LCV: A Verification Tool for Linear Controller Software

Junkil Park[1(✉)], Miroslav Pajic[2], Oleg Sokolsky[1], and Insup Lee[1]

[1] Department of Computer and Information Science, University of Pennsylvania, Philadelphia, PA, USA
{park11,sokolsky,lee}@cis.upenn.edu
[2] Department of Electrical and Computer Engineering, Duke University, Durham, NC, USA
miroslav.pajic@duke.edu

Abstract. In the model-based development of controller software, the use of an unverified code generator/transformer may result in introducing unintended bugs in the controller implementation. To assure the correctness of the controller software in the absence of verified code generator/transformer, we develop Linear Controller Verifier (LCV), a tool to verify a linear controller implementation against its original linear controller model. LCV takes as input a Simulink block diagram model and a C code implementation, represents them as linear time-invariant system models respectively, and verifies an input-output equivalence between them. We demonstrate that LCV successfully detects a known bug of a widely used code generator and an unknown bug of a code transformer. We also demonstrate the scalability of LCV and a real-world case study with the controller of a quadrotor system.

1 Introduction

Most safety-critical embedded and cyber-physical systems have a software-based controller at their core. The safety of these systems rely on the correct operation of the controller. Thus, in order to have a high assurance for such systems, it is imperative to ensure that controller software is correctly implemented.

Nowadays, controller software is developed in a model-based fashion, using industry-standard tools such as Simulink [31] and Stateflow [36]. In this development process, first of all, the controller model is designed and analyzed. Controller design is performed using a mathematical model of the control system that captures both the dynamics of the "plant", the entity to be controlled, and the controller itself. With this model, analysis is performed to conclude whether the plant model adequately describes the system to be controlled, and whether the controller achieves the desired goals of the control system. Once the control engineer is satisfied with the design, a software implementation is automatically produced by code generation from the mathematical model of the controller. Code generation tools such as Embedded Coder [30] and Simulink Coder [32]

T. Vojnar and L. Zhang (Eds.): TACAS 2019, Part I, LNCS 11427, pp. 213–225, 2019.
https://doi.org/10.1007/978-3-030-17462-0_12

are widely used. The generated controller implementation is either used as it is in the control system, or sometimes transformed into another code before used for various reasons such as numerical accuracy improvement [8,9] and code protection [2,4,5]. For simplicity's sake herein, we will call code generation even when code generation is potentially followed by code transformation.

To assure the correctness of the controller implementation, it is necessary to check that code generation is done correctly. Ideally, we would like to have verified tools for code generation. In this case, no verification of the controller implementation would be needed because the tools would guarantee that any produced controller correctly implements its model. In practice, however, commercial code generators are complex black-box software that are generally not amenable to formal verification. Subtle bugs have been found in commercially available code generators that consequently generate incorrect code [29]. Unverified code transformers may introduce unintended bugs in the output code.

In the absence of verified code generators, it is desirable to verify instances of implementations against their original models. Therefore, this work considers the problem of such instance verification for a given controller model and software implementation. To properly address this verification problem, the following challenges should be considered: First of all, such verification should be performed from the input-output perspective (i.e., input-output conformance). Correct implementations may have different state representations to each other for several possible reasons (e.g., code generator's choice of state representation, optimization used in the code generation process). In other words, the original controller model and a correct implementation of the model may be different from each other in state representation, while being functionally equivalent from the input-output perspective. Thus, it is necessary to develop the verification technique that is not sensitive to the state representation of the controller. Moreover, there is an inherent discrepancy between controller models and their implementations. The controller software for embedded systems uses a finite precision arithmetic (e.g., floating-point arithmetic) which introduces rounding errors in the computation. In addition to these rounding errors, the implementations may be inexact in the numeric representation of controller parameters due to the potential rounding errors in the code generation/optimization process. Thus, it is reasonable to allow a tolerance in the conformance verification as long as the implementation has the same desired property to the model's. Finally, such verification is desired to be automatic and scalable because verification needs to be followed by each instance of code generation.

We, therefore, present LCV (shown in Fig. 1), a tool that automatically verifies controller implementations against their models from the input-output perspective with given tolerance thresholds.[1] The verification technique behind this tool is based on the work of [24]. LCV uses the state-space representation form of the linear time-invariant (LTI) system to represent both the Simulink block

[1] We assume that a threshold value ϵ is given by a control engineer as a result of the robustness analysis that guarantees the desired properties of the control system in the presence of uncertain disturbances.

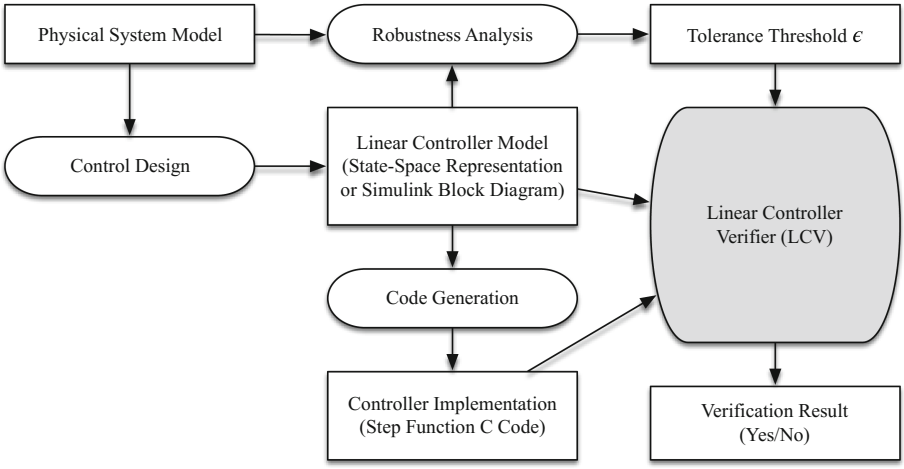

Fig. 1. LCV in the model-based development process.

diagram (i.e., controller model) and the C code (i.e., controller implementation). LCV checks the input-output equivalence relation between the two LTI models by similarity checking. The contribution of this work compared to the previous work [24] is as follows: As controller specifications are often given in the form of block diagrams, LCV extends the preliminary prototype [24] to take not only the state-space representation of an LTI system but also the Simulink block diagram as an input specification model. As a result, a real-world case study, where the controller specification of a quadrotor called Erle-Copter [11] is given as a Simulink block diagram, was conducted using LCV and demonstrated in this paper. In the case study with a proportional-integral-derivative (PID) controller, we demonstrate that LCV successfully detects a known (reproduced) bug of Embedded Coder as well as an unknown bug of Salsa [8], a code transformation method/tool for numerical accuracy.[2] Moreover, LCV has been enhanced in many ways such as improving in scalability, supporting fully automatic verification procedures, providing informative output messages and handling customized user inputs.

2 Related Work

To ensure the correctness of the controller implementation against the controller model, a typically used method in practice is equivalence testing (or back-to-back testing) [6,7,28] which compares the outputs of the executable model and code for the common input sequence. The limitation of this testing-based method is that it does not provide a thorough verification. Static analysis-based approaches [3,12,14] have been used to analyze the controller code,

[2] This bug has been confirmed by the author of the tool.

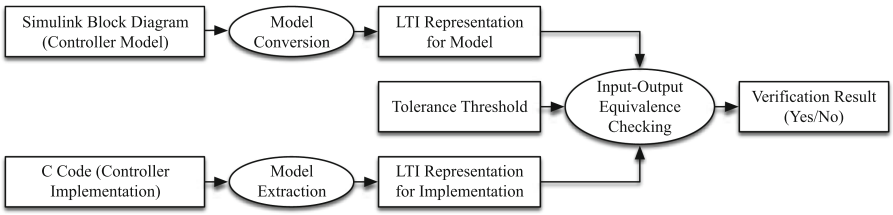

Fig. 2. The verification flow of LCV.

but focuses on checking common properties such as numerical stability, the absence of buffer overflow or arithmetic exceptions rather than verifying the code against the model. The work of [17,27] proposes translation validation techniques for Simulink diagrams and the generated codes. The verification relies on the structure of the block diagram and the code, thus being sensitive to the controller state while our method verifies code against the model from the input-output perspective, not being sensitive to the controller state. Due to optimization and transformation during a code generation process, a generated code which is correct, may have a different state representation than the model's. In this case, our method can verify that the code is correct w.r.t. the model, but the state-sensitive methods [17,27] cannot. [13,16,37,38] present a control software verification approach based on the concept of proof-carrying code. In their approach, the code annotation based on the Lyapunov function and its proof are produced at the time of code generation. The annotation asserts control theory related properties such as stability and convergence, but not equivalence between the controller specifications and the implementations. In addition, their approach requires the internal knowledge and control of the code generator to use, and may not be applicable to the off-the-shelf black-box code generators. The work of [19,24,25] presents methods to verify controller implementations against LTI models, but does not relate the block diagram models with the implementation code.

3 Verification Flow of Linear Controller Verifier

The goal of LCV is to verify linear controller software. Controllers are generally specified as a function that, given the current state of the controller and a set of input sensor values, computes control output that is sent to the system actuators and the new state of the controller. In this work, we focus on linear-time invariant (LTI) controllers [26], since these are the most commonly used controllers in control systems. In software, controllers are implemented as a subroutine (or a function in the C language). This function is known as the *step function* (see [23] for an example). The step function is invoked by the control system periodically, or upon arrival of new sensor data (i.e., measurements).

This section describes the verification flow (shown in Fig. 2) and the implementation details of LCV. LCV takes as input a Simulink block diagram

(i.e., controller model), a C code (i.e., controller implementation) and a tolerance threshold as a real number. In addition, LCV requires the following information to be given as input: the name of the step function and the interface of the step function. LCV assumes that the step function interfaces through the given input and output global variables. In other words, the input and output variables are declared in the global scope, and the input variables are written (or set) before the execution (or entrance) of the step function. Likewise, the output variables are read (or used) after the execution (or exit) of the step function.[3] Thus, the step function interface comprises the list of input (and output) variables of the step function in the same order of the corresponding input (and output) ports of the block diagram model. Since LCV verifies controllers from the input-output perspective, LCV does not require any state related information (i.e., the dimension of the controller state, or the list of state variables of the step function). Instead, LCV automatically obtains such information about the controller state from the analysis of the input C code and the input Simulink block diagram.

A restriction on this work is that LCV only focuses on verifying linear controller software. Thus, the scope of inputs of LCV is limited as follows: the input C program is limited to be a step function that only has a deterministic and finite execution path for a symbolic input, which is often found to be true for many embedded linear controllers. Moreover, the input Simulink block diagram is limited to be essentially an LTI system model (i.e., satisfying the superposition property). The block diagram that LCV can handle may include basic blocks (e.g., constant block, gain block, sum block), subsystem blocks (i.e., hierarchy) and series/parallel/feedback connections of those blocks. Extending LCV to verify a broader class of controllers is an avenue for future work.

The key idea in the verification flow (shown in Fig. 2) is that LCV represents both the Simulink block diagram and the C code in the same form of mathematical representation (i.e., the state space representation of an LTI system), and compares the two LTI models from the input-output perspective. Thus, the first step of the verification is to transform the Simulink block diagram into a state space representation of an LTI system, which is defined as follows:

$$\begin{aligned} \mathbf{z}_{k+1} &= \mathbf{A}\mathbf{z}_k + \mathbf{B}\mathbf{u}_k \\ \mathbf{y}_k &= \mathbf{C}\mathbf{z}_k + \mathbf{D}\mathbf{u}_k. \end{aligned} \tag{1}$$

where $\mathbf{u}_k$, $\mathbf{y}_k$ and $\mathbf{z}_k$ are the input vector, the output vector and the state vector at time k respectively. The matrices $\mathbf{A}$, $\mathbf{B}$, $\mathbf{C}$ and $\mathbf{D}$ are controller parameters. We convert the Simulink block diagram into the LTI model employing the 'exact linearization' (or block-by-block linearization) feature of Simulink Control Design [33] which is implemented in the built-in Matlab function `linearize`. In this step, each individual block is linearized first and then combined together with others to produce the overall block diagram's LTI model.

[3] This convention is used by Embedded Coder, a code generation toolbox for Matlab/Simulink.

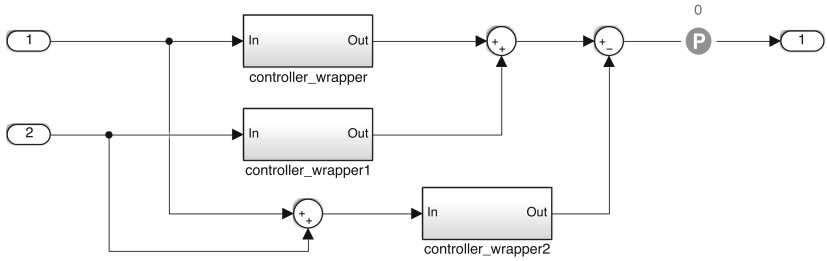

Fig. 3. The simulink block diagram for checking the additivity of the controller

This step assumes that the block diagram represents a linear controller model. A systematic procedure[4] can remove this assumption: one can check whether a given Simulink block diagram is linear (i.e., both additive and homogeneous) using Simulink Design Verifier [34], a model checker for Simulink. For example, to check if a controller block in Simulink is additive or not, as shown in Fig. 3, one can create two additional duplicates of the controller block, generate two different input sequences, and exhaustively check if the output of the controller in response to the sum of two inputs is equal to the sum of two outputs of the controllers in response the two inputs respectively. In Fig. 3, `controller_wrapper` wraps the actual controller under test, and internally performs multiplexing and demultiplexing to handle the multiple inputs and outputs of the controller. Simulink Design Verifier serves checking if this holds for all possible input sequences. However, a limitation of the current version of Simulink Design Verifier is that it does not support all Simulink blocks and does not properly handle non-linear cases. In these cases, alternatively, one can validate the linearity of controllers using simulation-based testing instead of model checking, which can be systematically done by Simulink Test [35]. This method is not limited by any types of Simulink blocks, and can effectively disprove the linearity of controllers for non-linear cases. However, this alternative method using Simulink Test may not be as rigorous as the model-checking based method using Simulink Design Verifier because not all possible input cases are considered.

The next step in the LCV's verification flow is to extract the LTI model from the controller implementation C code. The idea behind this step is to exploit the fact that linear controller codes (i.e., step function) used for embedded systems generally have simple control flows for the sake of deterministic real-time behaviors (e.g., fixed upper bound of loops). Thus, the semantics of such linear controller codes can be represented as a set of mathematical functions that are loop-free, which can be further transformed into the form of an LTI model. To do this specifically, LCV uses the symbolic execution technique which is capable of

[4] This procedure is currently not implemented in LCV because the required tools such as Simulink Design Verifier and Simulink Test mostly provide their features through GUIs rather than APIs. Thus, this procedure will be implemented in the future work once such APIs are available. Until then, this procedure can be performed manually.

identifying the computation of the step function (i.e., C function which implements the controller). By the computation, we mean the big-step transition relation on global states between before and after the execution of the step function. The big-step transition relation is represented as symbolic formulas that describe how the global variables change as the effect of the step function execution. The symbolic formulas associate each global variable representing the controller's state and output with the symbolic expression to be newly assigned to the global variable, where the symbolic expression consists of the old values of the global variables representing the controller's state and input. Then, LCV transforms the set of equations (i.e., symbolic formulas) that represent the transition relation into a form of matrix equation, from which an LTI model for the controller implementation is extracted [24]. LCV employs the off-the-shelf symbolic execution tool PathCrawler [39], which outputs the symbolic execution paths and the path conditions of a given C program in an extensible markup language (XML) file format.

Finally, LCV performs the input-output equivalence checking between the LTI model obtained from the block diagram and the LTI model extracted from the C code implementation. To do this, we employ the notion of similarity transformation [26], which implies that two minimal LTI models $\mathbf{\Sigma}(\mathbf{A}, \mathbf{B}, \mathbf{C}, \mathbf{D})$ and $\hat{\mathbf{\Sigma}}(\hat{\mathbf{A}}, \hat{\mathbf{B}}, \hat{\mathbf{C}}, \hat{\mathbf{D}})$ are input-output equivalent if and only if they are *similar* to each other, meaning that there exists a non-singular matrix $\mathbf{T}$ such that

$$\hat{\mathbf{A}} = \mathbf{T}\mathbf{A}\mathbf{T}^{-1}, \qquad \hat{\mathbf{B}} = \mathbf{T}\mathbf{B}, \qquad \hat{\mathbf{C}} = \mathbf{C}\mathbf{T}^{-1}, \qquad \text{and} \qquad \hat{\mathbf{D}} = \mathbf{D} \tag{2}$$

where $\mathbf{T}$ is referred to as the *similarity transformation matrix* [26].

Given the extracted LTI model (from the C Code) and the original LTI model (obtained from the Simulink block diagram), we first minimize both LTI models via Kalman Decomposition [26] (Matlab function `minreal`). Then, the input-output equivalence checking problem is reduced to the problem of finding the existence of $\mathbf{T}$ (i.e., similarity checking problem). LCV formulates the similarity checking problem as a convex optimization problem[5], and employs CVX [15], a convex optimization solver to find $\mathbf{T}$. In the formulation, the equality relation is relaxed up to a given tolerance threshold ϵ in order to tolerate the numerical errors that come from multiple sources (e.g., the controller parameters, the computation of the implementation, the verification process). We assume that the tolerance threshold ϵ is given by a control engineer as the result of robustness analysis so that the verified controller implementation preserves the certain desired properties of the original controller model (e.g., stability). ϵ is chosen to be 10^{-5} for the case study that we performed in the next section.

The output of LCV is as follows: First of all, when LCV fails to extract an LTI model from code, it tells the reason (e.g., non-deterministic execution paths for a symbolic input due to branching over a symbolic expression condition, non-linear arithmetic computation due to the use of trigonometric functions). Moreover, for the case of non-equivalent model and code, LCV provides the LTI models obtained from the Simulink block diagram model and the C code respectively, so that the user can simulate both of the models and easily find

[5] Please refer [24] for the details of the formulation.

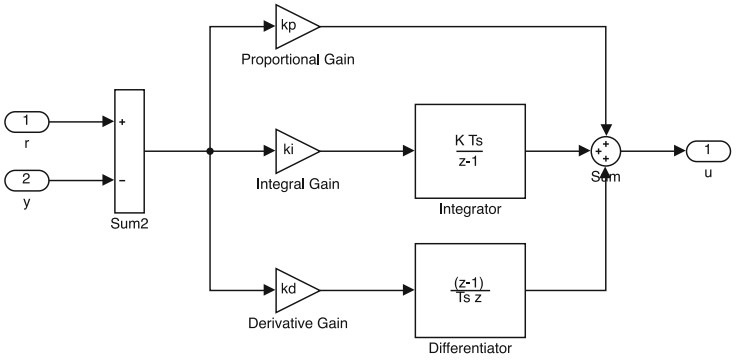

Fig. 4. The block diagram of the PID controller.

an input sequence that leads to a discrepancy between their output behaviors.[6] Finally, for the case of equivalent model and code, LCV additionally provides a similarity transformation matrix $\mathbf{T}$ between the two LTI models, which is the key evidence to prove the input-output equivalence between the model and code.

4 Evaluation

We evaluate LCV through conducting a case study using a standard PID controller and a controller used in a quadrotor. We also evaluate the scalability of LCV in the subsequent subsection.

4.1 Case Study

PID Controller. In our case study, we first consider a proportional-integral-derivative (PID) controller, which is a closed-loop feedback controller commonly used in various control systems (e.g., industrial control systems, robotics, automotive). A PID controller attempts to minimize the error value e_t over time which is defined as the difference between a reference point r_t (i.e., desired value) and a measurement value y_t (i.e., $e_t = r_t - y_t$). To do this, the PID controller adjusts a control input u_t computing the sum of the proportion term $k_p e_t$, integral term $k_i T \sum_{i=1}^{t} e_t$ and derivative term $k_d \frac{e_t - e_{t-1}}{T}$ so that

$$u_t = k_p e_t + k_i T \sum_{i=1}^{t} e_t + k_d \frac{e_t - e_{t-1}}{T}. \tag{3}$$

where k_p, k_i and k_d are gain constants for the corresponding term, and T is the sampling time. Figure 4 shows the Simulink block diagram for the PID controller, where the gain constants are defined as $k_p = 9.4514, k_i = 0.69006, k_d = 2.8454$, and the sampling period is 0.2 s.

[6] This feature to generate counterexamples will be implemented in a future version of LCV.

Table 1. Summary of the case study with the PID controller (Fig. 4) and its different versions of implementation

Impl.	Description	Buggy?	LCV output
`PID1`	Generated by Embedded Coder	No	Equivalent
`PID2`	Optimized from PID1 by Salsa (Level 1)	No	Equivalent
`PID3`	Optimized from PID1 by Salsa (Level 2)	Yes (due to a bug in Salsa)	Not equivalent
`PID3'`	Corrected from PID3 manually	No	Equivalent
`PID4`	Generated by Embedded Coder with a buggy option triggered	Yes (due to a bug in Embedded Coder)	Not equivalent

For the PID controller model, we check various different versions of implementations such as `PID1`, `PID2`, `PID3`, `PID3'` and `PID4` (summarized in Table 1). `PID1` is obtained by code generation from the model using Embedded Coder. `PID2` is obtained from `PID1` by the transformation (or optimization) of Salsa [8] to improve the numerical accuracy (using the first transformation technique (referred to as Level 1) presented in [8]). In a similar way, `PID3` is obtained by the transformation from `PID1` for an even better numerical accuracy (following the second transformation technique (referred to as Level 2) as Listing 3 in [8]). However, this transformation for `PID3` contains an unintended bug by mistake that has been confirmed by the authors of the paper (i.e., variable `s` is not computed correctly, and the integral term is redundantly added to the output), which makes `PID3` incorrect. `PID3'` is an implementation that manually corrects `PID3`. Using LCV, we can verify that `PID1`, `PID2` and `PID3'` are correct implementations, but `PID3` is not (see the verification result for `PID3` [21]).

Moreover, `PID4` is obtained by injecting a known bug of Embedded Coder into the implementation `PID1`. The bug with the ID 1658667 [29] that exists in the Embedded Coder version from 2015a through 2017b (7 consecutive versions) causes the generated code to have state variable declarations in a wrong scope. The state variables which are affected by the bug are mistakenly declared as local variables inside the step function instead of being declared as global variables. Thus, those state variables affected by the bug are unable to preserve their values throughout the consecutive step function executions. LCV can successfully detect the injected bug by identifying that the extracted model from the controller code does not match the original controller model (see the verification result for `PID4` [22]).

Quadrotor Controller. The second and more complex application in our case study is a controller of the quadrotor called Erle-Copter. The quadrotor controller controls the quadrotor to be in certain desired angles in roll, yaw and pitch. The quadrotor uses the controller software from the open source project Ardupilot [1]. Inspired by the controller software, we obtained the Simulink block diagram shown in Fig. 5. In the names of the Inport blocks, the suffix `_d` indicates the desired angle, `_d`, the measured angle, and `_rate_y`, the angular speed. Each component of the coordinate of the quadrotor is separately controlled by its own

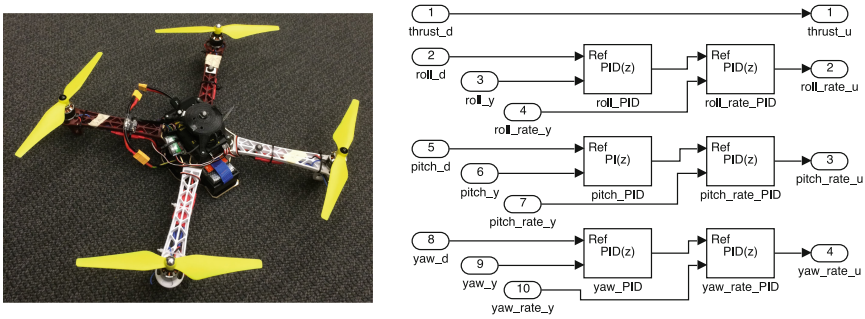

Fig. 5. Our quadrotor platform (Left). The quadrotor controller block diagram (Right).

cascade PID controller [18]. A cascade of PID controller is a sequential connection of two PID controllers such that one PID controller controls the reference point of another. In Fig. 5, there are three cascade controllers for the controls of roll, pitch and yaw. For example, for the roll control, `roll_pid` controls the angle of roll, while `roll_rate_PID` controls the rate of roll using the output of `roll_PID` as the reference point. The sampling time T of each PID controller is 2.5 ms. This model uses the built-in PID controller block of Simulink to enable the PID auto-tuning software in Matlab (i.e., `pidtune()`). The required physical quantities for controlling roll and pitch are identified by physical experiments [10]. We use Embedded Coder to generate the controller code for the model, and verify that the generated controller code correctly implements the controller model using LCV (see the verification result for the quadrotor controller [20]).

4.2 Scalability

To evaluate the scalability of LCV, we measure the running time of LCV verifying the controllers of different dimensions (i.e., the size of the LTI model). We randomly generate LTI controller models using Matlab function `drss` varying the controller dimension n from 2 to 50. The range of controller sizes was chosen based on our observation of controller systems in practice. We construct Simulink models with LTI system blocks that contain the generated LTI models, and use Embedded Coder to generate the implementations for the controllers. The running time of LCV for verifying the controllers with different dimensions is presented in Fig. 6, which shows that LCV is scalable for the realistic size of controller dimension. Compared to the previous version (or the preliminary prototype) of LCV [24], the new version of LCV has been much improved in scalability by tighter integration with the symbolic execution engine PathCrawler (i.e., in the model extraction phase, the invocation of constraint solver along with symbolic execution has been significantly reduced).

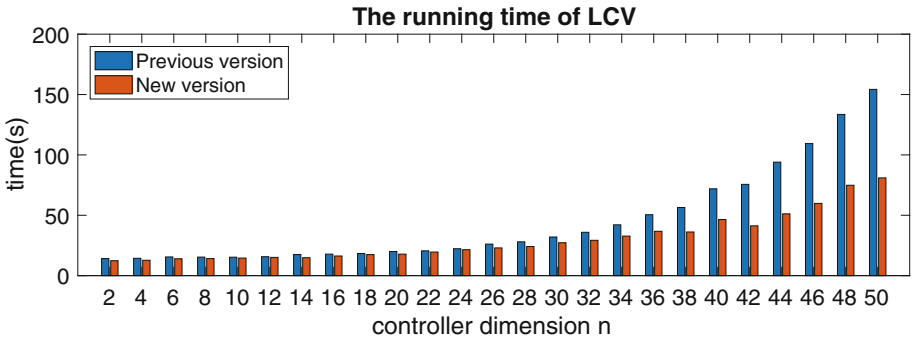

Fig. 6. The running time of LCV for verifying controllers with dimension n.

5 Conclusion

We have presented our tool LCV which verifies the equivalence between a given Simulink block diagram and a given C implementation from the input-output perspective. Through an evaluation, we have demonstrated that LCV is applicable to the verification of a real-world system's controller and scalable for the realistic controller size. Our current/future development work includes: relating the equivalence precision and the controller's performance, and handling nonlinear controllers.

Acknowledgments. This work is sponsored in part by the ONR under agreement N00014-17-1-2504, as well as the NSF CNS-1652544 grant. This research was supported in part by ONR N000141712012, Global Research Laboratory Program (2013K1A1A2A02078326) through NRF, and the DGIST Research and Development Program (CPS Global Center) funded by the Ministry of Science, ICT & Future Planning, and NSF CNS-1505799 and the Intel-NSF Partnership for Cyber-Physical Systems Security and Privacy.

References

1. Ardupilot Dev Team: Ardupilot, September 2018. http://ardupilot.org/
2. Behera, C.K., Bhaskari, D.L.: Different obfuscation techniques for code protection. Procedia Comput. Sci. **70**, 757–763 (2015)
3. Blanchet, B., et al.: A static analyzer for large safety-critical software. In: ACM SIGPLAN Notices, vol. 38, pp. 196–207. ACM (2003)
4. Cappaert, J.: Code obfuscation techniques for software protection, pp. 1–112. Katholieke Universiteit Leuven (2012)
5. Collberg, C., Thomborson, C., Low, D.: A taxonomy of obfuscating transformations. Technical report, Department of Computer Science, The University of Auckland, New Zealand (1997)
6. Conrad, M.: Testing-based translation validation of generated code in the context of IEC 61508. Form Methods Syst. Des. **35**(3), 389–401 (2009)

7. Conrad, M.: Verification and validation according to ISO 26262: a workflow to facilitate the development of high-integrity software. Embedded Real Time Software and Systems (ERTS2 2012) (2012)
8. Damouche, N., Martel, M., Chapoutot, A.: Transformation of a PID controller for numerical accuracy. Electron. Notes Theor. Comput. Sci. **317**, 47–54 (2015)
9. Damouche, N., Martel, M., Chapoutot, A.: Improving the numerical accuracy of programs by automatic transformation. Int. J. Softw. Tools Technol. Transfer **19**(4), 427–448 (2017). https://doi.org/10.1007/s10009-016-0435-0
10. Derafa, L., Madani, T., Benallegue, A.: Dynamic modelling and experimental identification of four rotors helicopter parameters. In: 2006 IEEE International Conference on Industrial Technology (2006)
11. Erle Robotics: Erle-copter, September 2018. http://erlerobotics.com/blog/erle-copter/
12. Feret, J.: Static analysis of digital filters. In: Schmidt, D. (ed.) ESOP 2004. LNCS, vol. 2986, pp. 33–48. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-24725-8_4
13. Feron, E.: From control systems to control software. IEEE Control Syst. **30**(6), 50–71 (2010)
14. Goubault, E., Putot, S.: Static analysis of finite precision computations. In: Jhala, R., Schmidt, D. (eds.) VMCAI 2011. LNCS, vol. 6538, pp. 232–247. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-18275-4_17
15. Grant, M., Boyd, S.: CVX: Matlab software for disciplined convex programming, version 2.1, March 2014. http://cvxr.com/cvx
16. Herencia-Zapana, H., et al.: PVS linear algebra libraries for verification of control software algorithms in C/ACSL. In: Goodloe, A.E., Person, S. (eds.) NFM 2012. LNCS, vol. 7226, pp. 147–161. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-28891-3_15
17. Majumdar, R., Saha, I., Ueda, K., Yazarel, H.: Compositional equivalence checking for models and code of control systems. In: 52nd Annual IEEE Conference on Decision and Control (CDC), pp. 1564–1571 (2013)
18. Michael, N., Mellinger, D., Lindsey, Q., Kumar, V.: The GRASP multiple micro-UAV test bed. IEEE Robot. Autom. Mag. **17**(3), 56–65 (2010)
19. Pajic, M., Park, J., Lee, I., Pappas, G.J., Sokolsky, O.: Automatic verification of linear controller software. In: 12th International Conference on Embedded Software (EMSOFT), pp. 217–226. IEEE Press (2015)
20. Park, J.: Erle-copter verification result. https://doi.org/10.5281/zenodo.2565035
21. Park, J.: Pid3 verification result. https://doi.org/10.5281/zenodo.2565023
22. Park, J.: Pid4 verification result. https://doi.org/10.5281/zenodo.2565030
23. Park, J.: Step function example. https://doi.org/10.5281/zenodo.44338
24. Park, J., Pajic, M., Lee, I., Sokolsky, O.: Scalable verification of linear controller software. In: Chechik, M., Raskin, J.-F. (eds.) TACAS 2016. LNCS, vol. 9636, pp. 662–679. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-49674-9_43
25. Park, J., Pajic, M., Sokolsky, O., Lee, I.: Automatic verification of finite precision implementations of linear controllers. In: Legay, A., Margaria, T. (eds.) TACAS 2017. LNCS, vol. 10205, pp. 153–169. Springer, Heidelberg (2017). https://doi.org/10.1007/978-3-662-54577-5_9
26. Rugh, W.J.: Linear System Theory. Prentice Hall, London (1996)
27. Ryabtsev, M., Strichman, O.: Translation validation: from simulink to C. In: Bouajjani, A., Maler, O. (eds.) CAV 2009. LNCS, vol. 5643, pp. 696–701. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-02658-4_57

28. Stuermer, I., Conrad, M., Doerr, H., Pepper, P.: Systematic testing of model-based code generators. IEEE Trans. Software Eng. **33**(9), 622–634 (2007)
29. The Mathworks, Inc.: Bug reports for incorrect code generation. http://www.mathworks.com/support/bugreports/?product=ALL&release=R2015b&keyword=Incorrect+Code+Generation
30. The Mathworks, Inc.: Embedded coder, September 2017. https://www.mathworks.com/products/embedded-coder.html
31. The Mathworks, Inc.: Simulink, September 2018. https://www.mathworks.com/products/simulink.html
32. The Mathworks, Inc.: Simulink coder, September 2018. https://www.mathworks.com/products/simulink-coder.html
33. The Mathworks, Inc.: Simulink control design, September 2018. https://www.mathworks.com/products/simcontrol.html
34. The Mathworks, Inc.: Simulink design verifier, September 2018. https://www.mathworks.com/products/sldesignverifier.html
35. The Mathworks, Inc.: Simulink test, September 2018. https://www.mathworks.com/products/simulink-test.html
36. The Mathworks, Inc.: Stateflow, September 2018. https://www.mathworks.com/products/stateflow.html
37. Wang, T., et al.: From design to implementation: an automated, credible autocoding chain for control systems. arXiv preprint arXiv:1307.2641 (2013)
38. Wang, T.E., Ashari, A.E., Jobredeaux, R.J., Feron, E.M.: Credible autocoding of fault detection observers. In: American Control Conference (ACC), pp. 672–677 (2014)
39. Williams, N., Marre, B., Mouy, P., Roger, M.: PathCrawler: automatic generation of path tests by combining static and dynamic analysis. In: Dal Cin, M., Kaâniche, M., Pataricza, A. (eds.) EDCC 2005. LNCS, vol. 3463, pp. 281–292. Springer, Heidelberg (2005). https://doi.org/10.1007/11408901_21