Frontiers of Information Technology & Electronic Engineering
www.jzus.zju.edu.cn; engineering.cae.cn; www.springerlink.com
ISSN 2095-9184 (print); ISSN 2095-9230 (online)
E-mail: jzus@zju.edu.cn

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

Xiao-rui ZHU†‡1, Chen LIANG1, Zhen-guo YIN1, Zhong SHAO†2, Meng-qi LIU2, Hao CHEN2

1*Department of Mechanical Engineering and Automation, Harbin Institute of Technology (Shenzhen), Shenzhen 518055, China*

2*Department of Computer Science, Yale University, New Haven 06511, USA*

†E-mail: xiaoruizhu@hit.edu.cn; zhong.shao@yale.edu

Received Oct. 12, 2018; Revision accepted Feb. 1, 2019; Crosschecked Mar. 14, 2019

Abstract: A new hierarchical software architecture is proposed to improve the safety and reliability of a safety-critical drone system from the perspective of its source code. The proposed architecture uses formal verification methods to ensure that the implementation of each module satisfies its expected design specification, so that it prevents a drone from crashing due to unexpected software failures. This study builds on top of a formally verified operating system kernel, certified kit operating system (CertiKOS). Since device drivers are considered the most important parts affecting the safety of the drone system, we focus mainly on verifying bus drivers such as the serial peripheral interface and the inter-integrated circuit drivers in a drone system using a rigorous formal verification method. Experiments have been carried out to demonstrate the improvement in reliability in case of device anomalies.

Key words: Safety-critical; Drone; Software architecture; Formal verification
https://doi.org/10.1631/FITEE.1800636 **CLC number:** V279; TP311.5

1 Introduction

In recent years, small unmanned aerial vehicles (UAVs) or drones have drawn more and more attention because of their low cost and compact size. As small UAVs come into our daily life, safety concerns are also rising. Failures of a drone may result in severe damage to the environment and serious injury to the public (Simpson and Stoker, 2006).

Aside from maneuver mistakes, software errors in the controller are one of the main reasons for UAV failures. The fault may come from the algorithm itself or its actual implementation (Malecha et al., 2016). A lot of work has been done to improve the reliability of UAV systems. Most efforts have focused on algorithms, such as improving modeling accuracy (Leishman, 2002), enhancing the robustness of control algorithms (Lee et al., 2010), and reducing sensor errors (de Marina et al., 2012). Réti et al. (2013) proposed a hardware solution to improve the safety by developing a smart mini actuator, which integrated measurements of position and angular rate with controlling microprocessors. Few people so far have addressed bugs in the implementation of algorithms at the source code level. For a safety-critical real-time system like a UAV, this negligence could result in problems such as loss of synchronization (caused by irregular response from external sensors) and high approximation errors (caused by floating-point computation) (Malecha et al., 2016). These problems are subtle but might degrade the performance or even cause the drone to crash.

Formal verification is a technique to conduct correctness proof of a program (or the contradiction

‡ Corresponding author

* Project supported by the National Natural Science Foundation of China (No. 91648012) and Shenzhen Science, Technology, and Innovation Comission, China (No. JCYJ20160401100022706)

ORCID: Xiao-rui ZHU, http://orcid.org/0000-0003-1400-059X; Zhong SHAO, http://orcid.org/0000-0001-8184-7649

if the program contains errors) in accurate and well-formed mathematical and logical constructs. It is used to prevent subtle errors in the source code of control systems (Ricketts et al., 2015; Malecha et al., 2016; Bohrer et al., 2018). Preventing such errors would increase the reliability and safety of drone systems. In 2015, foundational verification techniques in the theorem prover Coq were applied to a quadrotor system to verify the correctness of two shims (saturation blocks), which were used to limit the velocity and height of the quadrotor (Ricketts et al., 2015). In 2016, the same research group verified a runtime monitor to provide strong guarantees about maximum velocities and accelerations of a drone (Malecha et al., 2016). Bohrer et al. (2018) designed a verified pipeline for generating concrete controller code from high-level models. However, these efforts for formally verifying control systems are not enough for a hybrid real-time drone system.

Real-time operating system (RTOS) plays an important role in scheduling real-time processes and interacting with devices. Traditional RTOSs, including Nuttx (Nutt, 2007) and FreeRTOS (Barry, 2003), perform well in real-time scheduling. Some of them also support memory protection to improve security (Wang, 2017). However, none of them has provided a formal correctness proof of its source code.

A potential source of software failures lies in the implementation of device drivers. The driver has to rely on the behavior of that device, for instance, to tell when it is ready to read or write data, or whether a previous write is complete or not. However, due to the complexity of modern hardware, it is difficult to consider all possible abnormal situations when implementing the device driver. For example, it is common for a driver to loop until some status bit on the device is set. If the device does not update this bit in time, then this delays the execution of the driver, and potentially blocks the whole system if the driver runs in the kernel mode and is not interruptible.

The main contribution of this paper includes a new software architecture for improving the reliability and safety of drone systems at the source code level by introducing formal verification techniques. In particular, the proposed architecture is based on certified kit operating system (CertiKOS) (Gu et al., 2015), which enjoys a formal functional correctness guarantee. We adopt this methodology and formally verify the device driver for a drone control system layer by layer, and demonstrate that this indeed improves its safety and reliability. The same architecture could be extended to autonomous cars, home service robots, and other safety-critical systems.

2 Hierarchical software architecture

To improve the reliability and safety of the software stack of a drone system, a new hierarchical software architecture is proposed (Fig. 1). In this architecture, an operating system kernel, CertiKOS (Chen et al., 2016), plays the central role of managing devices such as motors and sensors, and scheduling user tasks such as the control loop and the sensor fusion program.

A Raspberry Pi3 board is equipped on the drone as its main controller, which connects with multiple sensors and actuators through general purpose input/output (GPIO) pins. CertiKOS-ARM, the ARM port of CertiKOS, is installed on the board to manage these devices, either directly or through bus drivers, and to expose them to user space programs. During each control period, the sensor fusion algorithm reads from sensors to generate a reliable

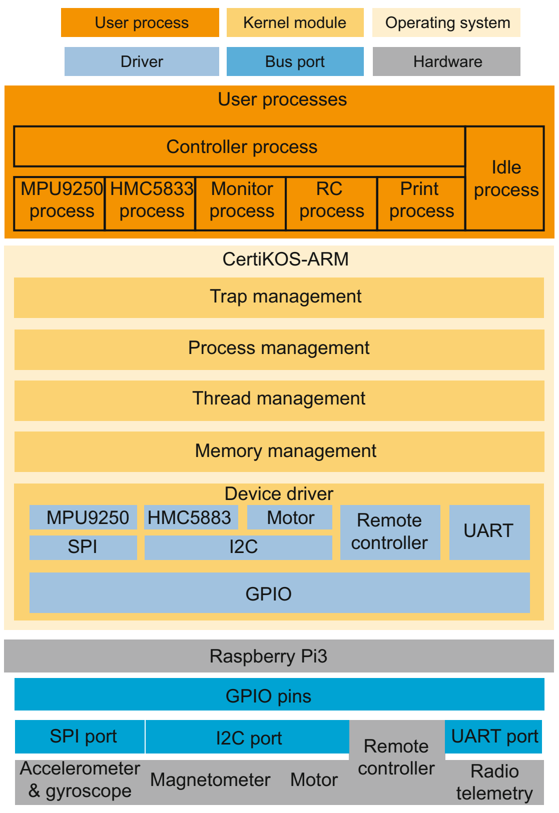

Fig. 1 Overall software architecture

attitude estimation. Then the controller decides its next movement and writes control signals to the corresponding motors. There is also a remote control (RC) task, which reads the receiver to obtain control signals from the remote controller. In this way, the reliability of a drone system depends heavily on the correctness of its device drivers.

In the proposed architecture, all software modules including the kernel and device drivers should be formally verified to ensure the functional correctness of their source code. CertiKOS has been formally verified on x86 in Gu et al. (2015), and its implementation has been ported to the ARM architecture successfully. We focus mainly on verification of device drivers for the drone system. The drone system relies on the partially verified CertiKOS-ARM, which includes modules for memory management (verified) and thread management (not verified).

3 Driver verification

In a typical drone control system, it is necessary and important to estimate the drone attitude accurately. Raw data for the drone attitude estimation are usually provided by three sensors: accelerometer, gyroscope, and magnetometer. In our system, the accelerometer and gyroscope depend on the serial peripheral interface (SPI) bus to transmit sensing signals, and the magnetometer uses the inter-integrated circuit (I2C) bus.

Following the same methodology as presented in Chen et al. (2016), driver verification can be divided into three phases. First, we build a bus model which abstracts machine registers and the physical memory into a state transition system. Afterwards, we define an abstract interface for reading and writing the bus (Fig. 2).

During the second phase, we divide the C code of the device driver into multiple layers according to their functionalities and dependencies (Fig. 2). We further convert these individual C functions into their corresponding Clight abstract syntax tree (Leroy, 2009), so that we can reason about their behaviors by using the Clight semantics, which is actually an extended semantics (Gu et al., 2015). The set of abstract syntax trees implementing a layer is called a module, i.e., M_n in Fig. 2.

Next, we abstract each C function into a Coq function (which is called a specification or a primitive), while still capturing everything we want to know about the behavior of its source code. This is achieved by following the approach of deep specifications (Gu et al., 2015). Then we define invariants for each layer, and prove that all primitives preserve these invariants, so that the higher layer will operate only on valid states of its underlay.

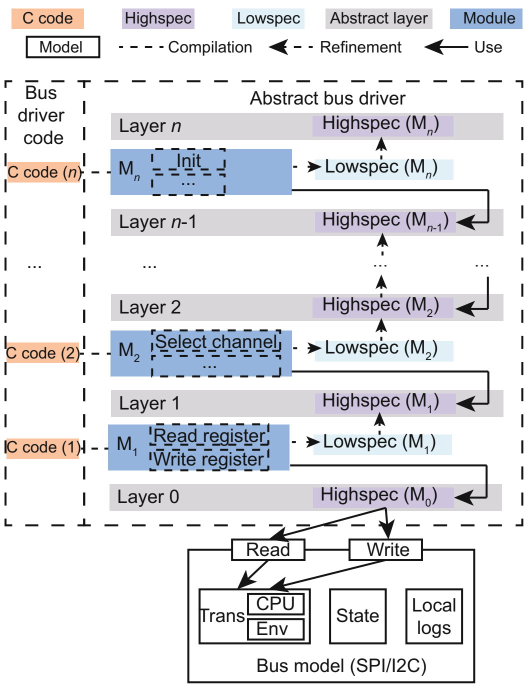

Fig. 2 Driver verification structure

It is not straightforward to prove the refinement between the module and its specification (highspec) in one step. Hence, we follow and introduce the lowspec to bridge the gap (Gu et al., 2015). While highspec focuses on the abstract states and high-level invariants, the lowspec deals with the memory state and low-level invariants. The set of highspecs constitutes the abstract layer of the corresponding module, which is relied upon by other modules. On the other hand, lowspec is used for only simplifying the refinement proof and hidden from higher layers.

The final phase is the verification of each driver, based on the bus model and abstract bus driver layers obtained from the first two steps. Two refinements have to be proved in each abstract layer (Gu et al., 2015), the refinement from lowspec to highspec and each module correctly implementing its lowspec. If any condition is not satisfied, we adjust either the lowspec or the original C code until all modules are verified. Then the deep specification framework links the verification of all layers together to achieve a verified program. This guarantees the functional correctness of our adjusted C code of device drivers, which in turn contributes to the reliability and safety of the drone system.

3.1 Bus model

The characteristics of the bus in an actual physical system depend on the I/O operations of the CPU and its interactions with the external sensor. Hence, the SPI/I2C bus could be modeled as finite state transition systems interacting with the CPU and external sensors. Different I/O operations or external sensor events lead to different corresponding changes to the state. Bus transitions (i.e., "Trans" in Fig. 2) therefore include these two types of interactions (Chen et al., 2016).

The CPU carries out read/write operations on bus registers through the I/O command. We model these operations on both the SPI and I2C buses as in Definition 1.

Definition 1 (CPU operation on bus)

$$\begin{aligned}\mathcal{O} ::=\ & \texttt{input n}\\ \mid\ & \texttt{output n v},\end{aligned}$$

where $\mathcal{O}$ ::= `input n` denotes reading a value from the register whose address is `n`, and $\mathcal{O}$ ::= `output n v` means that the CPU writes a value `v` to the register at address `n`.

The following subsections describe definitions of the state machine for I2C and SPI, and how they are updated by CPU operations and external actions.

3.1.1 I2C bus model

To formally define the I2C bus model, we first construct its abstract state.

Definition 2 (I2C bus abstract state)

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

Although the physical bus hardware is sophisticated and contains many more states and operating modes, most of them are irrelevant regarding the attached sensor, such as the 10-bit addressing mode and high-speed mode. Therefore, we fix its operation to the 7-bit addressing mode, and abstract only 10 registers (Definition 2) to formalize the state of a physical I2C bus. These include the base address of the device interface state `I2C_OA` and slave address state `I2C_SA`, which serve as identities when connecting to specific devices. We also model the data receiving buffer state `I2C_RX_DATA` and the data sending buffer state `I2C_TX_DATA` to describe the read/write buffer in a real I2C bus.

Based on Definitions 1 and 2, we define the CPU's read/write operations for I2C bus as Definition 3.

Definition 3 (I2C state transition function based on CPU operation)

```
δ_I2C^CPU(op: 𝒪) (s: I2CState) : I2CState :=
 |op = input n -> s
 |op = output n v -> s{I2C_OA: v}, if n = I2C_OA
                     s{I2C_SA: v}, if n = I2C_SA
                     ...,
```

where $\delta_{\mathtt{I2C}}^{\mathtt{CPU}}$ describes the interaction between the CPU and I2C bus, which takes the CPU operation `op`: $\mathcal{O}$ and the current state as arguments, and returns the resulting state after this operation. A read operation (`op= input n`) does not change the I2C state. A write operation (`op= output n v`) updates the corresponding field in the abstract state to `v`.

As mentioned previously, besides I/O operations issued by the CPU, external sensor events may affect the state of the I2C bus. There are three kinds of events for the I2C bus as listed in Definition 4, non-event, acknowledgment responding event, and data receiving event.

Definition 4 (I2C external sensor event)

$$\begin{aligned}\mathtt{E}_{\mathtt{I2C}}^{\mathtt{env}} ::=\ & \texttt{NullEvent}\\ \mid\ & \texttt{ACKEvent}\\ \mid\ & \texttt{RecvEvent(val}: \mathbb{Z}),\end{aligned}$$

where `NullEvent` represents a non-event in which the I2C bus is waiting for other functional events, `ACKEvent` represents the acknowledgment responding event in which the I2C bus receives an acknowledgment, and `RecvEvent` denotes the data receiving event in which the I2C bus receives an integer data `val`.

Based on Definitions 2 and 4, we model state transitions of the I2C bus triggered by external events as Definition 5.

Definition 5 (I2C state transition function based on external sensor events)

$$
\begin{aligned}
&\delta^{\texttt{env}}_{\texttt{I2C}}\texttt{(e: E}^{\texttt{env}}_{\texttt{I2C}}\texttt{)(s: I2CState) : I2CState :=}\\
&\quad \texttt{| s, if e = NullEvent}\\
&\quad \texttt{| s, if e = ACKEvent}\\
&\quad \texttt{| s}', \texttt{ if e = RecvEvent(val).}
\end{aligned}
$$

The acknowledgment responding event and the non-event will not change the I2C state. For receiving event, the I2C bus receives an integer data `val`, and copies this value to the register `I2C_RX_DATA` as

$$
\begin{aligned}
&\delta^{\texttt{env}}_{\texttt{I2C}}\texttt{(e: E}^{\texttt{env}}_{\texttt{I2C}}\texttt{)(s: I2CState) : I2CState :=}\\
&\texttt{| s, if e = NullEvent}\\
&\texttt{| s, if e = ACKEvent}\\
&\texttt{| s\{I2C\_RX\_DATA: val\}, if e = RecvEvent(val).}
\end{aligned}
$$

In the I2C bus model, an external sensor event list $\texttt{l}^{\texttt{env}}_{\texttt{I2C}}$ is also constructed to decide the order of all events being processed by the CPU. At the same time, a local event log (Fig. 2) is set up to record events which are already processed in the event list.

Once state transition functions of the I2C bus model are defined, we connect transitions caused by CPU operations with transitions triggered by external events to model the overall effect of reading/writing the I2C bus. They constitute the interface for the device driver to interact with the I2C bus.

Definition 6 (I2C bus read semantics)

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

where we first find that the next event `e` is handled by comparing the event list $\texttt{l}^{\texttt{env}}_{\texttt{I2C}}$ with local event log $\texttt{l}_i$ ($\texttt{next(l}^{\texttt{env}}_{\texttt{I2C}}\texttt{,l}_i\texttt{)}$). Then we apply the I2C state transition function $\delta^{\texttt{env}}_{\texttt{I2C}}$ on event `e` and the current I2C state `s` to obtain the next I2C state $\texttt{s}'$. The next step is to obtain the value `res` from the abstract state $\texttt{s}'$ and register address `n`. Finally, we update the I2C state again through the state transition function $\delta^{\texttt{CPU}}_{\texttt{I2C}}$. Given all above premises, semantics of reading the I2C bus is defined as $\texttt{read(n,s,l}_i\texttt{,l}^{\texttt{env}}_{\texttt{I2C}}\texttt{)=(res, s}''\texttt{, l}'_i\texttt{)}$. Similarly, Definition 7 is the write semantics on the I2C bus.

Definition 7 (I2C bus write semantics)

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

This concludes the definition of the I2C bus model, which is relied upon by the verification of device drivers explained in Section 3.1.2.

3.1.2 SPI bus model

The SPI bus is modeled by the same approach.

Definition 8 (SPI bus abstract state)

```
Record SPIState :=
 mkSPIState {
       SpiRx: ℤ
       SpiTx: ℤ
       SpiEn: bool
       SpiMs: SPI_MS
       ...
       }.
```

In the SPI bus model, integer elements, `SpiRx` and `SpiTx`, represent the data receive buffer and data transmit buffer of the actual physical SPI bus, which are abstracted from the data receive register and the transmit register, respectively. The boolean field `SpiEn` is an abstraction for modeling SPI enabling status. In summary, the SPI bus abstract state contains a total of 25 fields, and they are used in our drone control system.

3.2 Layer structure of the driver code

As mentioned at the beginning of this section, we divide the bus driver code into layers based to their functionalities and dependencies to enable the compositional verification (Chen et al., 2016). Three principles are followed during this process: (1) Similar functions, such as reading/writing a register, should be put in the same layer; (2) One layer should not contain too many functions, to make the proof easier; (3) Such layering should not change the overall behavior of the source code. We show the layer structure of the SPI bus driver, while the layering of the I2C bus driver is similar.

In Fig. 3, each block represents one module in the layer. For example, in layer `DSpiInOut`, module `RegRW` contains two functions for reading and writing

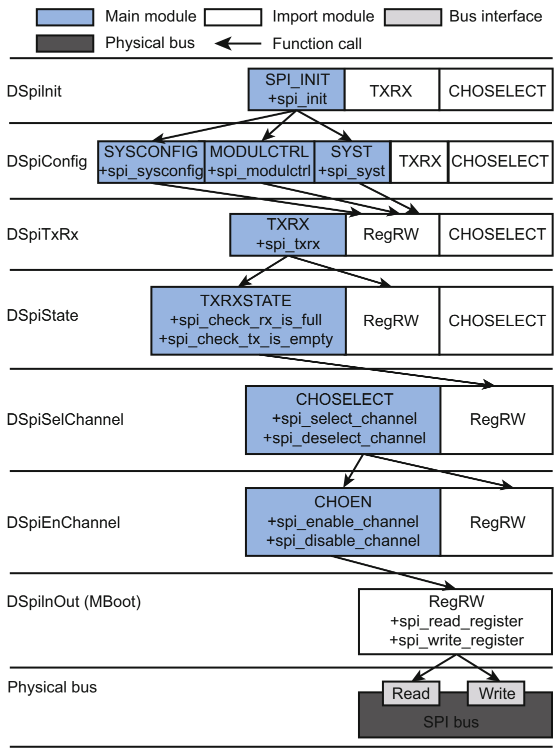

Fig. 3 Layering structure of the SPI bus driver

References to color refer to the online version of this figure

registers. The arrow between two modules indicates the calling relation between them, and a module is only allowed to call modules in the lower layer. For example, module `CHOEN` points to (invokes) module `RegRW`, and module `RegRW` points to the read and write interface of the SPI bus. The blue block indicates that functions in this module depend on at least one function in another module. The white block represents the module which is passed through from a lower layer without any modification. For example, module `RegRW` in `DSpiEnChannel` is passed through directly from layer `DSpiInOut`. Module `RegRW`, which consists of the read and write interfaces of the SPI bus, is located at the bottom of the layer architecture.

3.3 Verification of the driver

In this subsection, we follow the methodology proposed in Gu et al. (2015) to verify the SPI driver.

3.3.1 Functional correctness of the C code

We show the C code for enabling the channel and its corresponding Clight representation in Fig. 4. The main operation of the function is to write value

```
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

`ENABLE_CHANNEL` to address `CH0CTRL` to enable the SPI bus.

The workflow of proving the functional correctness of a module is elaborated in Fig. 5. Clightgen, provided by Compcert (Leroy, 2009), is used to translate the C code of SPI driver into a Clight abstract syntax tree. Then we write the highspec and lowspec of the corresponding module in Coq to establish the refinement relation.

The highspec describes the desired functionality of this module. For example, the above function `mcspi_enable_channel` is abstracted as

```
Function $\hat{\sigma}_{\texttt{mcspi\_enable\_channel}}$(abs: RData)
                             : option RData :=
  match(spi abs)with
  | SpiState _ SpiEN.en _ ->
  Some(abs{spi:(SpiState _ SpiEn.Enable _)})
  | _ -> None
  end,
```

where `RData` contains all states of the system, such as the page table and the process control block. This function updates only `spi`, and `SpiState` is an instance of `spi`. The enable bit (`SpiEn`) of the SPI bus state (`SpiState`) will be changed from the previous value to `Enable`, which describes the behavior of the SPI enable operation in the original C code. The lowspec also abstracts the behavior of each function in this module, but is specified in a way that is closer to the concrete hardware. In the case of enabling the SPI bus, it looks very similar to the corresponding highspec because only function invocation is involved. The following is the low specification of function `mcspi_enable_channel` written in Coq:

```
Inductive $\hat{\sigma}$LOW$_{\texttt{mcspi\_enable\_channel}}$(abs abs$'$:RData)
(m0:mem) :=|$\hat{\sigma}_{\texttt{mcspi\_enable\_channel}}$abs = Some abs$'$->
$\hat{\sigma}$LOW$_{\texttt{mcspi\_enable\_channel}}$(m0,abs) -> (m0,abs$'$),
```

where `RData` represents the abstract state and `mem` represents the memory state. Memory state `m0` does

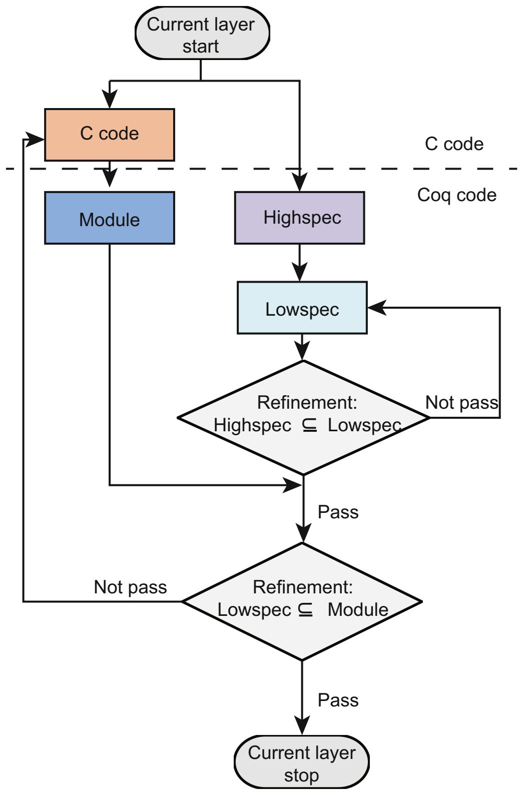

Fig. 5 Contextual refinement verification for a C function

not change since this function does not involve any direct memory operation. Thus, the overall behavior is a transition from `(m0,abs)` to `(m0,abs′)`. Then we prove the refinement between the highspec and the lowspec defined as

$$\begin{aligned}&\text{highspec} \subseteq \text{lowspec} :=\\ &\forall a, a', m,\ (a \xrightarrow{\text{highspec}} a') \wedge (a \sim m)\\ &\Rightarrow \exists m',\ (m \xrightarrow{\text{lowspec}} m') \wedge (a' \sim m').\end{aligned}$$

The highspec and lowspec may operate on different types of states, so that we use a, a', m, and m' to distinguish between them. However, we establish a relation $a \sim m$ between two states on these two different levels, which holds only if a is a valid abstraction of m. This refinement relation states that if the highspec takes one step from a to a', and its initial state a is a proper abstraction of m, then the corresponding lowspec must be able to step from m to m', where the relation $\sim$ also holds between a' and m'.

Similarly, we prove the refinement relation between the lowspec and the actual C code. Combining the above two refinements, we obtain the refinement from the highspec to the actual C code, which is exactly its functional correctness proof.

As shown in Fig. 5, it is possible during the verification process that we find that certain refinement relations do not hold. This either is due to a flaw in the specification which we need to revise and try again, or is indeed caused by a bug in the source code. In the latter case, we have to fix the bug so that the functional correctness of the source code could be verified.

3.3.2 Linking all layers together

The functional correctness proof of each layer assumes the functional correctness of the layer below it. Part of the layer architecture of the SPI bus driver is presented in Fig. 6, to illustrate how we build up the verification layer by layer.

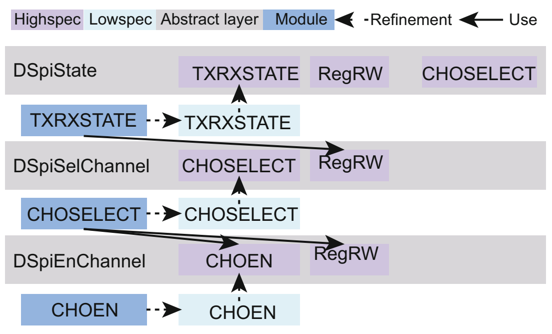

Fig. 6 Verification of the SPI driver

Module `CHOEN` is first verified, meaning the behavior of its C code indeed follows its specification. It then serves as the interface of layer `DSpiEnChannel`, which is invoked by layer `DSpiSelChannel`. Similarly, layer `DSpiSelChannel` exposes the highspec `CHOSELECT` as part of its interface, which could be used by upper layers.

The framework (Gu et al., 2015) enables us to link layers together and prove the following contextual refinement between layers. Assume that `P` is a program which uses function `CHOSELECT`. As shown in Fig. 7, the behavior of linking `P` with module `CHOSELECT` (written as `P` ⊕ `CHOSELECT`) and running them on layer `DSpiEnChannel` is equivalent to the behavior of running program `P` on layer `DSpiSelChannel` (written as `P@DSpiSelChannel`). We can write the refinement between these two executions as

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

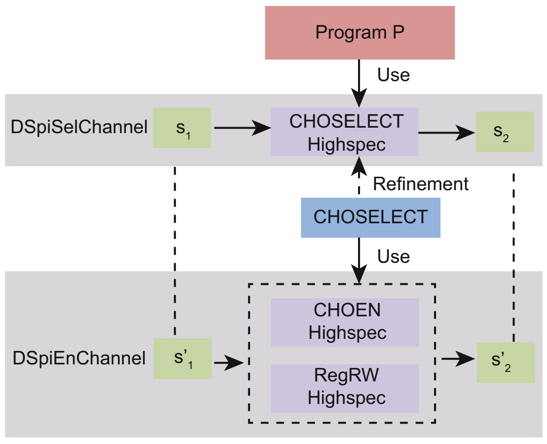

Fig. 7 Refinement between abstract layers

Fig. 8 Drone used in experiments

Once this refinement is proved, the actual implementation of function `CHOSELECT` is hidden under layer `DSpiSelChannel`, while we are still able to reason about all behaviors of program `P`.

Finally, we use Compcert (Leroy, 2009) to generate assembly code for all verified modules. Compcert carries the functional correctness property all the way down to the assembly code level (Gu et al., 2015).

4 Experiments

4.1 Methods and procedures

A drone (Fig. 8) was built for all experiments. Three basic sensors, an accelerometer, a gyroscope, and a magnetometer were used to estimate the attitude of the drone. Their configurations are listed in Table 1. A radio telemetry was used to record the flight data. Experiments were designed in this subsection to simulate erroneous situations or bugs of bus drivers. We set up the system so that bugs occur every 5–10 s, whose effect is to delay the execution of the driver code for as long as 0.2 s. This simulates the situation where the driver keeps polling for new data without enforcing any timeout mechanism. In this case, an anomaly in the device may block the driver for a long time, which in turn blocks the execution of the whole system.

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 as explained in the previous section. The second one is a system with an unverified SPI bus driver. Both of these two systems were equipped with the verified I2C bus driver.

Ten trials have been carried out with different bugs randomly occurring in the SPI bus driver. We recorded and compared attitudes of the drone (roll, pitch, and yaw) since they are the most critical metrics to its safety. The attitudes were computed by the same gradient descent method (Madgwick et al., 2011) using inertial measurement unit (IMU) data read from the SPI bus.

4.2 Results and discussion

Fig. 9 shows the roll angles of the unverified drone system. Solid lines in Fig. 9a represent the computed actual roll angles while dashed lines represent the desired values required by the remote controller. The differences between the actual and desired values (errors) are shown in Fig. 9b. Three peaks of errors are observed in the timeline 8.6 s,

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

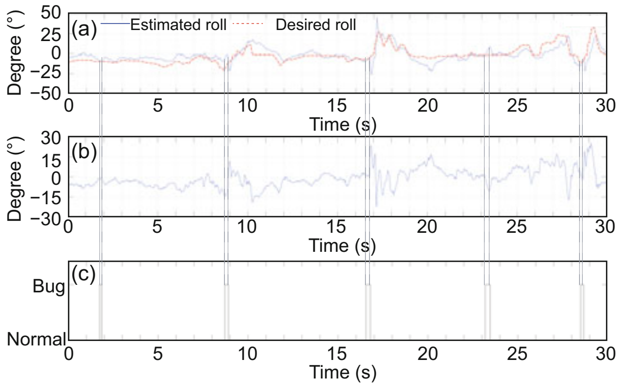

Fig. 9 Roll angle response of drone with an unverified SPI bus driver: (a) roll angle; (b) roll angle error; (c) SPI bus bug

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. Software bugs are also detected at 1.8 s and 23.2 s in the timeline (Fig. 9c). 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 it 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. The same phenomenon exists on the pitch angle (Fig. 10).

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

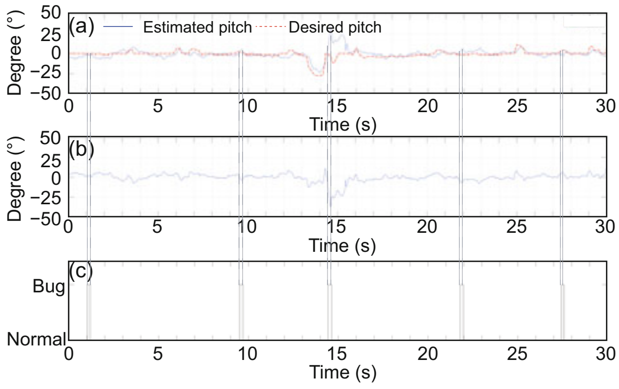

Fig. 10 Pitch angle response of drone with an unverified SPI bus driver: (a) pitch angle; (b) pitch angle error; (c) SPI bus bug

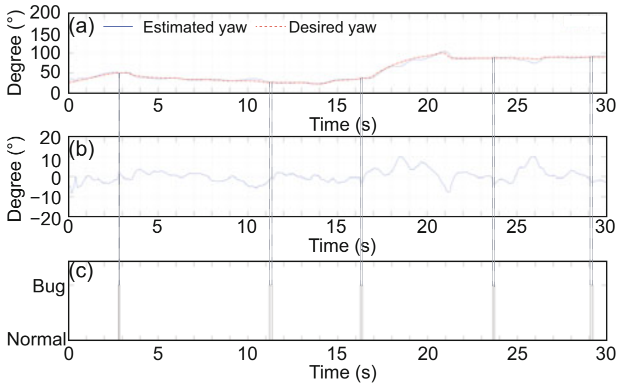

Fig. 11 Yaw angle response of drone with an unverified SPI bus driver: (a) yaw angle; (b) yaw angle error; (c) SPI bus bug

of the estimated yaw angle.

Fig. 12 shows comparison of attitude errors between these two drone systems. The existance of software bugs leads to significant differences between desired and actual pitch and roll angles.

Fig. 13 shows a series of snapshots for different drone flights in a consequent timeline. 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. Fig. 13 demonstrates that bugs in the SPI bus driver indeed degrade the stability of a drone.

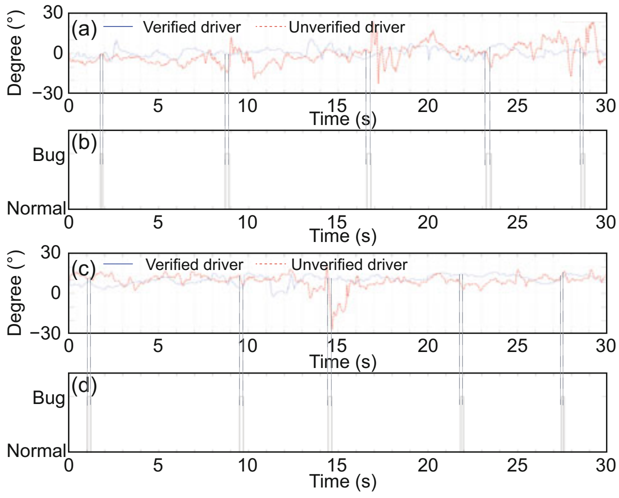

Fig. 12 Comparison of attitude errors in two drone systems: (a) roll angle error; (b) corresponding SPI bus bug; (c) pitch angle error; (d) corresponding SPI bus bug

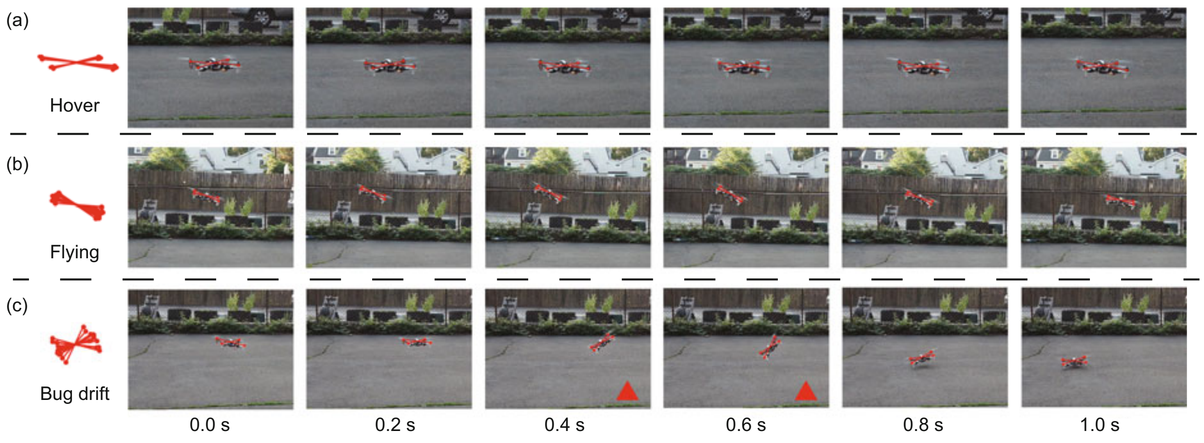

Fig. 13 Empirical comparison between systems with ((a) and (b)) or without (c) a verified SPI bus driver

5 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.

In our future work, we plan to extend the proposed architecture with virtualization support. A hypervisor could be introduced to support third-part systems without compromising the inherited safety and security by enforcing strong isolation and non-interference properties.

References

Barry R, 2003. The FreeRTOS^{TM} Kernel. https://www.freertos.org/ [Accessed on Feb. 12, 2019].

Bohrer B, Tan YK, Mitsch S, et al., 2018. Veriphy: verified controller executables from verified cyber-physical system models. Proc 39th ACM SIGPLAN Conf on Programming Language Design and Implementation, p.617-630. https://doi.org/10.1145/3296979.3192406

Chen H, Wu XN, Shao Z, et al., 2016. Toward compositional verification of interruptible OS kernels and device drivers. Proc 37th ACM SIGPLAN Conf on Programming Language Design and Implementation, p.431-447. https://doi.org/10.1145/2908080.2908101

de Marina HG, Pereda FJ, Giron-Sierra JM, et al., 2012. UAV attitude estimation using unscented Kalman filter and TRIAD. *IEEE Trans Ind Electron*, 59(11):4465-4474. https://doi.org/10.1109/TIE.2011.2163913

Gu RH, Koenig J, Ramananandro T, et al., 2015. Deep specifications and certified abstraction layers. Proc 42nd Annual ACM SIGPLAN-SIGACT Symp on Principles of Programming Languages, p.595-608. https://doi.org/10.1145/2676726.2676975

Lee T, Leok M, McClamroch NH, 2010. Geometric tracking control of a quadrotor UAV on SE(3). 49th IEEE Conf on Decision and Control, p.5420-5425. https://doi.org/10.1109/CDC.2010.5717652

Leishman JG, 2002. Principles of Helicopter Aerodynamics. Cambridge University Press, Cambridge, UK.

Leroy X, 2009. Formal verification of a realistic compiler. *Commun ACM*, 52(7):107-115. https://doi.org/10.1145/1538788.1538814

Madgwick SOH, Harrison AJL, Vaidyanathan R, 2011. Estimation of IMU and MARG orientation using a gradient descent algorithm. IEEE Int Conf on Rehabilitation Robotics, p.1-7. https://doi.org/10.1109/ICORR.2011.5975346

Malecha G, Ricketts D, Alvarez MM, et al., 2016. Towards foundational verification of cyber-physical systems. Science of Security for Cyber-Physical Systems Workshop, p.1-5. https://doi.org/10.1109/soscyps.2016.7580000

Nutt G, 2007. Nuttx Real-Time Operating System. http://nuttx.org [Accessed on Feb. 12, 2019].

Réti I, Lukátsi M, Vanek B, et al., 2013. Smart mini actuators for safety critical unmanned aerial vehicles. Conf on Control and Fault-Tolerant Systems, p.474-479. https://doi.org/10.1109/SysTol.2013.6693929

Ricketts D, Malecha G, Alvarez MM, et al., 2015. Towards verification of hybrid systems in a foundational proof assistant. ACM/IEEE Int Conf on Formal Methods and Models for Codesign, p.248-257. https://doi.org/10.1109/MEMCOD.2015.7340492

Simpson AJ, Stoker J, 2006. Safety challenges in flying UAVs (unmanned aerial vehicles) in non segregated airspace. IET Int Conf on System Safety, p.81-88. https://doi.org/10.1049/cp:20060206

Wang KC, 2017. Embedded real-time operating systems. In: Wang KC (Ed.), Embedded and Real-Time Operating Systems. Springer, Cham, Germany, p.401-475. https://doi.org/10.1007/978-3-319-51517-5_10