Architectural Software-Hardware Co-Modeling a Real-World Cyber-Physical System: Arduino-Based ArduPilot Case

Sergey Staroletov
Polzunov Altai State Technical University, Barnaul
Institute of Automation and Electrometry, Novosibirsk
Siberia, Russian Federation
serg_soft@mail.ru

***Abstract*—The study of good practices of architectural organization software systems for real-world cyber-physical systems (CPS) is possible to conduct on solutions with open-source code, which are developed by large communities of enthusiasts. Such solutions have been tested many times on real devices in various natural environments. The construction of models using the program code allows us to understand stable architectural solutions, present them in a graphical form and take a look at the proper organization. While UML models are suitable to represent relations between classes and show software design patterns, to create reliable software for CPS one should study patterns on the organization such systems that take into account both software and hardware parts.**

AADL (Architecture Analysis & Design Language) aims to establish clear, generally accepted semantics to express architectural models of interconnected hardware components as well as system software structure. Using the semantics, we can easily obtain graphical representations of the models to apply in the hardware design process. Also, having a formal architectural description, we can validate some valuable system properties.

In this paper, we create extended models for hardware parts and software components of a cheap reprogrammable controller for RC-based DIY systems based on the ArduPilot Mega board. The models are fully based on the actual open-source code, existing sample models and reference datasheets. We review most of the hardware and software parts, discuss the scheduler, all running tasks and their relations. Then we discuss our solution with interconnected distributed controllers using SPI and CAN bus connection.

I. Introduction

The life of modern people is unthinkable without the artificial cyber-physical devices that surround them. One example of these devices is quadcopters (or quadrotors), which are successfully used for videography, weather measurement, or monitoring agricultural lands [1]. On the other hand, in many countries, the use of such devices in places where people appear is prohibited due to the possibility of falling and their general low reliability.

In this paper, we proceed to model both software and hardware of the popular ArduPilot Mega controller, which is based on an Arduino-compatible board. This solution represents the cheapest flight controller, the code of which is entirely open. Thanks to the broad support of the community, this project can be considered one of the most straightforward, successful and energy-efficient solutions for performing simple flight or driving functions using self-assembled low-weight vehicles (quadcopters and rovers), from which a quick response is not required. Moreover, such devices are ideal for teaching children in STEM centers [2]. At the same time, due to platform limitations, the solution is the most balanced and shows examples of the professional architectural design of system software for embedded systems (that echoes the approach of design patterns for object-oriented programs [3]).

Although the system has Arduino roots, the code is a normal C++ solution that clearly uses all the necessary processor capabilities. Such decisions need to be learned, which is our motivation for writing this article.

Studying the internal structure of the ArduPilot project [4], we concluded that in order to assure the safety of such devices, it is necessary to know about the internal architecture of both software and hardware, up to the methods of exchanging with specific hardware components and the used pins. Accordingly, here we come to the concept of *co-modeling* [5]. We assume here that the *Architecture Analysis & Design Language* (AADL) is most suitable for such a process. The language is successfully used to express complex systems, for example, in automotive [6] and aero domains [7] and is accepted in communities that deal with safety [8]. We see that AADL contains significant opportunities to describe devices with their input/output pins, data buses for the intercommunications on one side, and tasks and their interaction methods on the other. Simultaneously, the language introduces a simple syntax of property sets, and open editing tools have been developed to support the AADL models creation. In addition, there is an unambiguous transition from textual models to a graphical representation, which allows us to interpret the models visually. Such a visual representation and the means of validation make it deeper to reason about the quality of the architectural solution being created.

The project is being implemented within the framework of studying the construction of cyber-physical systems using models [9]. Some preliminary results were discussed at the national OSDAYs [10] as well as the RTCSA 2021 [11] conferences.

ISSN 2305-7254

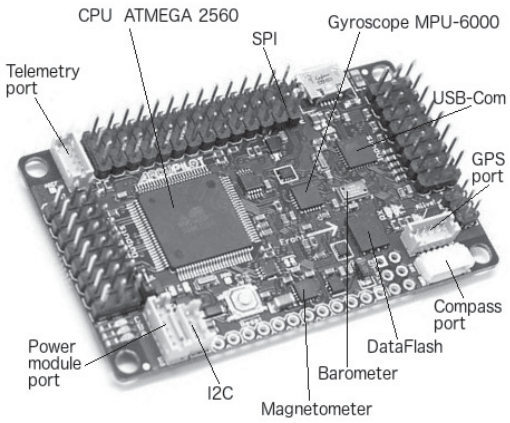

Fig. 1. Components of APM 2.6. from our study in paper [4]

II. Preliminaries: ArduPilot, AADL syntax and related open-source toolset

A. ArduPilot and APM

ArduPilot project [12] is an open-source project providing free software to command small DIY multicopters. With its ArduCopter codebase, the project introduces a high-level abstraction to operate hardware components for them (as well as for rovers and planes, refer to Ardurover and Arduplane codebases, respectively). The source code of control is based on the latest advances in research of the world quadrotor community, so it is an immeasurable base for tracking recent achievements in it. Inside, its code consists of HAL (Hardware Abstraction Layer) to run on different hardware platforms, the main scheduling loop and a set of various libraries that provide some code to communicate with particular devices, mathematical calculations and control algorithms [4].

In the beginning, ArduPilot/ArduCopter software was designed to be used accompanying with APM controller hardware ("APM" stands for *ArduPilot Mega* and "Ardu" stands for *Arduino*), so the project goal was to utilize the Arduino Mega board for the flight controller.

Consider in brief the APM 2.6/2.8 controller structure. The controller was designed in 2011-2012 [13] as a result of initial ArduPilot project development by 3D Robotics company, and then the non-profit organization *ardupilot.org* was established. Today such a version of the controller is outdated, but the APM board is cheap and available on the common market, so it is a large subject to research.

The components of the controller are shown in Fig. 1.

Onboard it comprises:

- ATmega 2560 processor (8bit) [14];
- barometer MS5611 [15], SPI connection;
- 3-axes magnetometer (compass) HMC5843 [16], I2C connection;
- 6-axes gyroscope and accelerometer MPU-6000, SPI connection;
- GPS: UART external interface;
- Frsky telemetry: UART external interface;
- an additional external SPI interface.

B. AADL: introduction

Architecture Analysis & Design Language is primarily targeted at safety-critical, real-time systems where sensors and actuators are tightly coupled with software components and facilitate analysis of interaction between hardware and software components [17]. Using tools described later, it allows system modeling engineers to:

- develop a top-level system design;
- see a graphical representation of a textual model;
- generate code using tools such as Ocarina [18], [19];
- automatically validate properties of developed systems;
- create certified solutions.

In essence, AADL acts like "executable UML", but at the very top level and it is primarily designed not for expressing algorithms, but for describing systems and components with additional safety requirements. The language offers to describe models as sets of extensible properties for specifying both hardware and software parts.

A model in AADL usually consists of the following sets of properties:

- hardware (device, processor, bus, memory, virtual bus (protocol), virtual processor);
- software (data, subprogram, thread, process);
- hybrid (whole system, abstract category).

In this paper, we model the APM board as a composition of devices, connected through their input/outport ports. They can represent device pins connected via tracks on the board, possibly using data buses. Each device is modeled as a set of properties and a set of features. The properties can represent the tuple of device state while the features model device connections. We use the following conventions for building the features:

- data port is used for logical data connections;
- event port is used for electrical connections (vcc/gnd).

C. AADL: syntax

Now we turn to the question of understanding the AADL syntax. Entity descriptions in it are usually represented by setting properties (name => value) and input/output ports. At the same time, the inheritance of sets of properties and ports is supported. For example, we can describe the ATmega 2560 processor as:

```
processor ATMEGA2560 extends Processors::
    Generic_CPU
features
PA4: in out data port;
PA7: in out data port;
--...
GND: in event port;
VCC: in event port;
AREF: in event port;
RESET: in event port;
```

```
properties
Deployment::Execution_Platform =>
   Native;
Processor_Properties::Processor_Family =>
    AVR;
Processor_Properties::Endianess =>
   Little_Endian;
Processor_Properties::Word_Length => 8
   bits;
Processor_Properties::FPU_Present =>
   False;
Thread_Limit => 1;
Processor_Properties::Processor_Frequency
    => 16 Mhz;
end ATMEGA2560;
```

In this case, we describe the processor pins and its properties according to the datasheet (in this case, [14]). Property sets are defined by the user and can be shipped in libraries (for example [20]), and their usage depends on further analysis tasks. With this description of the processor, it can be declared in the entire system and attached to other components by setting connections between pins.

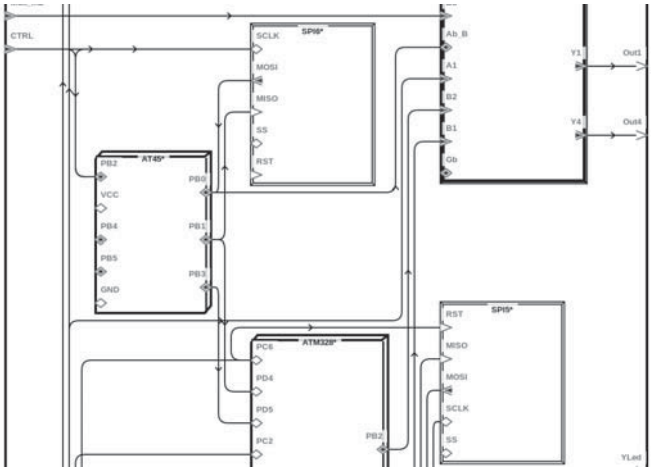

Fig. 2. A fragment of ArduPilot board model from the open library of AADL models [20], [21]

D. OSATE, MASIW and Ocarina

One of the most popular tools for editing AADL is OSATE (*Open-Source AADL Tool Environment*), which has been developed by the CMU-SEI [22]. This tool supports the latest version of the language and is mostly stable but it is still in a research project state. OSATE includes a textual and graphical editor based on the Eclipse platform as well as different tools (e.g., for latency or safety analysis).

A competitor for OSATE is the MASIW tool [7], which is being developed by ISP RAS, and it is promised to be a toolbox that comprises collecting system requirements, model creation and editing, validation and report generation.

Ocarina tool is a command-line code generator from given AADL models, it is able to create system software structure targeting various platforms with standardized APIs. Also, it includes a set of sample AADL models that can be used for education and as sources for users' extensions.

III. Related work

Analyzing existing open-source AADL models, we found a model for the ArduPilot board inside the library of reusable AADLv2 components [20]. A visualization of the model is presented in Fig. 2. We can remark that the model uses a lesser processor ATmega 328 (unlike 2560 on the actual APM board). Devices such as a barometer, accelerometer and others are not modeled. Instead of data buses, the authors use the concept of the device. It seems that some earlier version of the ArduPilot board was modeled here; that is, this model does not correspond to actual equipment. However, this library is helpful for training in AADL modeling, and we have taken the models from it as a basis for creating more adequate models discussed in this article.

IV. Analysis of the ArduPilot internals

The Arducopter project code for the interested board is located in the 3.2.1 branch [23] of the ArduPilot project. It is represented as a set of C++ classes that are compiled using the gcc toolchain for AVR. It should be noted that compilation with modern compilers requires a source code patch like this:

```
--- a/libraries/AP_Progmem/AP_Progmem_AVR.h
+++ b/libraries/AP_Progmem/AP_Progmem_AVR.h
@@ -17,11 +17,11 @@ typedef struct {

 #undef PSTR
 /* need to define prog_char in avr-gcc 4.7 */
-#if __AVR__ && __GNUC__ == 4 && __GNUC_MINOR__ > 6
+#if __AVR__ && __GNUC__ == 5 && __GNUC_MINOR__ > 0
 typedef char prog_char;
 #endif
 /* Need const type for progmem - new for avr-gcc 4.6 */
-#if __AVR__ && __GNUC__ == 4 && __GNUC_MINOR__ > 5
+#if __AVR__ && __GNUC__ == 5 && __GNUC_MINOR__ > 0
 #define PSTR(s) (__extension__({static const prog_char __c[] PROGMEM = (s); \
         (const prog_char_t *)&__c[0]; }))
 #else
```

To support various hardware, this project uses an approach that separates hardware-dependent entities (HAL) from all of the underlying code. In Fig. 3, we present a simplified class diagram of the principal items of the Arducopter project in the form of a UML class diagram [24]. It should be noted that such diagrams are also architectural but united exclusively to object-oriented software.

Analyzing this diagram, we note that the *Bridge* design pattern [3] was used, the main entities were identified in the form of abstract classes (*UARTDriver*, *I2CDriver*, *SPIDeviceManager*, ...), and then the implementation for the AVR processor and the APM2 board was made based on the corresponding classes (*AVRUARTDriver*, *AVRI2CDriver*, *APM2SPIDeviceManager*,...). A specific implementation is carried out by using conditional preprocessor directives, depending on the current configuration. The build is done using makefiles, where the required configuration variables are defined. All code for performing specific duties or operating with devices is segmented within *AP_** libraries; linking the

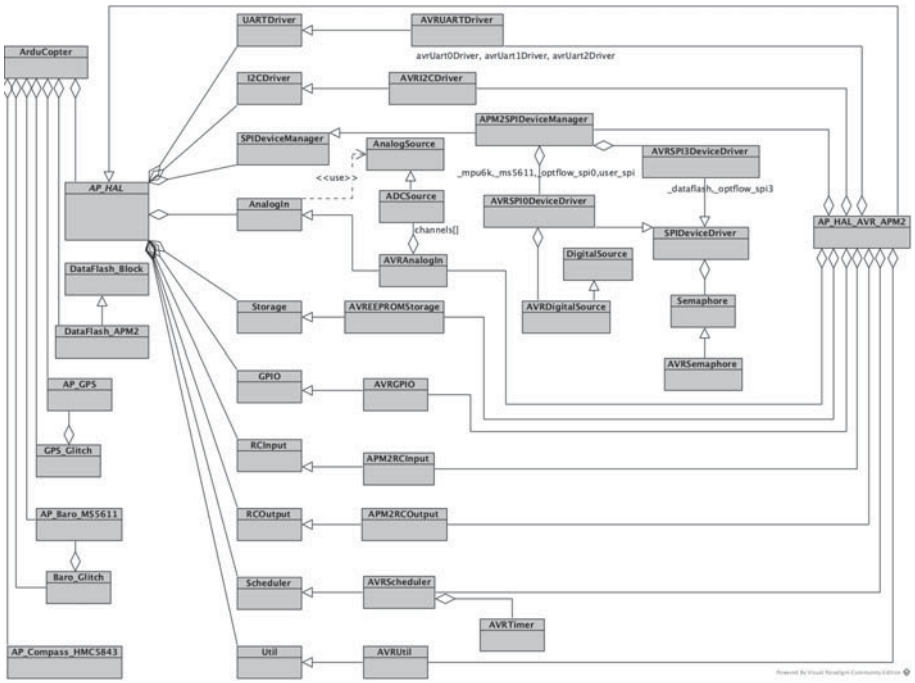

Fig. 3. Some notable classes and their relations in the implementation of Arducopter

necessary libraries is carried out during the build by analyzing the include sections of C++ files.

Later, after the development of other boards for processors supporting the run of real-time or Linux operating systems, the differences in the code and libraries became so significant that the project for AVR was issued as a separate branch, which is no longer supported. Nonetheless, this solution remains one of the most cost-efficient and energy-efficient entry-level reprogrammable quadcopters.

To build and upload the finished firmware in the form of a .hex file (about 240kb in size), the corresponding goals were written in the makefile. At the same time, an interesting feature of the project is that the developers have implemented tests for most of the libraries, which can be built and run directly from the corresponding directory beside the library, while only the necessary test code will be loaded into the APM board without starting the entire work cycle. It allows us to learn and debug the code to work with specific devices, such as motors.

V. Co-modeling the APM hardware and software

In this section, we discuss methods to model the hardware and software architectures. The results were obtained by analyzing the current open-source source code in a development environment and studying corresponding datasheets.

A. Modeling the APM hardware

Using the expressive capabilities of AADL, we implement the description of the main components of the system according to Section II-A. We model two processors with all their pins used: ATM2560 as the main processor and ATM32U2 as PWM and UART/USB converter. We describe the SPI0 and SPI3 buses (exactly as AADL logic buses), which are used to transfer data. By analyzing the source code of the libraries, we describe the devices on the board. Next, we create a system object which:

- composes all previously described devices;
- defines connections between processors, data buses and devices.

An example of such a description is given below (in Fig. 4, we then depict its graphical representation):

```
system implementation Ardupilot.impl
subcomponents
ATM2560: processor Processors::ATMEGA::
    ATMEGA2560.impl;
ATM32U2: processor Processors::ATMEGA::
    ATMEGA32U2.impl;
MagnetometerMC5843: device
    MagnetometerMC5843;
BarometerMS5611: device BarometerMS5611;
AccelerometerMPU6000: device
    AccelerometerMPU6000;
...
connections
--magnetometer
M1: port ATM2560.PD1 ->
    MagnetometerMC5843.SDA;
M2: port ATM2560.PD0 ->
    MagnetometerMC5843.SCL;
--barometer
B1: port BarometerMS5611.MOSI -> SPI0.
    MISO;
```

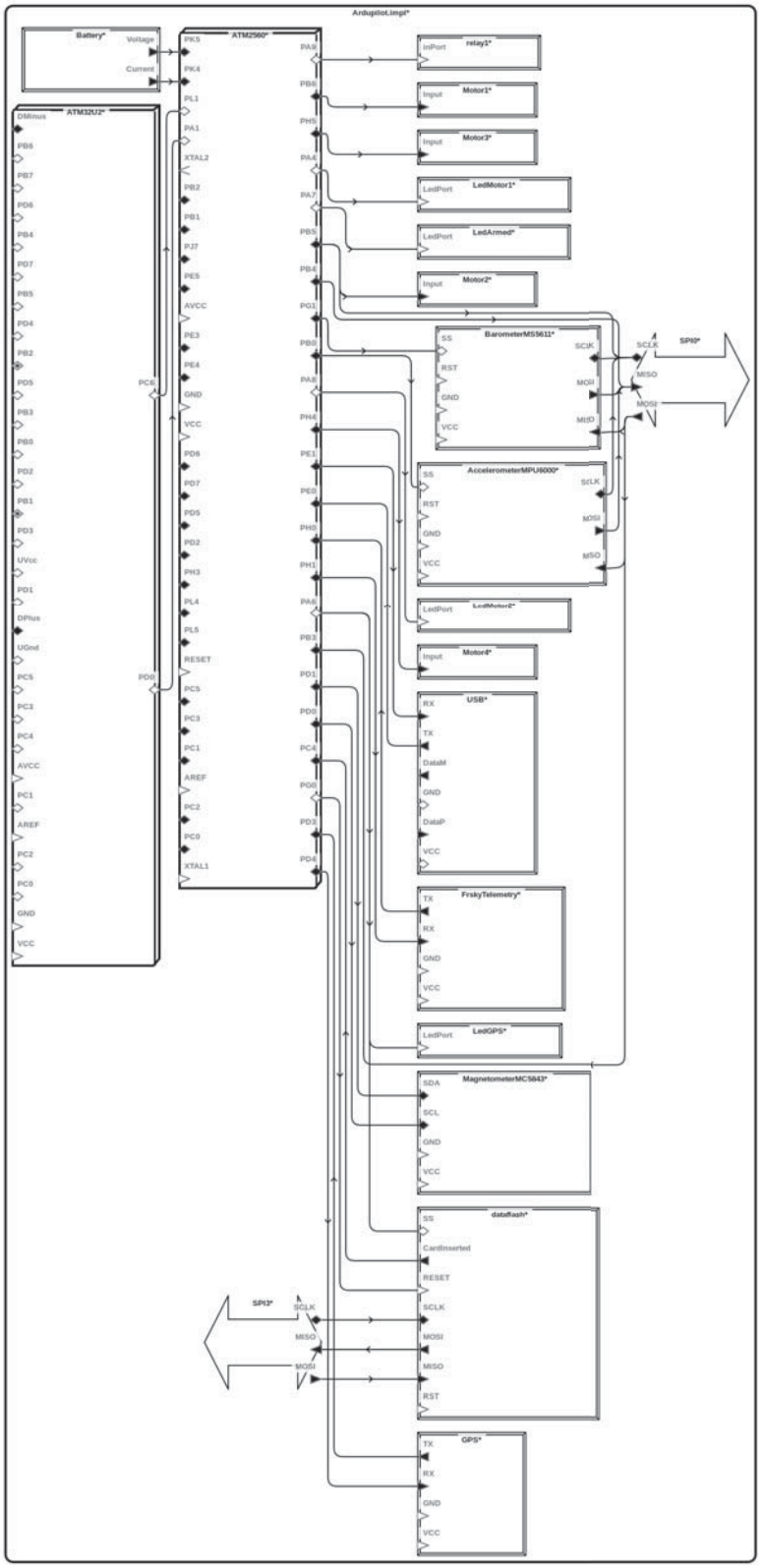

Fig. 4. A simplified APM hardware model

```
B2: port SPI0.MOSI -> BarometerMS5611.
    MISO;
B3: port BarometerMS5611.SCLK -> SPI0.
    SCLK;
B4: port ATM2560.PG1 -> BarometerMS5611.
    SS;
...
end Ardupilot.impl;
```

B. Scheduling in Arducopter

Due to processor limitations, task scheduling in ArduPilot/ArduCopter is specific. The developers apparently had to make a lot of efforts to make it work. Using the analysis of the source code, we can show its logic in Fig. 6.

Such scheduling refers to non-preemptive periodic tasks. Each task is characterized by its period and the maximum estimated operating time (worst execution time). This time can usually be estimated by measuring the exchange time with the corresponding devices. When choosing a task, the scheduler should take into account its interval and select those tasks that can still be completed by their maximum duration in a given time window. The remaining time is also taken into account.

By looking at the actual source code (see https://github.com/ArduPilot/ardupilot/blob/ArduCopter-3.2.1/ArduCopter/ArduCopter.pde#L777), we revealed that APM flies with 32 possible running task abstractions. In Table I, we present their parameters passed when creating the *AP_Scheduler::Task scheduler_tasks[]* object. We also profoundly analyzed the source code of these tasks and provided the table with descriptions of their work based on developers' comments.

The question of the correctness of this scheduling remains interesting. Since the algorithm is essentially a finite automaton, the variables are only integer, and there is no need to simulate switching between tasks, so it is possible to a) simulate this scheduling process based on Kripke transition systems and b) check its correctness using software model checking methods [25]. Correctness means the absence of situations of overrunning a task and slipping a whole run of a task (depicted in Fig. 6). In our repository [26], we include a Promela model for such a scheduler. We implemented the algorithm from the original C-code (presented in Fig. 6) in such a modeling language with formalized semantics [27]. Then we set requirements for control variables in LTL formulas and start to play with model parameters. For example, we can change maximum times to some unexpected values and check statically the schedulability in these cases. Such modeling and verification are the subjects of a further article.

C. Towards modeling ArduCopter software

To model a software architecture for reliable systems using the AADL approach, it is necessary to describe the processes and their interactions.

In AADL, a process is considered to consist of threads, so we need to define the required number of processes and the same number of threads within them. Using the predefined periodicity properties, we can specify the timing of the work of threads according to the data previously obtained in Table I. An example of such a description for the *crash_check* process is given below (see our GitHub repository for the full system description [26]):

```
thread thr_crash_check
properties
 Dispatch_Protocol => Periodic;
 Compute_Execution_Time => 0 ms .. 20 ms;
 Period => 100 Ms;
end thr_crash_check;
thread implementation thr_crash_check.i
end thr_crash_check.i;

process process_crash_check
end process_crash_check;

process implementation
    process_crash_check.i
subcomponents
 thr: thread thr_crash_check.i;
end process_crash_check.i;
```

Such data can be used to statically check the correctness of their schedulability for a given set of processes in the system according to one of the scheduling algorithms [28]).

As for the relationship of processes, not everything is so obvious here. Let us consider an example of describing a simple model of quadcopter processes from the POK repository [29] (Fig. 5).

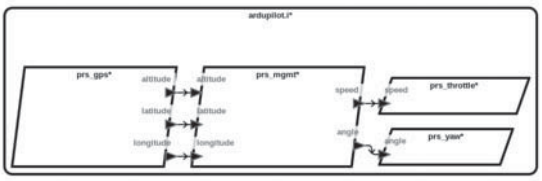

Fig. 5. An example of ArduPilot software model derived from AADL code [29]

In this example, connections are defined as the transmission of values of logical signals about speed, angle, coordinates. For reliable real-world systems, as a rule, such connections are formalized by using a message-oriented API standardized for the desired class of systems (for example, ARINC 653 [30] for avionics).

In our case, for a DIY device, the code is written subsequently. It uses a bunch of static (global) variables to transfer data between various components of the system. This approach is challenging to analyze and leads to errors, because a variable can potentially be changed to an incorrect value in many places. Building all dependencies by variables for about 30 processes in the system by hand is laborious. So, currently, we are implementing software for automatic source-code analysis to scan connections between such tasks through shared variables.

TABLE I. TASKS TO RUN IN APM

	Task id	Interval ticks	Max time (ms)	Description
1	rc_loop	1	100	Reads user input from transmitter/receiver
2	throttle_loop	2	450	Throttle loop: 1) gets altitude and climb rate from inertial lib; 2) checks if we have landed; 3) checks auto_armed status
3	update_GPS	2	900	GPS data obtaining, logging and glitch protection run after every GPS message, check for loss of GPS
4	update_batt_compass	10	720	Reads battery and compass, records throttle output
5	read_aux_switches	10	50	Checks aux switch positions and invokes configured actions
6	arm_motors_check	10	10	Checks for pilot input to arm or disarm the copter
7	auto_trim	10	140	Slightly adjusts the ahrs.roll˙trim and ahrs.pitch˙trim towards the current stick positions. Meant to be called continuously while the pilot attempts to keep the copter level
8	update_altitude	10	1000	Reads barometer and sonar altitude
9	run_nav_updates	4	800	Top level call for the autopilot. Ensures calculations such as ”distance to waypoint” are calculated before autopilot makes decisions. 1) fetches position from inertial navigation; 2) calculates distance and bearing for reporting and autopilot decisions; 3) runs autopilot to make high level decisions about control modes
10	update_thr_cruise	1	50	Updates throttle cruise if necessary: 1) gets throttle output; 2) calculates average throttle if we are in a level hover 3) updates position controller
11	three_hz_loop	33	90	3.3hz loop: 1) checks if we have lost contact with the ground station; 2) checks if we have breached a fence; 3) updates events
12	compass_accumulate	2	420	If the compass is enabled then tries to accumulate a reading
13	barometer_accumulate	2	250	Tries to accumulate a barometer reading
14	update_notify	2	100	Updates the status of notify (LEDs)
15	one_hz_loop	100	420	Runs at 1Hz: 1) logs battery info to the dataflash; 2) performs pre-arm checks and display failures every 30 seconds; 3) auto disarm checks; 4) makes it possible to change AHRS orientation at runtime during initial config; 5) checks the user has not updated the frame orientation; 6) updates assigned functions and enable auxiliar servos
16	ekf_dcm_check	10	20	Detects if EKF (the position and attitude estimation system) variances or DCM yaw errors that are out of tolerance and triggers failsafe
17	crash_check	10	20	Disarms motors if a crash has been detected. Crashes are detected by the vehicle being more than 20 degrees beyond its angle limits continuously for more than 1 second
18	gcs_check_input	2	550	Looks for incoming commands on the GCS links
19	gcs_send_heartbeat	100	150	GCS send message(MSG_HEARTBEAT)
20	gcs_send_deferred	2	720	GCS send message(MSG_RETRY_DEFERRED)
21	gcs_data_stream_send	2	950	Sends data streams in the given rate range on both links
22	update_mount	2	450	Updates camera mount position
23	ten_hz_logging_loop	10	300	Logging
24	fifty_hz_logging_loop	2	220	Logging
25	perf_update	1000	200	Logging performance
26	read_receiver_rssi	10	50	Reads the receiver RSSI as an 8 bit number for MAVLink RC_CHANNELS_SCALED message
27	telemetry_send	20	100	Sends FrSky telemetry if enabled
28	userhook_fast_loop	1	100	Runs 100Hz user hook if enabled
29	userhook_50_hz	2	100	Runs 50Hz user hook if enabled
30	userhook_medium_loop	10	100	Runs 10Hz user hook if enabled
31	userhook_slow_loop	30	100	Runs 3Hz user hook if enabled
32	userhook_super_slow_loop	100	100	Runs 1Hz user hook if enabled

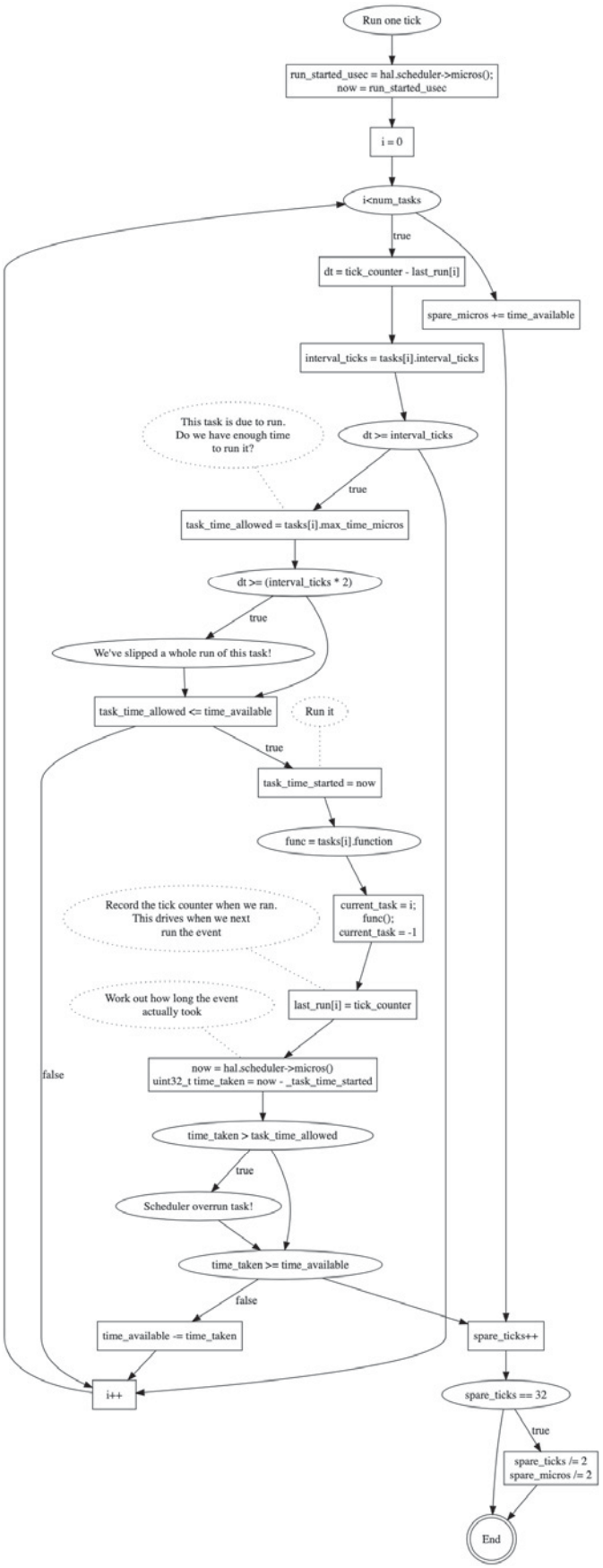

Fig. 6. Main scheduler loop for APM (built from its open-source code)

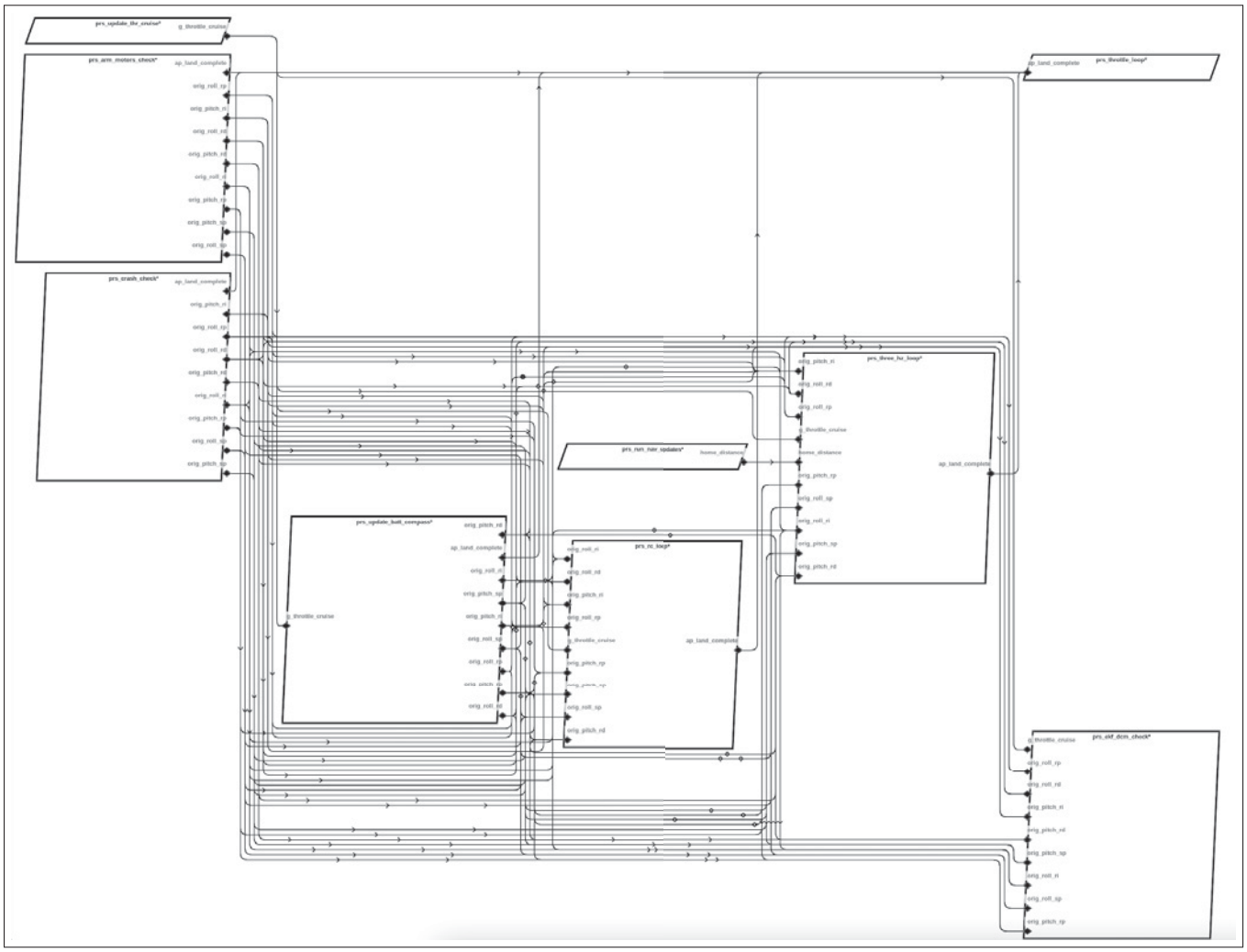

Fig. 7. Preliminary software model for ArduCopter tasks generated by our C/C++-code analyzer [26]

We programmable analyze forests of control-flow graphs [31], [32] with functions of interests, obtained from ASTs of ArduPilot .cpp/.pde files and monitor read and write (L-value/R-value) access to variables. Then we generate AADL code in the form of the example [29]. Current results are presented in Fig. 7. The analyzer only processes assignments to variables or field structures. One can see from the diagram that global information on Euler angles is distributed between six tasks. In this way, we can track such data paths without looking at complicated code in various source files.

VI. Distributed controllers with respect to safety

Considering the scheduling issues (see Fig. 6 and Table I), we see how difficult it is to select heuristics so that the processes in the system on such weak processors work correctly. Accordingly, if we need some complex calculations to assess our state, for example, various kinds of physical models to check the safety of operation [33], it is evident that it is easier to transfer data for processing to another device and not implement another process inside. In this case, we come to the concept of distributed controllers.

Distributed controllers act in real-world cyber-physical systems to process dedicated information and interchange it with other system parts. The examples of such systems can be cars that have more than one electronic control unit related to different functions (engine, transmission, safety systems) and airplanes where different controllers are located in several parts of chassis, and they are joined together to process information faster and to make reservations according to reliability requirements.

In this section, we discuss passing state to another controller in a unified way. Since APM is an Arduino-compatible device, as we can see in Fig. 4, there are several SPI buses for peripherals and user actions. Analyzing the code, we found that it is possible to use the SPI bus [34], [35], which is potentially utilized to connect an external optflow device. Examining the source code of the interaction, we came to a way to connect a different controller via SPI, as shown in Fig. 8.

The code for operating with an external controller consists of initializing and registering the exchange process (in this case, writing some code in *sensors.pde*). The exchange includes taking time on the SPI bus, turning on the transfer by giving a signal to the *SS* pin, sending and simultaneously receiving data using the existing library to work with SPI. Our tests of the implemented exchange with an *STM32f106* controller show that such an exchange is possible. However, it requires voltage matching using a 5V <> 3.3V logic converter, as well as an additional software check for data integrity.

Further, the successful operation of the SPI bus leads us to the idea that it is possible to use SPI devices to work with field buses, allowing for increased distances and an increased number of nodes on the network. We assume that the CAN bus [36] can also be used here. The architectural diagram obtained

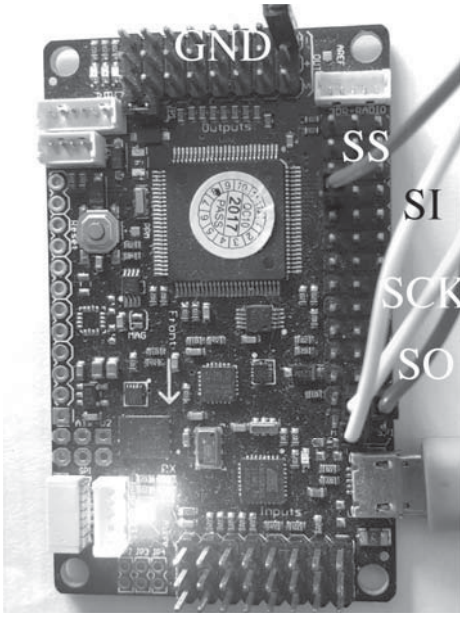

Fig. 8. APM connection to a different controller using SPI

by writing an AADL code for such a case is shown in Fig. 9. So far, we have connected the MCP2515 CAN transceiver [37] to the APM via SPI and ported the open-source code [38] for such a device, initially developed by Seeed company. Our results are presented in a dedicated repository on GitHub [39].

VII. Conclusion

As a result of a large number of modeling in this work, we can conclude that this kind of design analysis dramatically helps in understanding the essence of the operation of a complex computer-appliance cyber-physical system and teaches how to design such systems correctly. We applied the co-modeling by obtaining architectural models of a real APM board. These models can be further used to assess the properties of the safe behavior of such a system. We also created a solution for the distributed connection of controllers to offload the scheduler. All current results are only based on free software and are freely available.

We can note that to represent a real-world CPS, we have used both UML, automata, AADL models, create code in special modeling languages. So, the usage of only one language is not possible for expressive modeling. Also, currently, there are no such tools exist to make it possible to easily move between various parts of the CPS model (or their interconnected relations). One solution can be the usage of AADL annexes [40] to create a composite model and the various visualizers as well as validators for it, but it is not yet done by the communities.

As further work, we are thinking of analyzing the full source code and building connections between processes in the system through variables, assessing their number and fault tolerance of the solution. We will also continue to work with the CAN bus.

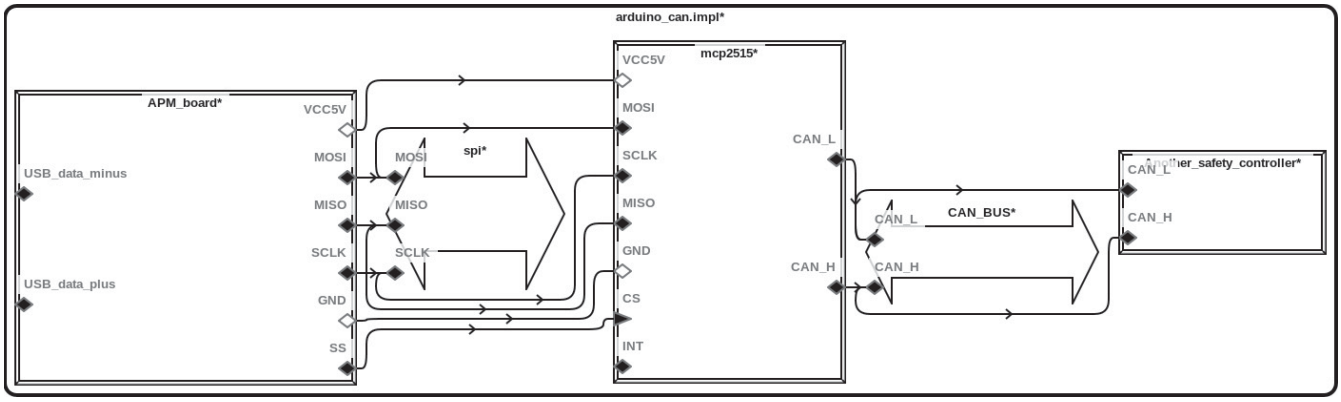

Fig. 9. Architectural diagram of a CAN bus usage

REFERENCES

[1] K. R. Krishna, *Agricultural drones: a peaceful pursuit*. CRC Press, 2018.

[2] C.-H. Lai and C.-M. Chu, "Development and evaluation of STEM based instructional design: An example of quadcopter course," in *International Symposium on Emerging Technologies for Education*. Springer, 2016, pp. 176–191.

[3] E. Gamma, R. Helm, R. Johnson, J. Vlissides, and D. Patterns, "Elements of reusable object-oriented software," *Design Patterns. massachusetts: Addison-Wesley Publishing Company*, 1995.

[4] S. M. Staroletov, M. S. Amosov, and K. M. Shulga, "Designing robust quadcopter software based on a real-time partitioned operating system and formal verification techniques," *Proceedings of the Institute for System Programming of the RAS*, vol. 31, no. 4, pp. 39–60, 2019.

[5] T. Myers, G. Dromey, and P. Fritzson, "Comodeling: From requirements to an integrated software/hardware model," *Computer*, vol. 44, no. 4, pp. 62–70, 2010.

[6] S. Shiraishi, "An AADL-based approach to variability modeling of automotive control systems," in *International Conference on Model Driven Engineering Languages and Systems*. Springer, 2010, pp. 346–360.

[7] A. Khoroshilov, D. Albitskiy, I. Koverninskiy, M. Olshanskiy, A. Petrenko, and A. Ugnenko, "AADL-based toolset for IMA system design and integration," *SAE International Journal of Aerospace*, vol. 5, pp. 294–299, 2012.

[8] P. H. Feiler, B. A. Lewis, and S. Vestal, "The SAE architecture analysis & design language (AADL) a standard for engineering performance critical systems," in *2006 IEEE Conference on Computer Aided Control System Design, 2006 IEEE International Conference on Control Applications, 2006 IEEE International Symposium on Intelligent Control*. IEEE, 2006, pp. 1206–1211.

[9] S. Staroletov, N. Shilov, I. Konyukhov, V. Zyubin, T. Liakh, A. Rozov, I. Shilov, T. Baar, and H. Schulte, "Model-driven methods to design of reliable multiagent cyber-physical systems," in *CEUR Workshop Proceedings*, vol. 2478, 2019, pp. 74–91.

[10] *OSDAY (in Russian)*, 2019. [Online]. Available: https://osday.ru

[11] S. Staroletov, "Work-in-progress abstract: Revealing and analyzing architectural models in open-source ardupilot," in *2021 IEEE 27th International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, 2021, pp. 207–209.

[12] *ArduPilot Copter*, 2020. [Online]. Available: https://ardupilot.org/copter/

[13] *History of ArduPilot*, 2019. [Online]. Available: https://ardupilot.org/planner2/docs/common-history-of-ardupilot.html

[14] *Atmel ATmega640 / V-1280 / V-1281 / V-2560 / V-2561 Datasheet*, 2014. [Online]. Available: https://microchip.com/downloads/en/devicedoc/atmel-2549-8-bit-avr-microcontroller-atmega640-1280-1281-2560-2561_datasheet.pdf

[15] *MS5611-01BA03 Barometric Pressure Sensor, with stainless steel cap*. [Online]. Available: https://www.te.com/commerce/DocumentDelivery/DDEController?Action=showdoc&DocId=Data+Sheet%7FMS5611-01BA03%7FB3%7Fpdf%7FEnglish%7FENG_DS_MS5611-01BA03_B3.pdf

[16] *Honeywell 3-Axis Digital Compass IC HMC5843*. [Online]. Available: https://www.sparkfun.com/datasheets/Sensors/Magneto/HMC5843.pdf

[17] J. Delange, "AADL in practice: Become an expert in software architecture modeling and analysis," *Reblochon Development Company*, 2017.

[18] G. Lasnier, B. Zalila, L. Pautet, and J. Hugues, "Ocarina: An environment for AADL models analysis and automatic code generation for high integrity applications," in *International Conference on Reliable Software Technologies*. Springer, 2009, pp. 237–250.

[19] J. Hugues, B. Zalila, L. Pautet, and F. Kordon, "From the prototype to the final embedded system using the Ocarina AADL tool suite," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 7, no. 4, pp. 1–25, 2008.

[20] J. Hugues, "AADLib, a library of reusable AADL models," 2013.

[21] *OpenAADL/AADLib boards-ardupilot*, 2018. [Online]. Available: https://github.com/OpenAADL/AADLib/blob/master/src/aadl/boards/boards-ardupilot.aadl

[22] P. Feiler, "The open source AADL tool environment (OSATE)," Carnegie Mellon University Software Engineering Institute Pittsburgh, Tech. Rep., 2019.

[23] *ArduPilot Project*, 2015. [Online]. Available: https://github.com/ArduPilot/ardupilot/tree/ArduCopter-3.2.1

[24] R. Miles and K. Hamilton, *Learning UML 2.0: a pragmatic introduction to UML*. "O'Reilly Media, Inc.", 2006.

[25] E. M. Clarke Jr, O. Grumberg, D. Kroening, D. Peled, and H. Veith, *Model checking*. MIT press, 2018.

[26] *Co-modeling-Ardupilot*, 2021. [Online]. Available: https://github.com/SergeyStaroletov/Co-modeling-Ardupilot

[27] V. Natarajan and G. J. Holzmann, "Outline for an operational semantics of Promela." *The Spin Verification System*, vol. 32, pp. 133–152, 1996.

[28] G. C. Buttazzo, *Hard real-time computing systems: predictable scheduling algorithms and applications*. Springer Science & Business Media, 2011, vol. 24.

[29] *POK Ardupilot example*. [Online]. Available: https://github.com/pok-kernel/pok/tree/main/examples/case-study-ardupilot

[30] S. Santos, J. Rufino, T. Schoofs, C. Tatibana, and J. Windsor, "A portable ARINC 653 standard interface," in *2008 IEEE/AIAA 27th Digital Avionics Systems Conference*. IEEE, 2008, pp. 1–E.

[31] L. Serrano, "Automatic inference of system software transformation rules from examples," Ph.D. dissertation, Sorbonne Université, 2020.

[32] J. Brunel, D. Doligez, R. R. Hansen, J. L. Lawall, and G. Muller, "A foundation for flow-based program matching: using temporal logic and model checking," in *Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 2009, pp. 114–126.

[33] S. Staroletov, "Automatic proving of stability of the cyber-physical systems in the sense of Lyapunov with KeYmaera," in *2021 28th Conference of Open Innovations Association (FRUCT)*. IEEE, 2021, pp. 431–438.

[34] S. Srot, "SPI master core specification," *OpenCores*, 2004.

[35] F. Leens, "An introduction to I2C and SPI protocols," *IEEE Instrumentation & Measurement Magazine*, vol. 12, no. 1, pp. 8–13, 2009.

[36] R. Bosch, “Bosch CAN Specification Version 2.0,” 1991. [Online]. Available: https://www.nxp.com/docs/en/reference-manual/BCANPSV2.pdf
[37] Microchip, “Stand-alone CAN controller with SPI interface,” 2005.
[38] *Arduino MCP2515 CAN interface library*. [Online]. Available: https://github.com/autowp/arduino-mcp2515
[39] *Ardupilot-SPI-CAN*, 2021. [Online]. Available: https://github.com/SergeyStaroletov/Ardupilot-SPI-CAN
[40] J. Delange and P. Feiler, “Architecture fault modeling with the AADL error-model annex,” in *2014 40th EUROMICRO Conference on Software Engineering and Advanced Applications*. IEEE, 2014, pp. 361–368.