Software Test Automation with Robot Framework

Joshua T. Walker
Georgia Tech Research Institute (GTRI)
Georgia Institute of Technology
Atlanta, Georgia, USA

ABSTRACT
Automated testing allows for releases to be quickly verified and regression tested multiple times over a product's development period, providing a level of quality assurance unmatched by a purely manual process. This article discusses the driving objectives, implementation strategy, and general overview of a test automation framework created using the open source project Robot Framework as its test automation core. The framework described in this article has been used to provide test automation capabilities to both Graphical User Interface (GUI) based Windows applications and embedded systems. In addition to an overview of the developed framework, this article also provides the results of automated testing for multiple releases and a discussion of selected lessons learned from its creation and maintenance.

General Terms
Automated Testing, Software Testing, Test Automation, Test Engineering.

Keywords
Robot Framework, Test Automation Framework, Test Automation Strategy, Test Execution.

1. INTRODUCTION
Automated testing refers to the use of software and/or machinery to perform evaluation of a produced item without a human performing the actions necessary to stimulate the item [1]. Generally, automated testing is preplanned by a human using a scripting language or framework to set up scenarios with which to exercise the functionality covered by the requirements [2]. The tester uses their knowledge of the system to define inputs and expected outputs to be covered by the automation code [3]. The automation framework executes the test scenarios and provides detailed feedback as to the results of the testing [4]. It takes the procedures that would be performed by a human via manual means and executes them automatically through the use of specific equipment or applications, requiring the creation of test scripts to take the place of manual test procedures [5].

The incorporation of automated testing on a project can produce several benefits. Test documentation organization can improve due to an increase in maintainability of the test procedures and better management of test procedures [6]. Quality of the system under test may improve due to increased confidence of the product, reduction of human errors during test execution, and the ability to perform much more test executions than could be possible with a manual-only approach [7]. Finally, it provides opportunities by allowing new types of testing that were impossible to perform manually and it allows for more dedicated exhaustive testing of the most important components [8].

With the rise of iterative development methodologies coming out of the ideas presented by Rapid Application Development in the early 1990s and the Agile Manifesto of the early 2000s, automated testing has continually been a key component of rapid product development [9]. Essential to support an incremental development process, it allows for releases to be quickly verified and regression tested multiple times over the product's development period and can provide a level of quality assurance unmatched by a purely manual process [10].

This article discusses the driving objectives, implementation strategy, and general overview of a test automation framework created using the open source project Robot Framework as its test automation core. The framework described in this article has been used to provide test automation capabilities to both GUI-based Windows applications and embedded systems.

2. PATH TO TEST AUTOMATION
The decision to implement automated testing on a project can be multi-faceted, requiring a number of aspects to be considered. It may not always be the best testing approach and depends on a variety of factors such as project scope, development timelines, required test coverage, and cost [11].

For this project specifically, the main driver was the need to reduce test execution times significantly in order to support the transition to an Agile development process and to shorten the overall release schedule for the system. The system under test, an embedded airborne Electronic Warfare Management System (EWMS), could take up to 18 weeks to perform a full regression test manually. Its supporting Windows-based simulation, monitoring, and file creation tool suite could take up to an additional 20 weeks of manual testing for a full regression test. As updates to this system were usually released on a 6 to 12-month timeline, these regression test durations were prohibitive to the continued development of features and quick deployment to the field.

The path to test automation for this project was an arduous one. For the first several years, there was little support for its pursuit, due to the size of the system and the existence of a high-quality manual test suite. While the typical recurring budgets for the project were relatively high, the costs to start building automated testing infrastructure were still too high to justify. Taking time to officially build in this capability would prevent the development of necessary features that had to be released quickly.

Because of this, test automation was pursued as a side project targeting one of the system's file creation tools. This tool was a data entry application with a large number of configurable settings for the main system, resulting in a text-based file that would be loaded onto the embedded system for a specific scenario. The large amount of options in the tool resulted in its manual test procedures numbering approximately 2,200 pages, requiring six weeks of testing time to execute. This system was targeted first for automation because of the simplicity of its environment; it was a single software

application with no additional software or hardware interfaces. It could be executed in a virtual environment, making easy use of parallel testing and continuous integration tools [12].

The development of test automation capabilities for this project started initially as an experiment in 2013 with a team primarily composed of engineering student workers. This helped keep initial costs low while providing a worthwhile student project that created many opportunities for learning. This initial version, while somewhat successful, was not developed to be sustainable, leading to a largely unorganized codebase that was scrapped and rebuilt multiple times along the way. Over the next three years, the effort suffered a number of self-imposed issues from less than ideal coding practices and implementation strategies. Regardless, the automation project produced several significant test execution time savings and provided evidence for the value of a test automation capability.

Due to this evidence, the author secured internal funding in 2016 to start development of an official test automation framework. Using the knowledge gained through the various false starts over the previous several years, an initial design was developed that centered around the use of Robot Framework, along with an initial set of guiding objectives and an implementation strategy for the targeted system. This prework was necessary to build a proper foundation that would guide the creation of a sustainable automated testing capability.

2.1 Objectives

The initial set of objectives for the test automation framework is provided below.

1-Develop an automation capability that mimics the real-time actions of manual testing performed by a test engineer.

2-Provide a framework that can coordinate the actions of multiple machines.

3-Develop an automation capability that can control the actions of a machine while the display is locked.

4-Create a maintainable framework that can be easily updated for future versions of the software under test given normal time constraints.

5-Create a sustainable architecture that can be updated and expanded without affecting existing automation capabilities.

6-Provide an adaptable automation capability for future expansion into other in-house applications.

7-Promote understandability of test cases and automation libraries by utilizing standard structures, standard naming conventions, and natural language.

8-Minimize the need for significant coding experience for automated test case developers.

2.2 Strategy

The strategy employed when implementing test automation was to utilize as much existing functionality and components as possible in order to:

1-Minimize costs.

2-Limit the impact to the system under test.

3-Maintain the ability to execute existing manual testing.

4-Provide an easier transition to automated testing.

5-Provide automated testing capabilities faster.

These goals provided for the establishment of an automated test framework that supported the system under test without the need to change its development path or schedule. The approach focused on providing automation capabilities through the existing suite of support tools that were used to interact with the system. These tools were used manually through a GUI by the test team to provide simulation and monitoring of the system's execution environment. The initial approach was to create automation libraries that could control and monitor these tools via their respective GUIs, which would mimic the exact actions of a test engineer performing a manual test.

While this approach succeeded at providing test automation capabilities while adhering to the goals provided above, it was not the most impactful approach. However, other initial approaches would have been too costly or could have affected the development of the system. These observed inefficiencies combined with the difficulties of implementing test automation on a project as complex and established as the system under test necessitated a long-term automation strategy that would eventually result in an optimized test automation solution.

This led to the concept of a phased approach to test automation, which is described below.

Phase 0: No test automation implemented.

Phase 1: Develop GUI-based test automation capabilities for new system features being developed for the next release. Create test cases for new features in the new automated format.

Phase 2: Develop GUI-based test automation capabilities for existing manual test cases. Reassess existing test cases for adherence to requirements, applicability, length, complexity, etc. Convert manual test cases into the new automated format, making any necessary improvement steps based on the previous assessment.

Phase 3: Develop headless test automation capabilities for existing automated test cases, to remove the latency involved with commanding a GUI to perform an action. Convert existing GUI-based automated test cases into headless automated test cases.

Phase 4: Develop lower-level test harnesses that can talk directly to software components within the system, providing the ability to perform testing on each component in isolation from the rest of the system. Develop automation capabilities to stimulate and monitor lower-level components. Create headless test cases that target component-level functionality.

Phase 5: Introduce high-level commands that provide more capability to the tester (e.g., move threat 1 to location 5 within 20 minutes using route 2, maintain aircraft altitude and fly in a circle with radius of 10 miles), allowing more natural and realistic test scenarios in a lab environment.

It should be noted that these phases are not necessarily completed sequentially; some could be worked concurrently, while some could be skipped altogether. For example, currently for this system, a mixture of Phase 0 through Phase 4 has been implemented, depending on the component. This specific system may never need test automation capability at Phase 5, but it has been discussed as a future option.

3. FRAMEWORK OVERVIEW

The following section will provide an overview of the test automation framework that was created. First, a discussion of Robot Framework is necessary to understand the core automation capability. Next, the basic architecture of the framework is described. Finally, an example test case is provided with commentary to show a test case from the end user's perspective.

3.1 Robot Framework

Robot Framework is a Python-based automation project that provides the infrastructure for developing a test automation capability, including the translation of code to user-defined actions, automatic test execution, and results reporting [13]. It allows automated tests to be written in simple English to support their easy creation and understanding. The basic idea of Robot Framework is to define libraries that include functions for performing necessary actions required by the test case and then call them from the high-level test case using a simple English keyword or phrase.

3.1.1 Example 1

To illustrate how Robot Framework works, a simple test case named "notepad_save_file.robot" is defined in Figure 1 below.

```
Library    notepad.py

Test 1 – Save File
        Open Notepad
        Add Text            my added text
        Save File As        myNote.txt
        Close Notepad
        Open Notepad
        Open File           myNote.txt
        Verify Text         my added text
```

Fig 1: Simple Robot Framework Example

The test case above verifies Notepad's ability to save a file with a specific line of text. Items in blue are referred to as keywords, while items in red are referred to as keyword inputs. The code to execute the keywords is stored in the library "notepad.py," which is called at the beginning of the test case. As can be seen above, the power of Robot Framework comes with being able to define the functionality of a keyword once and use it multiple times within a test case [14]. This is even more apparent with keywords such as "Add Text" in the above example, as the keyword input can be any string, depending on the restrictions of the implemented function.

To execute the test case, a tester would only need to run the command "robot notepad_save_file.robot" from a command prompt, assuming Robot Framework is installed and the user has navigated to the correct directory. Robot Framework would the output the results of the test case within the same directory in a variety of formats for later review.

3.1.2 Example 2

Keyword text can also be expanded and modified to give a more natural feel to the automation language within test cases. For example, consider the following keyword in Figure 2.

```
Select the [Miles] option in the [Units] combobox within the
[Distance Converter] window
```

Fig 2: Keyword Usage

The entire line in Figure 2 is considered a keyword with three keyword inputs. For ease of comprehension, keyword inputs are now required to be surrounded by square brackets. This is customizable in the definition of the keyword, which can be seen below in Figure 3.

```
Select the [${option}] option in the [${control}] combobox
within the [${window}] window
    Select Combobox Item  ${window} ${control} {$option}
```

Fig 3: Keyword Definition

The keyword refers to the "Select Combobox Item" method defined elsewhere, which takes three inputs. Robot Framework is able to interface with almost any language or automation tool to provide more powerful automation capabilities. In this case, the Windows UI automation language AutoIt, translated to a Python module PyAutoIt, was used to provide UI interaction capabilities. The code to perform this action of selecting an option within a combobox can be seen in Figure 4 below.

```
def select_combobox_item(self, json_title,
control_name, item_to_be_found):
    window_title  =
self.json_parser.get_window_title(json_title)
    control_id =
self.json_parser.get_control_id_with_type(json_title,
control_name, 'combobox')
    previously_selected_item = ""
    while (True):
        currently_selected_item =
baseline.autoit_control_get_text(self, window_title,
control_id)
        print(currently_selected_item)
        if currently_selected_item) = = item_to_be_found:
            return True
        elif currently_selected_item = =
previously_selected_item:
            assert False, "Selection [" + item_to_be_found
+ "] could not be found in the [" + control_name  + "]
combobox"
        else:
            previously_selected_item =
baseline.autoit_control_get_text(self, window_title,
control_id)
            baseline.autoit_control_send(self, window_title,
control_id, "{DOWN}")
```

Fig 4: Keyword Function

The method above accepts the parameters passed from the keyword usage through the keyword definition to the final method. It uses the passed values to identify the actual names associated with those values as stored in the code by looking them up in a JavaScript Object Notation (JSON) file that associates aliased labels (i.e., the word/phrase that the test engineer wants the specific item to be known by) with the

code-based name (i.e., the name that the developer used for the specific object). An example of a JSON table to perform this aliasing can be seen in Figure 5 below.

```
"Units": {
        "type": "combobox",
        "control": "[NAME:_UnitCombo]"
},
```

Fig 5: JSON Combobox Label Association

The main purpose of the JSON file is to:

1-Maintain a list of supported objects outside of the code to interact with them.

2-Allow aliasing so that the final test procedures are not reliant on standardized control names to make sense to an external representative.

3-Define an object type to use in the event of a collision of preferred name (e.g., a textbox and combobox that use the same label property).

3.2 Basic Architecture

A basic architectural view of the test automation framework can be seen in Figure 6 below.

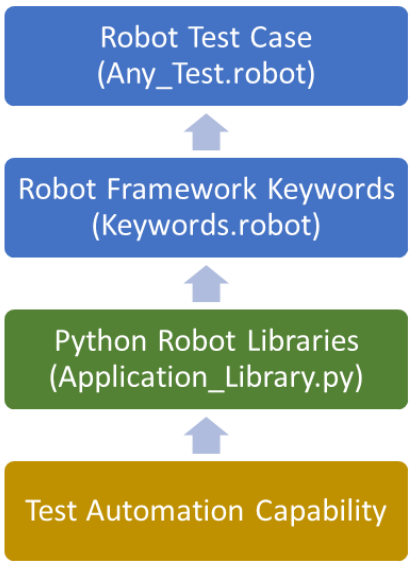

Fig 6: Basic Architecture

Starting at the bottom of the diagram, to create a Test Automation Capability targeting a specific application or system, Robot Libraries are written to interface with the system under test. These libraries contain functions that define methods for interacting with the system. In the Robot Framework Keywords file, the methods from the Robot Libraries are mapped to commands that will be visible to the test engineer to use for building test cases in the Robot Test Case file. This essentially means that an automated test case (a .robot file) is a sequence of pre-defined keywords written in natural language that reads like a set of manual test steps. This allows the test case to be executed manually for debugging purposes or if for some reason the automated capability breaks, and it allows the automated test case to be easily translated to a more presentable form for delivery to a customer.

The test automation framework is fundamentally a collection of custom developed automation libraries that control interaction with the system, keyword definitions that provide the standardized interaction language used by test engineers, and Robot Framework that provides the management of those connected pieces. Additionally, several usability features such as syntax highlighting, automatic keyword completion have been incorporated to make the creation of test cases faster and more user-friendly.

3.3 Example Test Case

Figure 7 is an example automated test case, shown as a screenshot from Visual Studio Code using the custom highlighting scheme developed for the test automation framework. Specific sections of the test case are described here to provide an overview of the organization and flow of a typical test case.

3.3.1 Settings Section (Lines 1-5)

The Settings section is used to declare keyword libraries that are expected to be used in this test case. The test automation framework architecture encapsulates keyword libraries by function or component for easier management. In this section, the test engineer imports the libraries that include keywords they need to perform the actions of the test case.

3.3.2 Variables Section (Lines 7-10)

The Variables section provides the ability to declare groups to be used later in the test case. Currently, the only use of this is to group requirements traced to this test case and aircraft platforms that are applicable for this test case. These labels are used to make it easier to identify these items quickly.

3.3.3 Test Cases Section (Lines 12-51)

The Test Cases section contains the body of the test procedure that will be executed. Subsections are further discussed below.

3.3.3.1 Title (Line 13)

The title of the test case is declared after the Test Cases section header. In this example, a global test case number, its specific name, and the platform this test case uses (Plat_1) during the execution of the test case are listed.

3.3.3.2 Tags (Line 14)

Tags are a feature of Robot Framework that allow test cases to be queried/executed based on custom phrases that are included in the test case. For example, a test engineer could target the entire test suite and command all test cases that include the tag "Requirement-01" to be executed. Robot Framework is able to identify which test cases those are and execute them.

In this specific example, the variables previously defined are also tagged so that a query can be run based on requirements, functionality, or platform applicability.

3.3.3.3 Comments (Lines 15; 19)

Comments are allowed with the framework in order to provide explanatory text for a specific section of test steps. Comments are not executable and are ignored when running a test case.

```
1   *** Settings ***
2   Library   Dialogs
3   Library   C:/Test_Automation_Framework/system_controller/system_controller
4   Library   C:/Test_Automation_Framework/sys_mon/sys_mon
5   Library   C:/Test_Automation_Framework/sys_sim/sys_sim
6
7   *** Variables ***
8   @{REQUIREMENTS}     SS-320_1     SS-320_2     SS-320_3
9
10  @{APPLICABILITY}    Plat_1      Plat_2      Plat_3
11
12  *** Test Cases ***
13  [SS_0010] Scenario Selection Plat_1
14  [tags]      @{REQUIREMENTS}     @{APPLICABILITY}    System      MD4     Scenario Selection
15  #System Simulator Setup
16  Open the System Simulator application and open the [Plat_1 No Errors.sim] Simulator file
17  Press the Start button within the Control Panel window
18
19  #System Monitor Setup
20  Open the System Monitor application and open the [Standard Messages.mon] Monitor file
21  Load the [Common Release (Plat_1 with Config_2)] protocol files
22  Press the Play button within the System Monitor window
23
24  Set up the test case with the [General Config Plat_1] configuration
25
26  Navigate to the [Training] menu
27  Press the [Down] button on the Controller
28  Press the key under [TRN] on the Controller
29
30  This begins the verification for requirement: SS-320_1
31  Verify the [bottom] line of the system display contains [SCN1]
32  Verify the [bottom] line of the system display contains [SCN2]
33  Verify the [bottom] line of the system display contains [SCN3]
34  This completes the verification for requirement: SS-320_1
35
36  Press the key under [SCN1] on the Controller
37  Wait for [SCN:SCENARIO1] to appear on the system display
38
39  This begins the verification for requirement: SS-320_2
40  Verify the [top] line of the system display contains [SCN:SCENARIO1]
41  This completes the verification for requirement: SS-320_2
42
43  Press the key under [OFF] on the Controller
44  Wait for [SCN:OFF] to appear on the system display
45
46  This begins the verification for requirement: SS-320_3
47  Verify the [top] line of the system display contains [SCN:OFF]
48  This completes the verification for requirement: SS-320_3
49
50  Press the Step button within the Control Panel window and verify the new sequence is [Test End.seq]
51  [TEARDOWN] Tear down the test case and log recorded errors
```

Fig 7: Example Test Case

3.3.3.4 Setup (Line 24)

This line is a custom keyword that sets up a test case for the system under test after being provided a configuration. There are many actions rolled into this singular keyword that prepare the system under test for user input, such as loading the system hardware, verifying that configuration loaded correctly, and clearing the catalog of errors. Configurations are defined by test engineers in a separate file that the test automation framework can read. This keyword also prints the details of the specific configuration used for this execution of the test case for future reference.

3.3.3.5 Actions (Lines 16-17, 20-22, 26-28, 36-37, 43-44, 50)

These lines are typical actions that would be defined in a manual test case as steps that a user should perform.

3.3.3.6 Requirements Declarations (Lines 30, 34, 39, 41, 46, 48)

While requirements are traced to the test case using the Tags section, more specificity is often required by the customer or project. The test automation framework provides a method for defining the specific steps that satisfy requirements through defining where the verification of the requirement begins and ends.

3.3.3.7 Expected Results (Lines 31-33, 40, 47)

Expected results are defined using a keyword that begins with the word, “Verify.” This was purposely defined when building the test automation framework so that expected results would be evident throughout the test case. Syntax highlighting is also used to draw attention to these lines.

3.3.3.8 Teardown (Line 51)

The teardown keyword is used here to prepare the system hardware for the execution of the next test case. Errors that occurred are logged in the test results and the environment is cleaned to reduce any variability that may affect the results of future test cases.

4. RESULTS

The test automation framework described herein has been used successfully during the formal test execution of the targeted system for two releases, as well as two releases of the data entry support tool described previously. The results of the automation effort for these releases are summarized in Table 1 below and further described in the following subsections.

Table 1. Test Automation Results

Release	Percent Automated	Time Saved	Money Saved
Tool Release 1	95%	6 weeks	$60K
Tool Release 2	95%	7 weeks	$70K
System Release 1	23%	3 weeks	$15K
System Release 2	38%	6 weeks	$30K

4.1 Data Entry Tool Release 1

The data entry tool was the first application targeted by the test automation framework, due to its limited interactions with other components. During the first release, 95% of the manual test procedures were automated, equating to more than 2,200 pages and six weeks of execution time. The entire suite of manual and automated test cases for the final release was able to be executed in two days. Typically, the manual test procedures were run twice during a release effort, once for a dry run and once for a formal test. Factoring that time in, the test automation framework saved approximately $60,000 of customer funds due to test execution costs. This initial use of the test automation framework significantly reduced the overall cost of developing the tool.

4.2 Data Entry Tool Release 2

The second release of the data entry tool was similar to the first, in that 95% of the test procedures remained automated even with additional functionality being added. With the additions, running the testing manually would have taken approximately seven weeks, but the entire test was still able to be accomplished within two days. The approximate cost savings for this effort equated to roughly $70,000, increasing the value of the test automation framework on this effort.

4.3 System Release 1

As the embedded system is significantly more complicated than the data entry tool, the transition to automated testing is still ongoing, even after two years of a functional test automation solution. This is partly because feature development of the system was not stopped to implement this transition and partly due to the reworking of test cases to better support automation. Due to the phased approach limiting impact to the system under test, the transition was less impactful than it could have been or will be in the future.

During the first year of the test automation effort, the development team created automation infrastructure to support approximately 84% of the functionality of the system under test. The test team was able to create 76 automated test cases, which equated to approximately 23% of the entire expected test suite. This percentage of completed testing saved approximately three weeks of manual execution time, equating to a cost savings of approximately $15,000 per test run. While this initial cost savings was smaller than on the data entry tool, it was still a major step towards increasing the efficiency of the test effort for the system.

4.4 System Release 2

During the second year of the test automation effort, the development team increased the automation infrastructure to support approximately 90% of the functionality of the system under test. The test team was able to create 166 test cases, which equated to approximately 38% of the entire expected test suite. This percentage of completed testing saved approximately six weeks of manual execution time, equating to a cost savings of approximately $30,000 per test run. During development of this release, the savings due to test automation was doubled in one year.

4.5 Qualitative Results

Several qualitative results were discovered during this process. A selection of key discussion topics is provided below.

4.5.1 Team Morale

The field of software engineering can be subject to a high level of burnout, due to a variety of factors [15]. For a test engineer, running the same set of regression tests manually over and over again can become tedious, especially if that effort takes significant time in comparison to other, more creative or skill-intensive tasks. Test automation can reduce this burnout, as it removes some of the monotonous aspects of the job of a test engineer. In place of the time usually spent on regression testing, it may provide the potential for more exploratory testing to be performed, which has been found to identify more defects that scripted testing [16].

4.5.2 Test Quality

The quality of the test procedures increased, due to improved flow and organization, increased maintainability, and reduced variability between approaches. When the actions that can be performed are controlled through the use of automated keywords, the test cases become more standardized, leading to a greater level of understanding. As with the development of software, different test engineers may take different approaches while still solving the problem, but the variations of how this can be accomplished are easier to comprehend.

5. CONCLUSION

The creation of an automated testing framework can be a valuable, but also costly, venture that should be carefully considered before pursuing. It can provide considerable reductions in test execution times, but the benefits must be weighed against the costs involved. The implementation of test automation described in this article provided significant time and cost savings, allowing more efficient use of customer funds.

5.1 Selected Lessons Learned

Several lessons were learned throughout the creation of this test automation framework. A selection of key topics is described below.

5.1.1 Treat Test Automation as a Product

Even if it is never intended to be delivered to an external customer, the developed automation code and the accompanying test procedures that use it should be treated

like a real, releasable product. It should be designed and documented well, conforming to standard practices and procedures. This leads to a higher level of maintainability, which is a common issue with the longevity of test automation infrastructure [17]. Treating the test procedures as code also allows for the use of development tools, which can be an exceptional aid to productivity.

5.1.2 Start Small and Advertise Results

Management, customers, and other stakeholders that control project funding can be hard to convince that test automation is worth the effort, especially if a set of manual procedures already exist that work as expected. If it truly makes sense for the project and would provide value, find a way to incorporate automation somewhere along the process in a small way as an example of future potential [18]. Use that as evidence for the value that can be created by transitioning to an automated world.

5.1.3 Automate Functionality That Makes Sense

Not everything should be automated; sometimes test cases are too difficult or just too small to make sense [19]. Sometimes an automated approach does not always result in the most efficient method of testing [20]. Discretion and experience should be used to identify when functionality should be tested automatically.

5.1.4 Standardize the Approach

Put significant effort into the automation language being created. Standardizing the approach to the keywords that will be visible to the end users is an important part of providing value. If a test engineer trying to use the test automation framework has trouble identifying what functionality is available, it can impact productivity and result in confusion.

5.2 Future Expectations

Based on the last two years of results so far and by implementing future automation phases, it is expected that within two to four more release cycles, the execution of the test procedures for the system will be reduced to less than one week. This type of reduction, from an original timeline of 18 weeks, shows the potential improvement that developing an automated testing capability can provide.

6. REFERENCES

[1] Engel, A. (2010). *Verification, validation and testing of engineered systems*. John Wiley & Sons.

[2] Taipale, O., Kasurinen, J., Karhu, K., & Smolander, K. (2011). Trade-off between automated and manual software testing. *International Journal of System Assurance Engineering and Management*, *2*(2), 114-125. https://doi.org/10.1007/s13198-011-0065-6

[3] Fewster, M., & Graham, D. (1999). *Software test automation*. Addison-Wesley.

[4] Lewis, W. E. (2017). *Software testing and continuous quality improvement*. CRC press.

[5] Pan, J. (1999). Software testing. *Dependable embedded systems*, 5.

[6] Rankin, C. (2002). The software testing automation framework. *IBM Systems Journal*, *41*(1), 126-139. https://doi.org/10.1147/sj.411.0126

[7] Methong, S. (2012). Model-based automated GUI testing for Android web application frameworks. *2nd International Conference on Biotechnology and Environment Management, Singapore*, 106-110.

[8] Kasurinen, J., Taipale, O., & Smolander, K. (2010). Software test automation in practice: empirical observations. *Advances in Software Engineering*, *2010*.

[9] Dustin, E., Rashka, J., & Paul, J. (1999). *Automated software testing: Introduction, management, and performance*. Addison-Wesley Professional.

[10] Dustin, E., Garrett, T., & Gauf, B. (2009). *Implementing automated software testing: How to save time and lower costs while raising quality*. Pearson Education.

[11] Ramler, R., & Wolfmaier, K. (2006). Economic perspectives in test automation: Balancing automated and manual testing with opportunity cost. *Proceedings of the 2006 international workshop on automation of software test*, 85-91. https://doi.org/10.1145/1138929.1138946

[12] Dösinger, S., Mordinyi, R., & Biffl, S. (2012). Communicating continuous integration servers for increasing effectiveness of automated testing. *2012 Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, *Essen*, 374-377. https://doi.org/10.1145/2351676.2351751

[13] Bisht, S. (2013). *Robot framework test automation*. Packt Publishing Ltd.

[14] Stresnjak, S., & Hocenski, Z. (2011). Usage of robot framework in automation of functional test regression. *Proceedings of the 6th International Conference on Software Engineering Advances*, 30-34.

[15] Sonnentag, S., Brodbeck, F. C., Heinbokel, T., & Stolte, W. (1994). Stressor- burnout relationship in software development teams. *Journal of occupational and organizational psychology*, *67*(4), 327-341. https://doi.org/10.1111/j.2044-8325.1994.tb00571.x

[16] Shah, S. M. A., Torchiano, M., Vetrò, A., & Morisio, M. (2013). Exploratory testing as a source of technical debt. *IT Professional*, *16*(3), 44-51. https://doi.org/10.1109/MITP.2013.21

[17] Martin, D., Rooksby, J., Rouncefield, M., & Sommerville, I. (2007). Good organisational reasons for bad software testing: An ethnographic study of testing in a small software company. *29th International Conference on Software Engineering*, 602-611. https://doi.org/10.1109/ICSE.2007.1

[18] Berner, S., Weber, R., & Keller, R. K. (2005). Observations and lessons learned from automated testing. *Proceedings of the 27th International Conference on Software Engineering*, 571-579. https://doi.org/10.1145/1062455.1062556

[19] Sabev, P., & Grigorova, K. (2015). Manual to automated testing: An effort-based approach for determining the priority of software test automation. *International Journal of Computer, Electrical, Automation, Control and Information Engineering*, *9*(12), 2456-2462.

[20] Karhu, K., Repo, T., Taipale, O., & Smolander, K. (2009). Empirical observations on software testing automation. *2009 International Conference on Software Testing Verification and Validation* 201-209. https://doi.org/10.1109/ICST.2009.16