DSValidator: An Automated Counterexample Reproducibility Tool for Digital Systems

Lennon Chaves, Iury Bessa
Federal University of Amazonas
Manaus, Brazil
lennonchaves@ufam.edu.br
iurybessa@ufam.edu.br

Lucas Cordeiro, Daniel Kroening
University of Oxford
Oxford, United Kingdom
lucas.cordeiro@cs.ox.ac.uk
kroening@cs.ox.ac.uk

ABSTRACT

We present an automated counterexample reproducibility tool based on MATLAB, called DSValidator, with the goal of reproducing counterexamples that refute specific properties related to digital systems. We exploit counterexamples generated by the Digital System Verifier (DSVerifier), which is a model checking tool based on satisfiability modulo theories for digital systems. DSValidator reproduces the execution of a digital system, relating its input with the counterexample, in order to establish trust in a verification result. We show that DSValidator can validate a set of intricate counterexamples for digital controllers used in a real quadrotor attitude system within seconds and also expose incorrect verification results in DSVerifier. The resulting toolbox leverages the potential of combining different verification tools for validating digital systems via an exchangeable counterexample format.

KEYWORDS

Model Checking; Digital Systems; MATLAB.

ACM Reference Format:

Lennon Chaves, Iury Bessa and Lucas Cordeiro, Daniel Kroening. 2018. DSValidator: An Automated Counterexample Reproducibility Tool for Digital Systems. In *HSCC '18: 21st International Conference on Hybrid Systems: Computation and Control (part of CPS Week), April 11–13, 2018, Porto, Portugal.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3178126.3178151

1 INTRODUCTION

Digital systems (e.g., filters and controllers) consist of a mathematical operator that maps one signal to another signal using a fixed set of operations [12]; they are used in a wide range of applications owing to advantages over their analog counterparts, such as reliability, flexibility and cost. However, digital systems have disadvantages: since they are typically implemented in microprocessors, errors might be introduced by quantization and round-off. The choice of hardware, the realization (e.g., delta and direct forms) and implementation aspects (e.g., the number of bits, usage of fixed-point arithmetic) have impact on the precision and performance of the digital system [3].

To detect errors in a digital system implementation, considering finite word-length (FWL) effects [13, 16], we have proposed a model checking procedure named *Digital System Verifier* (DSVerifier) [15]. It verifies digital filters and controllers given as transfer functions or state-space equations [1, 3, 4, 9, 18]. DSVerifier checks properties related to overflow, limit cycle, stability and minimum-phase; it also supports a variety of digital system realizations and numerical formats. If DSVerifier finds a property violation, then it produces a counterexample, i.e., a sequence of states that leads to the error state. The challenge to reproduce the counterexample provided by verifiers is justified by the complexity to compute the output and internal states of a digital system; in particular, depending on the counterexample size, the debugging process might consume too much time to be manually inspected by engineers.

There are already several toolboxes in MATLAB that support digital system design [17]. For instance, the *Fixed-Point Designer Toolbox* provides data-types and tools for developing fixed-point digital systems. There are further toolboxes with different objectives, e.g., optimization, design of control systems and digital signal processing [17]. However, there is no toolbox to reproduce counterexamples in digital systems generated by verifiers, i.e., to automatically reproduce a sequence of states that refutes a specific property with the goal of establishing trust in a verification result.

The closest academic work to ours are verifiers that validate counterexamples using the witness validation approach, which reproduces the verification results by checking a counterexample given in the graphml format [5]. For instance, CPAchecker [6] and Ultimate Automizer [14] employ the error witnesses to avoid false alarms produced by static analyzers, i.e., given a witness for a problematic program path, they re-verify that the witness indeed certifies that the specification is violated. However, those tools are unable to support the validation of systems that require fixed-point arithmetic, and consequently disregard FWL effects, which is needed to successfully validate implementation-level properties of typical digital systems.

Contributions. This paper presents and evaluates DSValidator, a novel MATLAB toolbox that automatically checks whether a counterexample given by a verifier is reproducible. We propose a format to represent the counterexamples, that can be used by CBMC [10] and ESBMC [11], which are used as back-end in DSVerifier. Here, a counterexample provides assignments to the digital system's variables. This counterexample allows us to reproduce the failed property, providing the concrete, low-level details that are needed to simulate the digital system in MATLAB. DSValidator is able to validate counterexamples related to overflow, limit-cycle, stability and minimum-phase. We show that DSValidator is able to reproduce a set of intricate counterexamples for digital controllers

Supported by EPSRC grant EP/J012564/1, ERC project 280053 (CPROVER) and the H2020 FET OPEN 712689 SC^2.

HSCC '18, April 11–13, 2018, Porto, Portugal

ACM ISBN 978-1-4503-5642-8/18/04.
https://doi.org/10.1145/3178126.3178151

used in a real quadrotor attitude control system within seconds. DSValidator is also able to expose incorrect verification results in DSVerifier caused by wrong computation of the system output.

Availability of Data and Tools. The experiments described in this paper are based on a set of publicly available benchmarks. All tools, benchmarks, videos and results of our evaluation are available on a supplementary web page `http://dsverifier.org/`. In particular, the source code of DSValidator is available in a public repository located at https://github.com/ssvlab/dsverifier/tree/master/toolbox-dsvalidator.

2 DSVALIDATOR DIGITAL SYSTEM REPRODUCIBILITY ENGINE

DSValidator is able to simulate digital controllers and filters considering implementation features (e.g., FWL effects and realizations) by replaying a given counterexample provided by a verifier.

2.1 Representation of the Digital System

DSValidator supports digital systems (digital controllers and filters), represented by transfer functions, i.e., frequency domain equations that are able to represent input-to-output relations in a digital system. The following expression presents the general form of a digital system transfer function:

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

where z^{-1} is called backward-shift operator; $A(z)$ and $B(z)$ are the denominator and numerator polynomials; and N and M represent the denominator and numerator polynomials order, respectively. Another representation is the difference equation, which can be described as

$$y(n) = -\sum_{k=1}^{N} a_k y(n-k) + \sum_{k=0}^{M} b_k x(n-k). \tag{2}$$

Eq. (2) allows DSValidator to compute the system output $y(n)$ at the n-th instant (i.e., at time $t = n \cdot T$, where T is the system sample time) using values of the past outputs and the present and past inputs, i.e., $x(n)$.

2.2 Properties and their Counterexamples

2.2.1 Stability and Minimum-phase. A digital system is stable *iff* all of its poles are inside the z-plane unitary circle; poles must have the modulus less than one. Minimum-phase is also a desirable property for digital systems. A digital system is a minimum-phase system *iff* all of its zeros are inside the z-plane unitary circle. The counterexample reproducibility for both minimum-phase and stability does not require DSValidator to compute output and states since polynomial analysis is performed, but FWL effects over the coefficients of Eq. (1) must be computed.

Definition 1. ***(Finite Word-Length)*** *$\mathcal{FWL}[\cdot] : \mathbb{R}^{N+M+2} \to Q[\mathbb{R}^{N+M+2}]$ function applies FWL effects to a polynomial vector representation, where $Q[\mathbb{R}]$ represents the quantized set of representable real numbers in the chosen implementation format.*

Definition 2. ***(Roots of a Polynomial)*** *$\mathcal{R}[\cdot] : \mathbb{R}^{N+M+2} \to \mathfrak{S}$ function computes the set of roots of a polynomial, and $\mathfrak{S}$ is a family of sets. The poles of Eq. (1) is computed by $\mathcal{R}[A(z)]$, and the zeros are computed by $\mathcal{R}[B(z)]$.*

Definition 3. ***(Stability Reproducibility)*** DSValidator *computes the $\mathcal{FWL}[A(z)]$ roots for stability reproduction. If any root has modulus equal or greater than one, then the system is unstable; otherwise, it is stable.*

Definition 4. ***(Minimum-phase Reproducibility)*** DSValidator *computes the $\mathcal{FWL}[B(z)]$ roots for minimum-phase reproduction. If any root has modulus equal or greater than one, then the system is non minimum-phase; otherwise, it is minimum-phase.*

2.2.2 Overflow. When an operation result exceeds the limited range of the processor's word-length, overflow might occur in the output of the digital system realization, resulting in undesirable nonlinearities in the output. In order to reproduce an overflow counterexample, the output sequence must be computed for the given input sequence; the counterexample should contain an input sequence $x(n)$ that leads the digital system to overflow. DSValidator reads the counterexample provided by a given verifier, and then computes FWL effects over the coefficients, i.e., DSValidator computes $\mathcal{FWL}[A(z)]$ and $\mathcal{FWL}[B(z)]$ (cf. Definition 1).

After that, DSValidator checks the word-length representation limits, considering n-integer bits and l-fractional bits. The maximum representable value is computed as $2^{n-1} - 2^{-l}$ and the minimum representable value is computed as -2^{n-1}. Then, Eq. (2) is iteratively unrolled for a given realization form, considering the input $x(n)$ (from the counterexample) to produce the output $y(n)$.

Definition 5. ***(Realization Form)*** *A realization form represents a template to implement a given digital system in software by using directly the coefficients of Eq. (1) in its implementation [3, 13, 16].*

Definition 6. ***(Overflow reproducibility)*** DSValidator *checks whether each system's output is inside the word-length representation limits; the output does not lead to overflow if $-2^{n-1} < y(n) < 2^{n-1} - 2^{-l}$ is inside the word-length limits. A detected overflow violation must be similar to the counterexample indicated by the verifier; otherwise, the counterexample is not reproducible.*

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

Definition 7. ***(Granular LCO)*** *Granular limit cycles are autonomous oscillations due to round-offs in the least significant bits [12].*

Definition 8. ***(Overflow LCO)*** *Overflow limit cycles appear when an operation results in overflow using the wrap-around mode [12].*

To reproduce LCO counterexamples, constant inputs and initial states are used as test signals in DSValidator to compute the output sequence $y(n)$, considering a given realization form (cf. Definition 5). First, DSValidator obtains FWL effects on the numerator and denominator coefficients (cf. Definition 1). The constant input, initial states and realization form are provided by a given counterexample and employed to compute $y(n)$ based on the fixed-point arithmetic and also to simulate the respective digital system in MATLAB.

Definition 9. ***(LCO reproducibility)*** *If the system's output $y(n)$ provided by* DSValidator *leads to oscillations in the output with the same characteristics (i.e., amplitude and period) from that indicated by the verifier, then the LCO counterexample is reproducible; otherwise, the verifier presents an error.*

In order to confirm the LCO absence, the algorithm proposed by Bauer [2, 19] was implemented in DSValidator. The aim of that algorithm is to exhaustively search for the absence of limit cycle; it is applicable to all direct form realizations, besides being independent on the quantization and system order. Therefore, Bauer's method decides about the asymptotic stability of (linearly stable) digital systems, by employing an exhaustive search method. If it detects that a digital system is asymptotic stable, then the latter is limit cycle free; otherwise, it is susceptible to overflow or granular LCO.

3 AUTOMATED COUNTEREXAMPLE REPRODUCIBILITY FOR DIGITAL SYSTEMS

3.1 Proposed Counterexample Format

CBMC [10] and ESBMC [11] construct counterexamples whether a property violation is found. A counterexamples is a trace that shows that a given property does not hold in the model. Counterexamples allow the user: (*i*) to analyze a failure, (*ii*) to understand an error, and (*iii*) to correct either the respective specification or model, in this case, from the property and the program that has been analyzed [20].

Definition 10. ***(Counterexample)*** *A counterexample for a property ϕ is a sequence of states $s_0, s_1, \ldots, s_k$ with $s_0 \in S_0$ (initial state), $s_k \in S$ (bad state), and $\gamma(s_i, s_{i+1})$ for $0 \leq i < k$, that refutes ϕ.*

If it is unsatisfiable (*i.e.*, returns $false$), then one can conclude there is no error state in k steps or less. Finally, we can define $\psi_k = \bigwedge_{\pi} \psi_k^{\pi}$ and use it to check all paths.

For our current work, DSValidator exploits counterexamples provided by verifiers [10, 11, 15]; if there is a property violation, then the verifier provides a counterexample, which contains inputs and initial states that lead the digital system to a failure state. Fig. 1 shows an example of the present counterexample format related to an overflow LCO violation for the digital system represented by Eq. (3):

$$H(z) = \frac{2002 - 4000z^{-1} + 1998z^{-2}}{1 - z^{-2}}. \tag{3}$$

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

Figure 1: Proposed Counterexample Format Example.

The proposed counterexample format shown in Fig. 1 describes the violated property (represented by a *string*), transfer function numerator and denominator (represented by *fixed-point numbers*), bound (represented by an *integer*), sample time (represented by a *fixed-point number*), implementation aspects (integer and fractional bits represented by an *integer*), realization form (represented by a *string*), dynamical range (represented by an *integer*), initial states, inputs, and outputs (which are represented by *fixed-point numbers*). In particular, the counterexample provides the needed data to reproduce a given property violation via simulation in MATLAB.

DSValidator considers ".out" files to extract the counterexample and to transform them into MATLAB variables; those ".out" files are generated by the verifier after the digital system verification is performed. Currently, DSValidator is able to validate the minimum-phase, overflow, stability and limit-cycle properties for a digital system that is represented by a transfer function. Additionally, DSValidator is able to employ 6 direct and delta realization forms for digital systems: direct form I (DFI), direct form II (DFII), transposed direct form II (TDFII), delta direct form I (DDFI), delta direct form II (DDFII) and transposed delta direct form II (TDDFII) [3].

3.2 Automated Counterexample Validation

There are five steps to automatically perform the automated counterexample validation in DSValidator, as can be visualized in Fig. 2.

In step (1), DSValidator obtains the counterexample and then uses a shell script to extract the data related to the digital system, i.e., property, transfer function numerator and denominator, fixed-point representation, k-bound, sample time, implementation aspects, realization form, dynamical range, initial states, inputs and outputs. In step (2), DSValidator converts all counterexample attributes into variables that can be manipulated in MATLAB. In step (3), DSValidator simulates the counterexample (violation) for the failed property, which is derived from the counterexample by providing concrete, lower-level details needed to simulate the digital system in MATLAB. In this specific step, all FWL effects and quantizations are applied to the digital system and to every arithmetic operation to compute the outputs with FWL effects, considering realization form and the desired property, as previously mentioned (subsection 2.2). In step (4), DSValidator compares the result between the output provided by the verifier and that simulated by MATLAB. Finally, in step (5), DSValidator stores the extracted counterexample in a `.MAT` file and then reports its reproducibility.

3.3 DSValidator Features

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

- **Macro Functions:** functions to reproduce the validation steps (e.g., parsing, simulation, comparison and report).
- **Validation Functions:** check and validate a violated property (e.g., overflow, limit-cycle, stability and minimum-phase).
- **Realizations:** reproduce realizations forms to validate overflow and limit-cycle (for direct and delta forms).
- **Numerical Functions:** perform the quantization process; select rounding mode (nearest, floor and ceil) and overflow mode (wrap-around and saturate); fixed-point operations (e.g., sum, subtraction, multiplication); and delta operator.
- **Graphic Functions:** plot the graphical representation of overflow to show each output exceeding the supported word-length limits; limit-cycle to represent the system's output

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

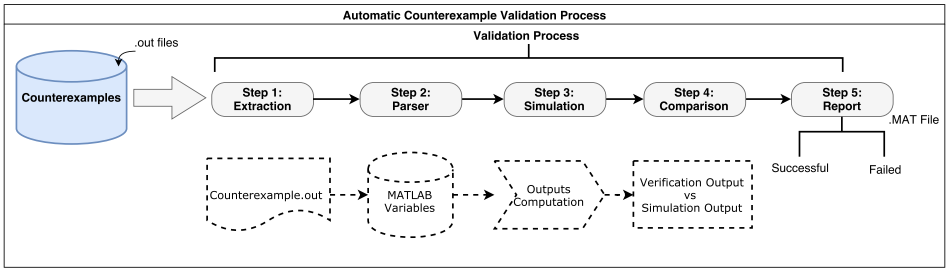

Figure 2: Automatic Counterexample Validation Process.

oscillations; and poles/zeros to show stability and minimum-phase with (or without) FWL effects inside a unitary circle.

3.4 DSValidator Result

The DSValidator result is structured with counterexample data composed by attributes and classes as shown in Fig. 3. The attributes are defined in the `.MAT` file with the following structure: *counterexample* that represents the counterexample identification; *digital system* that represents the numerator, denominator, and transfer function representation; *inputs* that represent the input vector and initial states; *implementation* that represents the integer and fractional bits, dynamical ranges, delta operator, sample time, bound and realization form; *outputs* report the verification and simulation results, execution time in MATLAB and comparison status, where it reports whether the counterexample is reproducible or not. Importantly, all execution times are actually CPU times, i.e., only the elapsed time periods spent in the allocated CPUs, which is measured with the `times` system call (POSIX system).

3.5 DSValidator Usage

DSValidator is called via the command line in MATLAB as follows:

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

where

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

After executing the `validation` command, DSValidator prints statistics about the counterexamples validation. Fig. 4 shows a report about the digital system represented in Eq. (3) for realizations DFI, DFII and TDFII.

4 CASE STUDY: DIGITAL CONTROLLERS FOR UAVS

4.1 Description of the Benchmarks

We evaluated DSValidator counterexamples reproducibility for a set of 11 digital controllers extracted from a real quadrotor unmanned aerial vehicle (UAV) [7], as shown in Table 1. These UAV attitude controllers were designed through four tasks, for each angle dynamics (pitch, roll and yaw): angle-dynamics modeling, selection and design of associated structures, coefficient tuning and controller discretization. The experiments evaluate overflow, minimum-phase, stability and limit-cycle in 33 different numerical formats: 3 for each digital controller, using 3 different realizations forms (i.e., DFI, DFII and TDFII), which resulted in 99 different verification tasks for each property (396 verification tasks in total).

Table 1: Digital controllers for the evaluated quadrotor attitude system.

Controller ID	Sample Time (ms)	Discrete Transfer Function
C_1	20	$\frac{1.5z-0.5}{z}$
C_2	20	$\frac{60z-50}{z}$
C_3	20	$\frac{110z-100}{z}$
C_4	20	$\frac{135z^2-260z+125}{z^2-z}$
C_5	1	$\frac{2002z^2-4000z+1998}{z^2-z}$
C_6	20	$\frac{0.93z-0.87}{z+1}$
C_7	20	$\frac{0.1z-0.09998}{z-1}$
C_8	2	$\frac{0.0096z-0.009}{0.002z}$
C_9	2	$\frac{0.1z-0.1}{z-1}$
C_{10}	20	$\frac{0.009z-0.0084}{z}$
C_{11}	20	$\frac{0.1z-0.09996}{z-1}$

The chosen number of bits, associated to each implementation, is based on the methodology presented by Carletta et al., which suggests a computation based on the impulse response sum [8]. All implementations and realizations used in the experiments are available online[2].

[2]DSValidator, benchmarks, and results are available at www.dsverifier.org

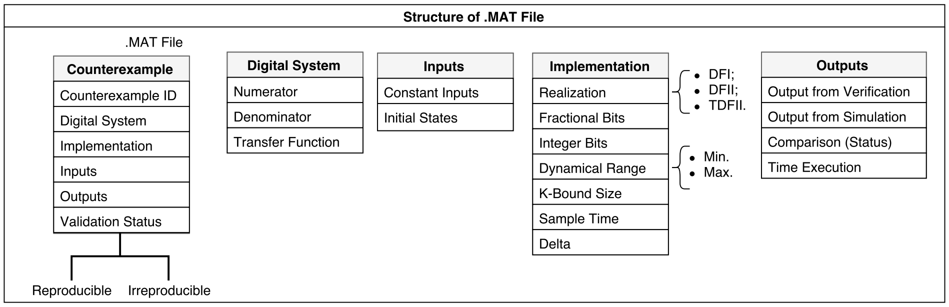

Figure 3: Structure of the `.MAT` file for representing counterexamples.

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

Figure 4: Counterexample Reproducibility Report.

4.2 Experimental Setup

For all tested digital systems implementation, we generated a set of ".out" files containing the counterexamples and the verification results (i.e., `successful` and `failed`). For all tested implementations, the signal input range lies between −1 and 1, that is, the sensor (gyroscope) output bound in normal conditions. Using this configuration for the signal input range, inputs employed during the verification of limit-cycle or overflow violations is limited between −1 and 1.

All experiments with DSValidator v1.0.1 were conducted on an otherwise idle Intel Core i7 − 2600 3.40 GHz processor, with 24 GB of RAM, running Ubuntu 64-bits. Importantly, all presented execution times are actually CPU times, i.e., only the elapsed time periods spent in the allocated CPUs, which was measured with the `times` system call (POSIX system). Additionally, the time presented in Table 2 is related to the average of 20 executions for each benchmark; the measuring unit is always in seconds based on the CPU time.

4.3 Experimental Objectives

DSValidator was employed to verify the soundness and the reliability of the verification results generated by the DSVerifier tool. Our experimental evaluation aims to answer two research questions:

RQ1 **(performance)** do the executable test cases take considerably less effort than verification, i.e., it produces a considerably smaller state space?

RQ2 **(sanity check)** are the counterexamples sound and can their reproducibility be confirmed outside of the employed verifier?

4.4 Experimental Results

According to Table 2, DSVerifier [15] produced 54 counterexamples for minimum-phase, 54 for stability, 27 for limit-cycle, and 24 for overflow (159 counterexamples in total). Table 2 shows the DSValidator results for the quadrotor attitude system digital controllers, where "Property" describes the property that is evaluated by DSValidator, "CE Reproducible" indicates the number of counterexamples that are successfully reproduced, "CE Irreproducible" indicates the number of counterexamples that are not reproducible, and "Time" provides the run-time in seconds for all simulations on each property. Indeed, all counterexamples were reproduced by DSValidator, which suggests that the counterexamples generated by DSVerifier are sound and reliable. Additionally, with our set of benchmarks, we were able to detect bugs in the DSVerifier implementation; and with the DSValidator toolbox, we were able to extract values from the counterexample to graphically reproduce the bugs that were found, to ensure the verifier correctness and to make the re-design process easy. Note that DSValidator returns a `.MAT`-file that represents the digital system with its implementation (e.g., realization, fixed-point format, inputs). By combining this implementation extracted from the counterexample, the re-design of the digital system is more practical, due to the fact that a control engineer is then able not only to implement the same digital system with a different realization or fixed-point format, but he/she can also check with DSValidator if the violation is still occurring, i.e., through graphs, property-verification simulation, and also result validation. Due to this reason, DSValidator is a strong tool to support the verification performed by DSVerifier.

Table 2: Results for the Quadrotor Attitude System.

Property	CE Reproducible	CE Irreproducible	Time
Overflow	24	0	0.190 s
Limit Cycle	26	1	0.483 s
Minimum-Phase	54	0	0.012 s
Stability	54	0	0.188 s

Note further that the automated validation of all counterexamples took less than 1 second. We consider these times short enough to be of practical use to control engineers, and thus affirm RQ1. The present results also show that all counterexamples generated by DSVerifier, considering FWL effects and different realizations forms (i.e., DFI, DFII and TDFII), are sound and reliable since DSValidator is able to simulate the underlying digital system in MATLAB and then reproduce the respective counterexamples, positively answering RQ2. The present results also show that all counterexamples (except one) generated by the underlying verifier, considering FWL effects and different realizations forms (*i.e.*, DFI, DFII and TDFII), are sound and reliable, since DSValidator is able to simulate the underlying digital system in MATLAB and then reproduce the respective counterexample. Nonetheless, for the limit cycle property, there is one counterexample that was not reproduced in DSValidator. Previously, DSValidator did not take into account overflow in intermediate operations to compute the system's output using the DFII realization form. Indeed, this bug was confirmed and fixed by the DSVerifier's developer via a github commit.[3]

4.5 Threats to Validity

We have reported a favorable assessment of DSValidator over a diverse set of real-world benchmarks extracted from a real quadrotor attitude system. However, this set of benchmarks is limited within the scope of this paper and DSValidator's performance needs to be further assessed on a larger benchmark set in future work to check whether the counterexample reproduction complexity is increased.

We have also evaluated the counterexamples of one specific model checker (DSVerifier), given the lack of available verifiers for digital systems represented by transfer functions. Since our goal is to verify a large number of real-world systems, it is then required to combine the strengths of different verification techniques and tools. In future, we expect that DSValidator can be further employed to leverage the potential of other verification tools for validating digital systems via the proposed counterexample format.

5 RELATED WORK

Currently, there are tools that can cope with the counterexamples reproducibility and validation, and in particular, to witness a specification violation. In order to eliminate manual inspection of false alarms, the study developed by Beyer et al. [5], which has inspired our study, works with the notion of stepwise testification; a verifier finds a problematic program path and, in addition to the verification result "false", it constructs a witness for that path. The technique is implemented in two verification tools, named CPAchecker [6] and Ultimate Automizer [14], but both verification tools neither consider FWL effects nor work for digital system properties. Here, our main goal is to reproduce the failure found in digital controllers w.r.t. FWL effects, while Beyer et al. [5] check whether the validation of states generated by the verifier are indeed correct.

In MATLAB, there is a toolbox for Ordinary Differential Equation [17] that can cope with reproducibility for systems from second-order, and in particular, considers the fixed-point numerical representation; it is able to show graphically the limit cycle behavior in digital systems using the phase portrait diagram. However, it does not take advantage of recent advances in bit-precise verification typically implemented in modern and efficient software verifiers. Given the current knowledge in verification, there is no toolbox or related standalone tool that considers the fixed-point numerical representation to reproduce and validate violations for the digital system counterexample provided by a verifier. In contrast, DSValidator supports the counterexamples validation and reproducibility, considering FWL effects for digital systems represented by transfer functions.

6 CONCLUSIONS

DSValidator reproduces counterexamples generated for a digital controller that implements a quadrotor attitude system, taking into account implementation aspects (fixed-point arithmetic and realization), the overflow mode (saturate or wrap-around) and the rounding mode (nearest, floor and round) by simulating the given digital system with its counterexample trace in MATLAB. Currently, DSValidator is able to reproduce counterexamples for stability, minimum-phase, limit-cycle and overflow. There is no other automated MATLAB toolbox that reproduces counterexamples for digital system generated by verifiers, or, if this is impossible, to identify the reason why the counterexample cannot be reproduced. This step validates and endorses the verification step and, most importantly, avoids false alarms. As future work, we expect to contribute to digital system verification by supporting further verifiers so that DSValidator can be applied to establish trust in verification results for high-complexity systems.

REFERENCES

[1] R. B. Abreu, M. Y. R. Gadelha, L. C. Cordeiro, E. B. de Lima Filho, and W. S. da Silva. Bounded model checking for fixed-point digital filters. *Journal of the Brazilian Computer Society*, 22(1):20, 2016.
[2] P. Bauer and L.-J. Leclerc. A computer-aided test for the absence of limit cycles in fixed-point digital filters. *IEEE Trans. Signal Processing*, 39(11):2400–2410, Nov 1991.
[3] I. Bessa, H. Ismail, L. Cordeiro, and J. Filho. Verification of fixed-point digital controllers using direct and delta forms realizations. *Design Autom. for Emb. Sys.*, 20(2):95–126, 2016.
[4] I. Bessa, H. Ismail, R. Palhares, L. Cordeiro, and J. E. C. Filho. Formal non-fragile stability verification of digital control systems with uncertainty. *IEEE Transactions on Computers*, 66(3), 2017.
[5] D. Beyer, M. Dangl, D. Dietsch, M. Heizmann, and A. Stahlbauer. Witness validation and stepwise testification across software verifiers. In *ESEC/FSE*, pages 721–733, 2015.
[6] D. Beyer and M. E. Keremoglu. Cpachecker: A tool for configurable software verification. In *CAV*, volume 6806 of *LNCS*, pages 184–190, 2011.
[7] S. Bouabdallah, P. Murrieri, and R. Siegwart. Design and control of an indoor micro quadrotor. In *ICRA*, volume 5, pages 4393–4398 Vol.5, April 2004.
[8] J. Carletta, R. Veillette, F. Krach, and Z. Fang. Determining appropriate precisions for signals in fixed-point IIR filters. In *DAC*, pages 656–661, 2003.
[9] L. Chaves, I. Bessa, L. C. Cordeiro, D. Kroening, and E. B. de Lima Filho. Verifying digital systems with MATLAB. In *ISSTA*, pages 388–391, 2017.
[10] E. Clarke, D. Kroening, and F. Lerda. A tool for checking ANSI-C programs. In *TACAS*, volume 2988 of *LNCS*, pages 168–176, 2004.
[11] L. Cordeiro, B. Fischer, and J. Marques-Silva. SMT-Based Bounded Model Checking for Embedded ANSI-C Software. *IEEE Transaction on Software Engineering*, 38(4):957–974, 2012.
[12] P. Diniz, S. Netto, and E. D. Silva. *Digital Signal Processing: System Analysis and Design.* Cambridge University Press, New York, NY, USA, 2002.
[13] S. Fadali and A. Visioli. *Digital Control Engineering: Analysis and Design*, volume 303 of *Electronics & Electrical.* Elsevier/Academic Press, 2009.
[14] M. Heizmann, J. Christ, D. Dietsch, E. Ermis, J. Hoenicke, M. Lindenmann, A. Nutz, C. Schilling, and A. Podelski. Ultimate automizer with smtinterpol - (competition contribution). In *TACAS*, volume 7795 of *LNCS*, pages 641–643, 2013.
[15] H. Ismail, I. Bessa, L. C. Cordeiro, E. B. de Lima Filho, and J. E. C. Filho. DSVerifier: A bounded model checking tool for digital systems. In *SPIN*, volume 9232 of *LNCS*, pages 126–131, 2015.
[16] R. Istepanian and J. Whidborne. *Digital Controller Implementation and Fragility: A Modern Perspective.* Advances in Industrial Control. Springer, 2001.
[17] MathWorks. Matlab toolbox, 2017.
[18] 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 *FSE*, pages 1151–1153, 2016.
[19] K. Premaratne, E. Kulasekere, P. Bauer, and L.-J. Leclerc. An exhaustive search algorithm for checking limit cycle behavior of digital filters. *IEEE Trans. Signal Processing*, 44(10):2405–2412, Oct 1996.
[20] H. Rocha, R. S. Barreto, L. C. Cordeiro, and A. D. Neto. Understanding programming bugs in ANSI-C software using bounded model checking counter-examples. In *IFM*, volume 7321 of *LNCS*, pages 128–142, 2012.

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