A Customizable Software Tool for Hardware in the Loop Tests

Markus Eider
Deggendorf Institute of Technology
Technology Campus Freyung
D-94078 Freyung, Germany
Email: markus.eider@th-deg.de

Stefan Kunze
Deggendorf Institute of Technology
Technology Campus Freyung
D-94078 Freyung, Germany
Email: stefan.kunze@th-deg.de

Wolfgang Dorner
Deggendorf Institute of Technology
Technology Campus Freyung
D-94078 Freyung, Germany
Email: wolfgang.dorner@th-deg.de

***Abstract*—In this paper an automated, software-based and easy to customize test tool for Hardware in the Loop (HIL) measurements is proposed. This system is originally designed for the test of a thermal power station control system. Due to its modular approach it may also be customized for a wide range of applications. The developed tool is independent of the Operating System (OS) or the used hardware platform. Especially when using embedded systems as host platform limited system resources are available. One demand is, therefore, the development of a lightweight tool. By implementing the tool in Python, which by itself provides various hardware abstraction modules, a suitable and efficient programming language is selected. To enable an easy adoption of this tool for further tests or even future projects, a modular software architecture is proposed. Therefore, the test functionality is divided into its basic core functionalities, which are then implemented in dedicated software components. Decoupled from each other and linked via a central communication system, continued development and improvement of these components is possible.**

For each test, that shall be performed, a dedicated test script that describes the used interfaces, instruments, as well as the test definitions has to be interpreted and executed at runtime. This approach allows easy modifications of the test script without the need for restarting the entire test tool.

By implementing an exemplary test, the functionality of the proposed test tool is illustrated. For this purpose a test measurement, which closely resembles the intended application field in thermal power station control systems is presented.

I. Introduction

At the current time, decentralized power generation with thermal power stations is already a major element of renewable energy generation. In an ongoing project, the idea of combining multiple thermal power stations and charging stations for electric vehicles is researched. With this combination, a smart grid, which can contribute to balancing the overall power grid, may be established.

In this context, an energy manager module is used for controlling the operational states of decentralized thermal power stations. Currently, the module supports controlling the heat generation of the power station and surrounding installations. For this purpose, a number of temperature sensors have to be interfaced. In a further development stage, output current and power control shall also be a possibility. To interact with the system components, the energy manager uses different interfaces. Temperature sensors, for example are accessed via the 1-Wire bus and the thermal power station is controlled with analog sensor-actuator and transducer interfaces.

To ensure correct functionality in future applications, the energy manager's physical interfaces need to be tested and verified. At the current development stage, the required tests are limited to the 1-Wire bus. Hence, the proposed test tool is primarily developed for this purpose. It shall, nevertheless, also be easily adoptable for future needs. Therefore, the following goals are defined for the development of the test tool. HIL and regression tests of the System under Test (SUT) are the main aspect of the program. However, it needs to be designed in a modular architecture, so that high flexibility for the future is assured. The implementation shall also be independent of the used hardware platform, as well as the operating system. This must be considered, when designing and implementing the system components. Due to the demands of the mentioned project, the tool will be used on multiple systems in parallel. Therefore, it should be able to run on low-budget hardware. While the same energy manager is used throughout the project, the surrounding systems may vary. Due to this, modifications of the test routines must be possible even when setting up the system in the field.

In order to achieve these goals, the following requirements have been defined. To achieve a high modularity, an object-oriented approach shall be utilized. The test tool must be able to communicate via common computer hardware interfaces (e.g. USB, Ethernet). This is especially important for controlling lab equipment in order to transmit stimuli to the SUT. It shall be capable of providing an analysis of the tested hardware's reactions to the transmitted stimuli. The importance of this is shown in [1]. Besides controlling hardware, the inclusion of other software components shall be possible. This provides the capability to use other software-based test equipment.

As A. Jha proposes, faster development cycles can be achieved by considering reusability of individual components of the overall test tool [1]. Thus, abstraction layers and test implementations of the test tool should be designed in a way, so that they may be reused by the test developers for future applications. The exact procedures for a particular measurement shall be defined in the according test implementation. Hence, the test implementations are to be programmed

ISBN 978-80-261-0602-9,

in a separate file, that can be loaded and displayed by a simple Graphical User Interface (GUI). Single test cases and the number of their repetitions shall be specified by the testing personnel via the GUI. Measured values, which are acquired during the runtime of the test, must be logged to a file. Runtime errors, such as connection faults, shall be logged as well.

Besides these requirements, the test tool shall also meet the following non-functional requirements. The test cases and the tool itself should be easy to configure. This shall hold true even if on-site modifications are necessary. In order to test several systems at the same time, multiple tool instances may be running on different test systems in parallel. These instances may require slightly different configurations, possibly due to varying models of the used test equipment or device under test. In order to still achieve a low-priced implementation in this case, commercial tools, such as LabVIEW or MATLAB, are not suitable for this application.

II. Related Work

Programming languages like C, C++ and Python are commonly used for the implementation of test software [2]. Besides the resemblance to C and C++, Python offers a relatively simple approach to scripting, and is also platform independent. This programming language is therefore suitable for implementing the proposed test tool. Python also allows to interface a large variety of tools and different programming languages [2]. Although the need for multiple languages is not given in this paper, being able to interface almost any electronic test equipment (ETE) contributes to achieve high flexibility for the developed test tool.

For increasing the modularity, software patterns are a well-known approach in software development. Therefore, two patterns, which are adopted in the proposed test tool, are discussed in this section. The first of these patterns is the Singleton, which is a creational pattern. It only allows one object of a class to be instantiated at runtime [3]. The Singleton could be wrapped around another class, whose instances get passed through the Singleton. The classical approach, however, is to nest the Singleton essentials inside a class definition, thus creating a Singleton class which includes other functionalities as well. Since the classical approach is often implemented with a private constructor, a different approach has to be taken, when using Python [4]. It uses integers to count the numbers of instances at the beginning of the object initialization. In case of instance exceedance, the initialization is interrupted by an exception and the object is removed by the garbage collector.

The second pattern, which is used to control the data flow and access between objects especially during development, is called Mediator pattern. It is a common, behavioral tool to handle the communication between different program components, as it decouples their method calls from each other. Instead of directly accessing object methods, they have to call the dedicated method duplicates of the Mediator, which act as abstraction layer to the original methods [3]. Since a great number of methods have to be duplicated, the test tool proposed in this paper uses a simplification of the scheme. A single method named *messageReceivers* is placed in the Mediator. This way, the interfaces between the program components may be modified in the future without having to replicate all receiver methods in the Mediator. However, because this method is more generic, an additional parameter has to be passed. This parameter instructs the Mediator, which receiver has to be called.

For the implementation of a software-based test tool, some key modules are needed. When possible, existing Python modules shall be used. At first, the logging of message strings for later analysis can be done with the module *logging*. It allows saving measurement results in different formats such as HTML or PDF [2]. This is a convenient solution, since the output data doesn't need to be converted to the desired format. In contrast to this, approaches that use XML as output, need an additional conversion to achieve this [1]. Other outputs such as streams or sockets would be possible as well [5], but a simple Comma-Separated Values (CSV) file is sufficient for the presented purpose. A major key advantage of a central logging unit is to prevent race conditions while writing to memory. Instead of every component writing to a file on their own, they can send their messages to the logging system, which stores the strings safely. The module also supports a variety of severity levels. These can be used for message categorization, since levels like DEBUG, INFO or WARNING are provided by the module [5]. Test results or errors could be generated as strings and sent to the logging component including the according keywords. This helps to identify errors quickly.

The continuous operation of program components in parallel can be achieved by using threads. In this way, blocking of components is prevented. Especially blocking of the GUI may be noticed almost immediately. The overall program performance is increased by using threads [6]. Another advantage is a more structured source code, since certain routines don't have to be called cyclically in between the program flow [6]. Thus, components like the logging module can be defined within threads, decoupling their functionalities into separate program flows. When using threads, developers need to be cautious about runtime problems such as deadlocks or race conditions [6]. To overcome such problems, a queue structure is used to handle information between threads. The *threading.Queue* module provides a thread-safe First In First Out (FIFO) data structure [7]. Race conditions and deadlocks are prevented by a locking mechanism, embedded in the module [7]. Especially when using the logging module, this can be a key attribute for the input buffer, because the messages are stored in a correct chronological order. The same principle can be applied for components dedicated to run test cases later on. The test cases can be sent as references and the chronological order won't be disrupted by the queue.

Another important feature of the test system is the capability to remote control electronic equipment. Often, this can be done with Standard Commands for Programmable Instruments (SCPI), which is a uniform instruction set for remote controllable instruments [8]. These commands consist of American Standard

Code for Information Interchange (ASCII) characters and can be sent over interfaces such as Universal Asynchronous Receiver Transmitter (UART) or Ethernet. The control software only needs to build the command strings and send them via the interface. A typical sequence for setting up remote controllable instruments is given in the following [9][10].

1) Set up the connection
2) Configure the instrument for desired measuring range
3) Instruct the measurement
4) Read out the acquired values
5) Close the connection

Although this approach makes it possible to use the same code for multiple instruments of the same type (e.g. oscilloscopes), there may be slight differences between models of various manufacturers [10]. This means, that SCPI command routines may not be fully reusable for other instruments. Therefore, abstraction layers for each used measuring instrument are recommended. The used instruments can be selected before the execution of a test implementation.

III. Architecture

Clearly differentiated modules help to structure the code and allow modifications or adjustments as well as adding of new features or extensions. Therefore, the system is structured in pattern-oriented modules, thus improving the expandability and reusability [3]. With this approach, the tool can be adopted for future projects, requiring minimal effort.

The architecture of the proposed test tool and the event flow between the modules are illustrated in Fig. 1. Each service, the test tool provides, is reflected in a dedicated program component.

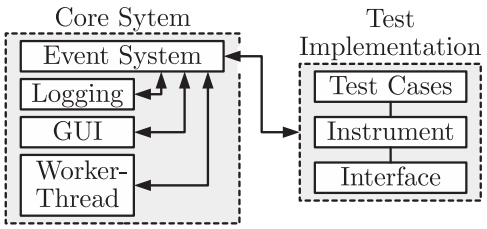

Fig. 1. Architecture of the test tool and the event flow between components

A. Core System

As functional base, the core system is a fixed software component that provides services for every test implementation. This includes the execution of test cases and the communication with the test implementation via events. Also the displaying and logging of data is handled by the core system. All core system components are initialized as Singletons since the individual services are only needed once at any given time. Due to the decoupling from the actual test, the core module doesn't need to be modified in order to implement new applications.

1) Events: For communication between the program components, a multi-purpose container class is defined. For abstraction of the several possible implementations, the parent class *Event*, that has a member holding the origin reference, is introduced. This is illustrated in Fig. 2. The member allows to reference the event's point of origin and can be helpful in the logging of measurements or in case of errors. Methods to set or call member values are also implemented. These are not included in Fig. 2.

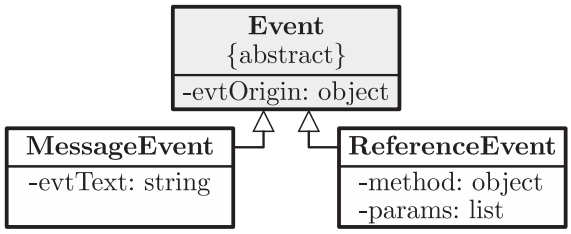

Fig. 2. Simplified generalization of the events

Events are divided into subclasses in order to define different purposes. For this application, two subclasses are proposed. The first is the *MessageEvent* class, which serves as message container for object communication. A class member of the type *string* holds the message, e.g. bus commands or text for logging. Through this design, there is the option of using the *string*-member as program command. By declaring variables with fixed keywords in a configuration file, commands are defined. These keywords should be reserved for this use only. On instantiation of a *MessageEvent*, the needed command string is used as parameter for the initializer method, to set the *string*. Subsequently, event recipients of other program parts can look up these keywords and react accordingly to predefined rules. The second subclass of the abstract event definition is called *ReferenceEvent*. It is used for transporting object or method references between the individual program components. Its additional member attribute is of the type *object* and can be used to reference these objects or methods. A second member of type *list* holds optional parameters for the method. These must be given in the right order, with regard to the parameter list of the method's declaration.

2) Event Server: As central communication system, for interaction between the individual components, an event server is introduced. For the purpose of modularity, the Mediator pattern is chosen. The server is a passive component, that only operates when called. By using a generic *eventHandler* method within every event receiving object, any class can be changed without altering the rest of the software, since the access between the objects is decoupled by a Mediator. The access is implemented with subclass instances of *Event*, which are routed to their destination by the event server. Therefore, receivers have to subscribe at the event server in order to receive events. The subscription takes place with an additional role, the class has in relation to the program context. This can be compared to email distribution lists. Since all events are sent as parameters in combination with the receiver information, the event server can find the receivers due to their prior subscriptions. Depending on the roles assigned to the receivers, the implemented

event system may utilize uni-, multi- or broadcasts. In order to transmit an event, the sending object first creates an instance of one of the event subclasses. Then it adds its own reference to the event and initializes the event-members with the needed data via the dedicated *set* methods. To send the event to the receiver, the object calls the *messageReceivers* method with the event and receiver role as parameters. After identification of the receivers, the event server passes the event to the *eventHandler* of each receiving object. In a final step, the receivers can extract the needed information.

3) Message Logging: A central logging system is implemented in the test tool for recording measured data as well as software faults, such as connection errors of the interfaces. For continuous operation in parallel to other components, it is embedded in its own thread. To prevent race conditions with other threads, a queue of the Python module *threading.Queue* receives and buffers the incoming *MessageEvents*. The logging process itself is realized with the Python module *logging*, which provides a time stamping function when saving each message string to a file. Multiple accesses on the log file from different components of the software would cause conflicts or data loss. The proposed implementation is a simple solution to prevent such problems.

4) Worker-Thread: To avoid blocking of the GUI and the rest of the program, test implementations are operated in a separate thread. The number of repetitions for each measurement has to be set via the event system, so that the same number of *ReferenceEvents* is sent. Each test case cycle can be paused or stopped from the GUI via *MessageEvents* using program commands. The event handling within the worker thread is performed in the following order. At first, incoming events are sorted by their class. If *MessageEvent* objects contain command values, the contained commands are used for the next state of the thread, e.g. start or stop. In case of *ReferenceEvent* instances, these are stored in the threadsafe queue. Provided that the thread is in its running state and the queue is not empty, *ReferenceEvent* objects are taken out of the queue. The contained method references in the events are then executed. If there are any additional parameters in the *params* list, the method is executed accordingly.

5) GUI: The user interface is implemented with wxPython, which is a C++ wrapper for the platform independent and threadsafe package wxWidgets [11]. It provides a basic GUI to display both the source code of the test implementations, as well as the progress of their execution. To run test implementations, they can be chosen from file and sent to the worker thread in the form of *ReferenceEvents*.

B. Test Implementation

While the core system implements the program flow of the test tool, it is independent of the actual measurements, and must therefore, not be modified when changing applications. The required routines for the measurements are implemented in the second major component of the proposed architecture. It provides the abstraction layers to the ETE and defines the measurement routine. It has to be customized for each individual application. Nevertheless the test implementation is in itself modular, as illustrated in Fig. 3. Existing components (such as instruments or SCPI routines) may therefore, be reused for new applications. The communication between the test implementation and the core system is handled by the event server.

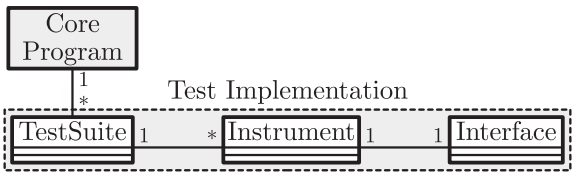

Fig. 3. Dependencies of the Test Implementation

1) Interfaces: On top of OS drivers for common computer interfaces, such as UART, USB etc., usually some adjustments have to be made in order to communicate with the ETE. For this purpose, interface classes need to be implemented. These class definitions handle the connection states and data transport. Any faults which could occur in this process are sent as *MessageEvent* to the logging module of the core system.

2) Instruments: The name of this component is derived from the remote controlled ETE. It is a software-based representation of the actual instrument and provides the functionality of the test equipment to the test suite. Each function of the instrument can be called with a dedicated method, that sends commands over the interface instance to communicate with the real instrument. A suitable protocol for this application is often SCPI, which creates the possibility to implement each method with consistent commands for different instruments. Since most instruments are available with various bus interfaces, instrument objects are instantiated as members within test cases and the preferred interface will then be referenced with the instrument. This enables the use of multiple meters of the same model while the choice of the used interface is up to the developer. All preferences concerning interface connection and instrument addressing (e.g. IP-address and port) have to be specified in a configuration file. The test suites can then read the equivalent information.

3) Test Suites: Each test case of a test specification may include one or more measurements. Thus, multiple instruments can be involved in a test case and need to be controlled by additional test case code. In order to avoid redundant implementation, the test cases are grouped within test suites. This allows to make test suites for every component of the SUT or to generate larger test routines. Test suites have to be defined as Python classes. These *TestSuite* classes contain the test cases as methods, whose names are not restricted by any naming convention. However, using the same style helps identifying them during development and test. Since all *TestSuite* implementations follow the Unittest principles and goals, they must implement *setUp* and *tearDown* methods [12]. Through *setUp*, the SUT is initialized to a defined state before test case execution. The opposite is done with *tearDown*, that powers down the system in a defined and safe way after

the test execution [1]. Developers have to define the *setUp* and *tearDown* methods for each *TestSuite* class. Further actions are only needed in test cases, if certain system components have to be configured before or after test. All *TestSuite* methods have to be referenced in a method, which gets called when the GUI is instantiating the test suite. The test case references are then sent to the GUI as *ReferenceEvents*.

IV. Exemplary Test Implementation

The proposed software tool is originally developed for the test of an energy manager module. For the exemplary test implementation, a measurement that resembles this application field is chosen. The energy manager module supports heat control of the thermal power stations. Therefore, temperature sensors of the type DS18B20 are connected via four 1-Wire bus networks. This bus system combines the power supply and data lines via one single wire in reference to a ground signal [13]. The hexadecimal temperature values supplied by the sensors are then converted to floating-point numbers and displayed in a web interface, which the energy manager provides over Ethernet.

For the exemplary test, the 1-Wire functionality of the energy manager shall be tested in terms of physical properties and logical behavior. In order to verify the functionality of the test tool, defined outputs from the sensors are required. This is realized by using an emulator for the 1-Wire temperature sensor, rather than actual sensors. The emulator is controlled by the test system proposed in this paper. The overall test setup is illustrated in Fig. 4. With this approach, the reactions of the energy manager can be analyzed in relation to defined inputs. The energy manager module only provides a web interface, in which the measured temperature of each sensor is displayed. This is why the reactions have to be checked by the testing personnel using an internet browser.

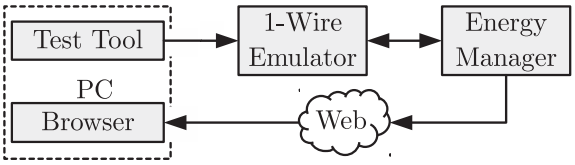

Fig. 4. Structure of the exemplary test

For the realization of the sensor emulator, a Field Programmable Gate Array (FPGA) development board with an Altera Cyclone IV EP4CE22F17C6N is used. It shall emulate the behavior of twelve DS18B20 sensors both physically and logically, as specified in the sensor's data sheet [14]. A detailed description of the implementation of the FPGA based emulator is given in a previous paper [15]. This emulator can be used in the terminology of a Smart Virtual Transducer (SVT), since there is no real sensor within the test circuitry [16]. A simple electrical interface is designed to switch the 1-Wire buses. Each emulated sensor can be switched arbitrary onto a bus via a Dual in-line Package (DIP) switch matrix. Furthermore, the physical data transmission is realized using an n-channel Metal-Oxide-Semiconductor Field-Effect Transistor (MOSFET) in open-drain configuration for each emulated sensor as illustrated in Fig. 5. The bus is driven to logic high by a pull-up resistor ($R = 4.7\,k\Omega$), which is usually provided by the 1-Wire master [13][14].

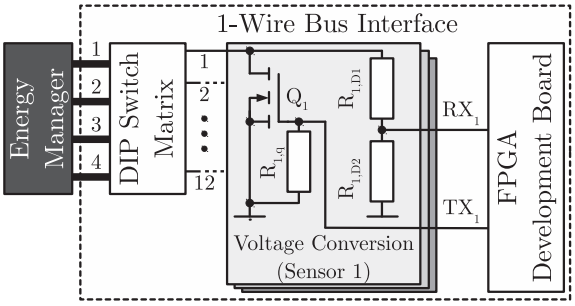

Fig. 5. Schematic of the 1-Wire bus control

Since 1-Wire slaves have to monitor the bus level, a voltage divider is used to reduce the nominal bus voltage of 5 V in order to match the FPGA's voltage of 3.3 V. Because the bus is operated on digital basis, a simple comparison to the FPGA's I/O pin voltage and ground is sufficient in order to implement the bus monitoring. The logical behavior of the sensor, as specified in the data sheet, is implemented with registers to hold temperature and threshold values as well as a Finite State Machine (FSM) to control the different functionalities, e.g. temperature conversion and register read-out. Due to the fact that the temperature register is set with a default value at power up [14], this value can be hardcoded. In contrast to a real DS18B20 sensor, where the ID is a fixed value given by the manufacturer, the emulator allows to set this number freely, increasing the test flexibility [15].

The FPGA is programmed in Very High Speed Integrated Circuit Hardware Description Language (VHDL). Each sensor analyzes the bus signal with a Timing Analysis module and communicates with the bus with a 1-Wire abstraction layer. This allows to keep track of the bus timings and control a switch case structure for the FSM states. Communication with the test computer utilizes a command set similar to SCPI via an UART connection. This way, artificial temperature values can be sent to the emulator in a consistent manner. Even if the firmware is changed, the test implementation on the test computer doesn't have to be modified.

A test suite to check the functionality with twelve sensor emulators is implemented. To communicate over UART, an *Interface* class, which controls the connection speed and data byte configuration, is defined. Additionally, an *Instrument* class provides methods such as *setTemperature* and *setID* to configure the emulators. The *TestSuite* implementation combines the interface with an instrument and provides the test cases. These test cases include the alteration of the resembled sensor IDs and temperature values in the emulator registers. *MessageEvents* are created at every communication with the FPGA. These are stored in a log file.

Overall, minimal development time is needed to incorporate the emulator into the test system. During test execution, the sensor IDs and artificial temperature values were changed in the test cases. Adequate reactions can be observed in the web interface. These

are also consistent with the logged *MessageEvents*. The successful execution of this test is a first step towards fully verifying the functionality of the test tool, before it may be applied in the field.

So far, the presented test cases don't implement a closed loop HIL simulation, since the test tool can only access the emulator and not the energy manager. A closed loop would supply artificial data for the SUT and observe the reactions automatically [16]. Alternatively, an oscilloscope could be used to monitor the 1-Wire bus. In this way, the test tool could analyze the data bytes of the communication and cross-check, whether the transmission was correct. If the oscilloscope provides a UART interface, the *Interface* that is used for the emulator could be reused. This simple example illustrates how the modular approach can speed up the development of new test applications in the future, by reusing existing components in a new context.

V. Conclusion

In this paper, a lightweight and highly customizable test tool for automated regression and HIL tests is proposed. It can be implemented on small embedded controllers running either a Linux or Windows operating system and allows the utilization of built-in hardware modules, peripheral sensors etc. The proposed test tool consists of two major components which are also modular in themselves. The core system provides the routines, necessary for the program flow, such as logging or the GUI. It also implements the event system, which is used for the inter-module communication. Decoupling the individual modules of the core system simplifies debugging and further development of these components. The core system is independent of the actual HIL test and doesn't need to be modified, when changing applications. The test implementations on the other hand are specific to a given application and need to be rewritten for each new HIL test. Since, this component is also modular in itself the development cycles will continue to get shorter in the future, as existing components may be reused in a new context. The first test implementation, as presented in this paper, yields promising results. In the next step full verification of the proposed test system is planed, before it is deployed in the field for testing the energy manger modules under real-life conditions.

Although the tool meets the defined goals and requirements, especially in terms of flexibility and customizability, a number of features that may be implemented in the future are currently being discussed. Currently the GUI only supports viewing the Python files of the test implementations. By implementing a simple text editor minor customizations could be done in a more efficient way. In a further step, a graphical interface for creating the test implementations (similar to LabVIEW or MATLAB Simulink) would be a valuable feature. At the moment writing the scripts requires some knowledge of Python scripting. By implementing a graphical interface, test customizations could also be performed by less experienced personnel.

Acknowledgment

The research presented in this paper is part of the project "Smart Charging Community" which is funded as ZIM cooperation project by the German Federal Ministry for Economic Affairs and Energy.

References

[1] A. Jha. "Development of test automation framework for testing avionics systems". In: *Digital Avionics Systems Conference (DASC), IEEE/AIAA 29th*. Salt Lake City, USA, 2010, pp. 6.E.5–1–11.

[2] A. Bansal, M. Muli, and K. Patil. "Taming complexity while gaining efficiency: Requirements for the next generation of test automation tools". In: *IEEE AUTOTESTCON*. Schaumburg, USA, 2013, pp. 1–6.

[3] E. Gamma et al. *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, USA: Addison-Wesley Professional, 1994.

[4] V. Savikko. "Design Patterns in Python". In: *Proceedings of the 6th International Python Conference*. San Jose, USA, 1997.

[5] S. Tumin and S. Encheva. "Enterprise-Wide Logging Through Software Life-Cycle". In: *IJCSI International Journal of Computer Science Issues* 10.2, No 3 (Mar. 2013), pp. 88–95.

[6] A. Gustafsson. "Threads Without the Pain". In: *Queue - Social Computing* 3.9 (Nov. 2005), pp. 34–41.

[7] D. Hellmann. *The Python standard library by example*. Upper Saddle River, USA: Addison-Wesley, 2011.

[8] *Standard Commands for Programmable Instruments (SCPI)*. SCPI Standard Volume 1: Syntax and Style. SCPI Consortium, 1999.

[9] A. Perić and M. Bjelica. "Cost–Efficient Phase Noise Measurement". In: *Journal of Electrical Engineering* 65.3 (May 2014), pp. 189–192.

[10] S. Michalak. "Raspberry Pi as a measurement system control unit". In: *International Conference on Signals and Electronic Systems, 2014 ICSES*. Poznan, Poland, 2014, pp. 1–4.

[11] J. M. Hughes. *Real World Instrumentation with Python: Automated Data Acquisition and Control Systems*. O'Reilly Media, 2010.

[12] U. Vigenschow. *Testen von Software und Embedded Systems: professionelles Vorgehen mit modellbasierten und objektorientierten Ansätzen*. dpunkt.verlag, 2010.

[13] D. Awtrey. "Transmitting Data and Power over a One-Wire Bus". In: *Sensors* (Feb. 1997).

[14] *DS18B20 Programmable Resolution 1-Wire Digital Thermometer*. Data Sheet Revision 042208. Maxim Integrated Products, Inc., 2008.

[15] M. Eider, S. Kunze, and R. Poeschl. "FPGA Based Emulation of Multiple 1-Wire Sensors for Hardware in the Loop Tests". In: *2016 IEEE Sensors Applications Symposium (SAS 2016) Proceedings*. Catania, Italy, 2016, pp. 279–284.

[16] M. Schlager, W. Elmenreich, and I. Wenzel. "Interface Design for Hardware-in-the-Loop Simulation". In: *2006 IEEE International Symposium on Industrial Electronics*. Vol. 2. Montreal, Canada, 2006, pp. 1554–1559.