The Journal of Systems & Software 176 (2021) 110933

Contents lists available at ScienceDirect

The Journal of Systems & Software

journal homepage: www.elsevier.com/locate/jss

On introducing automatic test case generation in practice: A success story and lessons learned ☆

Matteo Brunetto [a], Giovanni Denaro [a,*], Leonardo Mariani [a], Mauro Pezzè [b,c]

[a] Università di Milano-Bicocca, Milano, Italy
[b] USI Università della Svizzera Italiana, Lugano, Switzerland
[c] SIT Schaffhausen Institute of Technology, Schaffhausen, Switzerland

ARTICLE INFO

Article history:
Received 6 September 2019
Received in revised form 2 July 2020
Accepted 12 February 2021
Available online 6 March 2021

Keywords:
Software testing
Automatic system testing
GUI testing
Automatic test generation
Search based testing
ABT

ABSTRACT

The level and quality of automation dramatically affects software testing activities, determines costs and effectiveness of the testing process, and largely impacts on the quality of the final product. While costs and benefits of automating many testing activities in industrial practice (including managing the quality process, executing large test suites, and managing regression test suites) are well understood and documented, the benefits and obstacles of automatically generating system test suites in industrial practice are not well reported yet, despite the recent progresses of automated test case generation tools. Proprietary tools for automatically generating test cases are becoming common practice in large software organizations, and commercial tools are becoming available for some application domains and testing levels. However, generating system test cases in small and medium-size software companies is still largely a manual, inefficient and ad-hoc activity.

This paper reports our experience in introducing techniques for automatically generating system test suites in a medium-size company. We describe the technical and organizational obstacles that we faced when introducing automatic test case generation in the development process of the company, and present the solutions that we successfully experienced in that context. In particular, the paper discusses the problems of automating the generation of test cases by referring to a customized ERP application that the medium-size company developed for a third party multinational company, and presents $ABT_{2.0}$, the test case generator that we developed by tailoring *ABT*, a research state-of-the-art GUI test generator, to their industrial environment. This paper presents the new features of $ABT_{2.0}$, and discusses how these new features address the issues that we faced.

1. Introduction

Designing effective test suites requires a significant effort that dramatically affects both testing and development costs, with a relevant impact on the software development schedule, the project deadlines, and ultimately the quality of the final product. The recent advances in automated test case generation and the new tools for automatically generating test suites can reduce the effort required to generate effective test suites, and thus improve the cost-effectiveness of test case generation activities.

Several popular research prototypes (for instance, Randoop Pacheco et al., 2007, JBSE Braione et al., 2013, Evosuite Fraser and Arcuri, 2013 and SUSHI Braione et al., 2017) and commercial products (for instance, PEX Tillmann and de Halleux, 2008 and Parasoft Testing C/C++ Test Parasoft, 2021) provide good support for automatically generating *unit* test suites. The increasing success of test generation tools for unit testing already produced studies about the issues associated with their adoption in industry (Fraser et al., 2015).

Tools for generating *system* test cases can address applications in both critical (Wang et al., 2018b) and non-critical domains (Almasi et al., 2017; Arcuri, 2019). The industrial practice for automatically generating system test cases for GUI applications offers tools that capture, generalize, and execute behaviors observed when users interact with the system (GmbH, 2020). Studies about the effectiveness of research prototype tools for generating system test cases provide evidence of the effectiveness of interesting approaches on medium-scale and small-scale open source software systems, notably the work on ABT (Mariani et al., 2014), Exsyst (Gross et al., 2012), WATEG (Thummalapenta et al., 2013), GUITAR (Yuan and Memon, 2010), and Gazoo (Arlt et al., 2012).

☆ Editor: Raffaela Mirandola.

* Corresponding author.
E-mail addresses: matteo.brunetto@unimib.it (M. Brunetto), giovanni.denaro@unimib.it (G. Denaro), leonardo.mariani@unimib.it (L. Mariani), mauro.pezze@usi.ch (M. Pezzè).

https://doi.org/10.1016/j.jss.2021.110933
0164-1212/

However, introducing automated system testing solutions available in the form of research prototypes in industrial software development processes challenges automated test case generators with problems that can be hardly experienced with medium-scale and small-scale open source software (Braione et al., 2014). Some experiences reported so far refer to the successful introduction of automatic test case generation tools for unit testing (Almasi et al., 2017) and REST API testing in industrial settings (Arcuri, 2019), but there is little attention to the specific issues that must be addressed to effectively introduce automatic GUI test case generators within commercial organizations.

This paper reports an exploratory study that investigates the issues that arise when introducing a leading-edge-research GUI test generator, *ABT* (Mariani et al., 2012), in a commercial organization. *ABT* generates system test cases by sampling possible GUI interaction sequences. *ABT* relies on Q-Learning, a machine learning algorithm, to steer the test generation process towards relevant functionalities of the target application (Sutton and Barto, 1998). In this paper we (i) introduce the relevant issues that led a software house to consider an automated approach for generating system test suites, (ii) present the sometimes unexpected technical and organizational obstacles that prevented the straightforward adoption of *ABT* for automatically generating system test suites, (iii) discuss the technical improvements that we made to overcome those obstacles, and that led to $ABT_{2.0}$, a system test case generator that extends *ABT* to address the specific industrial process, and (iv) illustrate the lessons that we learned with our experience and that may generalize to other commercial applications that share the characteristics of our study.

The study reported in this paper is the result of a one-year pilot technology-transfer project. It follows an exploratory design because of the initial lack of clear propositions and hypotheses on the potential constraints that could limit the effectiveness of *ABT* in the considered industrial setting. During the project, we faced several challenges and learned several lessons about *scalability*, *test reporting* and *oracles* that are specific to the problem of automatically generating test cases, and that represent major barriers to the adoption of automated system test case generation tools. Interestingly, we also found that we could largely or totally address the most relevant issues with domain-tailored solutions that adapt well to the specific business oriented applications considered in our project, thus reducing the need for difficult-to-find general solutions that would have significantly delayed the adoption of the test generator.

In the light of the issues that we observed during the project, we extended the tool *ABT* with new solutions that exploit the domain specific characteristics of the applications under test, to produce an effective test generation tool, $ABT_{2.0}$, tailored for testing business oriented applications. Based on the results of our experiments with an ERP application considered in our project, the study reported in this paper provides initial empirical evidence that our extensions are the key ingredients to turn our academic test case generation tool *ABT* from an ineffective to an effective solution for testing industrial business oriented applications. Our experience refers to a specific system testing technology and a specific product under test, and cannot be directly generalized. However, the challenges that we faced and the solutions that we designed can be generally relevant for both researchers and practitioners, who might think to apply similar strategies in different contexts to make automatic system testing more effective.

In a nutshell, this paper contributes to the state of the art by:

- identifying some important challenges to automatically generating system test suites in the context of industrial software systems;
- proposing original extensions of the test generator *ABT* that improve the effectiveness of the approach for automatic system testing of industrial business oriented applications. While the core algorithm of *ABT* refers to our previous work (Mariani et al., 2012), the extensions reported in this paper are entirely novel contributions that we present for first time (Sections 4 and 5). The proposed extensions improve the scalability of the GUI exploration strategy of *ABT*, produce effective test reports during the test generation process, and support the definition of domain specific test oracles;
- presenting the experience of exploiting the proposed extensions in the context of a case study concerned with generating tests for a business oriented application developed by our industrial partner.

This paper is organized as follows. Section 2 overviews *ABT*, the technology that we identified as a viable test case generation tool for the target industrial case. Section 3 discusses the main challenges in introducing automatic test case generation in an industrial development process by referring to our experience with a medium-size company that produces software solutions on demand. The section describes the industrial context of the technology transfer project, the challenges that we faced, and the domain-specific opportunities offered by the application under test. Sections 4 and 5 discuss the solutions to the technical and organizational challenges, respectively. These solutions led to $ABT_{2.0}$, and to the extension of *ABT* to business oriented applications that meet the industrial requirements of our project. The sections present the new features, and discuss experimental evidence of their effectiveness. Section 6 distills some key lessons learned that can be useful to researchers and practitioners working on automatic system testing. Section 7 discusses the main threats to validity. Section 8 surveys the related research efforts, and Section 9 summarizes the contributions of the paper and outlines our current research plans.

2. AutoBlackTest

AutoBlackTest (*ABT*) is the test generation technology that we referred to in our project. In the first part of our exploratory case study, we used *ABT* to gain sample data on the extent of the applicability and the limitations of a representative state-of-the-art test generator, when challenged with the industrial applications considered in the pilot project. Later in the project, we used *ABT* as a platform to concretely develop and experience with new test generation strategies that we designed to exploit the opportunities identified in the project. Our initial experience with introducing *ABT* in the industrial context was the driver that enlightens both the challenges and the opportunities that we discuss in Sections Section 3.2, which led to $ABT_{2.0}$ that enriches *ABT* with new functionalities to meet both technical and organizational industrial requirements. In this section we introduce *ABT* to make this paper self-contained. The interested readers can refer to Mariani et al. (2011, 2012, 2014) for a comprehensive discussion of *ABT*.

ABT generates system test cases for applications that rely on GUI driven interactions based on Q-Learning (Sutton and Barto, 1998), a learning algorithm that *ABT* uses to steer the testing activity towards the most relevant functionalities of an application. In this paper, we refer to the *ABT* Q-Learning exploration strategy as *Reinforcement Learning based Strategy (RLS)*. Q-Learning is extensively used to address the problem of agents that learn how to interact with unknown environments. Q-learning agents learn how to interact with the environment by iteratively selecting and executing actions, and updating a model of the system that estimates the utility of the actions according to the reactions of the

environment. Q-learning agents interleave random and model-based selection of actions, to take advantage of the incrementally build models, and increasingly explore relevant portions of the environments.

The *ABT* Q-learning agent computes the utility of actions as the impact of the actions on the GUI, based on the intuition that the impact of an action on the state of the application should produce observable effects. Actions that produce relevant unseen transitions in the GUI, for instance actions that lead to successfully submitting a complete registration form, likely correspond to important interactions, and are thus given high utility values. While actions that produce negligible or already seen transitions, for instance actions that repetitively lead to incomplete forms or to error windows that bring the application back to its initial state, likely correspond to scarcely relevant interactions, and are thus given low utility values. By rewarding actions based on their impact on the GUI, the *ABT* Q-learning agent steers the testing activity towards combinations of actions that correspond to important interactions, and reduces the amount of repetitive actions with respect to selecting actions with a purely random process.

The *ABT* testing process initializes the Q-learning model to the home page of the application, and builds interactions (i.e., test cases) as sequences of episodes, where the number of sequences and the length of each sequence are parameters. Each episode starts from a random state of the current Q-learning model, and executes a fixed number of actions that are selected according to the ϵ-greedy policy.[1] The ϵ-greedy policy alternates exploration and exploitation. Exploration selects actions never executed before, exploitation executes the most useful action according to the knowledge accumulated so far by the agent. In particular, when *ABT* is in a state not in the Q-learning model yet, the policy selects a random action (exploration). When *ABT* is in a state already in the Q-learning model, the policy executes a random action with probability ϵ (exploration) and the action with the highest Q-value according to the model with probability $1-\epsilon$ (exploitation), where ϵ is a user-provided parameter.

Executing actions may require some data, for instance, editing a textfield requires identifying the data to be entered in the field. *ABT* determines values by using a catalog of values that associates labels, such as `email` and `date`, to a set of values, such as `mariani@disco.unimib.it` and `20-04-2019`. When interacting with an input widget, *ABT* analyzes the GUI to determine the label of the GUI that better describes the values expected by the input widget (Becce et al., 2012). For instance, *ABT* can determine that an email address should be entered in an input field from the presence of the label `email` next to the input widget. When entering data, *ABT* randomly selects a value from the set of values associated with the label, `email` in the example. When the label is not present in the catalog, *ABT* selects a string from a default set of values.

Software applications may include complex functionalities that are hard to successfully execute using random exploration only. For instance, an application may include a form that can be successfully submitted only by filling several fields. An execution that enters values in all the fields of the form and then submits the form has little probability to be produced randomly. To address these cases, *ABT* supports the execution of complex actions, which are short workflows that execute a sequence of actions according to a specific strategy. For instance, when an application displays a large form, a complex action may facilitate submitting the form by entering a value in each field and then clicking the submit button.

[1] Q-Learning supports several policies, the ϵ-greedy policy is demonstrated to be effective for test case generation (Mariani et al., 2014).

ABT considers the execution of complex actions before applying the standard ϵ-greedy policy. In particular, if the current state of the application allows to execute a complex action, *ABT* selects it with probability *pcomplex*. If the complex action is not executed, *ABT* applies the regular ϵ-greedy policy.

ABT tool is implemented in Java and integrates IBM Functional Tester (IBM, 2021), a commercial capture and replay tool, to interact with the GUI of an application. *ABT* can support the same GUI frameworks supported by IBM Functional Tester. The experience reported in this paper exploits the support for .NET applications. *ABT* also uses the free library TeachingBox (Ertel et al., 2009) to handle the Q-learning model.

In our industrial experience, we executed *ABT* using the ϵ-greedy policy with $\epsilon = 0.7$ and with $pcomplex = 0.5$. We initialized the catalog of input values with 35 entries comprised of valid user names, emails and data item identifiers, for instance, the identifier of an invoice, to satisfy the input fields with domain specific constraints, and used random strings for the unconstrained input fields. The testing process consisted of 50 episodes, each one composed of 30 actions. We determined the values of the parameters based on both our experience with the tool (Mariani et al., 2014) and a trial and error fine-tuning process executed on the target industrial application.

In the next sections, we discuss how we adapted *ABT* to face the challenges of business oriented applications, ending up with extending *ABT* to $ABT_{2.0}$ by redesigning the GUI exploration strategy and extending its test reporting capabilities.

3. Testing business oriented applications: Challenges and strategies

In a recent joint project between the LTA research laboratory[2] and an industrial partner,[3] we evaluated the benefits of automating the generation of system test suites for business applications that rely on GUI-driven interactions. In this section we report the technical obstacles that derive from the complexity of the target application, and the organizational issues due to the specificity of the software process of the industrial partner. We discuss obstacles and opportunities offered by the distinctive nature of the business oriented application targeted in the project, by focusing on scenarios that are common to many diverse applications and domains, and that can be exploited beyond the scope of our project to effectively automate the generation of system test cases. We introduce the business oriented application that we targeted in the project, by presenting an essential overview and discussing challenges and opportunities for automatically generating system test suites, effective in the business context.

3.1. Business oriented applications: A sample

We conducted our exploratory study on an application that our industrial partner selected as representative of typical customized software applications developed on demand for third parties. In this section, we outline the distinctive characteristics of the selected application, which we refer to as *ERP* here on. In a nutshell, *ERP* is an application that handles the commercial process of a company by managing data entities like orders, invoices, shipping activities, and maintenance requests. We illustrate our industrial experience using a simplified GUI with the same characteristics of the GUI of the original application,

[2] LTA — Laboratory for software Testing and Analysis, Università degli studi di Milano Bicocca.

[3] A medium company producing customized software solutions on demand operating in the area of North Italy, whose identity is not disclosed for non-disclosure agreement policies.

and using examples with realistic although not real data, due to confidentiality reasons that do not allow us to disclose the original application.

We present the application *ERP* through its GUI and the corresponding test plans as designed by the company testers according to the customers' requirements, to illustrate the requirements for the automatic test generator.

The GUI of *ERP* is organized by data entities, following a typical interaction with the GUI, where users first select the type of entity, such as orders or invoices, and then manipulate it. Fig. 1(a) shows the set of six main data entities through the (simplified and anonymized) *ERP* home page. Buttons in the top of the home page lead to entity specific pages that offer the first tier action sets for the entity types. Fig. 1(b) shows the *Invoices* page with the top tier actions for entity Invoices, actions that lead to other pages when selected, for instance the *Edit Invoice* page illustrated in Fig. 1(b). *ERP* manages entities that include several data fields editable across multiple tabs, and editing an *ERP* entity may require long GUI interaction sequences that can terminate successfully (*Save* button) or can abort (*Close* button). *ERP* manages six types of entities, with an average of 20 fields per entity. Editing an entity requires interacting with five tabs in average.

New *ERP* releases are delivered with test plans that are composed of test objectives, where the test objectives are a set of behaviors exercised during testing, each consisting of interactions executed against the target applications along with corresponding checks on the validity of the test outputs. Test plans are organized in sections, one for each tested entity type. A sample test objective for entity invoices is

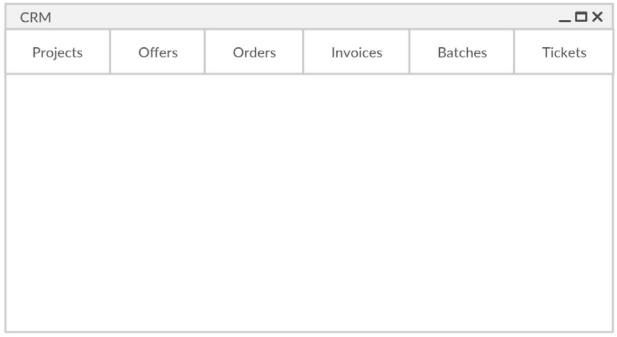
> "when the *new invoice* button is pressed, a form with five tabs and only empty fields must be shown to the user"

Test plans are stored as spreadsheets with a sheet for each entity type. A test objective corresponds to a row in a sheet with three fields (columns): identifier, GUI interactions, and test checks. The identifier uniquely identifies the test case. The GUI interactions are the set of interactions that characterize the test, for instance "press the *new invoice* button, …". The test checks are a set of checks to determine the correctness of the results, for instance "check that a form with five tabs and all empty fields is shown".

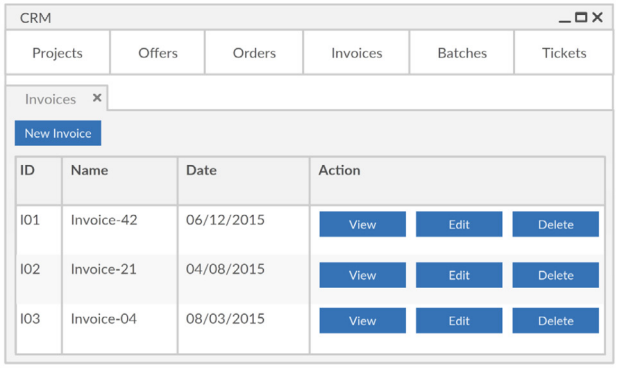
The *ERP* test plans resemble the structure of many test plans used in small and medium size software companies that work with internally established though informally defined test procedures, and handle the test documentation with common back office tools, like spreadsheets.

3.2. Challenges and opportunities

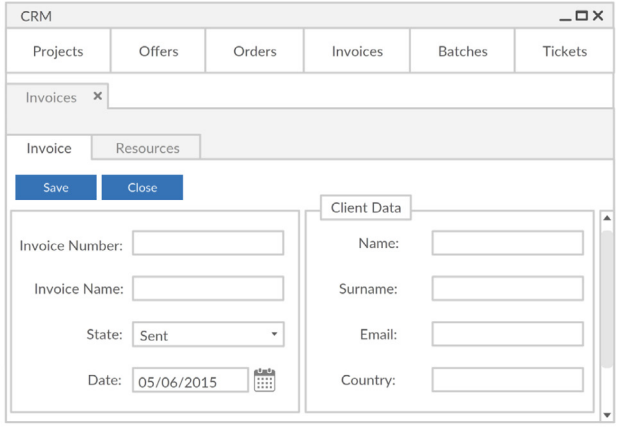
While integrating the test case generator *ABT* (AutoBlacktest Mariani et al., 2012, the test generator that we described in Section 2) in the process of our industrial partner,[4] we faced both technical and organizational challenges: (i) the *size* of the *ERP* GUI opens a variety of interaction choices, much more than classic benchmarks used in our previous academic validation, with a strong impact on the effectiveness of *ABT*, (ii) many failures derive from incorrect results that can be observed only with non trivial *test oracles*, (iii) the commercial practice of the industrial partners requires *test reports* that consistently match the spreadsheet-style test plans maintained by the organization,



(a) Home page



(b) Invoice page



(c) Edit Invoice page

Fig. 1. GUI screens of the sample application (adapted and simplified from the original user interface).

which are not directly compatible with the outputs produced by *ABT*. While size and oracles are technical challenges, generating test reports that meet the customers' standards is mainly an organization-specific issue, which nonetheless must be taken into careful consideration to successfully transfer research to industry.

These challenges generalize to many industrial applications beyond the application domain considered in this paper: (i) the difficulty of test generators to cope with huge, deep and non uniform program state spaces is well-known, (ii) automatic test generators suffer from the test oracle problem, since they are seldom able to automatically determine the exact expected behavior for the functionalities tested by each test case, (iii) automatic

[4] Our target application, *ERP*, is developed in C# with Microsoft graphical libraries. *ABT* supports different programming languages, including C# and the Microsoft graphical libraries used in *ERP*.

test generators often fail to meet the required test documentation standards. This section contextualizes the impact of these challenges in our project and discusses some opportunities that can be exploited to address them. In fact, though general solutions seldom exist, industrial business oriented applications can be addressed with domain specific solutions that derive from recurring design patterns, which relate the GUI structure to both the business logic and the test requirements of these applications.

Size of the interaction space. The classic structure of business oriented applications and the many choices that they offer at each interaction step produce an interaction space that is both extremely large and strictly structured.

As an example at each interaction step, *ERP* users can interact with 95 different menus and buttons available in the top menu bar, and with many widgets that become available in the many displayed windows. Since the top menu bar is continuously available for interactions, the *space of possible interactions grows exponentially* with the length of the interaction sequence, making a dense exploration of the execution space impossible. The 95 permanently enabled actions in the *ERP* top bar produce a space of test sequences with at most five steps that contains more than 7 billion cases, and a several orders of magnitude larger space when considering also the actions that become available in the different windows after selecting the menu items. Interactions sequences with more than five steps, which are common in *ERP*, exponentially widen the already giant execution space. Effective test generation strategy must select a relevant subset of test cases.

The main problem of testing GUIs is the huge spaces of interaction sequences that derive from the combination of windows and actions (Banerjee et al., 2013). Current approaches address this problem by relying on the availability of some GUI models to assist the identification of relevant interaction sequences. For example, White and Almezen concentrate on *complete interaction sequences*, defined as sequences that correspond to activities of direct interest to the user, in turn defined as activities that produce observables effect on the surrounding environment of the GUI, such as changes in memory, changes in the behaviors of some peripheral devices, or changes in the underlying software or application software (White and Almezen, 2000). White and Almezen's approach assumes that testers provide a finite-state machine model that identifies the sets of complete interaction sequences on which testing shall focus separately. Similarly, Paiva et al. requires hierarchical state machines to steering the test selection strategy (Paiva et al., 2005). Yet, Memon et al. require testers to annotate GUI events with pre- and post-conditions, and identify the initial and goal states that the test generator shall focus on Memon et al. (2001). Finally, Saddler et a. also relies on pre-defined constraints to direct the test case generation process Saddler and Cohen (2016).

In our project, we failed in proposing these solutions, due to the expertise, costs and time required to define the models that these approaches require. On the other hand, we successfully took advantage of the characteristics of the GUIs of the target applications, GUIs that are organized by data entities, as discussed in the previous section. We rely on the characteristic organization of the target GUIs to automatically partition the interaction sequences into relevant and irrelevant sequences. The solutions that we discuss in this paper explicitly exploits the specific characteristics of the application domain, and as such, our work radically differs from the above-surveyed approaches, which generate test cases for general purpose productivity applications, like Microsoft Windows, the GVISUAL multimedia database (White and Almezen, 2000), Microsoft WordPad (Memon et al., 2001), Microsoft Notepad (Paiva et al., 2005).

We defined an industrially-acceptable solution for the considered application domain, by relying on both the structured organization of the GUI and by carefully addressing operations that depend on long interaction sequences in forms with many input widgets. The *strictly structured organization of the GUI* drives the common interactions with the application, thus providing important guidelines to generate meaningful test sequences, if properly handled. For instance, *ERP* users follow goal-oriented interaction patterns that correspond to the GUI organization. The top menus partition the interaction space according to the primary goal of the interactions, for instance users add orders by navigating across windows enabled after selecting the order menu, and unlikely move to windows entered through top tier actions alternative to the order menu before completing the overall add operation. Thus the execution space contains many interaction sequences that do not correspond to reasonable interactions, and test case generators that explore the execution space without considering this information produce enormous amount of almost useless test cases.

Opportunity

Graphical menus partition the functional logic. In most business oriented applications, top tier menus gather the operations that manipulate each type of data entities, and thus menus partition the functional logic of the application in areas. For example, the menus in Fig. 1 indicate that the functionalities to manipulate invoices are accessed with action sequences that start with menu *Invoices*. This common type of GUI structure is designed to facilitate the users to easily navigate and operate on different data entities. Test generators can exploit this structure to mitigate the combinatorial explosion of interaction sequences that the test generator may produce, by both focusing on interaction sequences that start from distinct graphical menus, and ignoring the sequences that include actions that jump across different menus before completing any operation. This simple heuristic can largely reduce both the size and complexity of the execution spaces to be sampled.

We frame our second observation by defining the concept of *minimal interaction sequence* for a meaningful use of a system functionality as the sequence of GUI actions that must be necessarily executed to exercise a functionality relevant to the user. The goal of testing is to maximize the set user-relevant functionalities exercised when executing the test suite. State-of-the-art randomized test generation strategies, which range from purely random strategies (Developers A. Monkey UI, 2021; Bertolini et al., 2010) to approaches interleaving random decisions with heuristic decisions (Mariani et al., 2012), are less and less effective in producing test cases that exercise all relevant functionalities when the length of the corresponding minimal interaction sequences grows. For example, state-of-the-art randomized test generation strategies can effectively generate test cases that exercise the functionalities that correspond to minimal interaction sequences of two actions, for instance simple CRUD operations like deleting an invoice. They are much less effective in generating test cases that exercise functionalities that correspond to long minimal interaction sequences, for instance CRUD operations like modifying an invoice. For instance, the minimal interaction sequence for deleting an invoice consists of selecting the menu *Invoices* in the screen in Fig. 1(a), and then clicking *Delete* in the screen in Fig. 1(b). In contrast, the minimal interaction sequence for modifying an invoice consists of selecting the menu *Invoices* in the screen in Fig. 1(a), clicking *Edit* in the screen in Fig. 1(b), executing a sufficient number of

fill-in interactions with the input fields, and then clicking *Save* in the screen in Fig. 1(c). In a nutshell, state-of-the-art randomized test generation strategies can effectively produce test cases for the 'delete invoice' operation, but not for the 'modify invoice' operation.

To give a concrete feeling of the impact, the length of the minimal interaction sequence of operations that modify data in *ERP* ranges from 8 to 38 steps. When the probability of picking up a specific action is close to 1%, such interaction sequences are singularities in the execution space, hard if not impossible to exhaustively cover with randomized strategies. For instance, a ten hour run of our test generator *ABT* against *ERP* generated test cases that exercised most of the short minimal interaction sequences, but hardly executed any operation whose minimal interaction sequence involved more than three actions. In particular, *ABT* easily explored different selection orders of the graphical menus, but never executed the interaction sequences to modify the data entities in *ERP*.

Opportunity

Long interaction sequences derive from many input widgets. Operations that depend on long interaction sequences, such as operations to create and modify data entities, are very challenging for automatic test case generators. Long interaction sequences often derive from forms with many input widgets to be filled-in. For instance, the simple input form to create a new invoice of Fig. 1 (c) includes eight input fields, and input forms with tens of input fields are common in industrial business oriented applications. Handling input forms as special entities can largely improve the effectiveness of test case generators, by increasing the probability of a form to be completely filled-in before being submitted.

Test oracles. Effective test cases pair test inputs with relevant test oracles that can detect misbehaviors with respect to specific users' expectations.

Several state-of-the-art test case generators, such as Pex (Tillmann and de Halleux, 2008), JBSE (Braione et al., 2013), Evosuite (Fraser and Arcuri, 2013), BiTe (Baluda et al., 2015), and AFL (Gutmann, 2016), focus on computing test suites that exercise the set of executions of the applications under test as thoroughly as possible, mostly referring to code coverage metrics as indicators of testing thoroughness. These approaches do not generate test oracles, thus can reveal only *blatant failures*, that is, uncaught exceptions and system hangs, which represent a small fraction of failures that software companies aim to reveal with in-house testing. Some techniques generate assertions to reveal regression failures in future releases of the software. Such assertions are useful for regression testing, but not for testing new and extended functionalities.

For example, the large majority of the test requirements in the original *ERP* test plan include checks of the correctness of the execution that require inspecting the GUI and the database state, and cannot be identified by simply detecting uncaught exceptions and system hangs. Common examples of checks included in the original *ERP* test suite are: to make sure that graphical menus, buttons, and text fields appear with correct labels and can or cannot be modified in given screens and application states; to ensure that data change operations result in correct updates of the corresponding tables in the database; to check that the application visualizes the expected subsets of data items from the many items in the database, according to filters set in the input forms. Checks like these go beyond the revealing capabilities of simple implicit oracles that reveal only uncaught exceptions and system hangs.

The problem of designing cost-effective test oracles has been largely investigated in the last decades (Barr et al., 2015). Oracles are often provided as assertions embedded either in the test scripts, as in Junit, or in the code in the form of contracts (Meyer, 1988; Briand et al., 2003). Many approaches generate oracles from formal or semi-formal specifications, when available (Spivey, 1989). Yet other approaches infer test oracles from informal specification in natural language (Blasi et al., 2018; Böttger et al., 2001). Regression testing derives oracles by monitoring the execution of the software under test (Xie, 2006), while dynamic analysis infers invariants (Ernst et al., 1999) or models (Lorenzoli et al., 2008) that can be used as test oracles.

Unfortunately none of these approaches applied well to our context: code was given without either embedded assertions or any formal specifications, regression oracles were not sufficient, and dynamic analysis oracles were largely insufficient to represent the properties of the test plans of our industrial partner. Nonetheless, although we could not automate the test oracles, we mitigated the cost of manually checking the test results while executing the test suites, by automatically producing test reports with data about the effects on both the GUI widgets and database tables. These data were relevant for the manual inspection of the software requirements, and for post mortem analysis of the failures.

Opportunity

Small sets of outputs capture most relevant classes of non-blatant failures. There often exists a well identified and relatively small set of output data that capture the most relevant classes of *non-blatant* failures. In fact, many test requirements are concerned with checking the correctness of either attributes of widgets that frequently appear in the GUI after some operation, for instance graphical menus, buttons, text fields and data grids, or database changes produced as a result of an operation. For example, an entry of the *ERP* test plan requires to check that selecting the menu *Invoices* leads the GUI to display an editing panel *Invoices* that visualizes all the buttons as shown in Fig. 1(b), and to check that filling the input form of Fig. 1(b) with valid data and saving it leads to inserting a new record in the database table *INVOICES*, with data consistent with the data in the form. Thus, augmenting the output of the test generator with custom test reports that include the relevant data from both affected GUI widgets and affected database tables could effectively assist testers in the identification of the relevant failures.

Test documentation. In the industrial context of customized software developed on demand, such as the *ERP* case, a fundamental requirement for the test reports is to map the test results to the test objectives in the test plans (which in turn map to the requirements). In fact, in many industrial sectors, software companies ask for test reports that help testers easily identify both tested and not-yet tested functionalities, to efficiently plan the test campaign and deployment plans: Reports that include *detailed information about the tested functionalities* are necessary to integrate a test generator in an industrial testing process.

The traceability between requirements and test cases is well supported when test cases are derived from the requirements (Ramesh and Edwards, 1993; Lago et al., 2009; Winkler and Pilgrim, 2010), but becomes a hard problem when test cases are automatically generated by sampling the input space on the implementation. State-of-the-art test case generators document the generated test suites as (i) executable test scripts that instantiate the test inputs, (ii) uncaught exceptions and system hangs experienced while executing the test scripts, and (iii) statement

coverage. Unfortunately, mapping the generated test cases to the system functionality by manually replaying each test script almost nullifies the benefits of using a test generator, and deducing the missed functionality by identifying inputs that lead to uncovered code can be extremely hard.

Opportunity

Mapping between actions on widgets and test requirements suggest mapping from test results to system functionalities. The neat mapping between actions executed on the widgets in the GUI, such as clicks on graphical menus and buttons, and relevant classes of test requirements, such as validating GUI screens and database changes, provides useful information for test reports. Pairing the actions executed on GUI widgets with relevant results on both other widgets and database tables simplifies the identification of the tested and untested test objectives, as well as the comparison between test results and test oracles.

In summary, our experience highlights that (i) graphical menus partition the functional logic of many business oriented applications, (ii) long interaction sequences often derive from forms with many input widgets to be filled-in, (iii) often a small set of output data captures the most relevant classes of *semantic* failures, and (iv) the mapping between actions on widgets and test requirements provides useful information for mapping test results to system functionalities.

4. GUI exploration strategy

We addressed the main technical challenges of introducing automatic test case generation in industrial settings by exploiting the opportunities offered by (i) the partition of the functional logic induced by the menus defined in the GUI, (ii) the relations among action sequences induced by complex input forms, (iii) the relation between outputs and semantic faults, and (iv) the mapping between test results and system functionalities induced by the widget–test requirements mapping.

In this section, we present the new strategy that we defined to efficiently explore the execution space though the GUI, by exploiting the first two opportunities listed above, and discuss experimental results that confirm the effectiveness of the new strategy for the considered class of applications. Section 5 presents the contributions related to the other opportunities.

4.1. The SSRLS GUI exploration strategy

$ABT_{2.0}$ extends *ABT* with a new *Semi Systematic RLS (SSRLS) exploration strategy* that substitutes the *ABT* RLS strategy with a systematic exploration of the GUI in two cases:

- *Menu-driven constraints:* SSRLS constrains RLS to enforce a *systematic partitioning* (Yilmaz, 2013; Yilmaz et al., 2014) of the generated test cases in groups, based on the items in the graphical menus of the GUI, such that each group of test cases starts accessing a graphical menu and then exercises only actions associated with the initially-accessed menu. Thus, SSRLS prevents useless and time consuming exploration of the many irrelevant interactions sequences that jump across different graphical menus.
- *Input-form constraints:* When dealing with input forms SSRLS supersedes RLS and executes complex actions that properly fill out and submit the input forms, thus increasing the coverage of the functional logic of the application under test, and avoiding the useless and time consuming exploration of too many incorrect completion of forms before correct submissions.

In a nutshell, SSRLS combines the ability of reinforcement learning to quickly reach many different GUI states, with both the ability that systematic testing offers to optimally partition the execution space and the capability that complex actions offer to handle the implicit dependencies present in a form-based window.

Algorithm 1 presents the SSRLS exploration strategy by highlighting the steps added to the *ABT* RLS strategy: Light gray (steps 3, 4, 6 and 14) and dark gray background (steps 8 and 9) highlight the statements that implement the partitioning of the execution space and the ad-hoc handling of input forms, respectively.

The main loop (steps 1–15) generates an episode (a test case) per iteration until the testing budget is over. The algorithm initializes episodes with a sequence of actions that reach a randomly selected state of the model learned so far, that is a state observed while executing a previous episode (step 2). In particular, at the first iteration of the loop the model is empty, and the algorithm generates an episode from the main page of the application. In this initial state, no menu action has been executed yet, making the condition of step 3 evaluate to true, and the algorithm augments the episode with a menu action (step 4) followed by a sequence of GUI actions selected among the ones that become dynamically available on the GUI (loop at steps 7–13).

Algorithm 1 SSRLS: the Semi Systematic RLS

```
Require: RLS: the Reinforcement Learning based Strategy
1: while the testing budget is not over do
2:     current_state = RLS(goToARandomState)        ▷ reach a random state of the Q-learning model
3:     if MenuAction not done yet then
4:         current_state = RLS(doMenuAction)        ▷ select a menu
5:     end if
6:     RLS(disableMenuActions)  ▷ actions on graphical menus cannot be anymore selected by RLS
7:     for 1 ⇝ number of actions per episode do
8:         if current_state is InputForm AND with some probability π then
9:             current_state = doFillAndSubmit      ▷ Fill and submit the input form
10:        else
11:            current_state = RLS(doAction)        ▷ Execute next action
12:        end if
13:    end for
14:    RLS(enableMenuActions)  ▷ actions on graphical menus can again be selected by RLS
15: end while
```

Support to the partitioning of the execution space (light gray background): An episode starts with a menu action and then continues only with actions associated with the selected graphical menu.

Support to input forms (dark gray background): The input forms are entirely filled in and submitted with probability π.

The new steps 6 and 14 of the algorithm disable the possibility of executing actions on the graphical menus immediately before the main *for* loop, and re-enable the menu actions at the end of an episode, respectively. These new steps guarantee that each generated test case starts with an access to a given graphical menu and exercises only actions associated with that graphical menu. If step 2 of the algorithm executes an initial sequence of actions that already includes a menu action, step 3 of the algorithm prevents including additional menu actions in the current episode. Thus, step 2 of the algorithm can select the initial sequence of actions of an episode by navigating the current model, without any additional constraint, since the algorithm maintains the invariant that current model contains only sequences of actions that previous episodes selected according to the SSRLS strategy.

The algorithm generates episodes of a fixed parametric length, and selects actions according to the original RSL strategy (step

11), unless differently driven (the gray background steps that we discuss next).

Steps 8 and 9 of the algorithm control the generation of sequences of actions in the presence of input forms. Step 8 of the algorithm checks if the current GUI state corresponds to a screen with an input form, and if so, it executes step 9 with some probability π. Step 9 of the algorithm fills out all the input fields of the form with predefined values before clicking on the submit button. Step 9 exploits the *ABT* capability to execute a set of actions. The probability π is a parameter of the algorithm. In the experiments reported in this paper, we set $\pi = 0.5$, after an empirical evaluation of different values for π.

4.2. SSRLS effectiveness

We extended the ABT prototype with SSRLS to validate the new approach on the industrial business oriented application made available by our project partner.

We investigated two research questions related to the effectiveness of SSRLS:

- RQ1: Does SSRLS *thoroughly exercise* the functionality of the business oriented application under test?
  We evaluate the capability of SSRLS to generate test cases that thoroughly exercise the functionality of the application under test both in absolute terms and in comparison to RLS. We measure the effectiveness of the test cases generated with both SSRLS and RLS based on the frequency with which they execute different classes actions, with particular attention to actions that depend on long interaction sequences.
- RQ2: Does SSRLS contribute to satisfy the *test objectives*?
  We evaluate the quality of the generated test cases with respect to the test plan, both in absolute terms and in comparison to RLS. We measure the significance of the generated test cases as the percentage of test objectives (and thus corresponding requirements) that the automatically generated test cases exercise.

We mitigated the impact of randomness by repeating each experiment five times, and reporting mean values. We could not negotiate a higher number of repetitions with our industrial partner, due to the cost of the experiments and the limited access to the ERP application. The stability of results across the repeated experiments convinced us about the significance of the results in the scope of our exploratory study, but the limited amount of repetitions does not allow strong claims that our results generalize.

RQ1: Does SSRLS thoroughly exercise *the functionality of the business oriented application under test?*

We measure the effectiveness of SSRLS test suites by comparing the amount of action types that the test cases generated with SSRLS and RLS exercise, respectively. We identify five main classes of actions that reflect the nature of GUI interaction:

- *Menu* actions that interact with menus;
- *CRUD* actions that initiate a CRUD (Create, Read, Update, Delete) operation;
- *Input* actions that enter data in input fields;
- *SaveKO* actions that cancel submissions or submit invalid forms;
- *SaveOK* actions that submit valid forms.

To thoroughly exercise the functionalities of the application under test, it is important to execute both *SaveKO* and *SaveOK* actions, many of which are executed only with non trivial combinations of *Menu*, *CRUD* and *Input* actions that reach specific windows and states. We measure the effectiveness of RLS and SSRLS test cases as the amount of executed actions of different types, with specific attention to *SaveKO* and *SaveOK* actions.

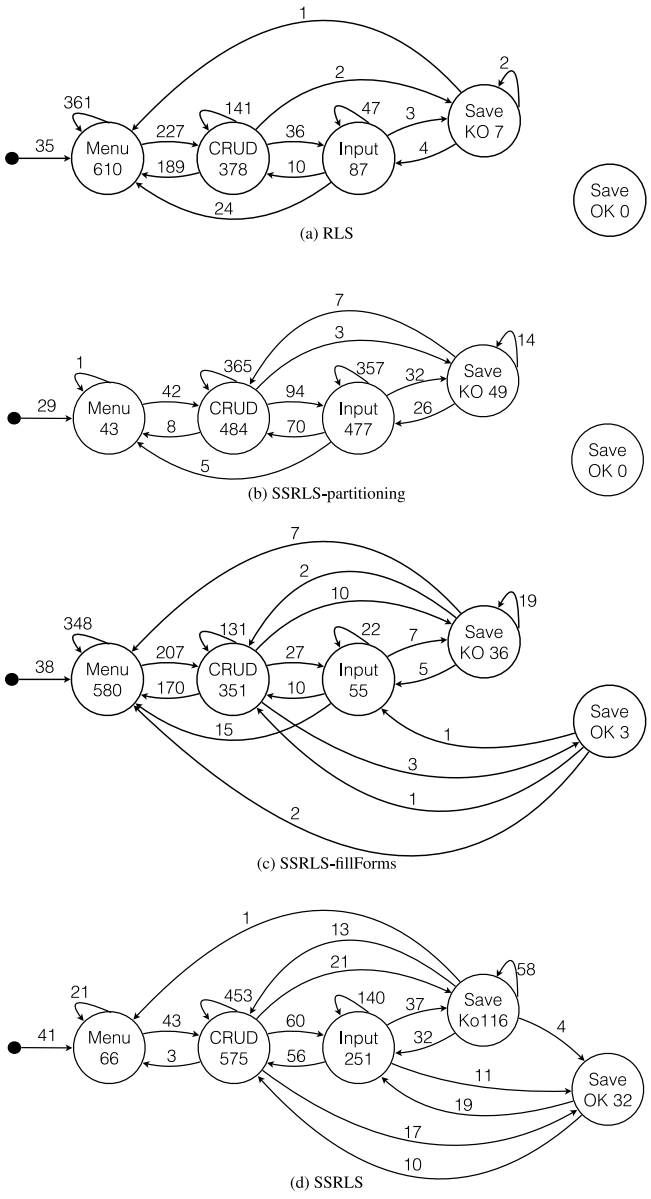


Fig. 2. Execution frequency of interaction sequences of different length.

We study the impact of the individual optimizations introduced in this paper, by considering three configurations for SSRLS:

- *SSRLS-partitioning* that uses only the strategy that restricts the access to the menus; it corresponds to Algorithm 1, but without the statements with a dark gray background;
- *SSRLS-fillForms* that uses the complex action specifically designed to deal with forms; it corresponds to Algorithm 1, but without the statements with a light gray background;
- *SSRLS* that exploits both optimizations; it corresponds to Algorithm 1 with both the statements with light and dark gray background.

We generated test cases in overnight sessions of six hours each, which is a practical timeframe for companies that use continuous testing solutions, using each of the four strategies: ABT

Table 1
Mean (m) and standard deviation (s) of the execution frequency for the classes of actions.

Strategy	Menu		CRUD		Input		SaveOK		SaveKO	
	m	s	m	s	m	s	m	s	m	s
RLS	610	53	378	60	87	31	0	0	7	3
SSRLS-partitioning	43	13	484	210	477	38	0	0	49	22
SSRLS-fillForms	580	106	351	103	55	37	3	4	36	23
SSRLS	66	19	575	170	251	105	32	10	116	40

Table 2
Wilcoxon (paired, one tail) test.

Alternative HP that RLS < …	Menu	CRUD	Input	SaveOK	SaveKO
SSRLS-partitioning	0.98	0.16	**0.03**	1	**0.03**
SSRLS-fillForms	0.71	0.82	0.98	0.05	0.05
SSRLS	1	**0.03**	**0.03**	**0.03**	**0.03**

The Wilcoxon (paired, one tail) test indicates the statistical significant values in support of the alternative hypothesis that the strategies SSRLS-partitioning, SSRLS-fillForms and SSRLS improve on RLS for the classes of actions.

with RLS, and ABT with the three variants of the SSRLS strategy. Fig. 2 illustrates the actions executed with the different test suite as finite state models that show the sequences of GUI actions that the different test suites execute. The states identify the type of executed action, and the transitions represent consecutively executed action types. The weights of the states indicate how often the test cases executed that action type, and the weights of the transitions indicate how often the test suite executes consecutive action pairs. For instance, the weight 610 of state Menu in Fig. 2(a) indicates that the RSL test suite executed Menu actions 610 times, and the weight of the transition from state Menu to state CRUD indicates that the test suite executes CRUD immediately after Menu actions 227 times. Table 1 completes the data in Fig. 2 by reporting the mean (columns *m* – values reported in Fig. 2) and the standard deviation (columns *s*) of the execution frequency for the four considered strategies (rows *RSL*, *SSRLS-partitioning*, *SSRLS-fillFroms*, *SSRLS*) and five class of actions (columns *Menu*, *CRUD*, *Input*, *SaveOK*, *SaveKO*).

Fig. 2(a) shows the limitations of the RLS strategy, limitations that we discussed in Section 3: RSL test cases execute many irrelevant length 1 sequences of *Menu* actions, and miss many relevant functional behaviors that depend on long interaction sequences. RLS test cases execute an average of $610 + 378 + 87 + 7 = 1,082$ actions, more than half of which (610) exercise only irrelevant length 1 sequences of *Menu* actions.

Fig. 2(a) also indicates that the longer it is the sequence of actions required to execute an action of a given type, the lesser RLS test suites execute that type of action. RLS test suites executes *CRUD* and *Input* actions, which (in *ERP*) depend on interaction sequences of length at-least 2, a mean of 378 and 87 times, respectively, *SaveKO* actions, which depend on interaction sequences of length at-least 3, only 7 times, and never executes *SaveOK* actions, which depend on sequences of length at-least 8. Table 1 shows that the mutual strength of execution frequencies remains stable across the considered classes of actions, despite the observation that the data related to each class of actions are subjected to some variance across the repetition of the RLS experiment.

The poor results of the RLS test suites depend on the GUI structure of the subject application, a structure that is misleading for the RLS strategy that executes many actions that seldom produce significant computations. In fact, we observe that (i) RSL reinforcement learning assigns high reward values to interactions that cause a strong discontinuity in the GUI state, (ii) selecting one of the main data entries (top-level menu buttons) reaches windows that strongly differ from the currently visualized ones, (iii) the main data entries (top-level menu buttons) are always executable in the GUI of the subject application (see for example in Fig. 1). As a consequence, reinforcement learning always favors selecting the main data entries (top-level menus buttons) over other actions, thus wasting most of the testing budget in interactions with the top-level menus buttons, and missing most of the application logic.

Figs. 2(b) and (c) show the individual contributions of the two SSRLS systematic components. SSRLS-partitioning largely reduces the time that RLS wastes in interacting with top-level menus (SSRLS-partitioning visits the Menu state 43 times compared to 610 RLS visits) in favor of a high rate of SaveKO Actions (SSRLS-partitioning visits the SaveKO state 49 times compared to 7 RLS visits). Although SSRLS-partitioning improves over RLS, SSRLS-partitioning test suites still miss many functionalities that require long interaction sequences: SSRLS-partitioning test suites execute *SaveKO* actions that depend on interaction sequences up to length 3, but not *SaveOK* actions that depend on longer sequences.

SSRLS-fillForms also improves over RLS, but is still quite ineffective in executing actions that require long interaction sequences. In fact, SSRLS-fillForms test suites waste a lot of time interacting with menus (they visit the Menu state 580 times) and rarely completes longer interaction sequences (they reach *SaveKO* and *saveOK* states 36 and 3 times, respectively). However, SSRLS-fillForms test suites improve over both RLS and SSRLS-partitioning with respect to executing *saveOK* actions, thanks to being specifically designed to interact with forms. The analysis of the 3 cases in which SSRLS-fillForms executed *saveOK* actions indicates that it can successfully execute operations that require interaction sequences up to length 5 in *ERP*.

Fig. 2(d) shows that by combining the two optimization strategies, the test suite executes many relevant application actions with a significant amount of valid and invalid cases: SSRLS test cases execute *saveKO* and *saveOK* states 116 and 32 times, respectively, thus showing a reasonable capability to complete long and useful interaction sequences. The two strategies are synergetic: the SSRLS-partitioning strategy reduces the time wasted in executing action sequences with only menu actions thus enabling the SSRLS-fillForms strategy explore long and relevant interaction sequences. Table 1 further supports that SSRLS is the only strategy that can steadily complete interaction sequences that lead to execute *saveOK* actions.

We used the (paired, one tail) Wilcoxon test to evaluate the statistical significance of the alternative hypothesis that any of the strategies SSRLS-partitioning, SSRLS-fillForms and SSRLS improve the execution frequency of each action class with respect to RLS, based on the results of our experiments. Table 2 reports the *p*-values of each test, with reference to a considered action class (in the columns of the table) and strategy (in the rows). *p*-values lower than 0.05 (in bold in the table) indicate statistical significant improvements. The table indicates that none of the strategies overcomes RLS with respect to the execution frequency of *Menu* actions. This is expected and not particularly relevant, because the execution frequency of *Menu* actions is reasonably high with any strategy, and interacting with menu items does not imply exercising relevant functionalities per se. The relevant observation is the substantial improvement of the SSRLS strategy over RLS with respect to the actions of all other classes, and for *CRUD* and *SaveOK* actions when compared to SSRLS-partitioning and SSRLS-fillForms.

The overall results show that SSRLS can execute not only operations that require short interaction sequences, such as deleting entities, but also operations that require long interaction sequences, such as filling-in complex form, thus significantly outperforming RLS.

RQ2: Does SSRLS contribute to satisfy the test objectives*?*

We measure the thoroughness of SSRLS test suites as the percentage of test objectives that test cases satisfy over the test objectives in the test plan that the testers of our industrial partner provided to us. The test objectives in the test plan represent the set of behaviors that are relevant to test according to the test engineers of the application.

Table 3 reports (i) the functional areas of the application (column *functional area*), where a functional area is a collection of functionalities designed to handle a same type of data entities, (ii) the number of test objectives per functional area (column *Test Objectives*), (iii) the number and percentage of test objectives that SSRLS and RLS test suites executed (columns *Satisfied w/ SSRLS* and *Satisfied w/ RLS*), respectively.

The test engineers identified 350 test objectives, the RLS and SSRLS test cases exercised 42% and 72% of the test objectives, respectively. This result is a clear indicator of the progresses of SSRLS over RLS. The higher effectiveness of the SSRLS strategy over the RLS strategy is confirmed in every functional area. In our project we observed that, while test engineers were reluctant in considering the original version of *ABT* based on RLS to automatically generate test suites, due to the relatively low coverage of the test plan that they considered insufficient to improve the overall costs of the testing tasks, they warmly welcomed $ABT_{2.0}$ that achieves reasonable functional adequacy figures thanks to SSRLS.

Overall, the results of the case study show that SSRLS is significantly more effective than RLS: it fosters the exploration of deeper regions of the execution space, and generates test cases that cover a significant portion of the functional logic of the application under test.

5. Test reports

In Section 3 we discussed two important challenges related to the inspection of the output generated by the automatic tests: the need to support the semi-automatic validation of the test oracles specifically designed to detect relevant classes of failures in the target application domain, and the need to facilitate the evaluation of the test cases with respect to the test requirements of the applications under test. Solving these challenges is crucial to make *ABT* effective in industrial testing processes.

In this section, we present the new test reports that $ABT_{2.0}$ can produce and we discuss their ability to assist test analysts in the validation of the test outcomes. We first exemplify the structure of a sample test report, and discuss the design and the structure of the test reports in detail, then we present early empirical results on the effectiveness of the test reports.

5.1. Structure and content of the test reports

$ABT_{2.0}$ generates the new test reports in tabular form for the sake of readability. The test reports indicate, for each generated test case, the actions executed on the GUI and the corresponding effect on both the GUI and the database. For example, Fig. 3 shows a test report that refers to a test case generated with $ABT_{2.0}$ for the application presented in Section 3.1. Each row in the table represents an operation that has been executed with: a sequential number that identifies the operation in the scope of the test case (column *ID*); the actions executed on the GUI (column *Actions*); the effect (i.e., the changes) produced by the operation on the GUI and the database (column *Outputs*). Each individual change reported in the output starts with a tag that indicates whether the item that has been changed is an item in the GUI (tag *GUI*) or in the database (tag *DB*).

In detail, the test report in Fig. 3 specifies the flow of test case *T3*, which starts by selecting the graphical menu *Invoices* (*T3.1–Actions*), continues by clicking on the button *New invoice* (*T3.2–Actions*) that has appeared in the GUI as a result of the previous operation (*T3.1–Outputs*), and then concludes by filling out and saving (*T3.3–Actions*) the input form visualized at the previous step (*T3.2–Outputs*), which in the end causes the insertion of a new record in the database table *INVOICES* (*T3.3–Outputs*).

A test report like this one has four main qualities:

Intelligible: The descriptions reported under the column *Actions* consist of labels that represent high-level domain operations. In some cases, multiple GUI actions can be grouped into a same logical Action to favor conciseness and abstraction, and produce reports that can be more easily interpreted by testers.

Easy to Validate: The descriptions reported under the column *Output* are designed to include the items that usually testers want to check to detect failures, while balancing completeness and conciseness.

Use of Domain Terminology: Test reports favor the use of domain terminology to low level technical terminology to simplify the comprehension of the reports and the comparison of the test results with the test oracles.

Efficient to Inspect: The structure of the test report facilitates the matching between the test results and the corresponding test requirements.

To achieve these goals, $ABT_{2.0}$ exploits domain specific information in several ways. To generate test reports that are concise and *intelligible*, $ABT_{2.0}$ groups consecutive GUI actions executed on input fields as part of a same logical operation, thus acknowledging the common characteristic of business oriented applications that rely on input forms to gather the inputs required for their operations. For example, with reference to the test report in Fig. 3, executing the operation *T3.3*, which adds a new Invoice, requires first filling in six input fields and then clicking on the *Save* button. The report collapsed all these fine-grained and correlated actions into a single action of the test report.

To generate test reports that are *easy to validate*, $ABT_{2.0}$ documents the changes that occur in the GUI and in the database during the execution of the test cases. These events are currently detected and reported as specified in the column *Outputs* of Table 4. Column *Source* indicates the GUI widget or the database event that may originate an entry in the test report. Column *Outputs* lists the information that is extracted from the element indicated in column *Source* (each variable is assigned with the value specified at the right of the ← symbol). Column *Format* specifies the entry that is added to the test report. Note that the entry is parametric and some values are filled in with the information extracted at runtime during test case execution.

The extracted data generally consist of the values stored on the titles and the body of the most common types of widgets, as well as the changes produced by database operations. Our experience with industrial business oriented applications suggests that the above output data is in most of the cases sufficient to validate business oriented applications against several classes of failures. The collected output data can be clearly adapted to the specific needs of an application.

We remark that in general, as the data in the test reports are intentionally incomplete, validating some test oracles may require manual inspection of the results while replaying the generated test cases. For instance, to avoid the excessive bloating of the reports in the presence of execution states that may contain

Table 3
Coverage of the test plan.

Functional area	Test objectives (#)	Satisfied w/ SSRLS (#)	Satisfied w/ RLS (#)
Projects	73	51 (70%)	18 (25%)
Orders	119	82 (69%)	39 (33%)
Invoices	52	32 (62%)	29 (56%)
Tickets	21	20 (95%)	18 (90%)
Modules	10	9 (90%)	6 (67%)
Offers	75	57 (76%)	38 (51%)
Total	350	251 (72%)	148 (42%)

Test report of test case T3

ID	Actions	Outputs
T3.1	Select menu "Invoices"	GUI: Window "Invoices" in foreground GUI: Button "New Invoices" enabled GUI: Grid with columns "ID", "Name", "Data", "Action" as 3 items GUI: Button "View" enabled GUI: Button "Edit" enabled GUI: Button "Delete" enabled
T3.2	Click button "New Invoice"	GUI: Window "Invoice" in foreground GUI: Window "Resources" in background GUI: Button "Save" enabled GUI: Button "Close" enabled GUI: Text field "Invoice Number" as ⟨empty⟩ enabled GUI: Text field "Invoice Name" as ⟨empty⟩ enabled GUI: List field "State" ("not Sent", "Sent", "Replied") as "Sent" enabled GUI: Calendar "Date" as 05/06/2015 enabled GUI: Text field "Client Data - Name" as ⟨empty⟩ enabled GUI: Text field "Client Data - Surname" as ⟨empty⟩ enabled GUI: Text field "Client Data - Email" as ⟨empty⟩ enabled GUI: Text field "Client Data - Country" as ⟨empty⟩ enabled
T3.3	Fill field "Invoice Number" as "2015.2" Fill field "Invoice Name" as "Payment" Fill field "Client Data - Name" as "Paul" Fill field "Client Data - Surname" as "Red" Fill field "Client Data - Email' as "paul@red.it" Fill field "Client Data - Country" as "Italy" Click button "Save"	DB: new record in Table INVOICES ⟨NUMBER=2015.2, LABEL=Payment, NAME=Paul, SURNAME=Red, EMAIL=paul@red.it, COUNTRY=Italy⟩

Fig. 3. A test report generated by $ABT_{2.0}$ for the case study application introduced in Section 3.

massive amounts of data, we decided to omit the content of data grids from the report and only include the titles of their columns. Oracles that check the content of the grid may require replaying a generated test case to manually inspect the grid.

To generate reports that *use the domain terminology*, $ABT_{2.0}$ describes the observed outputs with domain specific terms that facilitate the comprehension of the report, as shown by the formats reported in column *Format* of Table 4.

Finally, to generate reports that are *efficient to inspect*, $ABT_{2.0}$ produces browsable test reports. The browsing capability consists of the generation of a list with all the unique operations executed by the test cases included in a given test report, organized according to the graphical menus of the application. Fig. 4 shows the unique operation list produced by $ABT_{2.0}$ for a test report generated for the sample application considered in this paper. It shows the number of test cases executed for each graphical menu, shows the number of occurrences of each operation in the test cases, and lists the individual tests that executed each operation.

The unique operation list facilitates the tester in the inspection of the testing activity that has been automatically performed on each area of the application. The unique operation list in Fig. 4 shows that 15, 18, 27 and 21 test cases have been executed on the *Projects*, *Offers*, *Orders*, and *Invoices* menus, respectively. The operations in a same menu have been covered with different frequencies. For instance, the operation *Invoices.View* has been execute 12 times, while the operation *Invoices. Invoice* only twice. Finally, the unique operation list also indicates which operation of the test reports has executed a specific menu action. In the example, *Invoices.Save* has been executed 4 times in the 21 test cases that covered the *Invoices* menu. The *Invoices.Save* operation occurred as the third operation of test case *T3* (shown in Fig. 3), as the eighth operation of the test case *T6* and so forth. When clicking on the identifiers, testers are redirected to the corresponding item of the test report.

Since a test generator may generate many more test cases than the ones that could be manually inspected, the test reports tend to grow quickly. The unique operation list produced by $ABT_{2.0}$ facilitates testers in the inspection of the result produced by relevant operations, for instance by selectively inspecting the outputs produced by the execution of a same operation in many different test cases. The unique operation list also provides a concise reference to quickly identify the functional logic missed by the test generator and optimize the allocation of additional testing effort.

The implementation of the test report functionality required three main extensions to ABT: the ability to track GUI output data, the ability to track database changes, and the ability to generate browsable reports. We implemented the ability to track output data by simply adapting the GUI exploration component of $ABT_{2.0}$. We implemented the ability to track database changes by adding a monitoring layer that relies on the standard change tracking functionality available in database servers.[5] When change tracking is active, the database server records change data in additional tables of the database, and $ABT_{2.0}$ simply retrieves data from

[5] The interested reader may refer to the change tracking functionality of Microsoft SQL Server documented at https://technet.microsoft.com/en-us/library/cc280462(v=sql.105).aspx [Last read February 2016].

Table 4
Outputs in the test reports.

Source	Outputs	Format
Graphical menu	$\langle M \rangle \leftarrow$ the menu title label $\langle S \rangle \leftarrow$ the menu state *enabled*/*disabled*	GUI: Menu $\langle M \rangle$ in state $\langle S \rangle$
Button	$\langle B \rangle \leftarrow$ the button title label $\langle S \rangle \leftarrow$ the button state *enabled*/*disabled*	GUI: Button $\langle B \rangle$ in state $\langle S \rangle$
Text field	$\langle T \rangle \leftarrow$ the field title label $\langle I \rangle \leftarrow$ the initial value, if any, or $\langle empty \rangle$ $\langle S \rangle \leftarrow$ the field state *editable*/*blocked*	GUI: Text field $\langle T \rangle$ as $\langle I \rangle$ in state $\langle S \rangle$
List field	$\langle L \rangle \leftarrow$ the field title label $\{\langle V \rangle\} \leftarrow$ the possible values $\langle I \rangle \leftarrow$ the initial value $\langle S \rangle \leftarrow$ the field state *selectable*/*blocked*	GUI: List field $\langle L \rangle$ with values $\{\langle V \rangle\}$ as $\langle I \rangle$ in state $\langle S \rangle$
Combo-box field	$\langle C \rangle \leftarrow$ the field title label $\{\langle V \rangle\} \leftarrow$ the markable values $\{\langle I \rangle\} \leftarrow$ the initial values, if any, or $\langle empty \rangle$ $\langle S \rangle \leftarrow$ the field state *selectable*/*blocked*	GUI: Combo-box field $\langle C \rangle$ with values $\{\langle V \rangle\}$ marked at $\{\langle I \rangle\}$ in state $\langle S \rangle$
Data grid	$\{\langle C \rangle\} \leftarrow$ the column title labels	GUI: Grid with columns $\{\langle C \rangle\}$
Window	$\langle W \rangle \leftarrow$ the window title label $\langle S \rangle \leftarrow$ window in *foreground*/*background*	GUI: Window $\langle W \rangle$ in $\langle S \rangle$
Database insert	$\langle T \rangle \leftarrow$ the table name $\langle R \rangle \leftarrow$ the inserted record	DB: new record in table $\langle T \rangle$ as $\langle R \rangle$
Database delete	$\langle T \rangle \leftarrow$ the table name $\langle R \rangle \leftarrow$ the deleted record	DB: deleted record in table $\langle T \rangle$ was $\langle R \rangle$
Database update	$\langle T \rangle \leftarrow$ the table name $\langle R' \rangle \leftarrow$ the record before the update event $\langle R \rangle \leftarrow$ the updated record	DB: update in table $\langle T \rangle$ as $\langle R' \rangle \rightarrow \langle R \rangle$

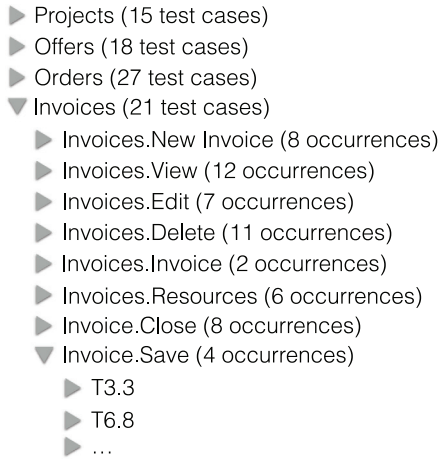


Fig. 4. The unique operation list of the test reports of $ABT_{2.0}$: excerpt out of a test report generated for the sample application of our case study.

these tables. We implemented the generation of the reports as the generation of Excel workbooks that mimic the format of the test plan that the test analysts are familiar with. The test report shows the data that refer to distinct graphical menus of the GUI on separate spreadsheets.[6] We designed the features for tracking GUI output data and database changes, and producing browsable reports, by taking advantage of the characteristics of the application domain. Although designed for the specific domain, these features exploit common characteristics, and can be easily adapted to other environments and processes. Independently for the generalizability of the overall approach, some components of the approach, such as grouping actions to foster intelligibility, can be easily reused in other contexts.

5.2. Effectiveness of the test reports

We experienced with the test reports of $ABT_{2.0}$ while working with the industrial business oriented application made available by our project partner. In this section we discuss the empirical results on the effectiveness of the test reports.

In the light of the challenges stated in Section 3, we evaluate the effectiveness of the test reports by investigating the following research questions:

- RQ3: Do test reports *suffice to verify* test oracles of business oriented applications?
- RQ4: Do test reports facilitate *scheduling additional testing activities*?

To answer RQ3, we compare the test oracles manually defined by the testers in the official test plan to the information tracked in the test reports produced by $ABT_{2.0}$, and we quantify the portion of oracles that can be verified uniquely based on the information in the reports.

To answer RQ4, we qualitatively discuss how the test reports that $ABT_{2.0}$ generates support the definition of additional testing tasks to increase functional coverage.

5.2.1. RQ3: Do test reports suffice to verify test oracles of business applications?

We quantify the effectiveness of the test reports by measuring the number of oracles that belong to the test plan and that can be checked using the information in the test reports. In particular, we measure the number of *verifiable oracles*, that is the number of oracles defined in the test plan that can be at least potentially verified based on automatically generated test reports, and the number of *verified oracles*, that is the number of oracles that can be verified based on the actual test reports of our experiments.

[6] The possibility of formatting the reports on a per graphical menu basis is enabled by the SSRLS GUI exploration strategy described in Section 4 that fosters the explicit mapping between test cases and graphical menus.

Table 5
Test oracles satisfaction rates.

Functional area	Verifiable oracles (#)	Verified oracles (#)	
Projects	81	65	(80%)
Orders	132	100	(76%)
Invoices	56	41	(73%)
Tikets	38	28	(74%)
Modules	16	9	(56%)
Offers	85	67	(79%)
Total	408	310	(76%)

As we discuss in Section 3.1, the test plan classifies the test objectives by entity type. Fig. 5 shows an excerpt of the test plan that exemplifies a test objective for the *Invoice* entities. A test objective corresponds to a row in a spreadsheet with three fields (columns): the identifier of the test objective, the actions that the tester shall perform to satisfy the test objective, and the checks that the tester shall perform correspondingly. The checks represent the test oracles. In out experiments, we count each individual check as an oracle. With reference to the example in Fig. 5, to satisfy the test objective *8.3: Correctness of the form when adding new invoices*, the tester shall *click on the button "New Invoice"*, check that (i) *the foreground window is "Invoice"* and (ii) *The name of the fields are* as expected. Thus, we count two oracles for this test objective. In this case, these oracles are verifiable, since they can be crosschecked on the outputs of our test reports, such as, the test report in Fig. 3.

Table 5 shows the mean results obtained for the different functional areas encompassed in the test plan. The table reports both the number and percentage of verified oracles. The data indicate that we have been able to verify the large majority of the test oracles, that is, a total of 310 out of the 408 (76%) test oracles.

Investigating the missed oracles in further detail, we found that 13% (53 out of 408) of the oracles could not be verified because they map on execution data that $ABT_{2.0}$ does not currently track in the reports, that is, content of data grids (35 oracles), graphical attributes, such as the color of GUI widgets (6 oracles), and database data not involved with change events (12 oracles). These oracles may indicate directions to extend the scope of the tracked data, though the extensions must be carefully evaluated with respect to the balance between thoroughness and conciseness, which we have discussed as an important aspect of the reports. The remaining 11% (45 out of 408) of the oracles express test requirements that map on execution data that do not explicitly belong to the GUI or to the database, such as verifying that an email has been sent or that some data have been written in a file. Although the test oracles that could not be checked directly using the test reports require additional testing effort to replay the test cases generated with $ABT_{2.0}$ and manually check their results, we believe that this could be an acceptable cost if limited to a small percentage of the automatic tests, as in our case where only 24% of the test oracles cannot be checked directly from the reports. Note that the number of test cases that must be replayed is often significantly smaller than the number of test oracles that have not been checked because a single test case may be used to check multiple test oracles.

5.2.2. RQ4: Do test reports facilitate scheduling additional testing activities

To answer this research question, we qualitatively evaluated the cost of pairing the results in the test reports with the corresponding test requirements in the test plan, so that additional activities could be suitably defined to cover the missed items. The procedure we experienced consists of the following steps: we scan the test plan sequentially and, for each test oracle defined in the plan, we exploit the browsing capability of the test reports to identify whether the operation referred in the oracle has been executed by some test cases generated with $ABT_{2.0}$. For each reached oracle, we further exploit the browsing capability to explode the list of test actions that executed the operation associated with the oracle, and used the first action in the list to access the output data tracked by $ABT_{2.0}$ after the execution of the test action and manually verify the oracle.

This procedure is similar to other types of methods proposed in the scientific literature to select representative samples out of the massive amount of test results that can be produced with a test generator, for example in contexts where testers need to inspect the generated test cases. For instance, a commonly used strategy is to retain only the test cases that cover additional branches in the code, based on the order in which the test generator computes them (Tillmann and de Halleux, 2008; Braione et al., 2014). Similarly, our experimental procedure samples the test report by pairing the test results in the reports with the operations that map to test requirements in the test plan, and selecting the results that correspond to the first occurrence of each operation.

The results reported in the paper (see Tables 3 and 5) indicate that, out of the test objectives and the test oracles defined in the test plan, the test cases and the test reports that $ABT_{2.0}$ generates successfully exercise more than 70% of the test objectives, and allow for verifying more than 70% of the oracles related with these test objectives, respectively.

Inspecting the test reports we have been able to schedule the manual generation of a set of test cases necessary to cover the test objectives that have not been covered automatically, and a set of automatic test cases that need to be replayed to check the test oracles could not been verified from the data in the test report.

Overall, the test reports enabled the scheduling of new focused test activities with significantly reduced effort. In particular, we reduced the test effort from the *manual design, execution and inspection* of a set of test cases to cover the 350 test objectives in the test plan, and verify the corresponding test oracles, to:

- the *manual design, execution and inspection* of a set of test cases to cover only 99 test objectives, since the test cases that $ABT_{2.0}$ yields automatically already exercise 251 of the 350 test objectives in the test plan (Tables 3),
- the *manual verification* of 98 additional test oracles, but by means of test cases generated with $ABT_{2.0}$: the test cases generated with $ABT_{2.0}$ hit test objectives that relate to 408 oracles, and 310 of these oracles can be directly verified in the test reports (Table 5), thus only 98 oracles require to actually replay test cases,
- the validation of 310 test oracles by simply browsing the test reports that $ABT_{2.0}$ produces automatically.

Based on these data, we claim that $ABT_{2.0}$ has been able to significantly reduce the effort necessary to the design and execute the test cases, and also simplified the checking process for a large portion of the test oracles. We interpret these results as a positive indication of the possibility to optimize testing activities using a technique like $ABT_{2.0}$.

6. Lessons learned

In this section, we report the main insights that we gained with the project activity described in this paper. We believe these insights can be helpful to drive future research effort in system testing and its application to industrial contexts. We describe our insights as a list of lesson learned.

Test plan of application *ERP*		
Test objective identifier	**Actions**	**Checks**
...	...	...
8.3: Correctness of the form when adding new invoices	Click button "New Invoice"	- The foreground tab is "Invoice" - The name of the fields are as in Table 12 of the requirements
...	...	...

Fig. 5. Excerpt of the test plan of *ERP*: A sample test objective and corresponding checks (oracles).

- **Lesson Learned 1 — Automation is necessary but not sufficient**. Automation is highly appreciated in industry because it is a key factor to reduce development effort and costs. However, automation alone is insufficient to address the real needs of complex projects and large organizations. In particular, generating test cases that cover many of the functionalities in an application is useful only if testers can understand and interpret the testing activity that has been performed with respect to the test objectives. This is necessary to identify the functional areas that need to be further tested and checked manually (see the experience about test reports and oracles reported in Section 5). The only generation of tests and discovery of failures is useful but of limited value in industrial contexts, where the adequate validation of all the functionalities of an application is the priority.
- **Lesson Learned 2 — Domain-specific approaches might scale to industrial systems** Automatically testing an application without any information about its structure and semantics is extremely challenging, and outside the capability of current automatic system testing techniques. However, we can successfully exploit domain-specific characteristics to dramatically increase the effectiveness of testing tools and make them effective in specific domains. For instance, we can use EventFlowSlicer (Saddler and Cohen, 2017) that exploits application-specific knowledge provided by testers for generating effective test cases, or Augusto (Mariani et al., 2018) that exploits abstract domain knowledge to efficiently test some features of the target application. In our experience, we found useful to exploit the organization of the GUI in functional areas to make ABT more effective, up to the level of being useful in business oriented applications (see the extension to the GUI exploration strategy proposed in Section 4). In the future, domain specific solutions should receive greater attention from researchers and practitioners.
- **Lesson Learned 3 — Manually-specified oracles can dramatically increase the effectiveness of automated test cases** Automatically generated test cases use implicit oracles to detect failures, that is they can only detect crashes, uncaught exceptions and hangs [Barr et al., 2015] . The lack of powerful oracles is a major limit to failure detection. Our experience shown that the failure detection capability of the generated tests can be dramatically improved by manually specifying a few automatic oracles that can be checked at runtime (see results for RQ3). Our experience confirms Esbah, Deursen and Roest's results about the importance of human-defined oracles to improve the effectiveness of automated testing (Mesbah et al., 2012). Although this could sometime be perceived as an extra effort for test engineers, we noticed that once a tool is perceived as useful, for instance because it can automatically cover many test objectives that had to be covered manually in the past, industry people are willing to invest their effort in the definition of program oracles that can increase the effectiveness of the synthesized test cases. Although our experience has been positive, identifying proper classes of system-level oracles that can be exploited by automatically generated test cases still deserves further research.
- **Lesson Learned 4 — Cost effective definition of system-level automated oracles is still an open challenge** While there are many languages and approaches to define unit level oracles, such as program assertions and invariants, there is a lack of languages suitable for system level oracles. Capture and replay tools usually support the concept of checkpoint (IBM, 2021), some recent studies investigate the manually specification of automatic oracles (Mesbah et al., 2012; Memon, 2007), but there is no complete approach to conveniently and cost-effectively specifying system-level oracles . In fact, specifying an oracle for a test case requires either the execution of a complex sequence of interactions with the system under test or the definition of complex expressions that refer to GUI widgets. Our experience confirmed that designing system level oracles could be cumbersome. Defining more effective specification methods for the definition of system level oracles is an open challenge for the future.
- **Lesson Learned 5 — Inflexible outputs might be a barrier to tool adoption, regardless effectiveness** In our experience, we obtained the most from ABT once it has been integrated with the testing process of the organization ($ABT_{2.0}$). To enable the integration, it has been of critical importance to produce browsable outputs that could be exploited to guide the definition of the test strategy and plan the test effort. Without this integration, ABT would not have been considered for adoption. To make the system testing technology successful is thus important to produce solutions that can flexibly integrate with an organization process, also in terms of their input/output behavior and reporting capabilities.

7. Threats to validity

The main threats to internal validity of the results reported in this paper concern the procedures to collect and analyze data. While the tool was experimented in an industrial context by professional developers and the data are the direct consequence of such activities, the authors of the paper collected and analyzed the data. To mitigate the possibility of introducing any bias in this process, we defined the procedures and the analyses as objectively as possible, severely limiting subjective judgment. The paper describes the processes to allow third parties to replicate the process for different tools and subject applications.

Due to constraints imposed by the industrial context, it was impossible to replicate each experiment more than 5 times per configuration. Although a higher number of repetitions may generate results with higher stability, we observed an already good level of stability of the results across executions, so this limitation is not likely to affect the main conclusions of our study.

The nature of our study introduces some straightforward threads to the external validity of the results. Our goal was to investigate the adoption of *ABT* in an industrial scenario, and the experience reported in the paper is specific to the considered context: the company, the test case generation tools, and the software products. Still, our experience generated interesting insights that combined with similar studies can create a useful knowledge about the challenges that must be faced when a GUI system testing technology is used in industrial projects.

8. Related work

System test case generation techniques can automatically generate test cases that stimulate an application under test using its GUI. These approaches can address a range of platforms and GUI technologies, including desktop, Web, and mobile applications.

The test case generation process can be guided by different strategies. Several techniques are model-based, that is they first generate a state-based model of the GUI of the system under test and then generate test cases that cover the model according to a criterion. The model is usually extracted by ripping the GUI of the application (Yuan and Memon, 2010; Xun et al., 2011), and the test cases can be generated according to different criteria, such as covering sequences of events (Memon and Xie, 2005), sequences of interacting events (Xun et al., 2011), or data-flow relations among event handlers (Arlt et al., 2012).

Other techniques use search-based solutions to generate test cases that are optimal according to a given goal, for example satisfying a code coverage criterion (Gross et al., 2012). Other approaches simply generate test cases randomly or by combining random choices with strategies that influence randomness (Mariani et al., 2014; Bertolini et al., 2010). Our research prototype, *ABT*, uses Q-learning to steer the testing activity, which would be random otherwise, towards the most interesting areas of the application. In this paper, we have discussed our experience with introducing ABT in the software process of a medium size company that develops software for third parties. We selected *ABT* because we wanted to avoid model-based solutions, which can generate many infeasible test cases when the functionalities under test require long interaction sequences to be executed (Bae et al., 2014), like several functionalities in our commercial system. Moreover, among the techniques that do not generate tests by covering a model, our past results suggested that ABT was an effective solution (Mariani et al., 2014, 2012).

So far, only few studies considered commercial applications and the integration of automatic GUI testing solutions with the development process of a professional organization. A study that considers commercial software is the one by Bertolini et al. (2010) where the fault-revealing effectiveness of several GUI testing techniques is compared using Motorola smartphones as subject applications. Want et al. also empirically investigate the effectiveness of several Android testing techniques with a number commercial apps (Wang et al., 2018a). These studies focus on the comparison among techniques and do not consider the issue of introducing these techniques into the production process of an organization. Contrarily, our experience reports challenges and insights about the industrial exploitation of a GUI testing solution. Although our observations cannot be generalized to every organization and every application, and data from many other similar experiences are necessary to better understand the difficulties of introducing automatic GUI testing in industry, the experience reported in this paper represents a first step towards understanding the relationship between industry and automatic GUI testing.

In our experience we faced both challenges that are well-known to the scientific community and challenges that gained little attention so far. In particular, we faced the problem of dealing with the explosion of the execution space, which is a problem present in almost every non-trivial application, but is exacerbated by the size and structure of commercial applications. We found that exploiting explicit information about the GUI and the structure of the application under test might improve the scalability of existing approaches, as reported also by Saddler and Cohen (2016).

We also faced the oracle problem (Barr et al., 2015), which is a hot research topic. While research is mostly focusing on automatically generating program oracles (Carzaniga et al., 2014), in our experience we realized that manually specifying the oracles might be cost-effective, but we also realized the lack of languages and approaches to cost-effectively specify them.

Finally, the effective integration with the development process requires tools that can both produce proper outputs and suitably document the performed activity. The work about automatically documenting test cases focused on the generation of code-level documentation for unit test cases (Panichella et al., 2016; Li et al., 2016). In this work, we explored the generation of reports that can be easily interpreted by testers in terms of functional requirements that are/are not adequately tested. The solution that we defined is deemed relatively cost-effective in the specific context of our experimentation. Indeed, designing automated approaches that further improve efficiency and efficacy is an open challenge that deserves great attention in the future.

9. Conclusions

In this paper, we describe our experience in introducing a leading-edge research technology for automatically generating system tests, in a business oriented organization, by discussing the introduction of *ABT* for automatically testing a business application. As a result of our experience we identify several open challenges that must be faced to effectively address large industrial applications. The most relevant ones are (i) scalability, that is automatic system testing techniques must scale to large applications composed of many windows and functionalities that require complex input data to be executed, (ii) reporting, that is automatic system testing techniques must generate test reports that can be easily interpreted according to the test requirements of the application, and (iii) oracles, that is automatic system testing must be able to detect wrong outputs in addition to crashes.

Our experience indicates that it is possible to tailor effective system testing solutions to the characteristics of the application under test. In our industrial context, we exploited information about the structure of both the GUI and the test plan to extended *ABT* to cope with the identified challenges, leading to the development of $ABT_{2.0}$. Our results show that $ABT_{2.0}$ can reduce the effort necessary to test the system, by overcoming the main limitations that we faced with the original version of *ABT*. We illustrate the elements that led us identifying and implementing the improvements needed for introducing the approach in production, and discuss the open challenges towards a routinely approach for automating the generation of system-level test suites.

An exploratory case study is a preliminary step towards general solutions, which serves to generate research hypotheses for specific and focused causal research. Thus, we do not claim that our current results directly generalize to all contexts. Nonetheless, the experience that we report in this paper provides important empirical evidence about effective automation of system testing in industrial settings. The industrial study that we discuss in the paper indicates that oracles, efficient exploration of the interaction space, and generation of useful reports are critical enabling factors for transferring leading-edge approaches for automatic test case generation into the production line. The solutions that we developed to address these issues pave the way towards architecting industrial-strength system test case generation approaches.

Currently our industrial partner can autonomously run $ABT_{2.0}$ on the ERP application considered in the study, even though they can hardly address new applications without our support, due to the difficulty in setting up new application-specific configurations and deploying the tool in the context of new software projects. Based on the results of the experience that we discuss in this paper, we are now working with an industrial partner to productize

$ABT_{2.0}$, which is still in a prototype stage and needs to be properly embedded in a commercially usable solution.

Our current research agenda aims to study new solutions towards tailorable system testing approaches that can be easily adapted to the characteristics of specific classes of applications. We are collaborating with new industrial partners to collect additional evidence of the effectiveness of the approach discussed in this paper with new business oriented applications. We aim to investigate the practicality of the tool and the actionability of the generated outputs for different scenarios, aiming to assess the general validity of the results reported in this paper. We are also actively conducting research on developing effective automatic test generation approaches that address scalability, reporting and oracles.

CRediT authorship contribution statement

Matteo Brunetto: Conceptualization, Software, Validation, Investigation, Data curation, Visualization. **Giovanni Denaro:** Conceptualization, Methodology, Validation, Investigation, Data curation, Writing - original draft, Writing - review & editing, Supervision, Project administration, Funding acquisition. **Leonardo Mariani:** Conceptualization, Methodology, Validation, Investigation, Data curation, Writing - original draft, Writing - review & editing, Supervision, Project administration, Funding acquisition. **Mauro Pezzè:** Conceptualization, Methodology, Validation, Investigation, Data curation, Writing - original draft, Writing - review & editing, Supervision, Project administration, Funding acquisition.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

This work has been partially supported by the H2020 European Research Council Proof of Concept project AST (grant agreement n. 824939) and by the SISMA national research project, Italy (MIUR, PRIN 2017, Contract 201752ENYB).

References

Almasi, M.M., Hemmati, H., Fraser, G., Arcuri, A., Benefelds, J., 2017. An industrial evaluation of unit test generation: Finding real faults in a financial application. In: Proceedings of the 39th International Conference on Software Engineering: Software Engineering in Practice Track. ICSE-SEIP, IEEE Press, pp. 263–272. http://dx.doi.org/10.1109/ICSE-SEIP.2017.27.

Arcuri, A., 2019. Restful api automated test case generation with evomaster. ACM Trans. Softw. Eng. Methodol. 28 (1), http://dx.doi.org/10.1145/3293455.

Arlt, S., Podelski, A., Bertolini, C., Schaf, M., Banerjee, I., Memon, A.M., 2012. Lightweight static analysis for gui testing. In: Proceedings of the International Symposium on Software Reliability Engineering. ISSRE '12, IEEE Computer Society, pp. 301–310.

Bae, G., Rothermel, G., Bae, D.H., 2014. Comparing model-based and dynamic event-extraction based gui testing techniques. J. Syst. Softw. 97 (C), 15–46.

Baluda, M., Denaro, G., Pezzè, M., 2015. Bidirectional symbolic analysis for effective branch testing. IEEE Trans. Softw. Eng. 42 (5), 403–426.

Banerjee, I., Nguyen, B., Garousi, V., Memon, A., 2013. Graphical user interface (gui) testing: Systematic mapping and repository. Inf. Softw. Technol. 55 (10), 1679–1694. http://dx.doi.org/10.1016/j.infsof.2013.03.004.

Barr, E.T., Harman, M., McMinn, P., Shahbaz, M., Shin, Y., 2015. The oracle problem in software testing: A survey. IEEE Trans. Softw. Eng. 41 (5), 507–525.

Becce, G., Mariani, L., Riganelli, O., Santoro, M., 2012. Extracting widget descriptions from guis. In: Proceedings of the International Conference on Fundamental Approaches to Software Engineering. FASE, In: LNCS, Springer.

Bertolini, C., Mota, A., Aranha, E., Ferraz, C., 2010. Gui testing techniques evaluation by designed experiments. in: proceedings of the third international conference on software testing, verification and validation. icst.

Blasi, A., Goffi, A., Kuznetsov, K., Gorla, A., Ernst, M.D., Pezzè, M., Castellanos, S.D., 2018. Translating code comments to procedure specifications. In: Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis. ISSTA 2018, pp. 242–253. http://dx.doi.org/10.1145/3213846.3213872.

Böttger, K., Schwitter, R., Aliod, D.M., Richards, D., 2001. Towards reconciling use cases via controlled language and graphical models. In: 14th International Conference on Applications of Prolog, Web Knowledge Management and Decision Support. INAP 2001, pp. 115–128. http://dx.doi.org/10.1007/3-540-36524-9_10.

Braione, P., Denaro, G., Mattavelli, A., Pezzè, M., 2017. Combining symbolic execution and search-based testing for programs with complex heap inputs. In: Proceedings of the 26th International Symposium on Software Testing and Analysis. ISSTA '17, ACM.

Braione, P., Denaro, G., Mattavelli, A., Vivanti, M., Muhammad, A., 2014. Software testing with code-based test generators: data and lessons learned from a case study with an industrial software component. Softw. Qual. J. 311–333.

Braione, P., Denaro, G., Pezzè, M., 2013. Enhancing symbolic execution with built-in term rewriting and constrained lazy initialization. In: Proceedings of the 9th Joint Meeting on Foundations of Software Engineering. ESEC/FSE 2013, ACM, pp. 411–421.

Briand, L.C., Labiche, Y., Sun, H., 2003. Investigating the use of analysis contracts to improve the testability of object-oriented code. Softw. Pract. Exp. 33 (7), 637–672. http://dx.doi.org/10.1002/spe.520.

Carzaniga, A., Goffi, A., Gorla, A., Mattavelli, A., Pezzè, M., 2014. Cross-checking oracles from intrinsic software redundancy. In: Proceedings of the International Conference on Software Engineering. ICSE '14, ACM, pp. 931–942.

Developers A. Monkey UI, 2021. http://developer.android.com/tools/help/monkey.html.

Ernst, M.D., Cockrell, J., Griswold, W.G., Notkin, D., 1999. Dynamically discovering likely program invariants to support program evolution. In: Proceedings of the 21st International Conference on Software Engineering. ICSE '99, ACM, New York, NY, USA, pp. 213–224. http://dx.doi.org/10.1145/302405.302467.

Ertel, W., Schneider, M., Cubek, R., Tokic, M., 2009. The teaching-box: A universal robot learning framework. In: Proceedings of the International Conference on Advanced Robotics.

Fraser, G., Arcuri, A., 2013. Whole test suite generation. IEEE Trans. Softw. Eng. 39 (2), 276–291.

Fraser, G., Staats, M., McMinn, P., Arcuri, A., Padberg, F., 2015. Does automated unit test generation really help software testers? a controlled empirical study. ACM Trans. Softw. Eng. Methodol. 24 (4), 23:1–23:49.

GmbH, T., 2020. Tosca Manual 13.3. Tricentis GmbH.

Gross, F., Fraser, G., Zeller, A., 2012. Search-based system testing: high coverage, no false alarms. In: Proceedings of the International Symposium on Software Testing and Analysis. ISSTA.

Gutmann, P., 2016. Fuzzing code with AFL. ;login: 41(2). URL https://www.usenix.org/publications/login/summer2016/gutmann.

IBM, 2021. Ibm rational functional tester. . https://www.ibm.com/products/rational-functional-tester.

Lago, P., Muccini, H., van Vliet, H., 2009. A scoped approach to traceability management. J. Syst. Softw. 82 (1), 168–182. http://dx.doi.org/10.1016/j.jss.2008.08.026.

Li, B., Vendome, C., Linares-Vasquez, M., Poshyvanyk, D., Kraft, N.A., 2016. Automatically documenting unit test cases. In: 2016 IEEE International Conference on Software Testing, Verification and Validation. ICST. pp. 341–352.

Lorenzoli, D., Mariani, L., Pezzè, M., 2008. Automatic generation of software behavioral models. In: Proceedings of the 30th International Conference on Software Engineering. ICSE '08, ACM, New York, NY, USA, pp. 501–510. http://dx.doi.org/10.1145/1368088.1368157.

Mariani, L., Pezzè, M., Riganelli, O., Santoro, M., 2011. Autoblacktest: a tool for automatic black-box testing. In: Proceedings of the 33rd International Conference on Software Engineering. ICSE '11. pp. 1013–1015.

Mariani, L., Pezzè, M., Riganelli, O., Santoro, M., 2012. Autoblacktest: Automatic black-box testing of interactive applications. In: Proceedings of the International Conference on Software Testing, Verification and Validation. ICST '12, IEEE Computer Society, pp. 81–90.

Mariani, L., Pezzè, M., Riganelli, O., Santoro, M., 2014. Automatic testing of gui-based applications. Softw. Test. Verif. Reliab. 24 (5), 341–366.

Mariani, L., Pezzè, M., Zuddas, D., 2018. Augusto: Exploiting popular functionalities for the generation of semantic gui tests with oracles. In: Proceedings of the 40th International Conference on Software Engineering. ICSE '18, ACM, pp. 280–290. http://dx.doi.org/10.1145/3180155.3180162.

Memon, A., 2007. An event-flow model of gui-based applications for testing: Research articles. Softw. Test. Verif. Reliab. 17 (3), 137–157.

Memon, A.M., E, M.P., Soffa, M.L., 2001. Hierarchical gui test case generation using automated planning. IEEE Trans. Softw. Eng. 27 (2), 144–155. http://dx.doi.org/10.1109/32.908959.

Memon, A.M., Xie, Q., 2005. Studying the fault-detection effectiveness of gui test cases for rapidly evolving software. IEEE Trans. Softw. Eng. 31 (10), 884–896.

Mesbah, A., v Deursen, A., Roest, D., 2012. Invariant-based automatic testing of modern web applications. IEEE Trans. Softw. Eng. 38 (1), 35–53.
Meyer, B., 1988. Eiffel: A language and environment for software engineering. J. Syst. Softw. 8 (3), 199–246. http://dx.doi.org/10.1016/0164-1212(88)90022-2.
Pacheco, C., Lahiri, S.K., Ernst, M.D., Ball, T., 2007. Feedback-directed random test generation. In: Proceedings of the International Conference on Software Engineering. ICSE '07, ACM, pp. 75–84.
Paiva, A., Tillmann, N., Faria, J., Vidal, R., 2005. Modeling and testing hierarchical guis. In: Proceedings of Abstract State Machines 2005. pp. 8–11.
Panichella, S., Panichella, A., Beller, M., Zaidman, A., Gall, H.C., 2016. The impact of test case summaries on bug fixing performance: An empirical investigation. In: Proceedings of the 38th International Conference on Software Engineering. ICSE, Association for Computing Machinery, New York, NY, USA, pp. 547–558. http://dx.doi.org/10.1145/2884781.2884847.
Parasoft, 2021. Parasoft c/c++ test. . https://www.parasoft.com/product/cpptest/.
Ramesh, B., Edwards, M., 1993. Issues in the development of a requirements traceability model. In: Proceedings of IEEE International Symposium on Requirements Engineering. pp. 256–259. http://dx.doi.org/10.1109/ISRE.1993.324849.
Saddler, J., Cohen, M.B., 2016. Eventflowslicer: Goal based test generation for graphical user interfaces. In: Proceedings of the International Workshop on Automating Test Case Design, Selection, and Evaluation. A-TEST.
Saddler, J.A., Cohen, M.B., 2017. Eventflowslicer: A tool for generating realistic goal-driven gui tests. In: Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering. ASE 2017, IEEE Press, pp. 955–960.
Spivey, J.M., 1989. The Z Notation: A Reference Manual. Prentice-Hall, Inc., Upper Saddle River, NJ, USA.
Sutton, R.S., Barto, A.G., 1998. Introduction to Reinforcement Learning, first ed. MIT Press.
Thummalapenta, S., Lakshmi, V.K., Sinha, S., Sinha, N., Chandra, S., 2013. Guided test generation for web applications. In: Proceedings of the International Conference on Software Engineering. ISE.
Tillmann, N., de Halleux, J., 2008. Pex: White box test generation for .NET. In: Proceedings of the International Conference on Tests and Proofs. TAP '08, Springer, pp. 134–153.
Wang, W., Li, D., Yang, W., Cao, Y., Zhang, Z., abd T. Xie, Y.D., 2018. An empirical study of android test generation tools in industrial cases. In: Proceedings of the International Conference on Automated Software Engineering. ASE.
Wang, C., Pastore, F., Briand, L.C., 2018. Automated generation of constraints from use case specifications to support system testing. In: 11th IEEE International Conference on Software Testing, Verification and Validation. ICST. pp. 23–33.
White, L., Almezen, H., 2000. Generating test cases for gui responsibilities using complete interaction sequences. In: Proceedings of the 11th International Symposium on Software Reliability Engineering. ISSRE '00, IEEE Computer Society, Washington, DC, USA, pp. 110–121, URL http://dl.acm.org/citation.cfm?id=851024.856239.
Winkler, S., Pilgrim, J., 2010. A survey of traceability in requirements engineering and model-driven development. Softw. Syst. Model. 9 (4), 529–565. http://dx.doi.org/10.1007/s10270-009-0145-0.
Xie, T., 2006. Augmenting automatically generated unit-test suites with regression oracle checking. In: Proceedings of the 20th European Conference on Object-Oriented Programming. ECOOP'06, Springer-Verlag, Berlin, Heidelberg, pp. 380–403. http://dx.doi.org/10.1007/11785477_23.
Xun, Y., Cohen, M., AM, M., 2011. Gui interaction testing: Incorporating event context. IEEE Trans. Softw. Eng. 37 (4), 559–574.
Yilmaz, C., 2013. Test case-aware combinatorial interaction testing. IEEE Trans. Softw. Eng. 39 (5), 684–706. http://dx.doi.org/10.1109/TSE.2012.65.
Yilmaz, C., Fouche, S., Cohen, M., Porter, A.A., Demiröz, G., Koç, U., 2014. Moving forward with combinatorial interaction testing. Computer 47 (2), 37–45. http://dx.doi.org/10.1109/MC.2013.408.
Yuan, X., Memon, A.M., 2010. Generating event sequence-based test cases using GUI run-time state feedback. IEEE Trans. Softw. Eng. 36 (1), 81–95.

Matteo Brunetto got a master degree in Computer Science with honours at University of Milano-Bicocca in Italy. He is currently working as a professional software engineer.

Giovanni Denaro is Associate Professor of software engineering at the University of Milano-Bicocca. He received the Ph.D. degree in Computer Science and Engineering from Politecnico di Milano. His research interests include software testing and analysis, formal methods for software verification, distributed and service-oriented systems, and software metrics. He has been investigator in several research and development projects in collaboration with leading European universities and companies.

Leonardo Mariani received the Ph.D. degree in computer science from the University of Milano-Bicocca, in 2005, where he is currently a Full Professor. His research interests include software engineering, in particular software testing, program analysis, automated debugging, and self-healing and self-repairing systems. He has authored more than 100 papers appeared at top software engineering conferences and journals. He has been awarded with the ERC Consolidator Grant, in 2015, and an ERC Proof of Concept Grant, in 2018, and he is currently active in several European and National projects. He is regularly involved in organizing and program committees of major software engineering conferences.

Mauro Pezzé is a professor of software engineering at the University of Milano-Bicocca and at the Universitá della Svizzera Italiana (USI). He is editor in chief of the ACM Transactions on Software Methodologies, and has served as an associate editor of the IEEE Transactions on Software Engineering, as general chair of the ACM International Symposium on Software Testing and Analysis in 2013, program chair of the International Conference on Software Engineering in 2012 and of the ACM International Symposium on Software Testing and Analysis in 2006. He is known for his work on software testing, program analysis, self-healing and self-adaptive software systems. He is a senior member of the IEEE.