DOCTOR OF SCIENCES

Reverse Engineering Web Configurators

Abbasi, Ebrahim Khalil

Award date:
2014

Awarding institution:
University of Namur

Link to publication

PReCISE Research Centre

Faculty of Computer Science

University of Namur

Reverse Engineering Web Configurators

Ebrahim Khalil Abbasi

A thesis submitted in fulfilment of the requirements for the degree of Doctor of Science

in the

PReCISE Research Centre
Faculty of Computer Science
University of Namur

March 2014

Jury

Prof. Pierre-Yves Schobbens, University of Namur, Belgium (chair)
Prof. Kim Mens, Catholic University of Louvain, Belgium
Prof. Mathieu Acher, University of Rennes, France
Prof. Anthony Cleve, University of Namur, Belgium
Prof. Patrick Heymans, University of Namur, Belgium (advisor)

Abstract

In many markets, being competitive echoes with the ability to propose customized products at the same cost and delivery rates as standard ones. As a result, companies provide their customers with online *Web configurators* to facilitate the product customization task. Web configurators offer a highly interactive configuration environment for customers to specify products that match their individual requirements and preferences. They provide capabilities to guide the customer through the multi-step and non linear configuration process, check consistency, and automatically complete partial configuration.

To get a better grasp of what is the current practice in engineering Web configurators, we conducted a systematic empirical study of 111 configurators. We quantified their numerous properties, categorized patterns used in their engineering, and highlighted good and bad practices. We provided empirical evidence that Web configurators are complex information systems. Despite of this fact, this study revealed the absence of specific, adapted, and rigorous methods in their engineering. The lack of dedicated methods for efficiently engineering Web configurators leads to reliability, runtime efficiency, and maintainability issues.

To migrate legacy Web configurators to more reliable, efficient, and maintainable solutions, we offer to systematically *re-engineer* these applications. This encompasses two main activities: (1) *reverse engineering* Web configurators to extract their configuration-specific data and encoding it into dedicated formalisms, and then (2) *forward engineering* new improved configurators. In this study, we are concerned with the reverse-engineering process. We developed a consistent set of methods, languages and tools to semi-automatically extract configuration-specific data from the Web pages of a configurator. Such data is stored in *variability models* (e.g., feature models). These models can later be used for verification purposes (e.g., checking the completeness and correctness of the configuration constraints) as well as input for forward-engineering techniques.

To reverse engineer variability models from Web configurators, we developed techniques that target static structure and dynamic behaviour of Web configurators to locate and extract configuration-specific data. Experimental results on existing real Web configurators confirm the applicability of our contribution.

Résumé

Dans de nombreux marchés, la compétitivité passe par la possibilité de fournir des produits dédiés à des taux de production et à des prix identiques à ceux dits “standards”. A cette fin, les sociétés fournissent à leurs clients des “configurateurs” web afin que ceux-ci puissent spécifier les options des produits répondant à leurs attentes. Ces outils offrent des environnements interactifs qui guident les utilisateurs à travers des processus à plusieurs étapes et souvent non-linéaires, vérifient la correction des options et complètent automatiquement les configurations partielles.

Afin de mieux comprendre les pratiques actuelles de conception de ces configurateurs, nous avons effectué une étude empirique sur 111 configurateurs. Nous avons examiné et quantifié leurs divers attributs, organisé les différents patrons de conception utilisés et souligné les bonnes et les moins bonnes pratiques. Cette étude a révélé qu’un configurateur est en fait un système d’information complexe. Notre étude à aussi révélé qu’il n’y avait pas d’approche systématique, dédiée et rigoureuse pour construire de tels configurateurs. Ce manque nuit fortement à la fiabilité, la performance et la maintenance de ces systèmes.

Pour pouvoir migrer les configurateurs existants vers des solutions offrant une meilleure performance, fiabilité et evolutivité, nous pensons qu’il faut les reconcevoir de manière systématique. Cette approche comprend deux grandes étapes : 1) rétro-conception de configurateurs web afin d’en extraire les données de configuration et leur encodage dans des langages facilitant l’analyse, et 2) Génération de configurateurs améliorés. Dans cette thèse, nous intéressons à la première étape, pour laquelle nous avons développé une approche cohérente visant à extraire de manière semi-automatique les informations de configuration spécifiques des configurateurs web. Ces données sont ensuite encodées dans des modèles de variabilité (“feature models”). Ces modèles peuvent être utilisés par la suite pour pour vérifier la cohérence et la complétude des contraintes de configurations. Ils servent aussi de point de départ au processus de génération de nouveaux configurateurs.

Nos outils de rétro-conception ciblent la structure et le comportement dynamique des configurateurs pour localiser et extraire les informations spécifiques de configuration. Nos résultats empiriques obtenus sur des configurateurs existants établissent l’applicabilité de notre approche.

Acknowledgements

I would not have been able to finish my PhD thesis without the guidance, encouragement, and help of many people. I would like to extend my appreciation to the following.

My special appreciation and sincere gratitude goes to my advisor Prof. Patrick Heymans for the support of my PhD studies and for the motivation, guidance, and help he provided to me. Patrick has been a patient mentor as well as a great and funny friend. Without his help I would not be where I am today. Thank you Patrick.

I would like to express my thanks to my jury members, Prof. Pierre-Yves Schobbens, Prof. Kim Mens, Prof. Mathieu Acher, and Prof. Anthony Cleve, for letting my defense be an enjoyable moment, for their encouragement, excellent comments, and constructive questions. I would also thank Prof. Elliot Chikofsky for his awesome advice on my thesis.

I am indebted to my friends for keeping in touch, letting me present them my work, and helping me. I am especially grateful to Quentin Boucher, Andreas Classen, Maxime Cordy, Arnaud Hubaux, Nicolas Genon, Raphael Michel, Gilles Perrouin, and Germain Saval. I cannot thank them enough for their kindness and support.

A special thanks to my family, my mother, my father, my sisters, my brothers, and my beautiful nieces and nephews. Words cannot express how deeply thankful I am for their prayers and endless support throughout my life.

Finally and foremost, I am eternally grateful to my beloved Leila, who is my constant support. She has done so much for me. Leila has always been there for me with encouraging words and a lot of love. Without her, it would not have been possible to finish this work. Thank you Leila.

Contents

List of Figures

List of Tables

Abbreviations

UI	**U**ser **I**nterface
FM	**F**eature **M**odel
GUI	**G**raphical **U**ser **I**nterface
DOM	**D**ocument **O**bject **M**odel
vde	**v**ariability **d**ata **e**xtraction
LOC	**L**ines **O**f **C**ode

I dedicate this thesis to

my lovely Leila and my wonderful family

for their constant love and support.
I love you all dearly.

Chapter 1

Introduction

1.1 Mass Customization

After the industrial revolution, companies have set up a *mass production line.* Mass production is the production of a large amount of similar products to bring the products to market as quickly as possible at low costs. In mass production, low costs are achieved primarily through economies of *scale* – lower unit costs of a single product or service through greater output and faster throughput of production process [Pine, 1993]. While *standardized* products are best produced in an *assembly line* mass production environment [Hayes and Wheelwright, 1979], many companies are currently experiencing increasing demand from their customers for the delivery of *customized* products that have almost the same delivery time, price and quality as mass-produced products. One way this development is described is by the concept of *mass customization* – a production form in which customized products are delivered by exploiting the advantages of mass production [Hvam et al., 2008]. Whereas mass production's primary goal is to produce standardized products at a price that everyone can afford, the goal of mass customization is to produce enough variety in products and services so that nearly everyone finds exactly what he/she wants at a reasonable price [Pine, 1993].

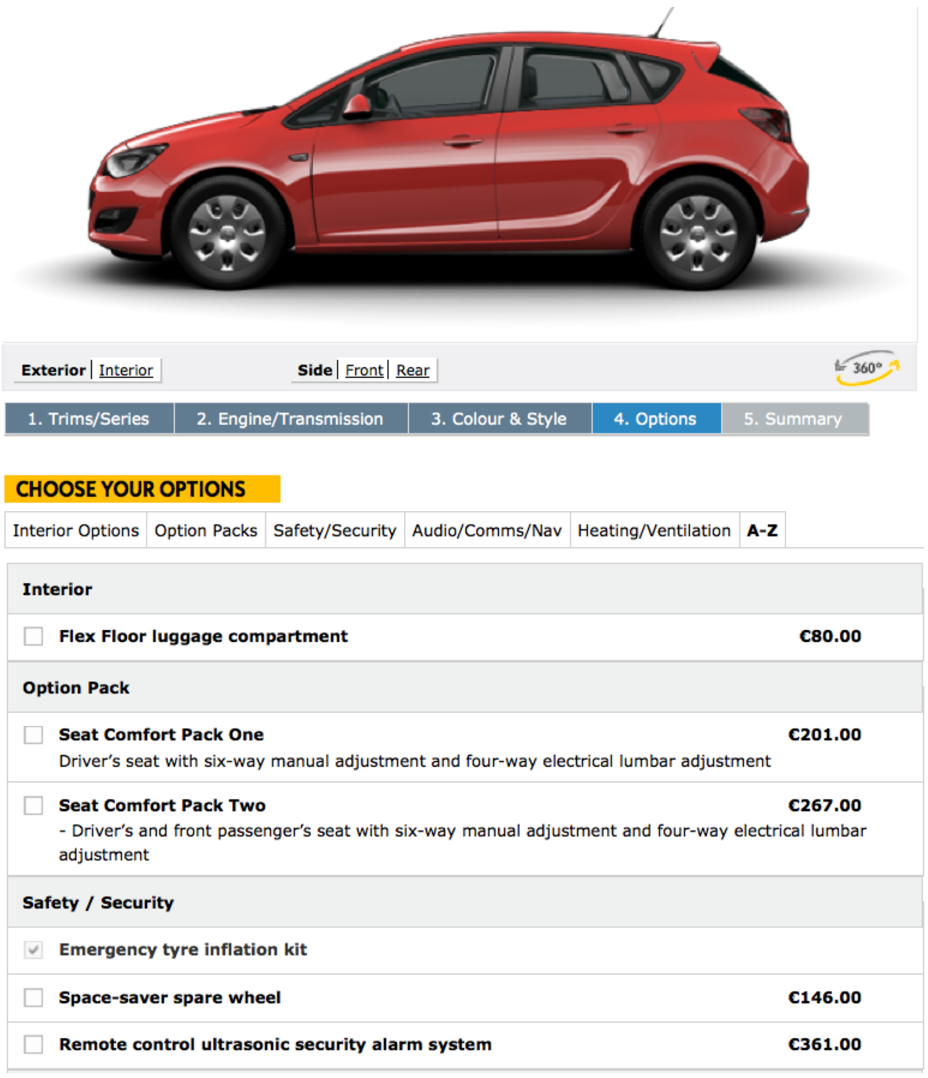
Mass customization is producing goods and services to meet individual customers' needs with near mass production efficiency [Tseng and Jiao, 2007]. It in fact aims to provide a trade-off between product variety (i.e., flexibility) and production cost (i.e., efficiency) [Hayes and Wheelwright, 1979; Kotha, 1995]. In mass customization, low costs are achieved primarily through economies of *scope* – the application of a single process to produce a greater variety of products or services more cheaply and quickly [Pine, 1993].

One key element in a mass customization strategy is to build products by selecting, combining and possibly adapting a set of standard modules. In other words, mass customization is mass production of standard modules and customer-initiated assembly of customized products based on the use of modules [Hvam et al., 2008]. These modules share more commonalities than variabilities. Commonality refers to the multiple use of modules within the same product and between different products. It aims to reduce the extent of special-purpose modules, which generally increases internal variety and costs [Blecker and Abdelkafi, 2006].

1.2 Web Configurators

In order to facilitate the product customization task, companies provide their customers with online software tools called *configuration systems* (see Figure 1.1 for an example), also referred to as *product configurators*, *Web configurators*, *sales configurators*, *design systems*, or simply *configurators*. A configurator requires a *product configuration model*, i.e., the core that contains the list of predefined *configuration options* (also known as modules or components) to be assembled, their variations, and configuration rules (i.e., constraints) existing between options. It presents configuration options to customers and guides them through the product configuration process. To specify a product that matches the customer's individual requirements and preferences, the customer selects the options to be included in the product and the configurator guarantees that all the design and configuration rules which are expressed in the product configuration model are satisfied [Blecker and Abdelkafi, 2006; Franke and Piller, 2002; Hedin et al., 1998; Ong et al., 2006; Tseng and Piller, 2003; von Hippel and Katz, 2002; Xie et al., 2005]. In particular, the configurator verifies constraints, propagates customer decisions, and handles conflictual decisions. In other words, given a set of customer requirements and a logical description of the product family, the role of the configuration system is to find a valid and completely specified product instance along all of the alternatives that the generic structure describes [Sabin and Weigel, 1998].

Configurators are used in many B2B and B2C applications to personalize products and services. They are used in installation wizards and preference managers. They are also extensively used in *software product lines* (SPLs) where multiple information system variants are derived from a base of reusable artefacts according to the specific characteristics of the targeted customer or market segment [Gottschalk et al., 2009; Pohl et al., 2005; Rosa et al., 2008; Schäler et al., 2012].



Interior		
☐	**Flex Floor luggage compartment**	**€80.00**
Option Pack		
☐	**Seat Comfort Pack One** Driver's seat with six-way manual adjustment and four-way electrical lumbar adjustment	**€201.00**
☐	**Seat Comfort Pack Two** - Driver's and front passenger's seat with six-way manual adjustment and four-way electrical lumbar adjustment	**€267.00**
Safety / Security		
✓	**Emergency tyre inflation kit**	
☐	**Space-saver spare wheel**	**€146.00**
☐	**Remote control ultrasonic security alarm system**	**€361.00**

FIGURE 1.1: Opel Web configurator (`http://www.opel.ie/`, February 21 2014).

1.3 Problem Statement

In many cases, configurators have become the privileged channel for identifying customer needs. If a configurator could not guide customers to specify the right products or if customers feel overwhelmed by the configuration process, they may make suboptimal decisions or abort the configuration process [Rogoll and Piller, 2004] which leads companies to experience a loss of sales [Trentin et al., 2012]. As such, configurators are strategic components of companies' information systems and must meet stringent reliability, usability and evolvability requirements.

Configurators still need substantial improvements, especially the way by which configurators present options to customers [Blecker and Abdelkafi, 2006]. An empirical study carried out by Rogoll and Piller shows that existing configurators cannot fulfil optimal requirements from companies' and customers' perspectives [Rogoll and Piller, 2004]. For instance, some configurators do not show the customer a picture of the whole proposal and functionalities the system offers, do not provide explanations on how to navigate through the configuration process, etc. The authors concluded that the state of the design of configurators' front-ends is rather weak. Von Hippel also showed that existing configurators just enable customers to select products among alternatives but do not facilitate customer learning [von Hippel, 2001].

Despite the abundance of Web configurators, the state of the art *lacks* knowledge, guidelines, and tools for *efficiently engineering* Web configurators. Our long-term objective is to develop a set of methods, languages, and tools to systematically engineer Web configurators. However, to realize this vision, we first need to understand the intrinsic characteristics of Web configurators. Therefore, we set out to answer the first research question:

RQ1 *What is the current practice in engineering Web configurators?*

To answer this research question, we conducted a systematic empirical study of 111 configurators and highlighted patterns used in engineering Web configurators. This study revealed the absence of specific, dedicated, and rigorous methods in their engineering. For instance, 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 [Benavides et al., 2010; Hubaux et al., 2012; Janota, 2010].

Some of our industry partners face similar problems and are now trying to migrate their legacy Web configurators to more reliable, efficient, and maintainable solutions [Boucher et al., 2012a]. To decrease the cost of migration, we offer to systematically *re-engineer* these applications. Figure 1.2 presents our proposed re-engineering process. This encompasses three activities: (1) *reverse engineering* legacy Web configurators, (2) encoding the extracted data into dedicated formalisms, and (3) *forward engineering* new improved configurators. Reverse engineering a Web configurator is the process of extracting variability data (i.e., configuration options, their associated descriptive information, constraints, etc.) from the Web pages of the configurator, and then constructing a variability model, for instance, a *feature model*. Once the feature model of the configurator is built, it can be used in the forward-engineering process to generate a customized and easily maintainable user interface with an underlying reliable reasoning engine. In this PhD thesis, we study the reverse-engineering process. The study of the forward-engineering process is out of the scope of this thesis.

Our next goal is to reverse engineer variability models from Web configurators. We address two main research questions that are related to the reverse-engineering process.

RQ2 *What generic Web data extraction methods can we use to collect accurate variability data from the Web pages of a configurator?*

This research question is concerned with two challenging issues. First, the Web data extraction approach should be *generic* enough so that it can be used for collecting data from Web configurators coming from different industry sectors with different characteristics. Second, considering the fact that not all data presented in the pages of a configurator is configuration-specific, eliciting the *right* data from the *noisy* data is another major concern. Moreover, the extracted data must be named, meaning that each data item in an extracted data record is required to be assigned a meaningful label. **RQ2** addresses the problem of extracting *structured variability data* from a Web page.

RQ3 *How to support the extraction of the dynamic variability content?*

Web configurators are highly interactive and dynamic applications. As a reaction to the user's configuration actions (e.g., exploring the configuration space, making configuration-specific decisions, etc.), the configurator may add new configuration-specific content to the page, or may change the existing content. This research question addresses this runtime behaviour of Web configurators. We should answer sub-questions like:

- *How to simulate the users' configuration actions to automatically generate dynamic content?*
- *How to deduce variability data from the dynamically generated content?*

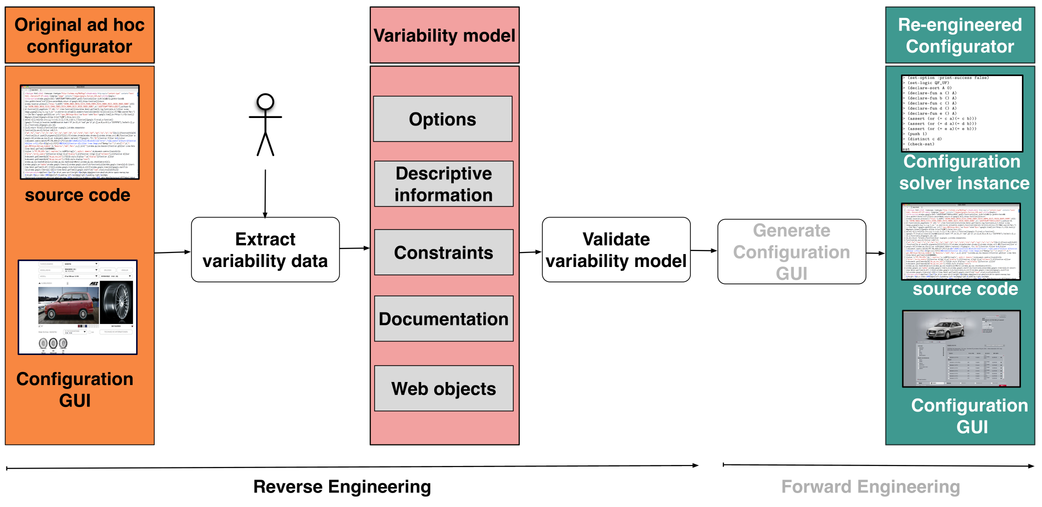


FIGURE 1.2: Re-engineering process [Boucher et al., 2012a].

1.4 Contributions

The main contributions of this thesis consist of:

C1 *A systematic study and understanding of Web configurators.* Although Web configurators have been studied from usability and visual aspects [Rogoll and Piller, 2004; Streichsbier et al., 2009; Trentin et al., 2012], their underlying concepts have not been investigated. We conduct a systematic and empirical survey of 111 Web configurators and aim at understanding how their underlying concepts are represented, managed, and implemented. We analyse the client-side source code of these configurators with semi-automated code inspection tools. We analyse the results along three essential dimensions: rendering configuration options, constraint handling, and configuration process support.

C2 *Identification and classification of patterns used in engineering Web configurators.* Based on empirical data, we identify and classify patterns used in engineering Web configurators. We then use these patterns to highlight the bad and good practices.

C3 *Development of an HTML-like language to extract structured variability data from Web pages.* We propose the notion of *variability data extraction* (vde) patterns, an HTML-like language to specify data to be extracted from the Web pages of a configurator. The *vde* patterns can be used to identify and manage the implicit templates (structure and layout of data) followed in Web development. A user uses a *vde* pattern to specify the structure of data objects of interest and data items to be extracted from these objects.

C4 *A source code pattern matching algorithm.* We propose an algorithm that given a *vde* pattern and a Web page, looks for code fragments (implementing data objects of interest) in the source code of the page that *structurally* match the pattern. It provides a two-step solution to find matching code fragments: (1) first *finding candidate code fragments* that may match the given pattern, and then (2) *traversing each candidate code fragment to find 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 syntactic tree representations. It uses a *bottom-up* tree traversal to find candidate code fragments and a mixture of both *depth-first* and *breadth-first* traversals to traverse each candidate code fragment.

C5 *An approach to (semi-)automatically and systematically extract dynamic variability content.* We present a solution to extract dynamic variability data. In particular, we introduce the notion of *dependency* between *vde* patterns, the main foundation on top of which our solution is developed. Our solution automates (1) the simulation of users' exploration and configuration actions to systematically generate new content, and then (2) the analysis of the new content to deduce the variability data.

C6 *A complete implementation of all the algorithms, approaches, and methods in an integrated tool.* We implemented a reverse-engineering tool that mainly consists of two collaborative components: *Web Wrapper* and *Web Crawler*. Web Wrapper seeks to find and extract variability data from a page given a set of *vde* patterns, and then transforms the extracted data into structured variability data represented in a feature model. Web Crawler automatically explores the configuration space (i.e., all objects representing variability data) and simulates some of the users' configuration actions. It systematically generates dynamic variability data which is then extracted by the Wrapper.

1.5 Roadmap

After this introductory chapter, the rest of the thesis is organized as follows.

Chapter 2 presents background information. It provides a general overview of variability modelling and briefly introduces TVL. This chapter also describes the nature of Web applications.

Chapter 3 reports on a systematic study of Web configurators (**RQ1**). It presents the diversity of representations for configuration options, different kinds of constraints are supported by Web configurators, and the way the configuration process is enforced by them (**C1**). This chapter also classifies *grouping* strategies used to categorize options in *semantic* constructs, patterns followed by the configurators for *decision propagation* and *consistency checking*, as well as patterns for designing the configuration process, its *activation* and *navigation*. The bad and good practices in designing Web configurators are also reported in this chapter (**C2**).

The empirical analysis of the configurators revealed reliability issues when handling constraints. These problems come from the configurators' lack of convincing support for consistency checking and decision propagation. Moreover, the investigation of client-side code implementation verifies, in part, that no systematic method (e.g., solver-based) is applied to implement reasoning operations. We also noticed that usability is rather weak in many cases (e.g., counter-intuitive representations, lack of guidance).

Chapter 4 reports a survey of existing approaches for *reverse engineering Web applications*, *Web data extraction*, and *synthesizing feature models*, three fields of study that can contribute to reverse engineer featured models from Web configurators. We found that none of these approaches tackle the extraction of variability data from Web configurators. 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).

Chapter 5 demonstrates our tool-supported and supervised reverse-engineering process for extracting variability data from Web pages. It outlines interactive and automatic activities required to produce a fully-fledged TVL model for a Web configurator (**RQ2** and **RQ3**).

Chapter 6 explains our solution to extract structured variability data from the Web pages of a configurator (**RQ2**). It introduces the notion of *variability data extraction* (vde) patterns (**C3**) using which a user manually marks and names variability data to be extracted. This chapter also describes the syntax of the patterns by giving examples and providing a context-free grammar.

Chapter 7 describes the data extraction procedure (**RQ2**) used by the Wrapper to find data objects of interest whose structure is specified in the given *vde* patterns. In particular, this chapter explains our proposed source code pattern matching algorithm (**C4**) implemented by the Wrapper (**C6**) to find matching code fragments.

Chapter 8 illustrates our solution to extract dynamic variability data (**RQ3**). It introduces the notion of *dependency* between *vde* patterns. This dependency provides a framework for the Wrapper and the Crawler to collaborate together to generate and extract dynamic variability data (**C5** and **C6**). In particular, they work together to trigger and extract cross-cutting constraints defined over options.

Chapter 9 presents the results of using the proposed techniques on a sample set of subject systems to evaluate our approach.

Chapter 10 concludes the thesis and highlights the future work.

1.6 Bibliographical Notes

The research presented in this PhD thesis, extends peer-reviewed publications of the author. We list below the relevant papers:

Journal Paper

- A. Hubaux, P. Heymans, P.-Y Schobbens, D. Deridder, and E. K. Abbasi. Supporting multiple perspectives in feature-based configuration. *Software and System Modeling (SoSyM)*, pages 641–663, 2013. Springer Berlin Heidelberg. (**Chapter 10**)

Conference Papers

- E. K. Abbasi, A. Hubaux, M. Acher, Q. Boucher, and P. Heymans. The Anatomy of a Sales Configurator: An Empirical Study of 111 Cases. *Advanced Information Systems Engineering*, volume 7908 of *Lecture Notes in Computer Science,* pages 162–177, 2013. Springer Berlin Heidelberg. (**Chapter 3**)
- E. K. Abbasi, M. Acher, P. Heymans, and A. Cleve. Reverse Engineering Web Configurators. *IEEE CSMR-WCRE 2014 Software Evolution Week*, Antwerp, Belgium, 2014. IEEE Computer Society. (**Chapters 6, 7, 8, 9**)
- E. K. Abbasi, A. Hubaux, and P. Heymans. A Toolset for Feature-based Configuration Workflows. In *Proceedings of the 15th International Software Product Line Conference (SPLC'11)*, pages 65–69, Munich, Germany, 2011. IEEE Computer Society. (**Chapter 10**)
- E. K. Abbasi, A. Hubaux, and P. Heymans. An interactive multi-perspective toolset for non-linear product configuration processes (tool demo). In *Proceedings of the 15th International Software Product Line Conference (SPLC'11), Volume 2*, pages 50:1–50:1, Munich, Germany, 2011. IEEE Computer Society. (**Chapter 10**)

Workshop Paper

- Q. Boucher, E. K. Abbasi, A. Hubaux, G. Perrouin, M. Acher, and P. Heymans. Towards More Reliable Configurators: A Re-engineering Perspective. In *Proceedings of the International Workshop on Product LinE Approaches in Software Engineering (PLEASE'12), co-located with ICSE'12*, pages 29–32, Zurich, Switzerland, 2012. IEEE Computer Society. (**Chapter 1**)

Doctoral Symposium Paper

- E. K. Abbasi and P. Heymans. Reverse Engineering Web Sales Configurators. In *Proceedings of the 29th IEEE International Conference on Software Maintenance (ICSM'13)*, pages 586–589, Eindhoven, The Netherlands, 2013. IEEE Computer Society. (**Chapter 5**)

Chapter 2

Background: Variability Modelling and Web Applications

In this chapter, we provide background information. We first explain variability modelling that is closely associated with product lines and Web configurators (Section 2.1). In particular, we briefly present the syntax of TVL, a variability modelling language we use in our reverse engineering process to represent the extracted variability data. We then provide some basic definitions for Web applications (Section 2.2).

2.1 Variability Modelling

2.1.1 Product lines

A *product line* is a set of products that together address a particular market segment or fulfil a particular mission [SEI, 2014]. Products in a product line are produced from *a common set of core assets* in a prescribed way [Clements and Northrop, 2002], therefore, all those products share a significant amount of *commonality* and differ in their specific configuration of *variability* [Stoiber, 2012]. A commonality thereby is a characteristic of all products of a product line, and a variability, in contrast, is a varying characteristic whose value is changed from one product to another. In other words, variabilities can be seen as parameters that support a more concise identification of products in a product line [Becker et al., 2002].

An efficient and popular approach to model variants in a product line is *variability modelling*. A variability model, in fact, captures commonalities and variabilities in a product line and can be used to understand, create, and manage the product line. It also supports product derivation [Czarnecki et al., 2012] which takes place during a configuration process [Benavides et al., 2010].

2.1.2 Feature model

Feature models (FM) are the de-facto standard to express variability in software product lines (SPL). This technique was first introduced by Kang *et al.* [Kang et al., 1990] to capture commonality and variability in a software family as part of the *Feature-Oriented Domain Analysis* (*FODA*) method. FODA aims to identify prominent or distinctive *features* of software systems in a domain. These features are user-visible aspects or characteristics of the domain. They define both common aspects of the domain as well as differences between related systems in the domain. A feature, in fact, is the attribute of a system that directly affects end-users. The end-users have to make decisions regarding the availability of features in the system. Several other views of what a feature is can be found in the literature [Apel et al., 2008; Batory et al., 2006; Chen et al., 2005; Classen et al., 2008].

A FM represents the features of a family of systems in a domain and *relationships* between them in a tree structure. The structural relationship *"consists of"* represents a logical grouping of features. *Alternative* or *optional* features of each grouping must be indicated in the FM.

Kang *et al.* consider four different components for a FM:

- **Feature diagram:** A graphical hierarchy of features
- **Composition rules:** Mutual dependency (Requires) and mutual exclusion (Mutex-with) relationships
- **Issues and decisions:** Record of trade-offs, rationales, and justifications
- **System feature catalogue:** Record of features and feature values of actual existing systems

Composition rules, aka *cross-cutting constraints*, define the semantics existing between features that are not expressed in the feature diagram.

As an example, Figure 2.1 illustrates how a FM can be used to build software for mobile phones. The software of a phone is determined by the features that it provides. The root feature (i.e., "Mobile phone") identifies the SPL. Every mobile phone system must provide support for calls, display information, so "Calls" and "Screen" are *mandatory* features.

Furthermore, "GPS" and "Media" are *optional* features and so may not be included in all products of the SPL. The software for mobile phones may include support for one of "Basic", "Colour" or "High resolution" screens. It indicates that there is an *alternative* relationship between the set of child features of the "Screen" feature. Additionally, whenever "Media" is selected, "Camera", "MP3" player or both can be selected. It denotes the *or*-relationship between the set of child features of "Media".

In Figure 2.1, two composition rules are defined. The *requires* constraint indicates that if the "Camera" feature is selected to be included in a mobile phone system, it must also include support for a high resolution screen. The *excludes* constraint tells that "GPS" and "Basic" are incompatible features.

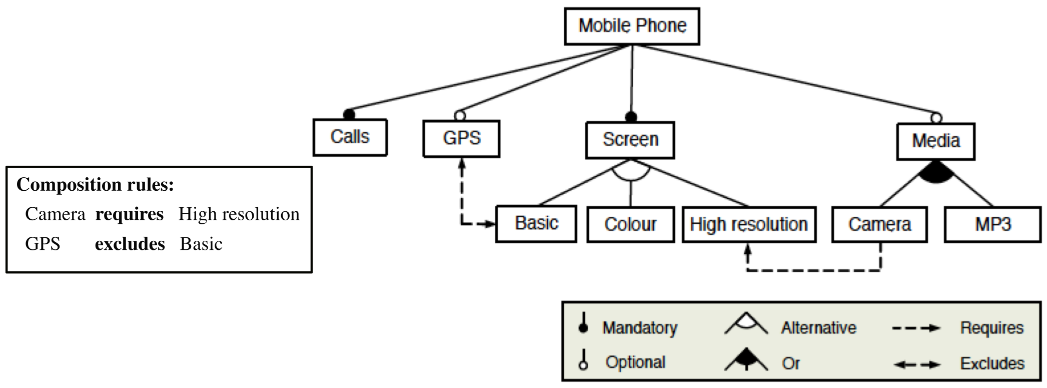


FIGURE 2.1: A FM for mobile phone systems ([Benavides et al., 2010]).

After the initial proposal by Kang *et al.*, several FM extensions have been proposed [Czarnecki and Eisenecker, 2005; Griss et al., 1998; Michel et al., 2011a]. Several feature modelling languages have been also designed and supported by editing, debugging, analysis, and configuration tools [Acher et al., 2013b; Bak et al., 2011; Benavides et al., 2007; Beuche, 2012; Botterweck and Schneeweiss, 2009; Classen et al., 2011a; Mendonca et al., 2009; Thum et al., 2014]. In this PhD thesis, we rely on the *Textual Variability Language (TVL)* to represent FMs. In Section 2.1.3, we briefly introduce TVL.

The semantics of a FM is the set of valid products that can be derived from the FM [Schobbens et al., 2006]. Product derivation is performed in a configuration process during which features to be included in the product are selected. A product is valid if it does not include any contradiction [Benavides et al., 2010]). For instance, consider the products presented below and the FM of Figure 2.1. *P1* and *P2* are valid products. Product *P3* is not valid since it does not include the mandatory feature "Calls". Product *P4* includes the incompatible features "Basic" and "GPS", and therefore is not a valid product.

```
P1 = {Mobile phone, Calls, Screen, Colour}
P2 = {Mobile phone, Calls, Screen, Colour, Media, MP3}
P3 = {Mobile phone, Screen, Basic}
P4 = {Mobile phone, Calls, Screen, Basic, GPS}
```

2.1.3 TVL

TVL [Classen et al., 2010, 2011a] is a text-based feature modelling language designed to address expressiveness, conciseness, and adequate tool-support shortcomings that exist for graphical FMs. It provides a human-readable and rich C-like syntax for easy and natural modelling with a formal semantics for powerful automation. Moreover, TVL is a lightweight and scalable language that offers several mechanisms for structuring feature models.

We use an excerpt of the configuration environment of the Dell configurator shown in Figure 2.2 and its TVL model (generated by our reverse engineering tool) visible in Figure 2.3 to illustrate the syntax of TVL. Configuration options presented in Web pages of a configurator are represented as features in a FM.

Feature declaration and hierarchy. The TVL language provides a C-like syntax to structure a feature model. Curly brackets are used to delimit blocks and semicolons to terminate statements. The feature model in TVL has a tree structure but, sometimes, a directed acyclic graph structure in which a feature (called a **shared** feature) can have several parents. The root feature of a feature model in TVL is declared by putting the **root** keyword before the feature name.

TVL has three predefined operators to define the decomposition type (defined with the **group** keyword): **allOf** for and-decompositions (line 2), **someOf** for or-decompositions (lines 4, 9, and 30), and **oneOf** for xor-decompositions (line 11). A cardinality-based decomposition can be specified too: **group [i..j]**, where i and j are the lower and upper bounds of the cardinality and j can be the asterisk character (*) as well. A decomposition type is followed by a comma-separated list of features, enclosed in curly brackets. Each feature can declare its own child features, attributes, and constraints with nested curly brackets.

In our example, the root feature, "Monitors_Docking_Solutions" (line 1), is decomposed into two features by an and-decomposition (line 2): "Monitors" (line 3) and "Docking_Solutions" (line 29). Furthermore, the "Dell_Wireless_Speaker_System_AC411" feature is optional (line 22). An optional feature is identified by the **opt** in front of the feature name.

Attributes. Descriptive information associated to an option in a Web configurator are modelled as attributes of the corresponding feature in a feature model. Attributes are declared by defining their *type* and *name* inside the block of their owner feature. TVL supports five different attribute types: integer (**int**), real (**real**), boolean (**bool**), enumeration (**enum**), and string (**string**). An attribute can be optionally assigned a value. To set the value of an attribute the **is** keyword is used. In our example, some features have a "price" attribute, e.g., "Dell_20_Touch_Monitor_E2014T", "Dell_Wireless_Speaker_System_AC411", etc.

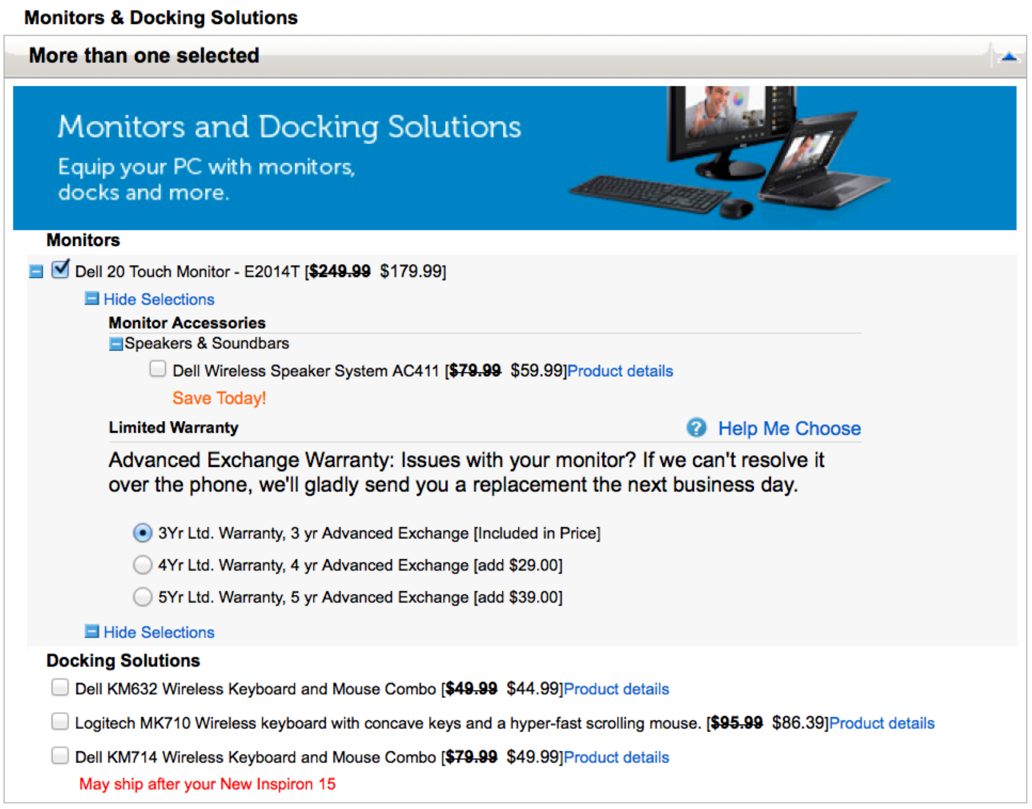
Constraints. In TVL, constraints are boolean expressions which are attached to features and can be added to the body of a feature definition. In our example, there is a *requires* constraint attached to "Dell_20_Touch_Monitor_E2014T" (lines 7 and 8): the selection of "Dell_Wireless_Speaker_System_AC411" implies the selection of "A_5Yr_Ltd_Warranty_5_yr_Advanced_Exchange". "–>" denotes implication (line 7).

In TVL, identifiers such as types, features, and feature attributes have to start with a character and can contain numbers as well as underscores. Identifiers in TVL are case sensitive. Feature names have to start with an uppercase letter. In Figure 2.2, feature names contain spaces and therefore in the generated TVL model each space is replaced with an underscore. Also, some feature names start with numbers (e.g., "3Yr Ltd Warranty, 3 yr Advanced Exchange"). We added "A_" to the beginning of the name of such features to make them valid feature names in TVL. We also removed invalid characters from feature names, e.g., comma (,) from "3Yr Ltd Warranty, 3 yr Advanced Exchange".

2.2 Web Applications

Conallen [Conallen, 1999] defined a *Web application* as a Web system (Web server, network, HTTP, browser) where user input (navigation and data input) affects the state of the business. This definition attempts to establish that a Web application is a software system with business state, and that its "front end" is in large part delivered via a Web system. A Web application is an extension of a *Website*. A Website is a collection of hypertextual documents, located on a Web server and accessible by an Internet user. It provides its users the opportunity to read information through the *World Wide Web* (WWW) window, but not to modify the status of the system [Tramontana, 2005].

Three main classes of Web applications are: (1) static applications, i.e., Websites, implemented in HTML and with no user interaction, (2) those providing client-side interaction with Dynamic HTML pages that can handle user events, and (3) applications containing dynamic content created "on-the-fly" using technologies such as Java Server Pages


Monitors & Docking Solutions

More than one selected

Monitors and Docking Solutions
Equip your PC with monitors, docks and more.

Monitors

✓ Dell 20 Touch Monitor - E2014T [~~$249.99~~ $179.99]

Hide Selections

Monitor Accessories

Speakers & Soundbars

Dell Wireless Speaker System AC411 [~~$79.99~~ $59.99]Product details

Save Today!

Limited Warranty — Help Me Choose

Advanced Exchange Warranty: Issues with your monitor? If we can't resolve it over the phone, we'll gladly send you a replacement the next business day.

● 3Yr Ltd. Warranty, 3 yr Advanced Exchange [Included in Price]
○ 4Yr Ltd. Warranty, 4 yr Advanced Exchange [add $29.00]
○ 5Yr Ltd. Warranty, 5 yr Advanced Exchange [add $39.00]

Hide Selections

Docking Solutions

Dell KM632 Wireless Keyboard and Mouse Combo [~~$49.99~~ $44.99]Product details

Logitech MK710 Wireless keyboard with concave keys and a hyper-fast scrolling mouse. [~~$95.99~~ $86.39]Product details

Dell KM714 Wireless Keyboard and Mouse Combo [~~$79.99~~ $49.99]Product details

May ship after your New Inspiron 15

FIGURE 2.2: Dell Web configurator (`http://www.dell.com//`, February 21 2014).

(JSP), Java Servlets, Active Server Pages (ASP), PHP, XML, etc. [Tilley and Huang, 2001].

The architecture of a Web application is a *client-server* model, a distributed structure where servers provide services and clients request services. Communication between the client and the server is established over a network using the *HTTP* protocol[1]. Figure 2.4 presents a general architecture for Web applications. The user uses a Web Browser to access information provided by a Web Server. She may need to request a new resource or service from the server. In this case, the user generates a *uniform resource locator* (URL) request, which is then translated in a HTTP request and sent to the Web Server. The Web Server decodes the request and retrieves the server page corresponding to the requested URL. The server page is sent to the Application Server which interprets the code of the server page and generates as output a Built Client Page. The generated client page is sent as response to the client. During the interpretation of a server page, the Application Server can communicate with a Database server through Database Interface objects, or it can request services to a third party, such as a Web Service. The Web Sever sends the Build Client Page to the client browser, packed in an HTTP response message. The Web Browser comprehends some active plug-ins that are able to interpret

[1]`http://www.w3.org/Protocols/`

```
root Monitors_Docking_Solutions {
    group allOf {
        Monitors {
            group someOf {
                Dell_20_Touch_Monitor_E2014T {
                    real price is 179.99 ;
                    Dell_Wireless_Speaker_System_AC411 ->
                                    Limited_Warranty.A_5Yr_Ltd_Warranty_5_yr_Advanced_Exchange;
                    group someOf {
                        Limited_Warranty {
                            group oneOf {
                                A_3Yr_Ltd_Warranty_3_yr_Advanced_Exchange {
                                },
                                A_4Yr_Ltd_Warranty_4_yr_Advanced_Exchange {
                                    real price is 29.00 ;
                                },
                                A_5Yr_Ltd_Warranty_5_yr_Advanced_Exchange {
                                    real price is 39.00 ;
                                }
                            }
                        },
                        opt Dell_Wireless_Speaker_System_AC411 {
                            real price is 59.99 ;
                        }
                    }
                }
            }
        },
        Docking_Solutions {
            group someOf {
                Dell_KM632_Wireless_Keyboard_and_Mouse_Combo {
                    real price is 44.99 ;
                },
                Logitech_MK710_Wireless_keyboard_with_concave_keys_and_a_hyper_fast_scrolling_mouse {
                    real price is 86.39 ;
                },
                Dell_KM714_Wireless_Keyboard_and_Mouse_Combo {
                    real price is 49.99 ;
                }
            }
        }
    }
}
```

FIGURE 2.3: TVL model for the Dell configurator shown in Figure 2.2.

code written using a client scripting language, such as JavaScript code. If the Built Client Page has scripting code, then the result of its execution is shown to the user, else the Web Browser displays directly the result HTML rendering [Tramontana, 2005]. A Web application is logically broken into three *presentation, application,* and *data tiers*:

- The *presentation tier* is about the Web browser and is responsible for the user interface.
- The *application logic tier* is behind the presentation tier and controls the application's functionality.
- The *data tier* is responsible to store and retrieve data.

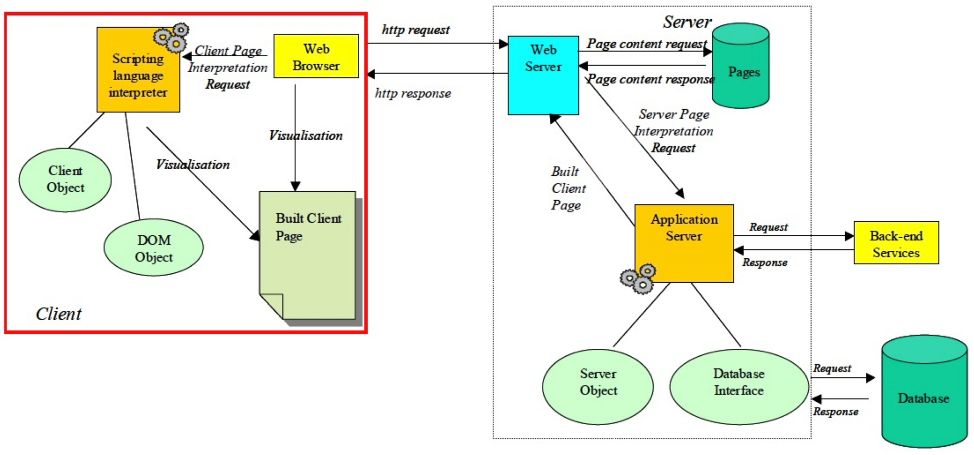


FIGURE 2.4: Architecture for Web applications [Tramontana, 2005].

2.2.1 Web technologies

This section briefly presents some technologies used in developing Web applications.

HTML. The hypertextual content of a client page (also called HTML document or Web page) and other information to render it in a Web browser is created using the *HyperText Markup Language* (HTML) which is a tagged language. The browser reads a client page, uses its tag to interpret the content of the page, and displays a visible page.

Document Object Model (DOM). HTML documents are presented using the *Document Object Model*. The DOM defines the logical structure of the document and the way the document is accessed and manipulated [DOM, 2014]. It presents an HTML document as a tree-structure.

Client-side scripting languages. These languages are used to write client script code in an HTML document. The client script code is used to interact with the user, control the browser, change the document's content at runtime, etc. It provides dynamic behaviour to client pages. The most common client-side scripting languages are JavaScript and VBScript.

Server-side scripting language. To write programs on the server side of a Web application and dynamically generate client pages, *server-side scripting languages* are used. There are a number of server-side scripting languages. Examples are: ASP, PHP, JSP, Python, Perl CGI, etc.

Asynchronous JavaScript and XML (AJAX). AJAX is a web development technique to asynchronously exchange data between the client and the server sides and update parts of a Web page without recreating and reloading the Web page. To exchange data, *JavaScript Object Notation*[2] (JSON) is often used. JSON is a text-based, language-independent, human-readable, and lightweight data-interchange format. It is easy for machines to parse and generate.

2.3 Chapter Summary

In this chapter, we explained product lines and presented that a variability model, e.g., a feature model, can be used to express commonality and variability in a product line. We then described the main components of a feature model and introduced TVL, a textual language to represent feature models.

We also provided an explanation of Web applications, their main classes, and the architecture of a Web application. We then presented some technologies, for instance, HTML, Document Object Model, and scripting languages, used in Web development.

[2] http://www.json.org/

Chapter 3

The Anatomy of a Web Configurator

Our main objective is to develop a set of methods, guidelines, languages, and tools to systematically re-engineer legacy Web configurators. To start this journey, we need to understand how Web configurators are currently designed and implemented. This means investigating the intrinsic characteristics of the configurators ranging from the GUI itself over constraint expressiveness to the reasoning procedures. For instance, different graphical representations of options (e.g., check boxes and radio buttons), constraint management techniques (e.g., by notifying the user), and configuration processes (e.g., the process can be single-step or multi-step) exist.

This chapter reports on a systematic and empirical study of 111 Web configurators we conducted to understand Web configurators [Abbasi et al., 2013]. We start with an introduction to Web configurators by giving an example Web configurator (Section 3.1). We then present the three research questions answered in this study, the research methodology, and the data extraction process to answer these questions (Section 3.2). We analyse the client-side code of the chosen configurators with semi-automated code inspection tools. We present the general observations emerged from this study (Section 3.3). We classify and analyse the results along three dimensions: *configuration options*, *constraints*, and *configuration process* (Section 3.4). For each dimension, we present quantitative empirical results and report on good and bad practices we observed (Section 3.5). We also describe the reverse-engineering issues we faced (Section 3.6) and the threats to validity (Section 3.7). We finally present the related work (Section 3.8).

3.1 Introduction

Despite similar goals, Web configurators are unique and vary significantly: they each have their own characteristics, spanning visual aspects (GUI) elements to constraint management. The Web configurator of Audi appearing in Figure 3.1 is thus one example out of hundreds existing configurators [Cyledge, 2014]. It displays the *configuration process* (Ⓐ) constituted of a sequence of *configuration steps* (e.g.,"1. Model" is followed by "2. Engine" – Ⓑ). Users follow the steps to complete the configuration of a *product* (a car in this example).

Each configuration step includes a subset of *configuration options* which are presented through specific widgets (radio buttons and check boxes – Ⓒ and Ⓓ, respectively). Users select options to be included in the product. Additionally, within a step, options are organized in different *groups* (e.g., "Exterior") and sub-groups (e.g., "Windows", "Mirrors". . .).

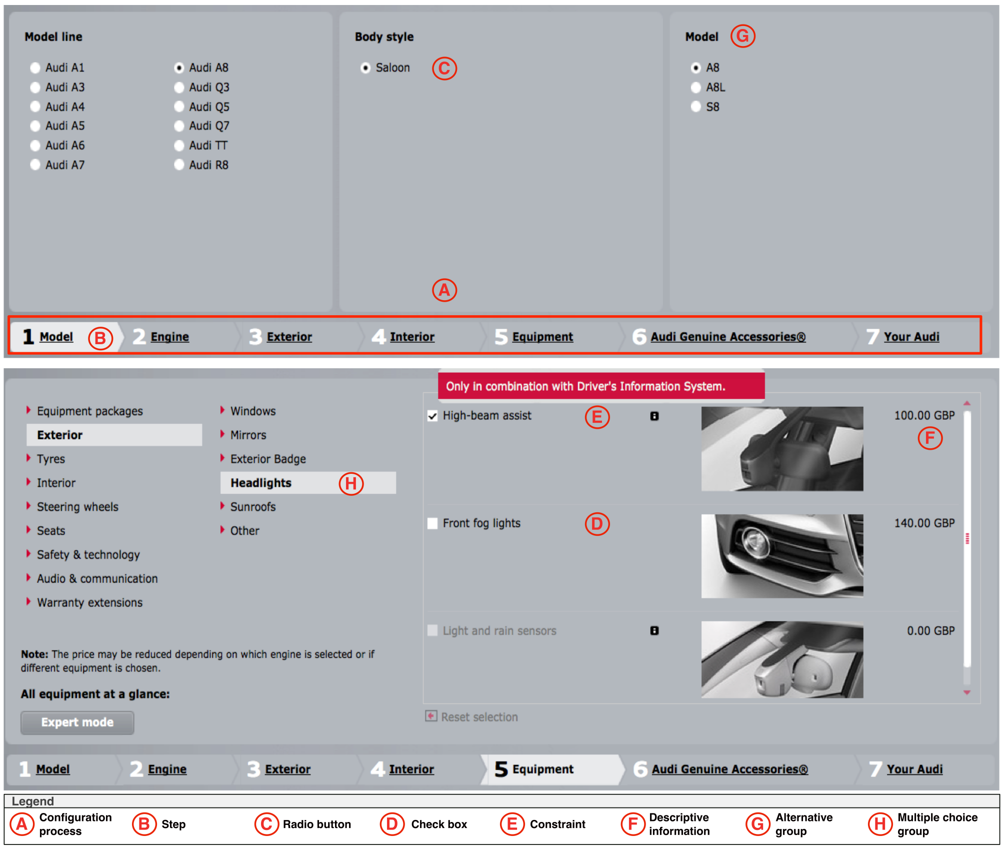
Options can be in different *configuration states* such as `selected` (e.g., "High-beam assist" is flagged with ✓), `undecided` (e.g., "Front fog lights"), or `unavailable` (e.g., "Light and rain sensors" is greyed out). A configurator can also implement *constraints* which determine valid combination of options (Ⓔ). 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.

In a Web configurator, a set of *reasoning procedures* control the configuration process. They verify constraints between options, propagate user decisions, and handle conflictual decisions [Rogoll and Piller, 2004; Streichsbier et al., 2009]. For instance, when an option is given a new value and one or more constraints apply, the reasoning procedure automatically propagates the required changes to all the impacted options and alters their configuration states.

Descriptive information (Ⓕ) is sometimes associated to an option (e.g., its price).

3.2 Problem Statement and Method

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



Only in combination with Driver's Information System.

Note: The price may be reduced depending on which engine is selected or if different equipment is chosen.

All equipment at a glance:

Legend

A Configuration process	B Step	C Radio button	D Check box	E Constraint	F Descriptive information	G Alternative group	H Multiple choice group

FIGURE 3.1: Audi Web configurator (`http://configurator.audi.co.uk/`, August 7 2013).

and server-side analyses. In this empirical study, we set out to answer three research questions:

RQ1 *How are configuration options visually represented and what are their semantics?*

By nature, 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, descriptive information, cloning, and grouping).

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

RQ3 *How is the configuration process enforced by the configurators?* The configuration process is the interactive activity during which users indicate the options to be included and excluded in the final product. It can, for instance, either be *single-step* (all the available options are presented together to the user) or *multi-step* (the process is divided into several steps, each containing a subset of options). Another criteria is navigation flexibility.

3.2.1 Configurator selection

To collect a representative sample of Web configurators, we used Cyledge's configurator database [Cyledge, 2014], which contains 800+ entries from a wide variety of domains. The selection process we followed to narrow down the 800+ configurators to 111 is shown in Figure 3.2.

Starting from the 800+ configurators (❶), the first step of our configurator selection process consisted in filtering out non-English configurators (❷). For simplicity, we only kept configurators registered in one of these countries: *Australia*, *Britain*, *Canada*, *Ireland*, *New Zealand*, and *USA*. This returned 388 configurators and discarded four industry sectors (❸).

Secondly, we excluded 26 configurators that are no longer available using a dedicated tool, *Jericho*[1]. Jericho is an HTML parser written in Java. A special function in this library takes as input the URL of a Website and returns its DOM[2] if the Website is available, otherwise returns an error. We considered a site unavailable either when it is not online anymore or requires credentials we do not have (❹).

Thirdly, we randomly selected 25% of the configurators in each sector (❻). We then checked each selected configurator with *Firebug*[3] to ensure that configuration options, constraints, and constraint handling procedures do not use Flash (❼). Firebug is a Firefox plugin used to monitor, modify, and debug CSS, HTML, and JavaScript. We excluded configurators using Flash because the Firebug extension we implemented (see next section) does not support that technology. We also excluded "false configurators". By this we mean 3D design Websites that allow to build physical objects by piecing graphical elements together, sites that just allow to fill simple forms with personal information, and sites that only describe products in natural language. The end result is a sample set of 93 configurators from 21 industry sectors (❽).

Finally, we added 18 configurators (❾) that we already knew for having used them in preliminary stages of this study. We used them to become familiar with Web configurators and test/improve our reverse-engineering tools, as discussed below. This raised the total number of Web configurators to 111 (❿). Figure 3.3 shows the distribution of the selected configurators by industry.

[1]http://jericho.htmlparser.net/docs/index.html
[2]Document Object Model: a standard representation of the objects in an HTML page.
[3]http://getfirebug.com/

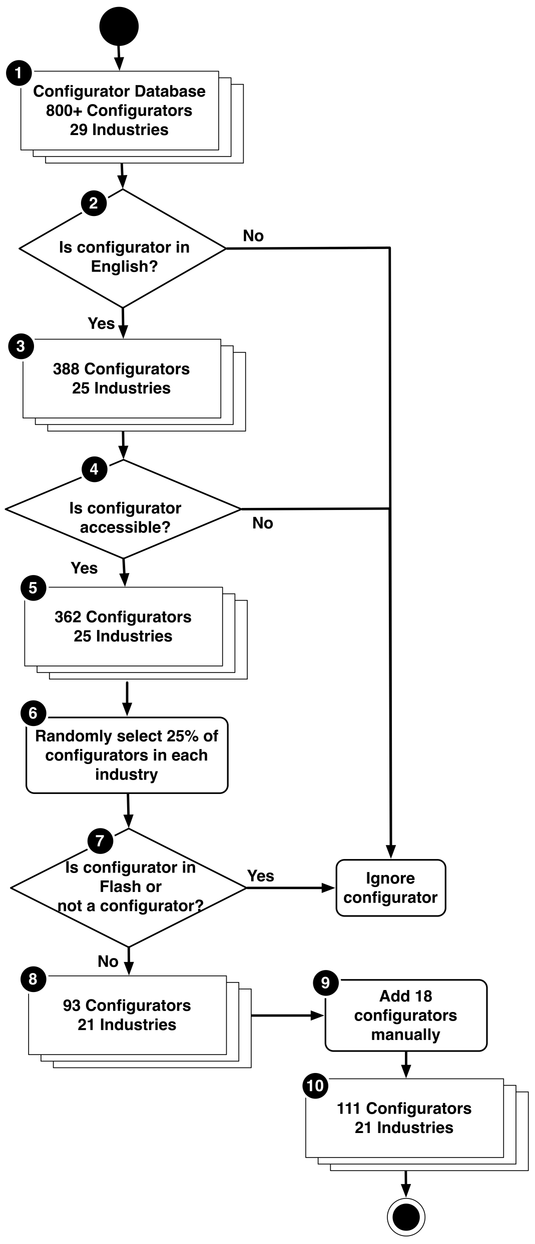


FIGURE 3.2: Configurator selection process.

3.2.2 Data extraction process

We used two complementary methods for studying the chosen Web configurators and gathering data on their behaviour and structure: *application analysis from the source code inspection* and *application analysis from the execution of the application.* To support these analyses, we developed a Firebug extension (Figure 3.4 – 3 KLOC, 1 person-month) that implements (*a*) a semi-automated and supervised data extraction approach, (*b*) support for advanced searches, and (*c*) DOM traversing.

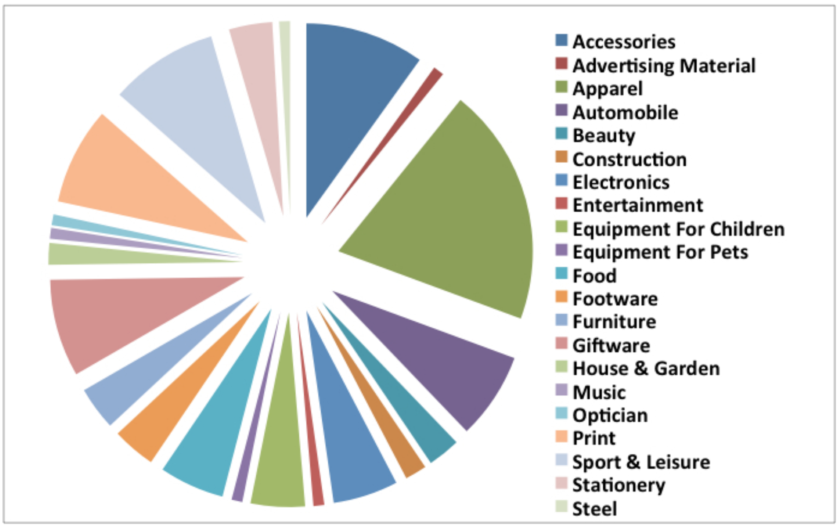


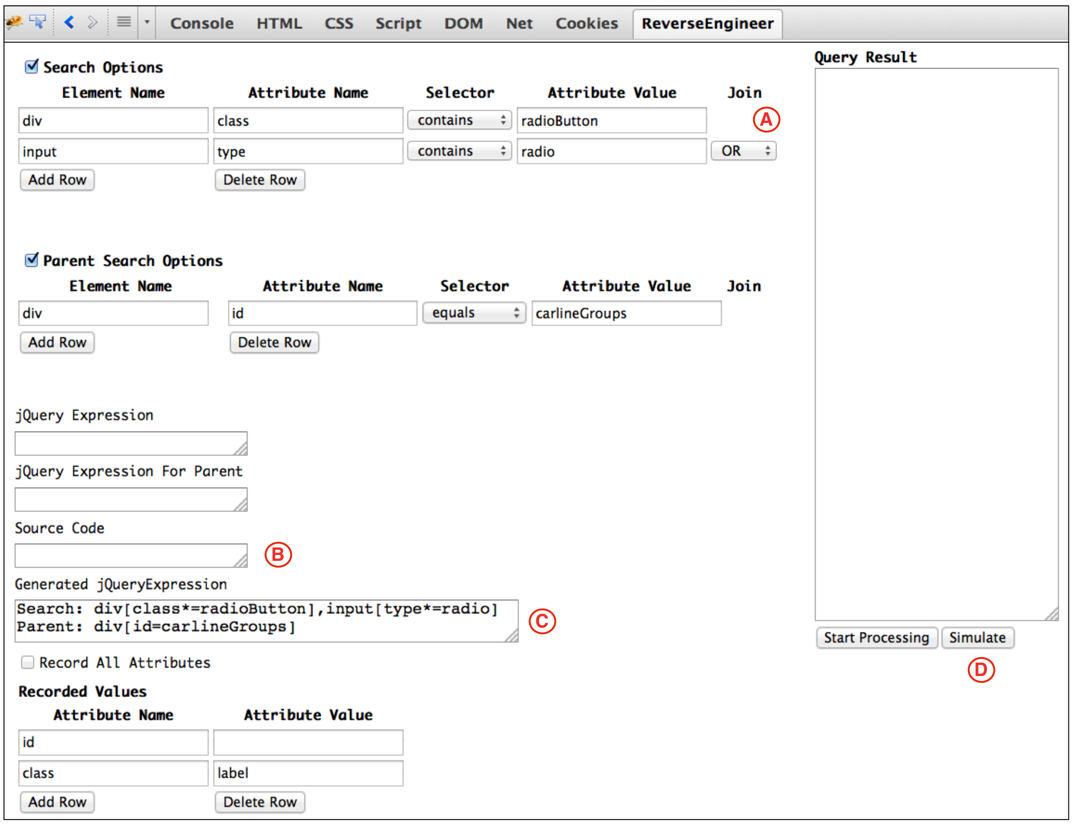
Figure 3.3: Distribution of selected configurators by industry.

To answer RQ1, we need to extract the types of widgets used to represent options. To that end, our extension offers a *search engine* able to search a given code pattern. Our approach relies on a training session during which we inspect the source code of the Web page to identify which code patterns (templates) are used to implement configuration options and their graphical widgets. These patterns vary from simple (e.g., *tag[attribute:value]*) to complex cases (e.g., a sequence of HTML tags). We then feed these patterns to the search engine (Ⓐ and Ⓑ) to extract all options. It uses jQuery[4] expressions (Ⓒ) and a pattern matching algorithm to search and find matching elements, extract an option name and its widget type.

To answer RQ2, we implemented a *simulator* to simulate the users' configuration actions (Ⓓ). Practically, the simulator selects/deselects each option and triggers constraints (if any). When a constraint is triggered, the reasoning procedures are fired to handle it. By inspection of the behaviour of the application under test, we identify and document strategies used for decision propagation and consistency checking. We also sometimes have to manually run (vs. automatically running by the simulator) Web configurators to analyse their behaviour and inspect the GUI of the configurators to gather some other data that is required to answer RQ2.

To answer RQ3, we inspected the GUI of each configurator to see how the configuration process is specified. In some cases, we run the application (either manually or by the simulator) and use it to configure products to find out how the configuration process is managed.

[4]http://jquery.com/



☑ Search Options

Element Name	Attribute Name	Selector	Attribute Value	Join
div	class	contains	radioButton	
input	type	contains	radio	OR

☑ Parent Search Options

Element Name	Attribute Name	Selector	Attribute Value	Join
div	id	equals	carlineGroups	

jQuery Expression

jQuery Expression For Parent

Source Code

Generated jQueryExpression

```
Search: div[class*=radioButton],input[type*=radio]
Parent: div[id=carlineGroups]
```

☐ Record All Attributes

Recorded Values

Attribute Name	Attribute Value
id	
class	label

Query Result

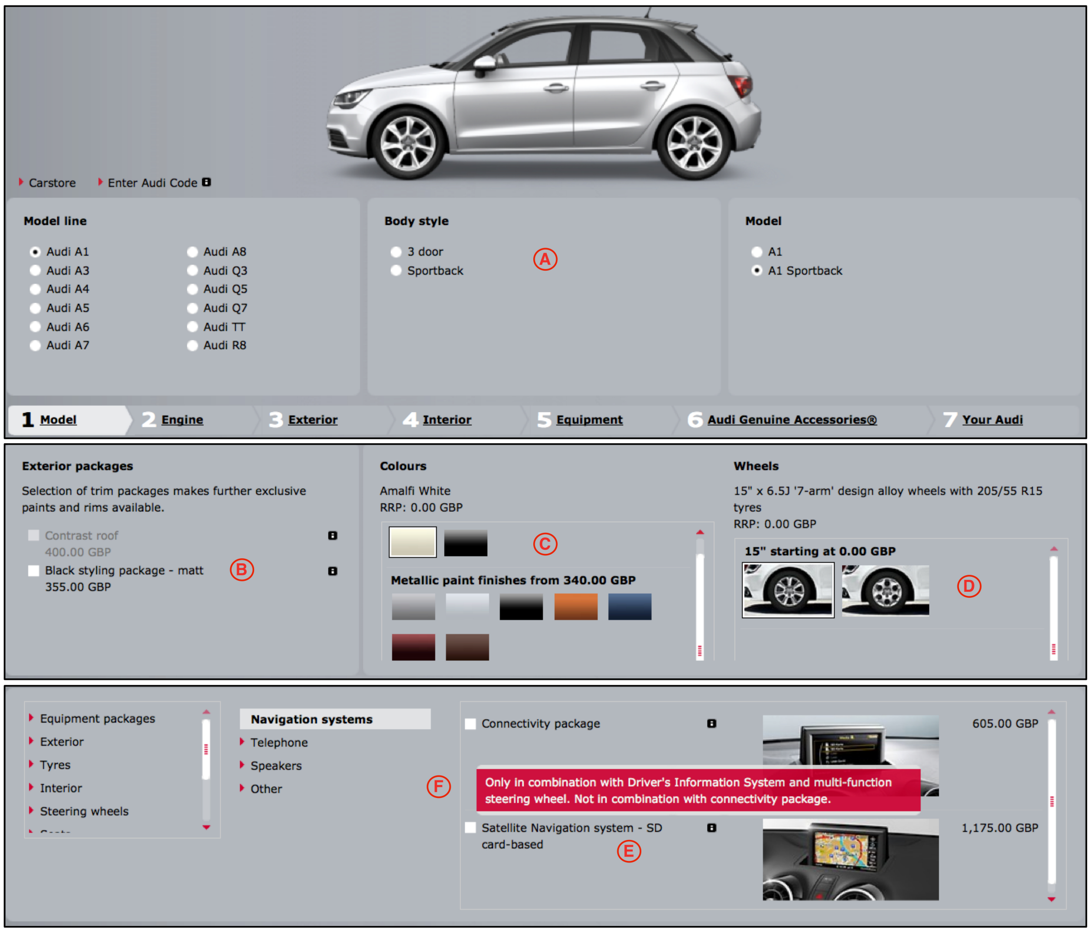
FIGURE 3.4: The Firebug data extraction extension.

3.3 General Observations

The client side of a configurator offers a highly interactive configuration environment for users to specify a product. Although Web configurators are usually developed in an unspecific way, i.e., like any other Web application, they have specific characteristics and are different from other classes of Web applications in terms of their visual aspects, data presentation, and business logic. In this section, we provide an insight into the main characteristics of the client side of Web configurators.

Variations in presentation and implementation of variability data. As it is presented in Section 3.4, Web configurators use a variety of Web objects (e.g., layouts and widgets) to visually represent configuration-specific objects (e.g., configuration steps, options, constraint-handling windows, etc.). Moreover, each configurator uses its specific Web objects. For instance, to implement alternative groups, one configurator may use radio buttons while another may propose single-selection list boxes. Figure 3.5 shows configuration steps that are visually presented in the GUI with navigational tabs, and options represented using radio buttons (Ⓐ), check boxes (Ⓑ), colours (Ⓒ), images (Ⓓ), and image-check box combinations (Ⓔ). In practice, there are as different structures and formatting features as configurators to implement these objects in the source code (*variations in implementation*).

Complex data objects. The data presented in the Web pages of a configurator can have different structures. It can be a single slot data item (Figure 3.5 – Ⓐ), a flat data record containing a block of related data items (Figure 3.5 – Ⓔ, an option name, its price, its image, and constraints), a complex data record with *multi-valued* [Chang et al., 2006] data items (Figure 3.6 – in the "Climate Pack" option, "Automatic lights and wipers" and "Dual zone climate control" are multi-valued data items), etc. A data object may have no attached textual explanation presented in the GUI, though. For instance, in Figure 3.5, the options included in the "Colours" group (Ⓒ) are presented using images. The data item to be extracted from each of these options is located in the tag attribute, i.e., the value of the `src` attribute of the corresponding `img` HTML element. Also note that a data item may be *shared* between several data objects. An example is the textual description "15" × 6.5J '7-arm' design alloy wheels with 205/55 R15 tyres" for the options contained in the "Wheels" group in Figure 3.5 (Ⓓ).



Model line
- [✓ Audi A1]
- Audi A3
- Audi A4
- Audi A5
- Audi A6
- Audi A7
- Audi A8
- Audi Q3
- Audi Q5
- Audi Q7
- Audi TT
- Audi R8

Body style
- 3 door
- Sportback

Model
- A1
- [✓ A1 Sportback]

Exterior packages

Selection of trim packages makes further exclusive paints and rims available.

- Contrast roof 400.00 GBP
- Black styling package - matt 355.00 GBP

Colours

Amalfi White
RRP: 0.00 GBP

Metallic paint finishes from 340.00 GBP

Wheels

15" x 6.5J '7-arm' design alloy wheels with 205/55 R15 tyres
RRP: 0.00 GBP

15" starting at 0.00 GBP

- Equipment packages
- Exterior
- Tyres
- Interior
- Steering wheels

Navigation systems
- Telephone
- Speakers
- Other

Connectivity package 605.00 GBP

Only in combination with Driver's Information System and multi-function steering wheel. Not in combination with connectivity package.

Satellite Navigation system - SD card-based 1,175.00 GBP

FIGURE 3.5: Presentation of configuration-specific objects (`http://configurator.audi.co.uk/`, February 22 2014).

Template-generated Web pages. Our analysis of the client-side source code of Web configurators shows that pages representing variability data are usually generated from a number of *templates*. A template is a code fragment that specifies the structure and layout of data to be visually presented in a page. In a template, text elements and tag attributes are data slots filled by data items when generating the page. Each configurator uses its specific templates to generate pages. Figure 3.6 depicts an excerpt of the configuration environment of a configurator as well as the two code fragments of the last two options, " Renault i.d. Metallic Paint" and "Climate Pack" (lines 1-7 and 8-19, respectively). The two code fragments are structurally rather similar, meaning that they are likely generated from a same template. Note that the second code fragment (lines 8-19) contains additional code lines (12-15).

Heterogeneous Web pages. A Web page in a configurator may contain various kinds of data objects with different structures, meaning that the page may be generated from several templates. For instance, we can identify two completely different templates for the options included in the "Exterior" step in Figure 3.5: a template that is used to generate options represented using check boxes (Ⓑ) and another template for image options (Ⓒ and Ⓓ).

Dynamic Web pages. When a Web configurator is executing and the user is making decisions, new content may be automatically created and added to the page, and existing content may be removed or changed. For instance, the "Model" step in Figure 3.5 contains three groups, namely "Model line", "Body style", and "Model". There are underlying constraints between options included in these groups. Consequently, the selection of an option from "Model line" loads its consistent options in "Body style", and in turn, the selection of an option from "Body style" loads its consistent options in "Model".

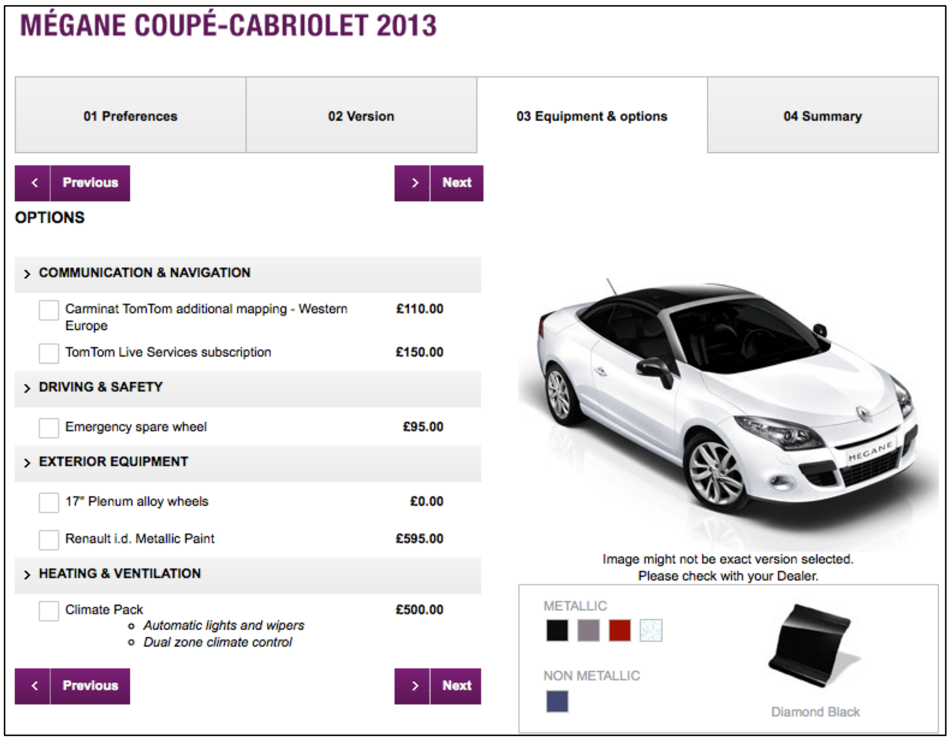
Figure 3.7 presents another example of dynamic content created and added to the page at runtime. By selecting an option (represented using a radio button), its full name, size, and price is dynamically created and presented to the user (Ⓘ).

Variations in business logic management. This characteristic reflects the fact that Web configurators use a diversity of patterns to load options in the pages, handle different kinds of constraints, and control the configuration process.

3.4 Quantitative Results

This section summarises the results of our empirical study[5]. Table 3.1 highlights our key findings. Each subsection answers the questions posed in Section 3.2.

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


MÉGANE COUPÉ-CABRIOLET 2013

01 Preferences | 02 Version | 03 Equipment & options | 04 Summary

< Previous > Next

OPTIONS

> COMMUNICATION & NAVIGATION

☐	Carminat TomTom additional mapping - Western Europe	£110.00
☐	TomTom Live Services subscription	£150.00

> DRIVING & SAFETY

☐	Emergency spare wheel	£95.00

> EXTERIOR EQUIPMENT

☐	17" Plenum alloy wheels	£0.00
☐	Renault i.d. Metallic Paint	£595.00

> HEATING & VENTILATION

☐	Climate Pack ∘ *Automatic lights and wipers* ∘ *Dual zone climate control*	£500.00

< Previous > Next

Image might not be exact version selected.
Please check with your Dealer.

METALLIC

NON METALLIC

Diamond Black

Code Fragments

```
<td>
    <input  type="checkbox"/>
    <label>
        <span class="sectionText"> Renault i.d. Metallic Paint </span>
        <span class="SectionPrice"> £595.00 </span>
    </label>
</td>

<td>
    <input  type="checkbox"/>
     <label>
         <span class="SectionText"> Climate Pack
             <ul>
                 <li> Automatic lights and wipers </li>
                 <li> Dual zone climate control </li>
             </ul>
         </span>
         <span class="SectionPrice"> £500.00 </span>
     </label>
</td>
```

FIGURE 3.6: Template-generated Web page (`http://www.renault.co.uk/`, July 15 2013).

3.4.1 Configuration options (RQ1)

Option representation. The diversity of representations for an option is one of the most striking results, as shown in Figure 3.8. In decreasing order, the most popular widgets are: *combo box item*, *image*[6], *radio button*, *check box* and *text box*. We also

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

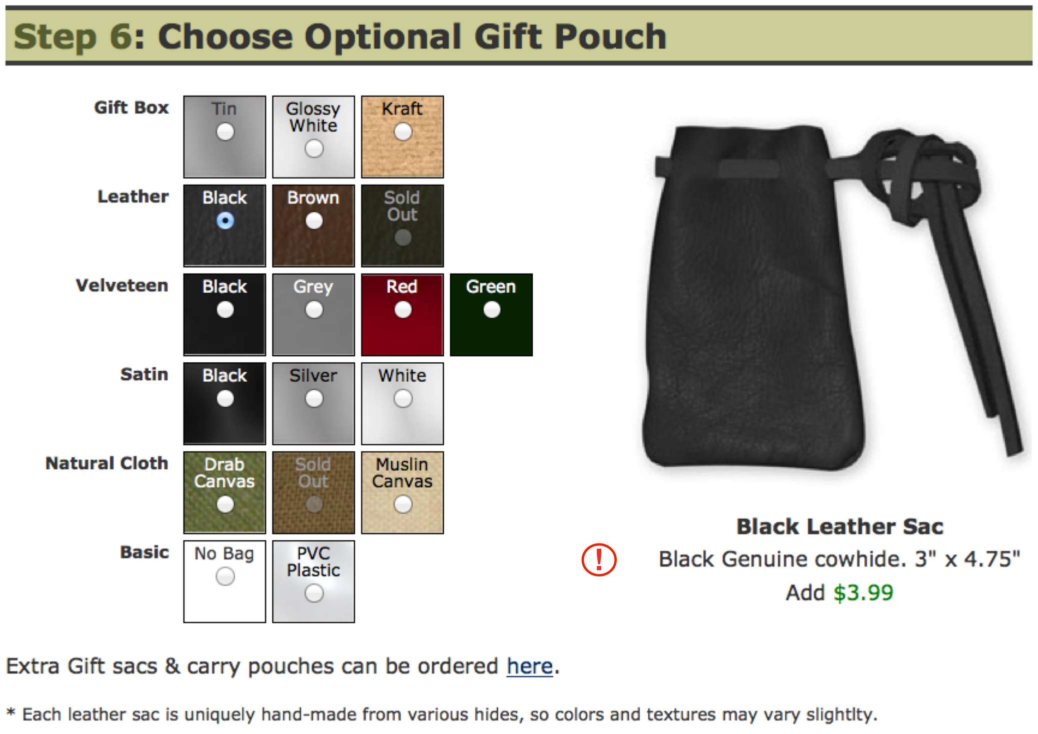

Step 6: Choose Optional Gift Pouch
Gift Box
Tin
Glossy White
Kraft
Leather
Black
Brown
Sold Out
Velveteen
Black
Grey
Red
Green
Satin
Black
Silver
White
Natural Cloth
Drab Canvas
Sold Out
Muslin Canvas
Basic
No Bag
PVC Plastic
Black Leather Sac
Black Genuine cowhide. 3" x 4.75"
Add $3.99
Extra Gift sacs & carry pouches can be ordered here.
* Each leather sac is uniquely hand-made from various hides, so colors and textures may vary slightltly.

FIGURE 3.7: Dynamic content (`http://www.mydogtag.com/`, July 3 2013).

observed that some widgets were combined with images, namely, *check box*, *radio button*, and *combo box item*. Option selection is performed by either choosing the image or using the widget. The *Other* category contains various less frequent widgets like *slider*, *label*, *file chooser*, *date picker*, *colour picker*, *image needle*, and *grid*.

Grouping. Grouping is a way to organise *related* options together. For instance, a group can contain a set of colours or the options for an engine. Three different semantic constraints can apply to a group. For *alternative* groups, one and only one option must be selected (e.g., the "Model" in Figure 3.1 – Ⓖ), and for *multiple choice* groups, at least one option must be selected (e.g., the "Headlights" in Figure 3.1 – Ⓗ). In an *interval* group (a.k.a. cardinality [Czarnecki and Kim, 2005]), the specific lower and upper bounds on the number of selectable options is determined (e.g., "mix-ins" in Figure 3.9). The *Semantic Constructs* row in Table 3.1 shows that *alternative* groups are the most frequent with 97% of configurators implementing them. We also observed multiple choice and interval groups in 8% and 4% of configurators, respectively.

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

Table 3.1: Result summary.

CONFIGURATION OPTIONS		
Semantic Constructs	Alternative group	97%
	Multiple choice group	8%
	Interval	4%
Mandatory Options	Default	46%
	Notification	47%
	Transition Checking	13%
	No checking	4%
Multiple instantiation	Cloning	5%
CONSTRAINTS		
Constraint Type	Formatting	59%
	Group	99%
	Cross-cutting	55%
Cross-cutting Constraint (61)	Visibility	89%
Formatting Constraint (66)	Prevention	62%
	Verification	41%
	No checking	26%
Constraint Description (61)	Explanation	11%
Decision Propagation (61)	Automatic	97%
	Controlled	8%
	Guided	3%
Consistency Checking (83)	Interactive	76%
	Batch	59%
Configuration Operation	Undo	11%
CONFIGURATION PROCESS		
Process	Single-step	48%
	Basic Multi-step	45%
	Hierarchical Multi-step	7%
Activation (58)	Step-by-step	59%
	Full-step	41%
Backward Navigation (58)	Stateful arbitrary	69%
	Stateless arbitrary	14%
	Not supported	17%
Visual Product	Yes	50%
	Not supported	50%
Configuration Summary	Search result	2%
	Final step	13%
	Shopping cart	82%
	Not supported	3%

- *Default configuration* (46%): When the configuration environment is loaded, (some or all) mandatory options are selected or assigned a default value.
- *Notification* (47%): Constraints are checked at the end of the configuration process and mandatory options left undecided are notified to the user. This approach can be mixed with default values, meaning that some of the configurators implement both default configuration and notification patterns.
- *Transition checking* (13%): The user is not allowed to move to the next step until all mandatory options have been selected. The difference with the previous pattern is that no warning is shown to the user.

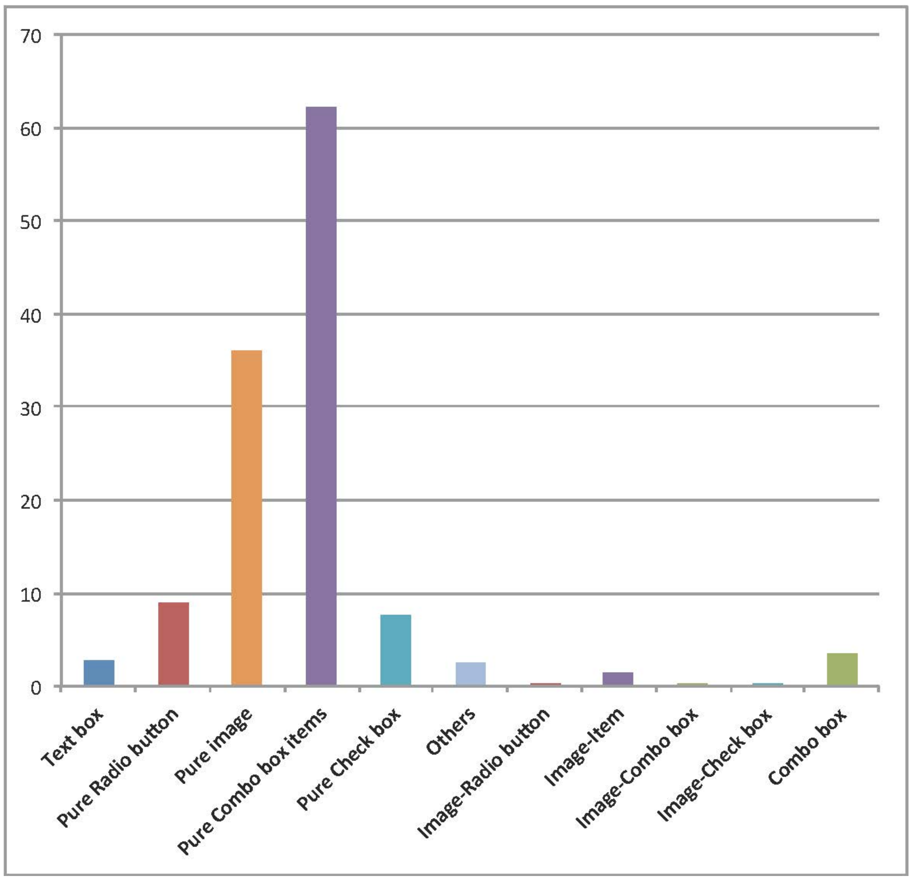
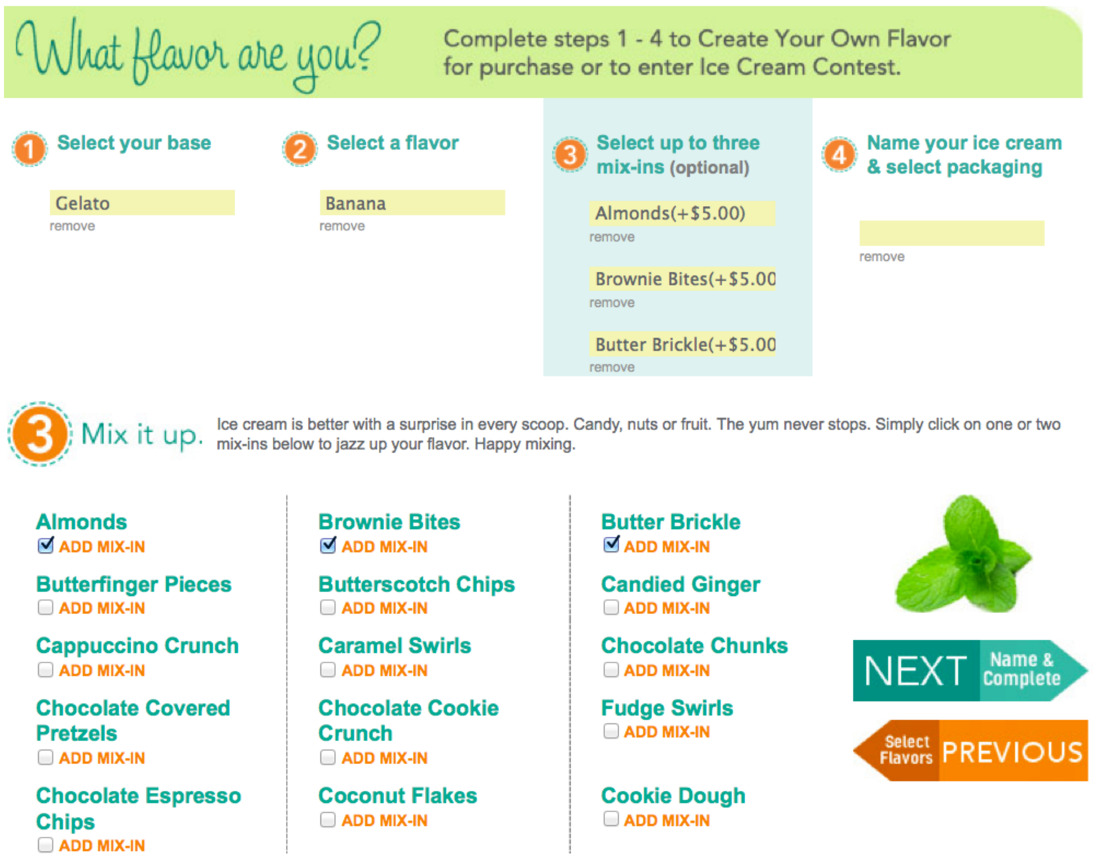


FIGURE 3.8: Widget types in all the configurators.

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

Mandatory options can be distinguished from optional ones through *highlighting*. For that, configurators use symbolic annotations (e.g., * usually for mandatory options), textual keywords (e.g., *required*, *not required*, or *optional*), or special text formatting (e.g., boldfaced, coloured text). 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, while 70% of the configurators have optional options in their configuration environments.

Cloning. Cloning means that the user determines how many instances of an option are included in the final product [Michel et al., 2011b] (e.g., a text element to be printed on a t-shirt can be instantiated multiple times and configured differently). We observed cloning mechanisms in only 5% of the configurators.


What flavor are you?

Complete steps 1 - 4 to Create Your Own Flavor for purchase or to enter Ice Cream Contest.

1 Select your base: Gelato
remove

2 Select a flavor: Banana
remove

3 Select up to three mix-ins (optional): Almonds(+$5.00)
remove
Brownie Bites(+$5.00
remove
Butter Brickle(+$5.00
remove

4 Name your ice cream & select packaging
remove

3 Mix it up. Ice cream is better with a surprise in every scoop. Candy, nuts or fruit. The yum never stops. Simply click on one or two mix-ins below to jazz up your flavor. Happy mixing.

Almonds ✓ ADD MIX-IN	Brownie Bites ✓ ADD MIX-IN	Butter Brickle ✓ ADD MIX-IN
Butterfinger Pieces ☐ ADD MIX-IN	Butterscotch Chips ☐ ADD MIX-IN	Candied Ginger ☐ ADD MIX-IN
Cappuccino Crunch ☐ ADD MIX-IN	Caramel Swirls ☐ ADD MIX-IN	Chocolate Chunks ☐ ADD MIX-IN
Chocolate Covered Pretzels ☐ ADD MIX-IN	Chocolate Cookie Crunch ☐ ADD MIX-IN	Fudge Swirls ☐ ADD MIX-IN
Chocolate Espresso Chips ☐ ADD MIX-IN	Coconut Flakes ☐ ADD MIX-IN	Cookie Dough ☐ ADD MIX-IN

FIGURE 3.9: Interval group (`http://www.ecreamery.com/createyourown.html`, August 9 2013).

3.4.2 Constraints (RQ2)

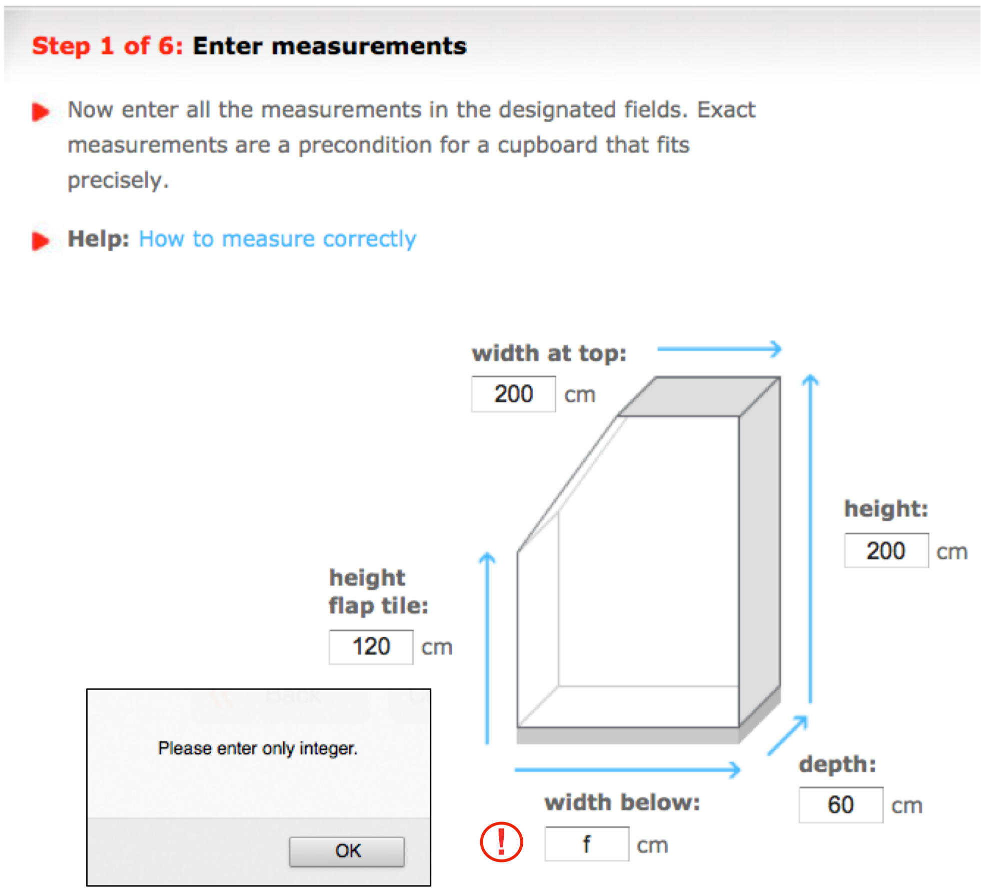
We split constraints in three categories depending on their target and implementation.

Formatting constraint. A formatting constraint ensures that the value set by the user is *valid*. Examples of formatting constraints are:

- *Type correctness:* Some options are strongly typed (e.g., String, Integer, Real), which means that types must be verified to produce valid configurations. For example, in Figure 3.10 only integer values can be set to the text inputs. If the user enters an invalid value (for example, a string value, ⓘ), the reasoning procedure prevents the illegal value.

- *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. For example, in Figure 3.11(a) the minimum and maximum integer values to be entered in the text boxes must respectively be 37 and 1000. Values beyond this range violate the configuration rule and is prevented by the reasoning procedure (!). Also, in Figure 3.11(b) allowed characters that the user can use in filling the text inputs are presented to her (!). The reasoning procedure automatically removes invalid characters from the text inputs.

- *Formatted values:* Some more values require specific formatting constraints such as date, email, 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 input the selected value. Similarly, to capture the deselection of an option, some configurators explicitly ask the user to enter values like *None*, or *No*. For example, to configure a colour option in Figure 3.12 (A) the user should enter a valid colour name. Also, as for the text style, if she wants a sample text she should put the "As Sample" string in the text input (B).


Step 1 of 6: Enter measurements

Now enter all the measurements in the designated fields. Exact measurements are a precondition for a cupboard that fits precisely.

Help: How to measure correctly

width at top: 200 cm

height: 200 cm

height flap tile: 120 cm

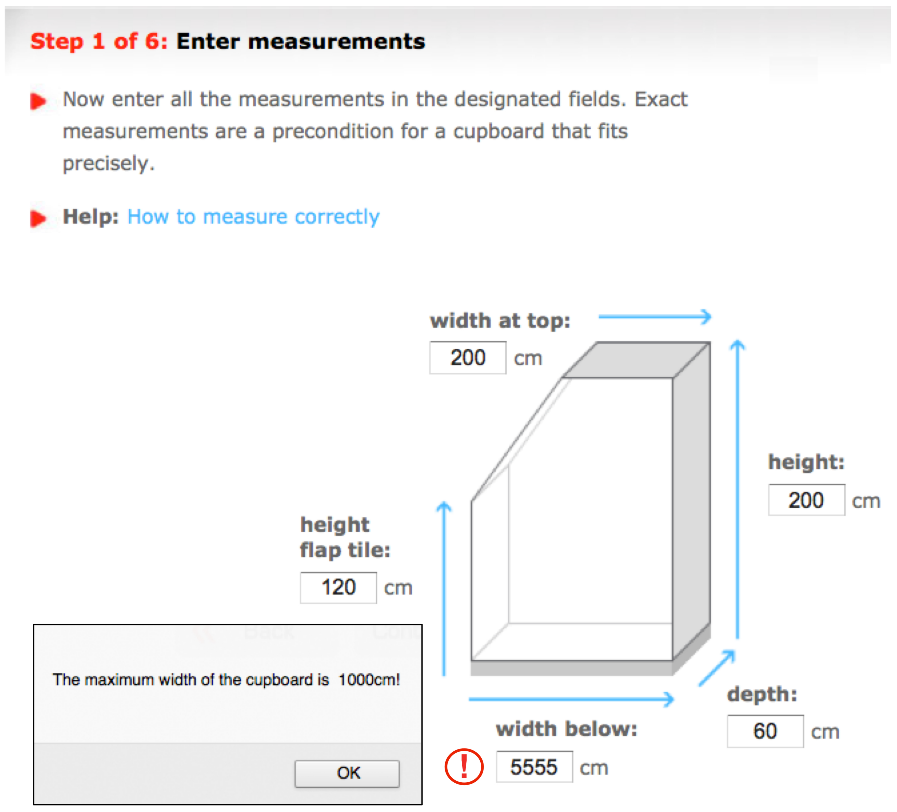
depth: 60 cm

width below: f cm

Please enter only integer.

FIGURE 3.10: Type correctness constraint (`http://www.cupboardyourway.co.uk`, June 13 2013).

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



Step 1 of 6: Enter measurements

Now enter all the measurements in the designated fields. Exact measurements are a precondition for a cupboard that fits precisely.

Help: How to measure correctly

width at top: 200 cm

height: 200 cm

height flap tile: 120 cm

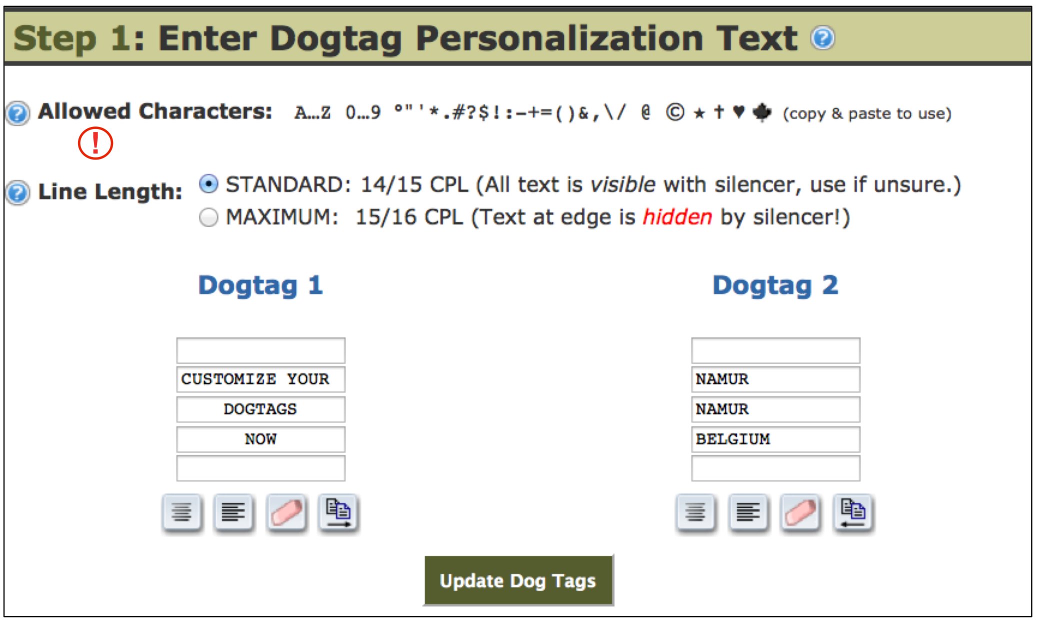
depth: 60 cm

width below: 5555 cm

The maximum width of the cupboard is 1000cm!

OK

(a) Upper bound (`http://www.cupboardyourway.co.uk`, June 13 2013).



Step 1: Enter Dogtag Personalization Text

Allowed Characters: A…Z 0…9 °"'*.#?$!:-+=()&,\/ @ © ★ † ♥ ♣ (copy & paste to use)

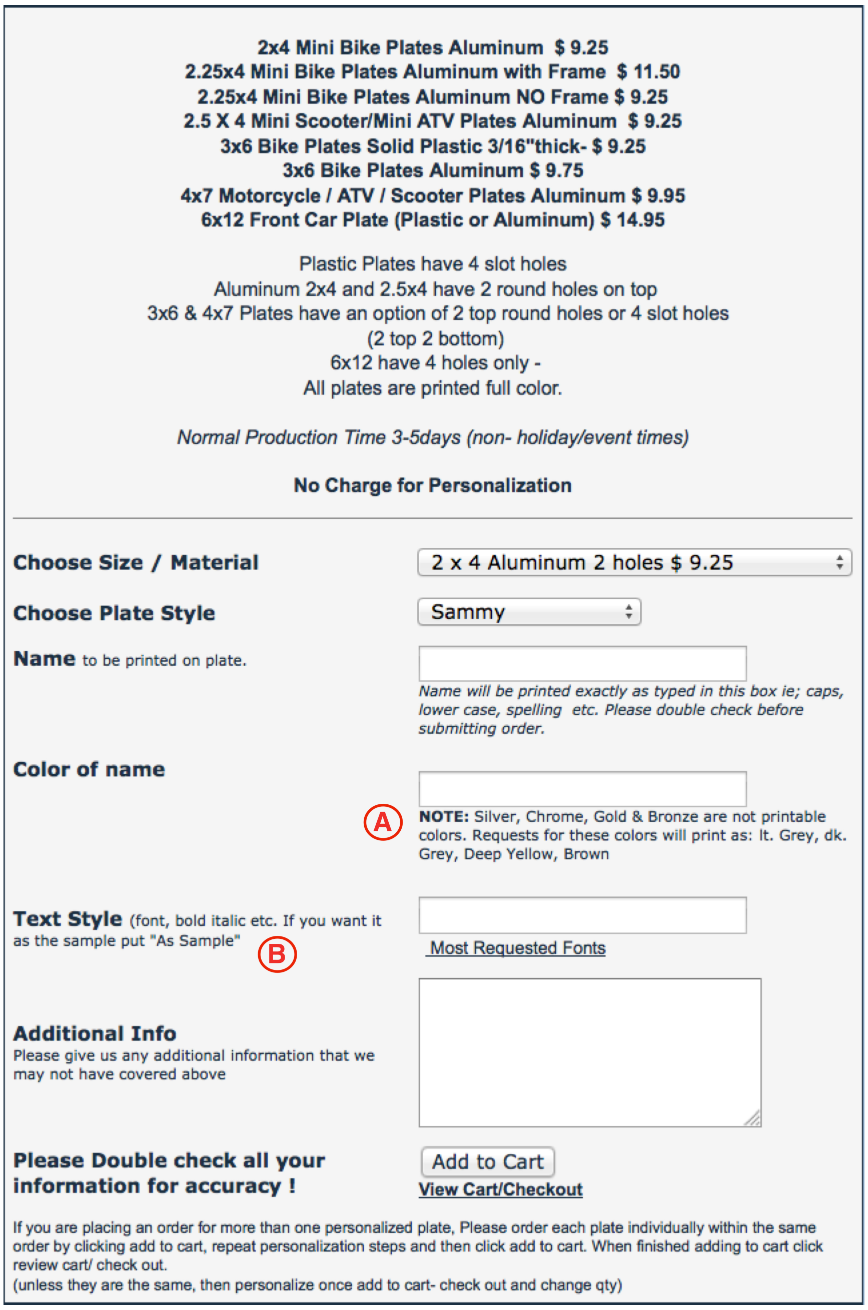
Line Length: [✓ STANDARD: 14/15 CPL (All text is *visible* with silencer, use if unsure.)] MAXIMUM: 15/16 CPL (Text at edge is *hidden* by silencer!)

Dogtag 1	Dogtag 2
CUSTOMIZE YOUR	NAMUR
DOGTAGS	NAMUR
NOW	BELGIUM

Update Dog Tags

(b) Valid characters (`http://www.mydogtag.com/`, June 13 2013).

Figure 3.11: Range control constraints.



2x4 Mini Bike Plates Aluminum $ 9.25
2.25x4 Mini Bike Plates Aluminum with Frame $ 11.50
2.25x4 Mini Bike Plates Aluminum NO Frame $ 9.25
2.5 X 4 Mini Scooter/Mini ATV Plates Aluminum $ 9.25
3x6 Bike Plates Solid Plastic 3/16"thick- $ 9.25
3x6 Bike Plates Aluminum $ 9.75
4x7 Motorcycle / ATV / Scooter Plates Aluminum $ 9.95
6x12 Front Car Plate (Plastic or Aluminum) $ 14.95

Plastic Plates have 4 slot holes
Aluminum 2x4 and 2.5x4 have 2 round holes on top
3x6 & 4x7 Plates have an option of 2 top round holes or 4 slot holes
(2 top 2 bottom)
6x12 have 4 holes only -
All plates are printed full color.

Normal Production Time 3-5days (non- holiday/event times)

No Charge for Personalization

Choose Size / Material 2 x 4 Aluminum 2 holes $ 9.25

Choose Plate Style Sammy

Name to be printed on plate.

Name will be printed exactly as typed in this box ie; caps, lower case, spelling etc. Please double check before submitting order.

Color of name

A

NOTE: Silver, Chrome, Gold & Bronze are not printable colors. Requests for these colors will print as: lt. Grey, dk. Grey, Deep Yellow, Brown

Text Style (font, bold italic etc. If you want it as the sample put "As Sample"

B

Most Requested Fonts

Additional Info
Please give us any additional information that we may not have covered above

Please Double check all your information for accuracy !

Add to Cart
View Cart/Checkout

If you are placing an order for more than one personalized plate, Please order each plate individually within the same order by clicking add to cart, repeat personalization steps and then click add to cart. When finished adding to cart click review cart/ check out.
(unless they are the same, then personalize once add to cart- check out and change qty)

Figure 3.12: Case-sensitive values (http://www.personalizedbikeplates.com/, June 13 2013).

- *Prevention:* The reasoning procedure prevents illegal values. For example, it stops accepting input characters if the maximum number is reached, defines a slider domain, uses a date picker, disables illegal options, etc.

- *Verification:* The reasoning procedure 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.

These patterns are not mutually exclusive, and configurators can use them for different subsets of options. Among the 66 configurators supporting formatting constraints, 62% implement prevention and 41% implement verification patterns. We also noticed that 26% of the configurators do not check constraints during the configuration session even if they are described in the interface. In some rare cases, the validation of the configuration was performed off-line, and feedback later sent back to the user.

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

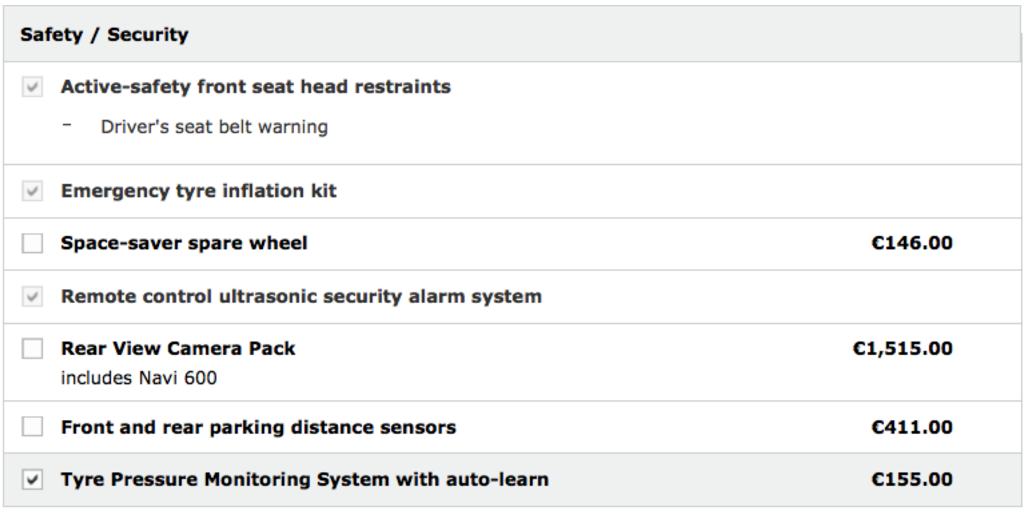
Cross-cutting constraint. A cross-cutting constraint is defined over two or more options regardless of their inclusion in a group. *Require* (selecting A implies selecting B) and *Exclude* (selecting A prevents selecting B and vice-versa) constraints are the most common. More complex constraints exist too. For instance, in Figure 3.13, the selection of "Active-safety front seat head restraints" implies the selection of "Driver's seat belt warning". Also, the selection of "Space-saver spare wheel" implies the deselection of "Emergency tyre inflation kit" .

Cross-cutting constraints were observed in 61 configurators (55%) and are either coded in the client side (e.g., using JavaScript) or in the server side (e.g., using PHP). Irrespective of the implementation technique, we noticed that only 11% of the configurators describe them in the GUI with a textual explanation.

Visibility constraint. Some constraints determine when options are shown or hidden in the GUI. They are called *visibility constraint* [Berger et al., 2010]. Automatically adding options to a combo box upon modification of another option also falls in this constraint category. From the 61 configurators with cross-cutting constraints, 89% implement visibility constraints.

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

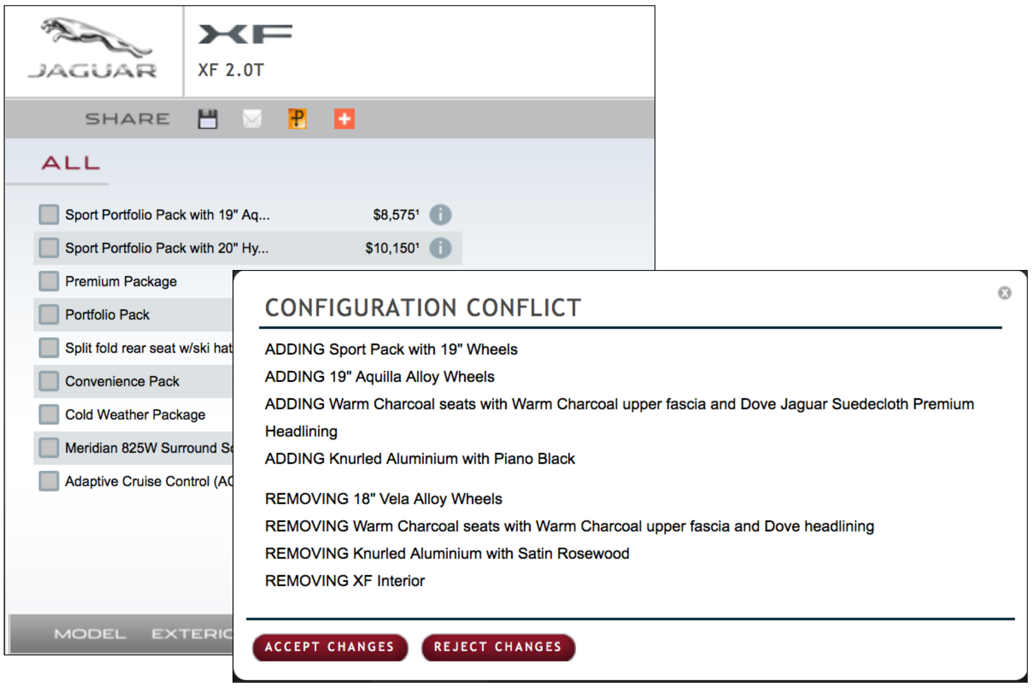
Decision propagation. In some configurators, when an option is given a new value and one or more constraints apply, the reasoning procedure automatically propagates the required changes to all the impacted options (Figure 3.13). We call it *automatic* propagation (97%). We observed that in some cases by selecting an option its consistent options are loaded in the page. We counted these cases as automatic decision propagation as well. For instance, in Figure 3.1, by selecting an option in the "Model line" group, new options are loaded to the "Body style" group. In other cases, the reasoning procedure asks to confirm or discard a decision before altering other options (Figure 3.14). We call this *controlled* propagation (8%). Finally, we also observed some cases of *guided* propagation (3%). For example, if option A requires to select option B or C, the reasoning procedure cannot decide whether B or C should be selected knowing A. In this case, the configurator proposes a choice to the user (Figure 3.15). Some of the configurators implement multiple patterns.


Safety / Security

✓	**Active-safety front seat head restraints** – Driver's seat belt warning	
✓	**Emergency tyre inflation kit**	
☐	**Space-saver spare wheel**	**€146.00**
✓	**Remote control ultrasonic security alarm system**	
☐	**Rear View Camera Pack** includes Navi 600	**€1,515.00**
☐	**Front and rear parking distance sensors**	**€411.00**
✓	**Tyre Pressure Monitoring System with auto-learn**	**€155.00**

FIGURE 3.13: Automatic decision propagation (`http://www.opel.ie/`, August 9 2013).

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

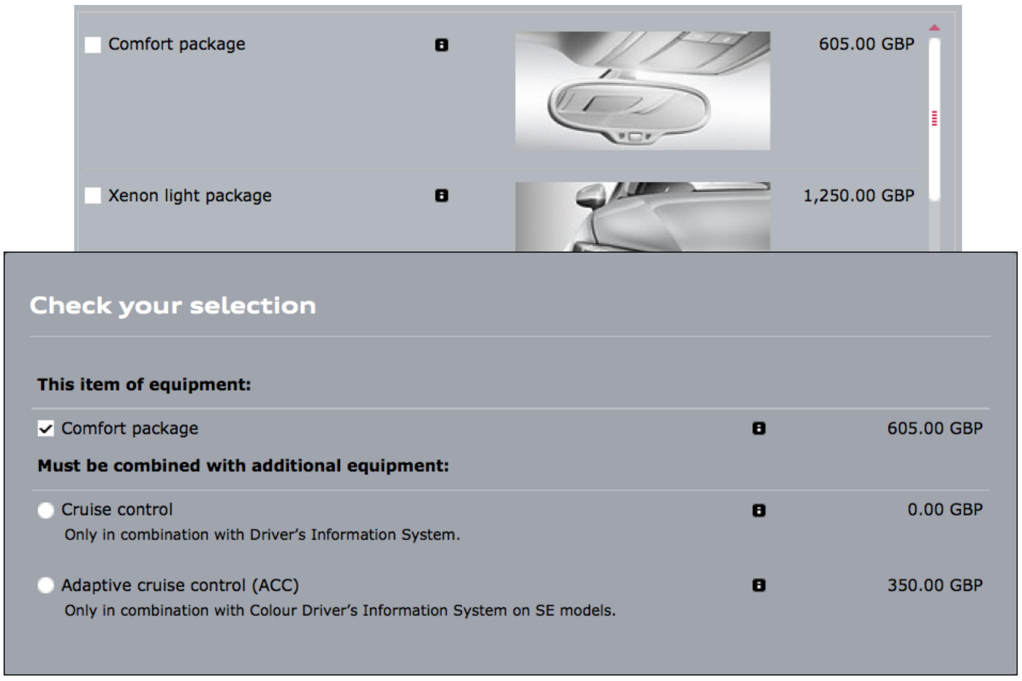

JAGUAR
XF
XF 2.0T
SHARE
ALL
Sport Portfolio Pack with 19" Aq... $8,575¹
Sport Portfolio Pack with 20" Hy... $10,150¹
Premium Package
Portfolio Pack
Split fold rear seat w/ski hat
Convenience Pack
Cold Weather Package
Meridian 825W Surround S
Adaptive Cruise Control (AC
MODEL
EXTERIO

CONFIGURATION CONFLICT

ADDING Sport Pack with 19" Wheels
ADDING 19" Aquilla Alloy Wheels
ADDING Warm Charcoal seats with Warm Charcoal upper fascia and Dove Jaguar Suedecloth Premium Headlining
ADDING Knurled Aluminium with Piano Black

REMOVING 18" Vela Alloy Wheels
REMOVING Warm Charcoal seats with Warm Charcoal upper fascia and Dove headlining
REMOVING Knurled Aluminium with Satin Rosewood
REMOVING XF Interior

FIGURE 3.14: Controlled decision propagation (`http://www.jaguarusa.com/`, July 31 2013).



Check your selection

This item of equipment:

✓ Comfort package	605.00 GBP

Must be combined with additional equipment:

○ Cruise control Only in combination with Driver's Information System.	0.00 GBP
○ Adaptive cruise control (ACC) Only in combination with Colour Driver's Information System on SE models.	350.00 GBP

FIGURE 3.15: Guided decision propagation (`http://configurator.audi.co.uk/`, August 9 2013).

configurators supporting both formatting and cross-cutting constraints, 76% implement interactive and 59% implement batch consistency checking patterns. Some configurators implement both mechanisms, depending on the constraint type.

Undo. This operation allows users to roll back on their last decision(s). Among all configurators in the survey, only 11% support undo. Note that, supporting undo requires the configurator to keep a log of operations done by the user.

3.4.3 Configuration process (RQ3)

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

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

Activation. Among the 58 multi-step configurators, we noticed two exclusive *step activation* strategies:

- *Step-by-step activation* (59%): 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* (41%): 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* (69%): The user can go back to any previous step and all configuration choices are saved.
- *Stateless arbitrary* (14%): The user can go back to any previous step but all configuration choices made in steps following the one reached are discarded.

We observed that all full-step activation configurators follow the stateful arbitrary navigation pattern. We also noticed that 17% of multi-step configurators do not support backward navigation.

We gathered two additional facts about configuration processes: *visual product* and *configuration summary*. 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.

3.5 Qualitative Results

The previous section focused on technical characteristics of 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 [Hubaux et al., 2012; Hvam et al., 2008; Rogoll and Piller, 2004; Streichsbier et al., 2009; Trentin et al., 2012]. Note that the impact of marketing or sales decisions on the behaviour of configurators falls outside our scope of investigation. We focus here on their perception by end-users that are likely to influence the way configurators are implemented.

3.5.1 Bad practices

- *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 becomes impossible to predict and visualise. 97% of the configurators automatically propagate decisions but rarely inform users of the impact of their decisions.
- *Incomplete reasoning:* Reasoning procedures are not always complete. Some configurators do not check that mandatory options are indeed selected, or do not

verify formatting constraints. 26% of the configurators do not check formatting constraints during the configuration session.

- *Counter-intuitive representation*: The visual discrepancies between option representations are striking. This is not a problem *per se*. The issue lies in the improper characterisation of the semantics of the widgets. For instance, some exclusive options are implemented by (non exclusive) check boxes. Consequently, users only discover the grouping constraint by experimenting with the configurator, which causes confusion and misunderstanding. It also increases the risk of inconsistency between the intended and implemented behaviour.
- *Stateless backward navigation*: Stateless configurators lose all decisions when navigating backward. This is a severe defect since users are extremely likely to make mistakes or change their mind on some decisions. 31% of the configurators do not support backward navigation or are stateless.
- *Automatic step transition*: The user is guided to the next step once all options are configured. Although this is a way to help users [Rogoll and Piller, 2004], it also reduces control over configuration and hinders decision review.
- *Visibility constraints*: When a visibility constraint applies, options are hidden and/or deactivated. This reduces the solution space [Hvam et al., 2008] and avoids conflictual decisions. However, the downside is that to access hidden/deactivated options, the user has to first undo decisions that instantiated the visibility constraint. These are known problems in configuration [Hubaux et al., 2012] 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 each time they want to alter their configuration.

3.5.2 Good practices

- *Guided consistency checking*: 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 [Hvam et al., 2008], which are known to improve usability [Rogoll and Piller, 2004].
- *Auto-completion* allows users to configure some desired options and then let the configurator complete undecided options [Janota et al., 2009]. Auto-completion is

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

- *Self-explanatory process*: A configurator should provide clear guidance during the configuration process [Hvam et al., 2008; Rogoll and Piller, 2004; Streichsbier et al., 2009]. The multi-step configurators we observed use various mechanisms such as numbered steps, "previous" and "next" buttons, the permanent display of already selected options, a list of complete/incomplete steps, etc. Configurators should also be able to explain constraints "on the fly" to the users. This is only available in 11% of the configurators.
- *Self-explanatory widgets*: Whenever possible, configurators should use standard widget types, explicit bounds on intervals, optional/mandatory option differentiation, item list sorting and grouping in combo boxes, option selection/deselection mechanisms, filtering or searching mechanisms, price live update, spell checker, default values, constraints described in natural language, and examples of valid user input.
- *Stateful backward navigation* and *undo*: These are must-have functionalities to allow users to revise their decisions. Yet, only 69% and 11%, respectively, of the Web configurators do support them.

3.6 Reverse Engineering Challenges

Our long-term objective, i.e. developing methods to systematically re-engineer Web configurators, requires accurate data extraction techniques. For the purpose of this study, we implemented a semi-automated tool to retrieve options, constraints and configuration processes (see Section 3.2.2). This tool can serve as a basis for the reverse-engineering part of the future re-engineering toolset. This section outlines the main technical challenges we faced and how we overcame them. The impact of our design decisions on our results are explored in the next section.

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

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

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

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

3.7 Threats to Validity

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

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

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

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

3.8 Related Work

Rogoll *et al.* [Rogoll and Piller, 2004] performed a qualitative study of 10+ Web configurators. The authors reported on *usability* and how visual techniques assist customers in configuring products. Our study is larger (100+ configurators), and our goal and methodology differ significantly. We aim at understanding how the underlying concepts of Web configurators are 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.* [Streichsbier et al., 2009] analysed 126 Web Configurators among those in [Cyledge, 2014]. 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 and validate existing standards in 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 process, which is in line with [Streichsbier et al., 2009]. Trentin *et al.* [Trentin et al., 2012] conducted a user study of 630 Web configurators to validate five capabilities: *focused navigation*, *flexible navigation*, *easy comparison*, *benefit-cost communication*, and *user-friendly product-space description*. We adopted a more technical point of view. Moreover, their observations are purely qualitative and no automated reverse engineering procedure is applied to produce quantitative observations.

3.9 Chapter Summary

In this chapter, we presented an empirical study of 111 Web configurators along three dimensions: configuration options, constraints and configuration process.

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

Qualitative insights. The empirical analysis of Web configurators reveals reliability issues when handling constraints. These problems come from the configurators' lack of convincing support for consistency checking and decision propagation. For instance, although verifying mandatory options and constraints are basic operations for configurators, our observations show that they are not completely implemented. Moreover, the investigation of client-side code implementation verifies, in part, that no systematic method (e.g., solver-based) is applied to implement reasoning operations. We believe that the use of variability models to formally capture configuration options and constraints, and solvers used in more academic configuration tools (e.g., SAT and SMT) to reason about these models, would provide more effective and reliable bases. We also noticed that usability is rather weak in many cases (e.g., counter-intuitive representations, lack of guidance).

Chapter 4

Reverse Engineering Web Applications: State of the Art

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*. This chapter is dedicated to describing relevant work in these fields of research. We first present several approaches applied to the reverse engineering of Web applications (Section 4.1), we then provide an overview of existing Web data extraction methods (Section 4.2) and continue with techniques used for synthesizing feature models (Section 4.3). Finally, we conclude this chapter by a discussion about the limitations of existing approaches to reverse engineer feature models from Web configurators (Section 4.4).

4.1 Reverse Engineering Web Applications

Reverse engineering is the process of analysing a subject system to identify the system's components and their interrelationships, and create a representation of the system in another form or at a higher level of abstraction [Chikofsky and Cross II, 1990]. This definition fits our purpose quite well. For us, the subject system is a Web configurator, its configuration options are components to be identified and extracted, and constraints defined over options are their interrelationships that will be documented. Variability models and process models are higher abstractions that are synthesised at the end of the reverse-engineering process.

In the context of Web application reverse engineering, it is important to understand and consider the types of target applications [Patel et al., 2007]. Web applications can be categorised into three classes [Tilley and Huang, 2001]:

- **Class 1:** Static applications implemented in HTML with no user interaction,
- **Class 2:** Client-side interaction with Dynamic HTML (DHTML), typically using mouse clicks, and
- **Class 3:** Contain dynamic content created " on-the-fly", typically use technologies such as JSP, Java Servlets, ASP, PHP, etc.

The degree of complexity associated with the reverse-engineering process of applications in Class 3, and therefore the effort required, is higher than that associated with classes 1 and 2 [Patel et al., 2007]. A Web configurator is a highly interactive application and new content is dynamically created and added to the page when the application is executing (see Chapter 3.3). Consequently, Web configurators fall in Class 3.

Over the years, many approaches have been proposed to reverse engineer Web applications for different purposes. Among all existing approaches, we present and discuss some here. We refer the reader to [Bouillon, 2006; de Silva, 2010; Patel et al., 2007] for a more detailed state of the art in reverse engineering Web applications.

GUITAR. GUITAR [Nguyen et al., 2013] is a flexible framework used for automated testing of GUI-driven software. It supports a wide variety of GUI testing techniques for different platforms. WebGUITAR [1] is a tool provided by this framework for automated testing of Web applications. The WebGUITAR workflow process has four major steps:

- *Web Ripper:* The purpose of Ripper is to discover as much structural information about the GUI as possible using automated algorithms and some human input. Web Ripper automatically executes the target Website and extracts elements such as links, buttons, images, etc. and creates a structure called *GUI Tree.* The GUI Tree is an XML file containing information about the ripped windows and their contained elements. The dependencies between these elements and windows are documented as well.
- *Event Flow Graph (EFG) Converter:* Once the GUI Tree is created, EFG Converter converts it into a format to be used by Test Case Generator to build test cases.

[1] `http://sourceforge.net/apps/mediawiki/guitar/index.php?title=WebGuitar`

- *Test Case Generator:* Based on the dependencies of the GUI, Test Case Generator creates meaningful test cases.
- *Web Replayer:* Given the GUI Tree, Event Flow Graph structure, and generated test cases, Web Replayer tests the Website for proper functionality and outputs results to a *State File*. State Files contain the Website's state after each intermediate step of the test case.

The use of WebGUITAR to reverse engineer variability models from Web configurators requires substantial changes to its core procedures. It implements algorithms to automatically generate GUI-based test cases. For this reason, during ripping it extracts only GUI objects (widgets, windows, etc.) and ignores data objects represented in the page. For instance, if we use WebGUITAR to extract options from a Web configurator, the generated GUI Tree will contain widgets representing these options and exclude key data items such as options' names and other associated descriptive information. Adapting those algorithms to consider configuration aspects and specific properties of Web configurators is an effortful task and we believe that will not lead to satisfactory results. For example, Ripper should be improved to also consider configuration semantics of GUI elements. It needs to be somehow parametrised to only consider GUI elements that represent configuration objects (e.g., radio buttons, check boxes, etc.) not all elements of the page. Moreover, in addition to GUI elements, Ripper should also extract and structure data objects.

VAQUISTA. VAQUISTA (reVerse engineering of Applications by Questions, Information Selection, and Transformation Alternatives) [Vanderdonckt et al., 2001] addresses the problem of migration of the UI of a Web page to another environment. Figure 4.1 presents the complete process envisioned with VAQUISTA. It provides a user-interface reverse-engineering process and recovers a *presentation model* from a single Web page at a time based on mapping rules between HTML elements and presentation elements. VAQUISTA does not aim to reverse engineer a whole Website, rather its goal is to export highly interactive parts (e.g., input forms) of a Web page to another context. Once the presentation model is created, it is then used in a forward-engineering process to generate a new UI in a given context. For instance, using *SEGUIA* they can automatically generate a Windows UI from a presentation model.

VAQUISTA scans the HTML code of a given Web page, identifies types of HTML tags, elements, and possible attached values, and represents them in a presentation model. The produced presentation model is just an abstraction of the DOM of the page and represents the visual elements provided by a UI to its user. Consequently, the presentation model of a page does not give an advantage over the page itself to extract

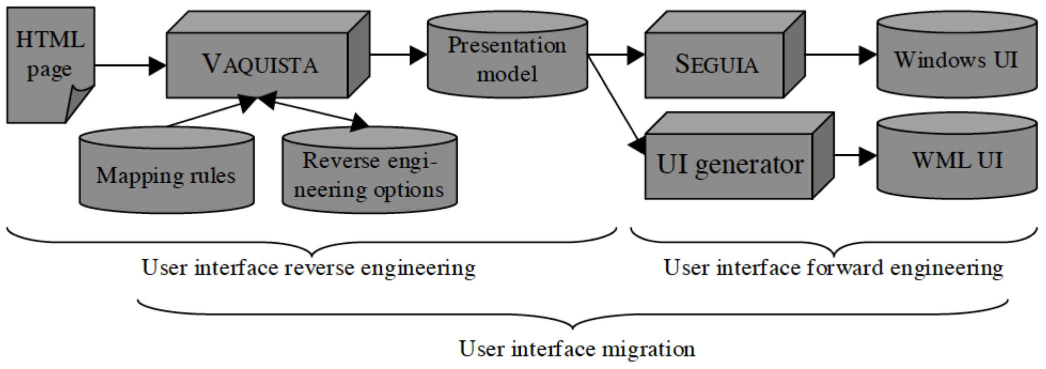


FIGURE 4.1: UI migration process with VAQUISTA ([Vanderdonckt et al., 2001]).

variability data. Moreover, VAQUISTA applies a static analysis of HTML elements of a Web page without executing the application. It means that VAQUISTA does not document runtime behaviour of the target application. A Web configurator is an interactive application and the amount of data as well as widgets that are dynamically generated and added to the page when the configurator is executing is considerable. VAQUISTA is not able to deal with this runtime behaviour.

GuiSurfer. GuiSurfer [de Silva, 2010; Silva et al., 2010] is a generic tool to reverse engineer the GUI layer of interactive computing systems. Its main goal is to enable analysis of interactive systems from source code. Figure 4.2 shows the architecture of the tool. GuiSurfer is composed of two phases: a language dependent phase and a language independent phase. Hence, for a new language to be targeted by GuiSurfer, only the language dependent phase should be transformed.

GuiSurfer first creates an *Abstract Syntax Tree* (AST) using a parser on the source code of the target application. The generated AST represents the entire code of the application. However, since GuiSurfer's focus is on the GUI layer, not the entire source code, it analyses the AST and retrieves only the GUI relevant nodes and ignores the others. This is achieved by using techniques such as strategic programming [Visser, 2004a] and code slicing. Strategic programming allows novel forms of abstraction and modularization that are useful for program construction in general [Visser, 2004b]. Code slicing, aka *program slicing*, is the task of computing program slices. A program slice consists of the parts of a program that (potentially) affect the values computed at some point of interest [Tip, 1995].

To create the GUI layer, GuiSurfer looks for the GUI elements in the source code. The considered elements are widgets that enable users to input data (*user input*), widgets that enable users to choose between several different options such as a command menu (*user selection*), any action that is performed as the result of user input or user selection (*user action*), and any widget that enables communication from the application to users (*output to user*). Once the GUI layer model has been created, GuiSurfer performs reasoning over the generated model. For instance, it creates *Event-Flow Graph* models that abstract all the interface widgets and their relationships.

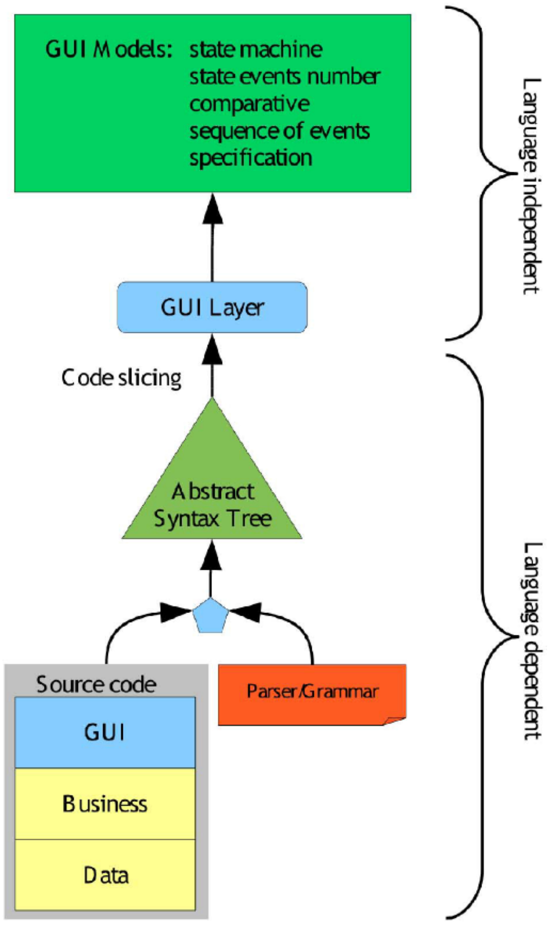


FIGURE 4.2: GuiSurfer's tool architecture ([Silva et al., 2010]).

We claim that GuiSurfer is not applicable to reverse engineer feature models from Web configurators. First, it considers only the GUI layer of the application and ignores the data layer, while our focus in reverse engineering Web configurators is on their data layers to extract variability data. Second, GuiSurfer relies on the static analysis of the source code of an application to reverse engineer the user interface behaviour and structure without executing the application. If GuiSurfer is applied to a Web configurator, it is not able to study its dynamic aspects, and consequently, it is not able to extract those widgets that are dynamically created at runtime and require dynamic analysis to be identified.

CRAWLJAX. CRAWLJAX is a tool for crawling AJAX-based applications through automatic analysis of user-interface-state changes in Web browsers. It scans the DOM tree, spots candidate elements that are capable of changing the states, fires events on those candidate elements, and incrementally infers a state machine that models the various navigational paths and states within an AJAX application. The inferred model can be used in program comprehension and in analysis and testing of dynamic Web states [Mesbah et al., 2012]. The main components of CRAWLJAX are shown in Figure 4.3. The *embedded browser* provides a common interface for accessing the DOM. The *robot* is used to simulate user actions (e.g., *click*, *mouseOver*, text input). The *controller* controls the robot's actions and updates the state machine when relevant changes occur on the DOM. The *DOM Analyzer* is used to check whether the DOM tree has changed after an event has been fired by the robot. The *Finite State Machine* is a data component maintaining the state-flow graph. The state-flow graph records the states and transitions between them. Figure 4.4 depicts the visualization of the state-flow graph of an AJAX site. Each vertex represents a runtime DOM state and each edge represents a transition between two participating states. The edges between states are labelled with an identification (either via its ID attribute or an XPath expression) of the clickable element. For instance, clicking on the `//DIV[1]/SPAN[4]` element in the index state leads to the s_1 state.

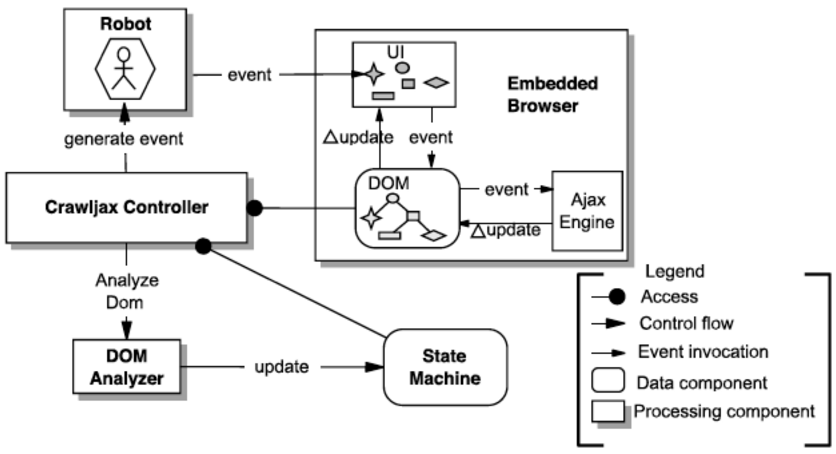


FIGURE 4.3: Processing view of CRAWLJAX ([Mesbah et al., 2012]).

Although CRAWLJAX analyses and records relationships between widgets in the state-flow graph, it does not consider the semantics behind these relationships. For instance, the selection of a check box that implies selection of another check box, for which CRAWLJAX creates and adds a new state to the state-flow graph, should be interpreted as the presence of a *require* constraint between the corresponding options of these two

Figure 4.4: The state- ow graph of an AJAX site created by CRAWLJAX ([Mesbah et al., 2012]).

check boxes. Moreover, CRAWLJAX's focus is on the GUI layer of a Web application and ignores the data layer, where most of the variability data resides.

WARE. The WARE (Web Application Reverse Engineering) approach [Di Lucca et al., 2004; Tramontana, 2005] implements a process, including reverse-engineering methods and a supporting software tool, that helps to understand existing undocumented Web applications to be maintained or evolved, through the reconstruction of UML diagrams. Figure 4.5 illustrates the reverse-engineering process in the WARE approach. In the rst step, the source code of the application is statically analysed in order to identify Web application entities (such as pages, forms, scripts, and other Web objects) and their static relations. The code instructions producing link, submit, redirect, build, and other relationships are identi ed as well. In the second step, dynamic analysis is executed with the aim of recovering dynamic information, for instance, retrieving the actual content of dynamically built client pages, deducing links between pages that were de ned at runtime, etc. In the third step, the problem of grouping together sets of components that collaborate to the realization of a functionality of the Web application is addressed, and the components are partitioned into clusters. In the nal step, UML diagrams are created on the basis of the information extracted in the previous steps. For instance, the class diagram describing the structure of the application is obtained by analysing the information extracted about the application entities and their relationships.

The goal of the WARE approach is to retrieve, from the source code of a Web application, These methods mostly focus on GUI elements and do not seek to extract and structure data objects that are associated with that elements. They also do not propose dedicated techniques for (1) locating configuration objects (e.g., options) in a Web page or for (2) analysing the dynamics and the specificity of a configuration process.

4.2 Web Data Extraction

A Web data extraction system is generally defined as a software system that extracts data from Web pages and delivers the extracted data to a database or some other application [Baumgartner et al., 2009; Laender et al., 2002b]. A Web data extraction system implements a sequence of procedures, called *Web wrappers*. A Web Wrapper, that might implement one or different class(es) of algorithms, *seeks* and *finds* data required by a human user, *extracts* them from unstructured or semi-structured Web sources, *transforms* them into structured data, *merges* and *unifies* this information for *further processing*, in a semi-automatic or fully automatic way [Ferrara et al., 2012].

The challenging aspect of wrappers is that they must be able to recognize the data of interest among many other uninteresting pieces of text [Laender et al., 2002b]. Moreover, when the content or structure of data of a page is changed, the wrapper should be adapted accordingly to keep working properly [Baumgartner et al., 2009; Ferrara et al., 2012]. A common goal in Web data extraction systems is that the wrapper developed for a given Web page or Website should be able to extract data from any other *similar* Web pages or Websites [Laender et al., 2002b].

Due to the difficulty in manually writing and maintaining wrappers [Laender et al., 2002b], several approaches have been proposed to address the problem of automatically generating wrappers [Crescenzi et al., 2001; Kushmerick, 2000; Liu et al., 2000; Muslea et al., 2001; Sahuguet and Azavant, 2001] in order to minimize the effort required from the wrapper developers. In practice, it should be a satisfactory trade-off between the degree of the automation of a tool and the accuracy of the extracted data [Phan et al., 2005]. Many automatic tools are either inaccurate or make many assumptions [Zhai and Liu, 2005].

Over the past years, many Web data extraction approaches have been proposed. A number of reviews surveyed these approaches and provided taxonomies to classify them. We first present three of those taxonomies:

Laender *et al.* [Laender et al., 2002b] presented a taxonomy for grouping the various tools based on the main technique used by each tool to generate a wrapper: *languages for wrapper development* offer formalisms for the development of wrappers [Arocena and Mendelzon, 1999; Crescenzi and Mecca, 1998; Hammer et al., 1997], *HTML-aware* tools rely on the formal structure of Web pages to extract data [Baumgartner et al., 2001b; Crescenzi et al., 2001; Liu et al., 2000; Sahuguet and Azavant, 2001], *natural language processing-based* (NLP) tools extract data of interest from highly grammatical and natural-language documents [Freitag, 2000; Soderland, 1999], *ontology-based* tools recognize and extract data presented in documents given an ontology [Embley et al., 1999], *modelling-based* tools, given a target structure for objects of interest, try to locate in Web pages portions of data that implicitly conform to that structure [Adelberg, 1998; Laender et al., 2002a], and *wrapper induction* tools generate extraction rules from a given set of training examples [Hsu and Dung, 1998; Kushmerick, 2000; Muslea et al., 2001]. There are cases where a tool could fit in two or more of the identified groups.

Chang *et al.* [Chang et al., 2006] surveyed the major Web data extraction approaches and tools and classified them into four classes: *manually-constructed*, *supervised*, *semi-supervised*, and *unsupervised* systems. In manually-constructed systems [Arocena and Mendelzon, 1999; Crescenzi and Mecca, 1998; Hammer et al., 1997; Liu et al., 2000; Sahuguet and Azavant, 2001] users program a wrapper for each Website by hand using general programming languages such as Perl or by using especially-designed languages. Supervised Web data extraction systems [Adelberg, 1998; Califf and Mooney, 1999; Freitag, 1998; Hsu and Dung, 1998; Kushmerick et al., 1997; Laender et al., 2002a; Muslea et al., 1999; Soderland, 1999] take a set of Web pages labelled with examples of the data to be extracted and output a wrapper. Semi-supervised systems require a rough example from users for extraction rules [Chang and Kuo, 2004; Hogue and Karger, 2005]. Unsupervised Web data extraction systems [Arasu and Garcia-Molina, 2003; Chang and Lui, 2001; Crescenzi et al., 2001; Liu et al., 2003; Wang and Lochovsky, 2002, 2003] do not need any labelled training examples and have no user interactions to generate a wrapper. The authors then compared these systems in three dimensions: the *task domain*, the *automation degree*, and the *techniques used*. They proposed some criteria for each dimension and then evaluated the capabilities of the surveyed systems based on these criteria.

Ferrara et al. [Ferrara et al., 2012] categorised Web data extraction approaches into two main categories: approaches based on *Tree Matching* algorithms and approaches based on *Machine Learning* algorithms. They also presented how each category addresses the problems of automatic wrapper generation and maintenance. Tree-based approaches [Baumgartner et al., 2001b; Meng et al., 2003; Sahuguet and Azavant, 1999; Zhai and Liu, 2005] rely on the semi-structured nature of Web pages presented using a labelled ordered rooted tree, usually referred to as *DOM* (*Document Object Model*). Machine-learning approaches [Hsu and Dung, 1998; Kushmerick, 2000; Muslea et al., 2001; Phan et al., 2005] are learning-based Web data extraction systems which require large amounts of manually labelled Web pages. These approaches are used to extract domain-specific data from Web sources, since they rely on training sessions during which a system requires a domain expertise. There are some hybrid approaches [Crescenzi et al., 2001] as well.

Considering the classification given by Laender *et al.*, languages designed for wrapper development require the user to have substantial computer and programming backgrounds, so they are expensive. Moreover, the user has to program a wrapper for each Website by hand [Chang et al., 2006; Ferrara et al., 2012; Laender et al., 2002b]. NLP-based tools are useful to solve specific problems such as extraction of facts from newspaper articles, email messages etc. [Ferrara et al., 2012], and from Web pages consisting of free text [Laender et al., 2002b]. Ontology-based approaches rely directly on the data (not the structure of presentation features of the data) to generate rules or patterns to perform extraction. This approach requires the careful construction of an ontology, a task that must be done manually by a domain expert [Laender et al., 2002b].

Due to the aforementioned limitations of languages for wrapper development, NLP-based, and ontology-based tools, we do not cover and discuss them further in this chapter. However, from HTML-aware, wrapper induction, and modelling-based tools, we choose the most representative tools and discuss their applicabilities to extract variability data from Web configurators. Since Web configurators usually have template-based and dynamically generated Web pages, and the modelling-based approach is a good approach for the extraction of data from Web sources based on templates [Ferrara et al., 2012; Laender et al., 2002b], we mostly focus on modelling-based tools.

STALKER. STALKER [Muslea et al., 2001] is a supervised learning-based wrapper induction tool to extract data from semi-structured Web pages. It presents a Web page in a tree-like structure called *embedded catalog* (*EC*) in which the leaves are the items of interest for the user. The internal nodes of the *EC* represent list of k-tuples. The *EC* tree represents the target document as a sequence of tokens (any piece of text or HTML tag is considered as a token in the document). Having the *EC* tree of the document and a set of training examples, STALKER generates extraction rules that cover the given examples. Extraction rules are described using directives such as *SkipTo*, *SkipUntil*, and *NextLandmark*. For example, *SkipTo (T)* tells that all tokens have to be skipped until the first occurrence of the token *T* is found.

For instance, considering a restaurant description presented in Figure 4.6, STALKER generates the following extraction rule to identify the beginning of the restaurant name, i.e., *Yala*:

R1 = SkipTo(<b>)

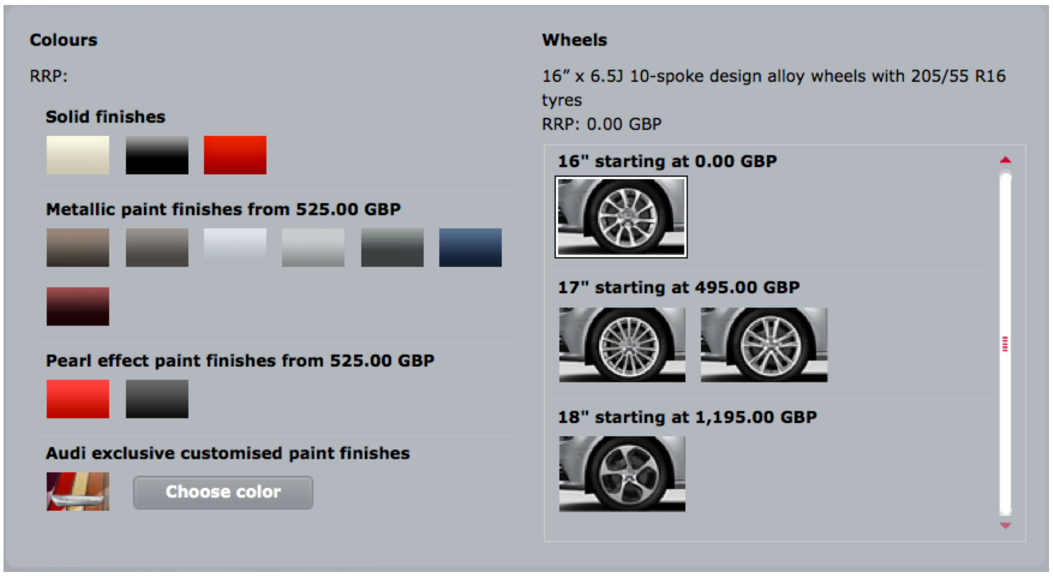
which means start from the beginning of the document and skip everything until the <b> landmark is found.

```
<p> Name: <b> Yala </b><p> Cuisine: Thai <p><i>
4000 Colfax, Phoenix, AZ 85258 (602) 508-1570
</i> <br> <i>
523 Vernon, Las Vegas, NV 89104 (702) 578-2293
</i> <br> <i>
403 Pico, LA, CA 90007 (213) 798-0008
</i>
```

FIGURE 4.6: An example input to STALKER ([Muslea et al., 2001]).

STALKER and other wrapper induction tools generate delimiter-based extraction rules derived from a given set of training examples and rely on formatting features that implicitly delineate the structure of the pieces of data found [Laender et al., 2002b]. These tools assume that the desired data to be extracted are surrounded by common tokens [Chang et al., 2006] and based on these tokens they generate extraction rules. The extraction rules are represented using regular grammars (e.g., regular expressions). We observed that in some Web configurators such common tokens surrounding the data to be extracted can be found. However, in some other cases it is rarely possible to generate a concise and formal grammar to locate the desired data in the page. Figure 4.7 shows an example Web page from a configurator in which options presented using images are not attached any individual textual description. Instead, data items to be extracted are located in the tag attributes of the corresponding HTML elements (e.g., the `src` and `alt` attributes of the `img` elements). STALKER is not able to deal with tag attributes, neither in creating the *EC* tree, nor in analysing training examples to generate extraction rules. It means that in the generated *EC* tree for this example, leaves are HTML tags, not data items. Consequently, STALKER will not extract any data for image-based options. In addition, the authors in [Crescenzi et al., 2001] presented nested structures that STALKER cannot handle.

Data Extraction By Example (DEByE). DEByE [Laender et al., 2002a; Ribeiro-Neto et al., 1999] is a supervised, modelling-based and interactive Web data extraction tool for wrapper generation (Figure 4.8). It targets specific *data rich* Web pages and assumes that there is an implicit structure associated with the objects in the pages. By analysing a set of input example objects taken from a sample Web page, DEByE recognizes the structure of the presented data in the given page and generates *objects extraction patterns* that denote the structure of the data. These extraction patterns are then used by a module called *Extractor* to find and extract new objects from the given page and also from other similar pages.



Colours

RRP:

Solid finishes

Metallic paint finishes from 525.00 GBP

Pearl effect paint finishes from 525.00 GBP

Audi exclusive customised paint finishes

Choose color

Wheels

16" x 6.5J 10-spoke design alloy wheels with 205/55 R16 tyres

RRP: 0.00 GBP

16" starting at 0.00 GBP

17" starting at 495.00 GBP

18" starting at 1,195.00 GBP

FIGURE 4.7: An example of STALKER fail.

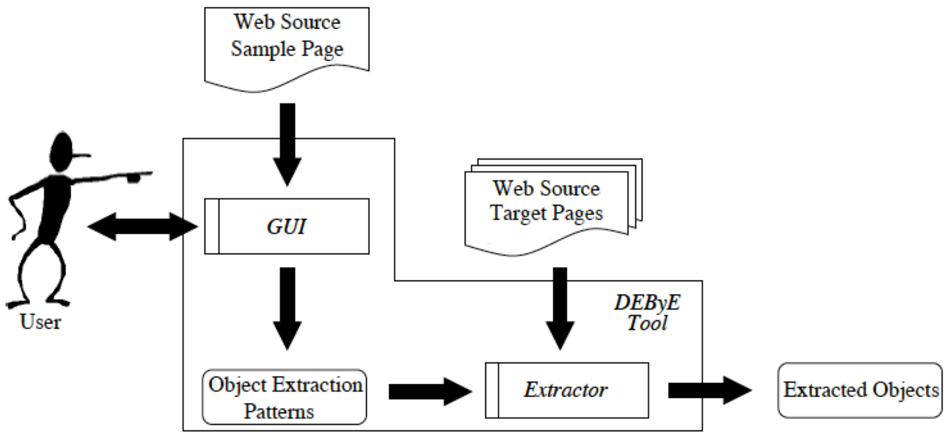


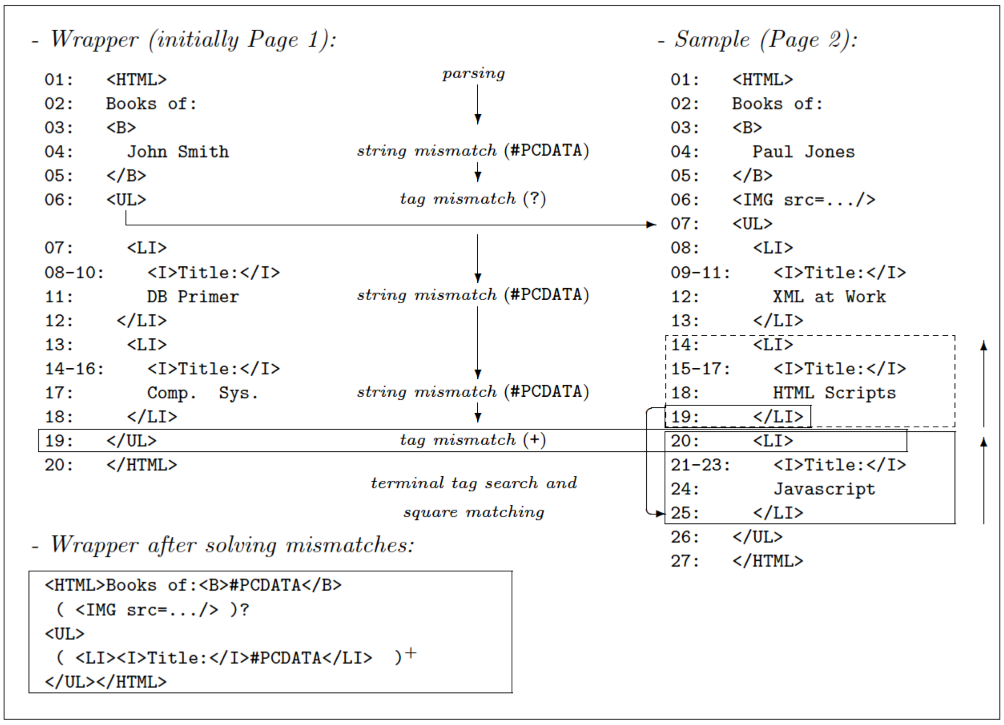
FIGURE 4.8: Modules of the DEByE tool ([Laender et al., 2002a]).

The DEByE approach is restricted to the specific domain of Web applications with data rich pages [Laender et al., 2002a] which contain uniform and regularly formatted data records [Phan et al., 2005]. In contrast to STALKER that also uses HTML tags to identify a common structure for the presented data in the page (and so is applicable to some Web pages in configurators), DEByE exclusively relies on the textual surroundings of the data to be extracted. We observed that inducing such a structure associated with the objects in Web configurators is difficult (if not impossible). It becomes even more challenging when we know that a page in a Web configurator may contain various kinds of data objects, objects with complex structures, objects that are not configuration-specific and so must not be extracted, objects that have no attached textual data items (e.g., images) etc. DEByE is not able to deal with these issues.

Our analysis of Web configurators shows that for data objects presented in the pages that have an identifiable implicit structure (required by DEByE), the structure may be slightly different from one object to another. It means that the data extraction approach needs to be flexible enough to deal with these variations. However, the experimental results in [Phan et al., 2005] show that DEByE fails to recognise and handle such changes (e.g., the change of record layout, order of fields, etc.) because its extraction rules are fixed.

ROADRUNNER. ROADRUNNER [Crescenzi et al., 2001] is an un-supervised Web data extraction system that proposes an approach to wrapper inference for Web Pages. It targets data-intensive Websites in which pages are automatically generated using scripts. A collection of pages in a Website produced by the same script is called a *class of pages*. ROADRUNNER receives a set of (at least two) sample Web pages that belong to the same class, analyses the schema of the data contained in the pages, infers a common structure, generates a wrapper considering the identified structure, and uses that wrapper to extract data from the sample pages as well as from other pages of the same class. The structure discovery is based on the study of similarities and dissimilarities between the given sample pages. ROADRUNNER is able to identify structural features such as tuples and lists, handle structural variations, and resolve string and tag mismatches during parsing of the sample pages. The wrapper generated to describe the identified common structure is presented as a *union-free regular expression (UFRE)*. Figure 4.9 shows two sample pages and the generated wrapper.

ROADRUNNER needs at least two Web pages to generate the wrapper and requires that these pages are generated from a same template. Our observation shows that Web configurators usually use a *single-page* user-interface paradigm to present the configuration space (i.e., all objects representing the configuration-specific data). Even if those follow the *multi-page* paradigm, the structure of the configuration-specific data is different form one page to another. ROADRUNNER will fail to discover a common structure and generate a wrapper for these cases. Another problem of ROADRUNNER is that it is not expressive enough to describe all structures presented in Web pages, so its applicability is limited. For example, ROADRUNNER assumes that Web pages are generated by a union-free grammar, and therefore fails for Websites that use disjunctions (e.g., multi-ordered attributes) [Chang et al., 2006; Crescenzi et al., 2001] , and disjunctions appear frequently in the grammar of Web pages [Lerman et al., 2004]. ROADRUNNER also assumes that any data represented in a page is the extraction target. This poses a serious data accuracy threat for Web configurators because not all data presented in the page is configuration-specific, and therefore, another effort is required to elicit the configuration-specific data from other irrelevant data.



- *Wrapper (initially Page 1):*

```
01:    <HTML>
02:    Books of:
03:    <B>
04:      John Smith
05:    </B>
06:    <UL>
07:      <LI>
08-10:     <I>Title:</I>
11:        DB Primer
12:     </LI>
13:      <LI>
14-16:     <I>Title:</I>
17:        Comp.  Sys.
18:      </LI>
19:    </UL>
20:    </HTML>
```

- *Sample (Page 2):*

```
01:    <HTML>
02:    Books of:
03:    <B>
04:      Paul Jones
05:    </B>
06:    <IMG src=.../>
07:    <UL>
08:      <LI>
09-11:     <I>Title:</I>
12:        XML at Work
13:      </LI>
14:      <LI>
15-17:     <I>Title:</I>
18:        HTML Scripts
19:      </LI>
20:      <LI>
21-23:     <I>Title:</I>
24:        Javascript
25:      </LI>
26:    </UL>
27:    </HTML>
```

- *Wrapper after solving mismatches:*

```
<HTML>Books of:<B>#PCDATA</B>
 ( <IMG src=.../> )?
<UL>
 ( <LI><I>Title:</I>#PCDATA</LI>  )+
</UL></HTML>
```

parsing
string mismatch (#PCDATA)
tag mismatch (?)
string mismatch (#PCDATA)
string mismatch (#PCDATA)
tag mismatch (+)
terminal tag search and square matching

FIGURE 4.9: Two sample pages and the generated wrapper by ROADRUNNER ([Crescenzi et al., 2001]).

XWRAP. XWRAP (XML-enabled Wrapper) [Liu et al., 2000] is an HTML-aware tool for semi-automatic generation of wrapper programs. Figure 4.10 shows the wrapper construction process. The first phase consists of fetching the remote document and repairing bad HTML syntax. This step inserts missing tags, removes useless tags, etc. Once the HTML errors and bad formatting are repaired, the clean HTML document is parsed into a *syntactic token tree.* The usual tokens are HTML tags (paired and singular tags), semantic token names, and semantic token values. In the generated tree, all non-leaf nodes are tags and all leaf nodes are text strings (i.e., semantic token nodes), each in between a pair of tags. The main phase of the wrapper construction process is the *information extraction* phase in which the target document is explored and its structure is specified in a declarative extraction rule language. This phase takes as input the syntactic token tree. It first interacts with the user to identify the semantic tokens and the important hierarchical structure. Then XWRAP annotates the tree nodes with semantic tokens in a comma-delimited format and nesting hierarchy in context-free grammar. Based on the semantic tokens and the nesting hierarchy specification and using a set of data extraction heuristics, XWRAP generates the wrapper code (described in the XWRAP's XML template-based extraction specification language) for the chosen document. The generated wrapper code is tested and once the user is satisfied with the test results, the release version of the wrapper program is obtained.

The wrappers generated by XWRAP can handle only pages where tables are used for layout, therefore its applicability is limited. In addition, XWRAP uses DOM tree paths to address elements in a Web page and assumes that the data to be extracted are co-located in the same path of the DOM tree of the target Web pages [Chang et al., 2006]. It means that the generated wrapper is strictly related to the structure of the page on top of which it is defined. Since the content and the structure of pages in Web configurators may be changed at runtime, this will corrupt the correct operation of the wrapper.

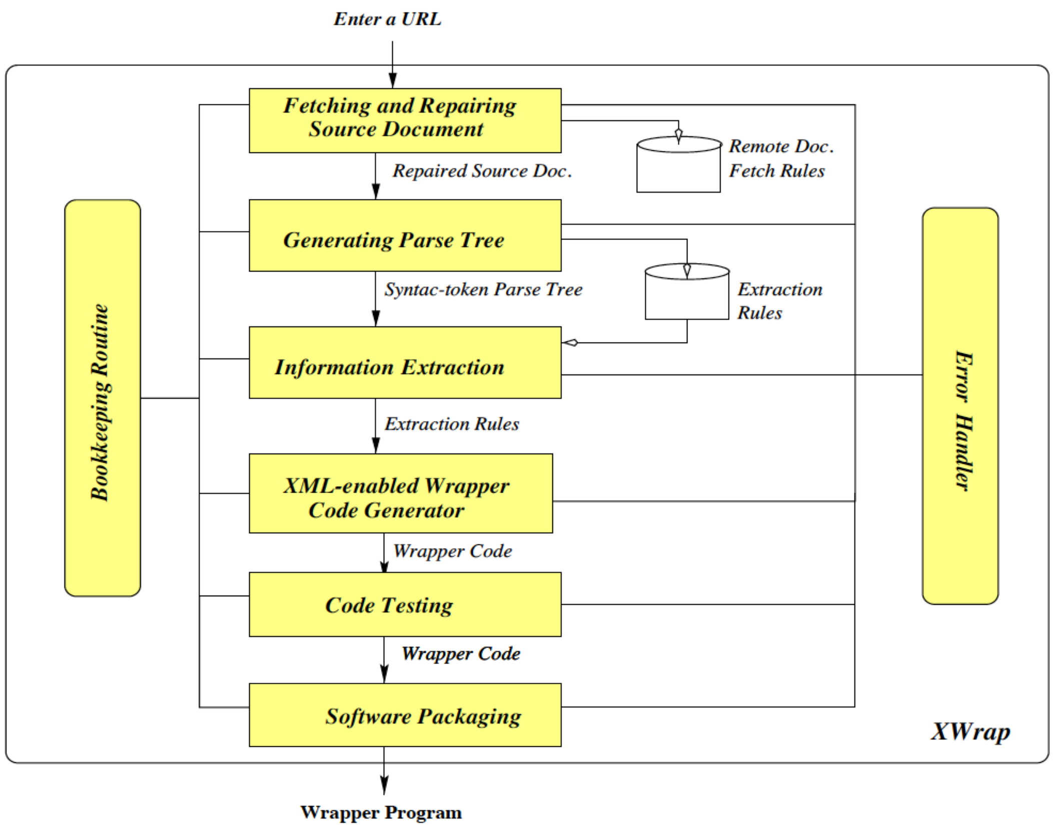


FIGURE 4.10: Data wrapping phases and their interactions in XWRAP ([Liu et al., 2000]).

DEPTA. DEPTA (Data Extraction based on Partial Tree Alignment) [Liu et al., 2003; Zhai and Liu, 2005] is a tree-based un-supervised Web data extraction method. The method targets those Web pages that contain regularly structured objects, called *data records*. DEPTA is based on two observations about data records in Web pages:

- A group of data records that contain descriptions of a set of similar objects are typically presented in a particular region of a page and are formatted using similar HTML tags. Such a region is called a *data region*.

- A group of similar data records being placed in a specific region is reflected in the tag tree[2] by the fact that they are under one parent node.

DEPTA proposes a two-step approach. In the first step, which is called MDR (Mining Data Records), the method segments the page to mine data regions in the given Web page and then identifies data records from each data region without extracting its data items. In the second step, a partial tree alignment algorithm is applied to align and to extract corresponding data items from the discovered data records. The extracted data items are put in a database table.

The drawback of DEPTA is that its recall performance might decay in case of complex HTML document structures. In addition, the functioning of the partial tree alignment is strictly related with the structure of the Web page at the time of the definition of the alignment. This implies that the method is very sensitive even to small changes, that might compromise the functioning of the algorithm and the correct extraction of information [Ferrara et al., 2012]. Since the structure of pages in Web configurators changes at runtime, the maintenance problem arises. Another important issue to note is that DEPTA does not know which regular data records are useful to a user and it simply finds all of them [Zhai and Liu, 2005]. Additional heuristics must be designed to identify and output those that are of interest. Moreover, this method assumes that every HTML tag is generated from the template from which the page is generated and other tokens are data items. However, the assumption does not hold for many collection of pages [Chang et al., 2006]. Finally, this method assumes that (1) exactly the same number of sub-trees must form all records, and (2) the visual gap between two data records in a list is bigger than the gap between any two values from the same record. These assumptions do not hold in all Web pages [Álvarez et al., 2010].

Automatic Web News Extraction. Reis *et al.* [Reis et al., 2004] proposed a domain-specific approach to extract content of news Websites based on the analysis of the structure of their Web pages. This approach relies on the basic assumption that pages in news Websites are generated from a *template*. A template is the set of common layout and format features that appear in a set of Web pages that is produced by a single program or script that dynamically generates the Web page's content. In addition, the proposed approach assumes that news Websites have almost the same organization: (a) a home page that presents some headlines, (b) several section pages that provide the headlines divided in areas of interests (e.g., sports, technology, etc.), (c) pages that actually present the news, containing the title, author, date and body of the news.

[2]The nested structure of HTML tags in a Web page naturally forms a *tag tree*.

Figure 4.11 depicts the main extraction steps. The approach first evaluates the structural similarities between pages in a target Website using a *tree edit distance* algorithm and clusters together pages with similar structure. It then finds a generic representation of the structure of the pages within a cluster. This generic representation is called *node extraction pattern (ne-pattern)*. Taking as input a page cluster, the approach generates a ne-pattern that accepts all the pages in this cluster. A ne-pattern is a rooted ordered labelled tree that contains special vertices called *wildcards*. Every wildcard must be a leaf in the tree and corresponds to a data-rich object in the template from which the page is generated. Once the ne-patterns have been generated, the approach matches the set of generated ne-patterns to the set of crawled pages. For each page, its tree and the tree of the relevant ne-pattern are traversed and for each wildcard found in the ne-pattern its matching text passage is extracted from the page. The extracted data is finally labelled as the title or the body of the news using simple heuristics.

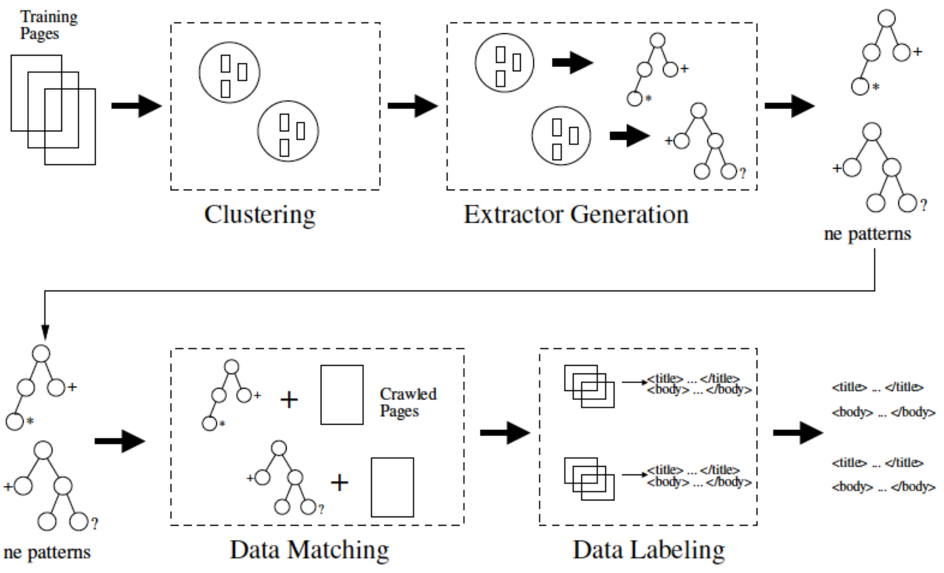


FIGURE 4.11: The main steps of news extraction process ([Reis et al., 2004]).

This approach is domain-specific and is strictly related to the common characteristics and organization of news Websites. Consequently, a substantial effort is required to generalize it to deal with extracting data from Web configurators.

Other related works. Baumgartner *et al.* developed a commercial interactive and visual Web data extraction system called *Lixto* [Baumgartner et al., 2001a,b]. It allows for extraction of target patterns based on surrounding landmarks, on the order of appearance, on semantic and syntactic concepts, etc. These generated patterns are then used to extract new data objects from the given page and also from other similar pages.

Arasu *et al.* [Arasu and Garcia-Molina, 2003] studied the problem of automatically extraction of database values from template-generated Web pages. The proposed approach, called *EXALG*, takes as input a set of template-generated pages, deduces the unknown template used to generate the pages, and extracts as output the values encoded in the pages. EXALG requires more than one training page as input to work. It also extracts data objects in whole pages which may contain records of multiple kinds. Another drawback of this approach is that it does not support multi-ordered attributes [Chang et al., 2006]. And finally, the proposed approach assumes that the data records are represented in a list, are shown contiguously in the page, and are formatted in a consistent manner: that is, the occurrences of each attribute in several records are formatted in the same way and they always occur in the same relative position with respect to the remaining attributes [Álvarez et al., 2010].

Lerman *et al.* [Lerman et al., 2004] proposed an approach to automate the extraction and segmentation of data records from template-generated Web pages. The approach relies on the common structure of many Websites, which present information as a list or a table, with a link in each entry leading to a detail page containing additional information about that item.

Hogue *et al.* [Hogue and Karger, 2005] developed *Thresher*, a system that lets non-technical users teach their browser how to extract semantic content from Web pages. The user specifies examples of semantic content by highlighting them in a browser and describing their meaning. Thresher then uses the tree edit distance between the DOM subtrees of these examples to create a general pattern for the content and allows the user to bind *Resource Description Framework* (RDF) classes and predicates to the nodes of these patterns. The system then matches the generated pattern against the target page by simply looking for subtrees on the page that have the same structure. Each time the system finds a match to the given pattern, the matched text of the subtree is labelled according to its RDF predicate in the pattern.

Zheng *et al.* [Zheng et al., 2009] proposed a record-level wrapper system. First, a set of training pages are converted to DOM trees by an HTML parser. Then, semantic labels of a specific extraction schema are manually assigned to certain DOM nodes to indicate their semantic functions. Based on these labels, an algorithm is applied on each DOM tree to extract records. The extracted records are fed to a module to generate their corresponding wrappers. When a new page enters the system, it is first converted to a DOM tree, and then from the wrapper set generated in the training process, one or more wrappers are automatically selected to align with the DOM tree. Labels on selected wrappers are accordingly assigned to the nodes of the DOM tree. Finally, data contained in those mapped nodes is extracted and saved in an XML file. This approach assumes that records in a Website can be grouped by their tag-paths. In addition, this approach is not able to deal with distinctive data items.

In [Álvarez et al., 2010] the authors presented a set of techniques for detecting structural records in a template-generated Web page and extracting their data values. The method starts by identifying the data region of interest in the page. Then it is partitioned into records by using a clustering method that groups similar subtrees in the DOM tree of the page. Finally, attributes of the data records are extracted by using a method based on multiple string alignment. This method is able to detect and extract lists of structured data records embedded in Web pages. It assumes that the pages containing such list are generated according to the page generation model described in [Arasu and Garcia-Molina, 2003], that is, all instances of an attribute have the same path in the DOM tree, and the same applies to the remaining attributes. The drawback of this approach is that it does not support multi-ordered attributes. Moreover, the authors declared that a limitation of their approach arises in the pages where attributes constituting a data record are not contiguous in the page, and their approach is unable to deal with them.

4.3 Synthesizing Feature Models

Synthesizing a feature model from an existing artefact is the task of locating and extracting features from the artefact, identifying the dependencies exist among features, and then constructing a consistent feature model. In this section, we present several approaches that have been proposed to reverse engineer feature models from existing artefacts.

Reverse Engineering Feature Models. She *et al.* [She et al., 2011] proposed a tool-supported approach for reverse engineering feature models. Given a list of feature names, descriptions and propositional formula specifying dependencies, the task of constructing the relevant feature model reduces to the selection of a parent for each feature in order to build a feature hierarchy. First, using the list of features and the propositional formula, a feature *implication graph* is constructed in which each vertex denotes a feature name and each directed edge indicates a dependency between the participating features. Then, to identify parents, two complementary forms of data are used: (1) dependencies that describe the configuration semantics of the feature model, and (2) descriptions that are used to approximate the feature model's ontological semantics. The authors presented heuristics for identifying the likely parent candidates for a given feature using this data. They also provided automated procedures for finding feature groups, *requires* and *excludes* constraints. The building process provided by this approach is interactive and requires a *domain expert* modeller. The procedures present a list of parent candidates (typically five or less, as shown by experiments) for a feature's parent and the modeller selects the most suitable one. The approach is evaluated on Linux, eCos, and FreeBSD kernels.

On Extracting Feature Models from Product Descriptions. Acher *et al.* [Acher et al., 2012] proposed a semi-automated approach to extract feature models from product descriptions. It is assumed that product descriptions are organized through semi-structured data, typically tabular data where each row of the table specifies a product, and each column has a label that will be used as a feature name in the extracted feature model. Each cell in the table specifies the value of a feature (identified by the label of the corresponding column) for a specific product (identified by the label of the corresponding row). Moreover, the cells may contain variability-specific data. For instance, in a table documenting several Wiki engines, a column (which denotes a potential feature) may contain a list of values for *Licence Cost Fee*. Values such as *US 10*, *Community*, *"Yes"* or *"No"* indicate that this is an optional feature. The authors also developed a language, called *VariCell*, using which the user can programmatically parameterize the extraction process.

Figure 4.12 shows the proposed semi-automated process. The process takes as input a set of product descriptions and directives expressed in VariCell and synthesizes a feature model for each product description. The built feature models are then merged to produce a new feature model that represents all the variability of the set of input products.

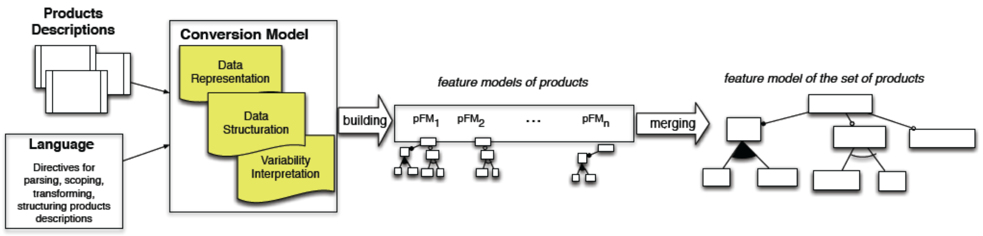

Products Descriptions
Language
Directives for parsing, scoping, transforming, structuring products descriptions
Conversion Model
Data Representation
Data Structuration
Variability Interpretation
building
feature models of products
pFM1
pFM2
...
pFMn
merging
feature model of the set of products

FIGURE 4.12: The process of extracting feature models from product descriptions ([Acher et al., 2012]).

Efficient Synthesis of Feature Models. Andersen *et al.* [Andersen et al., 2012] addressed the problem of automatically synthesizing feature models from propositional constraints. The proposed feature model synthesis takes as input a formula representing a set of feature dependencies or product configurations, and outputs a feature model or a *feature graph (FG)*. A FG is a symbolic representation of all possible feature models.

The synthesis process (Figure 4.13) includes two steps: (a) *Directed Acyclic Graph (DAG)* hierarchy recovery, and (b) group and *cross-tree constraint (CTC)* recovery.

The first step takes as input a formula in either *conjunctive normal form (CNF)* or *disjunctive normal form (DNF)*, and produces a DAG that contains all possible feature model tree hierarchies – possibly with multiple parents for a feature. The second step identifies all feature groups and CTCs given the propositional formula, DAG and an optional tree hierarchy. The output for this step is a FM or a FG.

The authors then used these two steps in three FM synthesis scenarios:

- *Scenario 1:* This scenario describes the process of synthesizing a FG from a set of product configurations represented as a formula in DNF.
- *Scenario 2:* This scenario describes reverse engineering a FM from code. This scenario can be used to build a FM for variability-rich software, such as the FreeBSD kernel. The dependencies among features can be extracted from the source code using static analysis, yielding a formula in CNF.
- *Scenario 3:* This scenario describes binary operations of two FMs, such as merging, diffing, comparing, and slicing. The two feature models are first translated to their propositional formulas, and then an operation is applied to merge the two models, resulting in a single formula. This formula is converted to CNF to serve as input for FM synthesis.

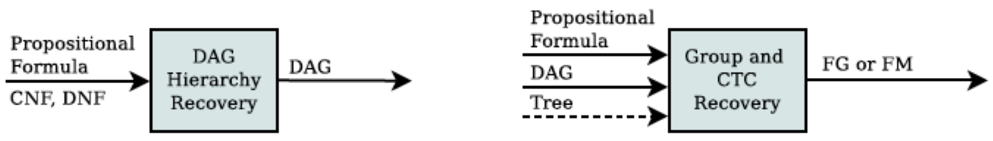


FIGURE 4.13: Components of feature model synthesis ([Andersen et al., 2012]).

Feature Model Extraction from Large Collections of Informal Product Descriptions. Davril *et al.* presented an automated approach for constructing FMs from publicly available product descriptions found in online product repositories and marketing Websites such as SoftPedia and CNET.

The proposed process (Figure 4.14) consists of two main phases. In the first phase, features are discovered and then are used in the second phase to build a FM. The first phase starts with mining product descriptions from online software repositories by using the *Screen-scraper* utility (❶). The extracted product descriptions are processed to identify features and generate a product-by-feature matrix in which the rows correspond to products and the columns correspond to features (❷). Meaningful names are selected for the mined features (❸).

In the second phase, using the product-by-feature matrix, a set of association rules are mined for the features (❹) which are then used to generate an *implication graph (IG)* (❺). The IG is a directed graph in which each node represents a feature and each edge represents a binary configuration constraint between the two participating features. Given the IG and the content of the features, the tree hierarchy and then the FD are generated (❻). Finally, CTCs and *or-groups* are identified (❼), and together with the extracted FD, form the final FM.

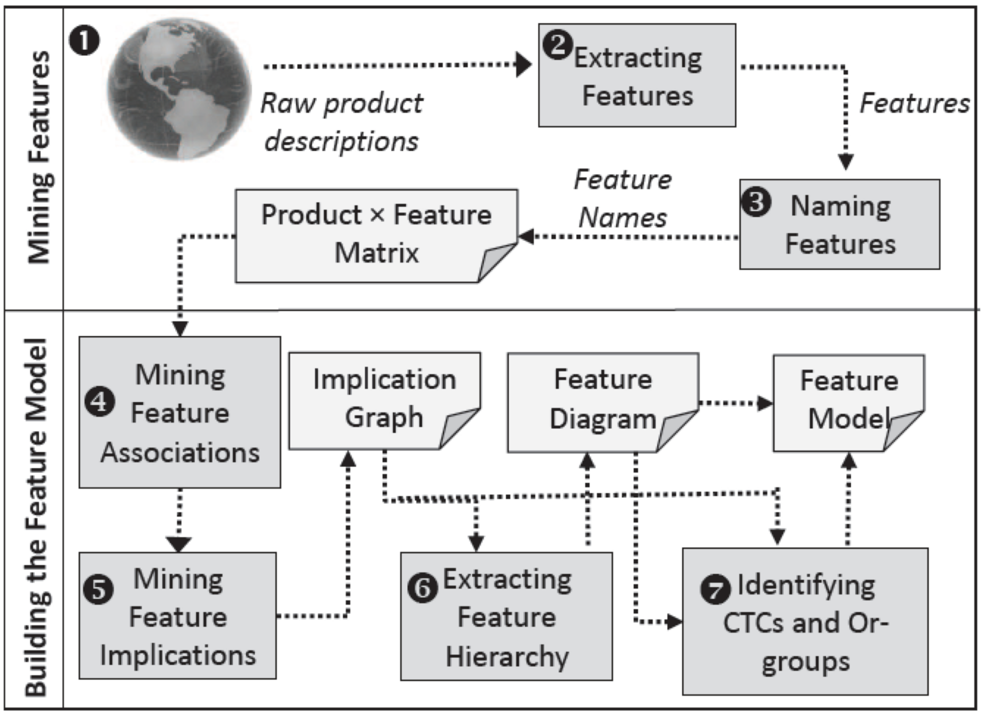


FIGURE 4.14: The two-phase process of mining features and building FM from informal product descriptions ([Davril et al., 2013]).

Reverse Engineering Architectural Feature Models. Acher *et al.* [Acher et al., 2011] proposed a tool-supported approach to reverse engineer software variability from an architectural perspective. The reverse engineered feature model is called *architectural feature model*, noted FM_{Arch}. It is extracted from several software artefacts (files, descriptions, informal documents) and combines different variability descriptions of the architecture of the target software system.

Figure 4.15 shows the process of extracting architectural FMs. The process starts with extracting a raw architectural feature model, noted $FM_{Arch_{150}}$, from a 150% architecture of the system (①). A 150% architecture consists of the composition of the architecture fragments of all the system plugins. The authors call it a 150% architecture because it is not likely that the system may contain them all. $FM_{Arch_{150}}$ thus includes all the features provided by the system.

To derive inter-feature constraints from inter-plugin constraints, the specification of the system plugins and their declared dependencies are analysed and based on this information a *plugin feature model* FM_{Plug} is built (②). Then, the bidirectional mapping that holds between the features of $FM_{Arch_{150}}$ and those of FM_{Plug} is reconstructed (③). Finally, this mapping is used to derive FM_{Arch}, where additional constraints have been added.

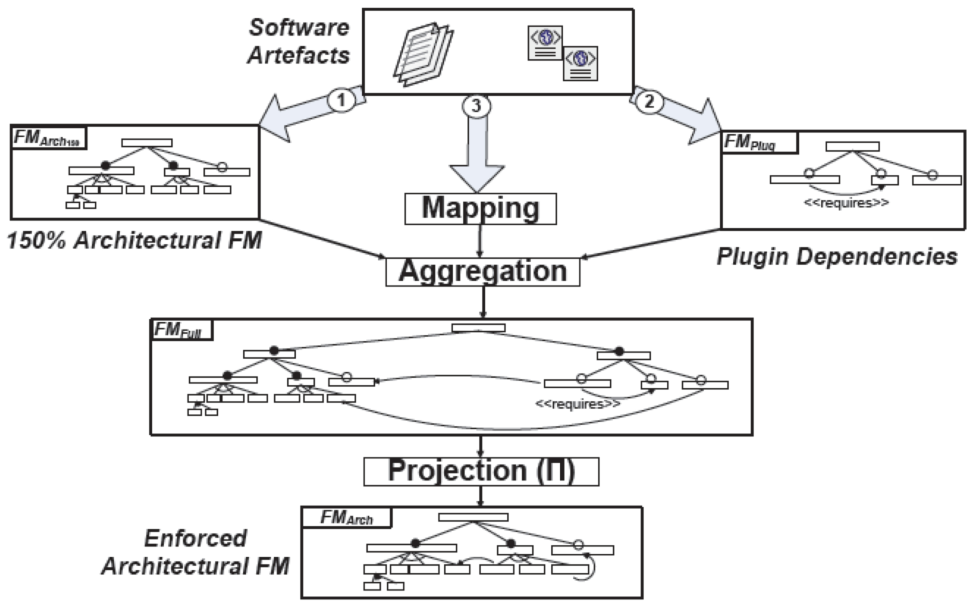


FIGURE 4.15: The process of extracting architectural FMs ([Acher et al., 2011]).

Other related works. Lora *et al.* [Lora-Michiels et al., 2010] proposed a method to construct *Product Line Models (PLMs)* by exploiting mining techniques (e.g., apriori algorithm, independence test, etc.) to identify candidate features, group cardinalities, and dependencies. The method starts with a collection of *Product Models (PMs)* and produces PLMs in the FORE [Streitferdt, 2004] notation. It arranges features of the collection of product models into a matrix of occurrences. Then, the process guides the construction of the general tree architecture by detecting candidate parent-child dependencies, mandatory and optional relationships, and completes it with group cardinalities. It also guides the identification of other dependencies such as requires and excludes.

In [Weston et al., 2009] the authors introduced a tool suite which automatically transforms natural-language requirement documents into a candidate feature model, which can be refined by the requirements engineer. The authors consider features as clusters of related requirements. Statistical methods are used to measure the similarity between the texts of the requirements and similar requirements are clustered together. The features (i.e., the clusters) are structured in a tree based on their similarity. The user can add, remove, and manually name the extracted features.

John [John, 2006] proposed an approach called *PuLSE-CaVE* (Commonality and Variability Extraction) for the extraction of requirements from user documentation, which gives guidance on how to elicit knowledge from existing user documentation and how to transform information from this documentation into product line models.

Alves *et al.* [Alves et al., 2008] proposed a framework for identifying commonalities and variabilities in requirement specifications for software product lines of a given domain. The framework takes as input a set of documents, where each document comprises requirements specifications of different applications. Then, for each such document, information retrieval (IR) techniques are used to automatically determine a similarity relationship between its requirements. Next, based on this relationship, clusters of requirements are identified and abstracted further into a configuration. Finally, the configurations corresponding to all requirement documents are merged into a fully-fledged feature model.

Ziadi *et al.* [Ziadi et al., 2012] presented a three-step approach to feature identification from source code of software products. In the first step, an abstract model is reverse engineered from the source code of each product by reducing the noise induced by spurious differences in the various implementations of the same feature. In the second step, feature candidates are produced by identifying pieces of software that appear identical in the available products. The proposed algorithm works on product abstraction induced by the first step. In the third step, the irrelevant candidates are manually pruned and missed features are added.

Haslinger *et al.* [Haslinger et al., 2011] presented an algorithm that reverse engineers a basic feature model from the feature sets which describe the features each system provides in a family of systems.

4.4 Conclusion

Reverse engineering Web applications. Existing methods to reverse engineer Web applications mostly focus on recovering models at a high level of abstraction, at GUI (presentation) and sometimes business levels. They do not target the data layer where most of the configuration-specific data resides. 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) nor specific properties of the highly dynamic and multi-step nature of a configuration process (e.g., choices may force the selection of/exclusion of some other options, make visible new options or even new steps).

Web data extraction. Most of the available Web data extraction methods are domain-specific as constructing a generic method is complex (if not impossible) [Reis et al., 2004]. Moreover, their scalability has not been adequately evaluated to verify if they can be applied across different application domains (see [Ferrara et al., 2012] for a brief discussion on this possibility). We conclude that existing Web data extraction methods do not meet three important requirements we face when reverse engineering Web configurators:

- Most of these approaches usually aim at providing automatic data extraction techniques. Consequently, they assume that meaningful naming of the extracted data is done as a post-processing task (notable exceptions are [Hogue and Karger, 2005; Zheng et al., 2009]). This poses a serious data accuracy threat for Web configurators, because: (1) not all data presented in a page is configuration-specific. Data not relevant should be ignored for reverse engineering; (2) each data item should be labelled meaningfully (option name, description, price, constraint, etc.). Therefore, another effort is required to elicit and properly name the configuration-specific data from otherwise irrelevant data.

- Existing Web data extraction approaches neither consider specific characteristics of Web configurators nor study the configuration semantics and relationships between extracted data. For instance, for options presented in a group and represented using radio buttons, not only the options should be extracted, but the fact that an *alternative* group constraint defined over these options exists should be documented as well. This means that in addition to the data presented in the pages of a configurator, some data must be deduced from the presentation and documented.

- Web configurators are highly interactive applications: as they are executing, new content may be automatically added to the page, and existing content may be removed or changed. A technique is required to automatically generate and extract this data. Existing Web data extraction approaches do not provide sufficient support for generation and extraction of dynamic content. Some available Web data extraction and reverse-engineering approaches offer a Web crawler to navigate hyper-linked Web pages and extract data. In Web configurators, a more specific Web crawler is required to be able (1) to automatically explore the "configuration space" (i.e., all object representing configuration-specific content), (2) to simulate the users' different configuration actions in order to automatically generate dynamic content, and (3) to track and detect changes made to the page with the aim of identifying and extracting newly added configuration-specific data through dynamic analysis. For instance, for options in the "Model line" group in Figure 3.1 (Section 3.1), the Web data extraction method should select each option, extract new options loaded in the "Body style" group, and record that there exist cross-cutting constraints between these options.

Synthesizing feature models. The approaches that address the (semi-automatic) reverse engineering feature models use sources such as user documentation, natural-language requirements, formal requirements, product descriptions, dependencies, source code, architecture, etc. to recover feature models. None of these approaches tackles the extraction of variability data from Web configurators.

In the light of these observations, we argue that a new method is needed. This new method is described in the next chapter. Wherever relevant, the new method takes inspiration from existing techniques. More precisely:

- The *variability data extraction pattern* language we developed takes inspiration from the notion of *node extraction pattern* (*ne-pattern*) [Reis et al., 2004] for its general structure, but it is substantially different in both syntax and semantics (see Chapter 6). Moreover, we implemented a different pattern-matching algorithm to locate in a Web page code fragments that structurally conform to a given pattern (see Chapter 7).

- In our approach, a data extraction procedure, i.e., the Wrapper, traverses the code fragments in the source code and extracts their data items. During the traversal, some HTML elements must be ignored by the Wrapper because they do not hold any variability data. We use a directive similar to the *SkipTo* directive presented in [Muslea et al., 2001] to guide the Wrapper to skip the noisy elements (see Section 6.3.2). The user can set up the Wrapper to skip a predefined number of consecutive sibling elements, or to ignore all descendants of an element until finding a specific HTML or text element.

- Inspired by the *robot* component of CRAWLJAX [Mesbah et al., 2012], we implemented a Web Crawler to simulate the users' configuration actions (see Chapter 8). Similarly, in our approach, the user tells the Crawler which elements should be clicked to simulate the user actions. However, unlike the *robot*, if the Crawler has changed the state of an element, it can change it back to the previous state. For instance, if the Crawler selected a check box, it can change back its state to *undecided*.

- Zheng *et al.* [Zheng et al., 2009] proposed to manually assign semantic labels of a specific extraction schema to certain DOM nodes to indicate their semantic functions. Based on these labels, an algorithm is applied on each DOM tree to extract records. Similarly, in our approach, the user uses a variability data extraction pattern to mark data objects of interest to be extracted from a page. Moreover, using the pattern, the user defines the structure of objects of interest. Finally, in our approach, the user can mark data items of interest that exist in the tag attributes of an HTML element.

However, to a large extent, the approach is novel as needed by the specificities of the domain of Web configurators. The major innovations we introduce are:

- A pattern specification language that can be used to identify and manage the implicit templates (structure and layout of data) followed in Web development.
- A novel pattern-matching algorithm to locate code fragments in the source code of a page that structurally match a given pattern.
- A technique for analysing the dynamics and the specificity of a configuration process.
- A set of methods to trigger, identify, and extract constraints defined over options in a Web configurator.

Chapter 5

The Reverse Engineering Process

Our investigation of existing approaches shows that none of those tackles the problem of extracting variability data from Web configurators (Chapter 4). The adaptation of these approaches to reverse engineer feature models from Web configurators requires substantial changes to their core procedures because the algorithms they implement do not consider specific properties of Web configurators. We conclude that we need a different approach.

This chapter presents our tool-supported reverse engineering solution for extracting configuration options, their associated descriptive information, and constraints, altogether called *variability data*, from the Web pages of a Web configurator, and then constructing a feature model [Abbasi, 2013; Abbasi et al., 2014]. The proposed supervised and semi-automatic reverse-engineering process implements a sequence of interactive and automatic activities to obtain a feature model by static and dynamic analyses of the client side of a Web configurator. The input for the reverse engineering process is a set of *variability data extraction patterns*, expressed in an HTML-like language to specify variability data to be extracted from a Web page, and the output is an XML file containing the extracted data represented in a predefined data model. We first describe the main reverse-engineering challenges (Section 5.1). We then present our reverse-engineering process and briefly explain its activities (Section 5.2). The chapters ahead elaborate the activities individually in great detail.

5.1 Main Challenges

5.1.1 Designing a generic Web data extraction approach adapted for configurators (RQ2)

The first challenge in designing an efficient Web data extraction approach is its *generality*. It should to deal with variations in data presentation in the heterogeneous pages of Web configurators. Although strong domain knowledge is needed, a convincing approach for the extraction of data from dynamically template-generated Web pages is a *modelling-based* approach in which given a target structure for data objects of interest tries to locate portions of data that implicitly conform to that structure [Ferrara et al., 2012; Hogue and Karger, 2005; Laender et al., 2002a,b; Zheng et al., 2009]. We use templates in reverse order to extract configuration-specific data encoded in the code fragments generated using these templates. Given a template, specified using a set of *variability data extraction patterns* (*vde* patterns), the Web Wrapper seeks to find code fragments that *structurally* match the template and then extract their data.

The second Web data extraction challenge is the *accuracy* of the extracted data. From all data presented in a page, only configuration-specific data must be extracted and labelled. Most of the existing (automatic) approaches assume that labelling (or naming) the extracted data is done as a post-processing step or they produce data without labels. In our approach, the user manually marks and names data to be extracted by giving it a meaningful label in a *vde* pattern specification. Consequently:

- The user distinguishes configuration-specific data from the other irrelevant and noisy content; technically, the Wrapper considers the data marked by the user and ignores the rest.
- The user explicitly and accurately organizes data items in the extracted data records by assigning them different labels.
- Representing the extracted data in a predefined data model becomes feasible, because the types and logical relationships of data to be extracted from pages of Web configurators are mostly known. We specified a data model (see 7.2.1) that defines the schema of the extracted data from Web configurators. Therefore, our Web data extraction system produces homogeneous structured data from unstructured or semi-structured Web pages of heterogeneous Web configurators.

A template-generated Web page contains a set of *template instances*, which are HTML code fragments in a page that are generated from a same template. Template instances are *syntactically* identical code fragments except for variations in values of data slots (text elements and tag attribute values) as well as minor changes to their structure. We take advantage of templates used in generating Web pages in reverse order to extract the required data. Our main proposal is the notion of *variability data extraction pattern (vde pattern)*, supported by an HTML-like language to specify templates. A *vde* pattern specifies (1) which configuration-specific data items from (2) which code fragments of the Web page will be extracted. For the former, the pattern marks text elements and attributes carrying the content of interest, and for the latter, it defines the structure of code fragments (in fact, template instances) that may contain the marked data. The Wrapper takes as input the specification of a set of *vde* patterns (all contained in a *configuration file*) and a Web page, seeks and finds code fragments in the source code of the page that structurally match the given patterns, and extracts as output data items (represented in the predefined data model) from those code fragments corresponding to the marked data in the patterns.

Several approaches attempt to automatically deduce the implicit and unknown templates used to generate the pages, and then use these templates for data extraction [Arasu and Garcia-Molina, 2003; Hogue and Karger, 2005; Reis et al., 2004] (Section 4.2). However, they are either inaccurate or make many assumptions [Zhai and Liu, 2005] (e.g., on the structure of the data presented in the pages). They usually assume that there is a structure associated with all the objects presented in a page, meaning that there is only one template from which almost all the data objects are generated. They conduct an automatic template-induction process to mine this implicit template. A page in a Web configurator may contain various kinds of data objects generated from several templates and therefore deducing a generic structure covering all these templates is a complex, or even impossible, task. Consequently, a fully automated Web data extraction approach in the context of Web configurators is neither realistic nor desirable. We consider that the data extraction process should be supervised. The user inspects the source code of the pages of a configurator, finds out templates used to generate configuration-specific objects, and then writes appropriate patterns to specify those templates to extract data from their instances.

The start element of a pattern specification is a `pattern` element. Each pattern is given a unique name in the `data-att-met-pattern-name` attribute of the pattern element in the configuration file. We consider three types of patterns:

- A *data* pattern specifies the structure of code fragments representing the data of interest and marks data items to be extracted from them.

- A *region* pattern highlights a portion of a page. It specifies within which part of the source code, code fragments may match the given data pattern and thus where the Wrapper should operate .
- An *auxiliary* pattern extends the specification of a data pattern.

The Wrapper requires the specification of at least one region pattern and one data pattern to operate. The type of a pattern is defined in the `data-att-met-pattern-type` attribute of the pattern element.

Figure 5.1 presents the specification of *vde* patterns to extract options from the page shown in Figure 3.6. The region pattern (lines 12-16) guides the Wrapper to look for code fragments that may structurally match the *equipment* data pattern (line 14) in the region starting with the `<table class="SectionCheckBoxList">` element (line 13). The data pattern (lines 1-11) defines the structure of the template used to generate options in the page in Figure 3.6. It also defines that from each matching code fragment three data items will be extracted:

- The string value of the immediate child text element of the element `<span class="SectionText">` will be extracted and labelled as `data-tex-mar-option-name` (line 4).
- The string value of the immediate child text element of the element `<li>` will be extracted and labelled as `data-tex-mar-require-option` (line 6).
- The string value of the immediate child text element of the element `<span class="SectionPrice">` will be extracted and labelled as `data-tex-mar-price` (line 9).

Note that the element `ul` is an *optional* element (line 5), meaning that it may be missing in some code fragments. The attribute `data-att-multiplicity="[0..1]"` denotes the optionality of this element (and obviously its descendant elements). Therefore, for some code fragments there will be no data named `data-tex-require-option`. The multiplicity of the `li` element (line 6) is defined to be `1..*`, which means that the Wrapper should seek for one or more `li` elements in the matching code fragments having the `ul` element.

5.1.2 Developing a Web crawler (RQ3)

While the previous section targets the static aspect of Web configurators, this section focuses on their dynamic aspect. We aim to provide a technique to automatically generate dynamic configuration-specific content. The two actions that usually add dynamic

```
<pattern data-att-met-pattern-type="data" data-att-met-pattern-name="equipment">
    <input  type="checkbox" data-att-met-clickable= "true"  data-att-met-unique="true"/>
    <label>
       <span class="SectionText"> data-tex-mar-option-name
          <ul data-att-met-multiplicity="[0..1]">
             <li data-att-met-multiplicity="[1..*]">data-tex-mar-require-option</li>
          </ul>
      </span>
      <span class="SectionPrice">data-tex-mar-price </span>
    </label>
</pattern>

<pattern data-att-met-pattern-type="region" data-att-met-pattern-name="equipmentRegion">
    <table class="SectionCheckBoxList">
       <pattern>equipment</pattern>
    </table>
</pattern>
```

FIGURE 5.1: The configuration file containing the specified *vde* patterns to extract options from the page shown in Figure 3.6.

configuration-specific content are the exploration of the configuration space and the configuration of options, both called *crawling* actions. For instance, when the user activates a configuration step (e.g., by clicking on a menu containing the step's options) its options are loaded in the page (Figure 3.6). Configuring an option may also dynamically generate new content and add it to the page (e.g., in the "Model" step in Figure 3.5, the selection of an option in a group loads new options in another group), or may change the configuration state of existing options (e.g., in the "Equipment & Options" step in Figure 3.6, the selection of the "Climate Pack" option implies the selection of "Automatic lights and wipers" and "Duel zone climate control", consequently, their corresponding check boxes are checked).

Manual exploration of a large-scale configuration space and configuration of all options in order to generate and extract dynamic configuration-specific data is tedious and error-prone. On the other hand, devising a generic and fully automatic crawling technique is not realistic, because Web configurators use different GUI paradigms, business-logic-management policies, and data presentations. Therefore, we developed a *Web Crawler* that automatically explores a simple configuration space and simulates some of the users' configuration actions. If the exploration and the configuration actions add new data to the page, the Wrapper extracts the newly added data.

5.2 The Reverse Engineering Process

Figure 5.2 depicts our proposed *supervised* and *semi-automatic* approach to reverse engineer feature models from Web configurators. Interactive (I) and automatic (A) activities are distinguished. We now describe the activities in detail.

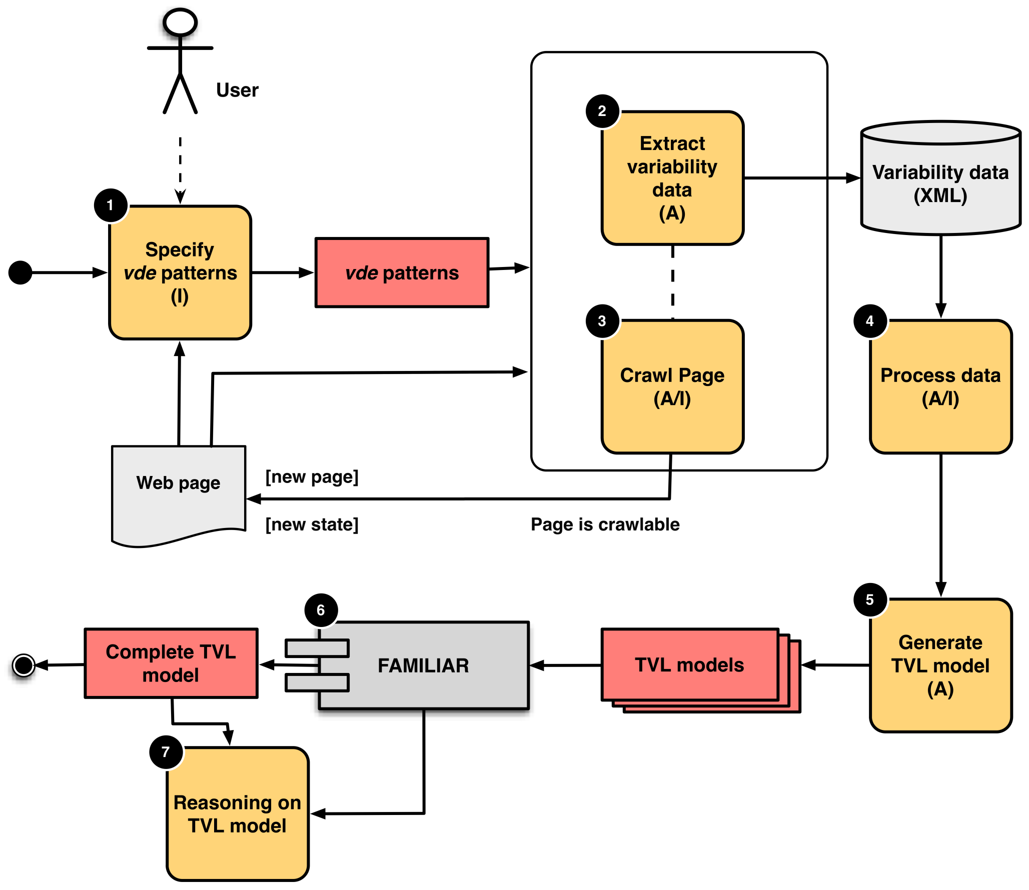


FIGURE 5.2: The Reverse Engineering Process.

Specify *vde* patterns (❶). The process starts with the specification of *vde* patterns for a given Web page. The user inspects the source code of the page, identifies templates from which the data objects of interest are generated, specifies the appropriate *vde* patterns defining the structure of those templates, and marks the required data in the patterns. All 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.

Extract variability data (❷). Once that the *vde* patterns are interactively defined by the user, they are given to the Web Wrapper. Given a configuration file containing the specified patterns 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. From the matching code fragments, the Wrapper extracts data items corresponding to the marked data in the data pattern.

The Web Wrapper implements a *source code pattern matching* algorithm to find matching code fragments. Our proposed algorithm provides a two-step solution to find mappings between elements of a code fragment and the given pattern using their tree representations. The algorithm first uses a *bottom-up* tree traversing to find *candidate* code fragments that may match the given data pattern. Note that a data pattern may be optionally extended by a set of auxiliary patterns. Once the candidate code fragments are identified, the algorithm uses a mixture of both *depth-first* and *breadth-first* traversals to find 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 excluded from the data extraction process.

Crawl Web page (❸). The whole configuration space is not presented in the currently loaded page. It 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). Note that, multi-page and single-page paradigms are not mutually exclusive and a Web configurator may use both. We observed that configurators following the multi-page paradigm usually consist of a relatively small set of Web pages and the user can manually explore them and run the data extraction process for each page (i.e., specifying *vde* patterns and extracting variability data).

For configurators following the single-page paradigm, we observed two common scenarios. In the first scenario, 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. Configuring an option and exploring configuration steps are common actions to change the content of the page. By configuring an option its implied options are loaded in the page. For instance, the selection of an option in the "Model line" group in Figure 3.5 loads new options to the "Body style" group. The activation of a step makes available/visible its contained options in the page and makes unavailable/hidden those of other steps. In the second scenario, all configuration options are loaded in the configuration space. However, by configuring an option, the configuration state of other impacted options is changed. For instance, in the "Equipment & Options" step in Figure 3.6, the selection of the "Climate Pack" option implies the selection of "Automatic lights and wipers" and "Duel zone climate control".

To generate and extract dynamic data we need to automatically crawl the configuration space in a Web page. Automatically crawling requires (1) the simulation of the users' configuration and exploration actions to systematically generate new content or alter the exiting 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 the 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.), both called a *clickable* element. The clickable element to be considered by the Crawler is identified in the *vde* pattern by the `data-att-met-clickable = "true"` attribute. Once the Wrapper has treated a matching code fragment and extracted its data, the Crawler looks for a clickable element in the pattern specification, and if it finds it, it identifies the mapping clickable element in the matching code fragment and simulates its click event. For instance, in the pattern specification given in Figure 5.1, the check box elements are marked as clickable elements (line 2). So, the Crawler clicks on each check box element shown in Figure 3.6.

The simulation of user actions may change the content of the page and move the page to a new state. Therefore, after simulating every clickable element, the page's content is analysed by the Wrapper to identify the newly added content and to deduce from that the configuration-specific data (❷).

The extracted data is hierarchically organized in a predefined data model and serialized using an XML format. The data model reflects the structure of variability data to be extracted from a Web page and is independent from the structure of the page presenting this data.

Process data (❹). Once the data has been extracted, it is further analysed. The following manual/semi-automatic activities are performed to achieve an accurate data and a complete model:

- **Add data.** The user may need to add more data to the automatically extracted data. It can be either *missing* data that the Wrapper and the Crawler could not identify and extract it, or *complementary* data that the user adds to achieve completeness and accuracy in creating feature models. For instance, the user may add a new parent option to categorize already extracted options in a group.
- **Remove noisy data.** An extracted data (or a portion of it) may not be relevant from a configuration perspective and should therefore be removed. For instance, we observed that in some configurators the price data items of options are prefixed with the term *price:* (e.g., *price: $25*) which may be removed by the user.
- **Build option hierarchy.** Our data extraction tool has to some extent the capability of identifying and documenting the hierarchical relationship between

options, i.e., their parent-child relationship. In some cases, either due to limitations of the approach or because the user did not set up the tool to record data on option hierarchy, the user may manually construct a hierarchy.

Generate TVL model (❺). The clean XML file is then given to a module (written in Java) which transforms it into a feature model represented in TVL [Classen et al., 2011b]. The module creates a TVL model for each XML file.

Reasoning on feature models (❻ and ❼). At the end of the reverse engineering process of a configurator there are typically several generated XML files (e.g., each corresponding to a specific configuration step), and accordingly several TVL models. To produce a fully-fledged TVL model, all these models are fed to *FAMILIAR* [Acher et al., 2013b], a tool-supported language to merge partial feature models into a single final feature model.

FAMILIAR also provides other useful feature model analyses. For instance, it can compute the *differences* between two feature models. Using this technique we are able to compare the model generated by our approach to the one generated by the expert to validate the accuracy of the extracted models.

5.3 Chapter Summary

In this chapter, we presented our supervised and semi-automated process to reverse engineer feature models from a Web configurator by static and dynamic analyses of the Web pages. We continue this dissertation with the presentation of the syntax and semantics of *vde* patterns in Chapter 6, and the data extraction procedure in Chapter 7. We then move on to Chapter 8 which is dedicated to the description of the proposed crawling technique. The evaluation of our reverse engineering process is presented in Chapter 9.

Chapter 6

Variability Data Extraction Patterns

In our approach to reverse engineer feature models from Web configurators we consider that the data extraction process should be supervised. We propose the notion of *variability data extraction pattern* (*vde* pattern in short) using which a user manually marks and names variability data to be extracted from the Web pages of a configurator. The user can specify a *vde* pattern, expressed in an HTML-like language, to define the structure of objects of interest and to mark data items to be extracted from those objects. A *Web Wrapper*, given a *vde* pattern, tries to locate in a Web page code fragments (presenting objects of interest) that *structurally* conform to that pattern, and extracts as output data items from those code fragments corresponding to the marked data in the pattern. The extracted data is hierarchically organized and serialized using an XML format.

We start this chapter by explaining observations based on which *vde* patterns are proposed (Section 6.1). We then present some preliminary definitions (Section 6.2) to clearly define the basics and concepts which are necessary to understand the rest of the chapter. At the main part of this chapter, we introduce *vde* patterns and describe their syntax and semantics using examples (Section 6.3). We then represent a context-free grammar for the syntax of *vde* patterns (Section 6.4).

6.1 Observations

Due to the heterogeneous nature of the Web, Web data extraction systems are rather domain-specific [Reis et al., 2004]. It means that the specific characteristics of the domain should be carefully studied and considered when developing such tools. Our analysis of the client-side source code of a sample set of Web configurators shows that Web objects representing variability data are usually generated from a number of *templates*. We think of a template as an HTML code fragment that defines the structure and layout of data to be visually presented in a page. In a template, text elements and tag attributes are data slots filled by *data instances* when generating the page. Each Web page contains a set of *template instances*, which are syntactically identical fragments except for variations in some values for data slots as well as minor changes to their structure. A Web page may be generated from several different templates.

We also observed that tag attributes play a dual role in a template. One the one hand, they can be data slots, therefore, their corresponding values should be extracted from template instances. For example, values of the `src` and `title` attributes in an `img` element might be of interest to a user. On the other hand, a tag attribute might be an *invariable* part of a template specification, and therefore, all template instances have the same value for this attribute. Consequently, this attribute can be used by a data extraction procedure to find instances of a template. For example, <`td class="optionName"`>`Space Gray Metallic` </`td`> shows that the child text element of the <`td class="optionName"`> element is the name of an option. Thus, we can extract option names by finding all `td` elements having the `class="optionName"` attribute and reading the value of their child text element.

Another observation is that in a Web page of a configurator, the configuration environment is divided into a number of *data regions* and each region contains a subset of configuration-specific data objects (which is in line with [Liu et al., 2003]). Configuration steps and options groups are examples of regions in Web configurators. Objects contained in a region are likely generated from a same template. A data region is usually identified with a special element. For example, <`div id="selection"`> is the parent element of a data region containing a set of options.

Based on these observations, we propose a method to extract variability data from Web pages of a configurator. We take advantage of templates used in generating Web pages in reverse order to extract the required data. Our main proposal is the notion of *variability data extraction pattern* (*vde* pattern) to specify templates and also to mark data required by a user. Then, a data extraction procedure seeks and finds code fragments (in fact, template instances) that structurally match the given pattern and extracts from them values marked in the pattern.

Among all information presented in the Web pages of a configurator, we mainly aim at extracting the following data, altogether called *variability data*:

- *Configuration options*: Users gradually select the options to be included in the final product.
- *Descriptive information*: Additional information associated to an option such as its price, size, using instructions, etc.
- *Constraints*: A constraint determines valid combinations of options to specify a valid product.

Moreover, we are sometimes able to extract the following complementary configuration-specific data:

- *Group:* Grouping is a way to organize related options together. A group is identified with a name.
- *The configuration process:* A process is a sequence of configuration steps that the user follows them to complete the configuration of a product.
- *Optionality of 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).

We also need to extract some data items that do not present variability data but can be used to deduce this data. For instance, for each option we extract the widget type used to represent the option. We then use the widget types of grouped options to deduce the *group constraint* defined over these options.

6.2 Preliminary Definitions

This section introduces a number of preliminary definitions used throughout this chapter.

Definition 6.1: HTML Code Fragment An HTML code fragment is a *valid* and *well-formed* sequence of HTML code lines that conform to the HTML specification given in [HTML5, 2014]. A code fragment is defined as a seven-tuple: $C = \langle\, V, A, S, E, R, \lambda, \rho\rangle$, where V is the nonempty set of HTML elements and each $v \in V$ corresponds to an opening and closing pair of tags (some HTML elements do not have the closing tag), A is the set of zero or more tag attributes, each coming in the form of `name="value"`, S is the set of zero or more text elements, each containing a single string value, E is the set of zero or more parent-child relationships between V and S, R is the nonempty set of root elements and $R \subseteq V$, λ is a function that assigns each HTML element a string label:

- For each $x \in V, \lambda(x) =$ the tag name of x,
- For each $x \in S, \lambda(x) = x$, meaning that string label of a text element is the same as its text value,

and ρ is a function that gives each HTML element a set of attributes: $\rho : A \to V$.

Definition 6.2: Ordered Tree (from [Nierman and Jagadish, 2002]) An ordered tree T is a rooted tree in which the children of each node are ordered. If a node x has k children then these children are uniquely identified, left to right as x_1, x_2, ..., x_k.

Definition 6.3: Labeled Tree (from [Nierman and Jagadish, 2002]) A labelled tree T is a tree that associates a label, $\lambda(x)$, with each node $x \in T$. $\lambda(T)$ denotes the label of the root of T.

Definition 6.4: HTML Tag Tree An HTML tag tree of a code fragment with a single root element is a *rooted ordered labeled* tree in which each node is either

- an *element* node corresponding to an HTML element and is labeled with the name of the HTML element in the code fragment, or
- a *text* node corresponding to a text element and is labeled with *#text*[1],

and the tree hierarchy represents the nested structure of HTML elements (parent-child relationships) forming the code fragment. A text node is a leaf node and so has no children. If C is a code fragment and T is an HTML tag tree, we represent T_C to denote that T is the corresponding tree structure of C.

We note that since an HTML tag tree is defined to have only one root node and a code fragment may have more than one root element, a code fragment might be represented as a *forest* of HTML tag trees in the case of having two or more root elements.

Definition 6.5: Data Item A data item is a unit of data describing a meaningful value. For instance, a configuration option name, a step name, a price value, etc., are considered data items in our context. A data item is modelled as (the substring of) the value of a text element or a tag attribute. Let di be a data item, then $di \in S \cup A_v$, in which S is the set of text elements and A_v is the set of values of tag attributes in a code fragment.

Definition 6.6: Data Record or **Data Object** A data record (DR) consists of one or more related data items of a specific object in a Web page: $DR = \{di_1, di_2, ..., di_n\}$

[1]According to the W3C specification, the element name returned for a text element is *#text*.

in which each di_j $(1 \leq j \leq n)$ is a data item. In our problem formulation, for instance, each configuration option is counted as an object, and for each configuration option a data record is created in the output data.

Definition 6.7: Template (adapted from [Reis et al., 2004] and [Arasu and Garcia-Molina, 2003]) A template is a code fragment that is comprised of a set of common layout and formatting features, and is used by a program (we call it *Web Page Maker*) to (most likely dynamically) generate the Web page content. A Web page might be produced using more than one template. Let M_D be a nonempty set of templates and DI_D a nonempty set of data items taken from a database. We denote the Web page D resulting from encoding DI_D using M_D by $\zeta(DI_D, M_D)$. The encoding means to fill the *fields* (see Definition 6.8) of the *template instances* (see Definition 6.9) with corresponding data items.

Definition 6.8: Field A field in a template denotes a data slot that is filled by a data item when a Web page is being generated using the given template(s). Therefore, the value of a text element or a tag attribute are places specified by a field. Fields in a template define the structure or *schema* of data to be encoded in the template.

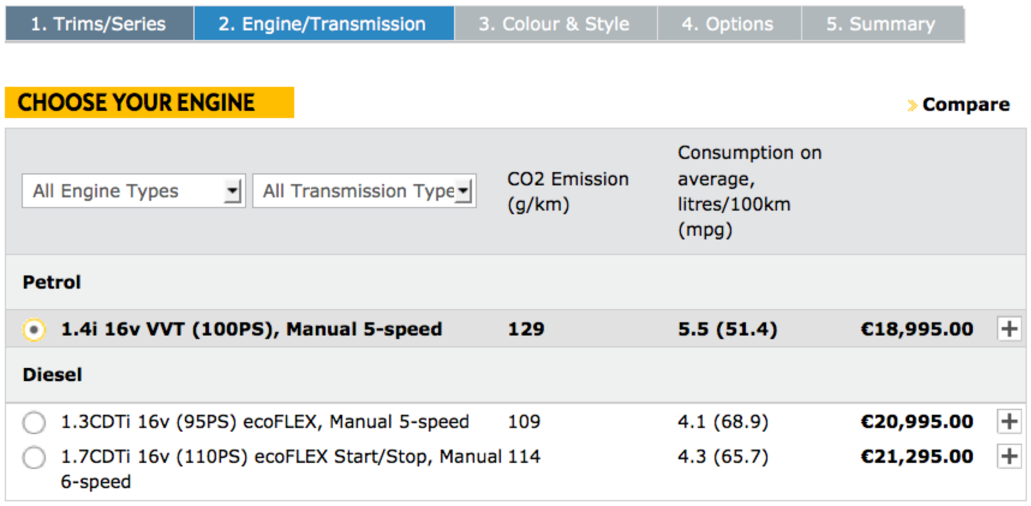
Definition 6.9: Template Instance An instance of a template is a code fragment in a Web page that is generated from the given template and its fields are filled with corresponding data items. Let C be a code fragment and M be a template, then we use C_M to denote that C is generated from M.

Definition 6.10: Data Region A data region is a portion in a Web page that contains a set of data objects. A Web page may contain one or more data regions each with variable number of data objects.

Example 6.1: An example HTML code fragment. Figure 6.1(a) shows an example Web page taken from a car Web configurator. The page presents configuration options which can be selected by users to be included in the final product (a car in this example). Each option is considered as a data object. Figure 6.1(b) depicts the corresponding code fragment[2] of the option "1.4i 16v VVT (100PS), Manual 5-speed". Data items are encoded in ***bold italic*** typeface. Figure 6.1(c) presents the fields, data items, and data record of the option shown in Figure 6.1(b).

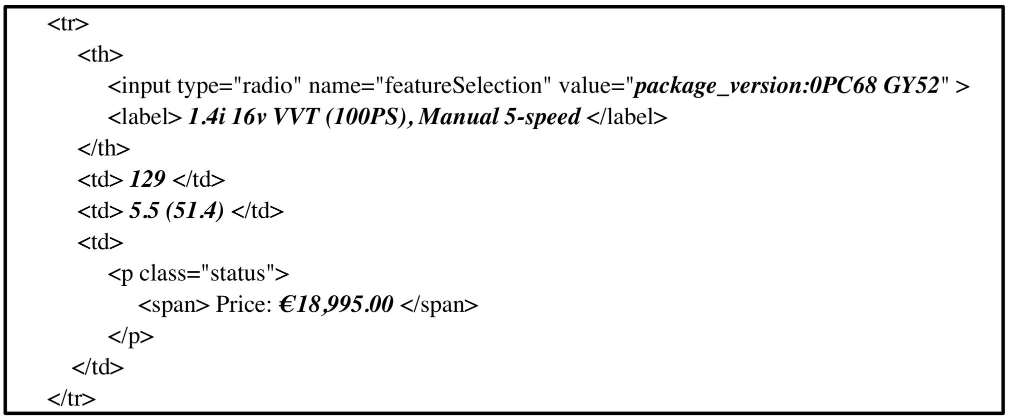
Web page generation model. Figure 6.2 presents our considered Web page generation model. *Web Page Maker* implements the function ζ described in Definition 6.7. It reads data items from a database, generates template instances from the given template(s), and for each instance fills its fields with their corresponding data items. Fields

[2]The text fonts are changed for the sake of readability.



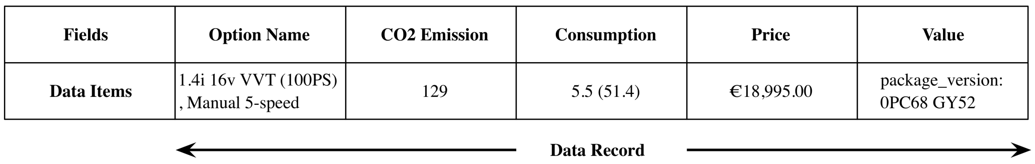
		CO2 Emission (g/km)	Consumption on average, litres/100km (mpg)		
Petrol					
●	**1.4i 16v VVT (100PS), Manual 5-speed**	**129**	**5.5 (51.4)**	**€18,995.00**	+
Diesel					
○	1.3CDTi 16v (95PS) ecoFLEX, Manual 5-speed	109	4.1 (68.9)	**€20,995.00**	+
○	1.7CDTi 16v (110PS) ecoFLEX Start/Stop, Manual 6-speed	114	4.3 (65.7)	**€21,295.00**	+

(a) A Web page containing objects (configuration options).



```
<tr>
    <th>
        <input type="radio" name="featureSelection" value="package_version:0PC68 GY52" >
        <label> 1.4i 16v VVT (100PS), Manual 5-speed </label>
    </th>
    <td> 129 </td>
    <td> 5.5 (51.4) </td>
    <td>
        <p class="status">
            <span> Price: €18,995.00 </span>
        </p>
    </td>
</tr>
```

(b) HTML code fragment for an option (“1.4i 16v VVT (100PS), Manual 5-speed”).



Fields	Option Name	CO2 Emission	Consumption	Price	Value
Data Items	1.4i 16v VVT (100PS) , Manual 5-speed	129	5.5 (51.4)	€18,995.00	package_version: 0PC68 GY52

Data Record

(c) Field, Data Item, Data Record

FIGURE 6.1: An example Web page (Opel Web Configurator: `http://www.opel.ie/tools/model-selector/cars.html`, May 8 2013).

are highlighted with boldfaced **FIELD** text string in the template and text strings in parentheses refer to the corresponding field names. Note that Web Page Maker may use different templates for different parts of a Web page. We are solely interested in those used to encode and present variability data. We also make no assumption neither about how data items are structured and modelled, nor about their underlying schema. Moreover, the Web Page Maker component might reside in the server side, in the client side, or even distributed over the two. Web Page Maker does not necessarily create the whole page in its every execution, it may only affect a part of an existing page, without creating and reloading the whole page from scratch. It means that Web Page Maker may create and add dynamic content to the page at runtime. The dynamic data can be generated by server-side activities (using technologies such as PHP, ASP, JSP, etc.) and/or client-side code (e.g., using JavaScript functions).

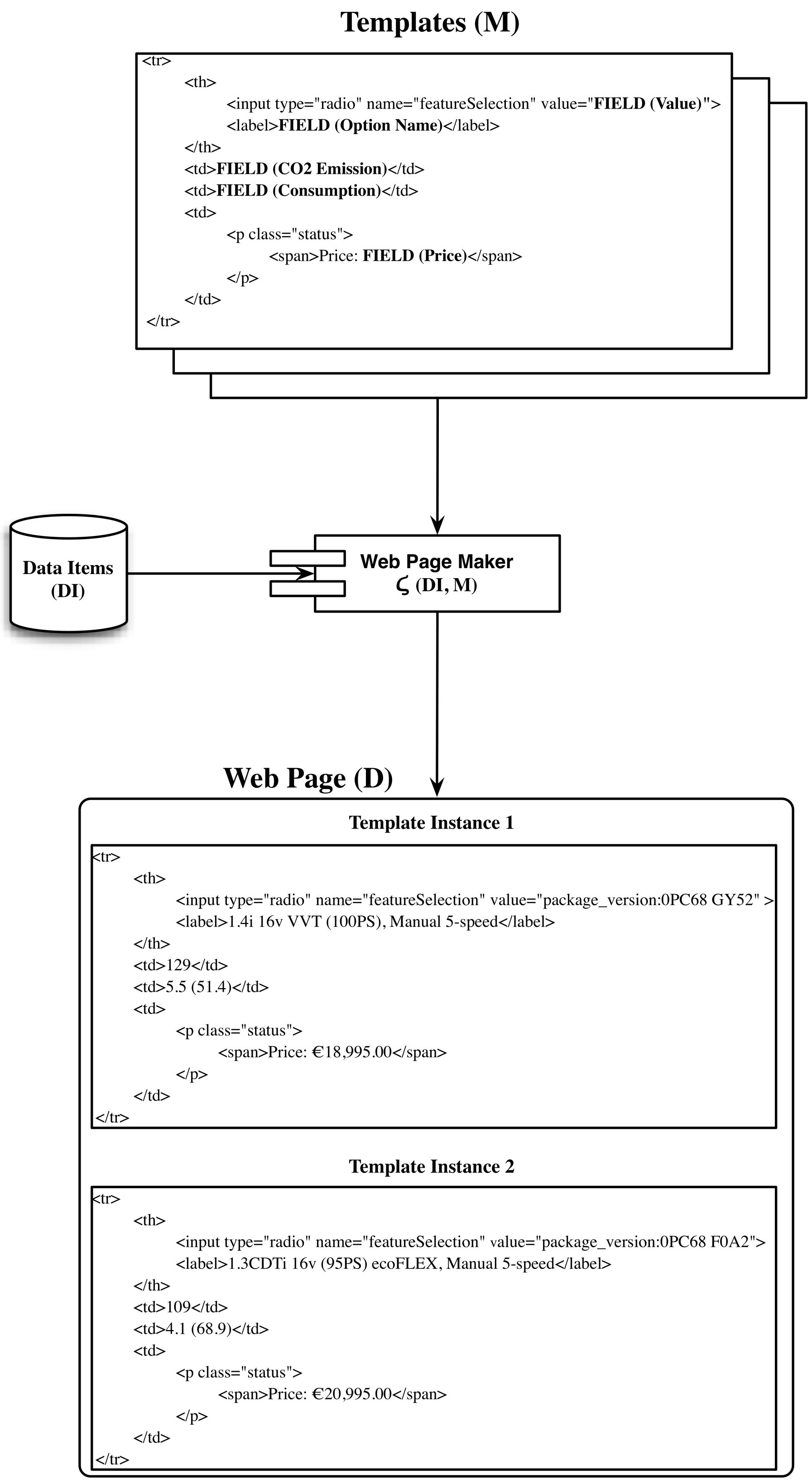
6.3 Variability Data Extraction Patterns

Definition 6.11: Variability Data Extraction Pattern A variability data extraction pattern (*vde* pattern in short) is a code fragment used to specify data to be extracted from Web pages. It defines the structure of data objects of interest to a user. It marks and names data items to be extracted from those objects. Patterns can be defined hierarchically, meaning that a pattern can be used in specification of another pattern. Patterns are uniquely identified by their names.

The structure of an object is specified by the code fragment that implements it in the source code. If this code fragment is generated from a template, it is also an instance of that template.

Definition 6.12: Data Extraction Procedure or **Web Wrapper** A Web Wrapper is a program that takes as input the specification of a set of *vde* patterns and a Web page, seeks and finds code fragments in the source code of the page that *structurally* match the given patterns, and extracts as output data items from those code fragments corresponding to the marked data in the patterns.

A *vde* pattern, in fact, is the representative of one or more templates used to generate objects in a Web page. It specifies which fields of those templates are of interest to a user and their corresponding data items will be extracted from the template instances. From this point of view, our data extraction system reverse engineers the page generation process shown in Figure 6.2. Figure 6.3 presents this reverse engineering process. ζ^{-1} is the inverse of ζ and takes a Web page D and a number of *vde* patterns P as input and



```
<tr>
        <th>
                <input type="radio" name="featureSelection" value="FIELD (Value)">
                <label>FIELD (Option Name)</label>
        </th>
        <td>FIELD (CO2 Emission)</td>
        <td>FIELD (Consumption)</td>
        <td>
                <p class="status">
                        <span>Price: FIELD (Price)</span>
                </p>
        </td>
</tr>
```

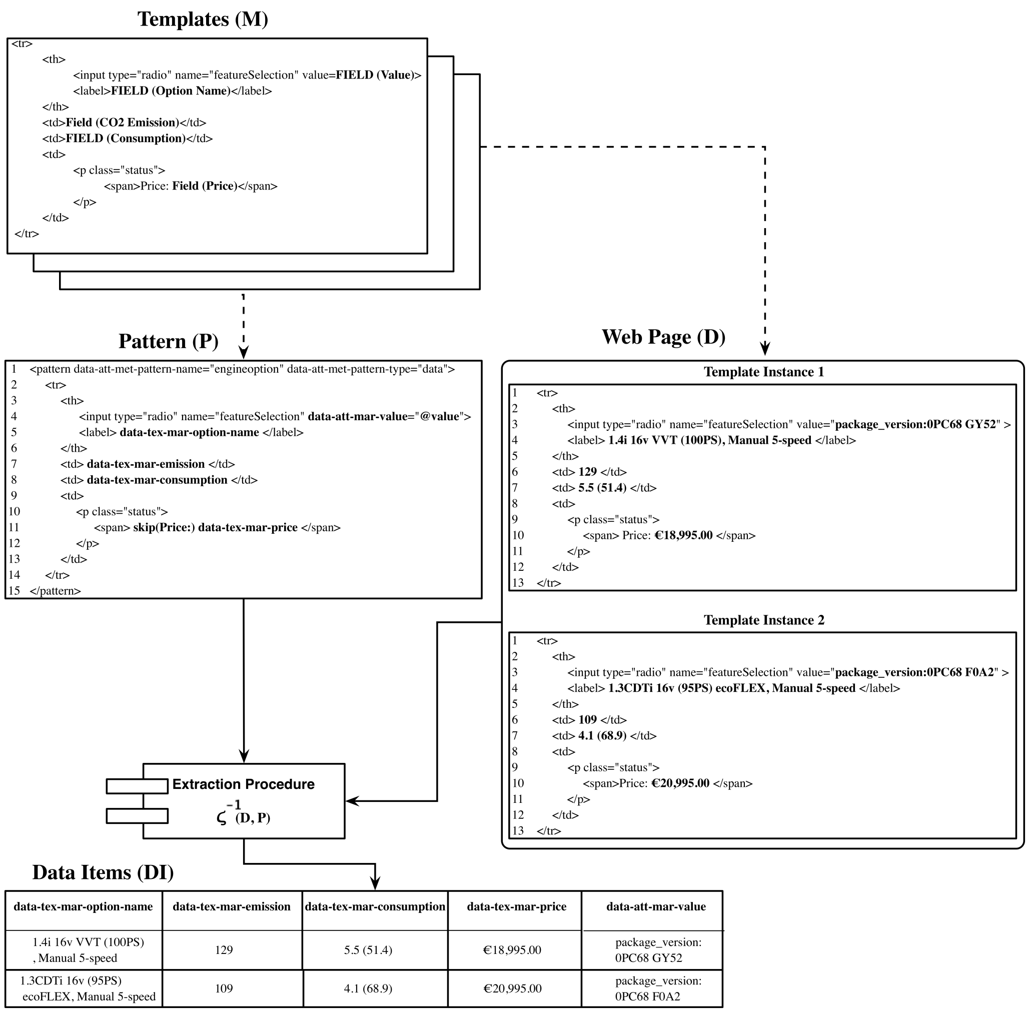
```
<tr>
        <th>
                <input type="radio" name="featureSelection" value="package_version:0PC68 GY52" >
                <label>1.4i 16v VVT (100PS), Manual 5-speed</label>
        </th>
        <td>129</td>
        <td>5.5 (51.4)</td>
        <td>
                <p class="status">
                        <span>Price: €18,995.00</span>
                </p>
        </td>
</tr>
```

```
<tr>
        <th>
                <input type="radio" name="featureSelection" value="package_version:0PC68 F0A2">
                <label>1.3CDTi 16v (95PS) ecoFLEX, Manual 5-speed</label>
        </th>
        <td>109</td>
        <td>4.1 (68.9)</td>
        <td>
                <p class="status">
                        <span>Price: €20,995.00</span>
                </p>
        </td>
</tr>
```

Figure 6.2: Web page generation model.

returns data items (DI): $\zeta^{-1}(D, P)$. Let M be a template and P be a *vde* pattern. We use $M \mapsto P$ to denote that M is represented by P.

In brief, given a *vde* pattern, the Wrapper first detects *candidate* code fragments that may match the pattern. From the candidates, it then selects those that are structurally matching the pattern. For each matching code fragment, the Wrapper binds one (or more) element, called *mapped* element, from the code fragment to one element, called *mapping* element, in the pattern and extracts data from the mapped elements with respect to the marked data in their corresponding mapping element. For instance, in Figure 6.3, the `input` element in *Template Instance 1* (line 3) is bound to the `input` element in the pattern (line 4). In the `input` element in the pattern specification, `data-att-mar-value = "@value"` is a *data marking attribute* (see Section 6.3.1) that marks the attribute `value`. Consequently, the string text of the attribute `value` from the `input` element in *Template Instance 1* is extracted as output.



Templates (M)

```
<tr>
    <th>
        <input type="radio" name="featureSelection" value=FIELD (Value)>
        <label>FIELD (Option Name)</label>
    </th>
    <td>Field (CO2 Emission)</td>
    <td>FIELD (Consumption)</td>
    <td>
        <p class="status">
            <span>Price: Field (Price)</span>
        </p>
    </td>
</tr>
```

Pattern (P)

```
1   <pattern data-att-met-pattern-name="engineoption" data-att-met-pattern-type="data">
2     <tr>
3       <th>
4         <input type="radio" name="featureSelection" data-att-mar-value="@value">
5         <label> data-tex-mar-option-name </label>
6       </th>
7       <td> data-tex-mar-emission </td>
8       <td> data-tex-mar-consumption </td>
9       <td>
10        <p class="status">
11          <span> skip(Price:) data-tex-mar-price </span>
12        </p>
13      </td>
14    </tr>
15  </pattern>
```

Web Page (D)

Template Instance 1

```
1   <tr>
2     <th>
3       <input type="radio" name="featureSelection" value="package_version:0PC68 GY52" >
4       <label> 1.4i 16v VVT (100PS), Manual 5-speed </label>
5     </th>
6     <td> 129 </td>
7     <td> 5.5 (51.4) </td>
8     <td>
9       <p class="status">
10        <span> Price: €18,995.00 </span>
11      </p>
12    </td>
13  </tr>
```

Template Instance 2

```
1   <tr>
2     <th>
3       <input type="radio" name="featureSelection" value="package_version:0PC68 F0A2" >
4       <label> 1.3CDTi 16v (95PS) ecoFLEX, Manual 5-speed </label>
5     </th>
6     <td> 109 </td>
7     <td> 4.1 (68.9) </td>
8     <td>
9       <p class="status">
10        <span>Price: €20,995.00 </span>
11      </p>
12    </td>
13  </tr>
```

Data Items (DI)

data-tex-mar-option-name	data-tex-mar-emission	data-tex-mar-consumption	data-tex-mar-price	data-att-mar-value
1.4i 16v VVT (100PS) , Manual 5-speed	129	5.5 (51.4)	€18,995.00	package_version: 0PC68 GY52
1.3CDTi 16v (95PS) ecoFLEX, Manual 5-speed	109	4.1 (68.9)	€20,995.00	package_version: 0PC68 F0A2

Figure 6.3: Data reverse engineering process.

We should indicate that a code fragment that implements a data object may match two or more different given patterns. It means that duplicate copies of a data object may be extracted. Duplicate data objects will be identified and eliminated when generating/merging the TVL models.

6.3.1 The Syntax of *vde* patterns

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

6.3.1.1 Attributes

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

Data marking attribute

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. It can mark an existing attribute or is given a string value when defining the pattern. A data marking attribute name is prefixed with `data-att-mar-`. We use the following syntax to define a data marking attribute:

- `data-att-mar-a-name` = `"@var"`, in which `data-att-mar-a-name` is the name of a data marking attribute and `var` is the name of an attribute. The symbol `@` tells the Wrapper to treat `var` as a named attribute, not a string value. When the Wrapper maps an element in a code fragment to the element containing the `data-att-mar-a-name` attribute in the pattern, it seeks to find an attribute with the name `var` in the mapped element. If the attribute is found, it then assigns its value to `data-att-mar-a-name` and extracts it.
- `data-att-mar-a-name` = `"Constant-String-Value"`, in which `Constant-String-Value` is a string value given by the user in the pattern specification. This syntax gives the user the opportunity to append user-defined data to the output.

We already predefined and reserved the following data marking attribute names:

- `data-att-mar-option-name` used to mark an attribute whose value denotes the option name.

- `data-att-mar-sub-option-name` used to mark an attribute whose value denotes a sub-option name of an option.
- `data-att-mar-widget-type` used to mark the widget type representing the option.
- `data-att-mar-sub-widget-type` used to mark the widget type representing the sub-option.
- `data-att-mar-step-name` used to mark the step name containing data objects are being extracted.
- `data-att-mar-group-name` used to mark the group name containing options are being extracted.
- `data-att-mar-id` used to temporarily assign an *id* to an element. It is internally used by the Wrapper and the Crawler and has no language-level semantics.

Example 6.2. Figure 6.4 presents the specification of a *vde* pattern, named *engine*, (and defined to extract data from the page shown in Figure 6.1(a)) and a code fragment which structurally matches the pattern. The Wrapper maps the `input` element in the code fragment (line 3) onto the `input` element in the pattern (lines 4 and 5). `data-att-mar-value` is assigned the `value` attribute. The Wrapper looks for this attribute in the `input` element of the code fragment, and assigns its value (i.e., "package_version:0PC68 GY52") to `data-att-mar-value` and records it as output. The value of `data-att-mar-widget-type` is set to `radio button` and extracted as output. In fact, using `data-att-mar-widget-type = "radio button"`, the user defines the widget type of the option as `radio button`.

Example 6.3. Figure 6.5(a) shows an excerpt of the configuration environment of a computer system configurator. The option "Ram" is represented using a list box and its sub-options are represented using list box items. Figure 6.5(b) presents a pattern (defined to extract data from list boxes) as well as the code fragment of the option "Ram" that matches the pattern. In the pattern specification, `data-att-mar-widget-type` is set to `listbox` (line 2) and `data-att-mar-sub-widget-type` to `listboxitem` (line 3). It means that the Wrapper will report `listbox` as the widget type of the option "Ram", and `listboxitem` as the widget type of its sub-options ("Kingston 8GB DDR3-1600 ECC (2x4GB)", "Kingston 16GB DDR3-1600 ECC (2x8GB)", and "Kingston 32GB DDR3-1600 ECC (4x8GB)").

Pattern Specification

```
<pattern data-att-met-pattern-name="engine" data-att-met-pattern-type="data">
    <tr>
        <th>
            <input type="radio|radio button" name="featureSelection"  value="package_version:*"
               data-att-mar-value="@value"  data-att-mar-widget-type ="radio button"
               data-att-met-unique="true">
            <label>data-tex-mar-option-name</label>
        </th>
        <td>data-tex-mar-emission</td>
        <td>data-tex-mar-consumption</td>
        <td>
            <p class="status">
                <span>skip(Price:) data-tex-mar-price</span>
            </p>
        </td>
    </tr>
</pattern>
```

Code Fragment

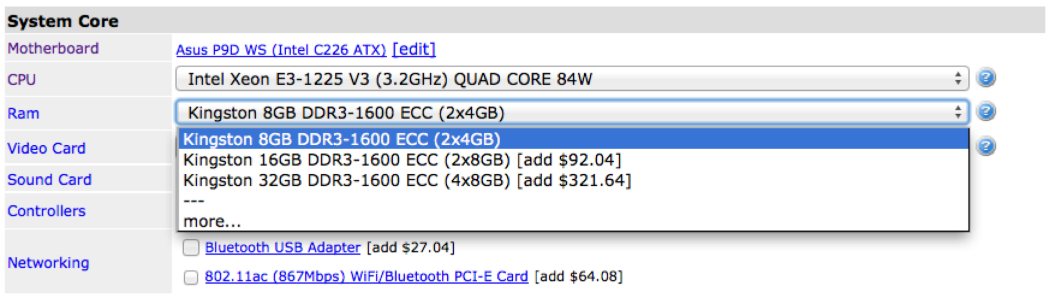
```
<tr>
    <th>
        <input type="radio" name="featureSelection" value="package_version:0PC68 GY52">
        <label>1.4i 16v VVT (100PS), Manual 5-speed</label>
    </th>
    <td>129</td>
    <td>5.5 (51.4)</td>
    <td>
        <p class="status">
            <span>Price: €18,995.00</span>
        </p>
    </td>
</tr>
```

FIGURE 6.4: *vde* pattern example (1).

Structural attribute

A *structural attribute* denotes a template-generated attribute, and therefore, all template instances generated from a template share the same list of structural attributes. Consequently, a structural attribute can be used to measure the similarity between a given pattern and a code fragment.

When the Wrapper maps two HTML elements from 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 the following three special symbols:



System Core

Motherboard	Asus P9D WS (Intel C226 ATX) [edit]
CPU	Intel Xeon E3-1225 V3 (3.2GHz) QUAD CORE 84W
Ram	Kingston 8GB DDR3-1600 ECC (2x4GB)
Video Card	
Sound Card	
Controllers	
Networking	☐ Bluetooth USB Adapter [add $27.04] ☐ 802.11ac (867Mbps) WiFi/Bluetooth PCI-E Card [add $64.08]

```
Kingston 8GB DDR3-1600 ECC (2x4GB)
Kingston 16GB DDR3-1600 ECC (2x8GB) [add $92.04]
Kingston 32GB DDR3-1600 ECC (4x8GB) [add $321.64]
---
more...
```

(a) Configuration environment (Puget Systems: `http://www.pugetsystems.com/`, May 9 2013).

Pattern Specification

```
<pattern data-att-met-pattern-name="listbox" data-att-met-pattern-type="data">
    <select  data-att-mar-option-name="@name"  data-att-mar-widget-type="listbox">
        <option  data-att-mar-sub-widget-type="listboxitem" data-att-met-multiplicity="[1..*]">
            data-tex-mar-sub-option-name
        </option>
    </select>
</pattern>
```

Code Fragment

```
<select id="Ram" class="field" name="Ram">
   <option id="Ram_opt_19865" value="19865" selected="">Kingston 8GB DDR3-1600 ECC (2x4GB)</option>
   <option id="Ram_opt_18552" value="18552">Kingston 16GB DDR3-1600 ECC (2x8GB)</option>
   <option id="Ram_opt_18553" value="18553">Kingston 32GB DDR3-1600 ECC (4x8GB)</option>
   <option value="---">---</option>
   <option value="more">more...</option>
</select>
```

(b) *vde* pattern and a matching code fragment.

FIGURE 6.5: *vde* pattern example (2).

- The *wildcard symbol (*)* that captures zero or more characters and may be discarded when mapping two structural attributes. The wildcard symbol can be placed at the beginning, at the end, or both, of an attribute value.
- The *OR operator (|)* that represents an *or*-relationship between values of the structural attribute.
- The *NOT operator (!)* that is the logical *not* operator.

The *OR* operator has the highest precedence in the attribute value. The *NOT* operator and the wildcard symbol can not be simultaneously used in the attribute value.

Example 6.4. In the pattern specification in Figure 6.4, the `input` element (line 4) contains three structural attributes, namely `type`, `name`, and `value`. An `input` element in a code fragment can be mapped onto the `input` element in the pattern, if it has the same tag position as that of the mapping `input` element (structural similarity), and has the attribute `type` whose value is "radio" or "radio button", the attribute `name` whose value is "featureSelection", and the attribute `value` whose value starts with "package_version:".

Note that the attribute `value` is a structural attribute and that its value is assigned to a data marking attribute, i.e., `data-att-mar-value` (line 5).

Meta attribute

A *meta attribute* represents a pattern-specific characteristic in the pattern specification and its name is prefixed with `data-att-met-`. All meta attributes are *predefined* and reserved words in the pattern specification language. Their purpose is to guide the Wrapper and the Crawler during the data extraction process. We predefined the following meta attributes:

- `data-att-met-pattern-name` defines the name of a *vde* pattern.
- `data-att-met-pattern-type` defines the type of a *vde* pattern. We specify three types of patterns: *region*, *data*, and *auxiliary*.
- `data-att-met-multiplicity` presents the multiplicity of an element or a pattern.
- `data-att-met-unique` indicates an HTML element in the pattern specification that there is one and only one instance of that element in the matching code fragments.
- `data-att-met-clickable` marks an HTML element in the pattern specification and tells the Crawler that this element should be clicked to simulate the user actions.
- `data-att-met-root-pattern` denotes a pattern with which the Wrapper should start the extraction process. It is used when there are several input patterns.
- `data-att-met-dependent-pattern` is an attribute with comma-separated values that is used to define dependencies between patterns.
- `data-att-met-reset-state` is an attribute with a boolean value (`true` or `false`) used to reset the configuration state (selected, deselected, etc.) of an option when it is changed by the Crawler when simulating the selection of that option.

The first four meta attributes are described in this chapter and the others in Chapter 8.

Example 6.5. In the pattern specification in Figure 6.4, `data-att-met-pattern-name` and `data-att-met-pattern-type` define the pattern name and the pattern type to respectively be `engine` and `data` (line 1). Also, `data-att-met-unique = "true"` for the `input` element (line 5) tells the Wrapper that in the matching code fragments, the `input` element (considering its structural attributes as well) should appear only once. `data-att-met-multiplicity="[1..*]"` in the pattern specification (line 3) in Figure 6.5

tells the Wrapper to look for one or more `option` elements in the matching code fragments.

6.3.1.2 HTML elements

In addition to the predefined HTML elements, we add two pattern-specific elements used in the pattern specification language, namely *pattern* and *relation*.

The *pattern* element is used to define a *vde* pattern or to refer to the name of a pattern in the specification of another pattern.

The *relation* element contains a mandatory child meta text element *or* and represents the logical disjunction between two patterns (see 6.3.2).

Example 6.6. Figure 6.6 presents the specification of three patterns: *engine*, *singleProperty*, and *listProperty*. The root element of a pattern specification must be a `pattern` element (lines 1, 16, 19). Moreover, when a pattern is used in the specification of another pattern, the name of the referred pattern is the mandatory single child text element of a *pattern* element in the pattern specification (lines 5-7 and 11-13).

6.3.1.3 Text elements

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

Data marking text element

A *data marking text element* indicates a text element representing a data slot required by a user. It, in fact, denotes a data item that will be extracted from the matching code fragments.

A data marking text element is prefixed with `data-tex-mar-`. We have reserved the following text elements:

- `data-tex-mar-option-name` used to mark a text element whose value denotes an option name.
- `data-tex-mar-sub-option-name` used to mark a text element whose value denotes a sub-option name of an option.

```
<pattern  data-att-met-pattern-type = "data" data-att-met-pattern-name="engine">
    <li>
        <input   data-att-met-unique="true" type="radio"/>
        <label>data-tex-mar-option-name</label>
        <pattern>
                  singleProperty
        </pattern>
        <relation>
                  or
        </relation>
        <pattern>
                  listProperty
        </pattern>
    </li>
</pattern>

<pattern  data-att-met-pattern-type = "auxiliary" data-att-met-pattern-name="singleProperty">
    <p class=property> data-tex-mar-property</p>
</pattern>

 <pattern  data-att-met-pattern-type = "auxiliary" data-att-met-pattern-name="listProperty">
    <ul class="property">
        <li data-att-met-multiplicity="[*]">
            <label>data-tex-mar-property</label>
        </li>
    </ul>
</pattern>
```

FIGURE 6.6: *vde* pattern example (3).

Example 6.7. `data-tex-mar-option-name` in the pattern specification (line 6) in Figure 6.4 is a data marking text element that tells the Wrapper to extract the immediate child text element of the `label` element in the matching code fragments and record it as the option name. Also, `data-tex-mar-sub-option-name` in the pattern specification (line 4) of Figure 6.5 indicates that the immediate child text element of `option` elements in the matching code fragments will be recorded as sub-option names in the output data.

Structural text element

A *structural text element* is used in the following situations:

- A structural text element denotes a template-generated text value, and therefore, all template instances generated from a template share the same structural text elements. Consequently, a structural text element can be used to measure the similarity between a given pattern and a code fragment (i.e., a template instance). We use a function-style syntax `skip(Text-Value)` to present a structural text element in which `Text-Value` is the template-generated text value (e.g., `skip(Price:)`).

- The user may need to set up the Wrapper to skip some elements during mapping code fragments and a pattern. We use two variants of the `skip` function to specify skipped elements: `skip(all)` and `skip(sibling, Multiplicity)`. These functions are further discussed in Section 6.3.2.

Example 6.8. Consider `<span>skip(Price:) data-tex-mar-price</span>` in the pattern specification (line 12) in Figure 6.4. The "Price:" string in the inner child text element of the `span` element is part of the template and the remaining suffix, that is the price value, is part of data. It, in fact, tells the Wrapper that the inner child text element of the `span` element of the matching code fragments must contain and start with the string "Price:". The Wrapper skips this string and records the remaining suffix portion of the text (i.e., "€18,995.00" – line 10 in the code fragment) in the data marking text element `data-tex-mar-price`. Using this syntax, the Wrapper partitions the template and the data segments in a text element.

Meta text element

A *meta text element* is used to present a pattern-specific text value in the following cases:

- A meta text element denotes a pattern name when this pattern is used in the specification of another pattern (lines 6 and 12 in Figure 6.6).
- A meta text element is used to define the disjunction between two patterns (line 9 in Figure 6.6).

6.3.2 The expressiveness of *vde* patterns

This section presents how *vde* patterns can be used to deal with well-known expressiveness problems already reported in the general field of Web data extraction [Chang et al., 2006] and likely to impact the extraction of data from 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.

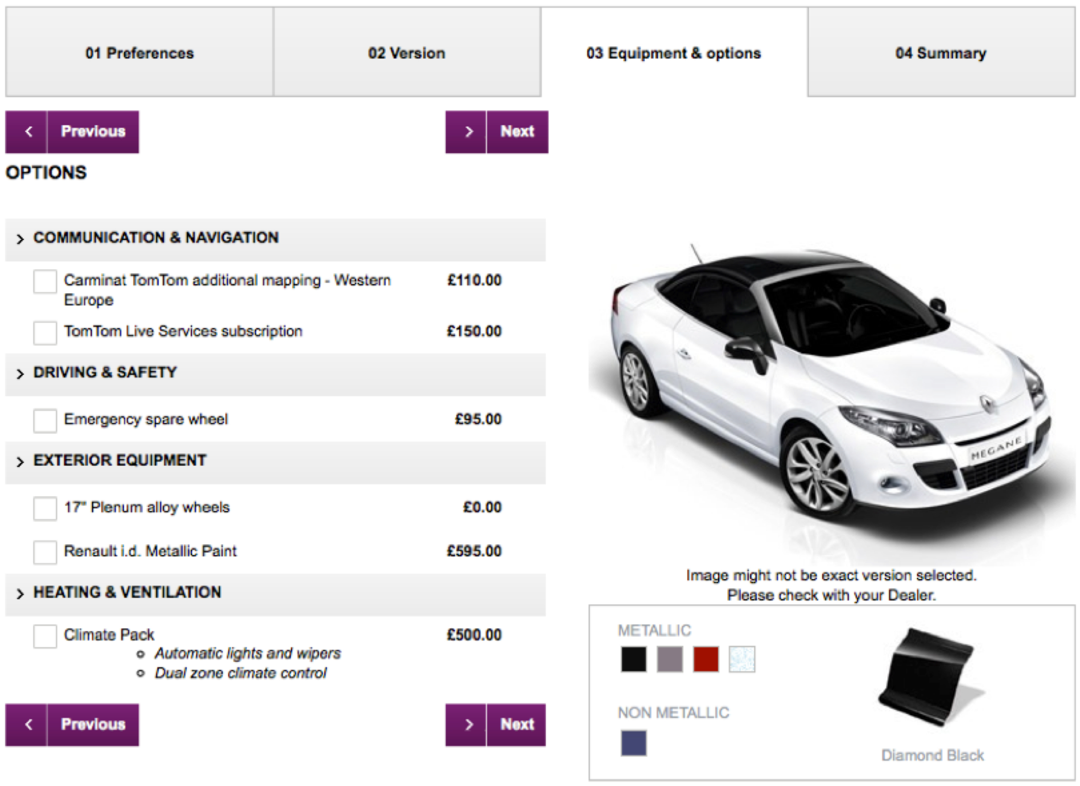
Multiplicity. The multiplicity of an element is defined in the `data-att-met-multiplicity` meta attribute . The value of this attribute is either a single positive integer number, the *infinity* symbol ("*"), or a range. A range is defined by stating the minimum and maximum positive integer values, separated by two dots. The maximum value can be the infinity symbol. The value of a multiplicity attribute should always be enclosed in square braces (e.g., `[1..5]`). The user may define the multiplicity of a pattern as well.

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. By definition, the multiplicity of a *vde* pattern is `1..*` and of an HTML element is `1`, if it is not explicitly defined. The pattern in Figure 6.5 is specified to define the list box structure. The multiplicity of the *listbox* pattern is not defined, but by definition, considered to be `1..*`. The multiplicity of the `option` element is explicitly defined as `1..*` (line 3). The Wrapper in the code fragments that match the pattern and within the `select` element looks for one or more `option` element(s).

Optional elements

In code fragments representing similar objects, one common variation is that an element may appear in some fragments and but not in all. This element is called an *optional* (or *missing*) element. To present this variation in a *vde* pattern, we define the `0..1` multiplicity for the optional element.

Example 6.9. Figure 6.7(a) depicts an excerpt of the configuration environment of a car configurator. Each configuration option is characterized by a name and its price, except for the last option in the list (i.e., "Climate pack") that additionally contains a list of sub-options ("Automatic lights and wipers" and "Dual zone climate control"). The list of the sub-options in this example is an optional data, meaning that not all objects contain this data. In Figure 6.7(b), two code fragments of the last two options are represented. The first code fragment (lines 1-7) is for an option without the optional data, and the second one is for an option including the optional data (8-19). The optional data is implemented using a code snippet with the `ul` element as the root element (lines 12-15). In the pattern specification, the pattern *sub-options* is defined (lines 10-14) to encode the optional data, then is used in the *options* pattern as an optional element (line 5). Note that the multiplicity of the *sub-options* pattern is defined to be *0..1* to denote its optionality.



01 Preferences | 02 Version | 03 Equipment & options | 04 Summary

Previous | Next

OPTIONS

› COMMUNICATION & NAVIGATION	
☐ Carminat TomTom additional mapping - Western Europe	£110.00
☐ TomTom Live Services subscription	£150.00
› DRIVING & SAFETY	
☐ Emergency spare wheel	£95.00
› EXTERIOR EQUIPMENT	
☐ 17" Plenum alloy wheels	£0.00
☐ Renault i.d. Metallic Paint	£595.00
› HEATING & VENTILATION	
☐ Climate Pack ◦ *Automatic lights and wipers* ◦ *Dual zone climate control*	£500.00

Previous | Next

Image might not be exact version selected.
Please check with your Dealer.

METALLIC

NON METALLIC

Diamond Black

(a) Configuration environment (Renault car configurator: `http://www.renault.co.uk/`, May 12 2013).

Pattern Specification

```
<pattern data-att-met-pattern-type="data" data-att-met-pattern-name="options">
    <input  type="checkbox"/>
    <label>
        <span class="SectionText"> data-tex-mar-option-name
            <pattern data-att-met-multiplicity="[0..1]">  sub-options </pattern>
        </span>
        <span class="SectionPrice">data-tex-mar-price </span>
    </label>
</pattern>
<pattern data-att-met-pattern-type="auxiliary" data-att-met-pattern-name="sub-options">
    <ul>
        <li data-att-met-multiplicity="[1..*]"> data-tex-mar-require-option </li>
    </ul>
</pattern>
```

Code Fragment

```
<td>
    <input  type="checkbox"/>
    <label>
        <span class="sectionText"> Renault i.d. Metallic Paint </span>
        <span class="SectionPrice"> £595.00 </span>
    </label>
</td>
<td>
    <input  type="checkbox"/>
     <label>
         <span class="SectionText"> Climate Pack
             <ul>
                 <li> Automatic lights and wipers </li>
                 <li> Dual zone climate control </li>
             </ul>
         </span>
         <span class="SectionPrice"> £500.00 </span>
     </label>
</td>
```

(b) *vde* pattern and two matching code fragments.

FIGURE 6.7: *vde* pattern example (4).

Pattern relationships

Let P_1 and P_2 be two *vde* patterns. Then, we define two different relationships between P_1 and P_2: *parent-child* and *disjunction*.

Parent-child. P_1 is a child of P_2 if P_1 is used in the specification of P_2. This presents the hierarchical relationship between patterns.

Disjunction. P_1 and P_2 are two disjunctive patterns if their occurrences are mutually exclusive. In this case, the `relation` element with a predefined mandatory child meta text element `or` defines the or-relationship of the two patterns. Two disjunctive patterns are the left and the right siblings of the `relation` element. Each pattern has its own multiplicity.

Example 6.10. In Figure 6.6, the *singleProperty* and *listProperty* patterns are used in specification of the pattern *engine* and they are the child patterns of the *engine* pattern. In fact, *singleProperty* and *listProperty* extend the specification of the *engine* pattern. The `relation` element is also used to define an or-relationship between the *singleProperty* and *listProperty* patterns (lines 5-13). Semantically, it means that after the `label` element (line 4), the Wrapper should look for a <`p class="property"`> element (the first element of the *singleProperty* pattern, line 17) or for a <`ul class="property"`> element (the first element of the *listProperty* pattern, line 20).

Skipped elements

When the user specifies a *vde* pattern to present a set of templates in the abstract, she might find that some HTML elements neither hold data of interest nor are useful in measuring the structural similarity between the pattern and code fragments. These noisy elements should be skipped by the Wrapper during the pattern-matching process (see Chapter 7). We use the function-style `skip` text element to denote skipped elements. We use the following variant forms of `skip`:

- `skip(sibling, Multiplicity)`, in which `sibling` is a reserved word and *Multiplicity* is a multiplicity specification. Semantically, it means that a number of consecutive sibling elements should be skipped by the Wrapper. The value of *Multiplicity* defines how many elements should be discarded. The right sibling element of the `skip(sibling, Multiplicity)` element must be an HTML element or a pattern.
- `skip(all)`: Let v be the parent element of the `skip(all)` text element in a pattern specification. Then, `skip(all)` tells the Wrapper that it should skip all descendants

of v and then consider the element that follows `skip(all)`. A `skip(all)` element must be followed by an HTML element, a data marking text element, or a pattern.

Example 6.11. Figure 6.8 shows a code fragment encoding an option and the corresponding pattern *options*. The user finds the two `img` elements (lines 3 and 4) in the first `td` element to be noisy. Therefore, in the pattern she uses a skip text element (line 4) to tell the Wrapper to ignore all elements until it finds an `input` element (line 5).

Pattern Specification

```
1     <pattern data-att-met-pattern-name="options" data-att-met-pattern-type="data">
2       <tr>
3         <td>
4           skip(sibling,[1..*])
5           <input type="checkbox">
6         </td>
7         <td>data-tex-mar-option-name</td>
8         <td>data-tex-mar-price</td>
9       </tr>
10    </pattern>
```

Code Fragment

```
1     <tr>
2       <td>
3         <img src="/FIAT_IRELAND/images/spacer.gif">
4         <img src="/FIAT_IRELAND/images/spacer.gif">
5         <input class="check" type="checkbox">
6       </td>
7       <td> Blue&Me </td>
8       <td> 330.00</td>
9     </tr>
```

FIGURE 6.8: *vde* pattern example (5) (FIAT car configurator: `http://www.fiat.ie`, May 13 2013).

Example 6.12. Figure 6.9 presents a code fragment that represents an option. Assume that the user wants to extract only the option name (line 5) and its price (line 24). These two data items can be uniquely identified by the <`span class="label"`> and <`div class="single"`> elements respectively. The *vde* pattern tells the Wrapper to discard all other elements (line 3) until it finds a <`span class="label"`> element (line 4) and discard all other elements (line 5) until it finds a <`div class="single"`> element (line 6).

Example 6.13. A code fragment representing an option and a *vde* pattern specified to extract data from that are presented in Figure 6.10. `skip(sibling,[1])` in the pattern specification (line 6) tells the Wrapper to discard one element after the first `td` element.

Pattern Specification

```
<pattern data-att-met-pattern-type="data" data-att-met-pattern-name="options">
    <div  class="row" data-att-met-unique="true">
        skip(all)
        <span class="label">data-tex-mar-option-name</span>
        skip(all)
        <div class="single">data-tex-mar-price</div>
    </div>
</pattern>
```

Code Fragment

```
<div class="row">
    <div class="column1">
        <div class="checkBox" key="MAFHUG4">
            <span class="icon jqCheckBox"></span>
            <span class="label"> Audi hill-hold assist</span>
        </div>
    </div>
    <div class="column2">
        <div class="htmlTooltip">
            <div class="benefits">
                <ul>
                    <li>
                        The driver is able to pull away conveniently on uphill
                        slops without rolling back
                    </li>
                </ul>
            </div>
        </div>
    </div>
    <div class="columns3">
        <div></div>
    </div>
    <div class="column4 price">
        <div class="single"> 65.00 GBP </div>
    </div>
    <div class="clear"></div>
</div>
```

FIGURE 6.9: *vde* pattern example (6) (Audi car configurator: `http://configurator.audi.co.uk`, May 13 2013).

Consequently, the Wrapper considers the first `td` (lines 2-4) but ignores the second `td` element (lines 5-9) in the code fragment. `skip(all)` (line 8) tells the Wrapper to ignore all other descendent elements of the `td` element and consider the `label` element (line 9). `skip(all)` (line 10) guides the Wrapper to ignore all other descendent elements of the `td` except a `p` element (line 11). `skip(all)` (line 12) tells the Wrapper to extract all text values contained in the `p` element. It means that all text values from lines 17 to 23 in the code fragment are extracted and assigned to `data-tex-mar-description`. Note that the last `td` element (line 26) in the code fragment is automatically discarded by the Wrapper because the element is not mapped to an element in the pattern.

6.3.3 Pattern types

We specify three types of patterns: *data*, *auxiliary*, and *region*.

Data pattern. A data pattern marks text elements and attributes carrying the content of interest and denotes code fragments (i.e., template instances) that match certain properties and thus contain the relevant data. The first-level children of a data pattern must not contain any variations of the `skip` element except `skip(sibling,[Integer])` in which *Integer* is an integer number. The value of `data-att-met-pattern-type` for a data pattern is `data`.

Auxiliary pattern. An auxiliary pattern provides additional specification to the definition of a data pattern. By definition, the first level of an auxiliary pattern must contain one and only one HTML element. Therefore, no variations of the `skip` element can appear in the first level of an auxiliary pattern. The value of `data-att-met-pattern-type` for an auxiliary pattern is `auxiliary`.

Region pattern. A region pattern highlights a portion of a page. It specifies which part of the source code of a Web page code fragments may match the given pattern and thus where the Wrapper should operate. A region pattern denotes the root element of the region. No variations of the `skip` element can appear in definition of a region pattern. The value of `data-att-met-pattern-type` for a region pattern is `region`.

All patterns required to extract data from a Web page are specified in a *configuration file*. A configuration file contains the specification of at least one region pattern, one data pattern, and optionally a set of auxiliary patterns. If more than one region pattern is contained in a configuration file, one and only one of those must have the attribute `data-att-met-root-pattern="true"`. Within a configuration file, patterns are uniquely identified by their names.

Pattern Specification

```
<pattern data-att-met-pattern-type="data" data-att-met-pattern-name="options">
    <tr>
         <td>
              <input  type="radio"  data-att-met-unique="true"/>
         </td>
         skip(sibling,[1])
         <td>
            skip(all)
            <label>data-tex-mar-option-name</label>
            skip(all)
            <p>
                skip(all)
                data-tex-mar-description
            </p>
         </td>
    </tr>
</pattern>
```

Code Fragment

```
<tr>
    <td>
        <input type="radio"  id="embossed" checked="checked" value="em" name="t1type"/>
    </td>
    <td>
       <label for="embossed">
            <img title="Embossed Characters " src="images/site/emboss_sample.png">
       </label>
    </td>
    <td>
        <h6>
            <label for = "embossed"> Embossed Characters</label>
            <a class="imgbutton" href="/images/photo/dt-std-matte.jpg">
            <a href="/images/photo/dt-std-shiny.jpg">
        </h6>
            <p>
               Characters are raised above surface.
               <br>
               This modern military font type is the most popular, select this if you are unsure. Choose from:
               <br>
               <span> A…Z 0…9 °"'*.#?$!:-+=()&,\/ @ </span>
                © ★ † ♥
               <span> ♣ </span>
            </p>
    </td>
    <td></td>
</tr>
```

FIGURE 6.10: *vde* pattern example (7) (`http://www.mydogtag.com/`, May 14 2013).

The Wrapper takes the configuration file as input and interprets it to find out within which part of the source code (defined by the region pattern) which code fragments (defined by the data and auxiliary patterns) should be parsed to extract data. Figure 6.11 depicts a pattern configuration file. The file contains a data pattern, named *options* (lines 1-12), an auxiliary pattern, named *description* (lines 13-18), and a region pattern, named *region* (lines 19-23). The region pattern tells the Wrapper to look within the first visited element in the source code that is identical to the `<table class="tagselect">` element (line 20) for code fragments that match the *options* pattern (line 21). The *description* pattern is used in the specification of the *options* pattern (line 10).

```
1    <pattern data-att-met-pattern-type="data" data-att-met-pattern-name="options">
2      <td>
3          <input  type="radio"  data-att-met-unique="true" data-att-mar-widget-type="@type"/>
4      </td>
5      skip(sibling,[1])
6      <td>
7        skip(all)
8        <label> data-tex-mar-option-name</label>
9        skip(all)
10       <pattern data-att-met-multiplicity="[1]"> description </pattern>
11     </td>
12   </pattern>

13   <pattern data-att-met-pattern-type="auxiliary" data-att-met-pattern-name="description">
14     <p>
15       skip(all)
16       data-tex-mar-description
17     </p>
18   </pattern>

19   <pattern data-att-met-pattern-type="region" data-att-met-pattern-name="region">
20     <table class="tagselect">
21       <pattern> options </pattern>
22     </table>
23   </pattern>
```

FIGURE 6.11: Pattern configuration file.

6.4 Grammar

In this section, we present a context-free grammar for the syntax of *vde* patterns. The grammar is given in *Extended Backus-Naur form* (EBNF). The following conventions are used:

- Each production rule starts with the rule's name, followed by the replacement symbol (::=) and the rule's value.
- Terminals are written in lowercase and non-terminals in uppercase.
- Parentheses are used for grouping.
- An optional element is followed by `?`: `E?` means `E` is optional.
- Repeated elements are enclosed in parentheses and followed by `+` or `*`: `(E)+` means `E` repeats one or more times and `(E)*` means `E` repeats zero or more times.
- The vertical bar (`|`) separates alternatives when the rule has several different values.
- When the value of a production rule is described outside our grammar, we use natural language to describe it.

Since a *vde* pattern has an HTML-like syntax, each HTML element is represented by its opening and enclosing pair of tags in the grammar. Moreover, whenever an HTML element is used in a production rule and the element has attributes, the attributes can appear in any order.

Note that Firefox (on top of which we implemented our data extraction system) treats empty white spaces or new lines as text elements. However, our pattern specification language is a free-form language, and therefore, there is no semantics behind indents in the grammar rules or presenting an HTML element in multiple lines. They are used to help the reader to easily determine where the body of a block begins and ends. We should also indicate that in our pattern specification language, `|` and `*` are also considered as terminals. From the context in which they are used, it is easy to distinguish them from punctuation elements (`|` and `*`) of the grammar.

Configuration file

The starting non-terminal is the configuration file that contains at least one region pattern, one data pattern, and optionally a number of auxiliary patterns. Patterns can appear in any order in a configuration file.

```
CONFIGURATION_FILE ::=  (REGION_PATTERN DATA_PATTERN  (AUXILIARY_PATTERN)* )+
```

Attributes

HTML elements and the *pattern* element can have attributes which are always specified in the opening tag of the element. An attribute comes in the form of `name="value"`. Attribute values are enclosed in double quotes, and in some rare situations when the attribute value itself contains double quotes, it is enclosed in single quotes. We present only double quotes in the grammar, but single quotes are allowed as well.

```
HTML_ATTRIBUTE ::= ATTRIBUTE_NAME = "ATTRIBUTE_VALUE"
ATTRIBUTE_NAME::= a valid HTML attribute name
ATTRIBUTE_VALUE ::= a valid HTML attribute value
```

Pattern name. Each *vde* pattern is given a unique name in the configuration file that contains it. The pattern name is specified in the `data-att-met-pattern-name` attribute of the `pattern` element.

```
PATTERN_NAME_ATTRIBUTE ::= data-att-met-pattern-name = "ATTRIBUTE_VALUE"
```

The pattern names must be unique in a given configuration file.

Unique element indicator. The attribute `data-att-met-unique="true"` marks an element in a data pattern that appears only once in the matching code fragments. In the current implementation of our data extraction system, one and only one HTML element must be marked as the unique element in the data pattern.

```
ELEMENT_UNIQUENESS_ATTRIBUTE ::= data-att-met-unique = "true"
```

Clickable element indicator. The attribute `data-att-met-clickable="true"` marks an element in a data pattern that is clickable.

```
CLICKABLE_ELEMENT_ATTRIBUTE ::= data-att-met-clickable = "true"
```

Root region pattern indicator. In a configuration file that contains several region patterns, the attribute `data-att-met-root-pattern="true"` denotes the region pattern with which the extraction process starts. In a configuration file, one and only one region pattern can have this attribute.

```
ROOT_REGION_PATTERN_ATTRIBUTE ::= data-att-met-root-pattern = "true"
```

Dependent pattern indicator. In a configuration file that contains several region patterns, the comma-separated value of the attribute `data-att-met-dependent-pattern` denotes the region patterns that depend on the region pattern owning this attribute (the independent pattern).

```
DEPENDENT_REGION_PATTERN_ATTRIBUTE ::=
  data-att-met-dependent-pattern = "ATTRIBUTE_VALUE1(,ATTRIBUTE_VALUE2)*"
```

ATTRIBUTE_VALUE1 and ATTRIBUTE_VALUE2 refer to the names of region patterns that exist in the configuration file. Note that * is a punctuation here, not a terminal.

Reset configuration state of an option. When an option is automatically configured when crawling, `data-att-met-reset-state = "true"` tells the Web Crawler to reset the configuration state of the option to the state it was in before crawling.

```
  RESET_CONFIGURATION_STATE_ATTRIBUTE ::=
    data-att-met-reset-state = "true"|"false"
```

Data marking attribute. A data marking attribute name must match the $data \backslash\backslash - tex \backslash\backslash - mar \backslash\backslash - [a - zA - Z0 - 9 \backslash\backslash -]+$ regular expression. Its value can be either a valid HTML attribute value or an attribute name starting with the @ symbol.

```
DATA_MARKING_ATTRIBUTE ::=
  DATA_MARKING_ATTRIBUTE_NAME = DATA_MARKING_ATTRIBUTE_VALUE

DATA_MARKING_ATTRIBUTE_NAME ::= a valid HTML attribute name matching
                                the data\\-att\\-mar\\-[a-zA-Z0-9\\-]+
                                regular expression

DATA_MARKING_ATTRIBUTE_VALUE ::=   "ATTRIBUTE_VALUE"
                                |  "@ATTRIBUTE_NAME"
```

If the Wrapper finds an attribute named ATTRIBUTE_NAME in the mapped element of the target code fragment, assigns its value to DATA_MARKING_ATTRIBUTE_NAME, otherwise, ignores the attribute.

Step name and group name attributes. These attributes are used to define the name of the step and the group that contain the options being extracted.

```
STEP_NAME_ATTRIBUTE ::= data-att-mar-step-name = "ATTRIBUTE_VALUE"
GROUP_NAME_ATTRIBUTE ::= data-att-mar-group-name = "ATTRIBUTE_VALUE"
```

Structural attribute. Structural attributes are used to evaluate if two HTML elements (one from the pattern and one from the target code fragment) are identical. If an HTML element in the target code fragment is mapped to an HTML element in the pattern, they are identical elements if they have the same tag name and an analogy can be drawn between their structural attributes. Let assume that the element in the pattern has an attributed named ATTRIBUTE_NAME:

(1) If `ATTRIBUTE_NAME="ATTRIBUTE_VALUE"` is in the element in the pattern, the mapped element in the code fragment must exactly have `ATTRIBUTE_NAME = "ATTRIBUTE_VALUE"`.

(2) If `ATTRIBUTE_NAME="ATTRIBUTE_VALUE*"` is in the element in the pattern, the mapped element in the code fragment must have an attribute named ATTRIBUTE_NAME which value *starts with* ATTRIBUTE_VALUE.

(3) If `ATTRIBUTE_NAME="*ATTRIBUTE_VALUE"` is in the element in the pattern, the mapped element in the code fragment must have an attribute named `ATTRIBUTE_NAME` which value *ends with* `ATTRIBUTE_VALUE`.

(4) If `ATTRIBUTE_NAME="*ATTRIBUTE_VALUE*"` is in the element in the pattern, the mapped element in the code fragment must have an attribute named `ATTRIBUTE_NAME` which value *contains* `ATTRIBUTE_VALUE`.

(5) If `ATTRIBUTE_NAME="!ATTRIBUTE_VALUE"` is in the element in the pattern, the mapped element in the code fragment must not have `ATTRIBUTE_NAME = "ATTRIBUTE_VALUE"`.

(6) If `ATTRIBUTE_NAME="ATTRIBUTE_VALUE1|ATTRIBUTE_VALUE2"` is in the element in the pattern, the mapped element in the code fragment must have an attribute named `ATTRIBUTE_NAME` which value is either `ATTRIBUTE_VALUE1` or `ATTRIBUTE_VALUE2`.

```
STRUCTURAL_ATTRIBUTE ::=
ATTRIBUTE_NAME= "STRUCTURAL_ATTRIBUTE_VALUE" | COMPOUND_STRUCTURAL_ATTRIBUTE_VALUE
STRUCTURAL_ATTRIBUTE_VALUE ::=   ATTRIBUTE_VALUE
                              |  ATTRIBUTE_VALUE*
                              |  *ATTRIBUTE_VALUE
                              |  *ATTRIBUTE_VALUE*
                              |  !ATTRIBUTE_VALUE

COMPOUND_STRUCTURAL_ATTRIBUTE_VALUE ::=
  "STRUCTURAL_ATTRIBUTE_VALUE(|STRUCTURAL_ATTRIBUTE_VALUE)+"
```

The or operator (`|`) has the highest precedence in the attribute value. Note that `|` in the last production rule is a terminal.

Multiplicity. One or more adjacent elements in the target code fragment may be mapped to an element in the pattern. The value of the `data-att-met-multiplicity` attribute in the pattern specifies up to how many adjacent elements in the target code fragment can be mapped to the element owning the `data-att-met-multiplicity` attribute in the pattern. The value of a multiplicity attribute should always be enclosed in square braces (`[]`) and it can either be a positive integer number, the wildcard symbol `*`, or a range.

```
MULTIPLICITY_ATTRIBUTE ::= data-att-met-multiplicity = MULTIPLICITY_SPECIFICATION
MULTIPLICITY_SPECIFICATION ::=  DEFINITE_MULTIPLICITY_SPECIFICATION
                             | INDEFINITE_MULTIPLICITY_SPECIFICATION
                             | RANGE_MULTIPLICITY_SPECIFICATION
DEFINITE_MULTIPLICITY_SPECIFICATION :: =  "[A]"
INDEFINITE_MULTIPLICITY_SPECIFICATION ::= "[*]"
RANGE_MULTIPLICITY_SPECIFICATION ::=      "[A..B]" | "[A..*]"

A ::= a positive integer number
B ::= a positive integer number equal to/greater than A
```

Region pattern

All specified patterns in the configuration file have a `pattern` element as the root element. Each pattern is given a name using the `data-att-met-pattern-name` attribute and its value must be unique in the configuration file. When the extraction process starts, the Wrapper first looks for a region pattern in the configuration file. A region pattern is distinguished from the others by the `data-att-met-pattern-type = "region"` attribute. The multiplicity of a region pattern is restricted to be *1* and the user is not allowed to define a new multiplicity value.

Structurally, a region pattern has only one child HTML element that optionally can have one or more attributes. This HTML element, in turn, must have a child `pattern` element whose child text element refers to a data pattern name (`DATA_PATTERN_NAME`). If the Wrapper finds more than one element matching the given HTML element of the region pattern, it considers the first one and ignores the others.

```
REGION_PATTERN ::=<pattern data-att-met-pattern-type="region"
                    PATTERN_NAME_ATTRIBUTE
                    ROOT_REGION_PATTERN_ATTRIBUTE?
                    DEPENDENT_REGION_PATTERN_ATTRIBUTE?
                    STEP_NAME_ATTRIBUTE?
                    GROUP_NAME_ATTRIBUTE?>
                    REGION_PATTERN_BODY
                  </pattern>

REGION_PATTERN_BODY ::= <HTML_TAG  (STRUCTURAL_ATTRIBUTE)*>
                          <pattern MULTIPLICITY_ATTRIBUTE?>
                                DATA_PATTERN_NAME
                          </pattern>
                        </HTML_TAG>
DATA_PATTERN_NAME ::= a defined data pattern name
HTML_TAG ::= a predefined valid html tag
```

Data pattern

A data pattern is the key pattern to extract data. It is identified from the other patterns by the `data-att-met-pattern-type = "data"` attribute. By definition, the multiplicity of a data pattern is considered to be `1..*`, but the user can specify a different multiplicity using the `data-att-met-multiplicity` attribute. The user also can optionally define the multiplicity for a data pattern where it is referred in the region pattern.

A data pattern can have one ore more HTML elements as first-level children. Moreover, `skip(sibling,DEFINITE_MULTIPLICITY_SPECIFICATION)` elements can appear in the first level of a data pattern, but each of those elements must be surrounded by two HTML elements.

```
DATA_PATTERN ::=<pattern data-att-met-pattern-type="data"
                    PATTERN_NAME_ATTRIBUTE
                    MULTIPLICITY_ATTRIBUTE?>
                    DATA_PATTERN_BODY
                  </pattern>

DATA_PATTERN_BODY ::=
(FIRST_LEVEL_HTML_ELEMENT)+
 |
(FIRST_LEVEL_HTML_ELEMENT SKIP_DEFINITE_SIBLING_ELEMENT FIRST_LEVEL_HTML_ELEMENT)*
```

```
 SKIP_DEFINITE_SIBLING_ELEMENT ::=
 skip(sibling,DEFINITE_MULTIPLICITY_SPECIFICATION)
```

An HTML element in the data pattern can be marked as a unique element and may have one or more data marking and structural attributes. Except for the first-level HTML elements, all other HTML elements can be optionally assigned a multiplicity attribute.

Each HTML element in turn can have as children elements any number of HTML elements, referred auxiliary patterns, skip elements, and `relation` elements specifying the or-relationship between two auxiliary patterns.

```
FIRST_LEVEL_HTML_ELEMENT ::= <HTML_TAG
                              ELEMENT_UNIQUENESS_ATTRIBUTE?
                              CLICKABLE_ELEMENT_ATTRIBUTE?
                              RESET_CONFIGURATION_STATE_ATTRIBUTE?
                              (DATA_MARKING_ATTRIBUTE)*
                              (STRUCTURAL_ATTRIBUTE)*>
                                  (STRUCTURAL_ELEMENT)*
                                | (SKIP_ALL_ELEMENT_FIND_HTML)*
                                | SKIP_ALL_ELEMENT_FIND_TEXT

                                | (STRUCTURAL_ELEMENT)*
                                  DATA_MARKING_TEXT_ELEMENT
                                  (STRUCTURAL_ELEMENT)*

                                | (STRUCTURAL_ELEMENT)*
                                  SKIP_TEXT_ELEMENT
                                  (STRUCTURAL_ELEMENT)*
                            </HTML_TAG>

STRUCTURAL_ELEMENT :: =           HTML_ELEMENT
                                | AUXILIARY_PATTERN_REFERENCE
                                | SKIP_SIBLING_ELEMENT
                                | PATTERN_RELATION_ELEMENT
```

```
HTML_ELEMENT ::= <HTML_TAG
                    ELEMENT_UNIQUENESS_ATTRIBUTE?
                    CLICKABLE_ELEMENT_ATTRIBUTE?
                    RESET_CONFIGURATION_STATE_ATTRIBUTE?
                    MULTIPLICITY_ATTRIBUTE?
                    (DATA_MARKING_ATTRIBUTE)*
                    (STRUCTURAL_ATTRIBUTE)*>
                        (STRUCTURAL_ELEMENT)*
                      | (SKIP_ALL_ELEMENT_FIND_HTML)*
                      | SKIP_ALL_ELEMENT_FIND_TEXT

                      | (STRUCTURAL_ELEMENT)*
                        DATA_MARKING_TEXT_ELEMENT
                        (STRUCTURAL_ELEMENT)*

                      | (STRUCTURAL_ELEMENT)*
                        SKIP_TEXT_ELEMENT
                        (STRUCTURAL_ELEMENT)*
                  </HTML_TAG>
```

A reference to an auxiliary pattern. An auxiliary pattern can be referred within a data pattern by giving its name as the only child text element of a `pattern` element.

```
AUXILIARY_PATTERN_REFERENCE ::=
  <pattern MULTIPLICITY_ATTRIBUTE?> AUXILIARY_PATTERN_NAME </pattern>
AUXILIARY_PATTERN_NAME ::= a defined auxiliary pattern name
```

The skip(sibling, Multiplicity) element. A skip sibling element must be either followed by an HTML element or a `pattern` element referring to an auxiliary pattern.

```
SKIP_SIBLING_ELEMENT ::=
  skip(sibling,MULTIPLICITY_SPECIFICATION) HTML_ELEMENT
  |
  skip(sibling,MULTIPLICITY_SPECIFICATION) AUXILIARY_PATTERN_REFERENCE
```

The relation element. A `relation` element is surrounded by two auxiliary pattern references and defines an or-relationship between those two disjunctive patterns. This element has a mandatory child meta text element `or`.

```
PATTERN_RELATION_ELEMENT ::= LEFT_AUXILIARY_PATTERN_REFERENCE
                              <relation> or </relation>
                             RIGHT_AUXILIARY_PATTERN_REFERENCE
LEFT_AUXILIARY_PATTERN_REFERENCE ::= AUXILIARY_PATTERN_REFERENCE
RIGHT_AUXILIARY_PATTERN_REFERENCE ::= AUXILIARY_PATTERN_REFERENCE
```

The names of the two disjunctive patterns can not be equal, meaning that they should refer to different patterns.

Data marking text element. An HTML element can have only one child data marking text element. The Wrapper extracts values of all the first-level children text elements of the HTML element and assigns them together to the data marking text element. A data marking text element name must match the
$data \backslash\backslash - tex \backslash\backslash - mar \backslash\backslash - [a - zA - Z0 - 9 \backslash\backslash -]+$ regular expression.

```
DATA_MARKING_TEXT_ELEMENT ::= a valid HTML text element matching
                              the data\\-tex\\-mar\\-[a-zA-Z0-9\\-]+
                              regular expression
```

The skip(STRING_VALUE) element. Only one instance of a `skip(STRING_VALUE)` can be contained within an HTML element. The Wrapper must visit the `STRING_VALUE` in the target code fragment but does not extract it. This element can be optionally followed by a data marking text element, meaning that values (except `STRING_VALUE`) of all the first-level children text elements of the HTML element will be extracted and assigned to the data marking text element.

```
SKIP_TEXT_ELEMENT ::= skip(STRING_VALUE) DATA_MARKING_TEXT_ELEMENT?

STRING_VALUE ::= a valid HTML string value
```

The skip(all) element. A `skip(all)` element can be followed by an HTML element, an auxiliary pattern reference, or a data marking text element. An HTML element can have multiple instances of the `skip(all)` element as the children, but only one of those can be followed by a data marking text element.

```
SKIP_ALL_ELEMENT::= SKIP_ALL_ELEMENT_FIND_HTM | SKIP_ALL_ELEMENT_FIND_TEXT


SKIP_ALL_ELEMENT_FIND_HTML ::=  skip(all) HTML_ELEMENT
                              | skip(all) AUXILIARY_ PATTERN_REFERENCE

SKIP_ALL_ELEMENT_FIND_TEXT ::=  skip(all) DATA_MARKING_TEXT_ELEMENT
```

Auxiliary pattern

An auxiliary pattern is syntactically similar to a data pattern, except that it can only have one first-level child HTML element. It is identified from the other patterns by the `data-att-met-pattern-type = "auxiliary"` attribute.

```
AUXILIARY_PATTERN ::= <pattern data-att-met-pattern-type="auxiliary"
                        PATTERN_NAME_ATTRIBUTE
                        MULTIPLICITY_ATTRIBUTE?>
                        AUXILIARY_PATTERN_BODY
                      </pattern>

AUXILIARY_PATTERN_BODY ::= FIRST_LEVEL_HTML_ELEMENT
```

The multiplicity of data and auxiliary patterns can be specified either where the pattern is specified or where it is referenced in the specification of another pattern. The Wrapper first looks for the multiplicity attribute where the pattern is referenced and if it finds it there then considers it. If not, it tries the pattern specification. In the case of not explicitly defined multiplicity, it relies on the default multiplicity.

6.5 Chapter Summary

Configuration options, descriptive information associated to an option, and constraints are the key variability data to be extracted from the Web pages of a configurator. Our observations reveal that the developers of a Web configurator usually use a set of templates to automatically generate the Web pages of the configurator. We thus use these templates in reverse order to extract variability data encoded in the code fragments generated using these templates. To this aim, we proposed the notion of variability data extraction pattern. The HTML-like structure of a pattern tells the Wrapper to look for code fragments whose structure is similar to the structure of the pattern. Data marking text elements and attributes specified in a pattern denote the data items that will be extracted from those similar code fragments.

In this chapter, we explained the syntax and semantics of variability data extraction patterns. We presented several real world examples taken from different Web configurators to show how patterns can be used to extract data from such configurators. We also showed how patterns can deal with well-known Web data extraction challenges such as multi-instantiated elements, distinctive elements, optional data, and noisy data. We also gave a context-free grammar to describe the legal syntax of variability data extraction patterns to make them unambiguous for tool implementation.