Deploying TESTAR to Enable Remote Testing in an Industrial CI Pipeline: A Case-Based Evaluation

Fernando Pastor Ricós[1(✉)], Pekka Aho[2(✉)], Tanja Vos[1,2(✉)], Ismael Torres Boigues[1,2,3(✉)], Ernesto Calás Blasco[1,2,3], and Héctor Martínez Martínez[3]

[1] Universitat Politècnica de València, 46002 Valencia, Spain
ferpasri@inf.upv.es
[2] Open Universiteit, Heerlen, The Netherlands
{pekka.aho,tanja.vos}@ou.nl
[3] Prodevelop, Valencia, Spain
{itorres,info}@prodevelop.es

Abstract. Companies are facing constant pressure towards shorter release cycles while still maintaining a high level of quality. Agile development, continuous integration and testing are commonly used quality assurance techniques applied in industry. Increasing the level of test automation is a key ingredient to address the short release cycles. Testing at the graphical user interface (GUI) level is challenging to automate, and therefore many companies still do this manually. To help find solutions for better GUI test automation, academics are researching scriptless GUI testing to complement the script-based approach. In order to better match industrial problems with academic results, more academia-industry collaborations for case-based evaluations are needed. This paper describes such an initiative to improve, transfer and integrate an academic scriptless GUI testing tool TESTAR into the CI pipeline of a Spanish company Prodevelop. The paper describes the steps taken, the outcome, the challenges, and some lessons learned for successful industry-academia collaboration.

Keywords: Automated testing · GUI level · TESTAR · CI · Technology transfer

1 Introduction

The development of cost-effective and high-quality software systems is getting more and more challenging for SMEs. Modern systems are distributed and become larger and more complex, as they connect multitude of components that interact in many different ways and have constantly changing and different types of requirements. Adequately testing these systems cannot be faced alone with traditional testing approaches.

T. Margaria and B. Steffen (Eds.): ISoLA 2020, LNCS 12476, pp. 543–557, 2020.
https://doi.org/10.1007/978-3-030-61362-4_31

New techniques for systematization and automation of testing are being researched in academia. To help the industry to keep up with the increasing quality requirements, it is important to guarantee the successful transfer of new techniques into use.

Unit tests are widely automated, especially if test-driven development process is followed. However, testing through graphical user interface (GUI) is more challenging to automate [1]. The most common way to automate GUI testing is based on scripts that are defined before the test execution. Manually recording or writing test scripts for all the possible paths of the GUI takes simply too much effort to be practical, and even if the test cases are built with keywords and a proper architecture, so many test scripts would result in serious maintenance issues [6]. To address this challenge, the academics are researching scriptless GUI testing to complement the script-based approach. In scriptless GUI testing, the test cases are generated during the test execution, based on observing the run-time state of the system under test (SUT).

The rest of this paper is structured as follows. First, in Sect. 2, we describe the context of this study. In Sect. 2.1, we describe TESTAR, an open source scriptless test automation tool developed in academia. In Sect. 2.2, we describe a Spanish company Prodevelop, their software product Posidonia that is used as the system under test (SUT) in this collaboration, and their continuous integration (CI) process. In Sect. 3, we describe the goals and the objectives to consider that the transfer of knowledge has been achieved. In Sect. 4, we describe the development improvements made into TESTAR in terms of functionality belonging to the tool. We discuss the results in Sect. 5 and summarize the lessons learnt about academia-industry collaboration in Sect. 6. Finally, we conclude in Sect. 7.

2 Context

The work described in this paper has been carried out within the context of the European ITEA3 TESTOMAT project[1]. Both the private company Prodevelop and the academic partners are funded through this project.

2.1 The TESTAR Tool

TESTAR[2] [14] is an academic open source tool for automated testing through the GUI currently being developed by the Polytechnic University of Valencia and the Open University of the Netherlands, funded by various national and European initiatives.

TESTAR is a tool for *scriptless* testing, meaning that it does not require the creation, use and maintenance of scripts to test and explore the SUT from the user's perspective. It is open source under BSD3 license and available on Github[3].

[1] https://www.testomatproject.eu/.
[2] https://testar.org/.
[3] https://github.com/TESTARtool/TESTAR_dev.

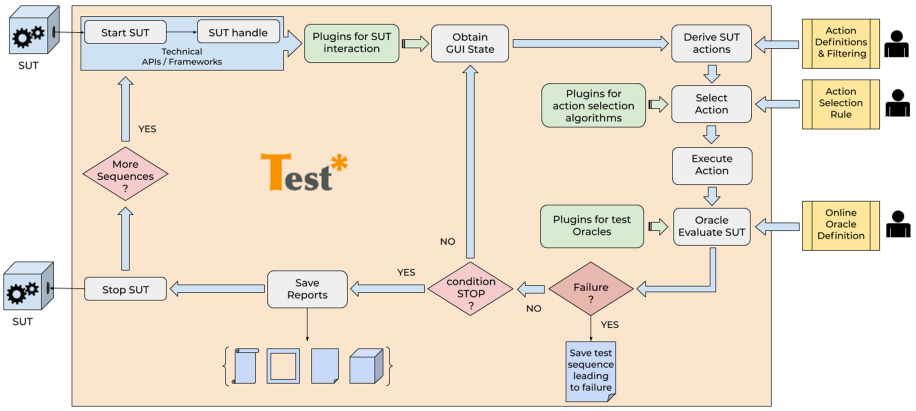

Fig. 1. TESTAR functional flow

The underlying principle of this testing approach is as follows (see Fig. 1): generate test sequences of (state, action)-pairs by starting up the SUT in its initial state and continuously select an action to bring the SUT in another state. The action selection characterizes the most basic problem of intelligent systems: *what to do next*. The difficult part is optimizing the action selection [ázar2018] to find faults, and recognizing a faulty state when it is found.

The default action selection of TESTAR focuses on random exploration of the SUT through processing of the state information extracted before and after each executed action. This way TESTAR can analyze the robustness of the SUT in a generic way and automate the testing of what we call *Non-User Stories*, detecting failures by implicit test oracles that check the violation of general-purpose system requirements, such as:

- the SUT should not *crash*,
- the SUT should not find itself in an *unresponsive* state (freeze), and
- the UI state should not contain any widget with *suspicious titles* like *error*, *problem*, *exception*, etc.

Implementing support for various technical APIs enables TESTAR to interact with different kinds of SUTs (desktop as well as Web). The modular architecture of TESTAR allows customizing and enriching the system specific protocol, for example, changing the action selection algorithms to take different exploratory paths, or defining system specific inputs or test oracles.

In addition to the SUT-specific protocol defining the behavior of TESTAR tool in terms of widgets interaction, actions exploration and test oracles, another set of configuration parameters is required to indicate how to connect to the desired SUT, define suspicious title patterns for SUT-specific test oracles or change between different protocols if these are customized to explore different parts of the SUT.

The configuration options for TESTAR are by default read from a local file, which allows to read and write the desired protocol implementation to adapt the functionality with the SUT requirements. For beginners and learning purposes, or facilitate the first SUT inspection and configuration of TESTAR, a GUI is offered to highlight the visibility of the most important configuration options. When changed, the GUI overwrites the local file with the new configuration.

As TESTAR obtains the information from the SUT about existing widgets, states, and available actions to execute, it selects and executes these actions generating the TESTAR test sequences. All the information obtained is stored in different formats and types of files creating output results for every sequence. After each executed action, TESTAR applies all the implicit and defined test oracles to obtain a verdict to determine whether the latest state of the sequence contains failures.

Every sequence creates the following types of files:

- Logs including step-by-step textual information about the executed actions, target widgets, and the verdicts from test oracles.
- Screenshots of each state and target widget on which an action is going to be executed, taken along the test sequence.
- HTML reports of each generated test sequence, including step-by-step screenshots and textual information about existing widgets and available actions of every state, and the executed action over the target widget.
- Binary files, used for saving the information about the executed actions in a form that allows a sequence to be replayed later.

2.2 Prodevelop

Prodevelop[4] is a Spanish company located in Valencia with an extensive network of clients in Europe, Africa, America and Oceania. From the beginning, Prodevelop has specialized on Geographic Information Systems and its application to the maritime transportation, especially in port domain.

The SUT used in this study is Posidonia Management, a web-based port management application developed and maintained by Prodevelop. Posidonia Management is conceived and designed to fulfil the management needs of different Port Authorities. The increasing port traffic and high competitiveness of the international market lead to increasingly complex systems. It is in this context that Posidonia Management, as a complete management system, can improve the efficiency, productivity, and competitiveness of a Port Authority.

Until a few years ago, Prodevelop followed the waterfall development cycle, but in the last few years, encouraged by the TESTOMAT Project, Prodevelop has oriented its development practices towards a more agile development cycle, with more frequent product deliveries, weekly in some products.

Continuous integration [8,10] (CI) is a process that focuses on increasing the client value through developing, updating, building and testing the software product as often as possible, for example after each code commit or once a day.

[4] https://www.prodevelop.es.

The continuous integration process of Prodevelop is made up of a series of linked and interrelated steps, illustrated in Fig. 2. The process begins when the Quality Assurance (QA) team configure the automatization orchestrator server Jenkins[5], a free and open source automation server that can be used to build, test and deploy software, facilitating continuous integration.

In parallel, the Business Analyst will gather the project requirements and analyse them to obtain the specification of the system. Based on this specification, on one hand, the testers will use TestLink[6] to define the Acceptance/Functional test, and on the other hand, the Developers will develop the system and create the Unit Tests. These tests will be evaluated by the task of Jenkins that performs the build of the deliverables.

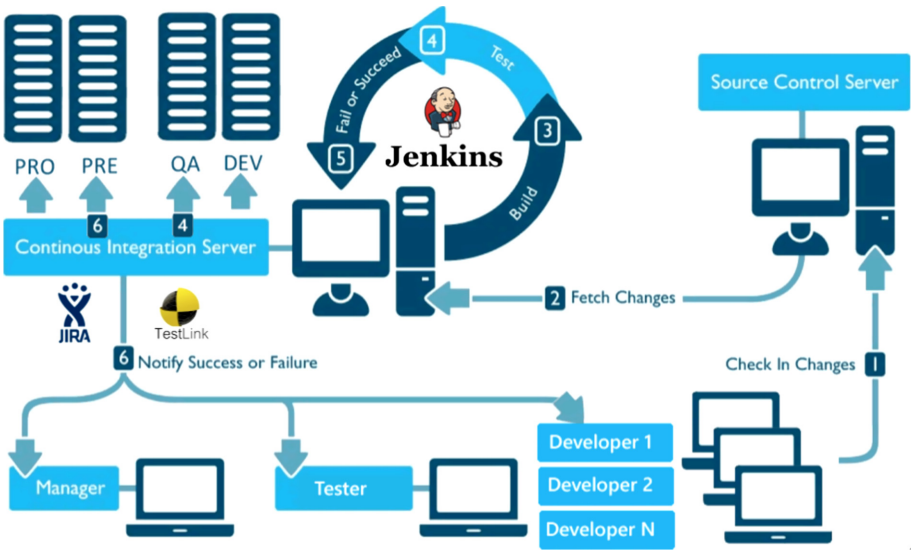

Fig. 2. Prodevelop CI/CD Pipeline

This automated process starts each time the Developers make a commit of source code to the repository. When a project that is assigned to a continuous integration environment receives an update of the source code, the Jenkins application will execute software code testing tasks: Static analysis, build and unit testing, to validate and compile the new source code.

If the build tasks in Jenkins end with the result "OK", the new version of the application will be deployed in the Quality Assurance (QA) environment, and the acceptance/functional tests are executed manually. If the tests are passed, the application will be deployed in the User Acceptance Test (PRE) and/or

[5] https://www.jenkins.io.
[6] http://testlink.org/.

Production (PRO) environment, which are located in the Client's own environments. The number of environments and deployment procedures are subject to the specific requirements of the Client.

In the case that any of the tasks that should be executed in Jenkins ends with "NOK" results, Jenkins informs the Developers detailing which test or tests have failed. In addition, Jenkins will generate an Incident-Ticket in Jira[7], an issue tracking and project management software, with all the necessary information, including also the phase of the process and specifically the test that fail, so that the Project Manager follows-up until the incident is resolved.

To ensure the quality of the software, Prodevelop relies mainly on functional testing. The QA staff assigned to a project defines functional test cases for each requirement and scenario using TestLink tool. These test cases are manually executed by the QA team when a new release is ready. A report with the results is generated and sent to the project manager to decide actions to be taken. On the other hand, developers are in charge of defining unit and static tests that are executed automatically.

The manual execution of functional tests is very time consuming as they have to be executed on each new delivery. The automation of these tests is one of the short-term objectives of Prodevelop. Another important issue to be improved is the time needed to solve an error. Since Posidonia is a large product with several million lines of code, and with several developers involved throughout the life of this product, a lot of time is spent looking for the origin of the problem.

To facilitate error detection and root cause analysis, the Posidonia Management application is instrumented with the intention to detect and debug all behaviour that is identified as an exception. All these exceptions are written to a log file with the information about the method where it occurred (the specific class and package it belongs to) and details about the exception that has been detected. This internal error information is added incrementally in the background log using local timestamps.

3 Objectives of the Study

From the academic point of view, the main goal of this collaboration was to evaluate the academic TESTAR tool on another real case in an industrial testing environment. TESTAR has already been evaluated in other industrial environments [2–4,7,9], and in order to be able to generalize these results based on individual cases [15] we need to study as many cases as we can and focus on their similarities. All the case studies so far shared one common aspect: before the introduction of TESTAR, GUI testing was done manually. For these studies we could see that TESTAR was considered a useful complement to the existing testing practices and interesting failures were found.

From the industrial point of view, Prodevelop is trying to achieve a high level of software quality by innovating its development processes. As indicated, the functional tests that are executed manually involve a high cost of running the

[7] https://www.atlassian.com/es/software/jira.

tests. For this reason, only a subset of them is executed in each release. So the objective of the study is clear: *integrate TESTAR into the current CI pipeline to automatically test Posidonia when the life cycle requires it and evaluate the performance.*

With TESTAR integrated into the CI pipeline, every time a new version is released and a nightly build is made, the following steps are taken:

- First, it will be checked whether there are failure sequences from previous versions, and in that case *replay* TESTAR test sequences to verify that errors were solved in the new release.
- Second, new test sequences will be *generated* with TESTAR to explore and verify the robustness of the application using the desired oracles and protocols. Depending on the configuration used, TESTAR can be steered to explore specific parts of Posidonia.
- Third, if a failure is detected, Prodevelop must verify that it is not a false positive, inspecting the sequence that found the failure. If it is not, all the logs generated during the test run should be filtered by the timestamps of the failure finding sequence, saved in a database and documented in TestLink. Then, a Jira ticket will be created with linked information about these results to be reviewed in the future.

To start the integration, Posidonia was tested with the default set-up of TESTAR to generate: test sequences, TESTAR logs, HTML test reports and GUI screenshots. All these artefacts generated by TESTAR were analyzed by Prodevelop. It was found that before the integration into CI could be realized, the following TESTAR extensions and improvements had to be implemented first:

1. Enable invocation of TESTAR through the CLI (Command Line Interface). This means that the configuration dialog should be disabled, and, instead of passing the test settings in a local file, they should be passed as parameters of the CLI command.
2. Enable TESTAR to correctly detect SUTs that have multiple processes handling the GUI, or that the GUI process change at run-time. Posidonia runs in a browser that starts with two main processes to which we should connect to properly verify the defined oracles.
3. Enable distributed execution of TESTAR by providing a remote API. This feature is fundamental if we want to integrate TESTAR into the CI methodology, or any other distributed process for that matter.
4. Improve the functionality of TESTAR Replay mode to observe changes between a previously executed and saved sequence and a newly executed test sequence.
5. Enable the synchronization of the logs produced by Posidonia with those of TESTAR. In order to find the root cause of the errors, it is important to be able to analyse the logs generated by Posidonia together with TESTAR logs. This information is needed by Prodevelop developers to understand and replicate the error.

These TESTAR adaptations will be described in the next section.

4 Extending TESTAR for the Case Study

This section describes the changes that had to be implemented into TESTAR to meet the requirements of Prodevelop and to be able to test Posidonia with TESTAR in the CI pipeline of Prodevelop.

4.1 Executing and Configuring TESTAR Through CLI

To allow TESTAR tool to be integrated into a CI pipeline, a new configuration option was added in addition to local settings files. When starting TESTAR through a CLI, the configuration can be passed on as parameters. This way any configuration setting can be overwritten through CLI, also disabling the GUI. This feature makes it easier to put TESTAR configuration into the settings of the CI job that starts TESTAR execution and change it from the CI tool.

4.2 Supporting SUTs with Multiple GUI Processes

By default, the execution of a SUT is started up by TESTAR using the path that contains the executable file, or in the case of web applications, by indicating the browser executable with the desired web URL. In case of running the SUT on Windows, first, we use this path to invoke a Windows function that will return the *process* handle of the SUT process that allow us to obtain the identifier of the SUT process, *pid*. However, to obtain the GUI state information (i.e., the widget tree and all the widget properties) through the Windows Accessibility API plugin, we need the *window handle*. To find the corresponding window handle, we probe all the existing window handles that are children of the Windows Desktop, to find the one that has the same *pid* as our SUT process.

The SUT in this case study, Posidonia, does not run in a single process. Instead, it starts execution with two GUI related processes. Some elements of one of these processes use warning pop-ups or lists of items. This prevented TESTAR from recognizing all the widgets. Therefore, TESTAR had to be changed to deal with SUTs that start with multiple GUI processes or launch new GUI handling processes at run-time. When a SUT starts up multiple processes, we do not have one main *pid*, but we have a list of *pid*s (i.e., including the child *pid*s of this main *pid*). In such cases we need to iterate over all the elements in the list to be able to get the GUI properties and information for each pid and merge them into one widget tree.

Supporting multiple GUI processes improved TESTAR's interaction with the SUT of the case study, making it possible to obtain the GUI information of both GUI handling processes. In addition to this, we also added a possibility to check whether there are new running processes in the environment after launching the SUT. If we find them, we save the *pid*s. This way we are able to use different Windows API functions to check whether the process *pid* of the window handle that is in the foreground exists in our internal processes list. This makes it possible to iterate and create a widget tree also for this new visible window handle.

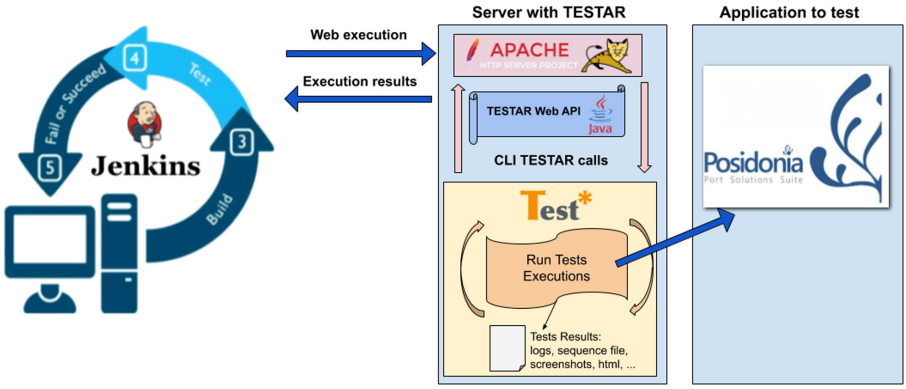

Fig. 3. Integration of TESTAR through an API in a distributed environment

4.3 Distributed TESTAR Execution with a Remote API

In order to integrate TESTAR into the Posidonia CI test cycle, the next step was to design a CI architecture [8,11] in which TESTAR can be invoked remotely in a distributed manner. First, suitable technologies were required for the communication between the: (1) CI server that launches the test execution, (2) the server that contains TESTAR, and (3) the server that executes the SUT.

Thinking about future deployments and enabling TESTAR execution in a test server environment, a Spring boot application was developed with an Apache Tomcat servlet that provides an API for TESTAR settings. Prodevelop offered the initial version of the API that was updated by the TESTAR developers with other necessary requirements, such as new settings parameters for remote login (instead of coding the user login inside the TESTAR Java protocol), and additional configuration options for the initialization of the GUI state model that is built during testing.

With the default implementation, the web API instance should be running in the same directory as the TESTAR tool. Subsequently, when receiving a POST request that is compatible with the TESTAR settings from the CI orchestrator, the contents of the web parameters will be parsed into CLI instructions using the configuration functionality described in 4.1. The flow of the invocation from the CI pipeline is depicted in Fig. 3. The main steps of the functionality are:

1. Upon receiving a web POST request, Posidonia CI orchestrator will send the desired configuration settings to run TESTAR. Only a couple of parameters were needed in the request payload.
2. The remote API is running in the same directory with TESTAR binaries to receive the requests and transform the parameters into a TESTAR configuration that is executed through the CLI.
3. If all the parameters were correct, TESTAR execution will start and a response will be sent back with the output information printed by TESTAR

on the CLI, which includes the test results, the path of the generated sequence and a timestamp to indicate when the sequence began.
4. If more detailed information about any sequence is required, a request will be sent indicating which sequence we want to obtain the resources from.
5. Then a response with the desired resources will be sent back.

4.4 Replay Mode

The objective of TESTAR Replay mode is to offer testers the possibility to re-execute a sequence of actions that has already been executed. This allows testers to verify and debug a sequence for which TESTAR reported finding a failure during automated unattended execution. It is also possible to use this mode to verify that a correct sequence of actions also does not throw any failure in the new SUT versions. Alternatively, we can use it to show that, after a bug fix, the sequence does no longer produce the failure.

A new sequence is started when TESTAR starts the SUT, and executed actions are saved in a Java object stream of the ongoing sequence every time TESTAR executes an action. Information about which action was executed is ready to be replayed, and the state of the SUT does not have to be used for deriving and selecting an available action. The discovered issue of the Replay mode was that TESTAR was not verifying if the SUT is changing between the desired states that we want to follow again by replaying a sequence.

To improve the Replay mode, the information related to the widget in which the action was executed and about the SUT states found, should be stored in the object stream associated with the action executed.

4.5 Output Results and the Structure of TESTAR Logs

The various logs and resources created by TESTAR could offer a large amount of information about the different GUI elements detected by TESTAR in the different states that conform the SUT. However, these files were not stored in a suitable structure for the case study. All the resources were stored in their corresponding directory (logs, sequences, HTML reports, screenshots), but they were stored incrementally according to the sequence number without taking into account the execution of TESTAR. With this structure the objective of synchronizing Prodevelop and TESTAR logs could not be achieved, and therefore, it had to be changed.

The solution was the creating an *index log* and restructuring the output directories according to the timestamp in which TESTAR was launched, in addition to the sequence number. This index can then be used by Posidonia every time it needs to obtain GUI information from TESTAR logs. Using its own logs and its own timestamps, Posidonia will filter the desired sequence in the TESTAR index and will be able to obtain the resource path with all the required information.

In Fig. 4 we can see that Posidonia creates its own logs based on its internal state. If an error occurs, a timestamp will be used to find the matching event from the TESTAR index log to obtain all existing resources and verify which front-end GUI action produced the back-end error.

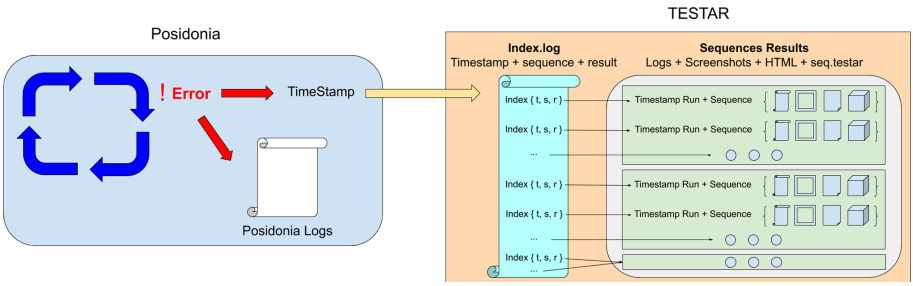

Fig. 4. Posidonia and TESTAR Logs Structure

5 Results

When doing academia-industry collaborations, there are several types of results. On the one hand, the academic tools have improved, they have successfully been adopted in an industrial context, and new ideas are generated for future research. On the other hand, the academic results are validated in an industrial context, and data shows that this improves the quality of the testing practices in the company. Naturally, our goal was to achieve all of these, but unfortunately the second part was not entirely achievable.

Due to various circumstances, the SUT Posidonia evolved into "maintenance only" phase, and Prodevelop decided not to make any changes to their existing testing processes because there are hardly any changes to the SUT anymore. This meant that, unfortunately, we could not really evaluate the performance of TESTAR in a real CI environment.

To try and get some data, we simulated a test with Posidonia by running TESTAR during 4 nightly builds for 12 h with random action selection protocol and a configuration of 30 sequences of 200 actions each night. Unfortunately, the SUT did not change in between, so the outcome of the runs could only differ due to the randomness of TESTAR. The runs showed that the CLI adaptions, the detection of multiple processes and the distributed execution did not fail during long unattended runs.

This outcomes of the test runs were:

- a total of 24000 actions in 120 test sequences
- 15 sequences resulting in *suspicious titles* (all found during the first run)
- 6 sequences resulting in *unexpected close* (all found during the first run)
- 0 sequences resulting in *unresponsiveness*

Analyzing these faulty sequences using the HTML report revealed that:

- 12 of the found *suspicious titles*-failures all lead back to a database connection error in Posidonia when TESTAR executed actions related to querying a port registry.

- the other 3 *suspicious titles*-failures lead to another database connection error in Posidonia trying to generate and obtain the expedient of a port activity.
- the 6 *unexpected close*-failures were all false positives related to the fact that TESTAR tries to bring the SUT to the foreground.

The two errors that were found executed different database requests, and both were related with an error in the Posidonia database connection. Prodevelop was aware of these glitches in the software, but decided not to fix them.

To validate the log synchronization, TESTAR and Posidonia logs were compared to check that the failure sequences found by TESTAR could be mapped to the internal error-logs from Posidonia. The mapping was found correctly and the names of the methods and classes that provoked the exception in Posidonia were meaningful in respect to the properties of the web elements on which the actions were executed. However, there was a delay of 5–10 s between timestamps. This is attributed to the time needed for the internal process to represent and detect the data at GUI level.

The mapping did not only help to verify the synchronization of errors after the execution of a sequence, but also motivated us to investigate the possibility of synchronizing TESTAR with other possible internal logs in order to find a way to improve the action selection based on the available internal methods.

6 Academia-Industry Collaboration

The fact that we could not validate the work completely in a real environment made us reflect again about the academia-industry collaboration. What went wrong here? Why did we find out that the company had stopped active development of the system we planned to test when we were ready with the adjustments to our tool to support their environment?

A myriad of articles [5,12,13] have been written with lessons learned from technology transfer. A simple Internet search with keywords such as university companies, academia industry, collaboration, cooperation, etc, will result in a massive number of hits discussing the issue. Looking at the factors mentioned in the literature, we had them covered (at least that is what we thought):

- we had *funding* through an European research project,
- that gave us the possibility to have *regular meetings*,
- as well as the *approval and commitment of the management* to do the study
- some practitioners at the company had been previously employed at a university, so we had their support and a *collaboration champion on site*
- the objectives of this collaboration were defined to address both the *needs* from academia as well as that of the company
- we worked in *agile sprints* due to the nature of the funded research project
- we *allowed solutions to emerge from the needs of the company* (e.g., the log synchronization, the multiple processes, the distributed execution) that were added to TESTAR to fulfill the requirements of the company)

Academics about industry:	Industry about academics:
- think all problems are solved by increased ROI - keep no track of data - talk lots of waffle - are short term focused - desperately need our solutions but do not (want to) understand this	- are single focused - have no eye for application - are stuck in theory - cannot write a catchy story - have no sense of urgency - only want to write papers - work with you for the funding but will not really give you a solution

Fig. 5. Preconceptions industry and academia have about each other

– a team of academics was *enthusiastic and committed* to contribute to the industry needs and had *previous experience* with working together with similar companies

We started a discussion round with the involved people to figure out what went wrong during the process that lead us to this situation and distill lessons learned for the next time.

We found out that it was mainly the preconceptions industry and academia have about each other, sometimes without even knowing it. These hindered the communication. Everybody thought we were on the same track, but we were not. Many of the preconceptions we detected are in Fig. 5. While the academics thought the company really needed their tool and because of that were working on the case study, the practitioners actually thought the academics only wanted to try this out for the sake of the project on some industrial system and so they provided us one. They were not really looking ahead to the future where the solution would be really used (before the system would go into maintenance).

Successful innovation transfer is about effective communication and emotional intelligence. Both soft skills should receive more attention in computer science curricula.

7 Summary, Conclusions and Future Work

We have presented a case-based evaluation of the academic TESTAR tool on the industrial SUT Posidonia. In order to do this, we integrated TESTAR into the existing CI pipeline of Prodevelop to automatically test Posidonia when the release cycle required it.

The results of this study are threefold. First, TESTAR has been extended with five new valuable features that will be useful also for other test environments (i.e., CLI invocation, multiple processes, distributed testing, replay mode and log synchronization). Second, it was shown to be a useful complement to the existing testing practices and find failures. Third, we learned some lessons on what went wrong during our seemingly perfect collaboration.

Although the collaboration was not without problems, both parties have shown mutual effort in understanding the cause of the problems with the intent to improve. Both parties are currently researching new ways to collaborate and improve their tools. Prodevelop has started the development of a new web application where modern web frameworks and technologies will be used. We intend to continue collaborating in this project. Having already integrated the TESTAR tool into a similar CI environment, the future work will additionally focus on:

1. Improving the visualization of HTML reports. Prodevelop already gave some initial proposals to improve the structure and aesthetic design of the information that TESTAR tool is currently generating.
2. Improving test oracles. In addition to searching for generic suspicious titles, such as error, exception, warning, and HTML error codes like 404, 40X, etc. at GUI level, we aim to define and analyze the usefulness of preparing TESTAR oracles more focused at the web level.
3. Evaluating the recently developed TESTAR functionality for automatically learning GUI state models capturing all the information found in the SUT.
4. Use these state models to optimize the action selection strategies, to automatically measure the GUI coverage, to find the shortest path to reproduce found failures, or to compare two state models from different versions of the same SUT to automatically detect changes at the GUI level.

Acknowledgment. This work has been funded through the ITEA3 TESTOMAT project (www.testomatproject.eu), the EU H2020 DECODER project (www.decoder-project.eu), the EU H2020 iv4XR project (iv4xr-project.eu) and the ITEA3 IVVES project (ivves.weebly.com).

References

1. Aho, P., Vos, T.: Challenges in automated testing through graphical user interface. In: 2018 IEEE International Conference on Software Testing. Verification and Validation Workshops (ICSTW), pp. 118–121. IEEE Computer Society, Los Alamitos, April 2018
2. Aho, P., Vos, T.E.J., Ahonen, S., Piirainen, T., Moilanen, P., Ricos, F.P.: Continuous piloting of an open source test automation tool in an industrial environment. Jornadas de Ingeniería del Software y Bases de Datos (JISBD) 1–4 (2019)
3. Bauersfeld, S., de Rojas, A., Vos, T.E.J.: Evaluating rogue user testing in industry: an experience report. In: 2014 IEEE Eighth International Conference on Research Challenges in Information Science (RCIS), pp. 1–10, May 2014
4. Bauersfeld, S., Vos, T.E.J., Condori-Fernández, N., Bagnato, A., Brosse, E.: Evaluating the TESTAR tool in an industrial case study. In: 2014 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement, ESEM 2014, Torino, Italy, 18–19 September 2014, p. 4 (2014)
5. Beckman, K., Coulter, N., Khajenoori, S., Mead, N.R.: Collaborations: closing the industry-academia gap. IEEE Softw. **14**(6), 49–57 (1997)
6. Coppola, R., Ardito, L., Torchiano, M.: Fragility of layout-based and visual GUI test scripts: an assessment study on a hybrid mobile application. In: Proceedings

of the 10th ACM SIGSOFT International Workshop on Automating TEST Case Design, Selection, and Evaluation, A-TEST 2019, pp. 28–34. ACM, New York (2019)
7. Chahim, H., Duran, M., Vos, T.E.J., Aho, P., Condori Fernandez, N.: Scriptless testing at the GUI level in an industrial setting. In: Dalpiaz, F., Zdravkovic, J., Loucopoulos, P. (eds.) RCIS 2020. LNBIP, vol. 385, pp. 267–284. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-50316-1_16
8. Fowler, M.: Continuous integration (2006). https://www.martinfowler.com/articles/continuousIntegration.html. Accessed 12 Dec 2019
9. Martinez, M., Esparcia, A.I., Rueda, U., Vos, T.E.J., Ortega, C.: Automated localisation testing in industry with test*. In: Wotawa, F., Nica, M., Kushik, N. (eds.) ICTSS 2016. LNCS, vol. 9976, pp. 241–248. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-47443-4_17
10. Meyer, M.: Continuous integration and its tools. Softw. IEEE **31**, 14–16 (2014)
11. O'Connor, R.V., Elger, P., Clarke, P.M.: Continuous software engineering: a microservices architecture perspective. J. Softw.: Evol. Process. **29**(11), e1866 (2017)
12. Rovegard, P., et al.: The success factors powering industry-academia collaboration. IEEE Softw. **29**(02), 67–73 (2012)
13. Sandberg, A., Pareto, L., Arts, T.: Agile collaborative research: action principles for industry-academia collaboration. IEEE Softw. **28**(4), 74–83 (2011)
14. Vos, T.E.J., Kruse, P.M., Condori-Fernández, N., Bauersfeld, S., Wegener, J.: TESTAR: tool support for test automation at the user interface level. Int. J. Inf. Syst. Model. Des. **6**(3), 46–83 (2015)
15. Wieringa, R., Daneva, M.: Six strategies for generalizing software engineering theories. Sci. Comput. Program. 101, 136–152 (2015). Towards general theories of software engineering