anderson, Fixed-Point Signal Processing, Morgan and Claypool Publishers, 2009.

[58] M. Joldes, J. Muller, V. Popescu, W. Tucker, CAMPARY: cuda multiple precision arithmetic library and applications, in: Mathematical Software - ICMS 2016 - 5th International Conference, Berlin, Germany, July 11-14, 2016, Proceedings, 2016, pp. 232–240.

[59] P. S. Duggirala, M. Viswanathan, Analyzing real time linear control systems using software verification, in: 2015 IEEE Real-Time Systems Symposium, IEEE Press, Piscataway, NJ, USA, 2015, pp. 216–226. doi:10.1109/RTSS.2015.28.

[60] M. Y. R. Gadelha, L. C. Cordeiro, D. A. Nicole, Encoding floating-point numbers using the SMT theory in ESBMC: an empirical evaluation over the SV-COMP benchmarks, in: SBMF, Vol. 10623 of Lecture Notes in Computer Science, 2017, pp. 91–106. doi:10.1007/978-3-319-70848-5_7.

[61] M. Y. R. Gadelha, H. I. Ismail, L. C. Cordeiro, Handling loops in bounded model checking of C programs via k-induction, STTT 19 (1) (2017) 97–114. doi:10.1007/s10009-015-0407-9.

[62] W. Rocha, H. Rocha, H. Ismail, L. C. Cordeiro, B. Fischer, Depthk: A k-induction verifier based on invariant inference for C programs - (competition contribution), in: Tools and Algorithms for the Construction and Analysis of Systems - 23^{rd} International Conference, TACAS 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings, Part II, Springer, Berlin, Heidelberg, 2017, pp. 360–364. doi:10.1007/978-3-662-54580-5_23.

[63] A. R. Bradley, Z. Manna, The Calculus of Computation: Decision Procedures with Applications to Verification, Springer-Verlag New York, Inc., 2007. doi:10.1007/978-3-540-74113-8.

[64] S. Bouabdallah, P. Murrieri, R. Siegwart, Design and Control of an Indoor Micro Quadrotor, in: ICRA, IEEE, Danvers, MA, 2004, pp. 4393–4398. doi:10.1109/ROBOT.2004.1302409.

[65] L. Chaves, I. Bessa, L. Cordeiro, D. Kroening, Dsvalidator: An automated counterexample reproducibility tool for digital systems, in: Proceedings of the 21st International Conference on Hybrid Systems: Computation and Control (Part of CPS Week), HSCC '18, ACM, New York, NY, USA, 2018, pp. 253–258. doi:10.1145/3178126.3178151.

[66] K. J. Astrom, R. M. Murray, Feedback Systems: An Introduction for Scientists and Engineers, Princeton University Press, 2008.

[67] Y. Li, K. Ang, G. Chong, W. Feng, K. Tan, H. Kashiwagi, CAutoCSD–evolutionary search and optimisation enabled computer automated control system design, Int J Automat Comput 1 (1) (2004) 76–88. doi:10.1007/s11633-004-0076-8.

[68] K. Tan, Y. Li, Performance-based control system design automation via evolutionary computing, Engineering Applications of Artificial Intelligence 14 (4) (2001) 473–486. doi:10.1016/S0952-1976(01)00023-9.

[69] J. C. Doyle, B. A. Francis, A. R. Tannenbaum, Feedback Control Theory, Prentice Hall Professional Technical Reference, 1991.

[70] G. F. Franklin, D. J. Powell, A. Emami-Naeini, Feedback Control of Dynamic Systems, Prentice Hall PTR, 2001.

[71] T. E. Wang, P.-L. Garoche, P. Roux, R. Jobredeaux, E. Féron, Formal analysis of robustness at model and code level, in: Proceedings of the 19th International Conference on Hybrid Systems: Computation and Control, ACM, New York, NY, USA, 2016, pp. 125–134. doi:10.1145/2883817.2883824.

[72] D. B. Williams, V. Madisetti, Digital Signal Processing Handbook, CRC Press, Inc., 1997.

[73] D. Kroening, J. Ouaknine, O. Strichman, T. Wahl, J. Worrell, Linear Completeness Thresholds For Bounded Model Checking, in: CAV, Vol. 6806 of LNCS, 2011, pp. 557–572.

[74] W. Rocha, H. Rocha, H. Ismail, L. C. Cordeiro, B. Fischer, DepthK: A k-Induction Verifier Based on Invariant Inference for C Programs - (Competition Contribution), in: TACAS, Vol. 10206 of LNCS, 2017, pp. 360–364.

[75] B. Wachter, D. Kroening, J. Ouaknine, Verifying multi-threaded software with impact, in: Formal Methods in Computer-Aided Design, FMCAD 2013, Portland, OR, USA, October 20-23, 2013, IEEE, 2013, pp. 210–217.

[76] K. L. McMillan, Interpolation and sat-based model checking, in: Computer Aided Verification, 15th International Conference, CAV 2003, Boulder, CO, USA, July 8-12, 2003, Proceedings, Vol. 2725 of Lecture Notes in Computer Science, Springer, 2003, pp. 1–13.

[77] D. Ishii, K. Ueda, H. Hosobe, An interval-based sat modulo ode solver for model checking nonlinear hybrid systems, International Journal on Software Tools for Technology Transfer 13 (5) (2011) 449–461. doi:10.1007/s10009-011-0193-y.

[78] N. Ramdani, N. Nedialkov, Computing reachable sets for uncertain nonlinear hybrid systems using interval constraint-propagation techniques, Nonlinear Analysis: Hybrid Systems 5 (2) (2011) 149–162. doi:10.1016/j.nahs.2010.05.010.

[79] D. Beyer, Status report on software verification (competition summary SV-COMP 2014), LNCS 8413 (2014) 373–388. doi:10.1.1.672.8243.

[80] G. Li, C. Wan, G. Bi, An improved /spl rho/dfiit structure for digital filters with minimum roundoff noise, IEEE Transactions on Circuits and Systems II: Express Briefs 52 (4) (2005) 199–203. doi:10.1109/TCSII.2004.842416.

[81] K. Zhou, J. Doyle, K. Glover, Robust and Optimal Control, Prentice Hall, 1996.

Appendix A

Table 10: Set of closed-loop systems (*i.e.*, benchmarks) used in our experimental evaluation. Here, C_n represents a controller and G_n a plant, regarding a chosen closed-loop system H_n.

Id	Closed-Loop system
H_1	$C_1 = \frac{0.00390625z+0.0009765625}{0.3134765625z-0.0009765625}$
	$G_1 = \frac{9.7541e^{-4}}{z-0.9512}$
H_2	$C_2 = \frac{-0.3466796875z+0.015625}{0.5z^2+0.19921875z}$
	$G_2 = \frac{0.1898z+1.8027e^{-4}}{z^2-0.9012z-1.0006e^{-16}}$
H_3	$C_3 = \frac{0.5z^3-0.96875z^2-0.875z-0.5}{0.001190185546875z^8+0.0008544921875z^7+0.000152587890625z^6+0.000335693359375z^5}$
	$G_3 = \frac{0.0001929z+6.814e^{-9}}{z^8-0.6921z^7}$
H_4	$C_4 = \frac{-0.580535888671875z^3+0.919769287109375z^2+0.11871337890625z-0.951934814453125}{0.7188720703125z^3-0.38751220703125z^2-0.415924072265625z+0.437286376953125}$
	$G_4 = \frac{-0.01285z^2 0.02582z-0.01293}{z^3-2.99z^2+2.983z-0.9929}$
H_5	$C_5 = \frac{-0.0009765625z^2}{0.76171875z^3}$
	$G_5 = \frac{15.1315z^2+17.8600z+17.4549}{z^3-2.6207z^2+2.3586z-0.6570}$
H_6	$C_6 = \frac{-0.96484375z+0.9833984375}{0.8896484375z^2-0.875z}$
	$G_6 = \frac{0.2039z+0.2039}{z^2+1.19999z+1.0}$
H_7	$C_7 = \frac{0.8359375z^2+0.265625z-0.96875}{0.9453125z^3+0.90625z^2-0.15625z-0.123046875}$
	$G_7 = \frac{1.25e^{-1}z+1.25e^{-1}}{z^2-2z+1}$
H_8	$C_8 = \frac{-4.6566128730773925781 25e^{-10}z^2+1.00000000046566128730773925781 25z-1.00000000046566128730773925781 25}{z^2-0.46566128730773925781 25e^{-9}z+0.46566128730773925781 25e^{-9}}$
	$G_8 = \frac{5.0e^{-5}z+5.0e^{-5}}{z^2-2z+1}$
H_9	$C_9 = \frac{-0.0224609375z^3}{0.138671875z^4}$
	$G_9 = \frac{0.25z^3+0.5z^2+0.25z-4.3341e^{-7}}{z^4+5.9150e^{-6}z^3+1.0480e^{-11}z^2-4.9349e^{-63}z+7.0168e^{-225}}$
H_{10}	$C_{10} = \frac{0.0625z}{0.517578125z^2-0.4990234375z}$
	$G_{10} = \frac{0.0200z-3.8303e^{-176}}{z-4.6764e^{-166}}$
H_{11}	$C_{11} = \frac{-4.4366z^6+9.177z^5-3.6362z^4-5.1444z^3+5.9167z^2-2.2791z+0.31329}{-0.23339z^5-1.5195z^4+0.73999z^3+0.51029z^2-0.41403z+0.073294}$
	$G_{11} = \frac{0.54869z-0.88674}{z^2-3.3248z+1.6487}$ -
H_{12}	$C_{12} = \frac{11.9255z-11.8089}{z-1.0729}$
	$G_{12} = \frac{0.01z-0.010101}{z^2-2.0103z+1.0101}$
H_{13}	$C_{13} = \frac{-2.7056z^3+4.9189z^2-2.9898z+0.60746}{z^3-0.24695z^2-0.80001z+0.35681}$
	$G_{13} = \frac{0.33528z-0.55879}{z^2-1.8906z+0.7788}$
H_{14}	$C_{14} = \frac{-45456.4327z^7+37928.1361z^6+25543.7663z^5-38701.0881z^4+16110.0087z^3-2847.8579z^2+182.2326z+0.38487}{z^7 0.47737z^6-1.4922z^5-0.6236z^4+0.64615z^3+0.10413z^2-0.12437z+0.018243}$
	$G_{14} = \frac{-0.0001492z^3-0.00051649z-7.2373e^{-5}}{z^4-7.8381z^2+2.9258z-0.25393}$