Reverse Engineering Web Configurators

Ebrahim Khalil Abbasi, Mathieu Acher, Patrick Heymans, Anthony Cleve

Reverse Engineering Web Configurators

Ebrahim Khalil Abbasi*, Mathieu Acher†, Patrick Heymans*, and Anthony Cleve*
*PReCISE, University of Namur, Belgium
{eab,phe,acl}@info.fundp.ac.be
† University of Rennes 1, Irisa, Inria France
mathieu.acher@irisa.fr

***Abstract*—A Web configurator offers a highly interactive environment to assist users in customising sales products through the selection of configuration options. Our previous empirical study revealed that a significant number of configurators are suboptimal in reliability, efficiency, and maintainability, opening avenues for re-engineering support and methodologies. This paper presents a tool-supported reverse-engineering process to semi-automatically extract configuration-specific data from a legacy Web configurator. The extracted and structured data is stored in formal models (e.g., variability models) and can be used in a forward-engineering process to generate a customized interface with an underlying reliable reasoning engine. Two major components are presented: (1) a *Web Wrapper* that extracts structured configuration-specific data from unstructured or semi-structured Web pages of a configurator, and (2) a *Web Crawler* that explores the "configuration space" (i.e., all objects representing configuration-specific data) and simulates users' configuration actions. We describe variability data extraction patterns, used on top of the Wrapper and the Crawler to extract configuration data. Experimental results on five existing Web configurators show that the specification of a few variability patterns enable the identification of hundreds of options.**

I. Introduction

A *Web Configurator* is an online product configuration environment for choosing products that match individual needs. Customized products are often characterised by hundreds of *configuration options*: customers gradually select the options to be included in the final product. A configurator provides an interactive graphical user interface (GUI) that guides the users throughout the configuration process (see Fig. 1 for an example). Web configurators are complex systems [1]–[3]: numerous kinds of constraints govern the options, the configuration process can be multi-step and non linear, and advanced capabilities are provided to check consistency, automatically complete undecided options, etc.

Our previous empirical study of 111 Web configurators [1] revealed the absence of specific, adapted, and rigorous methods in their engineering. Some of the Web configurators are developed like any other typical Web applications: proceeding this way leads to reliability, runtime efficiency, and maintainability issues. Specifically, we identified a large number of bad practices (incomplete reasoning, counter-intuitive representation of options, losing of all decisions when navigating backward, etc.) in the 111 configurators. Some of our industrial partners face similar problems and are now trying to migrate their legacy configurators to more reliable, efficient, and maintainable solutions [4]. To decrease the cost of migration, we propose to systematically *re-engineer* these applications. This encompasses two main activities: *(1)* reverse engineering a legacy configurator and encoding the extracted data into dedicated formalisms, and then *(2)* forward engineering new improved, customized configurator based on models [4]. The use of variability models to formally capture configuration options and constraints, and state-of-the-art solvers (e.g., SAT, CSP, or SMT) to reason about these models, would provide more effective bases [5]–[7].

In this paper, we focus on the *reverse-engineering* process. It consists of extracting configuration-specific data such as options, their associated descriptive information, and constraints, altogether called *variability data*, from the Web pages of the configurator, and then constructing a variability model, for instance, a *feature model* [5]. Building a complete feature model requires, ideally, analysing both the client and the server sides of a configurator. We investigate here the visible parts of configurators, i.e., the GUI of the *Web client* because it is the entry point for customer orders and most of the variability data is somehow represented in Web pages.

The major difficulty is that Web configurators, despite having a common goal and similar features, vary significantly: variation in implementation and presentation of configuration-specific objects as well as the way constraints govern the selection of options. For example, some options are all located in the same Web page; in other configurators, some options only appear in a new Web page once a certain selection has been performed. To the best of our knowledge, the problem of extracting feature models from Web configurators has not been studied. Existing techniques for reverse engineering feature models assume a formal representation of the constraints (e.g., through a formula [8]) or exploit specific artefacts (e.g., product descriptions [9], [10], dependency files [11], source code [12], [13]). Methods for reverse engineering Web GUIs (e.g., see [14]) do not propose *dedicated* techniques for (1) locating options in a Web page or for (2) analyzing the dynamics and the specificity of a configuration process.

This paper presents a novel tool-supported and supervised approach to reverse engineer Web configurators. The reverse-engineering tool consists mainly of two collaborative components: *Web Wrapper* and *Web Crawler*. A Web Wrapper extracts variability data from a Web page and transforms it into a structured data in a semi-automatic way. A Web Crawler focuses on the runtime behaviour of configurators. It explores the "configuration space" (i.e., all objects rep-

resenting configuration-specific data) and simulates (some of) users' configuration actions. The Crawler systematically generates dynamic variability data which is then extracted by the Wrapper. We describe the main foundations on top of which the Wrapper and the Crawler operate, that is, the notion of *variability data extraction pattern* (*vde* pattern in short). Users can specify a *vde* pattern, expressed in an HTML-like language, to extract the variability data from Web pages. The Web Wrapper, given a *vde* pattern (i.e., the specification of the structure of objects of interest), locates in a Web page code fragments (implementing objects of interest) that structurally conform to that pattern and then extracts their data . Experimental results show that the proposed language is expressive such that using a few patterns the user can extract hundreds of options presented in a page. They also confirm the ability of the Crawler to dynamically and automatically mine numerous additional configuration options and constraints.

Remainder. Section II presents a glimpse on Web configurators. Section III discusses related work. Section IV describes the overall reverse-engineering process. Section V is devoted to the basics of *vde* patterns. Section VI describes how patterns are used to crawl the configuration space. Section VII presents preliminary results of using the proposed techniques on five existing Web configurators. Section VIII concludes the paper and highlights future work.

II. A Glimpse on Web Configurators

Despite similar goals, Web configurators are unique and vary significantly: they each have their own widgets to represent options (check boxes, images, lists, etc.), reasoning techniques to handle constraints, and designs to organize the configuration process. The configurator of Audi (see Fig. 1) is only one example out of hundreds existing configurators [1]. Fig. 1 highlights a *configuration process* (Ⓐ) constituted of a sequence of *steps* (e.g., "1. Model" is followed by "2. Engine" – Ⓑ). The user follows the steps to complete the configuration of a product (a car in this example). Each step includes a subset of options which are presented through specific widgets (radio buttons and check boxes – Ⓒ and Ⓓ, respectively). Additionally, within a step, options are organized in different *groups* (e.g., "Exterior") and sub-groups (e.g., "Mirrors", "Windows"...). Options can be in different *configuration states* such as selected (e.g., "High-beam assist" is flagged with ✓), unavailable (e.g., "Light and rain sensors" is greyed out) or undecided (e.g., "Front fog lights"). Moreover, *descriptive information* (Ⓔ) may be associated to an option (e.g., its price).

A configurator can also implement *formatting*, *group*, and *cross-cutting* constraints [1]. A formatting constraint ensures that the text value set by the user is valid. A group constraint defines the number of options that can be selected from a group of options. An *alternative* group constraint determines that one and only one option must be selected (e.g., the "Model line" group), and a *multiple choice* group constraint tells that one and more options can be selected from grouped options (e.g., the "Headlights" group). Widget types used to implement these groups directly handle those constraints. For instance, radio buttons and single-selection list boxes are commonly used to implement *alternative* groups. A *cross-cutting* constraint is defined over two or more options regardless of their inclusion in a group and determines their valid combinations. For instance, the selection of "High-beam assist" implies the selection of "Driver's Information System" (Ⓕ), meaning that the user must select the latter if the former is selected. *Require* (selecting A implies selecting B) and *Exclude* (selecting A prevents selecting B and vice-versa) are the most common.

A variety of Web objects (e.g., layouts and widgets) can be used to visually represent configuration steps, options, constraint-handling windows, etc. In practice, there are as different structures and formatting features as configurators to implement these objects in the source code. Objects represented in the Web pages can be either a single slot data item (e.g., see Fig. 1 – Ⓒ) or a data record containing a block of related data items (e.g., in Fig. 1, the data object representing the "High-beam assist" option consists of an option name, its price, its image, and constraints).

Web configurators are highly interactive and dynamic applications. As they are executing, new content may be created and automatically added to the page, and existing content may be removed or changed. For instance, in Fig. 1, the selection of an option from the "Model line" group loads its consistent options in "Body style".

III. Related work

Reverse engineering a feature model from a Web configurator requires intersecting approaches coming from three fields of study: *reverse engineering Web applications*, *Web data extraction*, and *synthesizing feature models*.

Web and GUIs. Approaches have been proposed to reverse engineer GUIs and Web pages. *WebGUITAR* [15] discovers as much structural information about the GUI and behaviour as possible using automated algorithms and creates a structure called *GUI tree* for testing. *VAQUISTA* [16] recovers a *presentation model* from a single Web page for the purpose of migration of the user-interface to another environment. The WARE approach [17] seeks to understand, maintain and evolve undocumented Web applications by reverse engineering them to UML diagrams. *GuiSurfer* [18] analyses the source code of a Web application and generates its GUI layer model. *CRAWLJAX* [14] is a tool for crawling AJAX-based applications through dynamic analysis of user-interface-state changes.

Existing methods to reverse engineer Web applications mostly focus on recovering models at a greater level of abstraction at GUI and sometimes business levels. They do not seek to extract configuration data. Their use to reverse engineer feature models would require substantial changes to their core procedures: the algorithms they implement do not consider configuration aspects (e.g., configuration semantics of GUI elements) and specific properties of the highly dynamic and multi-step nature of a configuration process (e.g., choices may force the selection/exclusion of some other options, make visible new options or even new steps).

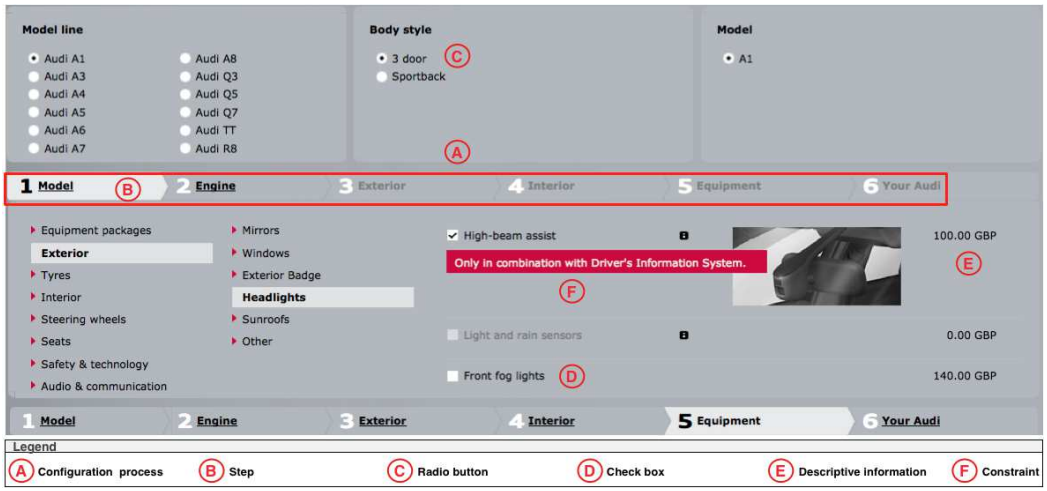

Fig. 1: Audi Web configurator (`http://configurator.audi.co.uk/`)

Web data extraction. Laender *et al.* [19] presented a method that aims to find an implicit structure associated with the objects in a given example page and uses this structure to extract new objects from similar pages. *ROADRUNNER* [20] automates the data extraction process by comparing Web pages of a Website and generating a wrapper based on their similarities and differences. Arasu *et al.* [21] studied the problem of automatically extracting the database value from template-generated Web pages. In [22], the authors investigated how to mine structured objects, called *data records*, in Web pages containing regularly structured objects. Web data extraction approaches are usually domain-oriented – constructing a generic method is complex (if not impossible) – and do not meet three important requirements we face when reverse engineering Web configurators.

Firstly, some of these approaches offer a Web crawler to navigate hyper-linked Web pages and extract data. In Web configurators, a more specific Web crawler is required both able (1) to automatically explore the configuration space, (2) to simulate users' configuration actions in order to automatically generate dynamic content, and (3) to dynamically detect changes made to the page with the aim of identifying and extracting newly added configuration-specific data. Secondly, these approaches do not study the configuration semantics and relationships between the extracted data. For instance, for options presented in a group (e.g., represented using radio buttons), not only the options should be extracted, but also the fact that an *alternative* group constraint is defined over these options. Thirdly, no user support is proposed to filter configuration data from other irrelevant data. This is especially important in Web configurators where a lot of information, not relevant from a configuration and thus a re-engineering perspective, can be found.

Reverse engineering feature models. Several authors have already addressed the reverse engineering of feature models from existing artefacts [8]–[13], [23]. Sources include user documentation, natural language requirements, formal requirements, product descriptions, dependencies, source code, architecture, etc. None of these approaches tackle the extraction of variability data from Web configurators.

IV. The Reverse Engineering Process

We address two main research questions:

RQ1. *What scalable Web data extraction methods can we use to collect accurate variability data from the Web pages of a configurator?* This research question addresses the problem of extracting *structured variability data* by static analysis of the source code of a page. We use *variability data extraction (vde)* patterns to specify variability information to be extracted. We also implemented a *Web Wrapper* which extracts structured variability data from a page given a set of *vde* patterns.

RQ2. *How to (semi-)automatically extract the dynamic variability content?* This research question addresses the runtime behaviour of Web configurators. We developed a *Web Crawler* that automatically explores the configuration space and simulates users' configuration actions. If the exploration and the configuration actions add new data to the page, the Wrapper extracts the newly added data.

Due to the high diversity of presentations and implementations found in Web configurators [1], a fully automated approach is neither realistic nor desirable. We consider that the data extraction process should be supervised. A user manually marks and names data to be extracted by giving it a meaningful label in a *vde* pattern specification. Consequently, (1) the user distinguishes configuration-specific data from the other irrelevant and noisy data, (2) she explicitly and accurately organizes data items in the extracted data records by assigning them different labels, and (3) representing the extracted data in a predefined data model becomes feasible, because the types and logical relationships of data to be extracted from Web pages of a configurator are rather known.

Fig. 2 depicts our proposed *supervised* and *semi-automatic* reverse-engineering process. Interactive (I) and automatic (A) activities are distinguished. The process starts with the specification of *vde* patterns for a given Web page (❶). The user inspects the source code of the page, identifies templates from

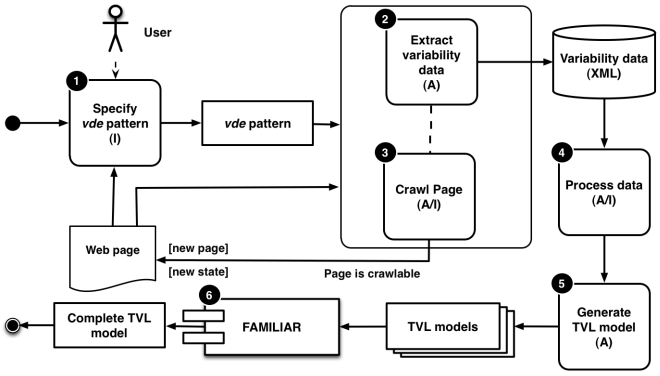

Fig. 2: The reverse engineering process

which the page is generated, specifies the appropriate *vde* pattern defining the structure of those templates, and marks the required data in the pattern. The specified *vde* pattern is given to the Web Wrapper. The Web Wrapper is a program that takes as input specification of a *vde* pattern and a Web page, seeks and finds code fragments in the page that *structurally* match the given pattern, and extracts as output data items from those code fragments corresponding to the marked data in the pattern (❷). The extracted data is hierarchically organized and serialized using an XML format. Most likely, the analysed Web page does not contain all configuration-specific data objects. New configuration content may be added to the page based on some selections of options. The Crawler simulates some of users' configuration actions in order to automatically generate dynamic content (❸). The newly added data is extracted by the Wrapper (❷). The content extracted in steps ❷ and ❸ can be edited (❹). The clean XML file is then given to a module which transforms it into a feature model (❺). We rely on the *Text-based Variability Language* (*TVL*) to represent feature models [24]. At the end of the reverse-engineering process there are typically several generated TVL models (e.g., each corresponding to a specific configuration step). To produce a fully-fledged TVL model, all these models are fed to *FAMILIAR* (❻), a tool-supported language to merge incomplete feature models into a single feature model [25].

V. Variability Data Extraction Pattern (RQ1)

Client-side source code is usually developed or generated from a number of Web templates. Web Configurators are no exceptions. Each Web page consists of a number of *template instances* which are syntactically identical fragments except for variations in values for data slots (text elements and tag attribute values) as well as minor changes to their structures. We take advantage of templates used in Web pages to extract the required data. Our main proposal is the notion of *vde* pattern, supported by an HTML-like language. All *vde* patterns required to extract data are specified in a *configuration file*. A configuration file contains the specification of at least one *data* pattern and one *region* pattern. A data pattern marks text elements and attributes carrying the content of interest, denoting code fragments (i.e., template instances) that match certain properties and thus contain the relevant data. A region pattern highlights a portion of the source code where the

```
1  <pattern data-att-met-pattern-name="dataPattern" data-att-met-pattern-type="data">
2    <div class="column1">
3      <div class="htmlTooltip" data-att-met-multiplicity="[0..1]">
4        <div class="message">data-tex-mar-constraint</div>
5      </div>
6      <div class="checkBox*"  data-att-met-clickable="true"
7                              data-att-met-unique="true">
8        skip(sibling,[1])
9        <span class="label">data-tex-mar-option-name</span>
10     </div>
11   </div>
12   <div class="column2">
13     <div data-att-met-multiplicity="[0..1]">
14       <img data-att-mar-image-src="@src"/>
15     </div>
16   </div>
17   <div class="column3 price">
18     <div class="single">data-tex-mar-price</div>
19   </div>
20 </pattern>
21 <pattern data-att-met-pattern-name="regionPattern" data-att-met-pattern-type="region">
22   <div id="selection">
23     <pattern>dataPattern</pattern>
24   </div>
25 </pattern>
```

(a) The configuration file containing the specified *vde* patterns

```
1  <div class="column1">
2    <div class="htmlTooltip">
3      <div class="message">
4        Only in combination with Driver's Information System.
5      </div>
6    </div>
7    <div class="checkBox checked" key="MSWS8N7" tooltip="html">
8      <span class="icon jqCheckBox"/>
9      <span class="label">High-beam assist</span>
10   </div>
11 </div>
12 <div class="column2">
13   <div>
14     <img alt="" class="equipmentImage" height="74"
15          src="http://mediaservice.audi.com/media/cdb/data/3ad.jpg"
16          style="display: inline;" width="159"/>
17   </div>
18 </div>
19 <div class="column3 price">
20   <div class="single">100.00 GBP</div>
21 </div>
22 <div class="column1">
23   <div class="checkBox disabled" key="MSWS8N6" tooltip="html">
24     <span class="icon jqCheckBox"/>
25     <span class="label">Light and rain sensors</span>
26   </div>
27 </div>
28 <div class="column2"></div>
29 <div class="column3 price">
30   <div class="single">0.00 GBP</div>
31</div>
```

(b) Input and targeted code source

Fig. 3

Wrapper will operate.

Example. A first example of a *vde* pattern is given in Fig. 3(a). Lines 1-20 specify a data pattern while lines 21-25 specify a region pattern. The overall *vde* pattern seeks to extract configuration data of the configurator presented early in Fig. 1. Fig. 3(b) lists an excerpt of the original source code used to visually represent the options – see lines 1-21 and 22-31, respectively for "High-beam assist" and "Light and rain sensors" options. Our Web Wrapper is then able to exploit the *vde* patterns in order to extract variability data, e.g., as contained in the source code of Fig. 3(b).

A. The syntax of vde patterns

We now further describe the syntactical constructs of a *vde* pattern, for operating over attributes, HTML and text elements.

Attributes. We distinguish three types of attributes in a pattern: data marking, structural, and meta.

A *data marking attribute* denotes the data item to be extracted from code fragments that match the pattern. The user uses a data marking attribute to mark an attribute whose value is of interest to her. A data marking attribute name is prefixed with *data-att-mar-*. For instance, `data-att-mar-image-src= "@src"` in Fig. 3(a) (line 14) tells the Wrapper from the `img` elements of the matching code fragments extract the value of the `src` attribute, assign its value to `data-att-mar-image-src`, and record it as output. The symbol @ tells the Wrapper to treat `src` as a named attribute, not a string value.

A *structural attribute* denotes a template-generated attribute, and therefore, all template instances share the same list of structural attributes. When the Wrapper maps two HTML elements in a given pattern and a code fragment, it counts the two elements identical if they have the same tag name and an analogy can be drawn between their structural attributes. The value of a structural attribute can contain (1) the *wild card* symbol (*) that captures zero or more characters and may be discarded when mapping two structural attributes, (2) the *or* operator (|) that represents an or-relationship between values of the structural attribute, and (3) the logical *not* operator (!). The *or* operator has the highest precedence in the attribute value. For instance, in Fig. 3(a) the `div` element (line 6) contains the `class="checkBox*"` attribute. A `div` element in a code fragment can be mapped onto the `div` element in the pattern, if it has the same tag position with that of the pattern's one (structural similarity) and has the `class` attribute whose value starts with `checkBox`. The `div` elements in lines 7 and 23 of the two code fragments shown in Fig. 3(b) are mapped to the `div` element in line 6 of the pattern.

A *meta attribute* represents a pattern-specific characteristic and its name is prefixed with *data-att-met-*. All meta attributes are *predefined* and reserved words in the pattern language. They guide information the Wrapper and the Crawler can exploit during the data extraction process. For instance, `data-att-met-clickable` (line 6) tells the Crawler which element should be clicked to simulate the user actions.

HTML elements. In addition to the predefined HTML elements, we add a pattern-specific element in our language, namely `pattern`. The `pattern` element is used to define a pattern (lines 1 and 21) or to refer to the name of a pattern in specification of another pattern (line 23).

Text elements. We consider three types of text elements in a *vde* pattern specification: data marking, structural and meta.

A *data marking text element* indicates a text element representing a data of interest to be extracted from the matching code fragments. It is prefixed with *data-tex-mar-*. For instance, `data-tex-mar-option-name` in Fig. 3(a) (line 9) tells the Wrapper that extract the child text element of elements that structurally map the `<span class="label">` element (line 9) and report out it as the option name.

`skip(sibling, Multiplicity)` is a *structural text element* and used to set up the Wrapper to skip some elements during mapping code fragments and a data pattern. `sibling` is a reserved word and the `Multiplicity` value is either a single positive integer number, the wild card (infinity) symbol (*), or a range. A range is denoted by stating the minimum and maximum positive integer values, separated by two dots and enclosed in braces (e.g., [1..5]). The maximum value can be the wild card symbol. A multiplicity value in the `skip` element indicates the number of consecutive sibling elements should be skipped by the Wrapper. For instance, the `skip(sibling,[1])` element in the pattern specification (line 8) guides the Wrapper to ignore the immediate sibling element of the `<span class="label">` element, consequently, the `span` elements in lines 8 and 24 of the two code fragments in Fig. 3(b) are not considered.

A *meta text element* denotes a pattern name when this pattern is used in specification of another pattern (Fig. 3(a), line 23).

B. The expressiveness of vde patterns

We now present how *vde* patterns can be used to deal with well-known expressiveness problems already reported in the general field of Web data extraction [26] and likely to impact the extraction of Web configurators.

Multi-instantiated elements. It is a common scenario for a code fragment to have multiple instances of an HTML element. To present this, we specify multiplicity of an element in the pattern specification. The multiplicity of an element is defined in the `data-att-met-multiplicity` meta attribute. The user may define the multiplicity of a pattern as well. By definition, the multiplicity of a pattern is *1..** and of an HTML element is *1*, if it is not explicitly defined. Semantically, the value of a multiplicity attribute specifies how many instances of the pertaining HTML element (respectively, the pattern) will be visited in a target code fragment (the source code) by the Wrapper. For instance, the multiplicity can be used to extract items (the `option` elements) of a list box:

```
<select>
 <option data-att-met-multiplicity="[*]">
    data-tex-mar-sub-option-name
 </option>
</select>
```

Optional elements. In the code fragments representing same data objects, one common variation is that an element may appear in some fragments and but not in all. This element is called an *optional* element. To present this variation in a pattern, we define the *0..1* multiplicity value for the optional element. For instance, in Fig. 3(b) the `<div class="column2">` element (line 12) in the first code fragment has a `div` child element (line 13), but the same element in the second fragment (line 28) does not have. To present this optionality in the pattern, the multiplicity of the optional `div` element defined to be *0..1* (Fig. 3(a), line 13).

C. The pattern matching algorithm

Given a configuration file and a Web page, the Wrapper operates within the block of the source code identified by the

region pattern and looks for code fragments that structurally match the data pattern. We briefly describe the matching algorithm with the code fragments and patterns depicted in Fig. 3. Our proposed algorithm provides a two-step solution to find matching code fragments: (1) first *finding candidate code fragments* may match the given data pattern and then (2) *traversing each candidate code fragment to find out if it is exactly matching the pattern*. The algorithm seeks to find mappings between elements of a code fragment and the given pattern using their tree representations.

The pattern matching algorithm is based on the following observation: a code fragment that matches the given data pattern is likely to have a unique element so that there is one and only one instance of that element in the code fragment. Our previous empirical observation [1] is that, for example, an option is represented using a widget and the element implementing the widget is most likely unique in the code fragment. The unique element in the data pattern is marked with the `data-att-met-unique = "true"` (line 7). We also define the *signature* of the unique element as all the elements in the path from the element up to the root element in the pattern. The signature of the unique element in the data pattern shown in Fig. 3(a) is: `<div class="column1">`. The *length* of a signature is the number of its included elements, e.g., the length of the unique element's signature in Fig. 3(a) is *1*.

A candidate code fragment of a given data pattern is a code snippet so that (1) there are one-to-one mappings between its first-level HTML elements and those of the data pattern, (2) it contains an element that is identical to the unique element and (3) the identical element has the same signature (with the calculated length) as the unique element. To find these candidate code fragments, the algorithm uses a *bottom-up* tree traversing. It first finds all HTML elements in the source code that are identical to the unique element and have the same signature as the unique element. For each found identical element, the algorithm then walks *l* steps up, in which *l* is the length of the signature. At the end of the bottom-up traversing, the algorithm stops on an HTML element, called the *index* element – in the source code it corresponds to a first-level HTML element in the data pattern. For instance in Fig. 3(b), the `<div class="column1">` elements (lines 1 and 22) are found and mapped to the similar element in the data pattern (see Fig. 3(a), line 2). The algorithm then propagates the mappings between the siblings of the index element in the source code and the data pattern. When the algorithm draws an analogy between the first-level elements of the data pattern and a code fragment in the source code, it records that code fragment as a candidate code fragment. Each candidate code fragment is identified with its first-level elements.

Once the candidate code fragments are identified, the algorithm uses a mixture of both *depth-first* and *breadth-first* traversals to find other mappings between each code fragment and the data pattern. During the traversal of a code fragment, its data items are also extracted. During the mapping, if a conflict is detected the target code fragment is ignored.

VI. Crawling the configuration space (RQ2)

The configuration space may be distributed over multiple pages each having a unique URL (*multi-page* user interface paradigm) or all the configuration-specific objects are contained in a page (*single-page* user interface paradigm). For configurators following the single-page paradigm, one common scenario is that when a Web page is loaded, the configuration space contains some configuration-specific objects and as the application is executing, new objects may be added to the page, and existing objects may be removed or changed. Configuring an option and exploring configuration steps are common actions to change the content of the page. By configuring an options its consistent options are loaded in the page. For instance, the selection of an option in the "Model line" group in Fig. 1 loads new options to the "Body style" group. Configuring an option may also change the configuration state of other impacted options. For instance, the selection of "High-beam assist" makes unavailable "Light and rain sensors". Note that both cases indicate that there are underlying constraints between those options, consequently, these constraints should be extracted as well. Activation of a step makes available its options in the page and makes unavailable those of other steps.

To extract dynamic data, we need to automatically crawl the configuration space in a Web page. Automatically crawling requires (1) the simulation of users' configuration and exploration actions to systematically generate new content or alter the existing content and then (2) the analysis of the changes made to the page to deduce and extract configuration-specific data. The Web Crawler and the Wrapper collaborate together to deal with these cases.

At present, the Crawler is able to simulate some of users' actions, for instance, the selection of items from a list box and the click on elements (e.g., button, radio button, menu, image, etc.). The element to be clicked by the Crawler is identified by the `data-att-met-clickable = "true"` attribute in the pattern specification (for instance, Fig. 3(a), line 6).

The simulation of user actions may change the content of the page, therefore, after simulating every clickable element, the page's content must be analysed to identify the newly added content and to deduce from that the configuration-specific data. We observed that when a configuration action is performed by the user, a few identifiable regions on the page are impacted and their content may be changed. Consequently, rather than analysing the whole page, only those regions should be investigated. Based on this observation, we divide the configuration-specific regions of a page into two groups: *independent* and *dependent* regions. When a configuration action is performed on a configuration-specific object in an independent region, new objects are added to the dependent regions or existing ones are changed. We define the notion of *dependency* between *vde* patterns to formulate this observation.

Definition. Let P_1 and P_2 be two data patterns. A *dependency* is a relationship that semantically relates a set of code fragments that match P_2 to a code fragment that matches P_1.

We respectively call P_1 and P_2 as *independent* and *dependent* patterns.

To define the dependency between two data patterns we use the `data-att-met-dependent-pattern` attribute. This attribute is specified in the region pattern owning the independent data pattern and its value is a comma-separated list of region patterns containing the dependent data patterns. In fact, the independent region pattern denotes the region of clickable elements and the dependent region pattern indicates the region of added/changed objects.

Fig. 4 presents the dependencies between *vde* patterns defined to crawl and extract options from the "Model" step shown in Fig. 1. The selection of an option from the "Model line" group adds its consistent options to the "Body style" group, and in turn, the selection of an option from the latter group loads new options to the "Model" group. The *modelLine*, *bodyStyle*, and *model* region patterns respectively denote the "Model line", "Body style", and "Model" groups. Two dependencies are defined between the *bodyStyle* and *modelLine* patterns (line 2) as well as between the *model* and *bodyStyle* patterns (line 8). *dataPattern* (lines 18-24) is specified to extract options represented using radio buttons and is used by all the region patterns (lines 4, 10, and 15). The Wrapper starts the process from the "Model line" group and extracts data from the first code fragment that matches *dataPattern*, consequently, the "Audi A1" is extracted. Then, the Crawler selects this option by clicking on the widget representing it (line 20). This selection loads the new options "3 door" and "Sportback" to the "Body style" group. From the `data-att-met-dependent-pattern = "bodyStyle"` the Crawler finds that the next region pattern to be investigated is *bodyStyle*, thus, it calls the Wrapper to process this pattern. The Wrapper extracts the "3 door" and then the Crawler selects this option which leads to load "A1" to the "Model" group. Similarly, by analysing the *bodyStyle* pattern, the Crawler detects that the *model* pattern is the next dependent pattern to be examined, therefore calls the Wrapper for this pattern. The Wrapper starts extracting data from the "Model" group and extracts "A1". Once done, the Crawler selects "A1" but since there is no dependent pattern defined for the *model* pattern, the Crawler stops. The Wrapper finds no more options to be extracted from the "Model" group, comes one step back, and considers the "Sportback" option in the "Body style" group. Once that all the options from the "Body style" and "Model" groups are extracted, the Wrapper returns back to the "Model line" group and takes "Audi A3" as the next option to be processed. This process iterates until all the options in the "Model line" group, and accordingly in "Body style" and "Model" are extracted.

It may happen that no new option is added to the page once a selection is performed. Instead, the configuration state of existing objects are changed. For instance, in the "Equipment" step in Fig. 1, when an option is given a new value, the configurator automatically propagates the required changes to all the impacted options. In this case, crawling is a way to instantiate and then extract such constraints. Technically, when the Crawler configures an option, the Wrapper extracts all the contained options and their states. Therefore, at the end of the process all the visited configuration states of all the options are documented in the output XML file. These state changes are then analysed to identify which constraints logically impact (e.g., through exclusions or implications) other options.

```
1  <pattern data-att-met-pattern-type="region" data-att-met-pattern-name="modelLine"
2           data-att-met-dependent-pattern="bodyStyle">
3    <div class="*gridLeft">
4      <pattern>dataPattern</pattern>
5    </div>
6  </pattern>
7  <pattern data-att-met-pattern-type="region" data-att-met-pattern-name="bodyStyle"
8           data-att-met-dependent-pattern="model">
9    <div class="*gridCenter">
10     <pattern>dataPattern</pattern>
11   </div>
12 </pattern>
13 <pattern data-att-met-pattern-type="region" data-att-met-pattern-name="model">
14   <div class="*gridRight">
15     <pattern>dataPattern</pattern>
16   </div>
17 </pattern>
18 <pattern  data-att-met-pattern-type="data" data-att-met-pattern-name="dataPattern">
19   <div class = "radioButton|*checked*"  data-att-met-unique="true"
20        data-att-met-clickable="true">
21     skip(sibling, [1])
22     <span class ="label">data-tex-mar-option-name </span>
23   </div>
24 </pattern>
```

Fig. 4: Dependency between *vde* patterns

VII. Implementation and Evaluation

A. Tool Support

We developed a *Firebug*[1] extension that consists of the Wrapper and the Crawler components. The extension generates an XML file which presents the output data. The generated XML file is then given to a Java application which converts it to the corresponding TVL model. The specification of the pattern language, tools, and the complete set of data are available at http://info.fundp.ac.be/~eab/result.html.

B. Evaluation

1) Experimental Setup:

Goal and scope. Our approach aims to reverse engineer feature models from Web configurators. We want to (1) evaluate the ability of the approach to deal with variations in presentation and implementation of variability data, (2) assess the *accuracy* of the extracted data, and (3) measure the users' *manual effort* required to perform the extraction.

Questions and metrics. We address four questions (Q):

- **Q1.** How accurate is the extracted data?
- **Q2.** How expressive is the pattern language?
- **Q3.** How applicable is the crawling technique?
- **Q4.** How much manual effort is needed to perform the reverse-engineering process?

The underlying metrics (M) are formulated as follows:

- **M1.** The number of patterns required to extract data.
- **M2.** The number of lines of code (LOC) of patterns. For each pattern, we put one and only one element in each line to count M2.

[1] http://getfirebug.com/

- **M3.** The number of pattern dependencies specified in order to crawl the configuration space.
- **M4.** The number of times the data extraction procedure is executed.
- **M5.** The precision of the automatically extracted options.
- **M6.** The percentage of correct objects extracted by the crawling technique.
- **M7.** The percentage of manually added options.
- **M8.** The precision of the automatically extracted constraints.
- **M9.** The percentage of automatically extracted correct cross-cutting constraints.
- **M10.** The percentage of manually added constraints.

Data set. We took the five configurators S1-S5 listed in Table I. S1 is the Dell's laptop configurator. We took the "Inspiration 15" model in this experiment. S2 is the car configurator of BMW. For this study, we chose the "2013 128i Coupe" model. S3 is a dog-tag generator, in S4 the customer can choose her chocolate and create its masterpiece and ingredients, and S5 is a configurator that allows customers to design their shirts.

Execution. We used the first author of the paper to supervise the extraction process. For each Web page, we first inspected its source code to find out which templates are used and then specified the required patterns to extract data (M1 and M2).

Our original intention was to specify a minimum set of patterns to extract all options presented in the page or identified by the crawling technique. For each option, we extracted its name, image source (for image options), and other attached descriptive information (e.g., price). After running the Wrapper for the given patterns (M4), we manually checked the extracted data objects to find out the missing or noisy data. We reported the precision of the data automatically extracted by the Wrapper (True Positive – M5). We either altered the existing patterns or specified new ones to achieve 100% accuracy (if possible). Using the Crawler we simulated the click event on every option to recognize if it creates and adds new data to the page. If yes, we then specified dependencies between corresponding patterns (M3) to extract this dynamic data (M6). We also manually added some options to the extracted data (M7). There are (1) options that the Wrapper failed to identify and extract them (false negative options), (2) options we did not specify any pattern to extract them (because manual extraction is much better than writing some lines of code), or (3) options present group names.

Considering constraints, we looked for formatting constraints in descriptive information attached to options, attributes such as "maxlength" and "size" of text elements representing options, etc. To extract group constraints, the Wrapper analyses the types as well as attributes such as the "name" of the widgets used to represent options. To trigger and extract cross-cutting constraints (M9), we simulated the user's configuration actions using the crawling technique (M3). For the automatically extracted constrains, we manually checked their correctness and reported their precision (M8). We also manually added the missing constraints and corrected the invalid ones (M10).

TABLE I: Experimental results. **S1**: *http://www.dell.com*, **S2**: *http://www.bmwusa.com*, **S3**: *http://www.mydogtag.com*, **S4**: *http://www.choccreate.com*, **S5**: *http://www.shirtsmyway.com*

PATTERN SPECIFICATION

System	Pattern (M1)		LOC (M2)	Dependency (M3)	Execution (M4)
	Region	Data			
S1	1	1	24		3
S2	3	5	90	5	14
S3	4	5	82	2	8
S4	2	2	40	1	2
S5	3	6	86		11
Total	13	19	322	8	38

OPTION

System	All		T.P. (M5)	Dynamic (M6)	Manual (M7)
	Automatic	Manual			
S1	233	3	100%		1.8%
S2	97	7	100%		6.7%
S3	200	8	100%	31.5%	3.8%
S4	119	2	100%	80%	1.7%
S5	233 F.P. 8	14	96.7%		5.7%
Total	882 F.P. 8	34	99%	18%	3.7%

CONSTRAINT

System	All		T.P. (M8)	Cross cutting (M9)	Manual (M10)
	Automatic	Manual			
S1	Group 49	Group 12	100%		19.7%
S2	Group 7 F.P. 1 Cross-cutting 30	Group 4	97.3%	73.1%	9.8%
S3	Group 14 Formatting 10	Formatting 1	100%		4%
S4	Group 7		100%		
S5	Group 26 Formatting 40		100%		
Total	Group 103 Formatting 50 Cross-cutting 30 F.P. 1	Group 16 Formatting 1	99.4%	16.3%	8.5%

2) Experience and results:

We now report on our experience and results (see Table I).

S1. In S1, we specified only one data pattern (M1) to extract all options presented using radio buttons or check boxes. S1 provides a three-step configuration process such that by activating each step the page is reloaded containing the step's options. So, we had to manually activate each step and then run the extraction procedure for that (M4). The Wrapper could extract all options presented in S1 (M5). It also identified 49 true group constraints (M8). We manually added three group names (M7) and 12 group constraints (M10).

S2. We specified four data patterns (M1) to correctly extract all options presented using images, check boxes and labels in S2 (M5). We also defined two dependencies (M3)

to document parent-child relationships between options and sub-options. As to constraints, the Wrapper could extract 8 group constraints one of which was incorrect (False Positive – M8). In this case, all the options extracted from a group were image options, and the Wrapper considered the group as an *alternative* group. However, manual testing revealed that they semantically implement two different *alternative* groups. S2 uses two strategies to handle cross-cutting constraints. First, when an option is configured and one or more constraints apply, the configurator presents a *conflict window* and lists the names of the impacted options. We specified a data pattern (M1) to extract the content of the conflict window, two dependencies (M3), and used the crawling technique to trigger and extract the cross-cutting constraints. In the second strategy, the selection of an option changes the configuration state of the impacted options. We therefore defined another dependency (M3) and used the Crawling technique to analyze the state changes. In total we extracted 30 cross-cutting constraints from S2 (M9). We also manually added seven group names (M7) and four group constraints (M10).

S3. Options in S3 are presented using either text boxes or radio buttons. We specified a data pattern for text options. We identified four different templates from which radio button options are generated and specified three data patterns to extract these options. The reason for using different templates for radio button options is that they present different attached descriptive information. For some options, only the option's short name is presented and when the user clicks on the option, its full name, size, and price are dynamically added to the page. We so defined a data pattern to denote this dynamically-generated data objects (M1). We used the crawling technique in two cases. In the first case, the selection of an option loads its consistent options in the page and hides the irrelevant options. We defined a dependency between the corresponding patterns to dynamically generate and extract these data objects by the Crawler. The second case corresponds to the dynamic loading of additional descriptive information based on the click of an option. We defined another dependency and used the Crawling technique to extract these dynamic data objects. In total, 31.5% of the automatically extracted data are identified and extracted by the crawling technique (M6). Considering constraints, the Wrapper identified 14 true group constraints and ten true formatting constraints (M8). We manually added eight group names (M7) and one formatting constraint (M10).

S4. We specified one data pattern to extract options presented using radio buttons and check boxes in S4. Some options are also categorized into a set of groups. We specified another data pattern to extract the name of these groups (M1). By activating a group (i.e., activating its corresponding menu) its contained options are presented to the user and of other groups become hidden. We defined a dependency between the pattern that denotes the groups and the one indicates the options (M3) and ran the Crawler. 80% of the automatically extracted data are identified by the Crawler (M6). The Wrapper identified seven true group constraints (M8) in S4. We only added two group names to the extracted data (M7).

S5. We specified six data patterns to extract options presented using images, text boxes, radio buttons, check boxes, and list boxes (M1). Given these patterns, the Wrapper extracted 241 options eight of which were incorrect options (M5). The reason for these false positives is that we did not consider the visibility of options and extracted all options presented in the source code of the page. The Wrapper could identify 66 true group and formatting constraints (M8). We also manually added 11 group names and three text options (M7).

3) Discussion:

Accuracy of the extracted data (Q1). For the cases we applied the proposed approach, the accuracy of the extracted data is promising. Hundreds of configuration options, their attached descriptive information, and constraints defined over these options are automatically extracted and hierarchically organized. 99% of the extracted options (M5) and 99.4% of the extracted constraints (M8) are true data.

Expressiveness of the pattern language (Q2). We could specify patterns to cover all code fragments that implement configuration-specific objects. Pattern-specific elements and operators we designed in the language gave us a lot of support in specification of patterns for templates we identified in this experiment. We specifically observed that the use of wild card (*) and *or* (|) operators in the attributes and the `skip(sibling,Multiplicity)` element is very useful in minimizing the set of required patterns. We also found the notion of multiplicity of an element very practical in this experiment. For instance, the items of list boxes in S5 and the list of attached sub-options in S2 are examples of multi-instantiated elements that we could model them in the patterns.

Applicability of the crawling technique (Q3). Using the crawling technique, we could study the dynamic nature of the configuration process. We gain numerous additional configuration options and constraints with the crawling technique. 18% of the automatically extracted options (M6) and 16.3% of the constraints (M9) are identified and extracted using the crawling technique. Moreover, dependency between patterns allowed us to document the parent-child relationships between options in S2. All this data are collected by specifying eight dependencies (M3). Nevertheless we cannot claim that the crawling technique can detect and extract *all* objects that may be dynamically generated at runtime. We neither have base models to which we could compare our generated models nor have access to the developers of the studied configurator who can validate our models. It is worth to mention other experiences in reverse engineering contexts [9], [11], [13] showing that incomplete feature models may be obtained, thus calling for the intervention of the user or any kind of knowledge/artefact to further refine the model [27].

The manual effort required to perform the extraction process (Q4). In this experiment, overall we specified 13 region and 19 data patterns (M1), wrote 322 lines of code for these patterns (M2), and executed them 38 times (M4) to extract all data. 3.7% of the collected options (M7) and 8.5% of the constraints (M10) are manually added to the

automatically extracted data. The manual writing 322 lines of code to specify the required patterns in this experiment led to generating TVL models with 4478 lines of code. We believe that our semi-automatic and supervised approach provides a realistic mix of manual and automated work. It acts as an interesting starting point for re-engineering a configurator while mining the same amount of information manually is clearly daunting and error-prone.

C. Threats to Validity

The main *external* threat to validity is the sample set of the subject systems involved in our evaluation. Among configurators coming from 21 industries, we only chosen samples from 5 sectors. A large-scale evaluation will confirm better the scalability of the approach.

An *internal* threat to validity is that our approach is supervised and the technical knowledge of the user running the extraction process, her choices, and interpretations can influence the results. This experiment was conducted by the first author. He proposed and developed the notion of *vde* patterns and implemented the tools. His choices on what data objects to be extracted from each Web configurator influenced the number of required data patterns. For instance in S3, if a user intends to extract only the name of options (and to exclude their widget type and price) 4 (instead of 5) different patterns are required.

VIII. Conclusion

We presented the first tool-supported and supervised process to reverse engineer feature models from Web configurators. The main contribution of our work is the formal foundation and the development of *variability data extraction (vde)* patterns. The user-defined patterns are specified through an HTML-like language to mark configuration-specific data. Patterns are given to a Web Wrapper which uses a source code pattern matching algorithm to find code fragments. For the dynamic aspect we devise a Web Crawler that systematically extracts data by simulating the user's exploration and configuration actions. We developed a Firebug extension and a Java application to support the reverse-engineering process. We evaluated the scalability of the approach, the accuracy of the extracted data, and the manual effort is required to perform the process. The evaluation shows that a small set of patterns can extract numerous options presented in the page. The experiment also confirmed the efficiency of the Web Crawler to extract dynamic data and detect constraints.

Future work. We will continue this work in different directions. First, we have plans to apply the proposed approach to a large set of configurators coming from different industry sectors. Second, we intend to integrate our approach with Web crawling techniques [14] aiming to explore pages in Websites that follow multi-page user interface paradigm. Our long term goal is to decrease the practitioners effort when re-engineering their (legacy) configurators towards more reliable solutions.

References

[1] E. Abbasi, A. Hubaux, M. Acher, Q. Boucher, and P. Heymans, "The anatomy of a sales configurator: An empirical study of 111 cases," in *CAiSE'13*, ser. LNCS. Springer, 2013, vol. 7908, pp. 162–177.

[2] 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.

[3] A. Trentin, E. Perin, and C. Forza, "Sales configurator capabilities to prevent product variety from backfiring," in *Workshop on Configuration (ConfWS)*, 2012.

[4] 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.

[5] D. Benavides, S. Segura, and A. Ruiz-Cortés, "Automated analysis of feature models 20 years later: A literature review," *Information Systems*, vol. 35, no. 6, pp. 615–636, 2010.

[6] M. Janota, "SAT solving in interactive configuration," Ph.D. dissertation, University College Dublin, Nov. 2010.

[7] Y. Xiong, A. Hubaux, S. She, and K. Czarnecki, "Generating range fixes for software configuration," in *ICSE'12*, 2012, pp. 58–68.

[8] N. Andersen, K. Czarnecki, S. She, and A. Wasowski, "Efficient synthesis of feature models," in *SPLC'12*. ACM, 2012, pp. 106–115.

[9] J.-M. Davril, E. Delfosse, N. Hariri, M. Acher, J. Cleland-Huang, and P. Heymans, "Feature model extraction from large collections of informal product descriptions," in *ESEC/FSE'13*, 2013.

[10] E. N. Haslinger, R. E. Lopez-Herrejon, and A. Egyed, "On extracting feature models from sets of valid feature combinations," in *FASE'13*, ser. LNCS, vol. 7793, 2013, pp. 53–67.

[11] M. Acher, A. Cleve, P. Collet, P. Merle, L. Duchien, and P. Lahire, "Extraction and evolution of architectural variability models in plugin-based systems," *Software and Systems Modeling*, pp. 1–28, 2013.

[12] T. Ziadi, L. Frias, M. A. a. A. da Silva, and M. Ziane, "Feature identification from the source code of product variants," in *CSMR'12*. IEEE, 2012, pp. 417 –422.

[13] S. She, R. Lotufo, T. Berger, A. Wasowski, and K. Czarnecki, "Reverse engineering feature models," in *ICSE'11*. ACM, 2011, pp. 461–470.

[14] 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.

[15] B. Nguyen, B. Robbins, I. Banerjee, and A. Memon, "Guitar: an innovative tool for automated testing of gui-driven software," *Automated Software Engineering*, pp. 1–41, 2013.

[16] J. Vanderdonckt, L. Bouillon, and N. Souchon, "Flexible reverse engineering of web pages with vaquista," in *WCRE'01*, 2001.

[17] 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.

[18] C. E. B. e. M. de Silva, "Reverse engineering of rich internet applications," Ph.D. dissertation, University of Minho, Portugal, 2010.

[19] A. H. F. Laender, B. Ribeiro-Neto, and A. S. da Silva, "Debye - date extraction by example," *Data Knowl. Eng.*, vol. 40, no. 2, pp. 121–154, Feb. 2002.

[20] V. Crescenzi, G. Mecca, and P. Merialdo, "Roadrunner: Towards automatic data extraction from large web sites," in *VLDB '01*, 2001.

[21] A. Arasu and H. Garcia-Molina, "Extracting structured data from web pages," ser. SIGMOD '03. ACM, 2003, pp. 337–348.

[22] B. Liu, R. Grossman, and Y. Zhai, "Mining data records in web pages," ser. KDD '03. ACM, 2003, pp. 601–606.

[23] 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.

[24] A. Classen, Q. Boucher, and P. Heymans, "A text-based approach to feature modelling: Syntax and semantics of tvl," *Sci. Comput. Program.*, vol. 76, no. 12, pp. 1130–1143, Dec. 2011.

[25] M. Acher, P. Collet, P. Lahire, and R. B. France, "Familiar: A domain-specific language for large scale management of feature models," *Science of Computer Programming*, vol. 78, no. 6, pp. 657 – 681, 2013.

[26] C.-H. Chang, M. Kayed, M. R. Girgis, and K. F. Shaalan, "A survey of web information extraction systems," *IEEE Trans. on Knowl. and Data Eng.*, vol. 18, no. 10, pp. 1411–1428, Oct. 2006.

[27] C. Henard, M. Papadakis, G. Perrouin, J. Klein, and Y. Le Traon, "Towards automated testing and fixing of re-engineered feature models," in *ICSE '13 (NIER track)*. IEEE, 2013, pp. 1245–1248.