Formal modeling of OpenLRSng radio control link for Unmanned Aerial Vehicles (Extended Abstract)

Maciej Szreter

Institute of Computer Science, Polish Academy of Sciences

Abstract. The paper introduces a radio control link OpenLRSng for Unmanned Aerial Vehicles (UAVs) and describes its internal details. The software is written in Arduino programming environment. We describe how the transmitter and receiver work in the link, including the frequency channel hopping rule and the failsafe routine. We motivate that the simplicity of programming model and C++ language used in Arduino can enable formal modeling by timed automata. This will enable automatic translation to the code, and more efficient verification both at the level of a model and the resulting code.

Unmanned aerial vehicles (UAVs, drones) are a not a new idea, because first airplanes controlled remotely by radio were flown in the 1920s, using decomissioned IWW-era bi-planes. The radio control (RC) technology was based on analog electronic until the first decade of the new millenium. The technology used by military and RC hobbyists was simple, but prone to jamming, suffering from signal interferences, and losing control.

The advent of the embedded microcontrollers was the game changer for this area of applications. Entirely new solutions have appeared, using the digital technology to provide such advanced features as frequency hopping, customized configuration, transmission of telemetry, and much more. The cost of this revolution is that the programs for RC links are complex and should be formally verified, because potential flaws can lead to losing the control and potentially serious consequences.

In this paper we will present an attempt to modeling a modern, open source RC link OpenLRSng[1]. The code is implemented in the Arduino programming environment for the embedded systems [13], marked by the simplicity of its programming model. However, surprisingly complex systems can be programmed in this environment even for relatively simple microcontrollers.

The paper is constructed as follows. First, we present the description of the related work. In Sec. 1 we describe how the OpenLRSng link works from the perspective of its user. In Sec. 2 we describe the hardware capable of running the described link, and in Sec. 3 we explain the implementation, by referring to the pseudocode for the main functionality of the transmitter and receiver. In Sec. 4 we motivate that the OpenLRSng software can be modeled by timed automata. Sec. 5 concludes the paper and presents the directions of the future work.

Related work

Formal verification of embedded devices is an active research field, with many applications in the IT industry. There are more and more requirements introduced forcing the formal verification or testing in life-critical areas, such as aviation, automotive industry, healthcare appliances, etc. However, despite the simpicity of the microcontrollers when compared to desktop and server computers (less program and RAM memory[2], simple program semantic, precisely defined input and output behavior) the full formal verification, i.e. examining of all the program executions, is still in general not feasible.

The conceptually simplest approach is to test the program code. The verification of code is focused on testing the correctness with respect to the programming language, such as the lack of uncaught exceptions, division by zero, range violations, improper type casts, finding unused code blocks etc. There are two main approaches.

The dynamic analysis is based on the execution of the tested code. An example technique is the *Modified condition/decision coverage (MC/DC)* [9] is a structural code coverage metric, originally defined in the standard DO-178B, intended to be an efficient coverage metric for the evaluation of the testing process of software incorporating decisions with complex Boolean expressions.

[1] `https://github.com/openLRSng/openLRSng`

[2] A modern desktop computer usually has nowadays at least 8GB of RAM memory. Atmega 328p microcontroller has 2 kB RAM, which is 4 million times less!

Static analysis is a group of methods based on analysing the code without executing the program. It is complete, i.e. considers all the behaviors of the program, but may analyse also infeasible paths.

Abstract interpretation [6] is a general technique to compute sound fixpoints for programs. It is an overapproximation defining the effect each statement of the program has on an abstract machine, being simpler that the program itself. Frama-C [7] and Polyspace [12] are the most popular and advanced tools based on the abstract interpretation.

However, testing code makes it difficult to separate the issues related to programming language from the properties related directly to the contents of the program. This problem is tackled by the model-based analysis, representing the program by a formalized model. It can be obtained from the code and/or translated to the code.

There are many papers showing the verification of abstract models. An example is [10]. The main problem is how to obtain such models and to what extent they represent the actual programs.

Very little research has been devoted specifically to Arduino, with focus on exploiting the simplicity of this environment (for example, lack of threading, simple system libraries, relatively small program memory for many platforms). The programming and simulation system Matlab/Simulink [5], enabling the definition of programs in the formally defined language of Stateflows. Such specifications can be verified, and automatically translated to the executable Arduino code. The resulting programs can be also verified using the program analysis techniques. However, there are significant restrictions. The Stateflows may contain C++ code, what restricts the formal verification to simple properties, because these code fragments do not have the formal semantics. The program analysis techniques not use any information about the models when verifying the results of translation of these models. In practice, Simulink is used rather for modeling and programming devices simpler than those used in UAVs area.

The verification of Arduino programs is no different from the general perspective presented above. [1] describes the general idea of model-based development in the context of embedded systems, but without any details about the concrete hardware. [2] describes an application of model-based approach to developing a hand prothese running Arduino software. A Simulink model is used for modeling a classifier processing signals from the patient sensors.

[8] shows a translation of a train controller system to Uppaal, verification of temporal properties It is not designed for the Arduino platform, thus not taking advantage of Arduino model. There is no verification of the resulting code, nor connecting the model and code level verification. [3] performs is another approach doing formal verification on Simulink.

There are only few approaches to formal verification of UAV systems. They are mostly focused on flight controllers. For example, [11] presents an attempt to implement a flight controller (autopilot) system. It only controls the roll axis. The aircraft dynamics is simulated on the X-Plane flight simulator, from which data are sent and received to a microcontroller modeling the autopilot. There is no verification shown, only a single response to the aileron deflection.

1 General link description from the user perspective

From the general perspective, the link can be seen as two devices, the transmitter (Tx) and receiver (Rx) connected over a bi-directional lossy channel. Here, this channel is provided by a pair of transceiver modules capable of transmitting sequencially the digital data in the form of integer numbers. The general setup is shown in Fig. 1. Tx is connected to the operator's console, getting from it the control input and possibly transmitting back the telemetry data. Rx is installed in UAV, outputting the received data to servo motors, and possibly also to a flight controller. In the latter case, the controller may provide the telemetry data to be transmitted to the operator.

The most important function of the link is to transmit the control signal from Tx to Rx. This signal is emitted by the operator's console at the basis of movements of sticks and switches, in the form of a fixed number of channels. Each channel is represented by an integer number from the range of $[0, 255]$ and drives a single control authority such as elevator, ailerons, flight controller modes, etc. The standard form of providing this signal is Pulse Width Modulation (PWM), with the value represented by width of impulses. Another form is Pulse Position Modulation Sum (PPM-SUM), representing the value of each channel by the delays between consecutive impulses, thus packing all the channels into a single signal.

Another important link feature is *failsafe.* After pressing a button on Tx while the link is working, the current values of control signals will be saved in Rx. These values will be emitted after losing the radio link (for example, after jamming a radio signal, a failure of Tx, etc), and allow for determining by the flight controller the desired action, such as returning to launch site.

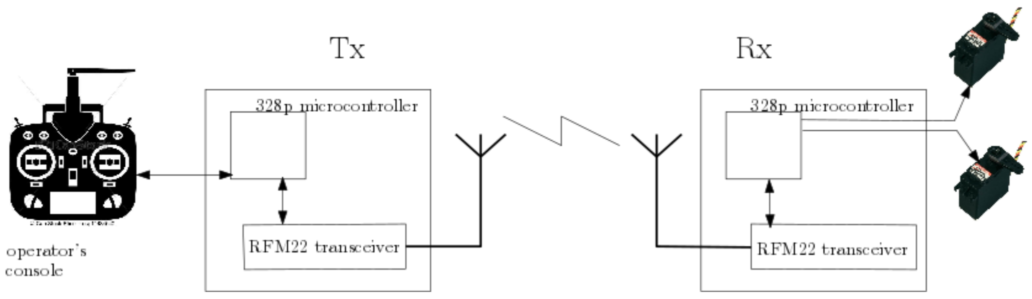

Fig. 1. The general setup of the control and telemetry link.

Yet another function of the link is telemetry, i.e. providing the data connection between Tx and Rx. One of available forms of this link is the standard serial port. It allows for providing the information about the current position and state of UAV, and more advanced control by the operator than by PWM control channels.

The link periodically changes working frequency cyclically going trough predefined channels (channel hopping). This is done so that if some frequency range is busy or deliberately jammed, it would not disable the link. It also allows more independent links to be active in the same area. The selection of channels depends on so called magic number, which can be given by the user or generated in a random way.

The user can directly specify many parameters, such as the channel frequencies, telemetry data rate, behavior in case of losing the link, etc. Tx can be connected to a desktop computer, using a TTL-USB adapter. This allows for configuring Tx directly either from a console-based interface or from a GUI being a Chrome application, and configuring Rx over the radio link.

Binding In order to establish a link, the Tx and Rx modules first need to be put into the binding mode (by pressing the button while powering on, for each module). Tx emits the *magic code*, which can be entered by the user or generated randomly. Rx in the binding mode accepts the first binding request it receives. After binding, both Tx and Rx have their transceivers programmed. Tx stores in its EEPROM memory the channel frequencies and other link parameters, and sends them to Rx which also stories them in its EEPROM memory.

2 Hardware

In general, the same hardware can be used either as Tx or Rx. The hardware consists of the two main components: the Atmega 328p microcontroller and RFM22b transceiver. The photos of a device capable of running the software are shown in Fig. 2, including the printed circuit board with these elements.

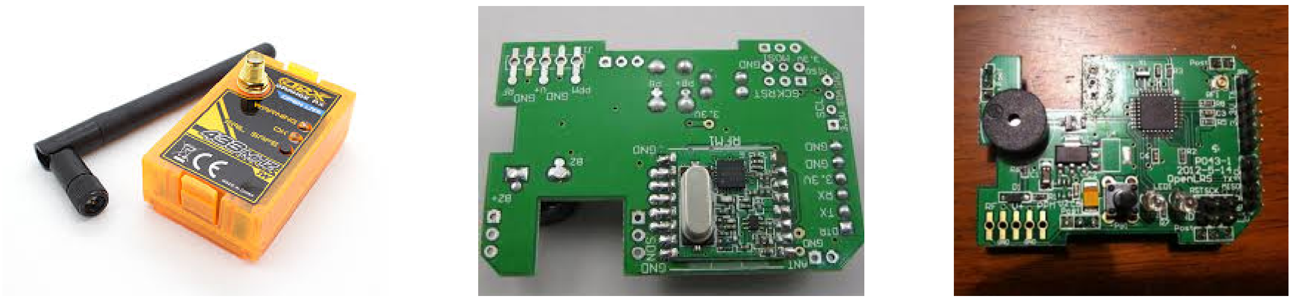

Fig. 2. Orange LRS device capable of running OpenLRSng software (left), RFM22b transceiver (middle), Atmega 328p microcontroller and helper elements (right).

Transceiver is an electronic device capable of transmitting and receiving data by using radio waves, for some range of frequencies (for example, 432-434 MHz). It is connected to the microcontroller by SPI interface, and enables the following operations:

– transmitting a data packet sequentially,

– receiving of a data packet sequentially,
– detecting if a packet is available for reading,
– changing frequency,
– configuring several parameters, such as CRC checking.

Atmega 328p is an 8-bit microcontroller produced by Atmel. It has EEPROM memory retaining stored data without the power supply, and RAM memory for program execution and dynamic variables. It can be programmed in several programming languages, including an assembler, Basic and a number of C/C++ dialects. A popular choice among developers is Arduino, a solution for embedded systems, available for several processor architectures and abstracting away the specific features of every architecture. The programming language in Arduino is C++ with some restrictions concerning both the language itself and the system libraries, and extended with constructions relating to the hardware, such as reading a voltage from a node, setting a voltage, There are system libraries, performing tasks such as providing communication with peripheral devices over I2C or SPI protocols, Bluetooth or TCP/IP internet capabilities (if corresponding devices are available).

The Arduino semantic model introduces the significant restrictions on the programs. Only a single program can run on the controller. It may contain only a single thread, consisting of a *loop*() function executed cyclically. Function *setup*() is called once after starting the device. The program cannot modify its code.

3 Software

In order to keep things simple, we present a basic working fragment of the original software, skipping (surprisingly many as for the simplicity of the hardware) functionalities such as the serial port telemetry supporting several telemetry protocols, changing profiles (up to four setups written in EEPROM memory), diversity, i.e. using two or more receivers (possibly with antenna of different types) connected with I2C link in the master-slave architecture, signal strength reporting, configuring the devices via serial port and console or GUI, remapping inputs and outputs into several pins for different hardware products, signalling the internal state and link loss by buzzer and LEDs, changing the transmitter power, and a beacon for seeking the lost UAV. What we show is a RC control link with channel hopping and failsafe. We also abstract away some technical details such as reading and storing data into EEPROM memory, or generating and reading the PPM-SUM signal.

OpenLRSng is entirely programmed in C++ in the Arduino environment, using only the standard Arduino libraries. The code consists of 14 files with around 5.5k lines of code (in the full version).

3.1 Transmitter (Tx)

First we describe the Tx software, but some of its fragments are common with Rx. Alg. 1 presents the pseudocode for Tx. In every execution of the control loop, Tx first gets the control signal from the transmitter$_{[7]}$[3] in the form of $numOfChannels$ integer numbers representing the consecutive channel values, put in an array. This procedure is based on the interrupts and timers, measuring the time between pulses at the value at the digital input, and detecting their slopes. We do not present it here.

Then, we check if the button is pressed$_{[8]}$, by checking the value at one of the input of the microcontroller, connected via the button to the ground (a more advanced solution would take care of possible jitter and introduce a timeout). If the button has been pressed, the current channel values should be transmitted as failsafe values. This is signalled by setting the first value of the packet to $0x01_{[9]}$. Otherwise, the normal data are transmitted and the $0x00$ value is set as the first byte$_{[11]}$. Then, the values of consecutive channels are copied to the send buffer$_{[13]}$. We use a separate buffer, because in the full version of the protocol not only control data can be transmitted.

Then it is tested if the next transmission is pending$_{[14]}$, by comparing the difference between the current value of the internal clock (returned by *millis*() system function call) and *lastSendTime* variable to the parameter represented by *getInterval*(), either set by the user or the default value. The shorter this interval is, the more data can be sent, but it is more likely that some packets will be lost. If the period has passed, we save the current time in the *lastSendTime*$_{[15]}$. Then, the packet is prepared and sent, without acknowledging at the level of packet contents$_{[16]}$. After performing the transmission, the transmitter increments the variable representing the frequency channel, modulo the

[3] The notation$_{[n]}$ refers to the line n of the described algorithm.

number of channels[17] and hops to it[18], effective for the next transmission. The default interval value is 30 miliseconds. It is limited to be not less than 20 miliseconds. This is much more then the average execution time of the main loop.

Algorithm 1: Tx code

```
1  $const\ int\ buttonPin = 12;$                                   // the digital input for the button
2  $const\ int\ numOfChannels = 8;$                                // number of control channels
3  $byte\ PPM[numOfChannels];$                                     // table for storing failsafe
4  $byte\ tx\_buf[numOfChannels + 1];$                             // transmission buffer
5  $long\ lastSendTime = 0;$                                       // time of last transmission sent
6  $loop()$
7  $PPM = getPPMvalues()$ ;                                        // read the input signals
8  if $\neg digitalRead(buttonPin)$ then
9      $tx\_buf[0] = 0x01$ ;                                       // failsafe request
10 else
11     $tx\_buf[0] = 0x00$ ;                                       // normal data
12 for $i = 1; i \leq numOfChannels; i++$ do
13     $tx\_buf[i] = PPM[i]$ ;                                     // copy the data to the buffer
14 if $millis() - lastSendTime > getInterval()$ then
15     $lastSendTime = millis()$
16     $async\_send(tx\_buf, packetLen)$
17     $RF\_channel = (RF\_channel + 1)\%numOfChannels$ ;          // increment the channel number
18     $setFreq(RF\_channel)$ ;                                    // set the frequency
```

3.2 Receiver (Rx)

Algorithm 2 shows the pseudocode for Rx. It is more complex than for Tx, because Rx tries to synchronize to with Tx while the latter transmits packets and hops channels at the constant period. The idea of the synchronization is shown in Fig. 3. Rx makes either 'short' or 'long' hops. Note that the hop times in general are not synchronized between Rx and Tx.

In the *setup* function, the failsafe values are read from the EEPROM memory[8]. In the main loop, the receiver first checks if a packet has been received by the transceiver[10]. If so, it stores the receiving time[11] and sets the variable *linkAcquired*, true fo the link being alive[12]. The received packet is copied to the read buffer[15]. Then, it is checked if there has not been a failsafe request[16]. If yes, the received values are stored in a table[17] and EEPROM memory[18]. Otherwise, the received values are decoded and set into outputs[20] (we do not give the details, the signals can be translated to PWM form for a single channel, or to PPM-SUM for all the channels).

Then, Rx tries to synchronize with Tx. If the Boolean variable *linkAcquired* is true (meaning that the link has been already established), and the basic period since receiving the last packet has elapsed[22], it means that a packet has been lost. The variable *numOfLostPackets* representing the number of lost packets is implemented[23] and the need to change the channel is signalled[24]. If the number of lost packets is equal to the number of channels[25], the *linkAcquired* variable is set to false[26] and the failsafe is activated: for every channel[27] the failsafe value is set[28].

If the link is not active (either it hasn't been established or it has been lost) and the prolonged period (base period multiplied by the number of channels) has elapsed[29], the channel change will be requested[30]. Then, if needed[31], the actual channel hop is performed[32,33] in the way analogous to Tx. Finally, *willHop* variable is reset[34].

4 Formal modeling

Now we motivate that the code described in the previous section can be modeled as timed automata. The most important fact is that there is a single thread, which can be represented by a state of the network of automata.

Moreover, the code consists of basic C++ control operations, such as *if*, *for*, *while*, operations on integer and array variables, plain-object datatypes, and a set of calls to the Arduino libraries with simple and well defined semantics, such as reading an input (for example, to read a button state), writing to an input (for example, for switching a LED diode), communication over SPI protocol, setting PPM-SUM values, etc. There are no more advanced C++ features such as dynamic polymorphism,

Algorithm 2: Rx code

```
const int numOfChannels = 8 ;                                  // number of transmitted channels
int PPM[numOfChannels] ;                                       // the values of each channel
int rx_buf[numOfChannels + 1] ;                                // buffer for received packets
int failSafe[numOfChannels] ;                                  // the failsafe values
long lastRcvTime = 0 ;                                         // last successful packet receive time
int numOfLostPackets = 0
setup()
failsafe = readFromEeprom(FAILSAFE, numOfChannels) ;           // read from EEPROM the failsafe values
loop()
if rx_dataAvailable then
    lastRcvTime = millis();
    linkAcquired = true
    numOfLostPackets = 0
    for i = 0; i < dataSize; i++ do
        rx_buf[i] = getChar() ;                                // read the packet byte by byte
if rx_buf[0]&0x01 then
    failSafe[i] = PPM[i]
    saveToEeprom(FAILSAFE, failSafe, numOfChannels)
else
    PPM[i] = rx_buf[i] ;                                       // set the output signals
if linkAcquired then
    if millis() − lastRcvTime > getInterval() then
        numOfLostPackets++
        willHop = true ;                                       // make 'short' hop
        if numOfLostPackets == hopCount then
            linkAcquired = false
            for i = 1; i < dataSize; i++ do
                PPM[i] = failsafe[i] ;                         // activate failsafe
if ¬linkAcquired && millis() − lastRcvTime > getInterval() * hopCount  then
    willHop = true;                                            // make 'long' hop
if willHop then
    RF_channel = (RF_channel + 1)%hopCount ;                   // modulo number of hop channels
    setFreq(RF_channel)
    willHop = false
```

complex type system, exceptions, templates. Even the dynamic allocation of variables is recommended to be avoided, given the very restricted size of the heap. There are also no fragments written in other languages, such as assembler.

Originally Uppaal has been supporting only the timed automata, but later it has been introduced a limited support for imperative code. Users can define functions and call them from the automata [4]. However, the Uppaal development is stagnating (the last version is from 2010). There are some restrictions. For example, they lack the real numbers (which are not present in OpenLRSng link, but are heavily used in flight controllers).

5 Conclusions and future work

The paper provides a general description of a modern digital radio control link for UAVs. It presents a preliminary analysis of its structure, explaining that it is suitable for formal modeling and verification. We have written the general structure of the link as the pseudocode. We have determined that it is suitable for modeling by using a language with precise formal semantics, such as Uppaal timed automata. In the full version of the paper, we will provide a model containing as much information as possible.

Moreover, the model checking available in Uppaal and similar tools is also insufficient for the task of verification of embedded systems. It is very sensitive to the length of the considered unfoldings of the models, thus having problems with long sequences of relatively simple instructions. This gives way to the new verification approaches making use of the abstraction and other reduction techniques. For example, it would be preferable to verify the properties such as *"if the link is lost, the failsafe values saved before will be sent to the output"* or *"if the failsafe is activated, the link will be reestablished after regaining the radio transmittion"*. The verification of these properties would need to focus only on the relevant parts of the code, dealing with failsafe and link loss detection, abstracting away the internals such as telemetry, configuration via GUI, switching profiles, etc.

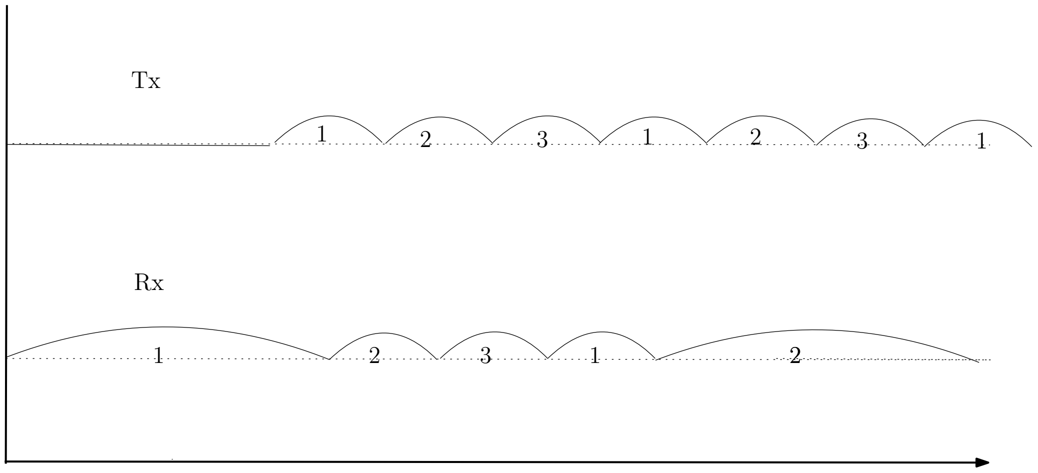

Fig. 3. Channel hopping: constant base period for Tx. Rx is trying to catch up (extended period) and follow Tx (base period). There are three channels, the numbers denote the channel currently used.

Next steps will be extending the automata formalism so that it will be more suitable for representing complete Arduino programs. The automatic Arduino code synthesis will be proposed from these automata.

References

1. Model-Based Development of Embedded Systems. Tech. Rep. TUM-I0204, 2002.
2. Attenberger, A., and Buchenrieder, K. An arduino-simulink-control system for modern hand protheses. In *Artificial Intelligence and Soft Computing - 13th International Conference, ICAISC 2014, Zakopane, Poland, June 1-5, 2014, Proceedings, Part II* (2014), L. Rutkowski, M. Korytkowski, R. Scherer, R. Tadeusiewicz, L. A. Zadeh, and J. M. Zurada, Eds., vol. 8468 of *Lecture Notes in Computer Science*, Springer, pp. 433–444.
3. Barnat, J., Brim, L., Beran, J., Kratochvila, T., and Oliveira, I. R. Executing model checking counterexamples in simulink. In *Sixth International Symposium on Theoretical Aspects of Software Engineering, TASE 2012, 4-6 July 2012, Beijing, China* (2012), T. Margaria, Z. Qiu, and H. Yang, Eds., IEEE Computer Society, pp. 245–248.
4. Behrmann, G., David, A., Larsen, K., Håkansson, J., Pettersson, P., Yi, W., and Hendriks, M. UPPAAL 4.0. In *QEST* (2006), IEEE Computer Society, pp. 125–126.
5. Beucher, O. *MATLAB und Simulink (Scientific Computing).* Pearson Studium, 08 2006.
6. Cousot, P., and Cousot, R. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Los Angeles, California, 1977), ACM Press, New York, NY, pp. 238–252.
7. Cuoq, P., Kirchner, F., Kosmatov, N., Prevosto, V., Signoles, J., and Yakobowski, B. Frama-c: A software analysis perspective. In *Proceedings of the 10th International Conference on Software Engineering and Formal Methods* (Berlin, Heidelberg, 2012), SEFM'12, Springer-Verlag, pp. 233–247.
8. Jiang, Y., Yang, Y., Liu, H., Kong, H., Gu, M., Sun, J., and Sha, L. From stateflow simulation to verified implementation: A verification approach and A real-time train controller design. In *2016 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS), Vienna, Austria, April 11-14, 2016* (2016), IEEE Computer Society, pp. 231–241.
9. Kandl, S., and Chandrashekar, S. Reasonability of mc/dc for safety-relevant software implemented in programming languages with short-circuit evaluation. *Computing 97*, 3 (2015), 261–279.
10. Rataj A., Woźna B., Z. A. A translator of java programs to tadds. In *CSP '08 17 th International Conference on Concurrency, Specification and Programming. Gross Vaeter See near Berlin, 29th September – 1 October 2008, pp. 524-535* (2008).
11. Ribeiro, L. R., and Oliveira, N. M. UAV Autopilot Controllers Test Platform Using Matlab/Simulink and X-Plane. In *40th ASEE/IEEE Frontiers in Education Conference* (Oct. 2010), IEEE.
12. Schneider, J. Tracking down root causes of defects in simulink models. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering* (New York, NY, USA, 2014), ASE '14, ACM, pp. 599–604.
13. Wheat, D. *Arduino Internals*, 1st ed. Apress, Berkely, CA, USA, 2011.