Xiao-rui ZHU, Chen LIANG, Zhen-guo YIN, Zhong SHAO, Meng-qi LIU, Hao CHEN, 2019. A new hierarchical software architecture towards safety-critical aspects of a drone system. *Frontiers of Information Technology & Electronic Engineering*, 20(3): 353-362. https://doi.org/10.1631/FITEE.1800636

A new hierarchical software architecture towards safety-critical aspects of a drone system

Key words: Safety-critical; Drone; Software architecture; Formal verification

Corresponding author: Xiao-rui ZHU
E-mail: xiaoruizhu@hit.edu.cn
ORCID: Xiao-rui ZHU, http://orcid.org/0000-0003-1400 059X;
Zhong SHAO, http://orcid.org/0000-0001-8184-7649

Motivation (1/2)

1. Unmanned aerial vehicle or drone is a typical safety-critical system. As the drone comes to our daily life, failures of a drone may result in severe damage to the environment and serious injury to the public.

A drone crash

Test of injury risk from a drone by Virginia Tech

Motivation (2/2)

2. Most efforts to improve reliability of the UAV systems have focused on algorithms.

Algorithm level	**Source code level**
Performance Improvements:	Potential software bugs:
(1) Improve modeling accuracy;	(1) Loss synchronization with an external sensor;
(2) Enhance the robustness of control algorithms;	(2) High approximation errors;
(3) Reduce sensor errors.	(3) Infinite loops.

3. Potential software bugs in the source code are subtle but might degrade the performance or even cause the drone to crash.

4. Few people so far have addressed bugs in the implementation of algorithms at the source code level.

Main idea

Improve safety and reliability of a drone system:

1. Design a new hierarchical software architecture of the drone system;

2. Verify serial peripheral interface (SPI) and inter-integrated circuit (I2C) bus drivers at the source code level by formal verification methods;

3. Evaluate improvements in reliability in case of device anomalies.

Software architecture

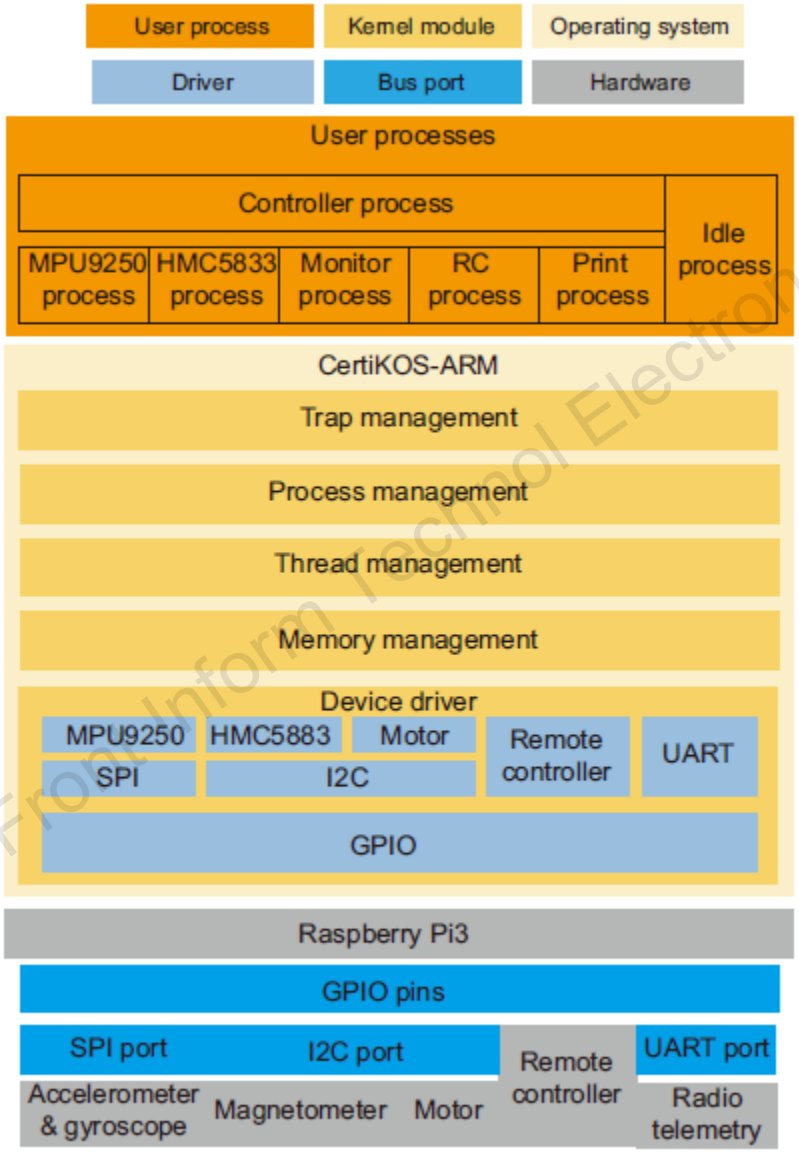

Fig. 1 Overall software architecture

Method (1/6)

Driver verification

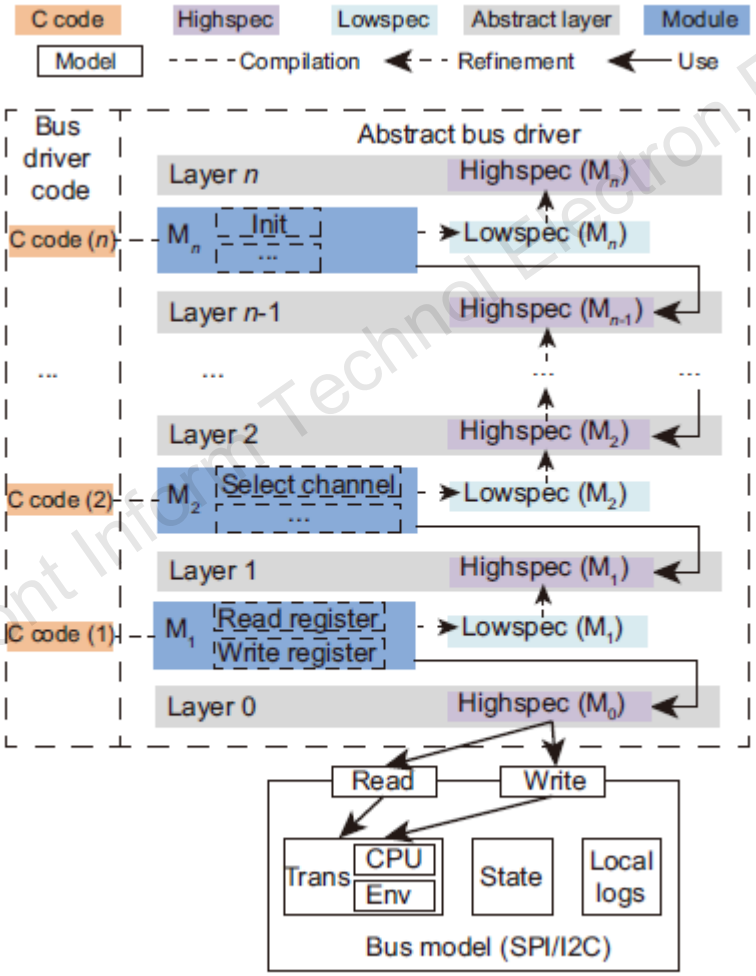

Fig. 2 Driver verification structure

Method (2/6)

Firstly, we build a **bus model** which abstracts machine registers and the physical memory into a state transition system.

Definition 2 (I2C bus abstract state)

```
Record I2CState :=
 mkI2CState {
       I2C_OA: ℤ
       I2C_SA: ℤ
       I2C_RX_DATA: ℤ
       I2C_TX_DATA: ℤ
       ...
       }.
```

Definition 6 (I2C bus read semantics)

$$\begin{aligned}&(\mathtt{e},\mathtt{l}_i') = \mathtt{next}(\mathtt{l}_{\mathtt{I2C}}^{\mathtt{env}},\mathtt{l}_i)\\&\mathtt{s}' = \delta_{\mathtt{I2C}}^{\mathtt{env}}(\mathtt{s},\mathtt{e})\\&\mathtt{res} = \kappa(\mathtt{n},\mathtt{s}')\\&\mathtt{s}'' = \delta_{\mathtt{I2C}}^{\mathtt{CPU}}(\mathtt{s}',(\mathtt{input}\ \mathtt{n})),\end{aligned}$$

Definition 7 (I2C bus write semantics)

$$\frac{\begin{aligned}&(\mathtt{e},\mathtt{l}_i') = \mathtt{next}(l_{\mathtt{I2C}}^{\mathtt{env}},\mathtt{l}_i)\\&\mathtt{s}' = \delta_{\mathtt{I2C}}^{\mathtt{env}}(\mathtt{s},\mathtt{e})\\&\mathtt{s}'' = \delta_{\mathtt{I2C}}^{\mathtt{CPU}}(\mathtt{s}',(\mathtt{output}\ \mathtt{n}\ \mathtt{v}))\end{aligned}}{\mathtt{write}(\mathtt{n},\mathtt{v},\mathtt{l}_i,\mathtt{l}_{\mathtt{I2C}}^{\mathtt{env}})=(\mathtt{s}'',\mathtt{l}_i').}$$

Method (3/6)

Secondly, we divide the C code of the device driver into **multiple layers** according to their functionalities and dependencies and abstract each C function into a Coq function.

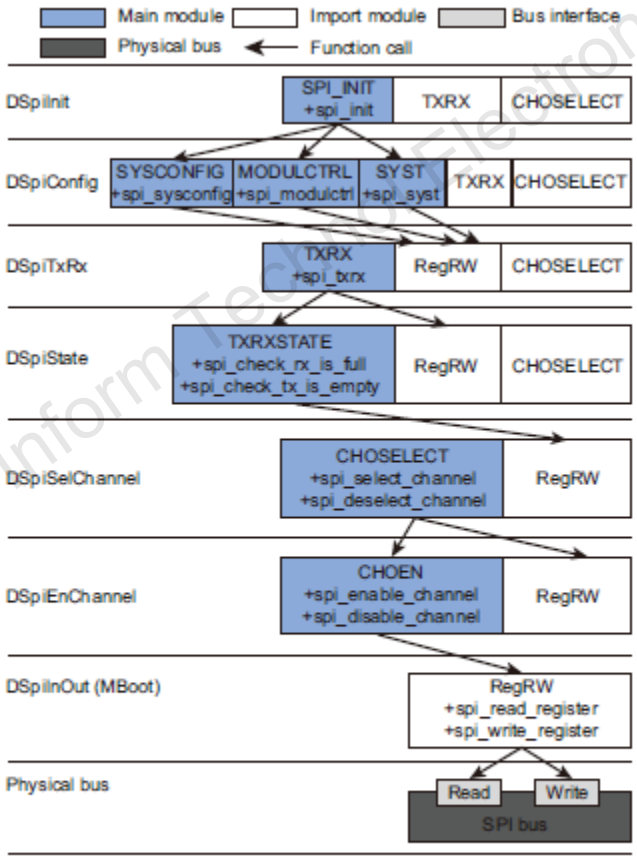

Fig. 3 Layering structure of the SPI bus driver

References to color refer to the online version of this figure

Method (4/6)

Lastly, we **verify the C code** based on the bus model and abstract bus driver layers. The verification method contains two steps: functional correctness of the C code and linking all layers together.

Functional correctness of the C code

Using Clightgen (provided by Compcert) to translate the C code of the SPI driver into a Clight abstract syntax tree.

```
void mcspi_enable_channel (void)
{
write_register(ENABLE_CHANNEL, CH0CTRL);
}
Definition mcspi_enable_channel :=
(Scall None
(Evar MCSPI_write_register (Tfunction
(Tcons tuint (Tcons tuint Tnil)) tvoid cc_default))
((Econst_int (ENABLE_CHANNEL) tint) ::
(Econst_int (CH0CTRL) tint) :: nil))).
```

Fig. 4 C source code and its Clight representation (in Coq) of function mcspi_enable_channel

Method (5/6)

Functional correctness of the C code

Then, we need to prove two refinements in the verification: the refinement between highspec and lowspec and the refinement between lowspec and actual C code.

Highspec describes the desired functionality of the module. The lowspec also abstracts the behavior of each function, but is specified in a way that is closer to the concrete hardware.

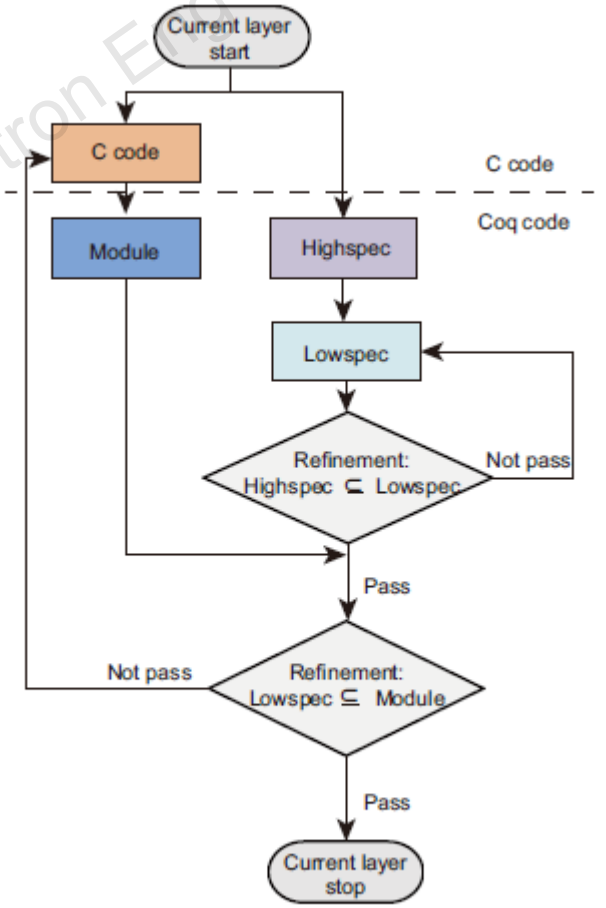

Fig. 5 Contextual refinement verification for a C function

Method (6/6)

Linking all layers together

The framework enables us to link layer together and prove the following contextual refinement between layers:

$$\texttt{P@DSpiSelChannel} \subseteq \texttt{P} \oplus \texttt{CH0SELECT@DSpiEnChannel}.$$

Once this refinement is proved, the actual implementation of the function CH0SELECT is hidden under layer DSpiSelChannel.

Following this method, we can link all layers together.

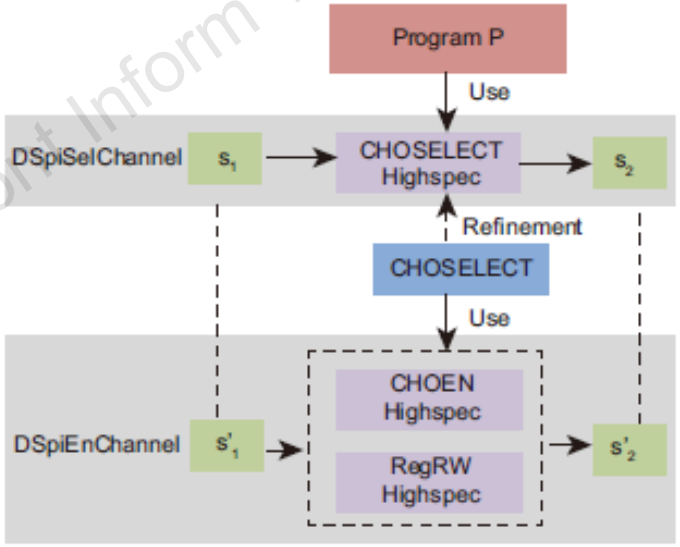

Fig. 7 Refinement between abstract layers

Experiments

1. Two drone systems were tested in the real field and the results were further compared. The first system is the drone system with a verified SPI bus driver, and the second one is a system with an unverified SPI bus driver.

2. Ten trials have been carried out with different bugs randomly occurring in the SPI bus driver. We recorded and compared attitudes of the drone.

Table 1 Configurations of three sensors of the drone*

Sensor	Chip name	Measurement range	Sensitivity	Sampling rate
Accelerometer	MPU9250	$\pm 8g$	4096 LSB/g	200 Hz
Gyroscope	MPU9250	± 1000 dps	32.8 LSB/dps	200 Hz
Magnetometer	HMC5883	± 1.3 Gs	1090 LSB/Gs	75 Hz

* Taken from datasheets of MPU9250 and HMC5883. g: standard gravity; dps: degree per second; Gs: gauss; LSB: least significant bit

Major results (1/5)

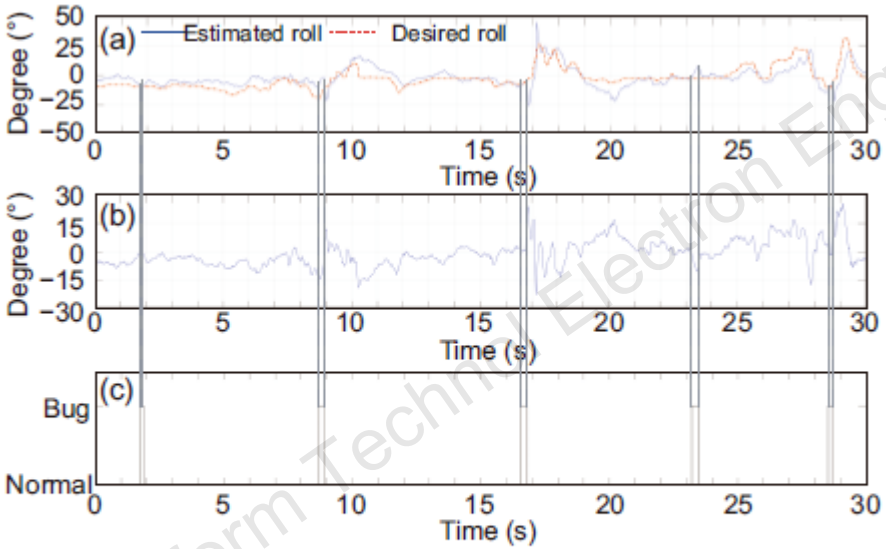

Fig. 9 Roll angle response of drone with an unverified SPI bus driver: (a) roll angle; (b) roll angle error; (c) SPI bus bug

Three peaks of errors are observed in the timeline 8.6 s, 16.8 s, and 28.5 s. At these time intervals, software bugs in the device drivers cause delayed process and response of sensor data, which further block the controller's execution for the next multiple control periods.

Major results (2/5)

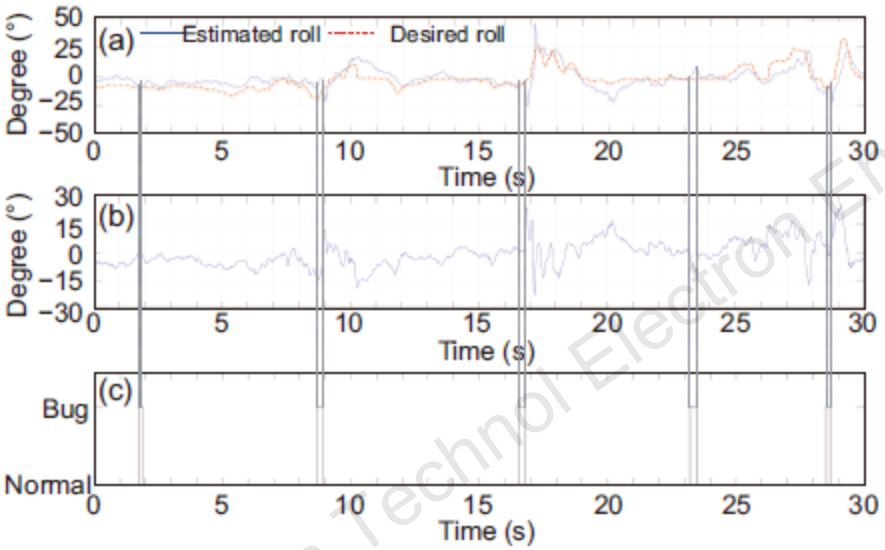

Fig. 9 Roll angle response of drone with an unverified SPI bus driver: (a) roll angle; (b) roll angle error; (c) SPI bus bug

Software bugs are detected at 1.8 s and 23.2 s. However, these bugs have no obvious impact on the roll angle, due to the relatively steady attitude of the drone. When these bugs occur, the input of each motor will be the same as in the previous period. If the current attitude of the drone does not change a lot compared with the previous one, the drone will stay stable using the same motor input.

Major results (3/5)

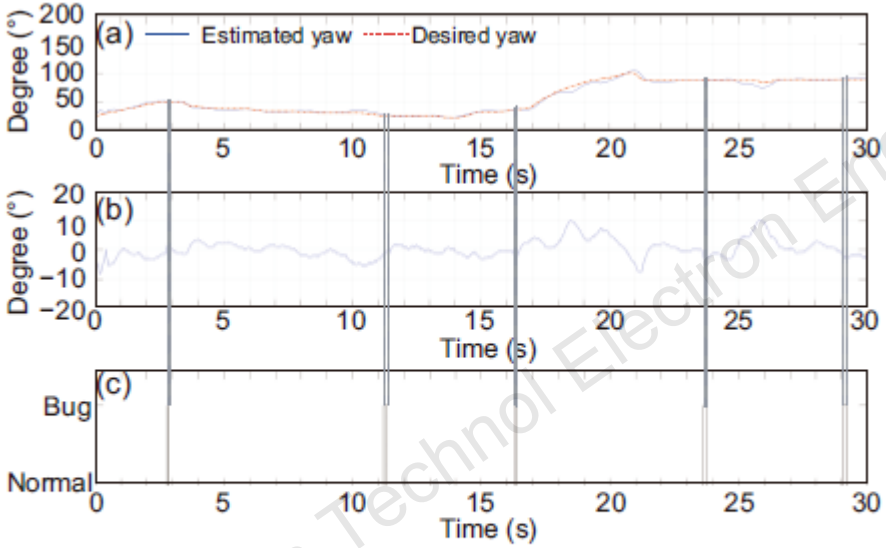

Fig. 11 Yaw angle response of drone with an unverified SPI bus driver: (a) yaw angle; (b) yaw angle error; (c) SPI bus bug

Fig. 11 shows the value of the yaw angle, which does not experience the same variation upon software faults caused by bugs. It is attributed to the sensor fusion algorithm, which uses data from both the IMU (connected with the SPI bus) and the magnetometer (connected to the I2C bus) to improve the accuracy of the estimated yaw angle.

Major results (4/5)

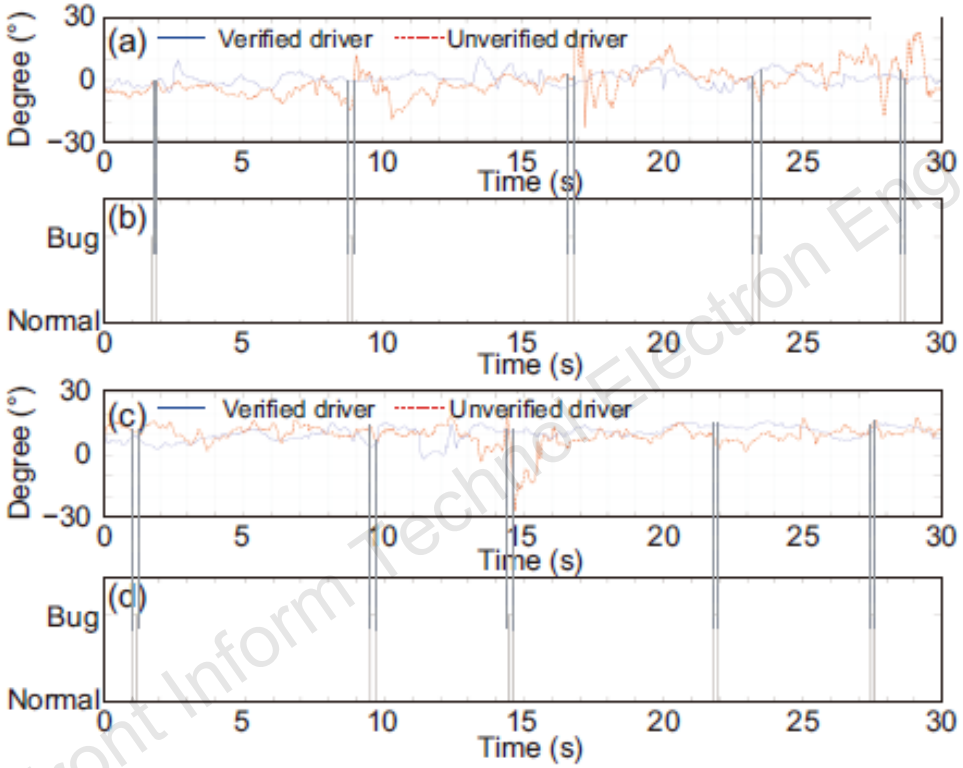

Fig. 12 Comparison of attitude errors in two drone systems: (a) roll angle error; (b) corresponding SPI bus bug; (c) pitch angle error; (d) corresponding SPI bus bug

The existence of software bugs leads to significant differences between desired and actual pitch and roll angles.

Major results (5/5)

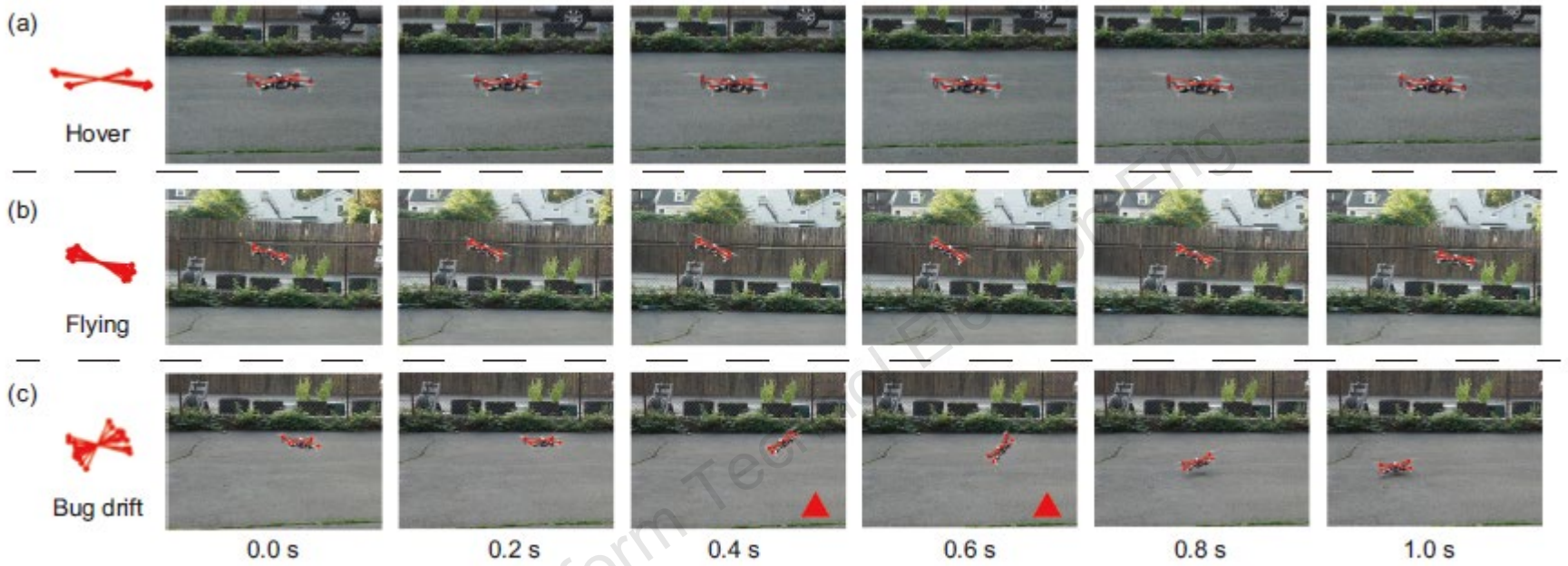

Fig. 13 Empirical comparison between systems with ((a) and (b)) or without (c) a verified SPI bus driver

Drones in Figs. 13a and 13b have installed verified device drivers. They could hover, and are able to change their attitudes and fly forward. Fig. 13c shows the situation where there are bugs in the drone's SPI bus driver, and shows greater variations of the drone's attitude compared to Figs. 13a and 13b, even if they are operated in the same manner.

Conclusions

A new software architecture and development method targeting at safety and reliability for a drone system has been proposed in this study. With the help of formal verification, several bus drivers which play critical roles in flight control were verified. Experiments in the filed tests showed that the proposed system enjoys improved reliability by eliminating the subtle bugs that can be introduced in software development.