User Interface Optimization using Genetic Programming with an Application to Landing Pages

PAULO SALEM, Salem Sistemas and Dell EMC

The design of user interfaces (UIs), such as World Wide Web pages, usually consists in a human designer mapping one particular problem (e.g., the demands of a customer) to one particular solution (i.e., the UI). In this article, a technology based on Genetic Programming is proposed to automate critical parts of the design process. In this approach, designers are supposed to define basic content elements and ways to combine them, which are then automatically composed and tested with real users by a genetic algorithm in order to find optimized compositions. Such a strategy enables the exploration of large design state-spaces in a systematic manner, hence going beyond traditional A/B testing approaches. In relation to similar techniques also based on genetic algorithms, this system has the advantage of being more general, providing the basis of an overall programmatic UI design workflow, and of calculating the fitness of solutions incrementally. To illustrate and evaluate the approach, an experiment based on the optimization of landing pages is provided. The empirical result obtained, though preliminary, is statistically significant and corroborates the hypothesis that the technique works.

CCS Concepts: • **Human-centered computing** → **User interface design**; *Empirical studies in interaction design*; • **Computing methodologies** → **Genetic programming**; • **Software and its engineering** → **Graphical user interface languages**; *Search-based software engineering*; • **Information systems** → **Crowdsourcing**;

Additional Key Words and Phrases: User Interface Generation, User Interface Optimization, Interaction Design, Fatigue Problem, Landing Pages

ACM Reference format:
Paulo Salem. 2017. User Interface Optimization using Genetic Programming with an Application to Landing Pages. *Proc. ACM Hum.-Comput. Interact.* 1, 1, Article 12 (June 2017), 17 pages.
https://doi.org/10.1145/3099583

1 INTRODUCTION

The design of user interfaces (UIs), such as World Wide Web[1] pages, usually consists in mapping one particular problem (e.g., the demands of a customer) to one particular solution (i.e., the UI). This is done by a human designer, who considers the various design alternatives, by combining different titles, images and other elements, to eventually select the one to be used. Such a procedure, however, is intrinsically constrained by the exponential complexity of combining basic elements and the necessity of evaluating the results empirically with users. Combinatorial complexity suggests the possibility of a systematic search through the space of possible designs and the experimental evaluation of candidate solutions can clearly be automated. Therefore, parts of the design process are susceptible to algorithmic optimization.

[1] Henceforth referred to merely as the *Web*.

2573-0142/2017/6-ART12 $15.00
https://doi.org/10.1145/3099583

In industrial practice, the usual approach to perform this kind of optimization is through A/B testing [9], in which two or more versions of a Web page are explicitly defined by a human designer and empirically tested with visitors in order to find the best one. In the research literature, a number of more sophisticated approaches have been proposed to automatically generate and optimize UIs. In particular, Evolutionary Computation [2] (e.g., Genetic Algorithms [11]) has been used [12, 16, 20]. These evolutionary methods rely on the initial definition of primitive design elements, which are then algorithmically combined and tested with actual users in order to determine the best combinations in relation to some performance metric. If users respond positively to the design, its relevance increases; otherwise, it decreases. Although such approaches cannot guarantee that the globally optimum design (i.e., the design that maximizes the desired user behavior as measured by the chosen metric) is found, they do allow for an effective search through the large state-spaces of possible designs that arise from the combinatorial explosion of primitive elements.

In this article a technology along similar lines, designed to optimize UIs (particularly Web pages), is proposed. The search is carried out using Genetic Programming (GP) [19], in which each individual is a possible design and its genome is a program that generates the design.[2] A more detailed comparison with existing UI optimization methods is left for the Section 3. For the moment, it suffices to note the following features, which together distinguish the approach. First, owing to GP, a tree-like programmatic genome representation is used instead of a simple sequence of characteristics. It is argued that such a tree structure is better suited to the task. As we shall see, this follows not only from the fact that programs are computationally more general than linear representations of features, but also from the observation that UIs are naturally represented as hierarchies of compositions (e.g., a button within a panel within a window) and, therefore, a GP approach provides a more appropriate tool to model them. Second, the fitness of solutions is calculated through the collaborative effort of many users (each user has to interact with a page only once), instead of depending on the repetitive interaction of a single user (which leads to the so-called *fatigue problem* [25]).

To validate the approach, the concrete application of crafting *landing pages* is used as an experiment. For commercial Web companies, a key problem concerns what is known as *conversion rates*: the probability that a user, once reaching the website, will perform some desired action (such as creating an account or purchasing a product). The higher such rates are, the lower the costs of customer acquisition is. Landing pages are special pages in which conversion rates are optimized for a particular audience and to which promising users can be directed (e.g., after they click on an advertisement). Since the purpose of these pages is solely to optimize some desirable outcome, it is a very natural place to apply UI optimization techniques. The experimental hypothesis to be tested, then, becomes this: can the technique generate optimized landing pages with higher conversion rates in relation to some (fixed) control page? As it will be seen, the answer is affirmative. The particular experiment presented here was performed with a real online furniture store[3] who desired to collect email addresses for its newsletter through such a page. The various elements used to perform this experiment will be introduced throughout the article in order both to make the experiment clear and to illustrate the general technique itself.

The text is organized in the following manner. First, an overview of the fundamental Evolutionary Computation concepts is provided, mainly with respect to Genetic Algorithms and Genetic Programming, since the reader might not be familiar with them (Section 2). Next, relevant related work is explored to position the contribution of the present article (Section 3). The main technical contribution is then presented and the landing page experiment is also gradually introduced (Section 4). Care is taken to

[2]The technique is implemented in the Scala programming language [15], but the principles described are general and can be carried out independently.

[3]http://www.agnolias.com.br

explain not only the algorithmic aspects of the approach, but also its methodological characteristics. The actual experimental results are given afterwards (Section 5). The main such result is a successful experiment which validates the approach and uses the exact algorithm presented. A previous failed experiment, based on an earlier version of the algorithm, is also provided in order to establish a contrast with the final successful version. This experimental baseline highlights some critical features of the approach that had to be found empirically. Finally, the article concludes (Section 6).

2 EVOLUTIONARY COMPUTATION OVERVIEW

Evolutionary Computation (EC) groups all techniques inspired by the process of biological evolution to solve computational problems [2]. Among these techniques, this article is concerned specifically with Genetic Programming (GP) [19], which can be understood as a particular kind of Genetic Algorithm (GA) [11]. Below a brief overview of GAs is provided and then it is explained in what respects GP is different.

GAs are search algorithms in which, at any given time, there are a number of candidate solutions defined. Each such solution is called an *individual* and collectively they form a *population*. Each individual has a *genome* (often referred to as the individual's *chromosomes* as well) which encodes its unique characteristics as *genes*. As in Biology, *genes* must be *expressed* somehow in order to generate the individual's observable characteristics, that is to say, its *phenotype*. Once available, a phenotype can be evaluated by a *fitness function*, which defines how good or bad a solution that particular individual is.[4]

A GA begins with an initial and possibly random population. It then repeats the following cycle. First, individuals are evaluated and then some of the best are selected for *mating*. Mating consists in producing a new individual from existing ones (typically from two parents). To do so, the parents' genomes are used in two operations that yield the child's genome: a *crossover*, which mixes the parental genomes; and a *mutation*, which changes randomly the resulting mixed genome. Once children are produced, the whole population is evaluated again and the worst individuals (as measured by their fitness and perhaps other auxiliary considerations) are excluded so that the size of the population remains as it was before reproduction. In this manner, the GA is able to explore the relevant solutions and create new ones in a directed manner, which gradually moves the population towards better solutions. In principle, the GA does not need to terminate, the cycle can be repeated forever. In practice, however, stopping criteria are usually defined, such as convergence criteria or simply a maximum number of cycles.

Traditionally, GAs' chromosomes are sequences of features. The most basic such sequence contains binary entries, but other kinds of elements are possible. For instance, in a numeric optimization problem, the chromosome can be a sequence with the objective function's real-valued inputs. Another example, closer to the application described in this article, is a sequence of UI features, such as the colors of various UI elements.

GP, in turn, is a kind of GA in which chromosomes are not just sequences of attributes, but whole structured programs. Since programs can be represented by syntax trees, in GP the genome is no longer a sequence, but a tree. Moreover, its phenotype is defined by the execution of this program. Such a tree-like structure brings some advantages for UI representation, which are examined in Section 4.2 later in the article.

When the relevant fitness function depends not only on some predefined computation but also on an external input (e.g., from a human user), the technique is called *Interactive Evolutionary Computation*

[4]This terminology is standard in the area, but it may become confusing in the context of Human-Computer Interaction. Because the word "individual", when not used in the technical sense described here, may refer to people, in the present text the following convention is adopted. Throughout the article, a "user" always denotes a human who interacts with the system. An "individual", on the other hand, always denotes a candidate solution of the genetic algorithm.

(IEC) [25]. Specific variations exist depending on the algorithm used, such as *Interactive Genetic Algorithm (IGA)* and *Interactive Genetic Programming (IGP)*. IEC has been extensively used to optimize systems employing user interaction as part of the fitness function, including systems in computer graphics [23], image processing [18], music [1] and also UI generation, which is detailed below.

3 RELATED WORK

Automated UI generation, optimization and evaluation has been explored in the broad area of Human-Computer Interaction through the use of various techniques. Rule-based and domain-specific template technology have been proposed by Nichols *et al.* [13, 14], and automatic UI tailoring for multiple devices has been explored by Raneburger *et al.* [21]. Although the work reported in the present article is quite different, it does employ the idea of reusable interface components and composition rules for UI generation. Toomim *et al.* [27], in turn, have proposed the crowdsourcing of user preference estimation, which is applicable in many contexts, including the present approach. Gajos *et al.* [6] developed SUPPLE, a sophisticated way to optimize UI design *a priori*, by casting the problem as a mathematical optimization problem. Given declarative definitions of widgets and other UI elements, as well as suitable user and device cost functions based on known design principles, SUPPLE is capable of creating UIs which are optimal with respect to the constraints provided without any *a posteriori* experimentation to guide the process. Tran *et al.* [28] have proposed a method to generate abstract UIs from task models, also without incorporating user behavior directly. Other similar examples exist [10, 17, 29]. Indeed, none of these techniques measure actual user behavior to guide the generation and optimization, but instead rely on design rules and heuristics. This contrasts with IEC approaches in general and with the present work in particular, in which constant user feedback is essential during the whole process. This allows optimizations that depend not only on the layout of the UI, but also on its actual contents (e.g., images, texts, etc.), which might affect user behavior in ways that cannot be known beforehand (e.g., that a user is motivated to act by certain subjects only).

With respect to IEC literature, the present work can be classified as an IGP applied to user interfaces (or partly, as Takagi [25] puts it, *editorial design*) and some similar approaches can be found. Monmarche *et al.* [12] present an IGA for the generation of style sheets of HTML[5] pages. Similarly, Oliver *et al.* [16] show an IGA capable of optimizing page style and also some rudimentary properties of layout. Quiroz *et al.* [20], in turn, provide a similar method, but instead of using HTML pages, they optimize user interface specifications written in the XUL language. In all cases, the same user is required to evaluate different versions of the design, which leads to the so-called *fatigue problem*. The present approach avoids this issue simply by diluting the required user feedback among a large set of users, each providing only one evaluation (in the form of measurable actions). Gatarski [7] reports a successful case study of advertisement banner optimization in which a similar way of evaluating individuals is taken.

Tamburrelli and Margara [26] have recently cast the notion of A/B testing in terms of Search-Based Software Engineering [8] concepts and used GAs to address it. Their work is different with respect to the IEC approaches seen above mainly in that they adopt a program instrumentation technology. Nevertheless, it seems that the underlying chromosomes are not themselves programs, but sequences of features. Genetify [3] is a somewhat similar approach, but instead of Java, it targets HTML page optimization via JavaScript.

All of these IEC methods employ sequences of characteristics as genomes, whereas the proposed approach uses structured programs. This has two main advantages: the structure of the genomes matches that of the UIs themselves, which allows more general specification abstractions; and programs can be

[5]Hyper Text Markup Language.

extended with more statements to allow the composition of different kinds of UIs, whereas sequences of characteristics are just parameters for an existing structure and therefore cannot be readily extended to allow greater expressive power. These features are further explored in Section 4.2 below.

4 USER INTERFACE OPTIMIZATION

There are five main aspects to the proposed technique, namely: design elements, which provide the fundamental material with which to build the UIs; genomes, which are programs that combine such elements in various ways; the evolutionary algorithm itself; the methodological implications of the technology; and the application domains to which it is suitable. Let us now examine each of these aspects.

4.1 Design Elements

The optimization of UI composition depends, first of all, on the availability of primitive UI building blocks. In practice, these are simple creative elements provided by a designer. For example, a designer may define several possibilities of page title, introductory text and background image. Algorithmically, these elements provide a source of variation required for the GP search.

In the experiment, the primitive design elements used are of two kinds: pieces of HTML that provide a template to specific parts of the page; and content variations (e.g., titles, taglines, features, headings, etc.). For example, here are some variations for the title and tagline of the page[6]:

- *Title variations:* "Learn how to furnish your home with style and sustainability"; "Lovely and recycled furniture"; "Good taste, recycled"; "Sustainability and decor in one place".
- *Tagline variations:* "Charming for your home, excellent for the Planet."; "Leave your email and get eligible for a special discount."; "We build lovely things out of recycled wood. Let's talk."

In practice, these and other content elements are provided as inputs to the genetic program, which uses them to initialize variables.

4.2 Genome Structure

Genetic Programming is characterized by the use of program trees instead of linear sequence of data to represent chromosomes. This practice is followed here, but a second genetic component beyond the program tree is also added, namely, a set of variables. Hence, the genome structure has two components:

- *Program tree.* A program that, when executed, generates a UI.
- *Variables.* A mapping from names to values, which can be accessed by programs.

A tree-like genome matches more closely the structure of UIs themselves, which have sections, subsections, components, subcomponents and other nested entities. This brings a number of advantages in relation to linear genomes which are leveraged in the present work:

- When manipulating genetic representations, it is convenient to be able to move or mutate meaningful units of the UI (e.g., a subsection). This is simple in a tree, in which it suffices to know the root node of the unit concerned, whereas in a linear genome this notion is not straightforward and would have to be encoded somehow.
- The specification of the *possible* UIs can be given as a structured program, so that sub-structures in the program correspond to sub-structures in the generated UIs. This allows the intelligible specification of complex design possibilities. See Figure 1 and the program snippet in Section 4.2.1 below.

[6]The experiment was performed in the Brazilian market and all content is originally written in Portuguese. Here that content is provided translated to English. Moreover, only some of the variations are shown, for the purpose of illustration. The full experiment specification is much longer.

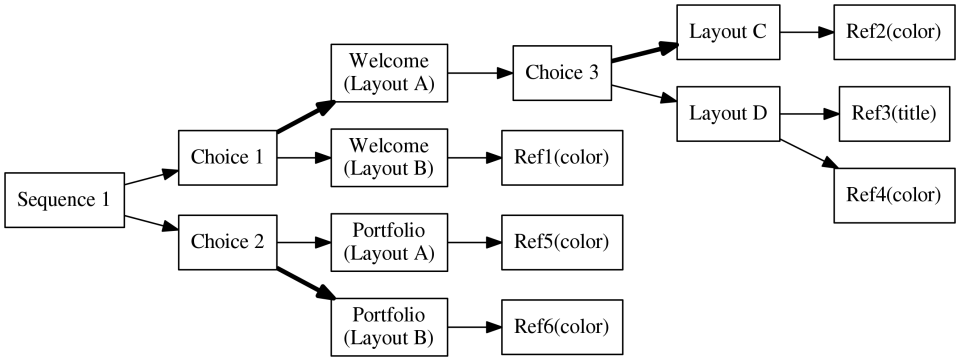


Fig. 1. A simplified genome encoded as a tree. It represents a UI composed of a sequence of two sections, namely, the Welcome and the Portfolio sections. There are some choices of layouts for these sections. Moreover, variables (title and color) are referenced at the leaves (the mapping of names to values is not shown but assumed to exist). The thicker arrows denote one possible set of fixed choices that would result in a concrete solution.

- Non-deterministic choices can be easily modeled as different branches, which allows for the specification of various design possibilities. When the genome is first created, these choices are said to be *free* and represent a non-deterministic computation. However, once the genome is expressed, a concrete, deterministic, path is taken and the choice becomes *fixed*. To do this, the structure of the program tree is not changed, only a marker is added to signal which choice was made. The reason to preserve the complete structure is both to permit the selection to be changed later (via mutation) and to allow a simpler crossover operation among different genomes (more details in Section 4.3). Note the thicker arrows in Figure 1, which denote one possible set of fixed choices that would result in a concrete solution.
- Since nodes in the tree are program statements of some kind, it follows that new statements can be added to the language to increase its power of expression. For example, a new kind of widget can be made available to designers by simply adding a new program statement to the underlying language.

Indeed, because genomes are programs, to specify the possible UIs one needs only to write a program, as described in the next section.

4.2.1 Program Statements. The main component of genomes is the program tree, which is itself built of simpler program statements[7]. Let us explore some of the most important such statements by means of an example, a simplified snippet from the experiment:

```
Include(LeadGenBlock, variation = 1,
  Sequence(
    Choice(
      Define("opacity", "0.2"),
      Define("opacity", "0.4")
    ),
    ParameterValue("title" -> Reference("title")),
    ParameterValue("tagline" -> Reference("tagline")),
```

[7]It would be more accurate to refer to these as program *expressions*, since they all reduce to values. However, because the word "expression" is also used with another sense in the Evolutionary Computing literature (i.e., the execution of a genome), the term "statement" is used instead.

```
    // (...) more parameters after this.
  )
)
```

An `Include` statement defines that the variation 1 of the template block `LeadGenBlock` must be included.[8] The parameters for this template piece are calculated by the substatements of a `Sequence` statement. The first substatement is a non-deterministic `Choice` which defines a styling property, namely, the opacity of the background. It is either 0.2 or 0.4. Then, the `title` and `tagline` parameters are set through `ParameterValue` statements. The `Reference` statements, in turn, fetch the value from a variable of the same name that has been defined by the system based on the content variations provided. This use of variables allows the decoupling of the genetic program and the particular contents used (e.g., the same program could be used for a different business, provided that the content elements were properly changed). While there are many variations of title and tagline, each set of variations is assigned a name. In the program, only the name of the set is required (e.g., "title"), the system picks a random value among all the possible ones. Alternatively, one could have explicitly defined this `Choice` in the program, like it was done for the opacity property. Though not shown, the whole `Include` statement is followed by other `Include`s, which define other parts of the page (e.g., a portfolio section, which shows some products of the store). Note the compositional nature of this program, particularly its hierarchical structures, which provide a clear way to organize the design choices.

4.2.2 Genome Expression. A genome is expressed by executing the program that constitutes it. The main effect of such an execution is the rendering of a UI (i.e., HTML), which is the final artifact that is displayed to users and with which they interact. This proceeds as a regular program execution, except that when non-deterministic choices are found (e.g., the `Choice` statement in the program snippet seen above, which selects a value for the *opacity* variable), they must be fixed and then the selected branch is executed in turn.

After the UI is rendered, the user's action are monitored and recorded. For instance, if the user submits an email through a form found in the UI, this action is recorded. The algorithm uses these action data to update the fitness of the GP individual that generated the corresponding UI, a process that is explained in the next section.

4.3 Algorithm

The fitness of any particular individual depends on how likely it is to elicit a desirable action on users. But this information can only be estimated after many trials of the individual with different users. Thus, it is not possible to devise a fitness function to assign, at once, a value to a genome. Rather, it is necessary to define an *incremental* fitness function, which is capable of gradually calculating the score of each genome.

This is achieved by presenting the renderings induced by each individual to users. Once a user observes one such rendering, he or she may take various actions (e.g., submit a form, click a link, select an option). These actions are recorded and used to update the fitness of the related individual. Therefore, the algorithm is divided in two major steps, presented below: (i) recruiting and processing users; and (ii) evolving the GP population. Some considerations concerning optimization are also provided afterwards.

4.3.1 Recruiting Users. Let us assume that a steady supply of new users is available. To each new user, a GP individual from the population is presented and the user's actions are recorded, in order to be employed according to Algorithm 1 to drive one iteration of the optimization.

[8]This piece provides a form for the user to submit his or her email. In Marketing, that is called a *lead generation* form.

Algorithm 1: Recruit(*user*, *population*, *timeout*)

Input: A user *user*, a set of GP individuals *population*, its base *size* and a duration *timeout* .
Output: An updated GP individual.
let *individual* be a random individual from *population*;
Render *individual* to *user*;
Wait *timeout*;
actions := actions of *user* w.r.t. the rendered UI;
`ActionScore(`*individual*`)` := `ActionScore(`*individual*`)` + `ActionsValue(`*actions*`)`;
`ExpressionCount(`*individual*`)` := `ExpressionCount(`*individual*`)` + 1;
`Fitness(`*individual*`)` := `ActionScore(`*individual*`)` / `ExpressionCount(`*individual*`)`;
if `ShouldEvolve?(`*size*`)` **then**
  `Evolve(`*population*`)`
return *individual*

The procedure is simple but requires a few comments:

- The *timeout* value is the time that the user needs to decide which actions (if any) he or she will perform on the rendered UI. For instance, one may assume this duration to be 30 minutes for a simple landing page optimization.
- The `ActionsValue` function computes the integer value of a set of actions. The exact values of each possible action are defined in an application-specific manner.
- `ActionScore(`*individual*`)` and `ExpressionCount(`*individual*`)` denote the total score of an individual and how many times an individual has been expressed (i.e., shown to a user), respectively. The fitness of the individual, `Fitness(`*individual*`)`, is simply the former divided by the latter, *which corresponds to the conversion rate of the individual*. Such a fitness definition allows for its gradual computation over many presentations of the individual. When created, individuals start with an expression count of 1 and an action score of 0.001 (for mathematical convenience, fitness is always greater than zero).
- The `ShouldEvolve?(`*size*`)` auxiliary procedure determines whether this user should trigger a GP evolution. This must not always be the case, because otherwise GP individuals may get little feedback before they are eliminated from the population. In the experiments performed, evolution takes place at every n^{th} recruited user, where $n = 3 \times size$ and the base *size* is how many individuals the population contains initially (15 in the experiment). In other words, on average each GP individual is expressed three times and therefore receives feedback from three different users. This particular choice of n is empirical and more details about it are given in the experimental results of Section 5.

The actual source of users for the experiment was an advertising campaign[9] designed to reach the store's target audience. The advertisement invites users to subscribe to the newsletter, but does not reveal that an experiment is taking place.

4.3.2 Evolution. The evolution algorithm itself follows a rather standard overall structure (see Algorithm 2). Some of its particularities are commented below.

The `Best` procedure employs a roulette wheel method to select n pairs of parents (where $n = 12$ in the experiment). Through such a selection process, each existing individual has a probability proportional to its current fitness of being chosen.

[9]Using Facebook Ads (https://www.facebook.com/business/products/ads)

Algorithm 2: Evolve(*population*, *size*)

Input: A set of GP individuals *population* and its base *size*.
Output: A new set of (evolved) individuals.
Eliminate the worst individuals in *population* such that only *size* individuals remain;
foreach $\{parent_1, parent_2\} \in$ `Best(`*population*`)` **do**
    child := `Crossover(`$parent_1$, $parent_2$`)`;
    child := `Mutate(`*child*`)`;
    population := *population* $\cup$ {*child*};
Record average fitness of individuals in *population*;
population := *population* $\cup$ `RandomIndividuals(`$\alpha \cdot$ *size*`)`;
return *population*

During some initial tests, it was found that the lack of diversity of individuals after a few generations was detrimental to the algorithm's performance. In order to address this issue in a simple manner, a last step was included, through which random individuals (a proportion α of the population's size, which was set to 20% in the experiment) are added to the population.

As explained previously, an individual's genome has two components: a program tree and a variables mapping. Crossover and mutation operators must, therefore, account for both.

Crossover. For simplicity, the `Crossover` operation assumes that all program trees have the same structure. The difference among genomes lies in whether non-deterministic choices are fixed or free, a property assigned to nodes that represent such choices. Under this assumption, the operation randomly exchanges sub-trees that have equivalent root nodes (i.e., are in the same structural position in each tree), starting at the root of both parent trees and proceeding recursively downwards in each. Such a procedure produces a new program tree that has the same structure as the parents but which may differ on which choices are actually selected.

With respect to the variables mapping, the crossover merely creates a child mapping such that: (i) it contains all names present in both parents; (ii) if a name has different values in each parent, the child is assigned one of them randomly (up to a limit of assignments, after which the values found on the first parent are picked). This simple strategy ensures that any variable needed by the new child program is available.

Mutation. Mutation affects program trees only with respect to which non-deterministic choices are fixed. So if a choice is fixed on alternative 1, the mutation operator will either leave it as it is or, with a certain probability, change it to alternative 2. Variables mappings, in turn, are not affected by mutation.

Mutation probability is dynamic. When the fitness of individuals over generations is improving quickly, mutation probability is reduced. On the other hand, when growth becomes slow, mutation probability is increased. Speed is measured using the slope of the linear regression of the last n (where $n = 5$ in the experiment) recorded average individual fitness values.

Mutation is a source of fresh variability, since it inserts choices that might not be present in the original population. Nevertheless, in its current form it was found to be insufficient. Hence, the last step in Algorithm 2 was introduced. It is possible, however, that an improved mutation operator might fully solve the problem of variability. For instance, mutation could not only change fixed choices, but also create new, structurally different, random sub-trees. In this case, the crossover operation would also have to be more sophisticated in order to account for program trees with different structures.

4.3.3 Optimization Aspects. If few UI compositions are possible, there is no need to use an EC approach such as the one proposed here, because it is possible to actually extensively experiment with each composition and obtain statistically strong guarantees of performance. This is the case, for instance, when a limited number of Web pages are manually crafted by designers and fully experimented with (i.e., A/B testing). However, when UIs are generated from the combination of basic elements, the problem quickly becomes intractable given its exponential nature. In practice, this affects even relatively simple UIs: 10 content blocks each having 6 alternatives (the same order of magnitude used in the experiment reported in this article) yields 6^{10} potential compositions, which would require millions of users to exhaustively test and obtain a strong statistical guarantee of global optimality. This is the matter addressed by the present approach and which justifies the use of EC.

Like any other approach based on EC, the proposed one cannot guarantee that a globally optimal solution will be found. Rather, the aim is to find increasingly better compositions, which typically means that local optima will be found during the algorithm's execution. This is achieved in part by employing randomization processes to guide the search (e.g., crossover and mutation), hence making the present approach an instance of Stochastic Optimization [24] and allowing the analysis of large state-spaces.

4.4 Methodological Considerations

The methodological core of the present approach is this: instead of conceiving one particular solution to a design problem, designers should conceive a *family* of solutions. In practice, this works as follows. The designer must consider the general structures that he or she thinks might be part of a good design, such as the possible layouts of a page or the presence or absence of menus and toolbars. These possible structures are then specified as programs. The possible contents of primitive elements, such as text values or images, are then specified as possible values that the program variables may take. Finally, the program and these basic content elements are subject to the optimization algorithm. The designer may choose to run the algorithm for some time and then pick the best composition at that point or simply allow the algorithm to run indefinitely, hence constantly adapting the UI to new user behavior.

Note that although automated optimizations can test many combinations, the fundamental combinatorial problem remains intractable, it is not possible to explore *all* options. Therefore, one cannot simply, say, request that all colors should be tested; rather, it is necessary to select a limited set of basic colors – and this selection emanates from the style of the designer making it. In this manner, the technique proposed here can be understood as a way to make designers more productive, not as a way to replace them; the ultimate objective is to *augment human intellect* (cf. Engelbart [5]).

The designer should have a few general themes in mind and select composition elements in accordance with those themes. Instead of trying to imagine every combination of the elements provided (exponential complexity), he or she is only expected to to relate each content element with a small set of themes (quadratic complexity). Combination of elements can then be fully delegated to the algorithm for three reasons:

- Nonsensical or otherwise bad combinations will not perform well and, therefore, will be automatically eliminated;
- Good combinations are likely to exist (and prosper) because of the base themes chosen;
- Surprisingly effective combinations may emerge, some which designers would not have thought about. This is particularly interesting because it suggests a kind of artificial creativity to supplement the designer's own creative process.

The present article considers only the fundamental characteristics of the proposed approach, not the final tools that can be built upon this base. However, because the genetic programs described here are

structurally close to what is actually seen by the user, it is clear that tools can be built to ease the designer's task of writing such programs. These would be similar to the existing graphical tools that allow non-programmers to create HTML documents. The difference would be the inclusion of proper provisions to account for the specificities of the present approach, such as how to graphically specify multiple variations of an element. In this manner, the approach could scale better to designers who prefer to use graphical tools.

4.5 Application Domains

Applications of the approach are not limited to Web pages seeking to improve conversion rates. Any user interface design that can be specified as a program and whose success is tied to a measurable metric can, in principle, be subject to IEC in general and to the proposed method in particular. Web pages were used in the experiments reported here simply because this is easier to implement in practice. To optimize, say, a desktop software would require users to download it and the software to be instrumented in various ways (e.g., to transmit user feedback to a remote server). Other potential applications of the technology include the following:

- Optimization of UIs (both Web and desktop) having some *usability* metric (instead of a commercial success metric such as conversion rate) as the measure of performance. Usability literature provides a number of such metrics (e.g., the time to complete tasks) [4] and even ways of combining them [22]. For instance, menus and toolbars could be automatically reorganized according to what different groups of users actually employ and to how they behave (e.g., young and elderly users, which typically have different ways of using software).
- Explicitly collaborative design applications. Although the experiment developed in this article make users collaborate implicitly (i.e., each user is not aware that he or she is helping to optimize a design), an explicit approach is possible. Users could, for example, be invited to consciously influence the design of a Web site or other UI. This could lead to a greater sense of community and perhaps motivate users to perform different or more complex interactions.
- Optimization of off-line artifacts, such as outdoor advertisements, by first performing on-line experiments (e.g., asking users to rate the quality of the advertisement). Because physical artifacts are typically more expensive than their digital representations, it is desirable to perform some optimization in their design before investing in their actual physical production.
- As an example of non-graphical UI, one can imagine a telephone service in which options are spoken to the user. How to phrase these options, as well as how to group them in each step of the overall interaction, could be subject to similar experimentation and optimization.

All of these possible applications contain the same core elements: a hierarchical structure of components, a performance metric and a set of users whose feedbacks guide the optimization.

5 EXPERIMENTAL RESULTS

The development of the approach took a number of failed experiments before it reached an effective point. This was the case because a number of details cannot be determined *a priori* and must be discovered empirically. In the present section, the main experimental result shown is a successful one, which incorporates all improvements developed in this process, reflects the algorithm given above and is used to validate it. It is instructive, however, to also examine one failed experiment, based on an earlier version of the algorithm, in order to have a better understanding of some key details that determine the effectiveness of the approach. Therefore, a such a failed experiment is presented afterwards in Section 5.1 and the critical differences are highlighted.

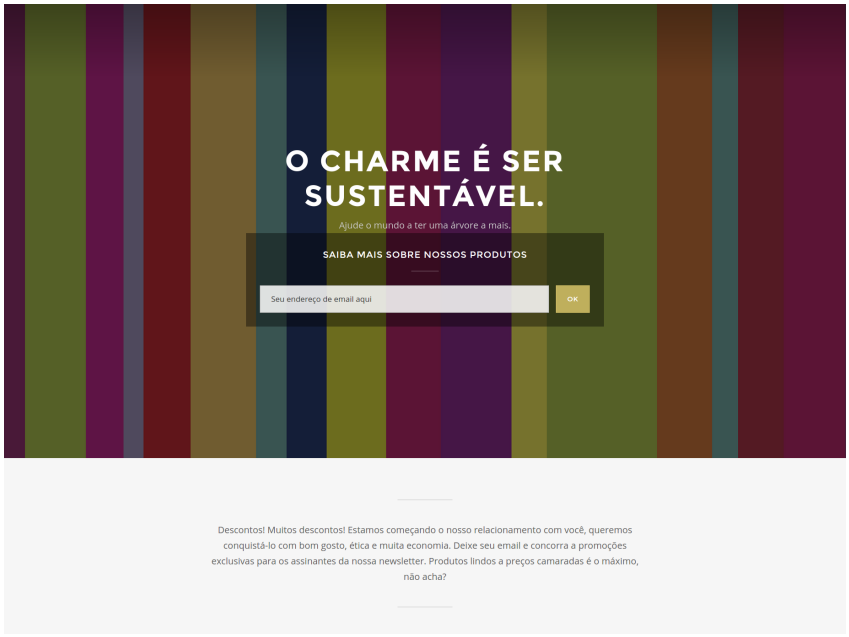


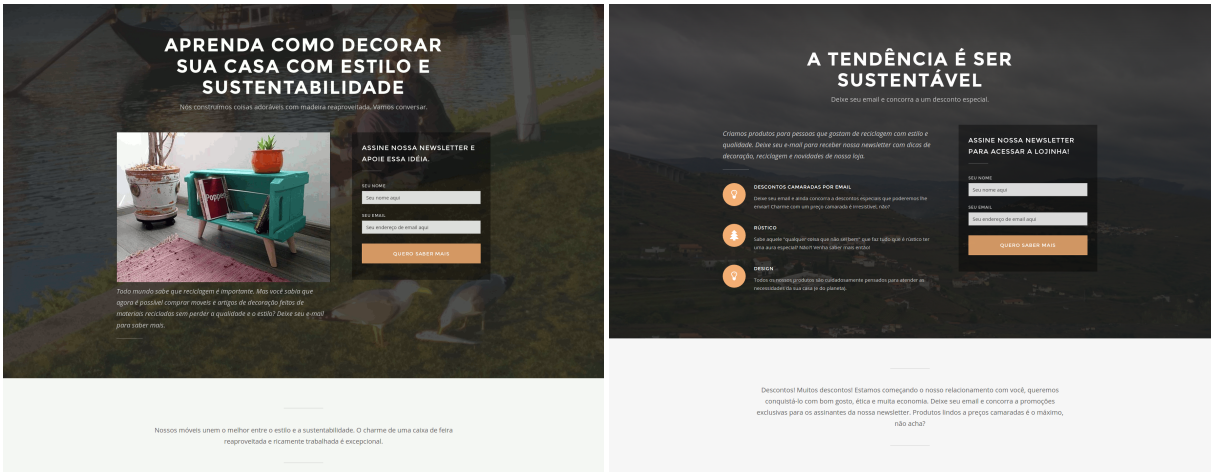
Fig. 2. The top part of one of the initial individuals, randomly composed from the specification. Notice that the layout differs considerably from what is seen in Figures 3a and 3b. Ultimately this layout was not the best, but it shows that the approach is capable of generating structurally different UIs.

The successful experiment reported here includes all data available up to the moment of writing (except for responses that were duplicated or did not provide the email address, which were removed). In total, 5,462 users were recruited to participate, half of which were used as a control group.

Users were acquired through an advertisement campaign. Once a user clicked on the advertisement, he or she was randomly redirected to one of the following locations: a control page, which is not self-adaptive (i.e., has a predefined, fixed, design) and models the store's best guess of a good landing page using the available primitive design elements (Figure 3a)[10]; and the adaptive GP-optimized page itself, which is of course the main component of the experiment. Figure 3b shows the best optimized page after the experiment was executed, whereas Figure 2 presents one of the pages in the initial population. Note, in particular, the rather different layout of the latter with respect to the former (e.g., only one text field, centralized, and with less text around it). Even more complex layout differences could be defined if desired. As argued above, because genetic programs provide a way to represent hierarchies, they are naturally capable of explicitly specifying arbitrary UI compositions. Such a capability is not present in similar approaches that employ sequences of characteristics as genomes.

To evaluate the relative performance of the optimized page versus the control one, the following procedure was adopted. Each response received (i.e., an email address) was marked either as *main* or *control*, depending on whether it was collected via the main GP-optimized page or the control page, respectively. For each sequential block of 50 such responses, the *efficiency ratio* of *main* responses per

[10]The store's owner decided what she thought would be more attractive to her customers. Hence, the control presently used is more of a sensible baseline. See Section 6 for a related discussion.



(a) The top part of the control page, which remains the same throughout the experiment.

(b) The top part of the individual with the best fitness after the experiment.

Fig. 3. Both pages are similar, but contain important differences. In (a) the title reads "Learn how to furnish your home with style and sustainability" in Portuguese, and a featured image of a product is provided on the left. In (b), the title reads "The trend is to be sustainable" in Portuguese, and a textual list of benefits is provided on the left.

control ones was calculated. A value above 1.0 indicates that the optimization is performing better than the control, whereas a value below 1.0 indicates the opposite. This emphasizes the relative nature of performance, which is always to be determined in relation to some baseline.

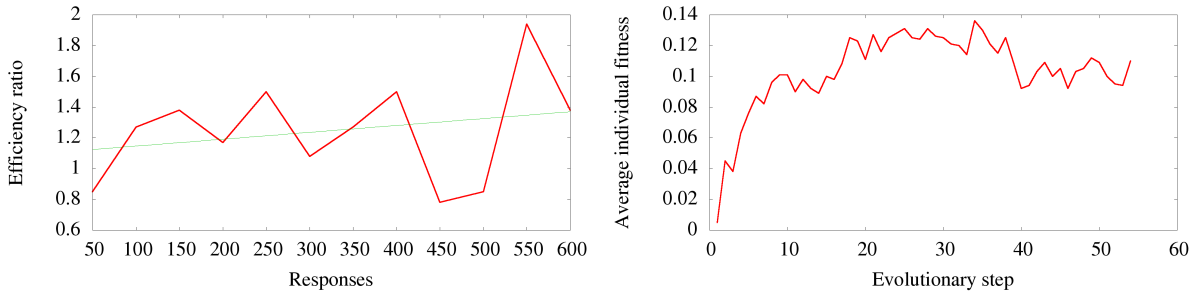
The results can be seen in Figure 4a, which was generated from the data shown in Table 1a. As expected, initially the efficiency ratio remains below 1, and then moves above 1. On average, the optimized page performed 20% better than the control one (i.e., an efficiency ratio of 1.2). In terms of conversion rates, the optimized page had 12% and the control one had 9.9%. This result is statistically significant: a T-test[11] using the data of Table 1a shows that, with $p < 0.0023$, the average number of main responses is greater than that of control responses (per block of 50 responses).

The linear upward trend, however, appears to have no statistical significance[12], which could mean that a good solution was found right in the beginning of the process. Indeed, Figure 4b shows the average fitness calculated at each evolutionary step. It fluctuates a little, but in general a convergence pattern can be seen forming around step 20, which is rather early in the overall experiment.

It is also worth to note that the larger drop in performance seen around response 450 in Figure 4a happens near a Friday. Previous smaller and more informal experiments have suggested that user behavior changes close to weekends, thus making the learning performed up to that point less effective. If this is indeed true, it would explain the observed drop and later rebound at the end of the weekend. Further experiments must be performed in order to properly determine behavior seasonality and ways to minimize its impact.

[11] Calculated using the `scipy.stats.ttest_ind` function from the SciPy library for the Python language.

[12] The first half of the efficiency ratios (i.e., up to response 300) was separated from the second half (i.e., up to response 600). A T-test considering these two groups found no evidence of a significant difference, although the average of the latter is slightly greater.



(a) The efficiency ratio per block of 50 responses.

(b) Average individual fitness per evolutionary step.

Fig. 4. As the successful experiment progresses, it is possible to calculate the efficiency ratio that measures how well the optimization is performing in relation to the control. A linear regression shows that the overall tendency is above 1.0 and increasing, as can be seen in (a). During the experiment, occasionally (as determined by the `ShouldEvolve()` function, see Section 4.3) an evolutionary step is given through the execution of Algorithm 2 and the average fitness of individuals at that moment is recorded, as shown in (b). The absolute value of the fitness is not important (it merely reflects implementation-specific decisions and can be scaled arbitrarily), only its change over time matters.

Beyond these numerical results, to inspect the actual individuals evolved can be enlightening too. For instance, 8 out of the 10 top-performing GP individuals had a mention to discounts or monetary savings (e.g., the tagline "Leave your email and get eligible for a special discount."), a feature not present in the control page. From a business perspective, this suggests that the target audience is price-sensitive, which in turn may prompt managers to take specific actions.

5.1 Fine-tuning

The algorithm and related experimental results presented above depend on a number of specific parameters and subtleties. Such a fine-tuning is not obvious. To examine the difficulties involved, let us consider a previous failed experiment. It was similar to the successful one, including its size (4,676 users were recruited, half of which formed the control group), but with three critical differences:

- The number of times a single GP individual receives feedback (i.e., is presented to a user) is different. The successful version used an average of three expressions per GP individual (see the explanation concerning the `ShouldEvolve?()` function in the fourth bullet point of Section 4.3.1), whereas the failed one used an average of only one.
- The second difference relates to Algorithm 2. In the successful version, the elimination of the worst individuals is carried out before everything, which means that the resulting population will contain all new children generated. In contrast, the failed version mistakenly performed this elimination right after reproduction, which meant that some new children could be eliminated from the population immediately after creation, before having a chance to express themselves. This was really a programming error that could have been avoided without empirical feedback, but it does highlight a subtle difference and potential pitfall when dealing with a fitness function that must be computed incrementally and not immediately.
- The third difference is the population base size. Although in principle a larger population is better because it allows more candidate solutions to be explored, in the failed experiment a larger (50) population was used in relation to the successful one (15). The results suggest that the successful

Responses	Ratio main/control	Main	Control
50	0.85	23	27
100	1.27	28	22
150	1.38	29	21
200	1.17	27	23
250	1.5	30	20
300	1.08	26	24
350	1.27	28	22
400	1.5	30	20
450	0.78	22	28
500	0.85	23	27
550	1.94	33	17
600	1.38	29	21
	1.20	328	272

(a) Data collected from the successful experiment.

Responses	Ratio main/control	Main	Control
50	1	25	25
100	0.923	24	26
150	0.724	21	29
200	1.083	26	24
250	1.380	29	21
300	1.5	30	20
350	1.5	30	20
400	0.724	21	29
450	1.272	28	22
500	0.851	23	27
550	0.515	17	33
600	1.38	29	21
638	0.72	16	22
	1.0	319	319

(b) Data collected from the failed experiment.

Table 1. Data was collected about every sequential block of 50 responses, for both the successful and the failed experiments. A ratio main/control (i.e., the efficiency ratio) above 1.0 indicates that the the optimization is producing pages that perform better than the control one; a value below 1.0 indicates the opposite.

algorithm can work even with relatively small populations and that a larger population does not seem to compensate for the other two problems reported above.

These differences led to a decreasing performance trend, although initially the optimization worked well (see Table 1b). In the end, 50% (319) of conversions came from the main page and 50% from the control page, hence no optimization actually took place.

These observations show that the various details involved are not arbitrary, but rather follow from a series of progressive improvements. They also highlight both the more sensitive parts of the approach (e.g., the average number of expressions per GP individual) and the more robust ones (e.g., population size).

6 CONCLUSION

This article showed how user interfaces can be composed out of simpler parts by using optimization technology based on Genetic Programming. The technique depends on the availability of user feedback, which is why the text and concrete example focused on Web pages, particularly landing pages, in which a performance metric is available (the so-called conversion rate) and plenty of users can be easily recruited. In the reported experiment, the GP-optimized page performed significantly better than the control page, which shows that the overall technique is promising and worthy of further research. Nevertheless, more experiments (targeting different audiences, types of page and products) and control pages completely created by independent designers are required in order to fully assess and improve the current system.

An important drawback is the fact that the genetic algorithm, in the search for good solutions, also produces and presents bad solutions to users. This problem is more prominent in initial iterations, but owing to mutation and other sources of variability it may happen at any time as long as the algorithm

is running. Whether this is tolerable or not is a business decision. A way to cope with this might be simply to run the optimization for some time, select the top-performing solution, and make it permanent. The issue can also be partly addressed by the cleverness of the designer, who may: pick primitive design elements that almost always make some sense together; and specify more complex programs that enforce various restrictions in the possible compositions.

Many technical improvements are possible. Alternatives can be devised both for the crossover (e.g., to be less conservative) and the mutation (e.g., to become a better source of variability) operators. All parameters used, such as population size and the number of new children per iteration, can probably be improved as well to deliver better population fitness or convergence times. More convenient program statements for the specification of genomes may further facilitate the task of defining sensible design possibilities, such as constraints to elaborate more concise or sophisticated specifications (e.g., to define that a certain background image can only be used with a certain set of color schemes, no matter their structural position in the program tree).

The design methodology appropriate to the presented technology can also be the subject of deeper studies. It is conceivable, for example, that some form of iterative process that alternates between the designer and the optimization algorithm could be developed. Insights from one optimization round could lead the designer to imagine new possibilities, change the design specification accordingly, start a new round and so on. Another related area of research would be how to modularize and later reuse parts of designs in different applications, which would help with scalability. In fact, there are many other similar concerns. Just like software engineers have created a profusion of different processes to allow them to work better, so too designers having such an optimization technology could develop various methods and best practices.

By automating one part of the design task, this technology makes designers more powerful: instead of devising a few very particular solutions, they become capable of investigating large families of solutions. Methodologically, this is perhaps the most interesting aspect of this and similar works. Far from eliminating the human component, automation may simply free up the human intellect for greater achievements.

ACKNOWLEDGMENTS

The author would like to thank Agnes Helena Chiuratto for permitting the use of her store's brand to recruit customers and perform the experiment described in Section 5.

REFERENCES

[1] John Biles. 1994. GenJam: A genetic algorithm for generating jazz solos. In *Proceedings of the International Computer Music Conference*. 131–131.

[2] Kenneth A De Jong. 2006. *Evolutionary computation: a unified approach*. MIT press.

[3] Greg Dingle. 2013. An unobtrusive way to A/B test and optimize webpages. `https://github.com/gregdingle/genetify`. (2013). Last accessed February 1st, 2016.

[4] Joseph S Dumas and Janice Redish. 1999. *A practical guide to usability testing*. Intellect books.

[5] Douglas C Engelbart. 1962. *Augmenting human intellect: a conceptual framework*. Technical Report. Stanford Research Institute.

[6] Krzysztof Z Gajos, Daniel S Weld, and Jacob O Wobbrock. 2010. Automatically generating personalized user interfaces with SUPPLE. *Artificial Intelligence* 174, 12-13 (2010), 910–950.

[7] Richard Gatarski. 2002. Breed better banners: Design automation through on-line interaction. *Journal of Interactive Marketing* 16, 1 (2002), 2–13.

[8] Mark Harman and Bryan F Jones. 2001. Search-based software engineering. *Information and Software Technology* 43, 14 (2001), 833–839.

[9] Ron Kohavi and Roger Longbotham. 2016. Online Controlled Experiments and A/B Testing. In *Encyclopedia of Machine Learning and Data Mining*, Claude Sammut and Geoffrey I. Webb (Eds.). Springer US, Boston, MA, 1–8.

[10] Kris Luyten, Tim Clerckx, Karin Coninx, and Jean Vanderdonckt. 2003. Derivation of a dialog model from a task model by activity chain extraction. In *International Workshop on Design, Specification, and Verification of Interactive Systems*. Springer, 203–217.
[11] Melanie Mitchell. 1998. *An introduction to genetic algorithms*. MIT press.
[12] N. Monmarche, G. Nocent, M. Slimane, G. Venturini, and P. Santini. 1999. Imagine: a tool for generating HTML style sheets with an interactive genetic algorithm based on genes frequencies. In *Proceedings of the IEEE International Conference on Systems, Man, and Cybernetics.*, Vol. 3.
[13] Jeffrey Nichols, Duen Horng Chau, and Brad A Myers. 2007. Demonstrating the viability of automatically generated user interfaces. In *Proceedings of the SIGCHI conference on Human factors in computing systems*. ACM, 1283–1292.
[14] Jeffrey Nichols, Brad A Myers, and Kevin Litwack. 2004. Improving automatic interface generation with smart templates. In *Proceedings of the 9th International Conference on Intelligent User Interfaces*. ACM, 286–288.
[15] Martin Odersky, Philippe Altherr, Vincent Cremet, Burak Emir, Sebastian Maneth, Stéphane Micheloud, Nikolay Mihaylov, Michel Schinz, Erik Stenman, and Matthias Zenger. 2004. *An overview of the Scala programming language*. Technical Report.
[16] A. Oliver, N. Monmarch, and G. Venturini. 2002. Interactive Design Of Web Sites With A Genetic Algorithm. In *Proceedings of the IADIS International Conference WWW/Internet*.
[17] Matthias Peissner, Dagmar Häbe, Doris Janssen, and Thomas Sellner. 2012. MyUI: generating accessible user interfaces from multimodal design patterns. In *Proceedings of the 4th ACM SIGCHI Symposium on Engineering Interactive Computing Systems*. ACM, 81–90.
[18] Riccardo Poli and Stefano Cagnoni. 1997. Genetic Programming with User-Driven Selection: Experiments on the Evolution of Algorithms for Image Enhancement. In *Genetic Programming 1997: Proceedings of the Second Annual Conference*. Morgan Kaufmann, 269–277.
[19] Riccardo Poli, William B. Langdon, and Nicholas Freitag McPhee. 2008. *A field guide to genetic programming*. Published via `http://lulu.com` and freely available at `http://www.gp-field-guide.org.uk`. (With contributions by J. R. Koza).
[20] Juan C Quiroz, Sushil J Louis, Anil Shankar, and Sergiu M Dascalu. 2007. Interactive genetic algorithms for user interface design. In *IEEE Congress on Evolutionary Computation (CEC 2007)*. IEEE, 1366–1373.
[21] David Raneburger, Hermann Kaindl, and Roman Popp. 2015. Strategies for Automated GUI Tailoring for Multiple Devices. In *System Sciences (HICSS), 2015 48th Hawaii International Conference on*. IEEE, 507–516.
[22] Jeff Sauro and Erika Kindlund. 2005. A method to standardize usability metrics into a single score. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. ACM, 401–409.
[23] Jimmy Secretan, Nicholas Beato, David B. D Ambrosio, Adelein Rodriguez, Adam Campbell, and Kenneth O. Stanley. 2008. Picbreeder: Evolving Pictures Collaboratively Online. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '08)*. ACM, 1759–1768.
[24] James C Spall. 2005. *Introduction to stochastic search and optimization: estimation, simulation, and control*. Vol. 65. John Wiley & Sons.
[25] Hideyuki Takagi. 2001. Interactive evolutionary computation: Fusion of the capabilities of EC optimization and human evaluation. *Proc. IEEE* 89, 9 (2001), 1275–1296.
[26] Giordano Tamburrelli and Alessandro Margara. 2014. Towards Automated A/B Testing. In *Search-Based Software Engineering*, Claire Le Goues and Shin Yoo (Eds.). Lecture Notes in Computer Science, Vol. 8636. Springer International Publishing, 184–198.
[27] Michael Toomim, Travis Kriplean, Claus Pörtner, and James Landay. 2011. Utility of human-computer interactions: Toward a science of preference measurement. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. ACM, 2275–2284.
[28] Vi Tran, Jean Vanderdonckt, Ricardo Tesoriero, and François Beuvens. 2012. Systematic generation of abstract user interfaces. In *Proceedings of the 4th ACM SIGCHI symposium on Engineering interactive computing systems*. ACM, 101–110.
[29] Jean Vanderdonckt. 2008. Model-driven engineering of user interfaces: Promises, successes, failures, and challenges. *Proceedings of ROCHI*. 8 (2008).