The Anatomy of a Sales Configurator: An Empirical Study of 111 Cases

Ebrahim Khalil Abbasi, Arnaud Hubaux, Mathieu Acher, Quentin Boucher, Patrick Heymans

▶ **To cite this version:**

HAL Id: hal-00796555

https://hal.inria.fr/hal-00796555

Submitted on 8 Sep 2013

The Anatomy of a Sales Configurator:
An Empirical Study of 111 Cases

Ebrahim Khalil Abbasi[1], Arnaud Hubaux[1], Mathieu Acher[2],
Quentin Boucher[1], and Patrick Heymans[1]

[1] PReCISE, University of Namur, Belgium
{eab,ahu,qbo,phe}@info.fundp.ac.be
[2] University of Rennes 1, Irisa, Inria France
mathieu.acher@irisa.fr

Abstract. Nowadays, mass customization has been embraced by a large
portion of the industry. As a result, the web abounds with sales config-
urators that help customers tailor all kinds of goods and services to
their specific needs. In many cases, configurators have become the sin-
gle entry point for placing customer orders. As such, they are strategic
components of companies' information systems and must meet stringent
reliability, usability and evolvability requirements. However, the state of
the art lacks guidelines and tools for efficiently engineering web sales
configurators. To tackle this problem, empirical data on current practice
is required. The first part of this paper reports on a systematic study of
111 web sales configurators along three essential dimensions: rendering
of configuration options, constraint handling, and configuration process
support. Based on this, the second part highlights good and bad prac-
tices in engineering web sales configurator. The reported quantitative
and qualitative results open avenues for the elaboration of methodolo-
gies to (re-)engineer web sales configurators.

Keywords: Configuration, Web, Variability, Reverse Engineering, Em-
pirical Study, Survey

1   Introduction

In many markets, being competitive echoes with the ability to propose cus-
tomised products at the same cost and delivery rate as standard ones. These
customised products are often characterised by hundreds of *configuration op-
tions*. For many customers, this repertoire of inter-related options can be dis-
concerting. To assist them during decision making, *sales configurators* (SCs)
were developed. As an example, Figure 1 shows a snapshot of a typical car con-
figurator (the circled letters and legend can be ignored for now). A SC provides
an interactive graphical user interface (GUIs) that guides the users through the
configuration process, verifies constraints between options, propagates user de-
cisions, and handles conflictual decisions [1–3].

SCs represent a significant portion of the configurators used in modern in-
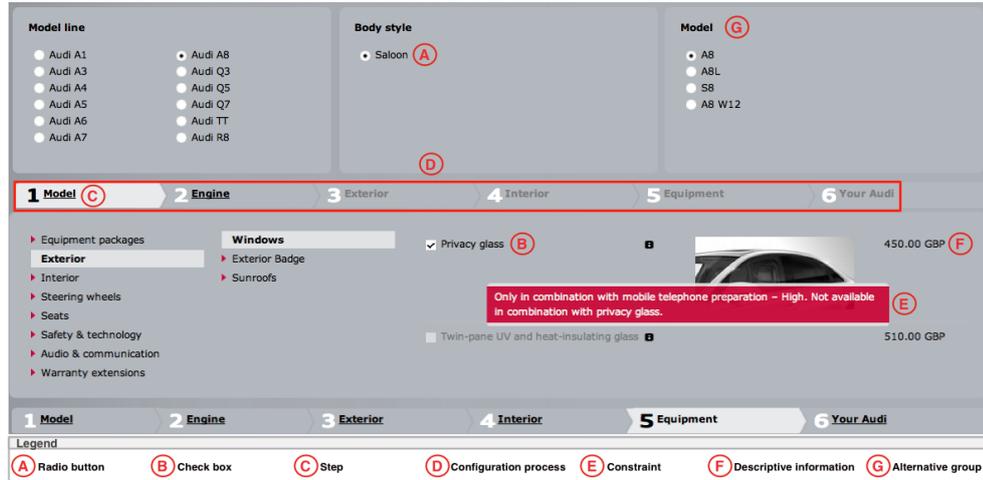formation systems. Configurators are used in many B2B and B2C applications

Fig. 1. Audi web SC (`http://configurator.audi.co.uk/`, Oct. 18, 2012)

to personalize products and services. They are used in installation wizards and preference managers. They are also extensively used in software product lines (SPLs) where multiple information system variants are derived from a base of reusable artefacts according to the specific characteristics of the targeted customer or market segment [4–7]. As privileged channels for identifying customer needs and placing orders, configurators are key assets for companies. In this paper, we focus on web configurators supporting online sales.

A significant share of existing SCs is web-based, irrespective of the market. The configurator database maintained by Cyledge is a striking evidence [8]. Since 2007, Cyledge collected more than 800 web SCs coming from 29 different industry sectors, including automotive, apparel, sport, and art. These configurators vary significantly. They each have their own characteristics, spanning visual aspects (GUI elements) to constraint management. The web SC of Audi appearing in Figure 1 is thus only one example. It displays different options through specific widgets (radio buttons and check boxes – Ⓐ and Ⓑ, respectively). These options can be in different states such as activated (e.g., "Privacy glass" is flagged with ✓) or unavailable (e.g., "Twin-pane UV and heat-insulating glass" is greyed out). Additionally, these options are organised in different tabs (e.g., "Equipment") and sub-tabs (e.g., "Equipment packages") which denote a series of steps (Ⓒ) in the *configuration process* (e.g., "1. Model" is followed by "2. Engine" – Ⓓ). A SC can also implement *cross-cutting*[3] *constraints* between options (Ⓔ). These are usually hidden to the user but they determine valid combinations of options. For instance, the selection of "Privacy glass" implies the deselection of "Twin-pane UV and heat-insulating glass", meaning that the user cannot select the latter if the former is selected. Moreover, descriptive information (Ⓕ) is sometimes associated to an option (e.g., its price).

---

[3] We call these constraints *cross-cutting* because they are often orthogonal to the hierarchy of options, sub-options, etc. supported by the configurator.

Despite the abundance of SCs, a consistent body of knowledge dedicated to their engineering is still missing. This absence of standard guidelines often translates into correctness or runtime efficiency issues, mismatches between the constraints exposed to the user and those actually implemented, and an unclear separation of concerns between the GUI and business logic. These issues in turn lead to expensive development and maintenance. Some of our industry partners face similar problems and are now trying to migrate their legacy SCs to more reliable, efficient, and maintainable solutions [9]. Our long-term objective is to develop a set of methods, guidelines, languages, and tools to systematically *(re-)engineer* SCs. This encompasses three activities: *(a)* reverse engineering legacy SCs, *(b)* encoding the extracted data into dedicated formalisms, and *(c)* forward engineering new improved SCs [9].

However, to realize this vision, we first need to understand the intrinsic characteristics of SCs. We conduct an empirical study of 111 web SCs from 21 different industry sectors (Section 2). We analyze the client-side code of these SCs with semi-automated code inspection tools. We classify and analyse the results along three dimensions: *configuration options*, *constraints*, and *configuration process* (Section 3). For each dimension, we present quantitative empirical results and report on good and bad practices we observed (Section 4). We also describe the reverse engineering issues we faced (Section 5). We discuss the threats to validity (Section 6) and related work (Section 7). Finally, we summarize the results and propose a research agenda for the (re-)engineering of SCs (Section 8).

2   Problem Statement and Method

(Re-)engineering web SCs requires a deep understanding of how they are currently implemented. We choose to start this journey by analysing the visible part of SCs: the *web client*. We analyse client-side code because (1) it is the entry point for customer orders, (2) the techniques used to implement web clients and web servers differ significantly, and (3) large portions of that code are publicly available. We leave for future work the study of server-side code and the integration of client- and server-side analyses. In this paper, we set out to answer three research questions:

RQ1 *How are configuration options visually represented and what are their semantics?* By nature, SCs rely on GUIs to display configuration options. In order to re-engineer configurators, we first need to identify the types of widgets, their frequency of use, and their semantics (e.g., optionality, alternatives, multiple choices, descriptive information, cloning, and grouping).

RQ2 *What kinds of constraints are supported by the SCs, and how are they enforced?* The selection of options is governed by constraints. These constraints are often deemed complex and non-trivial to implement. We want to grasp their actual complexity.

RQ3 *How is the configuration process enforced by the configurators?* The configuration process is the interactive activity during which users indicate the

options to be included and excluded in the final product. It can, for instance, either be *single-step* (all the available options are presented together to the user) or *multi-step* (the process is divided into several steps, each containing a subset of options). Another criteria is navigation flexibility.

2.1 Configurator selection

To collect a representative sample of web SCs, we used Cyledge's configurator database, which contains 800+ entries from a wide variety of domains. The first step of our configurator selection process consisted in filtering out non-English configurators. For simplicity, we only kept configurators registered in one of these countries: *Australia*, *Britain*, *Canada*, *Ireland*, *New Zealand*, and *USA*. This returned 388 configurators and discarded four industry sectors. Secondly, we excluded 26 configurators that are no longer available. We considered a site unavailable either when it is not online anymore or requires credentials we do not have. Thirdly, we randomly selected 25% of the configurators in each sector. We then checked each selected configurator with *Firebug*[4] to ensure that configuration options, constraints, and constraint handling procedures do not use Flash. We excluded configurators using Flash because the Firebug extension we implemented (see next section) does not support that technology. We also excluded "false configurators". By this we mean 3D design websites that allow to build physical objects by piecing graphical elements together, sites that just allow to fill simple forms with personal information, and sites that only describe products in natural language. The end result is a sample set of 93 configurators from 21 industry sectors. Finally, we added 18 configurators that we already knew for having used them in preliminary stages of this study. We used them to become familiar with web SCs and test/improve our reverse engineering tools, as discussed below. This raised the total number of web SCs to 111.

2.2 Data extraction process

To answer the first two research questions, we need to extract the types of widgets used to represent options (RQ1), the types of constraints and their implementation strategies (RQ2). To extract all this information, we developed a Firebug extension (3 KLOC, 1 person-month) that implements (*a*) a supervised learning-based data extraction approach [10], (*b*) support for advanced searches, and (*c*) DOM[5] traversing.

   Our approach relies on a training session during which we inspect the source code of the web page to identify which code patterns are used to implement configuration options and their graphical widgets. These patterns vary from simple (e.g., *tag[attribute:value]*) to complex cases (e.g., a sequence of HTML tags). We then feed these patterns to our Firebug extension to extract all options. In essence, our extension offers a *search engine* able to (*a*) *search* given code

---

[4] http://getfirebug.com/
[5] Document Object Model: a standard representation of the objects in an HTML page.

patterns, and (*b*) *simulate* user actions. It uses jQuery selectors and code clone detection to search matching elements, extract an option name, its widget type, place of occurrence, and discover constraints between options (RQ3). Practically, the simulator selects/deselects each option and logs the existence of possible constraints based on the previous state of the page.

3   Quantitative Results

This section summarises the results of our empirical study[6]. Table 1 highlights our key findings. Each subsection answers the questions posed in Section 2.

3.1   Configuration options (RQ1)

Option Representation. The diversity of representations for an option is one of the most striking results, as shown in Figure 2. In decreasing order, the most popular widgets are: *combo box item*, *image*[7], *radio button*, *check box* and *text box*. We also observed that some widgets were combined with images, namely, *check box*, *radio button*, and *combo box item*. Option selection is performed by either choosing the image or using the widget. The *Other* category contains various less frequent widgets like *slider*, *label*, *file chooser*, *date picker*, *colour picker*, *image needle*, and *grid*.

   Grouping. Grouping is a way to organise *related* options together. For instance, a group can contain a set of colours or the options for an engine. Three different semantic constraints can apply to a group. For *alternative* groups, one and only one option must be selected (e.g., the *Models* in Figure 1 – Ⓖ), and for *multiple choice* groups, at least one option must be selected (e.g., stone and band to put on a ring). In an *interval* group (a.k.a. cardinality [11]), the specific lower and upper bounds on the number of selectable options is determined (e.g., flavours in a milkshake). The *Semantic Constructs* row in Table 1 shows that *alternative* groups are the most frequent with 97% of SCs implementing them. We also observed multiple choice questions and interval groups in 8% and 4% of configurators, respectively.

   "Mandatory Options" and "Optional Options". Non-grouped options can be either mandatory (the user has to enter a value) or optional (the user does not have to enter a value). By definition, configurators must ensure that all mandatory options are properly set before finishing the configuration process. We identified three patterns for dealing with mandatory options:

 – *Default Configuration* (46%): When the configuration environment is loaded, (some or all) mandatory options are selected or assigned a default value.
 – *Notification* (47%): Constraints are checked at the end of the configuration process and mandatory options left undecided are notified to the user. This approach can be mixed with default values.

---

[6] The complete set of data is available at `http://info.fundp.ac.be/~eab/result.html`.

[7] A colour to choose from a palette is also considered an image.

Table 1. Result summary

CONFIGURATION OPTIONS		
Semantic Constructs	Alternative group	97%
	Multiple choice group	8%
	Interval	4%
Mandatory Options	Default	46%
	Notification	47%
	Transition Checking	13%
	No checking	4%
Multiple instantiation	Cloning	5%
CONSTRAINTS		
Constraint Type	Formatting	59%
	Group	99%
	Cross-cutting	55%
Cross-cutting Constraint (61)	Visibility	89%
Formatting Constraint (66)	Prevention	62%
	Verification	41%
	No checking	26%
Constraint Description (61)	Explanation	11%
Decision Propagation (61)	Automatic	97%
	Controlled	8%
	Guided	3%
Consistency Checking (83)	Interactive	76%
	Batch	59%
Configuration Operation	Undo	11%
CONFIGURATION PROCESS		
Process	Single-step	48%
	Basic Multi-step	45%
	Hierarchical Multi-step	7%
Activation (58)	Step-by-step	59%
	Full-step	41%
Backward Navigation (58)	Stateful arbitrary	69%
	Stateless arbitrary	14%
	Not supported	17%

– *Transition Checking* (13%): The user is not allowed to move to the next step until all mandatory options have been selected. The difference with the previous pattern is that no warning is shown to the user.

We noticed that 4% of the configurators either lack interactive strategies for handling mandatory options or have only optional options.

Mandatory options can be distinguished from optional ones through *highlighting*. For that, SCs use symbolic annotations (e.g., * usually for mandatory options), textual keywords (e.g., *required*, *not required*, or *optional*), or special

text formatting (e.g., boldfaced, coloured text). We observed that only 14% of the SCs highlight mandatory or optional options, while 70% of the SCs have optional options in their configuration environments.

Cloning. Cloning means that the user determines how many instances of an option are included in the final product [12] (e.g., a text element to be printed on a t-shirt can be instantiated multiple times and configured differently). We observed cloning mechanisms in only 5% of the configurators.
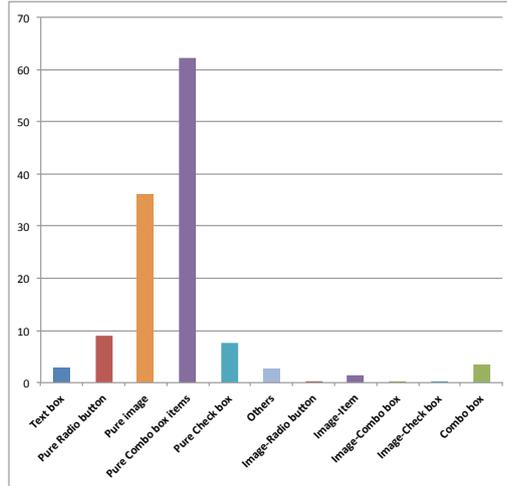

Fig. 2. Widget types in all the configurators

3.2   Constraints (RQ2)

Formatting Constraint. A formatting constraint ensures that the value set by the user is *valid*. Examples are: (1) *type correctness* (strongly typed, e.g. String, Integer, and Real), (2) *range control* (e.g., upper and lower bounds, slider domain, and valid characters), (3) *formatted values* (e.g., date, email, and file extension), and (4) *case-sensitive values*.

We observed that configurators provide two different patterns for checking constraint violation. The first is to *prevent* illegal values. For example, stop accepting input characters if the maximum number is reached, define a slider domain, use a date picker, disable illegal options, etc. The second is to *verify* the values entered by the user a posteriori, and, for example, highlight, remove or correct illegal values. These patterns are not mutually exclusive, and configurators can use them for different subsets of options. Among the 66 configurators supporting formatting constraints, 62% implement prevention and 41% implement verification patterns. We also noticed that 26% of the configurators do not check constraints during the configuration session even if they are described in the interface. In some rare cases, the validation of the configuration was performed off-line, and feedback later sent back to the user.

Group Constraint. A group constraint defines the number of options that can be selected from a group of options. In essence, constraints implied by *multiple choice-*, *alternative-* and *interval-*groups are group constraints. Widget types used to implement these groups directly handle those constraints. For instance, radio buttons and single-selection combo boxes are commonly used to implement *alternative* groups. We identified group constraints in 99% of the analysed configurators.

Cross-cutting Constraint. A cross-cutting constraint is defined over two or more options regardless of their inclusion in a group. *Require* (selecting A implies selecting B) and *Exclude* (selecting A prevents selecting B and vice-versa) constraints are the most common. More complex constraints exist too. Cross-cutting constraints were observed in 61 configurators (55%) and are either coded in the client side (e.g., using JavaScript) or in the server side (e.g., using PHP). Irrespective of the implementation technique, we noticed that only 11% of the configurators describe them in the GUI with a textual explanation.

Visibility Constraint. Some constraints determine when options are shown or hidden in the GUI. They are called *visibility constraint* [13]. Automatically adding options to a combo box upon modification of another option also falls in this constraint category. From the 61 configurators with cross-cutting constraints, 89% implement visibility constraints.

We now focus on the capabilities of the reasoning procedures, namely *decision propagation*, *consistency checking* and *undo*.

Decision Propagation. In some configurators, when an option is given a new value and one or more constraints apply, the reasoning procedure automatically propagates the required changes to all the impacted options. We call it *automatic* propagation (97%). In other cases, the reasoning procedure asks to confirm or discard a decision before altering other options. We call this *controlled* propagation (8%). Finally, we also observed some cases of *guided* propagation (3%). For example, if option A requires to select option B or C, the reasoning procedure cannot decide whether B or C should be selected knowing A. In this case, the configurator proposes a choice to the user. Some of the configurators implement multiple patterns.

Consistency Checking. An important issue in handling formatting and cross-cutting constraints is *when* the reasoning procedure instantiates the constraints and checks the consistency. In an *interactive* setting, the reasoning procedure interactively checks that the configuration is still consistent as soon as a decision is made by the user. For example, the permanent control of the number of letters in a text field with a maximum length constraint is considered interactive. In some cases, the reasoning procedure checks the consistency of the configuration upon request, for instance, when the user moves to the next configuration step. We call this *batch* consistency checking. Among the 83 configurators supporting both formatting and cross-cutting constraints, 76% implement interactive and 59% implement batch consistency checking patterns. Some configurators implement both mechanisms, depending on the constraint type.

Undo. This operation allows users to roll back on their last decision(s). Among all configurators in the survey, only 11% support undo.

3.3   Configuration process (RQ3)

Process pattern. A configuration process is divided into a sequence of steps, each of which includes a subset of options. Each step is also visually identified in the GUI with containers such as navigation tabs, menus, etc. Users follow these steps to complete the configuration. We identified three different configuration process patterns:

- *Single-step* (48%): All the options are displayed to the user in a single graphical container.
- *Basic Multi-step* (45%): The configurator presents the options either across several graphical containers that are displayed one at a time, or in a single container that is divided into several observable steps.
- *Hierarchical Multi-step* (7%): It is the same as a multi-step except that a step can contain inner steps.

Activation. Among the 58 multi-step configurators, we noticed two exclusive *step activation* strategies. In *step-by-step activation* (59%), only the first step is available and the other steps become available as soon as all the options in the previous step have been configured. Alternatively, in a *full-step activation* (41%) strategy, all steps are available to the user from the beginning.

Backward navigation. Another important parameter in multi-step configuration processes is the ability to navigate back to a previous step. In some configurators, the user can go back to any previous step and all configuration choices are saved. We call it the *stateful arbitrary* pattern (69%). In other cases, the user can go back to any previous step but all configuration choices made in steps following the one reached are discarded. We call it *stateless arbitrary* pattern (14%). We observed that all full-step activation configurators follow the stateful arbitrary navigation pattern. We also noticed that 17% of multi-step configurators do not support backward navigation.

4   Qualitative Results

The previous section focused on technical characteristics of SCs. We now take a step back from the code to look at the results from the qualitative and functional angles. We discuss below the bad and good practices we observed. This classification reflects our practical experience with configurators and general knowledge reported in the literature [1–3, 14, 15]. Note that the impact of marketing or sales decisions on the behaviour of SCs falls outside our scope of investigation. We focus here on their perception by end-customers that are likely to influence the way SCs are implemented.

4.1   Bad practices

– *Absence of propagation notification*: In many cases, options are automatically enabled/disabled or appear/disappear without notice. This makes configuration confusing especially for large multi-step models as the impact of a decision becomes impossible to predict and visualise. 97% of the configurators automatically propagate decisions but rarely inform users of the impact of their decisions.
– *Incomplete reasoning:* Reasoning procedures are not always complete. Some configurators do not check that mandatory options are indeed selected, or do not verify formatting constraints. 26% of the configurators do not check formatting constraints during the configuration session.
– *Counter-intuitive representation*: The visual discrepancies between option representations are striking. This is not a problem *per se*. The issue lies in the improper characterisation of the semantics of the widgets. For instance, some exclusive options are implemented by (non exclusive) check boxes. Consequently, users only discover the grouping constraint by experimenting with the SC, which causes confusion and misunderstanding. It also increases the risk of inconsistency between the intended and implemented behaviour.
– *Stateless backward navigation*: Stateless configurators lose all decisions when navigating backward. This is a severe defect since users are extremely likely to make mistakes or change their mind on some decisions. 31% of the configurators do not support backward navigation or are stateless.
– *Automatic step transition*: The user is guided to the next step once all options are configured. Although this is a way to help users [1], it also reduces control over configuration and hinders decision review.
– *Visibility Constraints*: When a visibility constraint applies, options are hidden and/or deactivated. This reduces the solution space [14] and avoids conflictual decisions. However, the downside is that to access hidden/deactivated options, the user has to first undo decisions that instantiated the visibility constraint. These are known problems in configuration [15] that should be avoided to ensure a satisfying user experience. 89% of the SCs with cross-cutting constraints support visibility constraints.
– *Decision revision*: In a few cases, configurators neither provide an undo operation nor allow users to revise their decisions. In these cases, users have to start from scratch each time they want to alter their configuration.

4.2   Good practices

– *Guided Consistency Checking*: 3% of the SCs assist users during the configuration process by, for instance, identifying conflictual decisions, providing explanations, and proposing solutions to resolve them. These are key operations of explanatory systems [14], which are known to improve usability [1].
– *Auto-completion* allows users to configure some desired options and then let the SC complete undecided options [16]. Auto-completion is typically

useful when only few options are of interest for the user. Common auto-completion mechanisms include default values. Web SCs usually support auto-completion by providing default configuration for mandatory options.

– *Self-explanatory process*: A configurator should provide clear guidance during the configuration process [1, 2, 14]. The multi-step configurators we observed use various mechanisms such as numbered steps, "previous" and "next" buttons, the permanent display of already selected options, a list of complete/incomplete steps, etc. Configurators should also be able to explain constraints "on the fly" to the users. This is only available in 11% of the configurators.

– *Self-explanatory widgets*: Whenever possible, configurators should use standard widget types, explicit bounds on intervals, optional/mandatory option differentiation, item list sorting and grouping in combo boxes, option selection/deselection mechanisms, filtering or searching mechanisms, price live update, spell checker, default values, constraints described in natural language, and examples of valid user input.

– *Stateful backward navigation* and *undo*: These are must-have functionalities to allow users to revise their decisions. Yet, only 69% and 11%, respectively, of the web SCs do support them.

5 Reverse Engineering Challenges

Our long-term objective, *viz.* developing methods to systematically re-engineer web SCs, requires accurate data extraction techniques. For the purpose of this study, we implemented a semi-automated tool to retrieve options, constraints and configuration processes (see Section 2.2). This tool can serve as a basis for the reverse-engineering part of the future re-engineering toolset. This section outlines the main technical challenges we faced and how we overcame them. The impact of our design decisions on our results are explored in the next section. The envisaged future developments are sketched in the conclusion.

Discarding irrelevant data. To produce accurate data, we need to sort out relevant from irrelevant data. For instance, some widgets represent configuration while others contain product shipment information, agreement check boxes, etc. A more subtle example is the inclusion of *false* options such as blank (representing "no option selected"), *none* or *select an item* values in combo boxes. Although obviously invalid, values such as *none* indicate optionality, which must be documented. To filter out false positive widgets, we either delimited a search region in the GUI, or forced the search engine to ignore some widgets (e.g., widgets with a given *[attribute:value]* pair).

Unconventional widget implementations. Some standard widgets, like radio buttons and check boxes, had unconventional implementations. Some were, for instance, implemented with images representing their status (selected, rejected, undecided, etc.). This forced us to use image-based search parameters to extract the option types and interpret their semantics. To identify those parameters, we had to manually browse the source of the web page to map peculiar implementations to standard widget types.

Detecting constraints. To detect cross-cutting constraints, we simulate click events of the user, i.e., selecting/deselecting options. When an event is triggered, we monitor changes in the states of the options to track the presence of a constraint. We had to take the cascading effect of variable changes into account in order to identify constraints *individually* rather than as a monolithic block. Once detected, the constraint is extracted. Once again, the differences in *nature* (e.g., natural language descriptions of options, requires/excludes attributes, on-the-fly injection of new content...), *technology* and *implementation tactics* further complicated our task. We thus had to define heuristics to extract valuable data from these different contents as well as from the DOM tree.

Discriminating between option groups and configuration steps. An option group and a configuration step are both option containers. But while the former describes logical dependencies between options, the latter denotes a process. To classify those containers, we defined four criteria: (1) a step is a coarse-grained container, meaning that a step might include several groups; (2) steps might be numbered; (3) the term 'step' or its synonyms are used in labels; and (4) a step might capture constraints between options. If these criteria did not determine whether it was a step or a group, we considered it a group.

The above issues give a sense of the challenges that we had to face for extracting relevant data from the SCs. They are the basic data extraction heuristics that a SC reverse-engineering tool should follow, and hence represent a major step towards our long-term goal.

6 Threats to Validity

The main *external* threat to validity is our web SC selection process. Although we tried to collect a representative total of 111 configurators from 21 industry sectors, we depend on the representativeness of the sample source, i.e. Cyledge's database.

The main *internal* threat to validity is that our approach is semi-automated. First, the reliability of the developed reverse engineering techniques might have biased the results. Our tool extracts options and detects cross-cutting constraints by using jQuery selectors, a code clone detector, and a simulator. For instance, to detect all cross-cutting constraints, all possible option combinations must be investigated but combinatorial blowup precludes it. The impact this has on the completeness of our results is hard to predict. This, however, does not affect our observations related to the absence of verification of constraints textually documented in the web pages.

Second, arbitrary manual decisions had to be made when analysing configurators. For example, some configurators allow to customise several product categories. In such cases, we randomly selected and analysed one of them. If another had been chosen, the number of options and constraints could have been different. We also had to manually select some options to load invisible options in the source code. We have probably missed some.

The manual part of the study was conducted by the first author. His choices, interpretations and possible errors influenced the results. To mitigate this threat, the authors of the paper interacted frequently to refine the process, agree on the terminology, and discuss issues, which eventually led to redoing some analyses. The collected data was regularly checked and heavily discussed. Yet, a replication study could further increase the robustness of the conclusions.

7   Related Work

Variability.   Valid combinations of configuration options are often referred to as *variability* in the academic community. Over the years, academic research has defined and studied many variability modelling languages such as feature models and decision models [17]. Yet, thorough evaluations of the adequacy and impact of such languages in practice are still missing [18], specifically for configurators. A notable exception is Berger *et al.* [13] who study two variability modelling languages used in the operating system domain. The authors compared the syntax, semantics, usage and GUI-based configuration tools of the two languages. They focus on one domain and two configurators while we study how variability concepts are implemented in a wide range of web SCs. Several authors have already addressed the (semi-automatic) reverse engineering of variability models from existing artefacts [19–26]. Sources include user documentation, natural language requirements, formal requirements, product descriptions, dependencies, source code, architecture, etc. To the best of our knowledge, none of existing reverse-engineering approaches tackles the extraction of variability patterns from SCs.

   GUIs and Web.   Approaches have been proposed to reverse engineer GUIs and web pages. Memon *et al.* proposed "GUI ripping" to extract models of the GUI's structure and behaviour for testing [27]. Staiger presents an approach to detect GUI widgets and their hierarchy [28]. With VAQUISTA [29], Vanderdonckt *et al.* reverse engineered the presentation model of a web page. The WARE approach seeks to understand, maintain and evolve undocumented web applications by reverse engineering them to UML diagrams [30]. None of these approaches considers configuration aspects (e.g., configuration semantics of widgets) nor specific properties of web SCs.

   Studies of Configurators.   Rogoll *et al.* [1] performed a qualitative study of 10+ web SCs. The authors reported on *usability* and how visual techniques assist customers in configuring products. Our study is larger (100+ configurators), and our goal and methodology differ significantly. We aim at understanding how the underlying concepts of web SCs are represented, managed and implemented, without studying specifically the usability of web SCs. Yet, the quantitative and qualitative insights of our study can be used for this purpose. Streichsbier *et al.* [2] analysed 126 web SCs among those in [8]. The authors question the existence of *standards* for GUI (frequency of product images, back- and forward-buttons, selection boxes, etc.) in three industries. Our study is more ambitious and also includes non-visual aspects of web SCs. Interestingly, our findings can help identify and validate existing standards in web SCs. For example, our study

reveals that in more than half of the SCs the selected product components are summarised at the end of the process, which is in line with [2]. Trentin *et al.* [3] conducted a user study of 630 web SCs to validate five capabilities: *focused navigation*, *flexible navigation*, *easy comparison*, *benefit-cost communication*, and *user-friendly product-space description*. We adopted a more technical point of view. Moreover, their observations are purely qualitative and no automated reverse engineering procedure is applied to produce quantitative observations.

8   Conclusion

In this paper, we presented an empirical study of 111 web SCs along three dimensions: configuration options, constraints and configuration process.

Quantitative insights. We quantified numerous properties of SCs using code inspection tools. Among a diversity of widgets used to represent *configuration options*, combo box items and images are the most common. We also observed that in many cases configuration options, though not visually grouped together, logically dependent on one another: more than half of the configurators have cross-cutting *constraints*, which are implemented in many different ways. As for the *configuration process*, half of the configurators propose multi-step configuration, two thirds of which enable stateful backward navigation.

Qualitative insights. The empirical analysis of web SCs reveals reliability issues when handling constraints. These problems come from the configurators' lack of convincing support for consistency checking and decision propagation. For instance, although verifying mandatory options and constraints are basic operations for configurators, our observations show that they are not completely implemented. Moreover, the investigation of client-side code implementation verifies, in part, that no systematic method (e.g., solver-based) is applied to implement reasoning operations. We also noticed that usability is rather weak in many cases (e.g., counter-intuitive representations, lack of guidance).

Future work. We provided empirical evidence that SCs are complex information systems for which qualities like usability and correctness are not always satisfied. The contribution of this paper is a first step toward their understanding, and a foundation for devising effective (re-)engineering solutions. Our ongoing work is to extend the search engine with advanced data extraction procedures so as to obtain all other necessary information (e.g., option hierarchy, descriptive information, CSS data). Moreover, at the moment, constraints are detected either by manual inspection or by simulating simple scenarios, which only covers a subset of the possible constraints. To increase completeness, we are now integrating web crawling and "hidden web" techniques [31] in our simulator and search engine. Furthermore, we believe that the use of variability models to formally capture configuration options and constraints, and solvers used in more academic configuration tools (e.g., SAT and SMT) to reason about these models, would provide more effective and reliable bases. Yet a lot more effort is needed for providing a systematic and comprehensive solution to practitioners.

References

1. T. Rogoll and F. Piller, "Product configuration from the customer's perspective: A comparison of configuration systems in the apparel industry," in *PETO'04*, 2004.
2. C. Streichsbier, P. Blazek, F. Faltin, and W. Frühwirt, "Are *de facto* Standards a Useful Guide for Designing Human-Computer Interaction Processes? The Case of User Interface Design for Web-based B2C Product Configurators," in *HICSS'09*. IEEE, 2009, pp. 1–7.
3. A. Trentin, E. Perin, and C. Forza, "Sales configurator capabilities to prevent product variety from backfiring," in *Workshop on Configuration (ConfWS)*, 2012.
4. K. Pohl, G. Böckle, and F. J. van der Linden, *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer-Verlag New York, Inc., 2005.
5. M. Schäler, T. Leich, M. Rosenmüller, and G. Saake, "Building information system variants with tailored database schemas using features," in *CAiSE'12*. Springer-Verlag, 2012, pp. 597–612.
6. F. Gottschalk, T. A. C. Wagemakers, M. H. Jansen-Vullers, W. M. P. van der Aalst, and M. L. Rosa, "Configurable process models: Experiences from a municipality case study," in *CAiSE*, 2009, pp. 486–500.
7. M. L. Rosa, W. M. van der Aalst, M. Dumas, and A. H. ter Hofstede, "Questionnaire-based variability modeling for system configuration," *Software and Systems Modeling*, vol. 8, no. 2, pp. 251–274, 2008.
8. http://www.configurator-database.com, 2011.
9. Q. Boucher, E. K. Abbasi, A. Hubaux, G. Perrouin, M. Acher, and P. Heymans, "Towards more reliable configurators: A re-engineering perspective," in *PLEASE'12, co-located with ICSE'12*, 2012.
10. E. Ferrara, P. D. Meo, G. Fiumara, and R. Baumgartner, "Web data extraction, applications and techniques: A survey," *CoRR*, vol. abs/1207.0246, 2012.
11. K. Czarnecki and C. H. P. Kim, "Cardinality-based feature modeling and constraints: A progress report," in *OOPSLA'05*, 2005.
12. R. Michel, A. Classen, A. Hubaux, and Q. Boucher, "A formal semantics for feature cardinalities in feature diagrams," in *VaMoS'11*. ACM, 2011, pp. 82–89.
13. T. Berger, S. She, R. Lotufo, A. Wasowski, and K. Czarnecki, "Variability modeling in the real: a perspective from the operating systems domain," in *ASE'10*. ACM, 2010, pp. 73–82.
14. L. Hvam, N. H. Mortensen, and J. Riis, *Product Customization*. Springer-Verlag Berlin Heidelberg, 2008.
15. A. Hubaux, Y. Xiong, and K. Czarnecki, "A survey of configuration challenges in linux and ecos," in *VaMoS'12*. ACM Press, 2012, pp. 149–155.
16. M. Janota, G. Botterweck, R. Grigore, and J. Marques-Silva, "How to complete an interactive configuration process?" *CoRR*, vol. abs/0910.3913, 2009.
17. K. Schmid, R. Rabiser, and P. Grünbacher, "A comparison of decision modeling approaches in product lines," in *VaMoS'11*. ACM, 2011, pp. 119–126.
18. A. Hubaux, A. Classen, M. Mendonça, and P. Heymans, "A preliminary review on the application of feature diagrams in practice," in *VaMoS'10*, 2010, pp. 53–59.
19. N. Weston, R. Chitchyan, and A. Rashid, "A framework for constructing semantically composable feature models from natural language requirements," in *SPLC'09*. ACM, 2009, pp. 211–220.

20. I. John, "Capturing product line information from legacy user documentation," in *Software Product Lines.* Springer, 2006, pp. 127–159.
21. M. Acher, A. Cleve, P. Collet, P. Merle, L. Duchien, and P. Lahire, "Reverse Engineering Architectural Feature Models," in *ECSA'11.* Springer, 2011, pp. 220–235.
22. S. She, R. Lotufo, T. Berger, A. Wasowski, and K. Czarnecki, "Reverse engineering feature models," in *ICSE'11.* ACM, 2011, pp. 461–470.
23. M. Acher, A. Cleve, G. Perrouin, P. Heymans, C. Vanbeneden, P. Collet, and P. Lahire, "On extracting feature models from product descriptions," in *VaMoS'12.* ACM, 2012, pp. 45–54.
24. M. Acher, B. Baudry, P. Heymans, A. Cleve, and J.-L. Hainaut, "Support for reverse engineering and maintaining feature models," in *VaMoS'13.* ACM, 2013, pp. 1–8.
25. A. Lora-Michiels, C. Salinesi, and R. Mazo, "A method based on association rules to construct product line models," in *VaMoS'10*, 2010, pp. 147–150.
26. V. Alves, C. Schwanninger, L. Barbosa, A. Rashid, P. Sawyer, P. Rayson, C. Pohl, and A. Rummler, "An exploratory study of information retrieval techniques in domain analysis," in *SPLC'08.* IEEE, 2008, pp. 67–76.
27. A. M. Memon, I. Banerjee, and A. Nagarajan, "GUI ripping: Reverse engineering of graphical user interfaces for testing," in *RE'03.* IEEE, 2003, pp. 260–269.
28. S. Staiger, "Static analysis of programs with graphical user interface," in *CSMR'07.* IEEE, 2007, pp. 252–264.
29. J. Vanderdonckt, L. Bouillon, and N. Souchon, "Flexible reverse engineering of web pages with vaquista," in *WCRE'01.* IEEE, 2001, pp. 241–248.
30. G. A. Di Lucca, A. R. Fasolino, and P. Tramontana, "Reverse engineering web applications: the WARE approach," *J. Softw. Maint. Evol.*, vol. 16, no. 1-2, pp. 71–101, Jan. 2004.
31. A. Mesbah, A. van Deursen, and S. Lenselink, "Crawling ajax-based web applications through dynamic analysis of user interface state changes," *ACM Trans. Web*, vol. 6, no. 1, pp. 3:1–3:30, Mar. 2012.