PReCISE – FUNDP
University of Namur
Rue Grandgagnage, 21
B-5000 Namur
Belgium

TECHNICAL REPORT January 8, 2013

AUTHORS	Ebrahim Khalil Abbasi, Arnaud Hubaux, Mathieu Acher, Quentin Boucher, Patrick Heymans
APPROVED BY	P. Heymans
EMAILS	{eab|ahu|mac|qbo|phe}@info.fundp.ac.be
REFERENCE	P-CS-TR CONF-000001
PROJECT	FSR, SAT, NAPLES
FUNDING	FSR, Walloon Region
SUBMITTED TO	CAiSE 2013

What's in a Web Configurator? Empirical Results from 111 Cases

What's in a Web Configurator? Empirical Results from 111 Cases

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

PReCISE Research Centre, Faculty of Computer Science, University of Namur, Belgium
{eab, ahu, mac, qbo, phe}@info.fundp.ac.be

Abstract. Nowadays, mass customization has been embraced by a large portion of industries. As a result, the web abounds with configurators that help users tailor all kinds of goods and services to their specific needs. These configurators are a privileged interface between customers and companies, and often the single entry point for customer orders. Their reliability is thus crucial. However, the state of the practice lacks guidelines for building reliable and efficient web configurators. To provide these guidelines, empirical data on the current practice is required. The first part of this paper reports on a systematic study of 111 web configurators along three dimensions: configuration options rendering, constraint handling, and configuration process support. It also describes the code inspection tools we developed to reverse engineer this information from web configuration clients. The second part highlights good and bad practices in web configurator engineering. This paper opens avenues for the elaboration of web configurator engineering guidelines and the construction of semi-automated re-engineering tools.

1 Introduction

In many markets, being competitive often echoes with the ability to propose customised products at the same cost and delivery rate as standard ones. These customised products are often characterised by hundreds of *configuration options*. For many customers, this repertoire of inter-related options can be disconcerting. *Configurators* were therefore developed (see Figure 1 for an example). Configurators are interactive graphical user interfaces (GUIs) that assist customers during decision making [1, 2]. They also keep the configuration environment consistent by verifying constraints between options, propagating user decisions, and handling conflictual decisions.

A significant share of existing configurators is web-based, irrespective of the market. The configurator database maintained by Cyledge is a striking evidence [3]. Since 2007, they collected more than 800 web configurators coming from 29 different industries, including automotive, apparel, sports equipment, and art. These configurators vary significantly. They each have their own characteristics, spanning visual aspects (GUI elements) to constraint management. For example, the web configurator of the Opel Astra Hatchback depicted in Figure 1, displays different configuration options through specific widgets (radio buttons, combo box items and check boxes). These options can be in different states such as activated (e.g., "Tyre Pressure Monitoring System" is highlighted in grey), deactivated (e.g., "Active front seat head restraints" is flagged with a

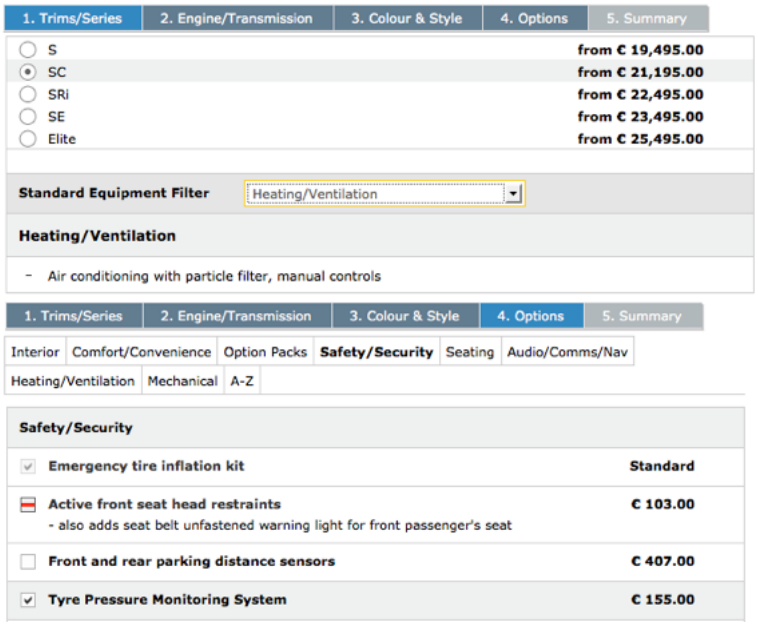

Fig. 1. Opel Astra Hatchback web configurator (screenshot captured on the 8th of April 2012 from http://www.opel.ie/tools/model-selector/cars.html)

red dash), or unavailable (e.g., "Emergency tire inflation kit" is greyed out). Additionally, these options are organised in different tabs (e.g., "Options") and sub-tabs (e.g., "Safety/Security") which denote a series of steps in the *configuration process* (e.g., "1. Trims/Series" is followed by "2. Engine/Transmission"). A configurator can also implement *cross-cutting constraints* between options. These are usually hidden to the user but they determine valid combinations of options. For instance, the selection of "Active front seat head restraints" implies the selection of "Seat belt unfastened warning light for front passenger's seat", meaning that the user cannot select the former without the latter. Moreover, descriptive information is sometimes associated to an option (e.g., its price).

Web configurators occupy a key position between customers and companies. They are a direct, and often unique, entry point for many customer orders. Their reliability is thus critical. Yet, a consistent body of knowledge dedicated to web configurator engineering is still missing. This absence of standard guidelines often translates in correctness or runtime efficiency issues, mismatches between the constraints exposed to the user and those actually implemented, and an unclear separation between the GUI and business logic. Some of our industry partners face similar problems and are now trying to migrate from legacy configurators towards more reliable, efficient, and maintainable solutions.

In order to improve reliability and maintainability, we first need to *understand* how web configurators are implemented. This means investigating characteristics ranging from the GUI itself over constraint expressiveness to the reasoning engine (Section 2). For instance, different graphical representations of configuration options (e.g., check boxes or radio buttons), constraint implementation (e.g., widget-based or JavaScript function), and configuration processes (e.g., multi- or single-step) prevail. Several authors have already proposed reverse engineering techniques to extract variability information like options and constraints from different kinds of artefacts [4–13]. However, none of them is dedicated to web configurators.

To gain a deep understanding of web configurator, we conduct an empirical study of 111 web configurators distributed in 21 different industries (Section 3). Specifically, we investigate the client-side code of all configurators with semi-automated code inspection tools we developed. We then classify and analyse the results along three dimensions: *configuration options*, *constraints*, and *configuration process* (Section 4). For each dimension, we present our quantitative empirical results and the reverse engineering issues we faced. We conclude the study with a discussion on the good and bad practices (Section 5) we observed.

The intended impacts of this research are twofold. First, the configuration patterns and the issues we report lay the foundations for semi-automated reverse engineering solutions. Second, the results and insights gained are the starting points for a collection of guidelines on web configurator (re-)engineering.

2 Problem Statement

In [14], we report on the needs from some partners to migrate their legacy configurators towards more reliable and maintainable solutions. We also outline the foundations of our configurator re-engineering approach. It breaks down in three steps:

- variability information *reverse engineering*. Variability information includes options, attributes, hierarchy, constraints, etc., the configuration process, and GUI information (widget types, styling directives, etc.);
- variability information *encoding* into dedicated languages (e.g., TVL [15]);
- configurator *forward engineering*. Once encoded, we use the variability information to generate (i) a website skeleton, and (ii) a reasoning engine that embeds a formal solver (e.g., SAT or SMT) to maintain the consistency of the configuration.

To implement re-engineering tools, we first need a realistic understanding of web configurators, which can only be obtained from empirical investigations. Our empirical study seeks to answer three questions:

RQ1 *How are configuration options visually represented and what are their semantics?* By nature, web configurators 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, attributes, cloning, and grouping).

RQ2 *What kinds of constraints are supported by the configurators and how are they enforced?* Besides semantic constructs, the selection of options is also governed by constraints. These constraints are often deemed complex and non-trivial to implement. This part of the survey sheds lights on the actual complexity of their implementation.

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

3 Reverse Engineering Method

This section explains how we selected the 111 web configurators and how we collected data.

3.1 Configurator selection

To collect a representative sample of web configurators, we used Cyledge's configurator database which contains 800+ entries from a wide horizon of products. 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 industries.

Secondly, we excluded 26 configurators that are no longer available using *Jericho*[1]. We considered a site unavailable either when it can no longer be accessed or it requires authentication.

Thirdly, we randomly selected 25% of the configurators in each industry. We then checked each selected configurator with *Firebug*[2] to ensure that configuration options, constraints, and constraint handling procedures do not use Flash. Firebug is a Firefox plugin used to monitor, modify, and debug CSS, HTML, and JavaScript. We excluded configurators using Flash, since the Firebug extension we implemented (see next section) does not support that technology. We also excluded false-configurators. By false-configurators, we mean 3D designer 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 was a sample set of 93 configurators from 21 industries. Finally, we added 18 configurators to the list. These configurators were used in preliminary stages of this systematic survey to become familiar with web configurators and to train our reverse engineering tools, as discussed below. This raised the total number of configurators to 111.

[1] http://jericho.htmlparser.net/docs/index.html
[2] http://getfirebug.com/

3.2 Data extraction process

The data extraction process is a mix of manual code inspection and automated extractions.

Manual code inspection is a preliminary step to automated analyses. Its goal is to elicit which patterns (e.g., *tag[attribute:value]*) implement graphical widgets. This task was performed by the first author of the paper. To perform automated extraction, we implemented a Firebug extension, and extended it to support more advanced search features. We also used JavaScript and jQuery[3] to add functionality and document traversing. The extension took one person-month effort to implement about 3KLOC.

In essence, our extension offers a *search engine* able to (1) *search* special elements parametrized by the user, and (2) *simulate* user actions. To bootstrap the option extraction process, we first had to manually identify the patterns used to represent options. We then fed those patterns to our extension to extract all options. Our extension uses jQuery selectors as well as a code clone detection technique to search matching elements and to extract an option name, its widget type, and their occurrence in a configurator.

Our simulator reproduces click events as executed by a user. The simulator was used to discover some constraints between options. In a nutshell, the simulator selects/deselects each option and reports the existence of possible constraints based on the previous state of the page.

Armed with these tools, we were able to extract all the necessary information to answer our three research questions.

4 Results

This section summarises the results of our empirical study. The complete set of data is available at `http://info.fundp.ac.be/~eab/result.html`. Each subsection answers the questions posed in Section 2 and describes the reverse engineering issues we faced.

4.1 Configuration options

In this subsection, we answer RQ1 and explain which types of widgets are used to present options.

Observations For clarity, we classify and successively expose the observations by concept.

Option Representation The diversity of representations for an option is one of the most striking result, as shown in Figure 2. In decreasing order, the most popular widgets are: *combo box item*, *image*, *radio button*, *check box* and *text box*. Image means here both picture and color. We also observed that some widgets were combined with images, namely, *check box*, *radio button*, and *combo box item*. Selecting either the image or the widget selected the corresponding option. The *Other* category contains miscellaneous

CONFIGURATION OPTIONS		
Semantic Constructs	XOR-group	97%
	OR-group	8%
	Interval	4%
Mandatory Options	Notification, Default	83%
	Transition Checking	13%
	Not checked	4%
Attribute	Descriptive information	86%
	User input	68%
	Additional requirements	12%
Multiple instantiation	Cloning	5%
CONSTRAINTS		
Constraint Type	Formatting	59%
	Group	99%
	Cross-cutting	55%
Formatting Constraint (66)	Prevention	62%
	Verification	41%
	Not checking	26%
Cross-cutting Constraint (61)	Visibility	89%
Constraint Description (61)	Annotation	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%
Visual Product	Yes	50%
	Not supported	50%
Configuration Summary	Search result	2%
	Final step	13%
	Shopping cart	82%
	Not supported	3%

Table 1. Result summary

and less frequent widgets like *slider*, *label*, *file chooser*, *date picker*, *color picker*, *image needle*, and *grid*.

Grouping Grouping is a way to organise *related* options together. Depending on the context, *related* can have different meanings. For instance, a group can contain a set

[3] http://jquery.com/

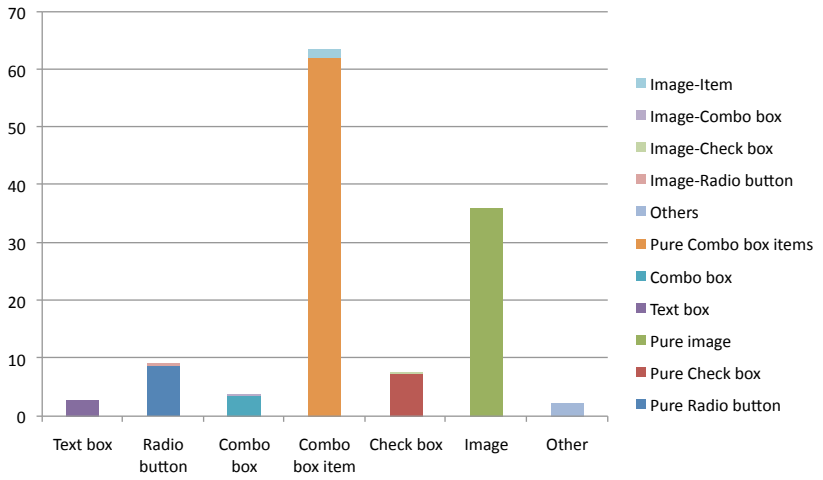

Fig. 2. Widget types in all the configurators

of colors, the options for an engine, or the stickers on a pair of shoes. Groups can be *basic* or *hierarchical*. Three different semantic constraints can apply to a group:

- *XOR-group*: one and only one option must be selected (e.g., the Trims/Series in Figure 1).
- *OR-group*: at least one option must be selected (e.g., stone and band to put on a ring).
- *interval* (a.k.a. cardinality [16]): the specific lower and upper bounds on the number of selectable elements is determined (e.g., flavours in a milkshake).

The *Semantic Constructs* row in Table 1 shows that *XOR*-groups are the most frequent with 97% of configurators implementing them. In contrast, 8% implement *OR*-groups, and only 4% intervals.

Figure 3 presents the frequency of widgets used to represent grouping in web configurators. Combo box items, images and radio buttons are the top 3 for *XOR*-groups. The representation of *XOR*-groups can also be implicit, i.e., not implemented by the widget but in underlying constraints. For OR-groups, check boxes are the most common widgets.

Mandatory 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*: When the configuration environment is loaded, (some or all) mandatory options are selected or assigned a default value.

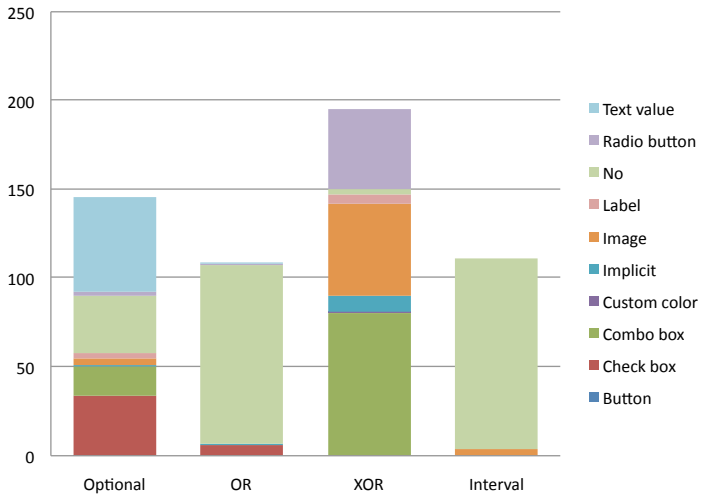

Fig. 3. Widget types in semantic constructs

- *Notification*: 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.
- *Transition Checking*: 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.

Table 1 shows that 83% of the configurators use default configuration or notification, while 13% of them follow the transition checking pattern. We also noticed that 4% of the configurators lack strategies for handling mandatory options.

Mandatory options can be distinguished from optional ones with *highlighters*. Highlighters can be: graphical notations (e.g., *), textual keywords (e.g., *required*, *not required*, or *optional*), or other highlighting mechanisms (e.g., boldfaced, coloured text, etc.). These highlighters are visible either as soon as the configuration environment is loaded, or when the user finalises the configuration (notification pattern) or moves through the next step (transition checking pattern). We observed that only 14% of the configurators highlight mandatory or optional options.

70% of the configurators have optional options in their configuration environments. The *optional* bar in Figure 3 shows in what proportion widgets like *text box*, *check box* and *combo box* are used to represent optional options.

Attributes The attribute of an option is any of its characteristics which can be measured [17]. In this paper we considered two kinds of attribute: *descriptive information* attached to an option; *typed values* inserted by the user (text box, file chooser, slider, etc.) that impact the configuration.

Descriptive information is represented in different ways such as: descriptions attached to the option, hints, tabular information, information in natural language opened with a *More Information* button or link, etc. *String*, *real*, *integer*, and formatted values (for color code and date) are data types we observed for both descriptive and user input values. The *Attribute* row in Table 1 shows that 86% of the configurators provide descriptive information for the options, and 68% of them allow users to input values. A special user input value is *additional requirement* that is a description of options that the user wants to be included in her custom product but are not presented in the configuration environment.

Cloning Cloning means that the user determines how many instances of an option are included in the final product [18] (e.g., a text element to be printed on a t-shirt can be instantiated multiple times and configured differently). The user might be able to configure each clone differently. We observed cloning mechanisms in only 5% of the configurators.

Reverse engineering issues Besides widgets representing options, other widgets containing product shipment information, agreement check box, etc. are also presented in the configuration environment. Since we concentrate on option-related widgets, we implemented various techniques to discard irrelevant widgets. The engineer can either delimit a search region in the GUI, or force the search engine to ignore some widgets. Both techniques were implemented by adding a new attribute to elements in Firebug and parametrizing the search engine with this new attribute.

Another issue is the presence of widgets visible in the source code but not in the GUI. In some cases, the invisible widget must be ignored by the search engine, e.g., if it does not represent a configuration option for the product being configured. In other cases, those widgets should be considered by the search engine since they represent currently invisible options due to some visibility constraints (see below). The engineer can parametrize the search engine to ignore invisible widgets.

Some radio buttons and check boxes were implemented with images representing their status (selected, rejected, undecided, etc.) rather than standard widgets. This forced us to use image-based search parameters to extract the option types in these cases.

Some combo boxes included irrelevant values like separators, *none* or *select an item* values. We used the *none*-like keywords to determine the optionality of a combo box. Our search engine can be parametrized to ignore or detect particular items in combo boxes during reverse engineering.

The search engine is able to extract option names and widget types. However, other data also must be extracted: Attributes, constraints, option hierarchy, CSS data, etc. We therefore need to extend the search engine with clone detection, mining, and data structuring methods. Finally, the mapping from the implementation to semantic construct still requires a human intervention.

4.2 Constraints

This subsection answers RQ2 and explains how the reasoning engine instantiates constraints to keep the configuration environment consistent.

Observations We split constraints in three categories depending on their target and implementation.

Formatting Constraint A formatting constraint ensures that the value set by the user is *valid*. Depending on the option, valid can have different meanings:

- *Type correctness:* Some options are strongly typed (e.g., String, Integer, Real), which means that types must be verified to produce valid configurations;
- *Range control:* Besides a type, the value range of an option might be further constrained by, for instance, upper and lower bounds, slider domain, valid characters, and maximum file size;
- *Formatted values:* Some more values require specific formatting constraints such as date, color code, email, image, and file extension;
- *Case-sensitive values:* Some configurators propose a selectable list of items. Instead, some explain in natural language the possible options and the user has to type in a text field the selected value. Similarly, to capture the deselection of an option, some configurators explicitly request the user to enter values like *None* or *No*. In such circumstances, case needs to be respected for the choice to be valid.

We observed that configurators provide two different patterns for checking constraint violation:

- *Prevention:* The reasoning engine prevents illegal values. For example, it stops accepting input characters if the maximum number is reached, defines a slider domain, uses a date picker, etc.
- *Verification:* The reasoning engine validates the values entered by the user a posteriori, and, for example, highlights, removes or corrects illegal values or prevents the transition to the next step.

From the 66 configurators with formatting constraints, 62% implement the prevention pattern and 41% the verification pattern. Those patterns are not mutually exclusive, and configurators can use them for different subsets of options. This is why the sum of the percentages exceeds 100%. We also noticed that 26% of the configurators do not check constraints even if they are described in the interface.

Group Constraint A group constraint defines the number of options which can be selected in a group of options. In essence, constraints implied by *OR*-, *XOR*- 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 *XOR*-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 independently of their inclusion in a group. *Require* and *Exclude* are the most common cross-cutting constraints. More complex constraints might also be specified in propositional logic or be directly encoded in plain JavaScript (client side) or PHP (server side), for instance. We observed cross-cutting constraints in 55% of the configurators.

In a GUI, some constraints lead to showing/hiding the related options. This is called *visibility constraint* [13]. Note that a button opening another container is also considered as a visibility constraint between the option associated with the button and options

included in the container. Automatically adding options into a combo box upon modification of another option also falls in the same family of constraints. From the 61 configurators with cross-cutting constraints, 89% support visibility constraints.

In web configurators, cross-cutting constraints are implemented on the client and/or on the server side. We noticed that only 11% of the configurators supporting cross-cutting constraints document them in the GUI (natural language).

We now focus on the capabilities of the reasoning engines, 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 engine automatically propagates the required changes to all the impacted options. We call it *automatic* propagation. In other cases, the reasoning engine asks to confirm or discard a decision before altering other options. We call this *controlled* propagation. Finally, we also observed some cases of *guided* propagation. For example, if option A requires to select option B or C, the reasoning engine cannot decide whether B or C should be selected knowing A. In this case, the configurator proposes a choice to the user. We observed the automatic pattern in 97%, the controlled pattern in 8%, and the guided pattern in 3% of the configurators. Here again, some configurators concurrently implemented more than one pattern.

Consistency Checking An important issue in option and cross-cutting constraints handling is *when* the reasoning engine instantiates the constraints and checks the consistency. We noticed two patterns:

- *Interactive:* The reasoning engine 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 as interactive.
- *Batch:* The reasoning engine checks the consistency of the configuration upon request, for instance, when the user moves to the next configuration step.

From the 83 configurators with formatting or cross-cutting constraints, 76% propose interactive consistency checking, while 59% operate in batch. The overlapping is caused by some configurators implementing both mechanisms, depending on the constraint type.

Undo It is an operation that allows users to roll back on their last choice(s). Among all configurators in the survey, only 11% of them support undo.

Reverse engineering issues We had to carry out static as well as dynamic analyses to extract constraints. For static analysis, we manually inspected the configurator. Formatting, group, and visibility constraints were extracted this way.

To extract other cross-cutting constraints, we conducted a dynamic analysis with our simulator. When an event is triggered, we monitor changes in the states of the options to track the presence of a constraint. We also had to take the cascading effect of variable changes into account in order to identify constraints individually rather than as an atomic block of changes. This task was further complicated by the differences between the technologies and implementations.

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 gain in completeness, we first need to integrate web crawling and hidden Web techniques [19, 20] in our simulator and search engine. Secondly, when a constraint is defined over three or more options, all combinations of options must be analysed to detect the constraint. This is a more challenging issue in large scale configuration environments due to the combinatorial nature of the problem. Thirdly, after detecting a constraint, it must be extracted and formally captured. There exist different presentations of constraints: Natural language statement attached to the option; Required/excluded attributes; Visibility constrains (on-the-fly content is created and presented to the user); etc. Heuristics are required to extract valuable data from these different contents as well as from the DOM tree state changes.

4.3 Configuration process

In this subsection, we answer RQ3 and explain how processes are managed in web configurators.

Observations To understand the process, we study its pattern, activation mechanism, and navigation.

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:* All the options are displayed to the user in a single graphical container.
- *Basic Multi-step:* 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:* It is the same as a multi-step except that a step can contain inner steps.

Activation In multi-step processes, we noticed two *step activation* strategies:

- *Step-by-step activation:* Only the first step is available and other steps become available as soon as all options in the previous step have been configured.
- *Full-step activation:* 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. We noticed two different strategies:

- *Stateful arbitrary:* The user can go back to any previous step and all configuration choices are saved.
- *Stateless arbitrary:* The user can go back to any previous step but all configuration choices made in steps following the one reached are discarded.

Among the 111 configurators, 48% are single-step, 45% are basic multi-step while the remaining 7% are hierarchical. Among the 58 multi-step configurators, 59% implement step-by-step activation (see Table 1).

Regarding backward navigation, 69% of the multi-step configurators are stateful, 14% stateless, and the other 17% do not support backward navigation. More precisely, 68% of basic multi-step configurators follow stateful arbitrary navigation while 14% are stateless. For hierarchical multi-step configurators, the values are 75% and 13%, respectively. Backward navigation is not supported by 18% of basic multi-step processes and 13% of hierarchical ones. We also observed that all full-step activation configurators follow the stateful arbitrary navigation pattern.

We gathered two additional facts about configuration processes: *visual product* and *configuration summary* (see Table 1). With the *visual product* criteria we assess whether the configurator offers a rendering mechanism to display the product being configured. 50% of the configurators support this feature. By *configuration summary*, we mean a summary of the selected options at the end of the configuration process. We observed that 13% of the configurators display this information in the final step. 82% of them show this summary in the shopping cart. 3% of them do not support this feature. If a configurator provides both final step and shopping cart summaries, we counted it as final step. Some configurators simply allow users to select an existing product in the database. In these configurators, options are search criteria and the configuration summary corresponds to the (set of) product(s) matching the search criteria. 2% of the configurators fall in this category.

Reverse engineering issues We sometimes had trouble distinguishing a group from a step since both are containers for options. We defined four criteria to differentiate them: (1) a step is a coarse-grained container compared to a group, meaning that a step might include several groups; (2) steps might be numbered; (3) the term 'step' or its synonyms might be used in steps' labels; and (4) a step might capture constraints between options. If, based on these criteria, we could not determine whether it was a step or a group, we considered it a group.

5 Discussion

The previous section focused on implementation-level characteristics of web configurators. 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, 2, 21–25].

5.1 Bad practices

Back-end

– *Absence of propagation notification*: In many cases, options are automatically enabled/disabled or appeared/disappeared without notice. This makes configuration

confusing especially for large multi-step models as the impact of a decision is downright impossible to predict and visualise. 97% of the configurators automatically propagate decisions but rarely inform users of the impact of their decisions.
– *Ad hoc consistency checking*: Be it in interactive or batch, consistency checking procedures are not always complete. 4% of the configurators do not check mandatory options are indeed selected, while 26% of the configurators supporting formatting constraints do not verify them.

Front-end

– *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 these widgets. For instance, some exclusive options are implemented by check boxes. Consequently, users only discover the grouping constraint by experimenting with the configurator, which is a cause of confusion and misunderstanding. It also suggests possible inconsistencies between the intended and implemented behaviours.
– *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 Constraint*: When a visibility constraint is applied, options might be hidden and/or deactivated. This has the advantage of reducing the solution space [23] and avoiding conflictual decisions. However, the downside is that to access hidden/deactivated options, the user has to first undo decisions which instantiated the visibility constraint. These are known problems in configuration [24] that should be avoided to ensure a satisfying user experience. 89% of the configurators 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 to alter their configuration.

5.2 Good practices

Back-end

– *Guided Consistency Checking*: Only 3% of the configurators 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 [23], which are known to improve their usability [1].
– *Auto-completion*: This operation allows users to configure some desired options and then let the configurator complete undecided options [21]. The auto-completion function is typically useful when only a few options are of interest for the user. Common auto-completion mechanisms include default values or the optimisation

of an objective function (e.g., price minimisation). Web configurators usually support auto-completion by providing default configuration for mandatory options.
– *Solver-based consistency checking*: Solver-based approaches embed a theorem prover (e.g., SAT or SMT) in the reasoning engine to handle the consistency of the configuration, and also to implement advanced functionalities such as undo, redo, auto-completion, or conflictual decisions management, all in an efficient and reliable way [25, 26]. Our investigation of the client-side code of some web configurators did not reveal signs of solvers being used for reasoning. Instead, we observed ad hoc implementations. For example, when selecting/deselection an option, JavaScript functions are called for performing the consistency checks in the client side (i.e., without calling any server where a reasoning engine can be executed). This kind of implementations is likely to raise correctness or even performance issues.

Front-end

– *Self-explanatory process*: A configurator should provide clear support during the configuration processes [1, 2, 23]. The multi-step configurators we observed offer a diversity of standard graphical mechanisms such as numbered steps, "previous" and "next" buttons, the permanent display of already selected options, a list of complete/incomplete steps, etc.
– *Self-explanatory widgets*: Although obvious, 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 as much as possible. On top of self-explanatory widgets, configurators should be able to explain constraints "on the fly" to the users. This is only available in 11% of the configurators.
– *Stateful backward navigation*: This functionality is a must-have to allow users to revise their decisions. 69% of the web configurators do support it.
– *Configuration summary*: 97% of the configurators allow users to review somehow their choices before ordering or purchasing a product. The configuration summary can be enriched with additional information such as the status of the option (manually vs automatically selected), the constraint that conditions its inclusion or related attributes (e.g., its price or delivery date).

6 Threats To Validity

The main *external* threat is our web configurator selection process. Although we tried to collect a representative total of 111 configurators from 21 industry sectors, we dependent of the representativeness of the sample source, i.e.. Cyledge's database of configurators. It is also hard to tell whether our observations hold for standalone applications. Our experience with standalone configurators shows similar patterns but further experiments are necessary to confirm this conjecture.

An *internal* threat to validity is that our approach is semi-automated. When analysing a configurator, we made arbitrary decisions that influenced the result set. For example,

some configurators allow one to customise several products. In these cases, we randomly selected and analysed one of them. If another product had been chosen, the number of options and constraints would have probably been different. Another example is related to the semi-automatic extraction of options and constraints. We manually had to select some options to load invisible options in the source code. However, for large configurators, it is not possible to test all possible combinations of options without automated tool support. Therefore, we have probably missed some. To elicit cross-cutting constraints, we rely on the simulator that emulates the click event of each widget or selects each item of a combo box. To identify constraints, the simulator has to monitor the states of each variable and track changes. For instance, a constraint is instantiated if the selection of A implies the selection of B, but not the other way around. The heterogeneity of constraint implementations prevented the integration of their execution by the simulator. As for visibility constraints, some cross-cutting constraints might have been missed.

The manual part of the study was conducted by the first author. His choices, interpretation and possible errors might have 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 similar study conducted by another pool of researchers could further increase the robustness of the conclusions.

7 Related Work

Variability Academic research has lead to the definition of variability modelling languages such as feature models [27] and decision models [28]. Yet, a thorough evaluation of the adequacy and impact of such languages in practice is still missing [29], 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 syntactical constructs, semantics, usage and GUI-based configuration tools of the two languages. They focus on the two languages and on the operating system domain to pilot a configuration whereas we study variability concepts as visually expressed and implemented in a wide range of web configurators. Their investigations were restricted to two configurators.

Several authors have already addressed the (semi-automatic) reverse engineering of variability models from existing artefacts. Some rely on user documentation [4], others on natural [5] or formal requirements [6], product descriptions [7], dependencies [8], source code [9–11] or architecture [12], etc. To the best of our knowledge, none of existing reverse engineering approaches tackles the extraction of variability patterns from web configurators.

GUIs and Web Other authors proposed to reverse engineer GUIs and web pages but did not consider configuration aspects. Memon *et al.* proposed a technique called GUI ripping to extract models of the GUI's structure and behaviour for testing purposes [30]. In [31], Staiger presents an approach that detects GUI widgets and their hierarchy.

Others proposed to reverse engineer web pages. With VAQUISTA [32], Vanderdonckt *et al.* reverse engineered the presentation model of a web page. The tool-supported WARE approach seeks to understand, maintain and evolve undocumented web applications by reverse engineering them to UML diagrams [33]. Amalfitano *et al.* also reverse engineered Rich Internet Applications' behaviour by dynamic analysis [34]. In [35], Patel *et al.* reviewed existing approaches to reverse engineer web pages.

Study of Configurators Rogoll *et al.* [1] performed a qualitative study of 10+ web configurators. The authors reported on *usability* of such configurators and how visual techniques assist customers in configuring products. Our study is larger (100+ configurators) while our goal and methodology differ significantly. We aim to understand how the underlying concepts of web configurators are visually represented, managed and implemented, without studying specifically the usability of web configurators. Yet, the quantitative and qualitative insights of our study can be used for this purpose.

Streichsbier *et al.* [2] analysed 126 web configurators among those collected in [3]. 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 configurators. Interestingly, our findings can help identify or validate existing standards in the domain of web configurators. For example, our study reveals that in more than half of the configurators "the selected product components are summarised at the end of the configuration process" or that "products available for configuration are presented as images"—in line with [2].

8 Conclusion

This paper presents an empirical study of 111 web configurators along three dimensions: configuration options, constraints and configuration process.

Among a diversity of widgets used to represent *configuration options*, combo box items and images are the most common one compared to other types such as radio buttons, check boxes, and text boxes. Similarly, XOR-groups are the most frequent ones and are generally represented using the same widgets. We observed that configuration options, though not visually grouped together, might be logically dependent on one another: for more than half of the configurators, the so-called cross-cutting *constraints* govern their selection and deselection. Furthermore, their implementation and verification vary significantly. As for the *configuration process*, half of the configurators propose multi-step configuration, two thirds of which enable stateful backward navigation.

The empirical analysis of web configurators reveals reliability and runtime efficiency issues when handling and reasoning about 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 observation shows that they are not completely and satisfactorily implemented. Moreover, the investigation of client-side code implementation verifies, in part, that no systematic method is applied to implement reasoning operations. We believe that the use of satisfiability solvers would provide more effective and reliable basis for implementing reasoning operations encountered in exist-

ing configurators like consistency checking, undo, redo, auto-completion, or conflictual decisions' resolution.

The front-end of a configurator provides an interactive configuration environment and its usability is of prior importance. We noticed that usability is rather weak in many cases, due to *bad* practices, e.g., counter-intuitive representation of options, lack of user guidance and freedom during configuration process. A self-explanatory configurator with *good* practices such as an easy to navigate, supportive, and suggestive configuration process that uses standard widgets to present options would improve its usability.

Maintenance, correctness, usability, and runtime efficiency issues reported in this paper underline the importance of re-engineering web configurators to produce applications founded on reliable solutions. The reverse engineering issues we reported all along the paper when extracting relevant information in configurators are opportunities for further work and research. The bad and good practices guide opens windows toward the creation of a body of knowledge for web configurators. We believe our findings and guidelines on back-end and front-end will be of interest to researchers and practitioners (re-)engineering web configurators.

References

1. Rogoll, T., Piller, F.: Product configuration from the customer's perspective: A comparison of configuration systems in the apparel industry. In: PETO'04. (2004)
2. Streichsbier, C., Blazek, P., Faltin, F., Frühwirt, W.: 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) 1–7
3. `http://www.configurator-database.com`: (2011)
4. John, I.: Capturing product line information from legacy user documentation. In: Software Product Lines. Springer (2006) 127–159
5. Weston, N., Chitchyan, R., Rashid, A.: A framework for constructing semantically composable feature models from natural language requirements. In: SPLC'09, ACM (2009) 211–220
6. Alves, V., Schwanninger, C., Barbosa, L., Rashid, A., Sawyer, P., Rayson, P., Pohl, C., Rummler, A.: An exploratory study of information retrieval techniques in domain analysis. In: SPLC'08, IEEE (2008) 67–76
7. Acher, M., Cleve, A., Perrouin, G., Heymans, P., Vanbeneden, C., Collet, P., Lahire, P.: On extracting feature models from product descriptions. In: VaMoS'12, ACM (2012) 45–54
8. She, S., Lotufo, R., Berger, T., Wasowski, A., Czarnecki, K.: Reverse engineering feature models. In: ICSE'11, ACM (2011) 461–470
9. Ziadi, T., Frias, L., da Silva, M.A.a.A., Ziane, M.: Feature identification from the source code of product variants. In: CSMR'12, IEEE (2012) 417 –422
10. Valente, M.T., Borges, V., Passos, L.: A semi-automatic approach for extracting software product lines. IEEE Transactions on Software Engineering **99** (2011)
11. Rabkin, A., Katz, R.: Static extraction of program configuration options. In: ICSE'11, ACM (2011) 131–140
12. Acher, M., Cleve, A., Collet, P., Merle, P., Duchien, L., Lahire, P.: Reverse Engineering Architectural Feature Models. In: ECSA'11, Springer (2011) 220–235
13. Berger, T., She, S., Lotufo, R., Wasowski, A., Czarnecki, K.: Variability modeling in the real: a perspective from the operating systems domain. In: ASE'10, ACM (2010) 73–82

14. Boucher, Q., Abbasi, E.K., Hubaux, A., Perrouin, G., Acher, M., Heymans, P.: Towards more reliable configurators: A re-engineering perspective. In: PLEASE'12, co-located with ICSE'12. (2012)
15. Classen, A., Boucher, Q., Heymans, P.: A text-based approach to feature modelling: Syntax and semantics of tvl. Science of Computer Programming **76** (2011) 1130–1143
16. Czarnecki, K., Kim, C.H.P.: Cardinality-based feature modeling and constraints: A progress report. In: OOPSLA'05. (2005)
17. Benavides, D., Segura, S., Ruiz-Cortés, A.: Automated analysis of feature models 20 years later: A literature review. Information Systems **35**(6) (2010) 615–636
18. Michel, R., Classen, A., Hubaux, A., Boucher, Q.: A formal semantics for feature cardinalities in feature diagrams. In: VaMoS'11, ACM (2011) 82–89
19. Mesbah, A., van Deursen, A., Lenselink, S.: Crawling ajax-based web applications through dynamic analysis of user interface state changes. ACM Trans. Web **6**(1) (2012) 3:1–3:30
20. Madhavan, J., Ko, D., Kot, L., Ganapathy, V., Rasmussen, A., Halevy, A.: Google's deep web crawl. Proc. VLDB Endow. **1**(2) (2008) 1241–1252
21. Janota, M., Botterweck, G., Grigore, R., Marques-Silva, J.: How to complete an interactive configuration process? CoRR **abs/0910.3913** (2009)
22. Nohrer, A., Egyed, A.: Conflict resolution strategies during product configuration. In: VAMOS'10. (2010)
23. Hvam, L., Mortensen, N.H., Riis, J.: Product Customization. Springer-Verlag Berlin Heidelberg (2008)
24. Hubaux, A., Xiong, Y., Czarnecki, K.: A survey of configuration challenges in linux and ecos. In: VaMoS'12, ACM Press (2012) 149–155
25. Groher, I., Egyed, A.: Selective backtracking of model changes. In: ICSE'09, IEEE (2009) 231–234
26. Xiong, Y., Hubaux, A., She, S., Czarnecki, K.: Generating range fixes for software configuration. In: ICSE'12, IEEE (2012) 201–210
27. Kang, K., Cohen, S., Hess, J., Nowak, W., Peterson, S.: Feature-oriented domain analysis (foda) feasibility study. Technical report, Software Engineering Institute (1990)
28. Schmid, K., Rabiser, R., Grünbacher, P.: A comparison of decision modeling approaches in product lines. In: VaMoS'11, ACM (2011) 119–126
29. Hubaux, A., Classen, A., Mendonça, M., Heymans, P.: A preliminary review on the application of feature diagrams in practice. In: VaMoS'10. (2010) 53–59
30. Memon, A.M., Banerjee, I., Nagarajan, A.: GUI ripping: Reverse engineering of graphical user interfaces for testing. In: RE'03, IEEE (2003) 260–269
31. Staiger, S.: Static analysis of programs with graphical user interface. In: CSMR'07, IEEE (2007) 252–264
32. Vanderdonckt, J., Bouillon, L., Souchon, N.: Flexible reverse engineering of web pages with vaquista. In: WCRE'01, IEEE (2001) 241–248
33. Di Lucca, G.A., Fasolino, A.R., Tramontana, P.: Reverse engineering web applications: the WARE approach. J. Softw. Maint. Evol. **16**(1-2) (2004) 71–101
34. Amalfitano, D., Fasolino, A., Tramontana, P.: Experimenting a reverse engineering technique for modelling the behaviour of rich internet applications. In: ICSM'09, IEEE (2009) 571 – 574
35. Patel, R., Coenen, F., Martin, R., Archer, L.: Reverse engineering of web applications: A technical review. Technical report, The University of Liverpool (2007)