Automated Testing of Simulation Software in the Aviation Industry: An Experience Report

Garousi, V., Tasli, S., Sertel, O., Tokgoz, M., Herkiloglu, K., Arkin, H. F. E., & Bilir, O. (2019). Automated Testing of Simulation Software in the Aviation Industry: An Experience Report. *IEEE Software*, *36*(4), 63-75. [8356168]. https://doi.org/10.1109/MS.2018.227110307

Published in:
IEEE Software

Document Version:
Peer reviewed version

This is the pre-print of a paper that has been published in the IEEE Software magazine:
https://doi.org/10.1109/MS.2018.227110307

Automated testing of simulation software in the aviation industry

Vahid Garousi
Wageningen University
Wageningen, the Netherlands
vahid.garousi@wur.nl

Seçkin Taşlı, Onur Sertel, Mustafa Tokgöz
Kadir Herkiloğlu,
Hikmet Ferda Ergüneş Arkın, Osman Bilir
HAVELSAN A.Ş.
{stasli, osertel, mtokgoz, kherkiloglu, fergunes, obilir}@havelsan.com.tr

1 INTRODUCTION

To develop high-quality software systems, software testing is a critical phase of any software development process. However, software testing is expensive and makes up about half of the development cost of an average software project [1]. According to a 2013 study [1], the global cost of finding and removing defects from software rose to $312 billion world-wide annually as of 2013.

Different steps of software testing can be conducted manually or automated. In manual testing, the tester takes over the role of an end-user executing the software under test (SUT) to verify its behavior and find any possible defects. However, in automated testing, using certain test tools (e.g., the JUnit test framework), test-code scripts are developed and executed without human testers' intervention to test the behavior of the SUT. If it is planned and implemented properly [2, 3], automated testing could provide various benefits over manual testing, e.g., repeatability and reducing testing effort and costs. However, if not implemented properly, automated testing could lead to extra cost and effort and might be even less effective than manual testing in detecting faults [4].

Motivated by a real industrial need, we started an industry-academia collaboration to develop and evaluate a test framework (named DORUK) for automated execution of black-box test cases on simulation software in the context of a major defense contractor in Turkey. The tool is currently widely adopted and is actively utilized by a large team of test engineers in the company to conduct testing of a suite of aviation and helicopter simulators.

This paper presents the motivations and needs for test automation in this context, the framework that we developed and its evaluations, and the success story that we had in deploying the technology among many test teams in the company. The paper aims at sharing our experience in developing this test automation approach and its evaluation with other practitioners and researchers for the purpose of contributing to the body of knowledge and evidence in this area, e.g., [5]. As another recent study [6] showed, the testing community in academia and industry are "*living in two different worlds*" (in terms of their focus areas). Many industrialists want to find ways to improve effectiveness and efficiency of testing (e.g., by using more and better test automation), they do not show much eagerness for sophisticated formal methods or techniques which they mostly find to be too complicated and hard to implement in practice, e.g., combinatorial testing and search-based test-case design, which may not work in large-scale practice. Thus, the project behind the work reported in this paper was initiated with the goal of connecting industrialists and researchers to develop innovative ways to improve effectiveness and efficiency of test execution.

2 A REVIEW OF THE INDUSTRIAL CONTEXT AND THE NEED FOR THE PROPOSED FRAMEWORK

HAVELSAN is a large Turkish software and systems company providing global solutions in the areas of defense and IT sector. HAVELSAN is mostly active in the fields of naval combat systems, e-government applications, reconnaissance surveillance and intelligence systems, simulation and training systems, homeland security systems, energy management systems, and Command and Control Communications, Computers, Intelligence, Surveillance and Reconnaissance Systems (C4ISR).

This is the pre-print of a paper that has been published in the IEEE Software magazine:
https://doi.org/10.1109/MS.2018.227110307

In terms of organization-wide software engineering capabilities, HAVELSAN has achieved the CMMI level-3 certification. There is an independent testing department in the company, consisting of five test teams:

- Test team for automation applications and image processing technologies
- Test team for ground support systems
- Test team for platform-stationed systems
- Test team for platform integration
- Test team for simulation, training and electronic-warfare technologies

Essentially, each test team is dedicated for each major business unit in the company. In total, more than 40 test engineers work in the above five test teams. The test department has been established based on the principles of Independent Software Verification and Validation (ISVV) which has been developed world-wide mostly by defense and aerospace industries, e.g., NASA and DoD. Almost all of the test activities conducted by the group are black-box testing, since the white-box test activities have been assigned to and are conducted by the development team before handing the software to the test team for black-box testing.

Based on the type of systems developed by HAVELSAN (i.e., safety- and mission-critical systems), the company in general, and the testing department in particular, is constantly striving to perform software engineering tasks more effectively and efficiently and various process improvement initiatives are constantly underway. One of those continuous improvement initiatives has been a recent industry-academia collaborative project in which university researchers and practitioners in the company have come together to improve the maturity of test practices and activities. Among the improvement areas is the introduction of test automation in various test teams and this paper falls in the scope of those efforts. Similar to many other firms, historically, the majority of test activities has been conducted manually in the company. However, adoption of test automation practices is constantly increasing across various test groups in the company since several years ago.

One of test teams (as discussed above) is the team responsible for testing simulation, training and electronic-warfare systems developed by HAVELSAN [7]. With continuous pipeline of new systems in this area, and also maintenance of existing systems (thus the need for regression testing), this team is constantly tasked with new testing projects. We focus on and discuss one particular type of such systems under test (simulation software for defense systems such as helicopters) for which the need for test automation arose, and a test framework was subsequently developed.

Figure 1 shows the actual hardware of one specific product, called *HELSIM* (helicopter simulator), and a view of inside the simulator cabin which highly resembles an actual helicopter. The software systems running on these simulators are complex, large-scale and have to interact with various external systems, e.g., hardware and devices onboard (e.g., sensors), and map systems. It goes without saying that quality in this context is of utmost importance since student pilots learn flying via these systems and will fly actual helicopters soon afterwards.

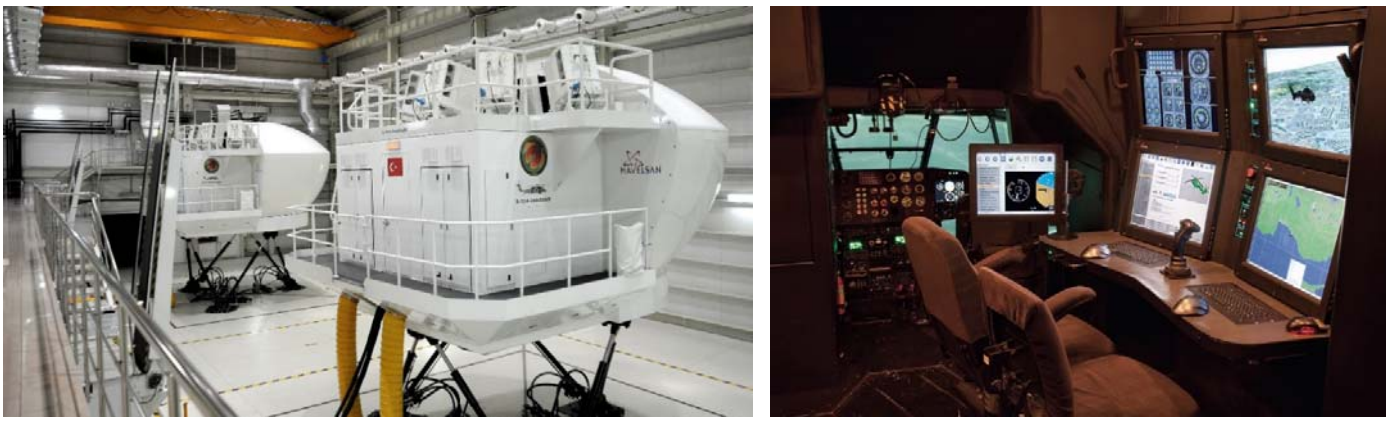


Figure 1-Screenshots from the *HELSIM* system (helicopter simulator)

This is the pre-print of a paper that has been published in the IEEE Software magazine:
https://doi.org/10.1109/MS.2018.227110307

Traditionally, similar to other systems, black-box testing of such simulators was being conducted in three classical phases: (1) unit testing of individual software and hardware components, (2) integration testing, and (3) system testing. Let us recall the reader that unit and integration testing can be done in both black-box and white-box approaches [8]. When unit testing is done in black-box approach, test cases are developed without the knowledge of source-code, e.g., branches and statements, and black-box techniques are used on the inputs of functions and classes, as provided by requirements. This is often conducted by calls to the public-facing API's. This approach is also sometimes referred to as "gray-box" testing. When unit testing is done in white-box approach, test cases are developed using the knowledge of the source-code, e.g., branches and statements, using control-flow and data-flow techniques.

In the recent past, all these three phases were carried out manually, with the help of a data injection and monitoring tool called *DIM (Data Injection Monitor)* which was acquired from a third-party vendor in the defense domain. This tool made it possible to monitor and modify the contents of messages sent/received by each software module in a typical simulation software.

The simulation systems were extensively tested in the past using manual testing practices. Since the company has CMMI level-3 certification, manual testing practices were already mature, e.g., test-case design of manual tests were done using systematic approaches and based on 100% requirements coverage. Careful traceability analysis of tests back to requirements were also conducted.

During manual system testing phase, test engineers would take over the role of end-users (student pilot and trainer) interacting with various controllers (as shown in Figure 1) to verify the system's behavior. DIM was also used in this stage for analysis of detected defects (e.g., root-cause analysis and fault localization). In fact, DIM played the roles of a test stub, a test driver and also a probing tool. As it is the case in almost any domain, manual testing using the DIM tool showed to be very tedious, error prone and effort-intensive. Thus, to improve effectiveness and efficiency of manual testing pratices, there was a need for a proper infrastructure and system for test automation in this context. To adopt or develop such a suitable system, we first conducted a planning activity in which we set out the requirements for the test tool, as reported next.

3 PLANNING FOR TEST AUTOMATION AND TEST TOOL REQUIREMENTS

To plan test automation, it was important to understand the software architecture of the simulation SUT. Figure 2 shows a typical software architecture of a typical simulation software, based on the 'event-driven' architectural pattern. There is a central '*Mission and Flight System*' which has direct connections to the actual simulation hardware (e.g., actuators, sensors and mechanical arms moving the cabin). As Figure 2 shows, all other systems are connected to the central mission and flight system, e.g., weather and terrain servers, and instructor console.

This is the pre-print of a paper that has been published in the IEEE Software magazine:
https://doi.org/10.1109/MS.2018.227110307

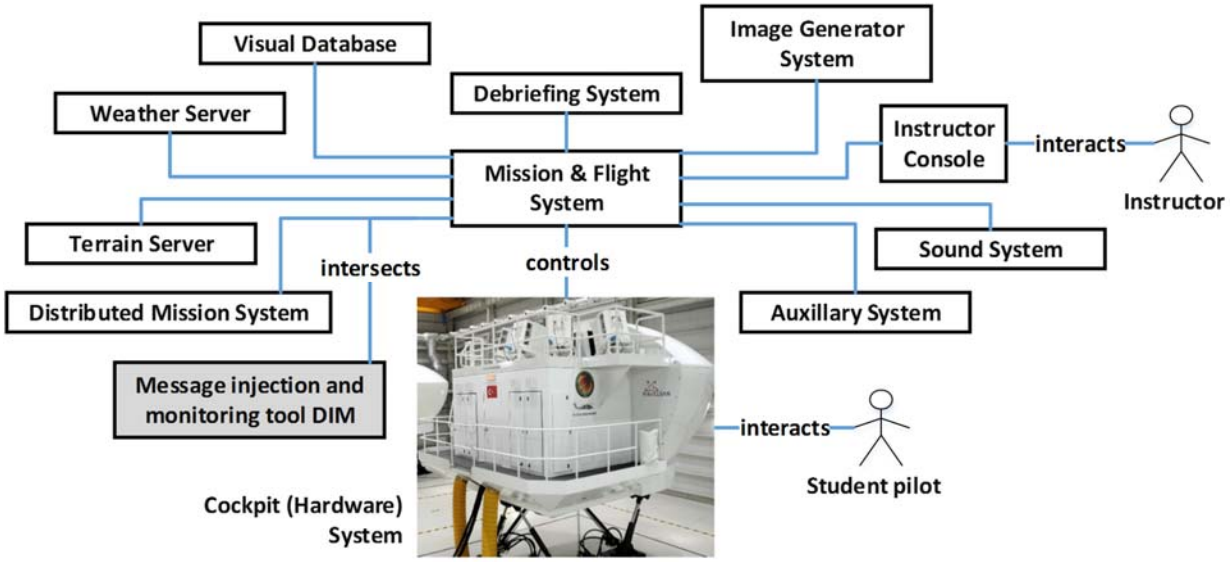


Figure 2-A typical software architecture of flight simulation software

As discussed above, the DIM tool was used to manually test each unit and their integration. As Figure 2 shows, this was done by having the tool intercept with each of the connected links between the modules or by interacting with each system as a unit under test. Each system provides a set of interfaces (as an API) to send/receive messages which should be carefully tested. DIM allows testers to generate messages on demand manually as well as by modifying messages and monitor messaging between integrated modules. This was achieved by selecting desired messages via DIM's UI and specifying content of each messages one by one if modification was necessary. DIM was already able to operate on almost all internal and external messages of the SUT. In addition to classical disadvantages of manual testing, there were several additional technical limitations when conducting manual testing using DIM:

- It was not possible to send specific messages when specific conditions were met, e.g., when a helicopter went above a certain altitude.
- It was not possible to define message sequences or execute those sequences over a specific time frame. Users could only send/mutate a single message by hand and then had to verify the system's output/behavior manually using human judgement.
- In the case of messages and system behavior which were so quick to be noticed by human eye, it was not possible to easily verify system outputs and behavior, e.g., when the helicopter's engine RPM (rounds per minute) was changing so fast.
- It was not easily possible to systematically get logs of and later draw graphs of specific parameters under test.
- It was not possible to send messages periodically or to conduct periodic controls of certain tasks, e.g., every one second.

As discussed earlier, manual testing practices in the company were already mature, e.g., test-case design of manual tests were based on 100% requirements coverage. The only open gap was the ineffectiveness and inefficacy of manual testing practices which our project aimed at addressing. With the right test automation infrastructure in place, similar to previous works, e.g., [9], we only needed to "transition" from manual test cases to automated tests. To transition from manual testing to test automation in this context, several meetings were conducted among testers and developers, and we laid down the test tool requirements as follows.

The automation tool that we were looking for had to be easily adaptable to test different SUTs, i.e., different models and types of simulation systems. It was also important for the new tool to be open for modifications to satisfy further feature requests from both testers and developers in the company. It was necessary that writing tests using the tool did not require expert-level coding skills and testers should be able to write test scripts with minimum training. Since the SUT are also real-time, the test tool should impose minimum 'probe effect', i.e., time overhead on test execution due to having the tool in the

This is the pre-print of a paper that has been published in the IEEE Software magazine:
https://doi.org/10.1109/MS.2018.227110307

system, or otherwise, the real-time properties of the system functionalities would be under jeopardy. Last but not the least, the test automation tool, that we were looking for, was expected to have a high performance (in terms of response time) as simulator testing, in our context, requires monitoring and generation of messages in very high frequencies (throughput rates), e.g., sending and receiving messages from/to helicopter radar systems at frequencies over 10,000 messages per second.

Based on the above needs (test tool requirements), the teams spent a few months to research and explore existing test tools in both the open-source community and the commercial tools market. As tools for testing simulation systems in the aviation and defense sector are usually "classified" in almost any firm, and are seldom offered in the public domain, we were unable to find any existing tool that would come close to fulfilling our requirements. We also already had a framework (a Python library), developed in-house a few years ago, to access messaging data which was primarily used by the development team in conjunction with DIM.

We were also aware of several available RESTful API frameworks for Python. However, after trying to integrate the existing REST API frameworks into our existing toolset, our experience was negative. At the same time, we wanted to reuse and build on top of our existing toolset (for tasks such as accessing message data). In the process of searching for existing test frameworks to integrate with our messaging framework, we came across a framework called the *Software Testing Automation Framework (STAF)* [10], an open-source, multi-platform, multi-language framework designed around the idea of reusable components and services (such as process invocation, resource management, logging, and monitoring). STAF removes the burden of building an automation infrastructure, thus enabling one to focus on building automation solutions. The STAF framework also provides the foundation upon which to build higher-level solutions, and provides a pluggable approach supported across a large variety of platforms and languages. We thus decided to adopt STAF and build our automation tool on top of it.

4 DORUK: FEATURES AND EXAMPLE USAGE

The framework that we developed, named DORUK, is essentially a Python library, a set of functions which serve as a framework allowing test engineers to access the contents of input/output messages, sent to/from sub-systems of a given simulation SUT. To provide an overview on DORUK, Figure 3 shows the general test architecture of our approach and the general usage scenario of DORUK framework. When reviewed together with Figure 2, Figure 3 shows that test engineers write test scripts in Python which then make function calls to DORUK interfaces which in turn trigger message calls to the SUT to test it. Those messages are sent over the LAN based on generic protocols such as TCP or the ones specific to the defense domain, e.g., MIL-STD-1553 and MilCAN. General usage and purpose of DORUK are quite similar to the popular xUnit frameworks (e.g., JUnit) in that a test engineer develops test scripts using the functions provided via DORUK's API to exercise the SUT and verify its functionality.

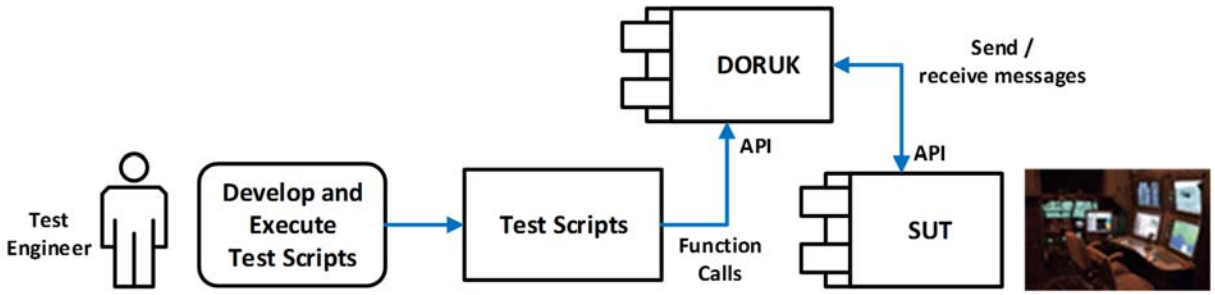


Figure 3- Test architecture and general usage scenario of DORUK

Here are few example test scenarios where DORUK provides its usefulness:

- Flying a helicopter from the point *a* to *b*, and verifying that fuel consumption and levels along the way change as expected
- Testing the effects of a certain intentional instructor-initiated malfunctions (e.g., engine fire) in the helicopter simulator, and verifying the proper behavior/outputs of the system and reaction of the student pilot

This is the pre-print of a paper that has been published in the IEEE Software magazine:
https://doi.org/10.1109/MS.2018.227110307

- Reproducing / replaying actions of student pilots for problem analysis and resolution after the training sessions

DORUK was designed to complement the manual GUI-based message-injection tool DIM by providing a test automation approach in the API level. The main features of DORUK are summarized as follows:

- Allowing automation for both black-box unit testing of each module and testing of the integrated system.
- Messages can be sent / modified either periodically or immediately.
- Message contents can be retrieved at any time.
- All messages can be monitored continuously. Previously executed actions can be re-executed when specific conditions are met.
- Predefined action sequences can be executed repeatedly, e.g., a take-off can be performed multiple times for different aerodynamic adjustments.
- A set of desired messages can be recorded for a specified duration. These recordings can be replayed later on.
- Graphs of desired parameters (value versus time) can be auto-generated.
- Capability of working on both Linux and Windows operating systems.
- Allowing semi-automated testing, where testers are asked to perform GUI actions or enter user inputs when necessary (when the automation of these tasks is not easily possible).
- Asynchronous execution of defined actions or controls is possible. In this case, parallel threads are generated and executed by the tool.

We provide below several example test scripts and explain some of the provided API functions. Figure 4 shows several example test scripts written and one output graph generated using DORUK.

Each test script (case) starts and ends with `start_flow()` and `end_flow()` functions. These two functions match the 'setup' and 'teardown' phases of the four-phase test pattern [12]. In our context, main test flows and sub-flows are defined as follows. A main flow is a standalone test procedure, e.g., to test fuel consumption during a given flight trip, we verify the verification of fuel consumption inside the main flow. However, the flight level variable, which is a separate condition that needs to be satisfied, is handled by another DORUK script, which is initiated from the main script and executes asynchronously while the main script measures fuel consumption.

The `note_arithmetic()` function assigns an arithmetic expression (value) to a variable name. The `set()` function sets the value of the given parameter to the given value and logs the result of this operation. The `note()` function assigns an alias name to the current value of a given parameter in a message. This value can be referenced later on, during the lifetime of the script, using the `${<noted_alias>}` notation. The `compare()` works similar to the `assert()` family of functions in JUnit, and matches the 'verify' phase of the four-phase test pattern [12]. If the value of the given parameter satisfies the comparison expression, the function logs a 'success', and else it logs a 'failure'.

The `manual_check()` function displays a message and blocks script execution until user presses the 'Enter' key. This function function prints the given message to the screen and waits for the user to evaluate the current situation manually to continue. This function is useful when the verification phase of th etest case cannot be easily automated and shall be done manually, e.g., in the test script of Figure 4-(b), the test script asks the user whether s/he can see a green box on top of the screen.

Last but not least, based on the need of the test team, we incorporated a powerful graphing feature in DORUK. The `start_graph()` function draws a graph of a set of given parameters. For example, the example test script in Figure 4-(c) asks DORUK to draw a graph of an important tactical variable, fuel tank level of the helicopter, for a duration of 1000 seconds in the interval of one second. The resulting graph is shown in Figure 4-(d).

This is the pre-print of a paper that has been published in the IEEE Software magazine:
https://doi.org/10.1109/MS.2018.227110307

```
#-*- coding:utf-8 -*-
from test_case_funcs import *

start_flow("User_defined_script_name")

note_arithmetic("8", "notedVal")
Set("TacticPlatformDataMsg.tacticPlatform[0].location.latitude", "(${notedVal} + 4)/2 ")
Set("TacticPlatformDataMsg.tacticPlatform[0].location.longitude", "10")
Set("SimControlMsg.simulationMode", "tsimulationmode_RUN")
note("TacticPlatformDataMsg.tacticPlatform[0].location.longitude", "notedLon")
compare([["TacticPlatformDataMsg.tacticPlatform[0].location.latitude", "lat"],
         ["TacticPlatformDataMsg.tacticPlatform[0].location.longitude", "lon"],
         ["SimControlMsg.simulationMode", "simMode"]], "((lat < ${notedLon})
         AND (simMode == $[tsimulationmode_RUN])) OR NOT lat > 0")
end_flow()
```

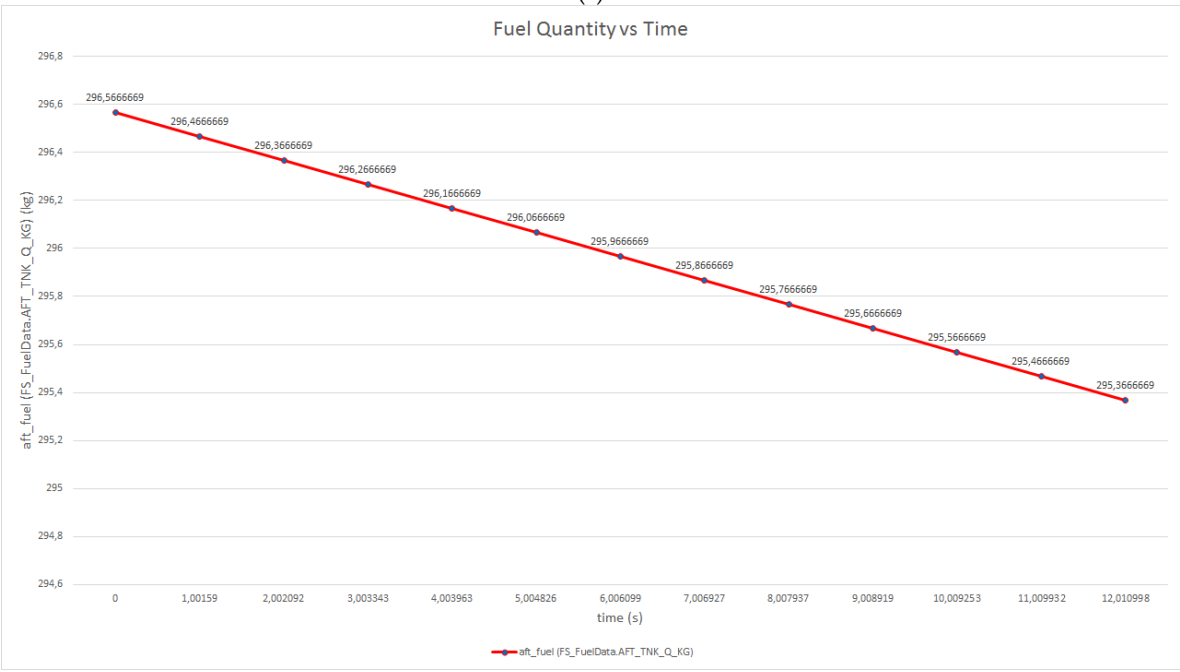
(a)

```
#-*- coding:utf-8 -*-
from test_case_funcs import *

start_flow("User_defined_script_name")

manual_check("Do you see a green box on top of the screen?")

end_flow()
```

(b)

```
#-*- coding:utf-8 -*-
from test_case_funcs import *

start_flow("fuel", "FUEL_TEST")

start_waiting_for([["FD_OwnshipData.location.altitude", "alt"]], "alt > 1100", duration=1000, interval=1)
end_wait()

start_graph([["FD_OwnshipData.location.altitude", "alt"],
             ["FS_FuelData.AFT_TNK_Q_KG", "aft_fuel"]], csvName="fuelConsumption", interval=1)
start_waiting_for([["FD_OwnshipData.location.altitude", "alt"]], "alt > 1200", duration=1000, interval=1)
end_wait()
end_graph()

end_flow()
```

(c)



This is the pre-print of a paper that has been published in the IEEE Software magazine:
https://doi.org/10.1109/MS.2018.227110307

(d)

Figure 4- Three example test scripts written, and an output graph generated automatically using DORUK

As we can see in the above examples, DORUK was designed to conduct API (message)-level 'horizontal end-to-end testing' [11]. Let us recall from the software testing literature that horizontal (scenario-based) end-to-end (E2E) testing is a scenario-based approach to test a system in an "end-to-end" manner, e.g., starting the engines of a helicopter, taking it off, flying it to a destination and then landing it. In design and development of DORUK, we also utilized the principles of the xUnit family of unit test frameworks and also several test patterns [12], e.g., the four-phase test pattern: setup, exercise, verify, teardown. We chose API-based automated testing as our strategy since, according to the latest recommendations in the test automation community, API-based automated is becoming very popular in the industry, e.g., according to a recent 2016 keynote [9] by a practitioner in an international workshop, "*Future test automation must focus on API testing*". On the same topic, we also utilized the concept and the metaphor of test automation "pyramid" which has become very popular in the last several years in the testing community, e.g., [13, 14]. This pyramid metaphor suggests that, instead of many automated UI tests, a good strategy is to develop more automated API and unit tests. This is what we aimed at by developing the DORUK framework.

In summary, DORUK aimed at reducing significantly the testing effort by automating all the steps of manual testing by enabling testers to specify all the manual test steps in automated scripts. For example, before having DORUK, for the case of test scenario in Figure 4-(c), the tester had to manually run the scenario using DIM and several other complex tools, then had to "pause" the execution in several intervals to probe and verify the value of various variables. Simply put, the manual test effort was a major issue before DORUK was developed.

5 DEVELOPMENT DETAILS AND ARCHITECTURE OF DORUK

DORUK has been developed in Python itself and has about 4 KLOC. Figure 5 shows the software architecture of DORUK. We explain next some of its components. `test_case_funcs.py` is the module for DORUK test scripts which is called by any test script developed by test engineers. It provides a set of predefined functions such as `set()`, `note()`, `graph()`, `record()`, `compare()`, `periodic_set()`, `wait()`. A formal test script shall only use these functions sequentially. `FlowManager.py` is a module for managing threads and processes. `Logger.py` is the logging module. `RawDataAccessor.py` is the module to access the contents of (raw) messages. It provides basic functions, mainly `set()`, `get()`, `send()`. This can be used as a point of access for scripts that are not used for formal tests (no logging of execution) and in cases when a tester only wants to access a particular data point in the SUT. The advantage in such cases is that since there will be no extra layers for data access (thus, no overhead), the operation will be quite fast. This is particular popular among developers and has made DORUK widely used for developer testing.

For any arbitrary number of messages under monitoring, the `set()` functions are used as shown in Figure 5, e.g., `SetBMA` standing for `SetBMessageAccess`. `DORUKConfigAccessor` is a configuration file to specify configuration parameters such as "logging level", "default interval between steps", and locations of .so files to be used by Python.

This is the pre-print of a paper that has been published in the IEEE Software magazine:
https://doi.org/10.1109/MS.2018.227110307

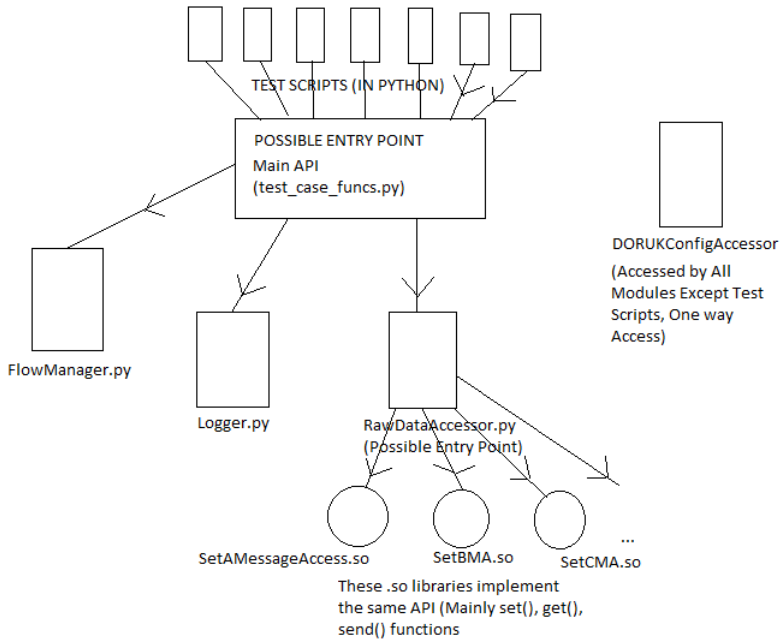


Figure 5-Software architecture of DORUK

6 EVALUATION OF THE AUTOMATED TESTING APPROACH

Since the tool was developed based on a real industrial need, once the tool development finished, we started evaluating the tool on a set of test projects to assess how it really addressed the need behind it. As of this writing, DORUK is actively being used by several full-time test engineers in several critical helicopter simulator and weapon simulator testing projects (exact numbers and project details cannot be disclosed due to confidentiality).

To assess the effectiveness of the approach and tool, we conducted an internal empirical case-study in the company. As the research approach, we used the Goal, Question, Metric (GQM) methodology [15]. Stated using the GQM's goal template, the goal of the case study has been to evaluate the extent to which DORUK helps our test engineers in increasing the efficiency and effectiveness of testing when compared to manual testing using the previous DIM tool. As metrics to assess efficiency and effectiveness, we selected the test effort (in hours) and the number of defects found (with and without the new test tool), respectively. Unfortunately, due to confidentiality, we cannot disclose the exact quantitative metrics values for efficiency (decreasing test efforts) and effectiveness (in detecting real faults), but the measurements have clearly indicated that the tool has been for sure instrumental in helping test engineers test more effectively and efficiently. Test efforts have been reduced several-fold. For readers interested in the types of measurements we gathered to assess the success of the approach (based on the above metrics), we would like to refer to our previous projects with industry for which we had the permission to publish details of test data and results, e.g., [16].

Here are also some lessons learnt after using DORUK for several months in multiple projects. The framework provides an easy to understand test scripting language (from both syntax and logic standpoints) which in turn leads to consistency and coherence across test teams, which is an important benefit and facilitates training and test maintenance activities. Thanks to the structured nature of DORUK, test scripts are easy to write and maintain. All these positive features have made DORUK a powerful and popular framework for automated testing of simulation software among many test teams already. In addition to its popular benefits in testing, DORUK is currently used also for simulating the behaviors of software components during software development and integration phases, i.e., by writing a script in DORUK, one can stimulate (exercise) a given component in a certain way, which was previously very tedious when done manually.

We have also observed that DORUK encourages inexperienced testers who usually have lack of exercise in development and using complex testing tools, to easily learn automated testing gradually. We have prepared a detailed user guide for

This is the pre-print of a paper that has been published in the IEEE Software magazine:
https://doi.org/10.1109/MS.2018.227110307

DORUK with many example scripts which has been helpful in training of our testers to use this framework. In summary, we have seen that our testers who were previously spending the majority of their time on manual testing have efficiently "transitioned" to test automation. This is fully in alignment with a quote from a keynote [9] in a recent 2016 testing conference (TAIC-PART): "*Manual testers become automation specialists*". Similar to other contexts and reported experience, e.g., [17], while some manual testers first treated the test automation as a competitor (threat) to their manual testing jobs, after seeing how this test framework and the new approach could help them in doing what they used to do more effectively and efficiently, they embraced the new approach and started to become test automation engineers. The management also supported this transition wholeheartedly. Test engineers could now execute more tests in less time and with higher precision (meaning less false-positive test results). Observing DORUK automate test steps like a robot also made the user-acceptance-test (UAT) sessions with clients more refreshing and efficient.

Last but not the least, we were aware of the "learning curve" and initial resistance factors in the context of our automation approach. As Robert Glass [18] nicely put: "*adopting new tools and techniques is worthwhile, but only if you (a) realistically view their value and (b) use patience in measuring their benefits*". We ensured that the value of the toolset and need for patience to see the benefits were clearly communicated across all team members.

7 Related work and tools

There is a shortage of activity in the scientific peer-reviewed literature about test automation in general and automated API testing in particular. Some activity on test automation has started recently in the research community (see the review paper in [3]). That is the reason why we have cited several sources from the grey literature throughout this paper.

As discussed earlier, tools for testing simulation systems in the defense sector are usually "classified" in almost any defense firm, and are seldom offered in the public domain. Thus, we were not able to find any tool/study that would closely match our context. Here we only discuss several remotely-related works, i.e., in the context of automated API testing. API testing has become popular in the industry, e.g., [19], and is reported to be critical for automating testing because APIs now serve as one of the primary interfaces to application logic of many systems.

Automated API testing has become popular in the industry, e.g., [19, 20], and is reported to be critical for automating testing because APIs now serve as the primary interface to application logic and because GUI tests are difficult to maintain. Automated API testing does not seem to be an active area of academic research, but there has been a lot of attention on this area by practitioners in the grey literature (e.g., blogs, white papers and web-pages) [21].

In a white paper entitled "*Produce Better Software by Using a Layered Testing Strategy*" [19], a practitioner discussed different approaches to test automation across multiple layers.

Among the academic studies on API testing is an industry-academic collaborative work in which the last author of this paper was involved in Canada in 2010 [16]. In that joint project, an open-source tool for automated generation of black-box NUnit API test code was developed and was later empirically evaluated in an industrial context. We can conclude that there is a need for more work in the research community on API testing and tool support for that purpose.

Last but not the least, many members of the community including the authors, believe that experience reports on actual applications of test automation in industrial practice are rare. Several interesting case studies have been reported in a 2012 edited book by Graham and Fewster [13]. We added to that body of evidence in this paper by proposing a tool to support automation and a supporting case study based on action-research.

8 Summary, ongoing and future works

Motivated by a real industrial need, we developed and presented in this work a test automation approach and framework (named DORUK) to conduct API-based black-box testing of a suite of simulators in the company. Although due to confidentiality, we could not provide quantitative data about defect detection rate using the tool, we have observed that it has been instrumental in helping test engineers test more effectively and efficiently by enabling them to write robust test scripts which are easy to maintain. Given the sensitive and classified nature of the context (defense), we were unable to

provide the tool as open-source or as a commercial tool. But we believe that practitioner readers of this paper can easily adopt the ideas behind this tool in their own test automation contexts and needs.

The project was initialed and completed based on the principles of "action-research" and the industry-academia collaborations was seen as a success story from both sides, thus adding to our experience in conducting industry-academia collaborations [5].

In terms of ongoing and future works, we are currently using a GQM-based approach to further evaluate and improve effectiveness and efficiency of DORUK and test suites developed using it. We are also exploring the applicability of DORUK in other testing needs in the company. In the future, we plan to improve DORUK so that it would support the "client-server" architecture. Such an architecture will enable us to test SUTs which are distributed. Also, we intend to integrate (combine) the API testing feature of DORUK with GUI testing features of a GUI testing tool used in the company (called Sikuli www.sikuli.org). GUI testing tools such as Sikuli automate anything that is seen on the screen, and use image recognition to identify and control GUI components. On the other hand, DORUK has the capability to access and manipulate system messages of simulation software without accessing the GUI. Based on our initial feasibly analysis, integration of DORUK with Sikuli seems to be a good solution to combine the automated testing powers of these two independent tools on graphical and API layers of a given SUT. Also, as another important usage for DORUK, discussions are underway in the company to use it as a support tool for Qualification Test Guide (QTG), which is a guide for certifying new flight simulation technologies in the context of regulatory bodies.

REFERENCES

[1] T. Britton, L. Jeng, G. Carver, P. Cheak, and T. Katzenellenbogen, "Reversible Debugging Software," University of Cambridge, Judge Business School2013.

[2] V. Garousi and F. Elberzhager, "Test automation: not just for test execution," *IEEE Software, In Press,* 2017.

[3] V. Garousi and M. V. Mäntylä, "When and what to automate in software testing? A multi-vocal literature review," *Information and Software Technology,* vol. 76, pp. 92-117, 2016.

[4] Y. Amannejad, V. Garousi, R. Irving, and Z. Sahaf, "A Search-based Approach for Cost-Effective Software Test Automation Decision Support and an Industrial Case Study," in *Proc. of International Workshop on Regression Testing, co-located with the Sixth IEEE International Conference on Software Testing, Verification, and Validation*, 2014, pp. 302-311.

[5] V. Garousi, M. M. Eskandar, and K. Herkiloğlu, "Industry-academia collaborations in software testing: experience and success stories from Canada and Turkey," *Software Quality Journal, special issue on Industry Academia Collaborations in Software Testing,* pp. 1-53, 2016.

[6] V. Garousi and M. Felderer, "Living in two different worlds: A comparison of industry and academic focus areas in software testing," *IEEE Software, In press,* 2017.

[7] HAVELSAN, "Simulation products developed by HAVELSAN," *http://www.havelsan.com.tr/ENG/Main/urun/1021/simulation-products*, Last accessed: Feb. 2017.

[8] S. Benli, A. Habash, A. Herrmann, T. Loftis, and D. Simmonds, "A Comparative Evaluation of Unit Testing Techniques on a Mobile Platform," in *2012 Ninth International Conference on Information Technology - New Generations*, 2012, pp. 263-268.

[9] G. Weishaar, "The Future of Testing: How to build in quality & efficiency right from the start?," *Keynote, the Workshop on Testing: Academia-Industry Collaboration, Practice and Research Techniques (TAIC PART), https://goo.gl/orDcTb,* Last accessed: May 2016.

[10] Authors unknown, "Software Testing Automation Framework (STAF)," *http://staf.sourceforge.net/*, Last accessed: May 2016.

[11] Author(s) unknown, "Why End to End Testing is Necessary and How to Perform It?," *http://www.softwaretestinghelp.com/what-is-end-to-end-testing/*, Last accessed: May 2016.

[12] G. Meszaros, *xUnit Test Patterns*: Pearson Education, 2007.

[13] D. Graham and M. Fewster, *Experiences of Test Automation: Case Studies of Software Test Automation*: Addison-Wesley, 2012.

[14] M. Cohn, "The Forgotten Layer of the Test Automation Pyramid," *https://www.mountaingoatsoftware.com/blog/the-forgotten-layer-of-the-test-automation-pyramid*, 2009, Last accessed: Feb. 2017.

This is the pre-print of a paper that has been published in the IEEE Software magazine:
https://doi.org/10.1109/MS.2018.227110307

[15] V. R. Basili, "Software modeling and measurement: the Goal/Question/Metric paradigm," Technical Report, University of Maryland at College Park1992.

[16] C. Wiederseiner, S. A. Jolly, V. Garousi, and M. M. Eskandar, "An Open-Source Tool for Automated Generation of Black-box xUnit Test Code and its Industrial Evaluation," in *Proceedings of the International Conference on Testing: Academic and Industrial Conference - Practice and Research Techniques (TAIC PART)*, 2010, pp. 118-128.

[17] U. Eriksson, "Will automation kill the software tester?," *http://reqtest.com/testing-blog/will-automation-kill-the-software-tester/*, Last accessed: Feb. 2017.

[18] R. L. Glass, "Frequently forgotten fundamental facts about software engineering," *IEEE Software,* vol. 18, pp. 112-111, 2001.

[19] S. Kenefick, "Produce Better Software by Using a Layered Testing Strategy," *https://www.gartner.com/doc/2645817/produce-better-software-using-layered*, 2014, Last accessed: Feb. 2017.

[20] A. Reichert, "Testing APIs protects applications and reputations," *http://searchsoftwarequality.techtarget.com/tip/Testing-APIs-protects-applications-and-reputations*, 2015, Last accessed: Feb. 2017.

[21] V. Garousi, M. Felderer, and M. V. Mäntylä, "The need for multivocal literature reviews in software engineering: complementing systematic literature reviews with grey literature," in *International Conference on Evaluation and Assessment in Software Engineering (EASE)*, 2016, pp. 171-176.