Towards Automated A/B Testing

Giordano Tamburrelli and Alessandro Margara

Faculty of Informatics. University of Lugano, Switzerland.
{giordano.tamburrelli | alessandro.margara}@usi.ch

Abstract. User-intensive software, such as Web and mobile applications, heavily depends on the interactions with large and unknown populations of users. Knowing the preferences and behaviors of these populations is crucial for the success of this class of systems. A/B testing is an increasingly popular technique that supports the iterative development of user-intensive software based on controlled experiments performed on live users. However, as currently performed, A/B testing is a time consuming, error prone and costly manual activity. In this paper, we investigate a novel approach to automate A/B testing. More specifically, we rephrase A/B testing as a search-based software engineering problem and we propose an initial approach that supports automated A/B testing through aspect-oriented programming and genetic algorithms.

1 Introduction

Modern software systems increasingly deal with large and evolving populations of users that may issue up to millions of requests per day. These systems are commonly referred to as *user-intensive* software systems (e.g., Web and mobile applications). A key distinguishing feature of these systems is the heavy dependence on the interactions with many users, who approach the applications with different needs, attitudes, navigation profiles, and preferences[1].

Designing applications that meet user preferences is a crucial factor that may directly affect the success of user-intensive systems. Underestimating its importance can lead to substantial economic losses. For example, an inadequate or distorted knowledge of user preferences in a Web application can lead to an unsatisfactory user experience with consequent loss of customers and revenues.

Domain experts typically provide valuable insights concerning user preferences that engineers can exploit to obtain an effective design of user-intensive applications. Unfortunately, this information could be inaccurate, generic, and obsolete. In practice, it is almost impossible to design applications that accurately capture all possible and meaningful user preferences upfront.

As a consequence, engineers typically design a user-intensive application relying on the initial available knowledge while, at run-time, they continuously monitor, refine, and improve the system to meet newly discovered user preferences. In this context, engineers increasingly rely on *A/B testing*[2] [11] to evaluate

[1] We collectively identify these factors under the term *user preferences*.

[2] A/B testing is also known as randomized experiments, split tests, or control/treatment. In this paper we always use the term A/B testing.

and improve their applications. In A/B testing, two distinct variants (i.e., variant A and B) of the same application are compared using *live experiments*. Live users are randomly assigned to one of the two variants and some metrics of interest (e.g., the likelihood for a user to buy in an e-commerce Web application) are collected. The two variants are compared based on these metrics, and the best one is selected, while the other is discarded. The iterative development of variants and their comparative evaluation through live experiments allow designers to gradually evolve their applications maximizing a given metric of interest. For example, an e-commerce application may be refactored adopting variants that maximize sales, while a mobile application may be refactored adopting variants that maximize the advertisements' views.

A/B testing is being increasingly adopted by the industry and proved to be effective [3]. Still, it suffers from several limitations. Indeed, conceiving, running, and summarizing the results of A/B tests is a difficult, tedious, error prone, and costly manual activity [5]. This paper tackles this issue laying the foundations of an automated A/B testing framework in which the generation of application variants, their run-time evaluation, and the continuous evolution of the system are automatically obtained by casting the process of A/B testing to a Search-Based Software Engineering (SBSE) [7] problem. This novel viewpoint on A/B testing brings to the table several research challenges defined and discussed in the paper.

The contribution of this paper is twofold. First, it lays the foundations and explores the potential of automated A/B testing as an optimization problem. Specifically, it proposes an initial approach based on aspect-oriented programming [8] and genetic algorithms [13], which can be considered as a primer to demonstrate the feasibility of the concepts introduced in the paper and a first concrete step towards their practical application. Second, it provides the SBSE community with a novel and crucial domain where its expertise can be applied.

The remainder of the paper is organized as follows. Section 2 provides a more detailed introduction to A/B testing and discusses some open issues. Section 3 rephrases the process of A/B testing as an optimization problem. Next, Section 4 reifies the illustrated concepts in the context of user-intensive Web applications with a solution based on aspect-oriented programming and genetic algorithms. Section 5 presents some preliminary results. Finally, Section 6 surveys related work and Section 7 draws some conclusions and discusses future work.

2 Background and Problem Statement

This section introduces A/B testing, partially recalling the definition reported in [11]. Next, it points out some of the existing limitations of A/B testing and discusses the need for automating it.

The diffusion and standardization of Web technologies and the increasing importance of user-intensive software represent a perfect playground to evaluate competing alternatives, ideas, and innovations by means of controlled experiments, commonly referred to as A/B tests in this context. The overall process of A/B testing is exemplified in Fig. 1. Live users are randomly assigned to one

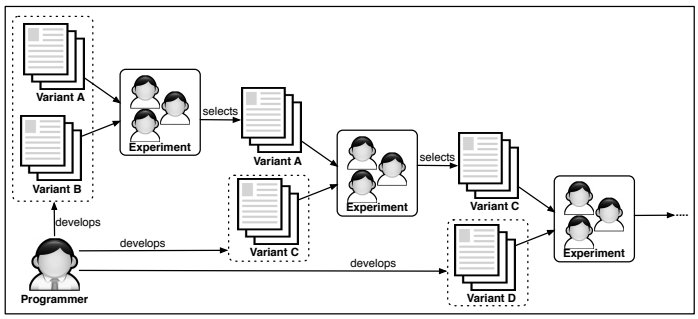


Fig. 1. A/B testing iterative process.

of two variants of the system under analysis: variant A (i.e., the *control* variant), which is commonly the current version, and variant B (i.e., the *treatment* variant), which is usually a newer version of the system being evaluated. The two variants are compared on the basis of some metrics of interest related to the user preferences. The variant that shows a statistically significant improvement is retained, while the other is discarded. As previously mentioned, the iterative development of variants and their comparative evaluation through live controlled experiments allow designers to gradually evolve their applications maximizing the metrics of interest.

Even if widely and successfully adopted in industry [3,10], A/B testing is still considered by the majority of developers as a complex and crafted activity rather than a well-established software engineering practice. Indeed, conceiving, running, and summarizing the results of A/B tests is a difficult, tedious, and costly manual activity. More precisely, an accurate and consistent A/B testing demands for several complex engineering decisions and tasks. The most relevant ones are illustrated hereafter.

1. *Development and deployment of multiple variants.* A/B testing requires a continuous modification and deployment of the application codebase to implement and evaluate variants. These variants are deployed and monitored concurrently serving at the same time a certain percentage of users.

2. *What is a variant.* Programs may be customized along different lines. Because of this, a critical choice for developers is the selection of how many and which aspects of the program to change when generating a new variant.

3. *How many variants.* We defined A/B testing as the process of concurrently deploy and evaluate two variants of the system. In the general case, developers may concurrently deploy more than two variants. However, they do not typically have evidences to select this number effectively and to tune it over time.

4. *How to select variants.* As previously mentioned, A/B testing works iteratively. At the beginning of each iteration, developers have to decide which variants to deploy and test. Prioritizing certain variants is critical for quickly finding

better program configurations. However, selecting the most promising variants is also difficult, especially for large and complex programs.

5. *How to evaluate variants.* A sound and accurate comparison of variants in live experiments with users requires mathematical skills (e.g., statistics) that developers do not necessarily have. For example, sizing the duration of tests and the number of users involved is a crucial factor that affects the quality of results.

6. *When to stop.* Usually, A/B testing enacts a continuous adaptation process that focuses on certain specific aspects of a program. Understanding when a nearly optimal solution has been reached for those aspects is critical to avoid investing time and effort on changes that provide only a minimal impact on the quality of the program.

Not only the factors mentioned above represent concrete obstacles for developers, but they also characterize A/B testing as an error-prone process that may yield to unexpected, counter-intuitive, and unsatisfactory results (see for example [9,12]). To facilitate the adoption of A/B testing and to avoid potential errors, we believe it is a crucial goal of the software engineering research to provide the developers with a set of conceptual foundations and tools aimed at increasing the degree of automation in A/B testing. So far, to the best of our knowledge, this research direction has received little attention.

3 A/B Testing as an Optimization Problem

In this section we take a different and novel perspective on A/B testing, rephrasing it as an optimization problem. The conceptual foundations and the notations introduced hereafter are used in the remainder of this paper.

1. *Features.* From an abstract viewpoint a program p can be viewed as a finite set of *features*: $F_p = \{f_1 \dots f_n\}$. Each feature f_i has an associated domain D_i that specifies which values are valid/allowed for f_i. The concept of feature is very broad and may include entities of different nature and at different level of abstraction. For example, a feature could be a primitive integer value that specifies how many results are displayed per page in an e-commerce Web application. Similarly, a feature could be a string that specifies the text applied to a certain button in a mobile application. However, a feature can also represent more abstract software entities such as a component in charge of sorting some items displayed to the user. The features above are associated to the domains of integers, strings, and sorting algorithms, respectively.

2. *Instantiation.* An instantiation is a function $I_{p,f_i} : f_i \rightarrow D_i$ that associates a feature f_i in F_p with a specific value from its domain D_i. Two key concepts follow: (i) to obtain a concrete implementation for a program p it is necessary to specify the instantiations for all the features in p; (ii) the specification of different instantiations yields to different concrete implementations of the same abstract program p.

3. *Variants.* We call a concrete implementation of a program p a *variant* of p. As a practical example, recalling the features exemplified above, three possible instantiations may assign: (i) 10 to the feature that specifies the number of items displayed per page, (ii) the label *"Buy Now"* to the button, (iii) an algorithm that sorts items by their name to the sorting component. These instantiations define a possible variant of the system.

4. *Constraints.* A constraint is a function $C_{i,j} : D_i \rightarrow \mathcal{P}(D_j)$ that, given a value $d_i \in D_i$ for a feature f_i, returns a subset of values in D_j that are *not* allowed for the feature f_j. Intuitively, constraints can be used to inhibit combinations of features that are not valid in the application domains. For example, consider a Web application including two features: font color and background color. A developer can use constraints to express undesired combination of colors. We say that a variant of a program p *satisfies* a constraint $C_{i,j}$ if $I_{p,f_j} \notin C_{i,j}(I_{p,f_i})$. A variant is *valid* for p if it satisfies all the constraints defined for p.

5. *Assessment Function.* An assessment function is a function defined as $o(v) : V_p \rightarrow \mathbb{R}$, where $V_p = \{v_1, \ldots, v_m\}$ is the set of all possible variants for program p. This function associates to each and every variant of a program a numeric value, which indicates the goodness of the variant with respect to the goal of the program. The assessment function depends on the preferences of users and can only be evaluated by monitoring variants at run-time. As previously mentioned, the likelihood for a user to buy is a valid assessment function for an e-commerce Web application. Indeed, this metric is evaluated at run-time for a specific variant of the application and concretely measures its goodness with respect to the ultimate goal of the program (i.e., selling goods): higher values indicate better variants.

Given these premises, we can rephrase A/B testing as a search problem as follows. Given a program p characterised by a set of features F_p, a set of constraints C_p, and an assessment function $o(v)$, find the variant $\hat{v} \in V_p$ such that $\hat{v}$ is valid and maximizes $o(v)$:

$$\hat{v} = \arg\max_{v} o(v)$$

4 Towards Automated A/B Testing

Section 2 identified some difficulties and open issues in the usage of A/B testing, mainly deriving from a limited degree of automation. We claim that, by formulating A/B testing as an optimization problem as shown in Section 3, we can effectively exploit automated search algorithms to investigate and enable automated A/B testing. This section reifies this idea.

In our vision, automated A/B testing can be achieved by combining together two ingredients: (i) an appropriate design-time declarative facility to specify program features, and (ii) a run-time framework in charge of automatically and iteratively exploring the solution space of possible concrete programs by: generating, executing, and evaluating variants. We captured these ideas in a

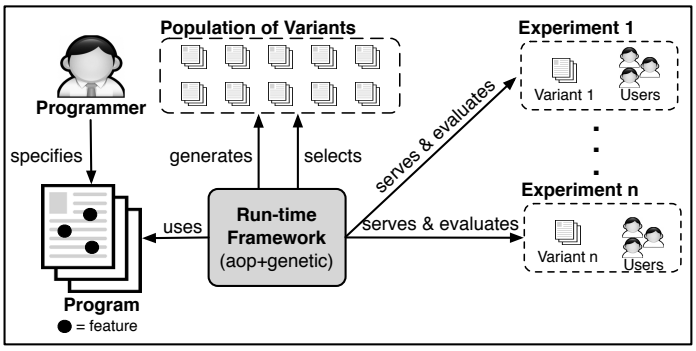


Fig. 2. A reference architecture for automated A/B testing.

reference architecture (see Fig. 2) and we implemented it in a prototype tool. The architecture consists of two main steps, summarized below and detailed in the following paragraphs.

Specifying Features. As stated in Section 2, A/B testing requires a significant effort in developing and deploying the conceived variants. To alleviate the developer from this burden, our architecture provides ad-hoc annotations to specify the set of relevant features for a program and their associated domain. In other words, this allows the developer to write a *parametric* program only once, that represents all possible variants that will be automatically instantiated later on at run-time.

Selecting and Evaluating Variants. As stated in Section 2, developers need to take several critical decisions to guide and control the iterative search for better solutions through A/B testing. To overcome these difficulties, our architecture provides a run-time framework that automates the search process by exploiting genetic algorithms. In particular, it creates and iteratively evolves a population of variants by selecting, mutating, and evaluating its individuals. At execution time, the framework instantiates concrete variants from the parameterized program by means of a *dependency injection* mechanism based on aspect-oriented programming [8]. The run-time framework assesses the goodness of instantiated variants by means of an appropriate application-specific assessment function (see Section 2) that measures the preferences of users. Such assessesment function is adopted as fitness function for the genetic algorithm and drives the generation and evolution of new variants at each iteration.

4.1 Specifying Features

In our approach, developers can specify features by means of annotated variable declarations[3]. Variables declared as features cannot be initialized or modified

[3] Our prototype is designed for the Java programming language. However, the illustrated concepts and techniques apply seamlessly to other languages and technologies.

```
1 @StringFeature(name="checkOutButtonText",
2    values={"CheckOut", "Buy", "Buy Now!"})
3 String buttonText; // Primitive feature specification
4
5 @IntegerFeature(name="fontSize", range="12:1:18")
6 int textSize; // Primitive feature specification
7
8 Button checkOutButton = new Button();
9 checkOutButton.setText(buttonText); // Primitive feature use
10 checkOutButton.setFontSize(textSize); // Primitive feature use
```

Listing 1.1. Primitive Type Feature Example.

since their value is automatically assigned at execution time by the run-time framework. As mentioned in Section 3, the concept of feature is very broad. In particular we distinguish two categories of features: (*i*) *Primitive Type* and (*ii*) *Abstract Data Type* features. The former refers to program features that can be modelled with primitive type (integers, double, boolean, and strings); the latter refers to features implemented through ad-hoc abstract data types. Hereafter we discuss in details both options.

Primitive Type Features. Let us consider the example in Section 3 of a feature of string type that specifies the label of a button in a mobile application. To represent this feature and its domain, developers can declare a string variable annotated with the **@StringFeature** annotation (see Listing 1.1). In this case, the features can assume values in the entire domain of strings. Developers can restrict such domain by specifying a set of valid values in the *values* parameter. Analogous annotations are provided for other primitive types (e.g., **@IntegerFeature**, **@BooleanFeature**, etc.). Differently from string features, the domain of numeric features can be specified as a range of valid values. For example, reasonable values for an integer feature that represents a font size may be in the range (12:18) with a step of 1.

The run-time framework instantiates and serves to users concrete variants of the program that are automatically generated by injecting values to the features declared by the developers. Thus, considering the code in Listing 1.1, a user accessing the system may experience a variant of the program in which the button is labeled with the text *"Buy Now!"* with font size equal to 12, while a different user, at the same time, may experience a different variant of the program with text *"Check-out"* and font size equal to 16.

Generic Data Type Feature. Real-world programs may be characterized by complex features that require ad-hoc abstract data types. We support them by relying on two ingredients: (*i*) an interface that specifies the abstract behavior of the feature and (*ii*) several implementations of the interface that contain the concrete realizations of this feature.

Let us recall the example mentioned in Section 3 of a component in charge of sorting some items displayed to the users. Possible realizations of this feature may sort items by name, price, or rating. Thus, the generic aspect of sorting items can be defined as a feature as shown in Listing 1.2. To do so, developers declare an

```
1 @GenericFeatureInterface(name="sort",
2    values={"com.example.SortByPrice", "com.example.SortByName"})
3 public interface AbstractSortingInterface{
4    public List<Item> sort();
5 }
6
7 public class SortByPrice implements AbstractSortingInterface{
8    public List<Item> sort(){
9       // Sort by price implementation
10   }
11 }
12
13 public class SortByName implements AbstractSortingInterface{
14   public List<Item> sort(){
15      // Sort by name implementation
16   }
17 }
18
19 // ...
20
21 @GenericFeature
22 AbstractSortingInterface sortingFeature; // ADT feature specification
23
24 sortingFeature.sort(..); // ADT feature use
```

Listing 1.2. Generic Data Type Feature Example.

interface that includes all the required methods (i.e., *sort*(...) in this example) and implement as many concrete realizations of this interface as needed. The interface must be annotated with the **@GenericFeatureInterface** annotation including the full class name of all its implementations.

Developers can declare variables of the type specified by the interface and annotated with **@GenericFeature**. Analogously to primitive type features, the run-time framework serves to users concrete variants of the program that are automatically generated by injecting an appropriate reference to one of the interface implementations. For example, the invocation to the sort method in the last row of Listing 1.2 may be dispatched to an instance that implements the *SortByName* algorithm or to an instance that implements the *SortByPrice* algorithm, depending on the type of the object injected at run-time.

4.2 Selecting and Evaluating Variants

So far, we explained how developers can declare features in a program and we delegated to the run-time framework the task of generating, executing, and evaluating variants. Now, we explore how the run-time framework actually implements these aspects.

In our prototype, the run-time framework relies on a genetic algorithm that runs online while the system is operating. A genetic algorithm encodes every possible solution of an optimization problem as a *chromosome*, composed of several *genes*. It selects and iteratively evolves a population of chromosomes by applying three main steps (discussed below): (*i*) selection, (*ii*) crossover, and (*iii*) mutation. Each chromosome is evaluated according to a *fitness function*. The algorithm terminates after a fixed number of iterations or when subsequent

iterations do not generates new chromosome with significantly improved values of fitness. Solving the problem of searching for variants via genetic algorithms requires to specify an encoding and a concrete strategy for each of the three main steps mentioned above. Next, we provide these ingredients for the specific case of A/B testing.

Encoding. Each feature declared by developers directly maps into a gene, while each variant maps into a chromosome. Analogously, the assessment function, which evaluates variants on live users, corresponds directly to the fitness function, which evaluates chromosomes. Two additional aspects are required to fully specify a valid encoding: the number of chromosomes used in each iteration and the termination condition.

Our framework enables the developers to specify application specific fitness functions. In addition, it accepts a preferred population size, but adaptively changes it at run-time based on the measured fitness values. Furthermore, the framework is responsible for terminating the experiment when the newly generated variants do not provide improvements over a certain threshold.

Selection. Selection is the process of identifying, at each iteration, a finite number of chromosomes in the population that survive, i.e., that are considered in the next iteration. Several possible strategies have been discussed in the literature. For example, the tournament strategy divides chromosomes into groups. Only the best one in each group (i.e., the one with the highest fitness value) wins the tournament and survives to the next iteration. Conversely, the threshold selection strategy selects all and only the chromosomes whose fitness value is higher than a given threshold. Traditional A/B testing, as described in Section 2, corresponds to the tournament strategy when the population size is limited to two chromosomes. Our framework supports several different strategies. This enables for more complex decision schemes than in traditional A/B testing (e.g., by comparing several variants concurrently). At the same time, selection strategies relieve the developer from manually selecting variants during A/B testing.

Crossover & Mutation. Crossover and mutation contribute to the generation of new variants. Crossover randomly selects two chromosomes from the population and generates new ones by combining them. Mutation produces instead a new chromosome (i.e., a new variant) starting from an existing one by randomly mutating some of its genes. One of the key roles of mutation is to widen the scope of exploration, thus trying to avoid converging to local minima. In traditional A/B testing, the process of generating new variants of a program is performed manually by the developers. Thanks to the crossover and mutation steps of genetic algorithms, also this critical activity is completely automated.

The architecture described so far represents a first concrete implementation of an automated solution to the A/B testing problem. This contributes to overcome some of the burdens of manual A/B testing discussed in Section 2. However, it is worth mentioning that some issues still remain open as exemplified hereafter. First, the behaviour of the architecture still needs to be configured by speci-

fying several parameters (e.g., the population size, selection strategy, and the termination condition). Section 5 discusses some relevant scenarios that exemplify their role. Experimental campaigns on real users will be required to further study and tune them appropriately. Second, further investigations are required on the mapping between the automated A/B testing problem and our proposed architecture. As an example let us consider mutation. Its capability are highly dependant from the encoding of features. Currently, we support mutations among different values of primitive types and among different implementations of an interface. In the future we envision more complex forms of mutation that tries to automatically modify the source code (e.g., by swapping the order of some statements). Finally, even if we did not discuss the role of constraints (see Section 3) in our architecture, they can be easily modelled and integrated in genetic algorithms as demonstrated in literature (e.g., [6]).

5 Preliminary Validation

In this section, we provide an initial empirical investigation of the feasibility of automated A/B testing. To do so, we generate a sample program, we simulate different user preferences, and we study to which extent an automated optimization algorithm converges towards a "good" solution, i.e., a solution that maximizes the quality of the program, as measured through its assessment function. In our evaluation, we consider different parameters to model the preferences of users, the complexity of the program, and to configure the computational steps performed by the genetic algorithm.

Experiment setup. For our experiments, we adopt the implementation described in Section 4. Our prototype is entirely written in Java and relies on JBoss AOP [2] to detect and instantiate features in programs and on the JGAP library [14] to implement the genetic algorithm that selects, evolves, and validates variants of the program at run-time. In our experiments, we consider a program with n features. We assume that each feature has a finite, countable domain (e.g., integer numbers, concrete implementations of a function). Furthermore, for ease of modeling, we assume that each domain D is a metric space, i.e., that we can compute a distance $d_{1,2}$ for each and every couple of elements $e_1, e_2 \in D$. To simulate the user preferences (and compute the value of the assessment function for a variant of the program), we perform the following steps.

1. We split the users into g groups. Each user u selects a "favourite" value $best_f$ for each feature f in the program; users within the same group share the same favourite values for all the features.

2. We assume that the assessment function of a program ranges from 0 (worst case) to 1000 (best case). When a user u interacts with a variant v of a program, it evaluates v as follows. It provides a score for each feature f in v. The score of f is maximum (1000) if the value of f in v is $best_f$ (the favourite value for u) and decreases linearly with the distance from $best_f$. The value of a variant is the average score of all its features.

Number of features in the program	10
Number of values per feature	100
Number of variants evaluated concurrently	100
Number of user groups	4
Distance threshold	80% of maximum distance
Number of evaluations for each variant	1000
Stopping condition	10 repetitions with improvement $< 0.1\%$
Selection strategy	Natural selection (90%)
Crossover rate	35%
Mutation rate	0.08%

Table 1. Parameters used in the default scenario.

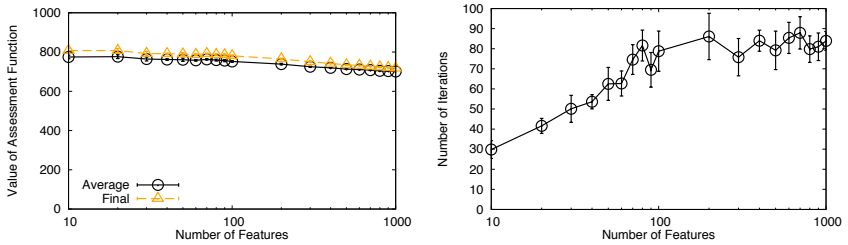
Measure	**Average**	**95% Confidence Interval**
Value of the assessment function (Best)	810.9	11.7
Value of the assessment function (Average)	779.2	10.7
Number of Iterations	34.2	5.9

Table 2. Results for the default scenario.

3. We set a distance threshold t. If the distance between the value of a feature and the user's favourite value is higher than t, then the value of the variant is 0.

Intuitively, the presence of multiple groups models the differences in the users' profile (e.g., differences in age, location, culture, etc.). The more the value of features differ from a user's favourite ones, the worse she will evaluate it. The threshold mimics the user tolerance: after a maximum distance, the program does not have any value for her. While this is clearly a simplified and abstract model, it is suitable to highlight some key aspects that contribute to the complexity of A/B testing: understanding how to select the variants of a program and how to iteratively modify them to satisfy a heterogeneous population of users may be extremely difficult. In addition to the issues listed in Section 2, a manual process is time consuming and risks to fail to converge towards a good solution, or it may require a higher number of iterations.

Default Scenario. To perform the experiments discussed in the remainder of this section, we defined a default scenario with the parameters listed in Table 1. Next, we investigate the impact of each parameter in the measured results. For space reasons, we report here only the most significant findings.
Our default scenario considers a program with 10 different features, each one selecting values from a finite domain of 100 elements. At each iteration, we concurrently evaluate 100 program variants, submitting each variant to 1000 random users. Users are clustered into 4 different groups. They do not tolerate features whose distance from their favourite value is higher than 80% of the maximum distance. At each iteration, the genetic algorithm keeps the number of chromosomes (i.e., program variants) fixed. We adopt a natural selection strategy that selects the best 90% chromosomes to survive for the next generation. Selected chromosomes are combined with a crossover rate of 35% and modified using a



(a) Value of Assessment Function (b) Number of Iterations

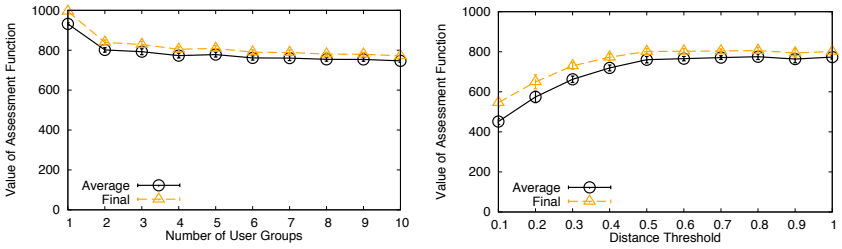
Fig. 3. Impact of the complexity of the program

mutation rate of 0.08%. The process stops when the improvement in fitness value is lower than 0.1% for 10 subsequent iterations.

In each experiment, we measure the number of iterations performed by the genetic algorithm, the value of the assessment function for the selected variant, and the average value of the assessment function for all the variants used in the iterative step. The first value tells us how long the algorithm needs to run before providing a result. The second value represents the quality of the final solution. Finally, the third value represents the quality of the solutions proposed to the users during the iterative process. In A/B testing, this value is extremely important: consider for example an e-commerce Web application, in which the developer wants to maximize the number of purchases performed by users. Not only the final version of the application is important, but also all the intermediate versions generated during the optimization process: indeed, they may be running for long time to collect feedback from the users and may impact on the revenue of the application. We repeated each experiment 10 times, with different random seeds to generate the features of the program, the user preferences, and the selection, crossover, and mutation steps. In the graph below, we show the average value of each measure and the 95% confidence interval.

Table 2 shows the results we measured in our default scenario. Despite the presence of 4 user groups with different requirements, the algorithm converges towards a solution that provides a good average quality. We manually checked the solutions proposed by the algorithm and verified that they always converged to near-optimal values for the given user preferences. The average value of the assessment function during the optimization process is particularly interesting: it is very close to the final solution, meaning that the algorithm converges fast towards good values. Although the average number of iterations is 34.2, the last iterations only provide incremental advantages. As discussed above, this is a key aspect for A/B testing, since developers want to maximize the revenue of an application during the optimization process.

Complexity of the program. In this section, we analyze how the results change with the complexity of the program, i.e., with the number of specified features. Fig. 3 shows the results we obtained. By looking at the value of the assessment function (Fig. 3(a)), we notice that both the final and the average



(a) Value of Assessment Function (b) Number of Iterations

Fig. 4. Impact of the profile of users

quality decrease with the number of features. This is expected, since a higher number of features increases the probability that at least one is outside the maximum distance from the user's preferred value, thus producing a value of 0. Nevertheless, even in a complex program with 1000 features, both values remain above 700. Moreover, the average value remains very close to the final one. Fig. 3(b) shows that the number of iterations required to converge increases with the number of features in the program. Indeed, a higher number of features increases the size of the search space. Nevertheless, an exponential growth in the number of features only produces a sub-linear increase in the number of iterations. Event with 1000 features less than 100 iterations are enough for the genetic algorithm to converge. These preliminary results are encouraging and suggest that automated A/B testing could be adopted with complex program with several hundreds of features.

Profiles of Users. In this section, we analyze how the profile of users impacts the performance of the optimization algorithm. Fig. 4 shows the results we measured by observing two main parameters: the number of user groups and the maximum distance tolerated by users. For space reason, we show only the value of the assessment function: the number of iterations did not change significantly during these experiments. Fig. 4(a) shows that the value of the assessment function decreases with the number of user groups. Indeed, a higher number of groups introduces heterogeneous preferences and constraints. Finding a suitable variant that maximizes the user satisfaction becomes challenging. With one group, the solution proposed by the genetic algorithm is optimal, i.e., it selects all the preferred features of the users in the group. This is not possible in presence of more than one group, due to differences in requirements. Nevertheless, the quality of the solution remains almost stable when considering from 2 to 10 user groups. Finally, also in this case, the average value of the assessment function remains very close to the final one. Fig. 4(b) shows how the selectivity of users influences the results. When reducing the maximum tolerated difference, it becomes more and more difficult to find a solution that satisfies a high number of users. Because of this, when considering a threshold of only 10% of the maximum distance, the final solution can satisfy only a fraction of the users. Thus, the quality of the solution drops below 600.

Discussion. Although based on a synthetic and simple model of the users' preferences, the analysis above highlights some important aspects of A/B testing. First, our experiments confirmed and emphasized some key problems in performing manual A/B testing: in presence of heterogeneous user groups, with different preferences and constraints, devising a good strategy for evolving and improving the program is extremely challenging. Most importantly for the goal of this paper, our analysis suggests that an automated solution is indeed possible and worth investigating. Indeed, in all the experiments we performed, the genetic algorithm was capable to converge towards a good solution. Moreover, it always converged within a small number of steps (less than 100, even in the most challenging scenarios we tested). Furthermore, intermediate variants of the programs adopted during the optimization process were capable of providing good values for the assessment function. This is relevant when considering live experiments, in which intermediate programs are shown to the users: providing good satisfaction of users even in this intermediate phase may be crucial to avoid loss of customs and revenues.

6 Related Work

The SBSE [7] community focused its research efforts on several relevant software engineering challenges covering all the various steps in the software lifecycle ranging from requirements engineering [17], design [16], testing [4], and even maintenance [15]. However, despite all valuable these research efforts, the problem of evolving and refining systems after their deployment – in particular in the domain of user-intensive systems – received very little attention so far and, at the best of our knowledge, this is the very first attempt to introduce this problem in the SBSE community.

Concerning instead the research on A/B testing we can mention many interesting related efforts. For example, Crook et al. [5] discuss seven pitfalls to avoid in A/B testing on the Web. Analogously, Kohavi et al. [9,12] discuss the complexity of conducting sound and effective A/B testing campaigns. To support developers in this complex activity, Kohavi et al. [11] also provided a practical tutorial. Worth mentioning is also [10], which discusses online experiments on large scale scenarios. Finally, worth mentioning is the Javascript project Genetify [1]. The project represents a preliminary effort to introduce genetic algorithms in A/B testing and demonstrates how practitioners actually demand for methods and tools for automated A/B testing as claimed in the motivation of this paper. However, this project is quite immature and does not exploit all the potentials of genetic algorithms as we propose in this paper: it only supports the evolution of HTML pages in Web applications.

7 Conclusions

In this paper we tackled the problem of automating A/B testing. We formalized A/B testing as a SBSE problem and we proposed an initial prototype that relies

on aspect-oriented programming and genetic algorithms. We provided two important contributions. On the one hand, we used our prototype to demonstrate the practical feasibility of automated A/B testing through a set of synthetic experiments. On the other hand, we provided the SBSE community with a novel domain where its expertise can be applied. As future work, we plan to test our approach on real users and to refine the proposed approach with customized mutation operators (e.g., changes to the source code) and full support for constraints.

References

1. Genetify. `https://github.com/gregdingle/genetify/wiki`. [Accessed 25-02-2014].
2. JBoss AOP. `http://www.jboss.org/jbossaop`. [Accessed 25-02-2014].
3. The A/B Test: Inside the Technology Thats Changing the Rules of Business. `http://www.wired.com/business/2012/04/ff_abtesting`. [Accessed 25-02-2014].
4. Wasif Afzal, Richard Torkar, and Robert Feldt. A systematic review of search-based testing for non-functional system properties. *Inf. Softw. Technol.*, 2009.
5. Thomas Crook, Brian Frasca, Ron Kohavi, and Roger Longbotham. Seven pitfalls to avoid when running controlled experiments on the web. In *ACM SIGKDD*, KDD '09, pages 1105–1114, New York, NY, USA, 2009. ACM.
6. Kalyanmoy Deb. An efficient constraint handling method for genetic algorithms. *Computer Methods in Applied Mechanics and Engineering*, 186(24):311 – 338, 2000.
7. Mark Harman and Bryan F Jones. Search-based software engineering. *Information and Software Technology*, 43(14):833–839, 2001.
8. Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. *Aspect-oriented programming*. Springer, 1997.
9. Ron Kohavi, Alex Deng, Brian Frasca, Roger Longbotham, Toby Walker, and Ya Xu. Trustworthy online controlled experiments: Five puzzling outcomes explained. In *ACM SIGKDD*, pages 786–794. ACM, 2012.
10. Ron Kohavi, Alex Deng, Brian Frasca, Toby Walker, Ya Xu, and Nils Pohlmann. Online controlled experiments at large scale. In *ACM SIGKDD*, KDD '13, pages 1168–1176, New York, NY, USA, 2013. ACM.
11. Ron Kohavi, Randal M Henne, and Dan Sommerfield. Practical guide to controlled experiments on the web: listen to your customers not to the hippo. In *ACM SIGKDD*, pages 959–967. ACM, 2007.
12. Ron Kohavi and Roger Longbotham. Unexpected results in online controlled experiments. *ACM SIGKDD*, 12(2):31–35, 2011.
13. John R Koza. *Genetic programming: on the programming of computers by means of natural selection*, volume 1. MIT press, 1992.
14. K Meffert, N Rotstan, C Knowles, and U Sangiorgi. JGAP - Java Genetic Algorithms and Genetic Programming Package, 2014.
15. M. O'Keeffe and M.O. Cinneide. Search-based software maintenance. In *Software Maintenance and Reengineering, 2006. CSMR 2006. Proceedings of the 10th European Conference on*, pages 10 pp.–260, March 2006.
16. Outi Räihä. A survey on search-based software design. *Computer Science Review*, 4(4):203–249, 2010.
17. Yuanyuan Zhang, Anthony Finkelstein, and Mark Harman. Search based requirements optimisation: Existing work and challenges. In *REFSQ*. Springer, 2008.