PReCISE Research Centre

Faculty of Computer Science

University of Namur

Reverse Engineering Web Configurators

Author:
Ebrahim Khalil Abbasi

Supervisor:
Prof. Patrick Heymans

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

January 2014

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 patterns. 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 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.

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

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 the same 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 [Pin93]. While *standardized* products are best produced in an *assembly line* mass production environment [HW79], 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 [HMR08]. 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 [Pin93].

Mass customization is producing goods and services to meet individual customers' needs with near mass production efficiency [TJ07]. It in fact aims to provide a trade-off between product variety (i.e. flexibility) and production cost (i.e., efficiency) [HW79, Kot95]. 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 [Pin93].

One key element in a mass customization strategy is to build products by selecting, combining and possibly adapting a set of standard modules. On the other words, mass customization is mass production of standard modules and customer-initiated assembly of customized products based on the use of modules [HMR08]. 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 [BA06].

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, she 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 [BA06, HOM98, vHK02, TP03, FP02, XHK05, OLN06]. 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 [SW98].

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 [PBvdL05, SLRS12, GWJV+09, RvdADtH08].

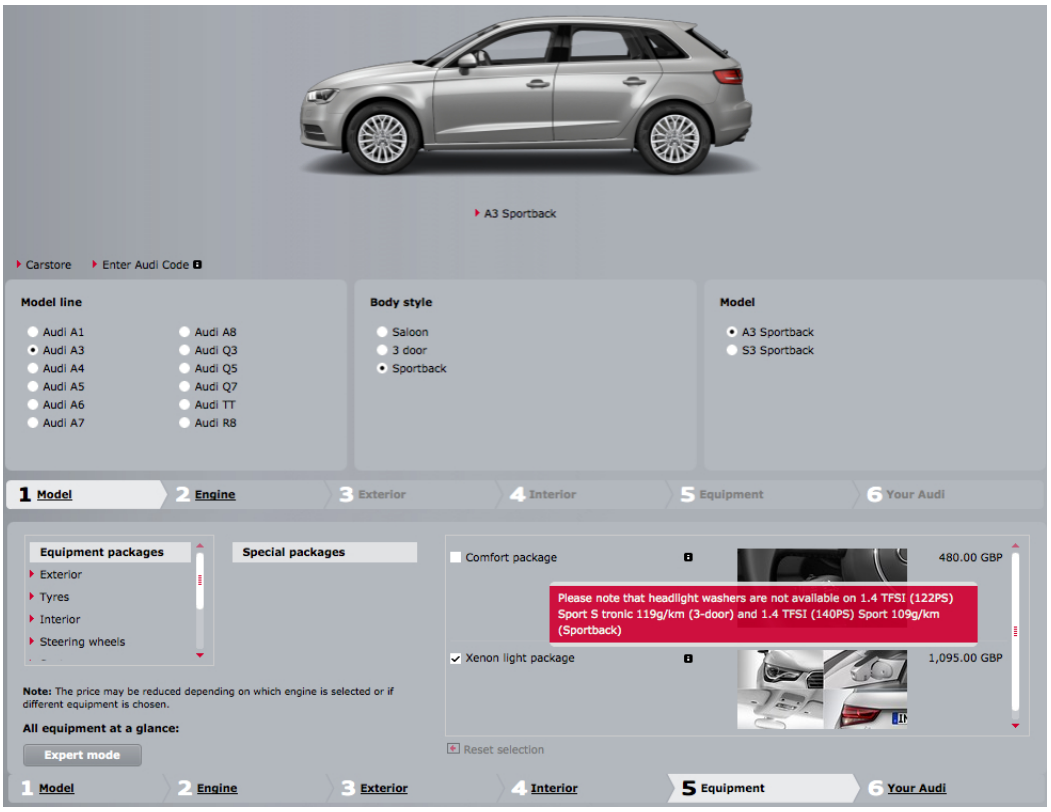


FIGURE 1.1: Audi Web configurator (`http://configurator.audi.co.uk/`, August 7 2013).

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 [RP04] which leads companies to experience a loss of sales [TPF12]. 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 [BA06]. An empirical study carried out by Rogoll

and Piller shows that existing configurators can not fulfil optimal requirements from companies' and customers' perspectives [RP04]. For instance, some configurators do not show the customer a picture of the whole proposal and functionalities the system offers, do not provide explanations how to navigate through the configuration process, etc. The authors concluded that the state of the design of the front ends of configurators is rather weak. Von Hippel also presented that existing configurators just enable customers to select products out of alternatives but do not facilitate customer learning [vH01].

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, adapted, 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 [Jan10, BSRC10, HXC12].

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 [BAH+12]. 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 not in 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 scalable 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 *scalable* 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* by static analysis of the source code of a Web page. We use the *variability data extraction patterns* (*vde* patterns) to specify variability data to be extracted from Web pages. We also implemented a *Web Wrapper* to support the process of extracting data. It seeks, finds, and extracts variability data from a page given a set of *vde* patterns, and then transforms the extracted data into structured variability data.

RQ3 *How to (semi-)automatically extract 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?*

We developed a *Web Crawler* to deal with dynamic behaviour of Web configurators. It automatically explores the configuration space (e.g., navigates through the configuration steps) and configures options in a given region of the page. If the exploration and configuration actions add new data to the page, the Wrapper extracts the newly added data.

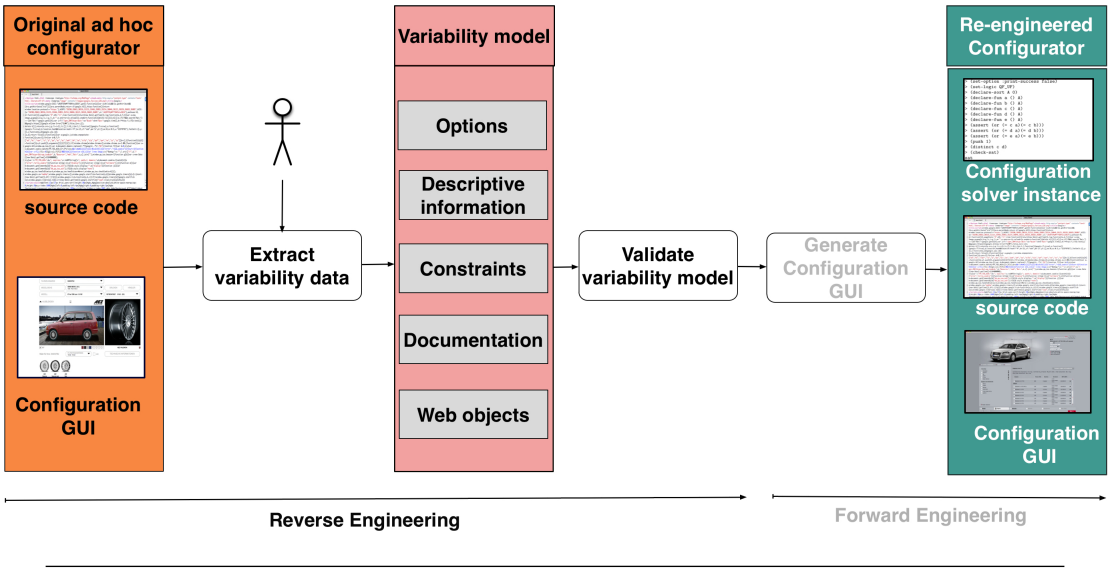


FIGURE 1.2: Re-engineering process [BAH+12].

1.4 Contribution

The main contribution of this thesis consists of:

C1 *A systematic study and understanding of Web configurators.* Although Web configurators have been studied from usability and visual aspects [RP04, SBFF09, TPF12], 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 a tool.* We implemented a reverse-engineering tool that consists mainly of two collaborative components: *Web Wrapper* and *Web Crawler*. The Web Wrapper seeks, finds, and extracts variability data from a page given a set of *vde* patterns, and then transforms the extracted data into structured variability data. The extracted data is hierarchically organized and serialized in an XML format. The Web Crawler automatically explores the configuration space (i.e., all objects representing variability data) and simulates users' configuration actions. It systematically generates dynamic variability data which is then extracted by the Wrapper. We also implemented a module that transforms the

output XML file into a feature model. We rely on the *Text-based Variability Language* (TVL) [CBFH10, CBH11a] to represent feature models.

1.5 Reader's Guide

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 algorithm 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 publications of the author. We list below the relevant papers:

Journal

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

Conference

- E. 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. 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. 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. 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

- Q. Boucher, E. 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 ICSE12*, pages 29–32, Zurich, Switzerland, 2012. IEEE Computer Society. (**Chapter 1**)

Doctoral Symposium

- E. 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 [Ins13]. Products in a product line are produced from *a common set of core assets* in a prescribed way [CN02], therefore, all that products share a significant amount of *commonality* and differ in their specific configuration of *variability* [Sto12]. A commonality thereby is a characteristic of all products of a product line, and a variability, in contrast, is a varying characteristic that its 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 [BGGB02].

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 [CGR+12] which is where configuration takes place.

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.* [KCH+90] 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 [CHS08, ALMK08, BBRC06, CZZM05].

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 FM.

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 a "Basic", "Colour" or "High resolution" screen but only one of them. 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.

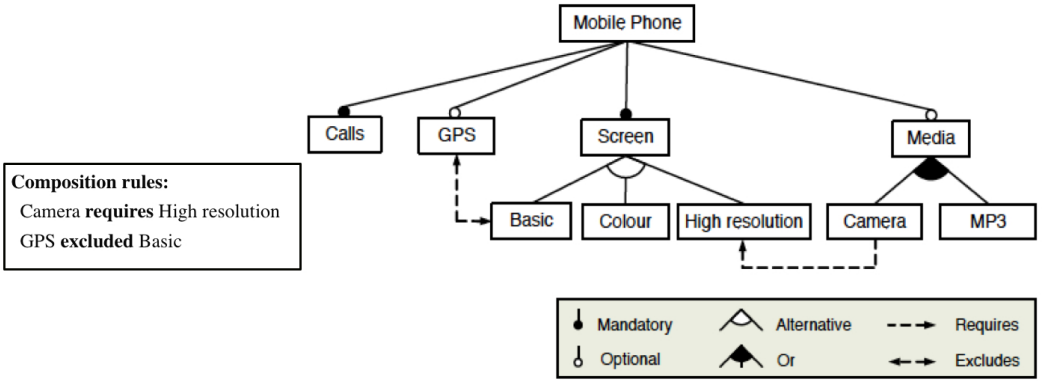


FIGURE 2.1: A FM for mobile phone systems ([BSRC10]).

After the initial proposal by Kang *et al.*, several FM extensions have been proposed [CE05, GFA98, MCHB11a]. Several feature modelling languages have been also designed and supported by editing, debugging, analysis, and configuration tools [ACLF13, TKB+12, MBC09, BSTRc07, Beu12, BCW11, BS09, CBH11a]. 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 [SHT06]. 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 [BSRC10]). 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 [CBH11a, CBFH10] 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".

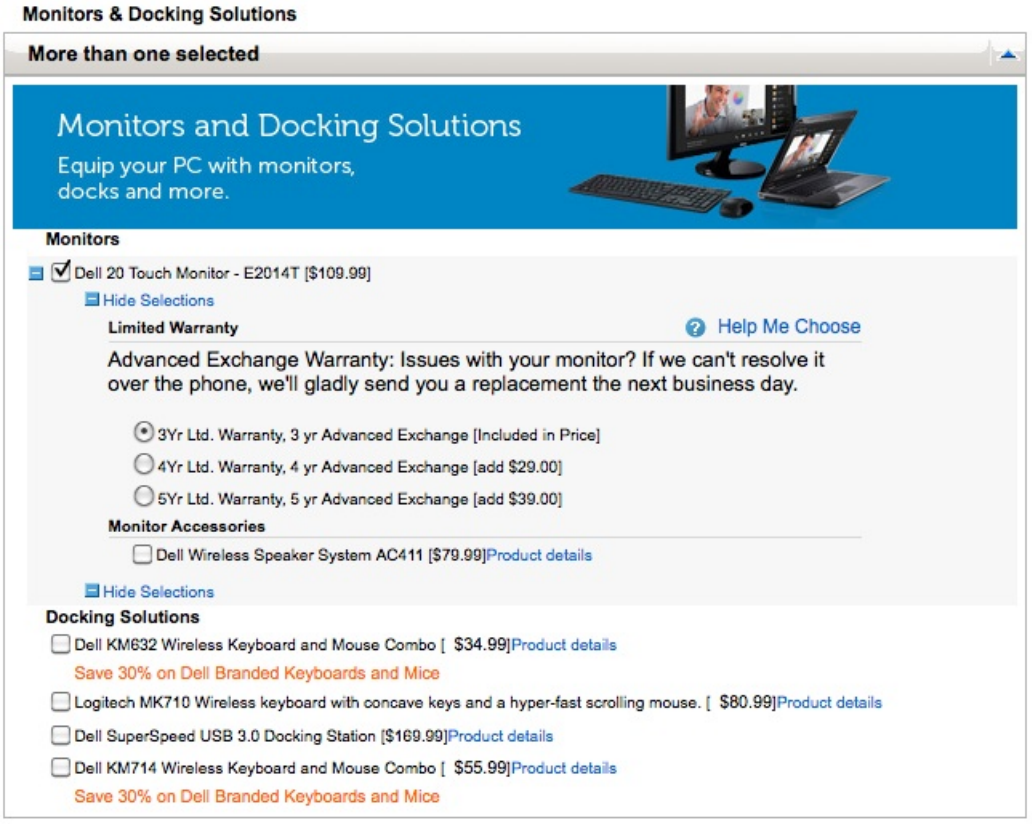


FIGURE 2.2: Dell Web configurator (`http://www.dell.com//`, October 25 2013).

2.2 Web applications

Conallen [Con99] 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 [Tra05].

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 (JSP), Java Servlets, Active Server Pages (ASP), PHP, XML, etc. [TH01].

```
root Monitors_Docking_Solutions {
    group allOf {
        Monitors {
            group someOf {
                Dell_20_Touch_Monitor_E2014T {
                    real price is 109.99 ;
                    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 79.99 ;
                        }
                    }
                }
            }
        },
        Docking_Solutions {
            group someOf {
                Dell_KM632_Wireless_Keyboard_and_Mouse_Combo {
                    real price is 34.99 ;
                },
                Logitech_MK710_Wireless_keyboard_with_concave_keys_and_a_hyper_fast_scrolling_mouse {
                    real price is 80.99 ;
                },
                Dell_SuperSpeed_USB_3_0_Docking_Station {
                    real price is 169.99 ;
                },
                Dell_KM714_Wireless_Keyboard_and_Mouse_Combo {
                    real price is 55.99 ;
                }
            }
        }
    }
}
```

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

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.

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

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 part, 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 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 [Tra05]. 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.

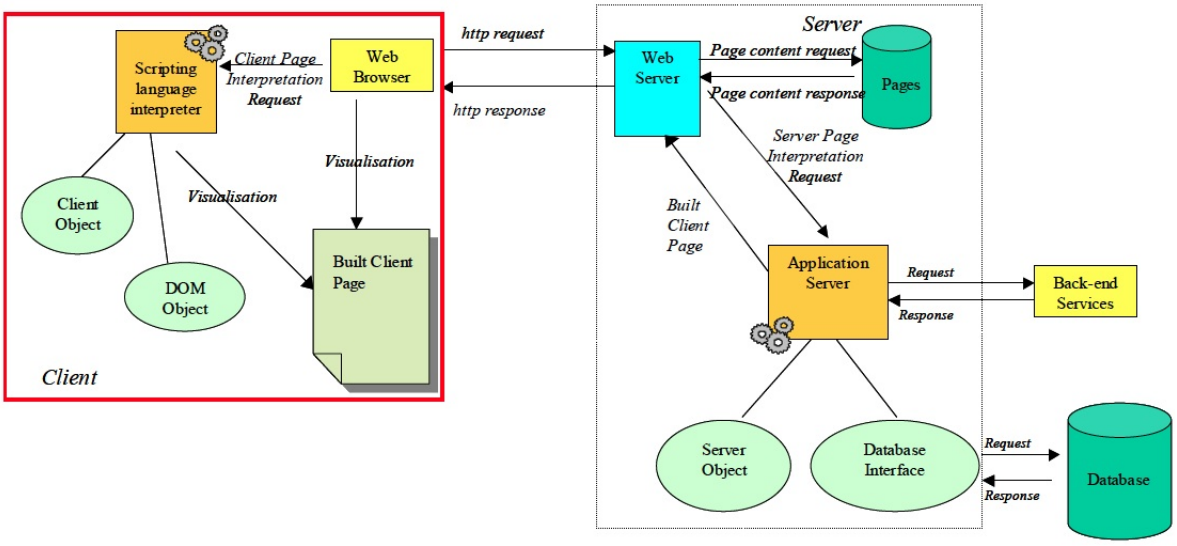


FIGURE 2.4: Architecture for Web applications [Tra05].

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 [hLC12]. 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] 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 [AHA+13]. 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 [hd11]. 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., "Mirrors", "Windows"...).

Configuration 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 [RP04, SBFF09]. 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).

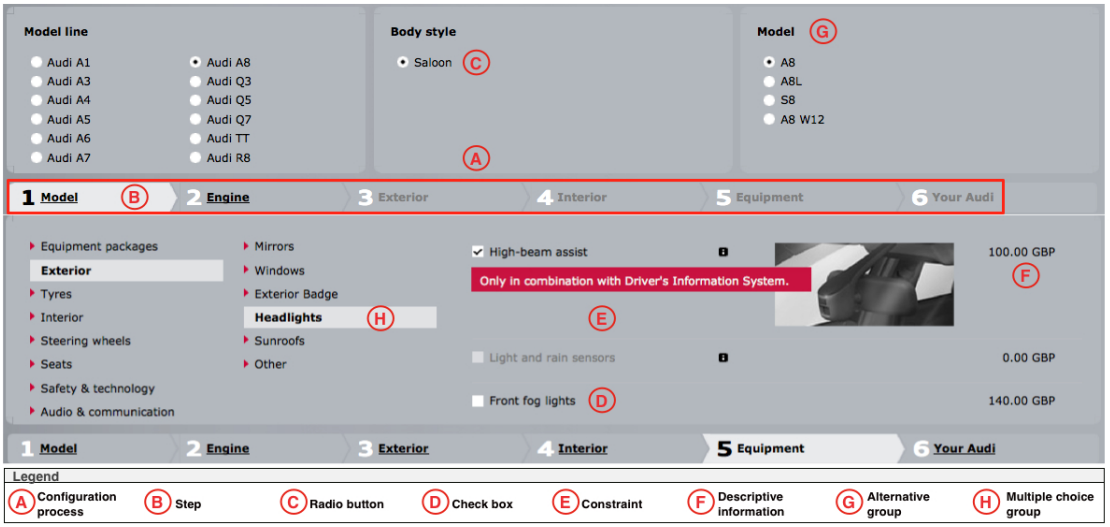


FIGURE 3.1: Audi Web configurator (`http://configurator.audi.co.uk/`, August 7 2013).

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- 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 [hd11], 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

[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/

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.

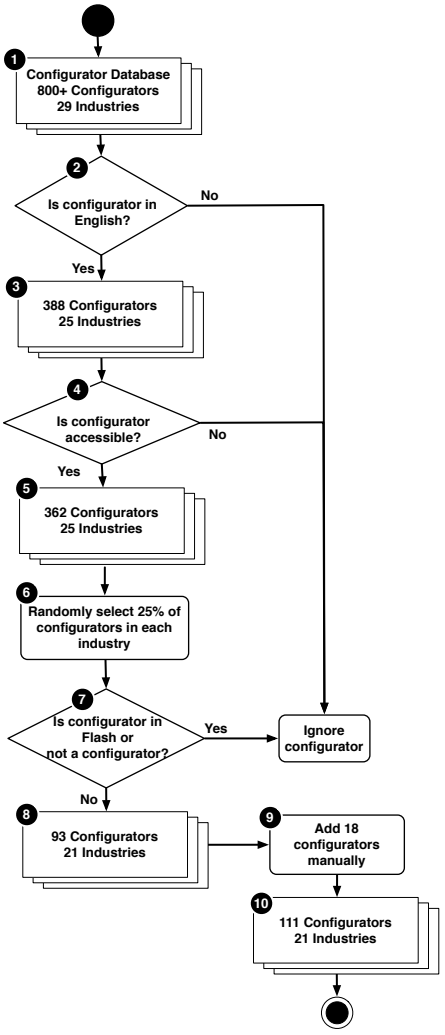


FIGURE 3.2: Configurator selection process.

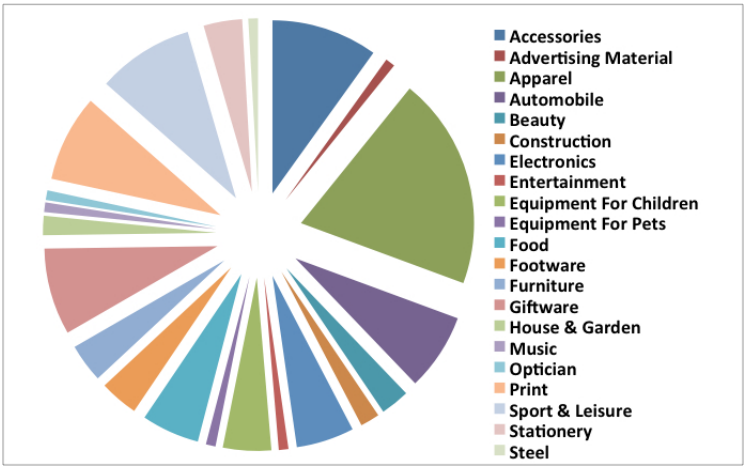


FIGURE 3.3: Distribution of selected configurators by industry.

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.

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

[4] http://jquery.com/

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.

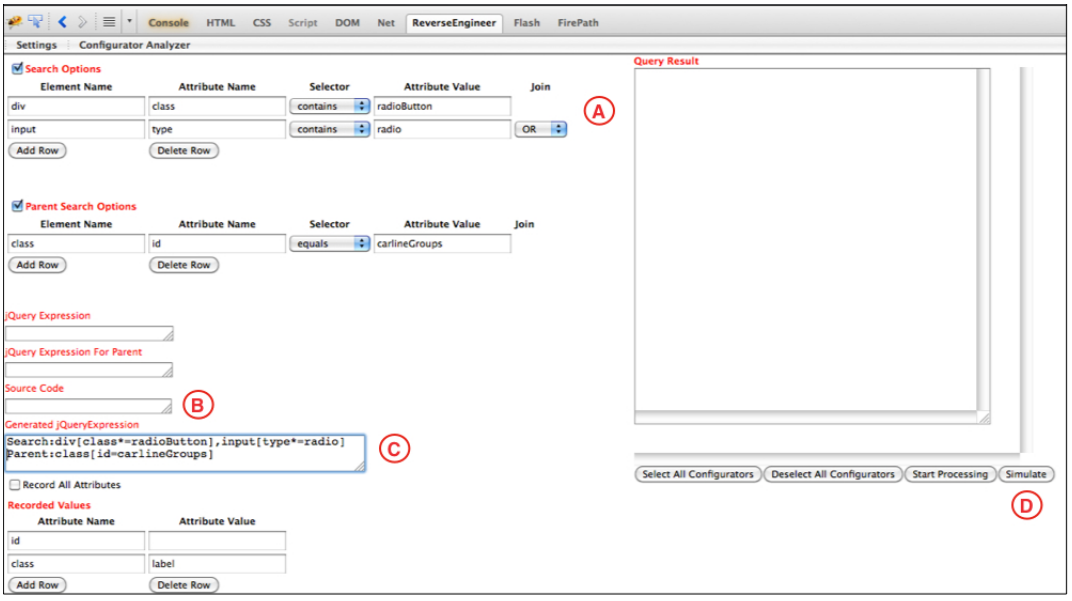


FIGURE 3.4: The Firebug data extraction extension.

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* [CKGS06] 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 (Ⓓ).

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).

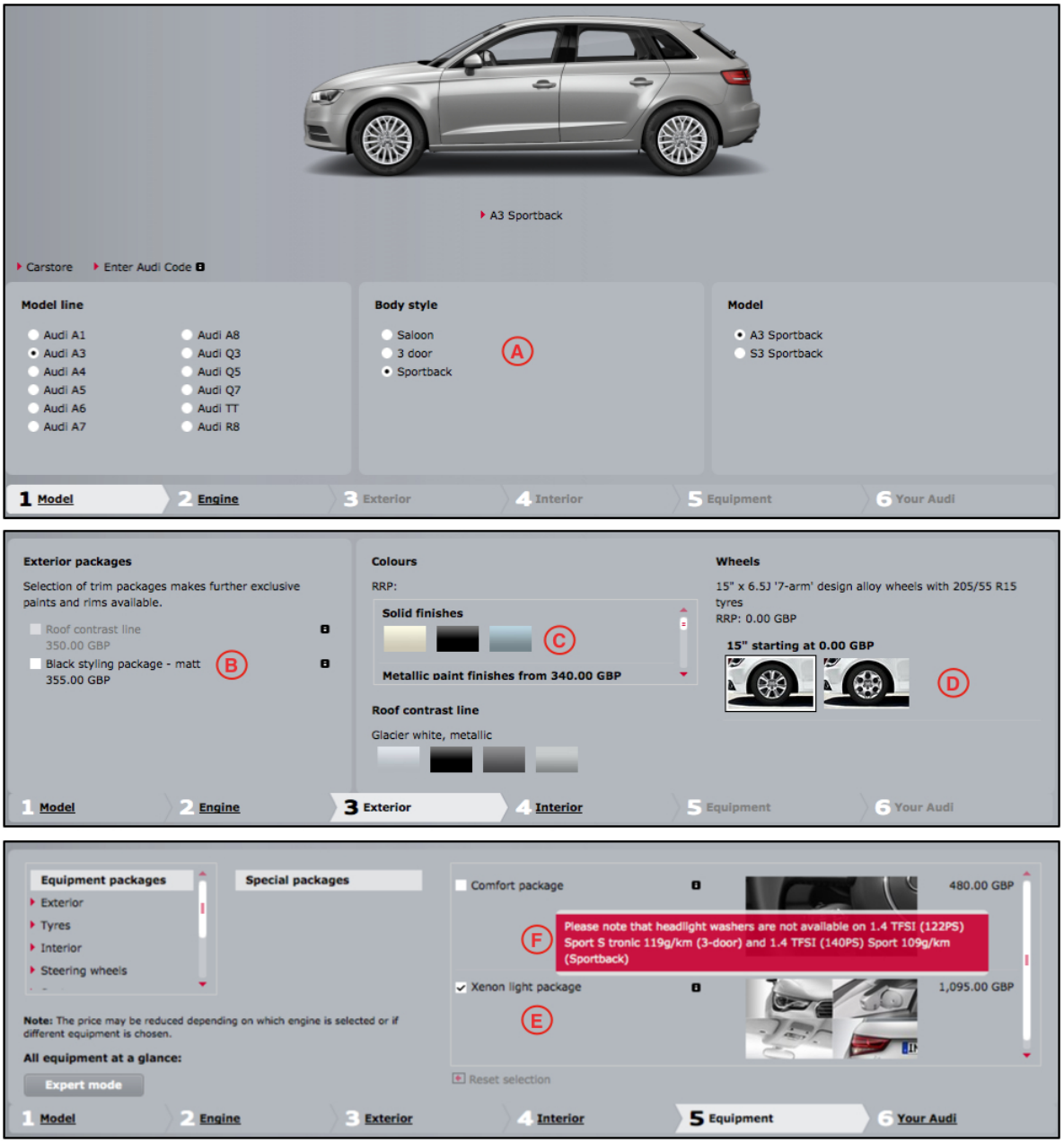
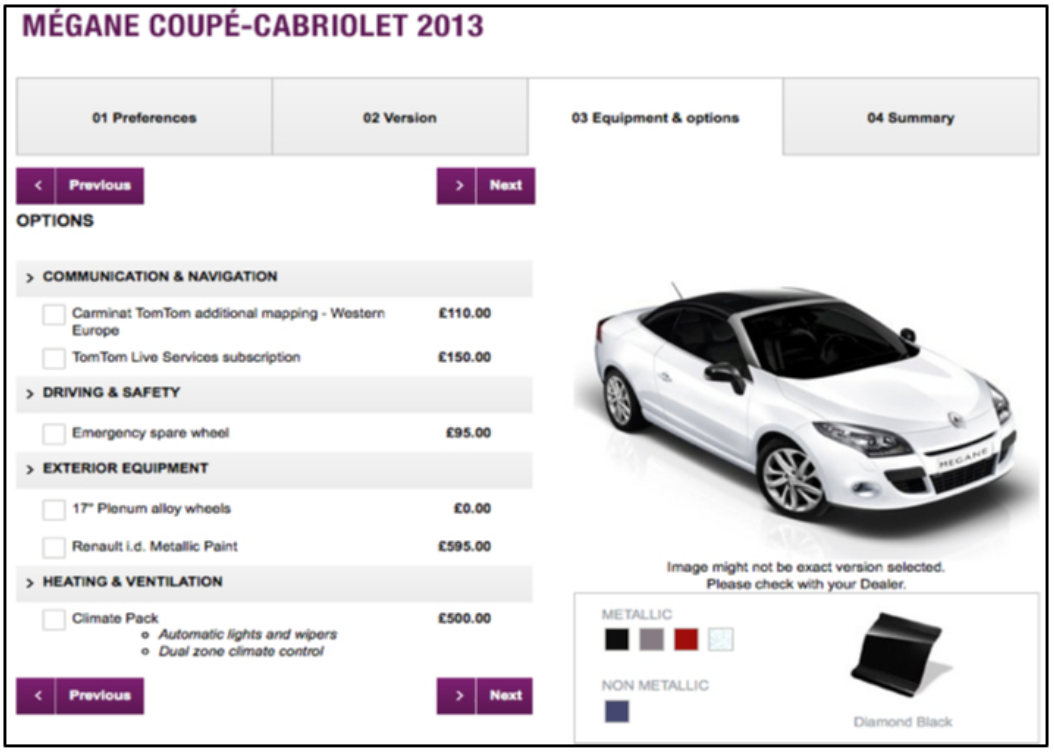


FIGURE 3.5: Presentation of configuration-specific objects (`http://configurator.audi.co.uk/`, July 3 2013).

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 Ⓓ).



Code Fragments

```
1   <td>
2     <input type="checkbox"/>
3     <label>
4       <span class="sectionText"> Renault i.d. Metallic Paint </span>
5       <span class="SectionPrice"> £595.00 </span>
6     </label>
7   </td>

8   <td>
9     <input type="checkbox"/>
10    <label>
11      <span class="SectionText"> Climate Pack
12        <ul>
13          <li> Automatic lights and wipers </li>
14          <li> Dual zone climate control </li>
15        </ul>
16      </span>
17      <span class="SectionPrice"> £500.00 </span>
18    </label>
19  </td>
```

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

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 (①).

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

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.

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

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

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%

widgets are: *combo box item*, *image*[6], *radio button*, *check box* and *text box*. We also observed that some widgets were combined with images, namely, *check box*, *radio button*, and *combo box item*. Option selection is performed by either choosing the image or using the widget. The *Other* category contains various less frequent widgets like *slider*, *label*, *file chooser*, *date picker*, *colour picker*, *image needle*, and *grid*.

Grouping. Grouping is a way to organise *related* options together. For instance, a group can contain a set of colours or the options for an engine. Three different semantic

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

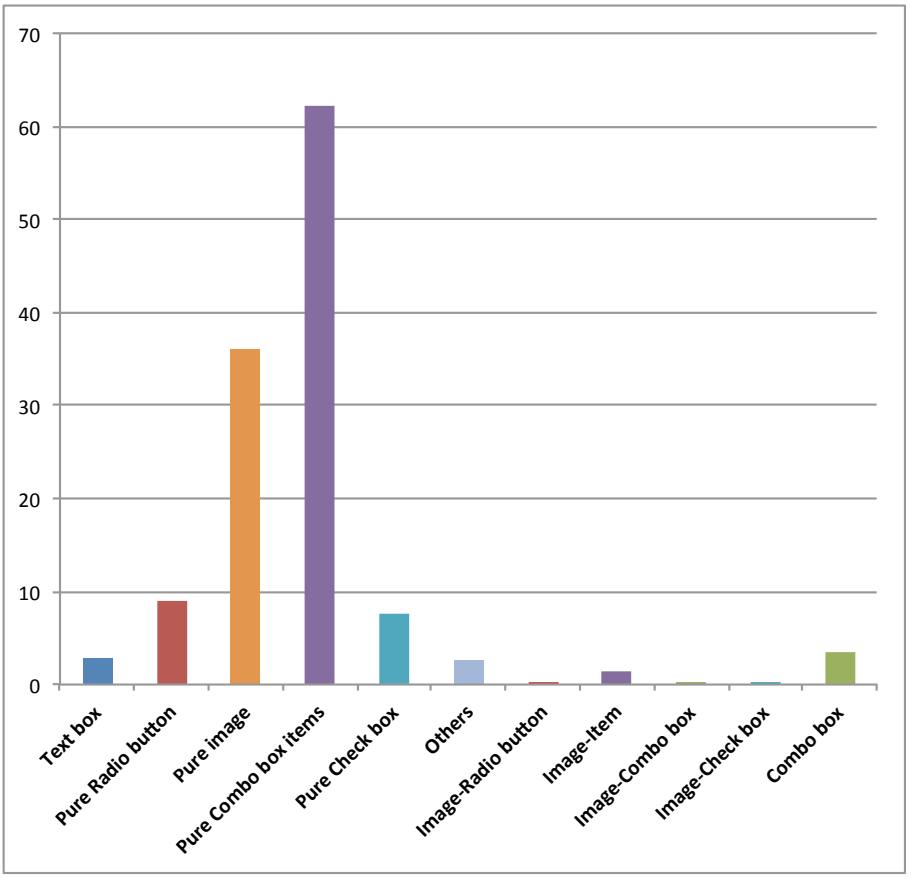


FIGURE 3.8: Widget types in all the configurators.

constraints can apply to a group. For *alternative* groups, one and only one option must be selected (e.g., the "Models" 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 [CK05]), 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:

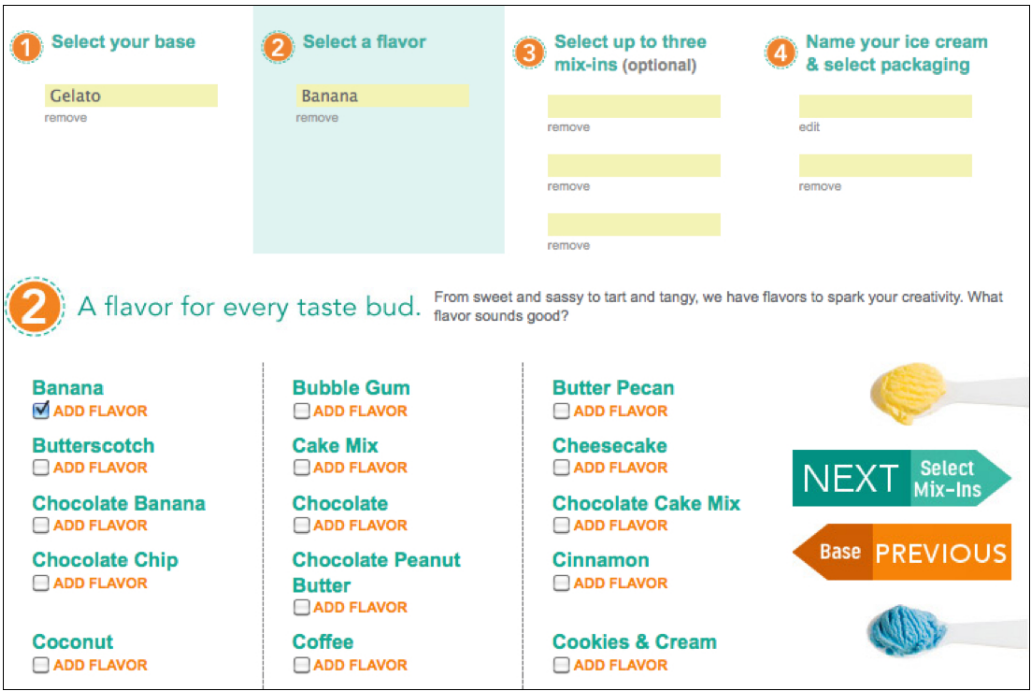


FIGURE 3.9: Interval group (`http://www.ecreamery.com/createyourown.html`, August 9 2013).

- *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.

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 [MCHB11b] (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.

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 color option in Figure 3.12 (Ⓐ) the user should enter a valid color name. Also, as for the text style, if she wants a sample text she should put the "As Sample" string in the text input (Ⓑ).

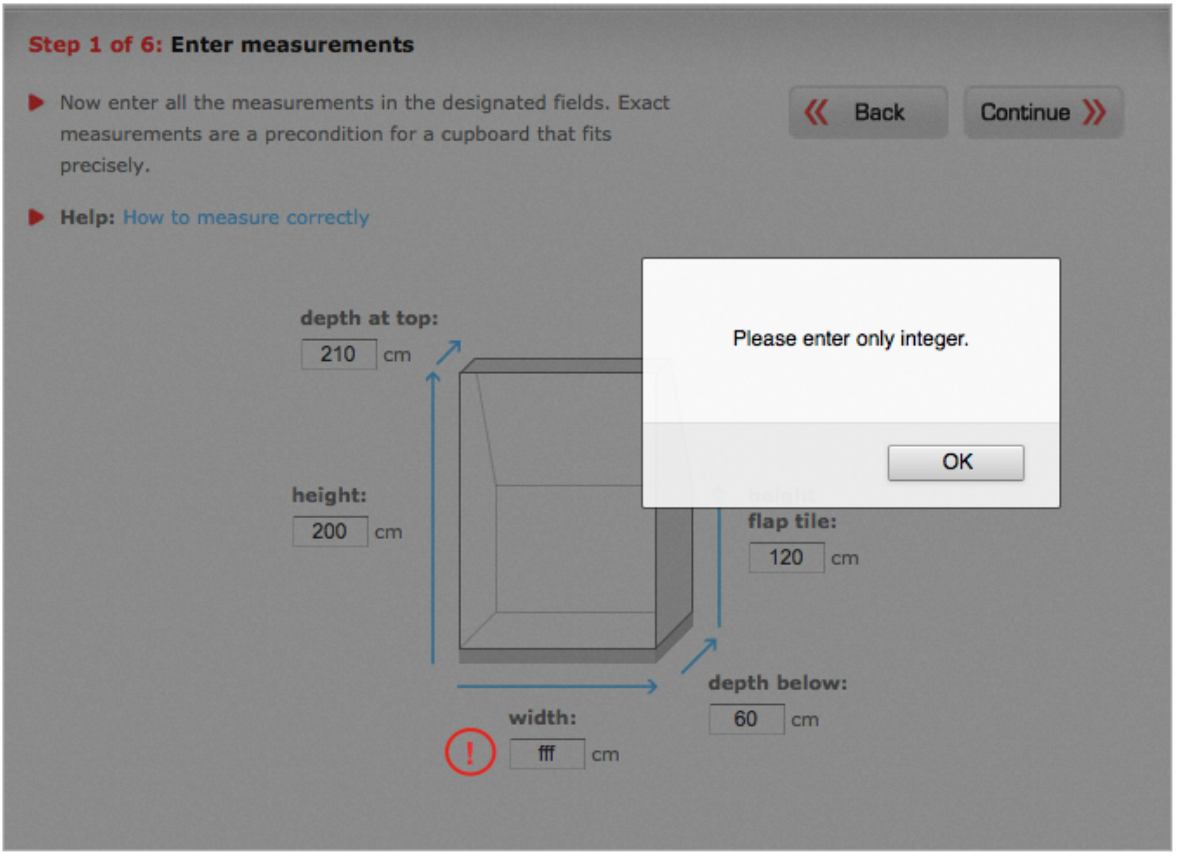
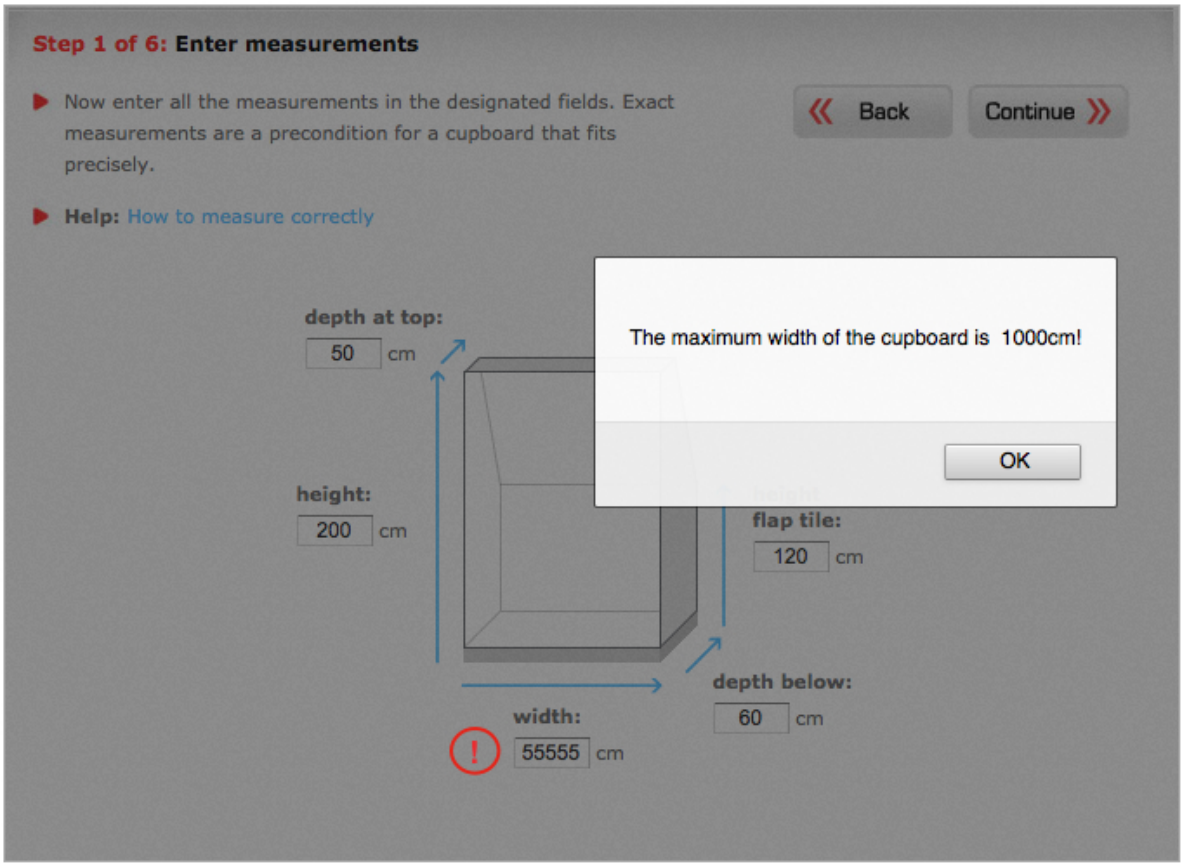


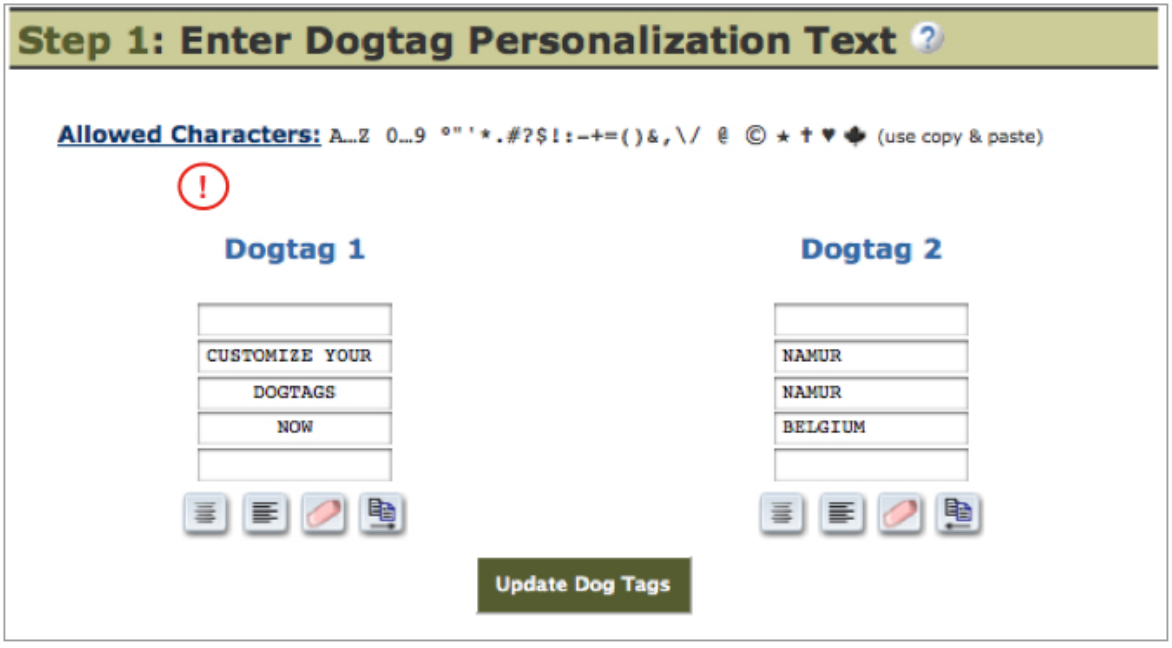
FIGURE 3.10: Type correctness constraint (`http://www.cupboardyourway.co.uk`, June 13 2013).

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

- *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.



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



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

FIGURE 3.11: Range control constraints.

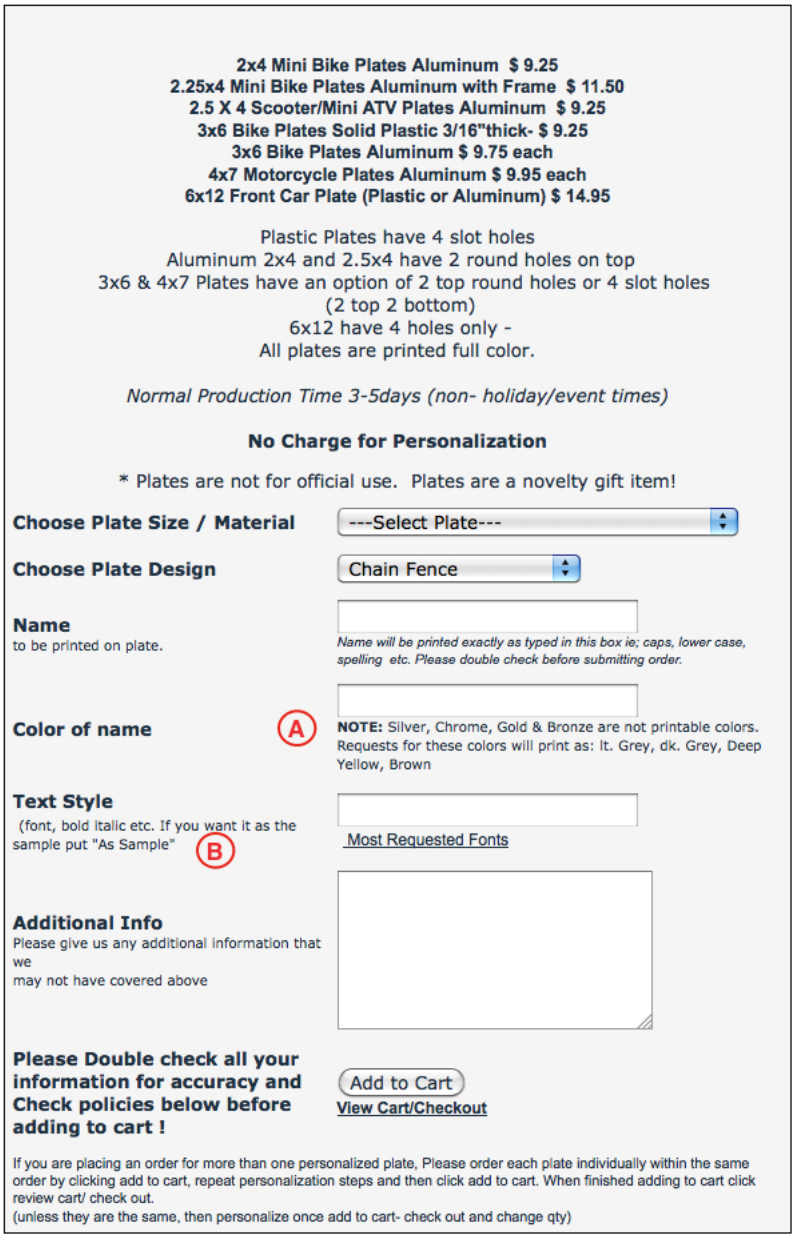


FIGURE 3.12: Case-sensitive values (`http://www.personalizedbikeplates.com/`, June 13 2013).

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 - buckle activated". 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* [BSL+10]. 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.

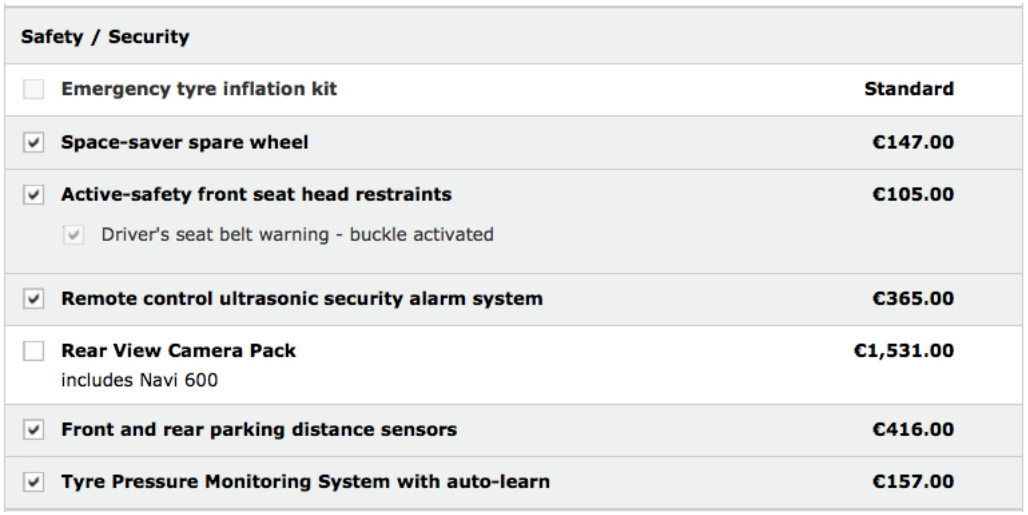


FIGURE 3.13: Automatic decision propagation (`http://www.opel.ie/`, August 9 2013).

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

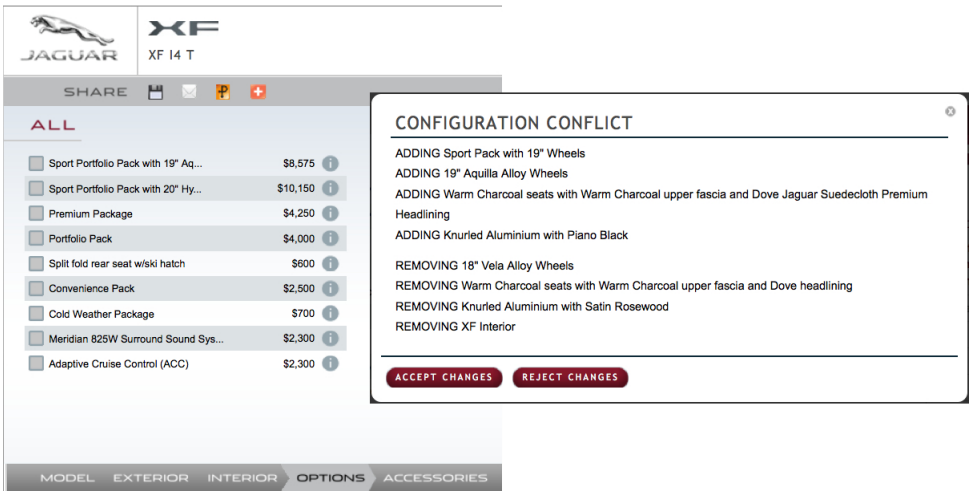


FIGURE 3.14: Controlled decision propagation (`http://www.jaguarusa.com/`, July 31 2013).

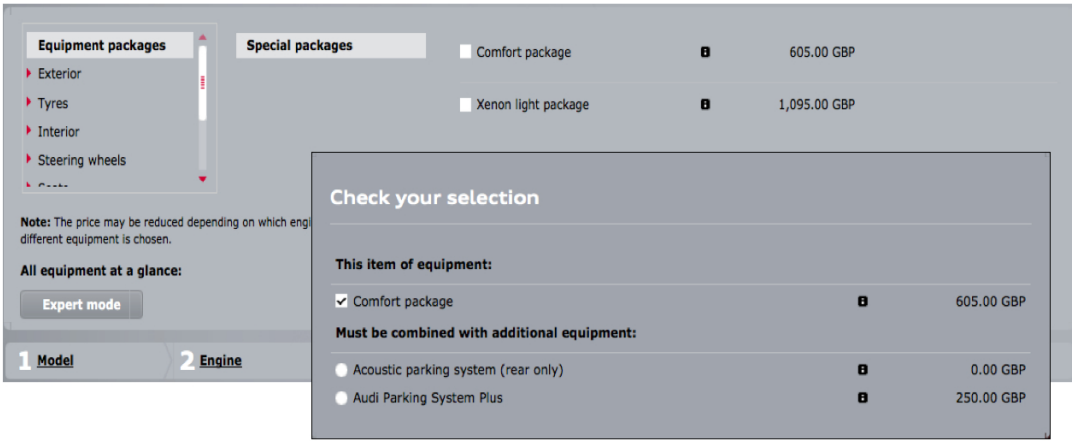


FIGURE 3.15: Guided decision propagation (`http://configurator.audi.co.uk/`, August 9 2013).

the next configuration step. We call this *batch* consistency checking. Among the 83 configurators supporting both formatting and cross-cutting constraints, 76% implement interactive and 59% implement batch consistency checking patterns. Some configurators implement both mechanisms, depending on the constraint type.

Undo. This operation allows users to roll back on their last decision(s). Among all configurators in the survey, only 11% support undo. 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 [RP04, SBFF09, TPF12, HMR08, HXC12]. 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 [RP04], 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 [HMR08] 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 [HXC12] 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 [HMR08], which are known to improve usability [RP04].

- *Auto-completion* allows users to configure some desired options and then let the configurator complete undecided options [JBGMS09]. 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 [RP04, SBFF09, HMR08]. 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, *viz.* 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 blowup precludes it. The impact this has on the completeness of our results is hard to predict. This, however, does not affect our observations related to the absence of verification of constraints textually documented in the Web pages.

Second, arbitrary 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.* [RP04] 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.* [SBFF09] analysed 126 Web Configurators among those in [hd11]. 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 [SBFF09]. Trentin *et al.* [TPF12] 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 [CCI90]. 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 [PCMA07]. Web applications can be categorised into three classes [TH01]:

- **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 [PCMA07]. 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 **??**). 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 [PCMA07, Bou06, dS10] for a more detailed state of the art in reverse engineering Web applications.

GUITAR. GUITAR [NRBM13] 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.

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

- *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.
- *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) [VBS01] 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.

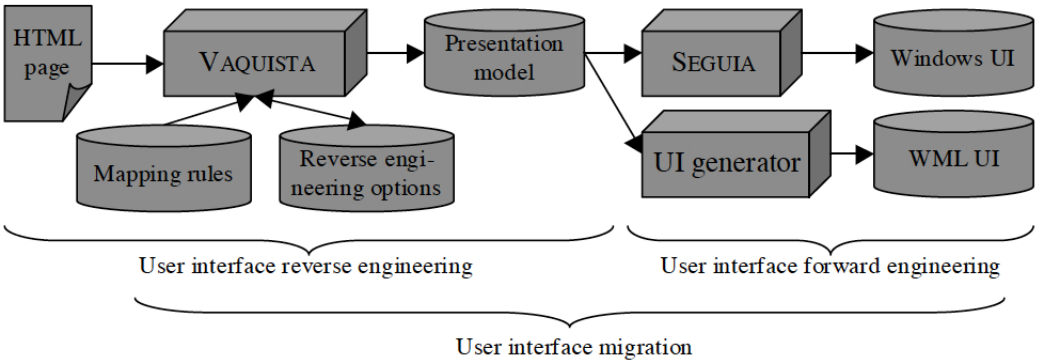


FIGURE 4.1: UI migration process with VAQUISTA ([VBS01]).

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 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 [dS10, JS10] 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 [Vis04a] and code slicing. Strategic programming allows novel forms of abstraction and modularization that are useful for program construction in general [Vis04b]. 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 [Tip95].

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.

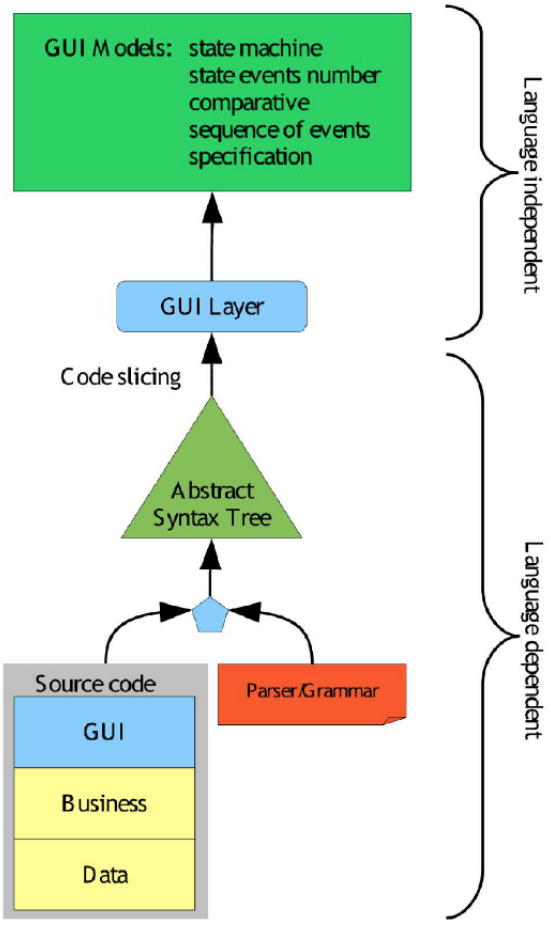


FIGURE 4.2: GuiSurfer's tool architecture ([JS10]).

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 [MvDL12]. 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.

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

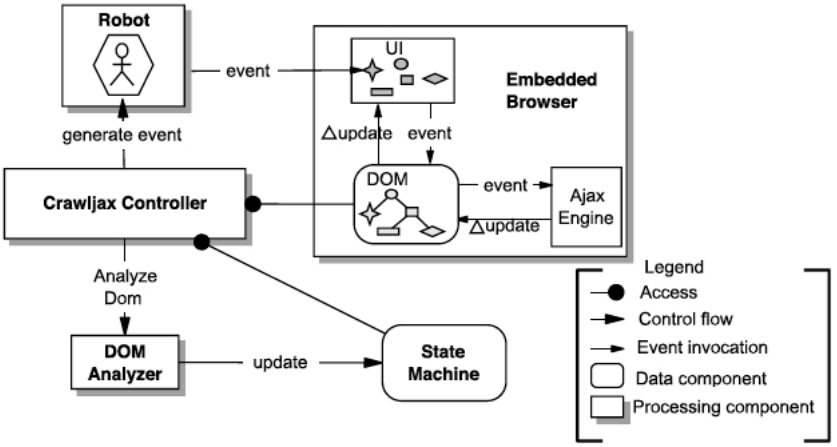


FIGURE 4.3: Processing view of CRAWLJAX ([MvDL12]).

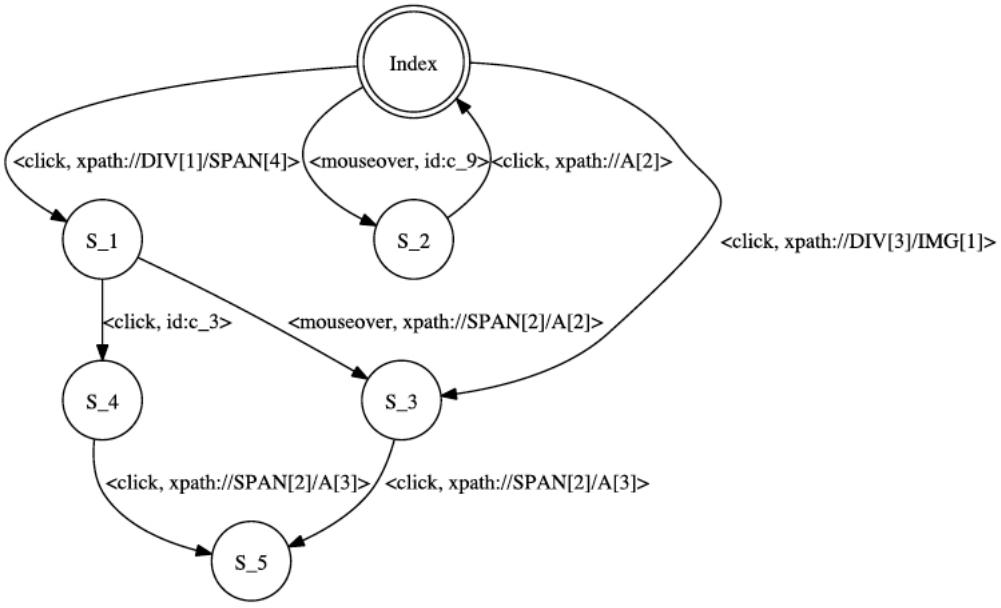


FIGURE 4.4: The state-flow graph of an AJAX site created by CRAWLJAX ([MvDL12]).

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 [DLFT04, Tra05] 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 first 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 identified 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 defined 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 final 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.

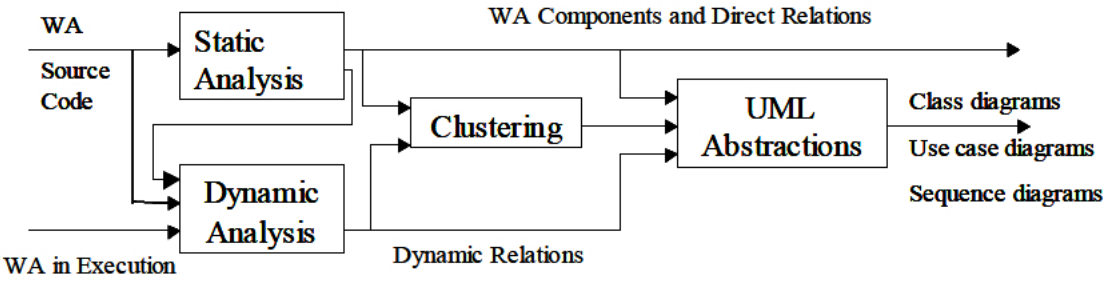


FIGURE 4.5: The reverse-engineering process in the WARE approach ([Tra05]).

The goal of the WARE approach is to retrieve, from the source code of a Web application, all information that can be useful to support the comprehension and maintenance of the application. To achieve this goal, it mostly focuses on recovering models at a high level of abstraction, i.e. at business layer. WARE does not consider fine-grained objects within a Web page (such as widgets) and their relationships. The data layer of an application is not targeted by this approach as well.

Other related works. Other methods that address the problem of reverse engineering Web applications have been defined. Maras *et al.* [MCC12] proposed a semi-automatic method for extracting client-side Web application code implementing a certain behaviour (e.g., the code executed as a click event-handler on a button).

In [DLW05], a source-code-independent reverse-engineering approach for Web applications is described. It presents *Revangie*, a tool that is able to recover the form-oriented model of a dynamic Web interface of a Website without looking at the source code. Therefore, the tool is independent of the languages, platforms, and architectures the Website is implemented in.

Ricca and Tonella [RT01] presented the *ReWeb* tool which provides a set of analyses to extract a description of a Website's main architectural features and of their evolution.

Bellucci *et al.* [BGPP12] implemented a tool to perform reverse engineering of interactive dynamic Web applications into MARIA [SS09], a model-based framework for describing interactive applications at high abstraction levels. To reverse engineer the HTML code, the tool parses the HTML tags within the given Web page and converts them into Concrete User Interface (CUI) elements in MARIA, according to the specification of the MARIA language. The tool also supports migration of the JavaScript code and the CSS. The generated models in MARIA can be used for various purposes (adaptation tools for multi-device environments, documentation, etc.).

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 [BGG09, LRNdST02]. 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 [FMFB12].

The challenging aspect of wrappers is that they must be able to recognize the data of interest among many other uninteresting pieces of text [LRNdST02]. Moreover, when the content or structure of data of a page is changed, the wrapper should be adapted accordingly to keep working properly [BGG09, FMFB12]. 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 [LRNdST02].

Due to the difficulty in manually writing and maintaining wrappers [LRNdST02], several approaches have been proposed to address the problem of automatically generating wrappers [CMM01, Kus00, LPH00, MMK01, SA01] 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 [PHH05]. Many automatic tools are either inaccurate or make many assumptions [ZL05].

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.* [LRNdST02] 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 [CM98, HMGM97, AM99], *HTML-aware* tools rely on the formal structure of Web pages to extract data [CMM01, LPH00, SA01, BFG01b], *natural language processing-based* (NLP) tools extract data of interest from highly grammatical and natural language documents [Fre00, Sod99], *ontology-based* tools recognize and extract data presented in documents given an ontology [ECJ+99], *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 [Ade98, LRNdS02], and *wrapper induction* tools generate extraction rules from a given set of training examples [HD98, Kus00, MMK01]. There are cases where a tool could fit in two or more of the identified groups.

Chang *et al.* [CKGS06] 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 [HMGM97, CM98, AM99, SA01, LPH00] 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 [Fre98, CM99, KWD97, Sod99, Ade98, HD98, MMK99, LRNdS02] 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 [CK04, HK05]. Unsupervised Web data extraction systems [CMM01, WL02, WL03, AGM03, LGZ03, CL01] 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. [FMFB12] 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 [SA99, BFG01b, ZL05, MHL03] 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 [Kus00, HD98, MMK01, PHH05] 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 [CMM01] 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 [LRNdST02, CKGS06, FMFB12]. NLP-based tools are useful to solve specific problems such as extraction of facts from newspaper articles, email messages etc. [FMFB12], and from Web pages consisting of free text [LRNdST02]. 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 [LRNdST02].

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 [LRNdST02, FMFB12], we mostly focus on modelling-based tools.

STALKER. STALKER [MMK01] 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.

```
1: <p> Name: <b> Yala </b><p> Cuisine: Thai <p><i>
2: 4000 Colfax, Phoenix, AZ 85258 (602) 508-1570
3: </i> <br> <i>
4: 523 Vernon, Las Vegas, NV 89104 (702) 578-2293
5: </i> <br> <i>
6: 403 Pico, LA, CA 90007 (213) 798-0008
7: </i>
```

FIGURE 4.6: An example input to STALKER ([MMK01]).

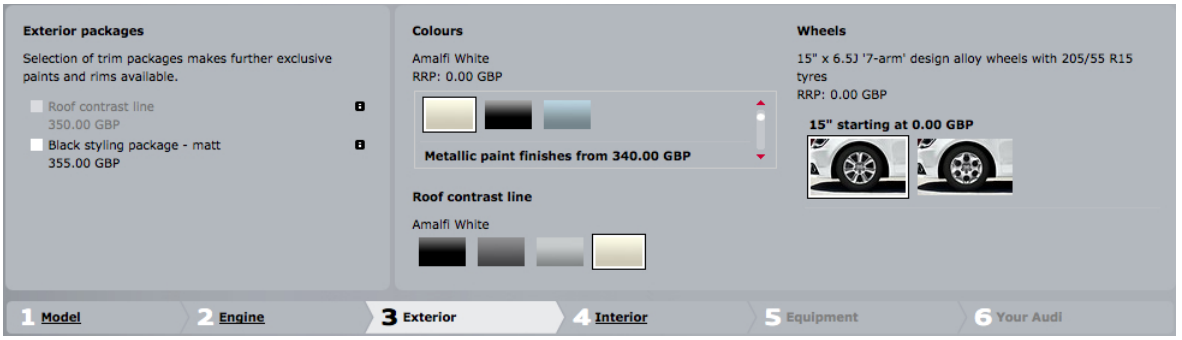
STALKER and other wrapper induction tools generate delimiter-based extraction rules derived from a given set of training examples and rely on formatting features that implicitly delineate the structure of the pieces of data found [LRNdST02]. These tools assume that the desired data to be extracted are surrounded by common tokens [CKGS06] 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 to be extracted are located in the tag attributes (`key` and `id` in Figure 4.7(b)) of the corresponding HTML 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. Moreover, as it is shown in Figure 4.7(b), all formatting features are encoded using `div` elements and the use of rules like

$$\mathbf{R} = \text{SkipTo}(<\text{div}>)$$

is not accurate enough to identify a specific element in the page. In addition, the authors in [CMM01] presented nested structures that STALKER cannot handle.

Data Extraction By Example (DEByE). DEByE [LRNdS02, RNLS99] 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.

The DEByE approach is restricted to the specific domain of Web applications with data rich pages [LRNdS02] which contain uniform and regularly formatted data records [PHH05].



(a) Image options (`http://configurator.audi.co.uk/`, August 13 2013).

```
<div class="tileList">
    <div class="tile">
        <div class="tileSel tileBorder" id="DOM_MRADC2V_W" key="MRADC2V"
            style="float:left;" tooltip="html">
            <div class="wheelsSprite" style="width: 79px; height: 47px; background-position: -0px 0px;"/>
        </div>
    </div>
    <div class="tile">
        <div class="tileBorder" id="DOM_MRADC5L_W" key="MRADC5L"
            style="float:left;" tooltip="html">
            <div class="wheelsSprite" style="width: 79px; height: 47px; background-position: -79px 0px;"/>
        </div>
    </div>
</div>
```

(b) The code fragment of the *Wheels* options.

FIGURE 4.7: An example of STALKER fail.

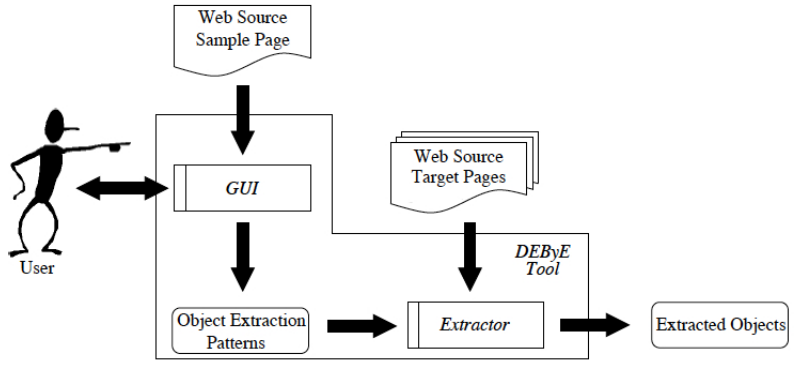


FIGURE 4.8: Modules of the DEByE tool ([LRNdS02]).

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 [PHH05] 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 [CMM01] 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 scrip 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) [CKGS06, CMM01] , and disjunctions appear frequently in the grammar of Web pages [LGMK04]. ROADRUNNER also assumes that any data represented in a page is the extraction target. This poses a serious data accuracy threat for

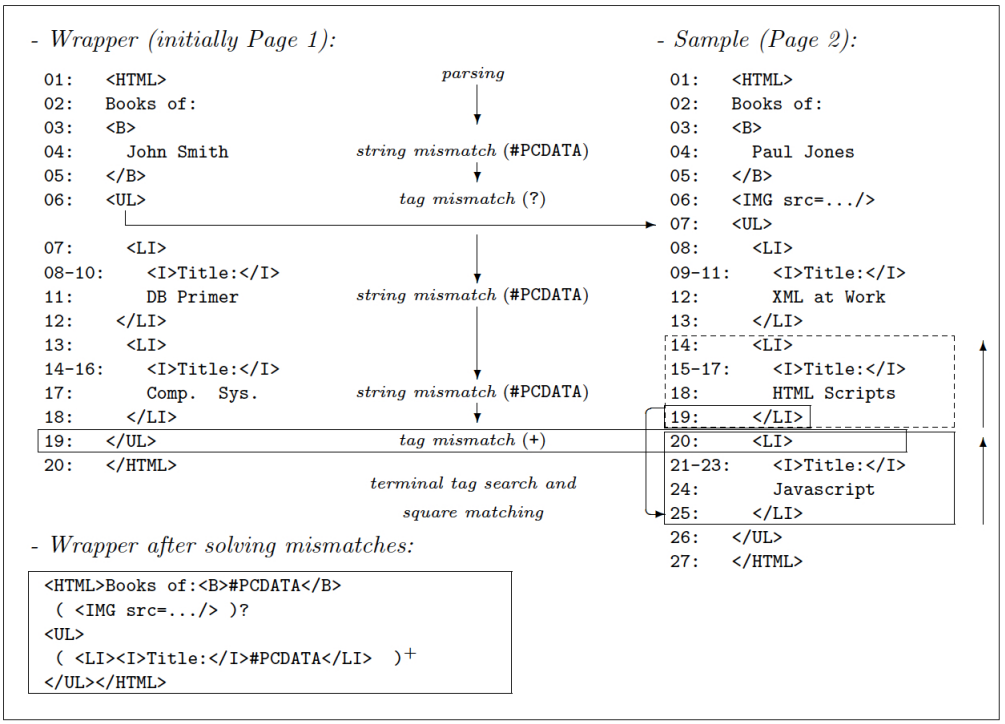


FIGURE 4.9: Two sample pages and the generated wrapper by ROADRUNNER ([CMM01]).

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.

XWRAP. XWRAP (XML-enabled Wrapper) [LPH00] 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 [CKGS06]. 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.

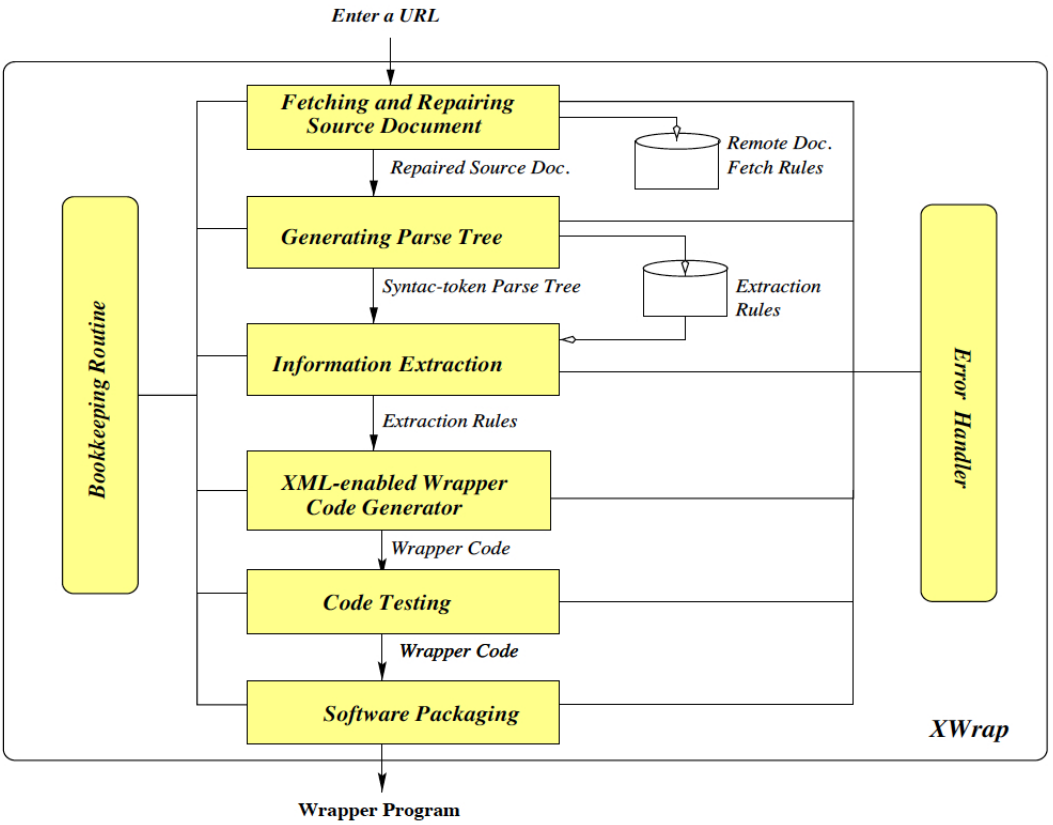


FIGURE 4.10: Data wrapping phases and their interactions in XWRAP ([LPH00]).

DEPTA. DEPTA (Data Extraction based on Partial Tree Alignment) [LGZ03, ZL05] 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 [FMFB12]. 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 [ZL05]. 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 [CKGS06]. 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 [APR+10].

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

Automatic Web News Extraction. Reis *et al.* [RGSL04] 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.

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.

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* [BFG01b, BFG01a]. 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.

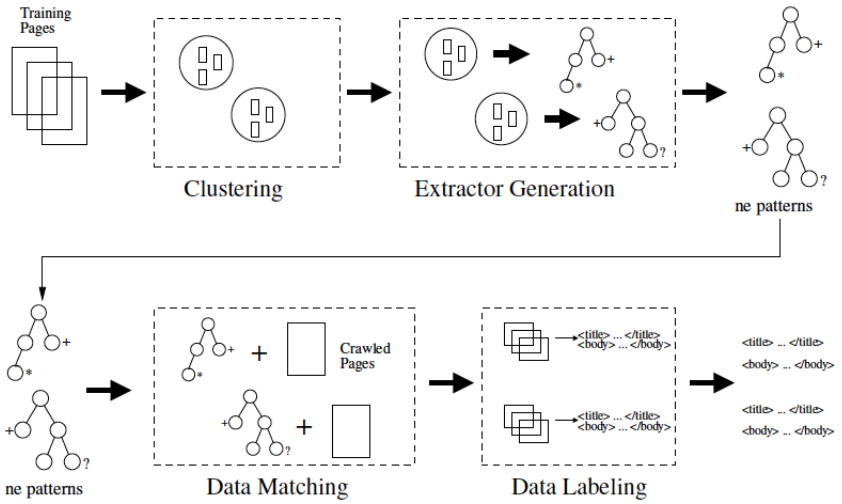


FIGURE 4.11: The main steps of news extraction process ([RGSL04]).

Arasu *et al.* [AGM03] 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 [CKGS06]. 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 [APR+10].

Lerman *et al.* [LGMK04] 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.* [HK05] 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.* [ZSWG09] 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 [APR+10] 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 [AGM03], 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.* [SLB+11] 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.* [ACP+12] 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.

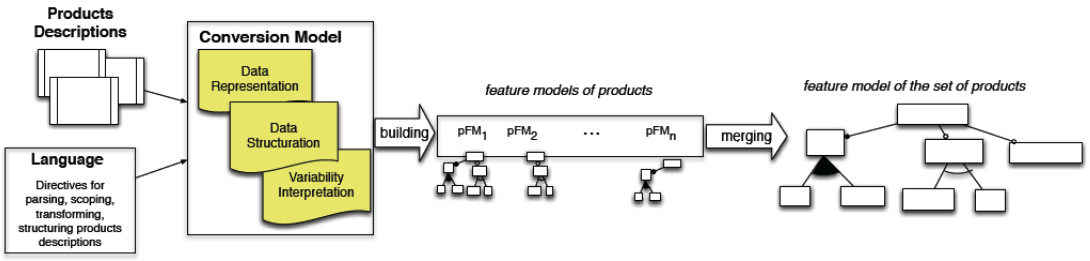


FIGURE 4.12: The process of extracting feature models from product descriptions ([ACP+12]).

Efficient Synthesis of Feature Models. Andersen *et al.* [ACSW12] 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.

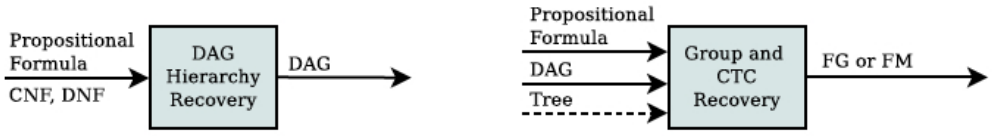


FIGURE 4.13: Components of feature model synthesis ([ACSW12]).

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.

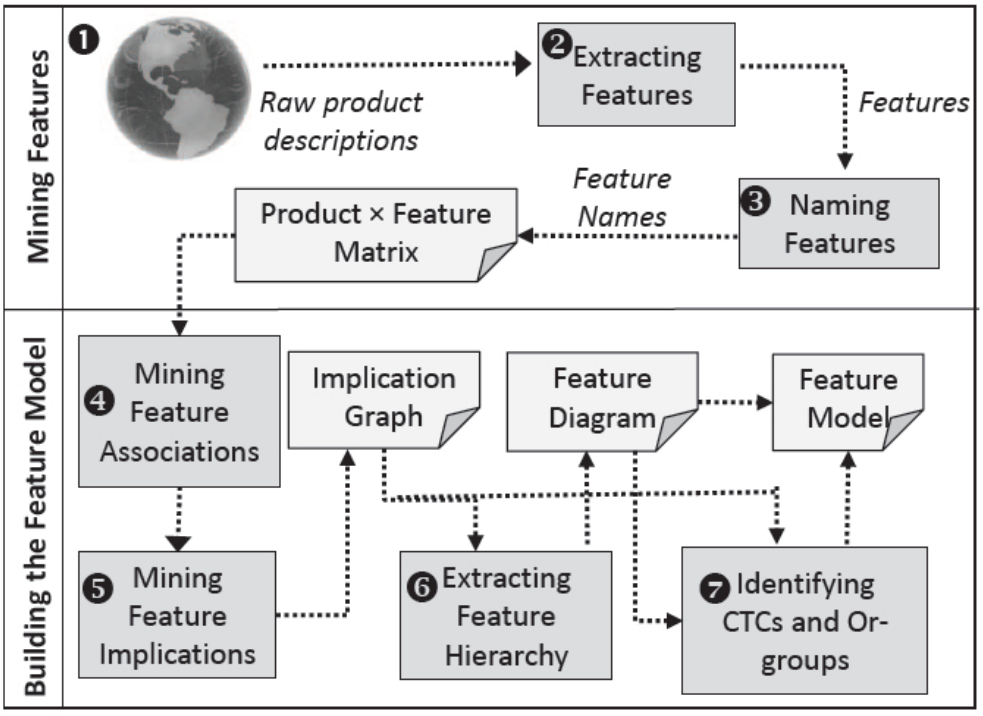


FIGURE 4.14: The two-phase process of mining features and building FM from informal product descriptions ([DDH+13]).

Reverse Engineering Architectural Feature Models. Acher *et al.* [ACC+11] 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.

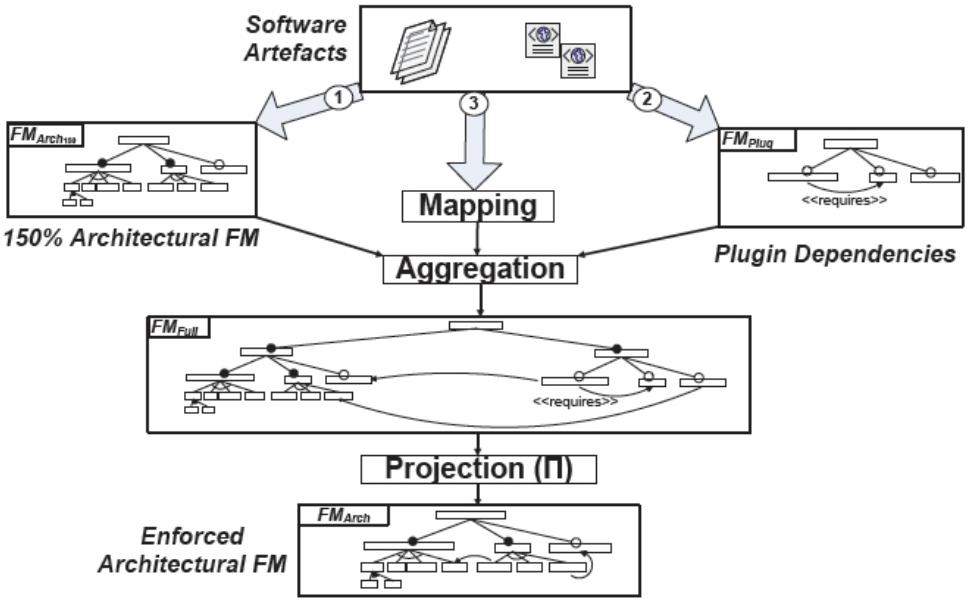


FIGURE 4.15: The process of extracting architectural FMs ([ACC+11]).

Other related works. Lora *et al.* [LMSM10] 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 [Str04] 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 [WCR09] 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 [Joh06] 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.* [ASB+08] 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.* [ZFdSZ12] 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.* [HLHE11] 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) [RGSL04]. Moreover, their scalability has not been adequately evaluated to verify if they can be applied across different application domains (see [FMFB12] 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 [ZSWG09, HK05]). 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 exiting 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 of 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*) [RGSL04] 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 [MMK01] 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 to find a specific HTML or text element.

- Inspired by the *robot* component of CRAWLJAX [MvDL12], 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.* [ZSWG09] 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:

- An HTML-like 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. 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 scalable Web data extraction approach adapted for configurators (RQ2)

The first challenge in designing an efficient Web data extraction approach is its *scalability*. It should scale up 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 [LRNdST02, FMFB12, LRNdS02, HK05, ZSWG09]. 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 [AGM03, HK05, RGSL04] (Section 4.2). However, they are either inaccurate or make many assumptions [ZL05] (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.

```
1   <pattern data-att-met-pattern-type="data" data-att-met-pattern-name="equipment">
2       <input  type="checkbox" data-att-met-clickable= "true"  data-att-met-unique="true"/>
3       <label>
4           <span class="SectionText"> data-tex-mar-option-name
5               <ul data-att-met-multiplicity="[0..1]">
6                   <li data-att-met-multiplicity="[1..*]">data-tex-mar-require-option</li>
7               </ul>
8           </span>
9           <span class="SectionPrice">data-tex-mar-price </span>
10      </label>
11  </pattern>

12  <pattern data-att-met-pattern-type="region" data-att-met-pattern-name="equipmentRegion">
13      <table class="SectionCheckBoxList">
14          <pattern>equipment</pattern>
15      </table>
16  </pattern>
```

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

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 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.

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.

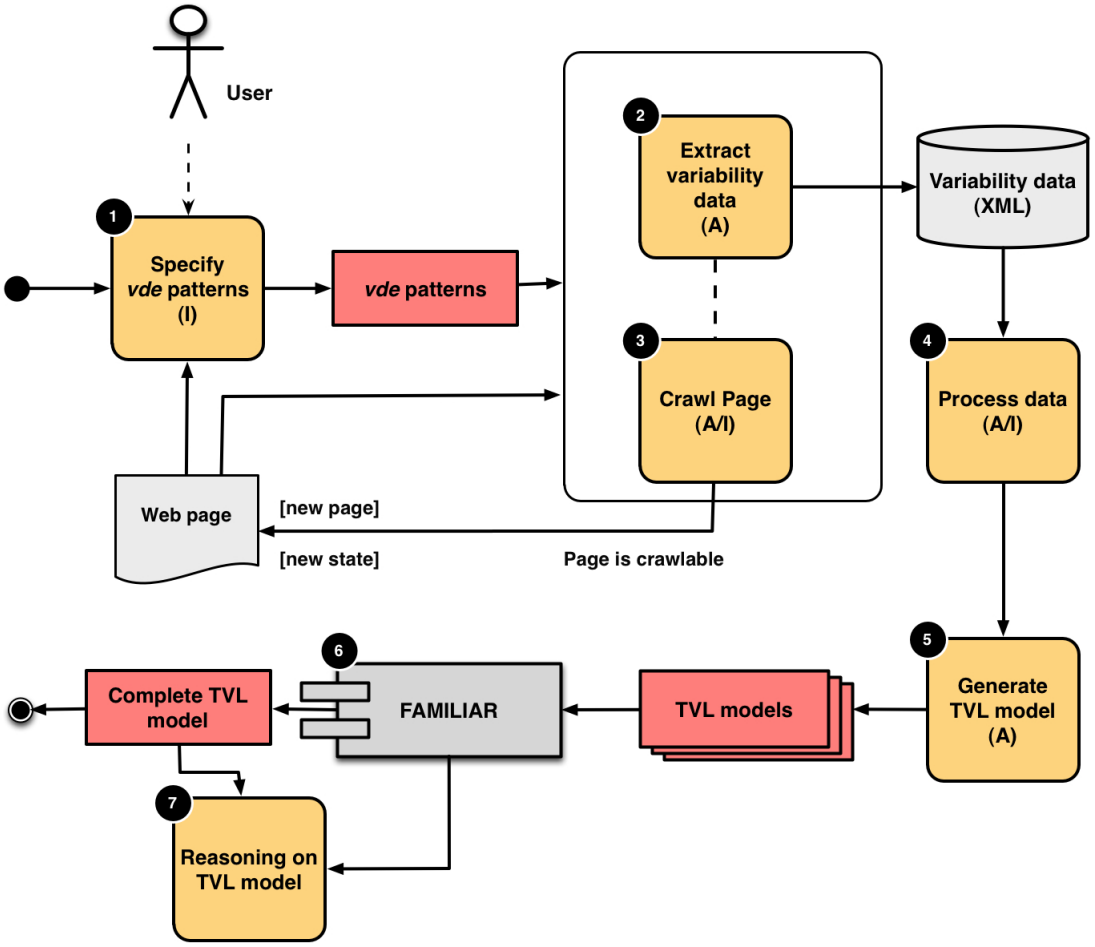


FIGURE 5.2: The Reverse Engineering Process.

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 sytematically 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 [CBH11b]. 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* [ACLF13], 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 [RGSL04]. 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 [LGZ03]). 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 [htt12]. 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: $\rho : A \rightarrow V$.

Definition 6.2: Ordered Tree (from [NJ02]) 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 [NJ02]) 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],

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

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: $DR = \{di_1, di_2, ..., di_n\}$ 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 [RGSL04] and [AGM03]) 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 $\zeta(DI_D, M_D)$.

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 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 the encoding of a set of data objects. A Web page may contain one or more data regions each with variable number of data objects.

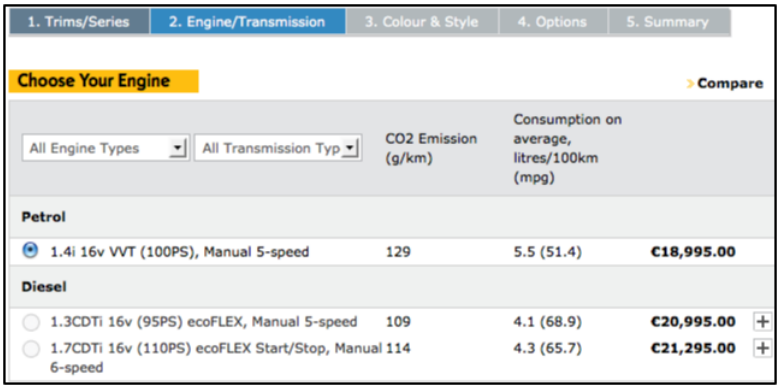
Example 6.1: An example HTML code fragment. Figure 6.1(a) shows an example Web page taken from a car Web configurator. The page presents configuration options which can be selected by users to be included in the final product (a car in this example). Each option is considered as a data object. Figure 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 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.

6.3 Variability Data Extraction Pattern

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.

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



(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).

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*

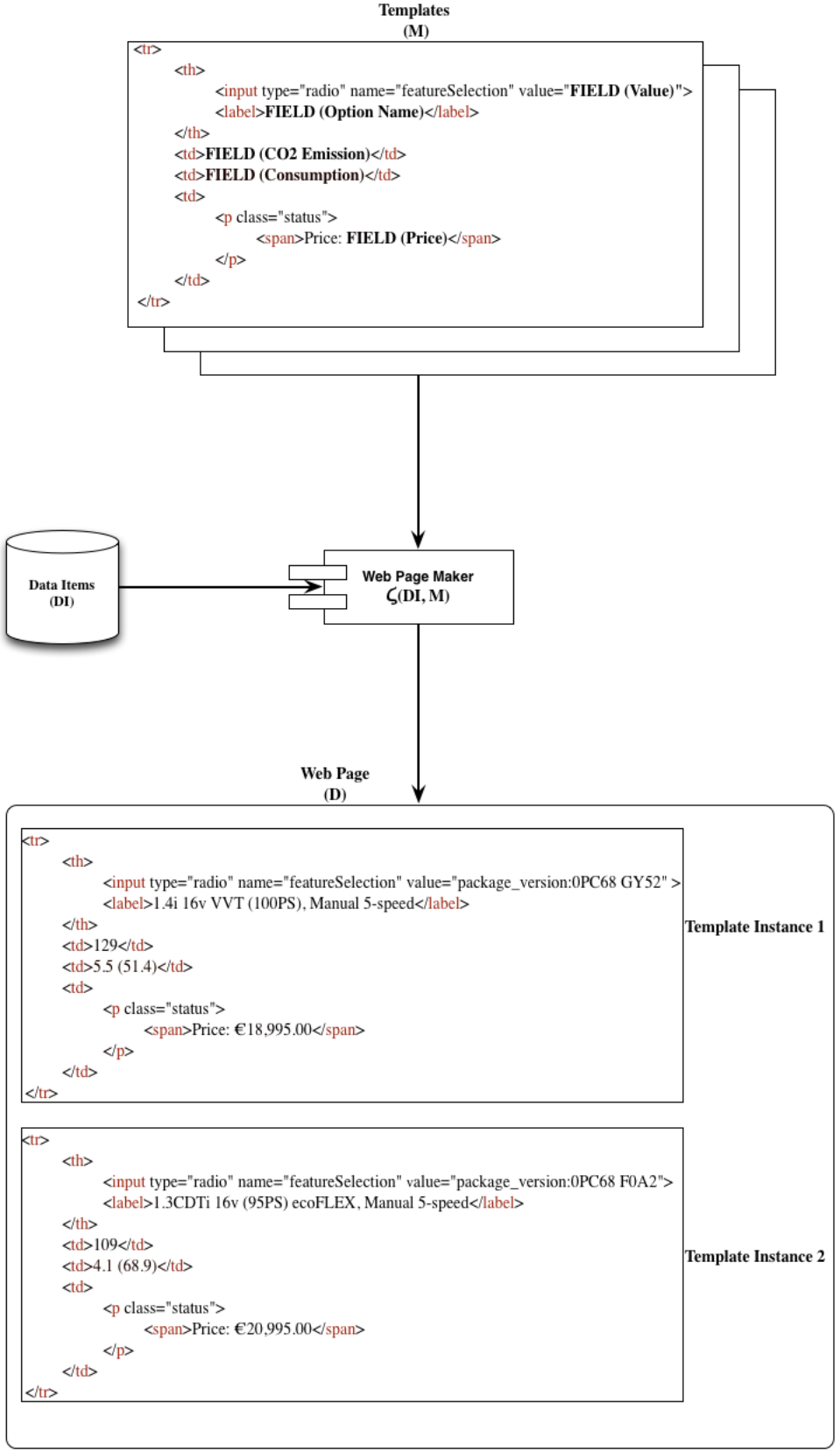


FIGURE 6.2: Web page generation model.

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 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.

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*.

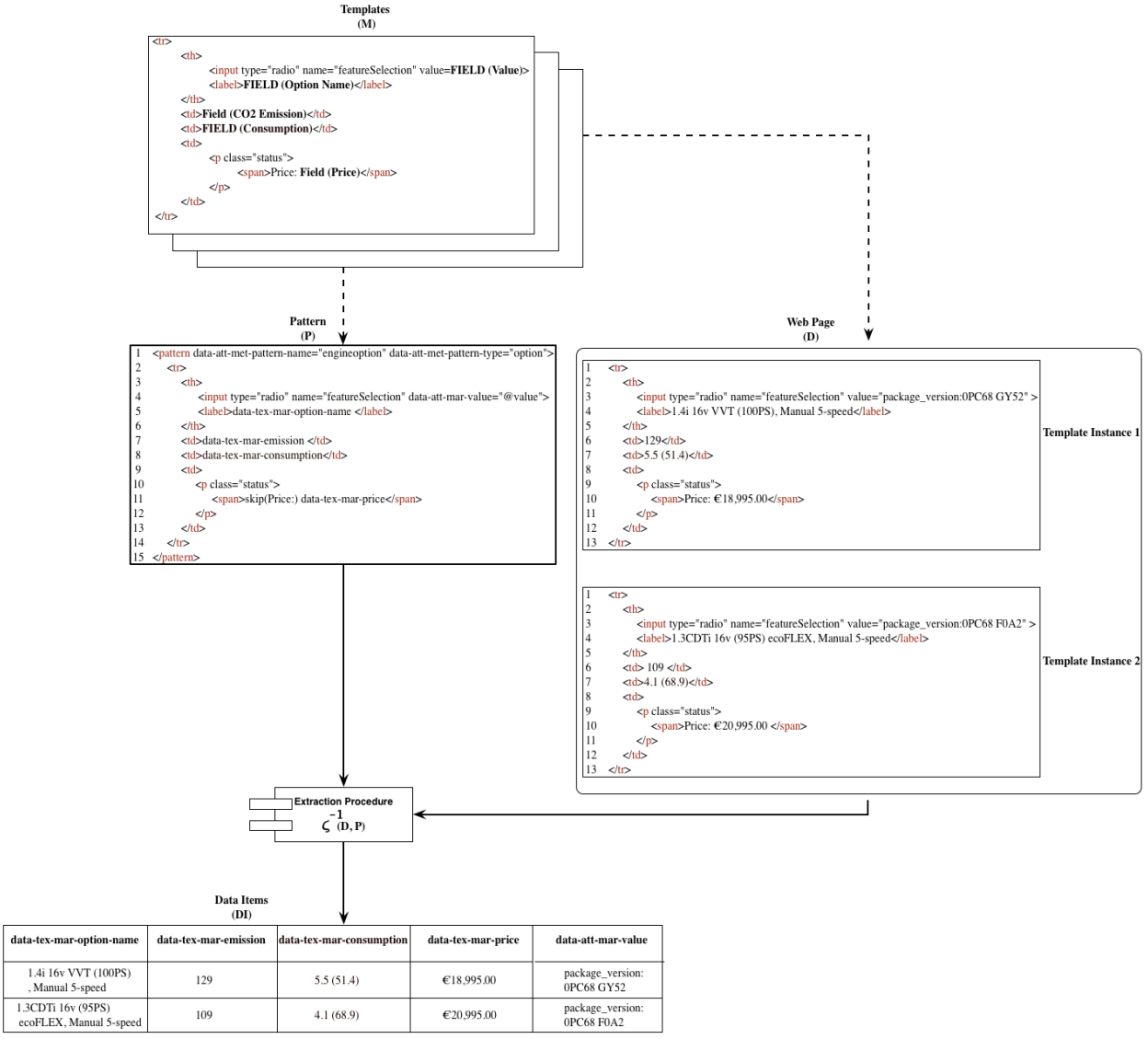


FIGURE 6.3: Data reverse engineering process.

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`.

Pattern Specification

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

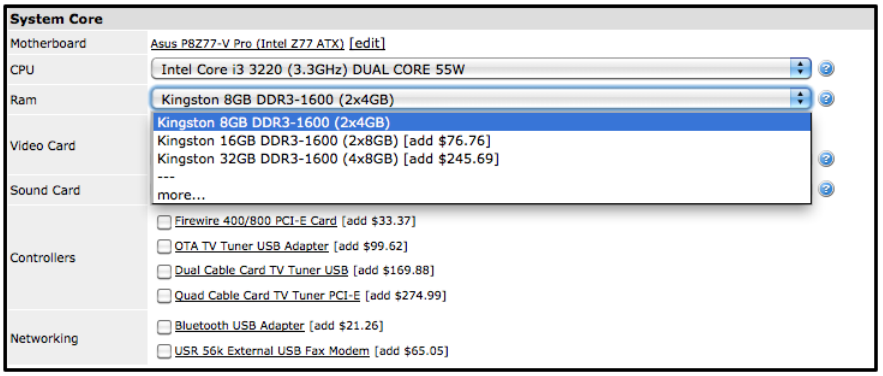
Code Fragment

```
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>
```

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

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 (2x4GB)", "ingston 16GB DDR3-1600 (2x8GB)", and "Kingston 32GB DDR3-1600 (4x8GB)").



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

Pattern Specification

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

Code Fragment

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

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

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

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:

- 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, *viz.* `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).

```
1  <pattern data-att-met-pattern-type = "data" data-att-met-pattern-name="engine">
2      <li>
3          <input  data-att-met-unique="true" type="radio"/>
4          <label> data-tex-mar-option-name</label>
5          <pattern>
6                  singleProperty
7          </pattern>
8          <relation>
9                  or
10         </relation>
11         <pattern>
12                 listProperty
13         </pattern>
14     </li>
15 </pattern>

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

19  <pattern data-att-met-pattern-type = "auxiliary" data-att-met-pattern-name="listProperty">
20     <ul class="property">
21         <li data-att-met-multiplicity="[*]">
22             <label> data-tex-mar-property</label>
23         </li>
24     </ul>
25 </pattern>
```

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

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.

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 [CKGS06] 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.

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*.

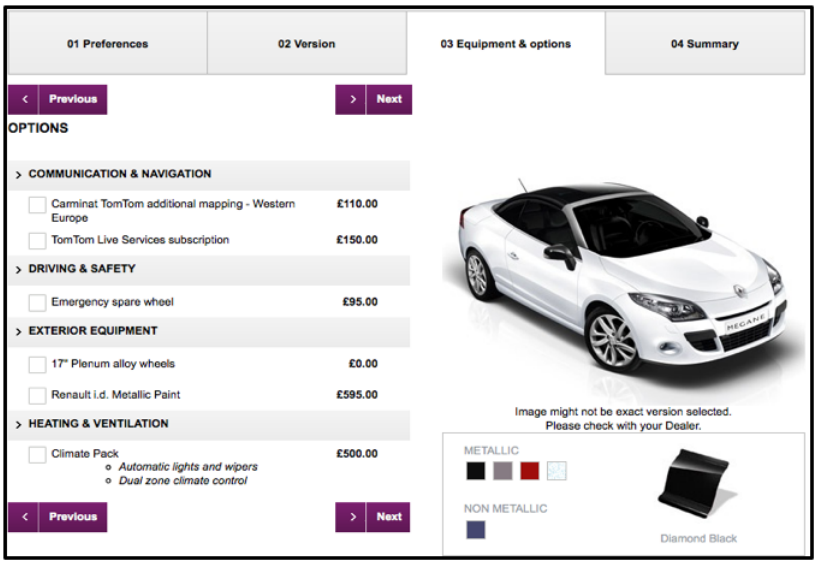
Parent-child. P_1 is a child of P_2 if P_1 is used in the specification of P_2. This presents the hierarchical relationship between patterns.

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



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

Pattern Specification

```
1   <pattern data-att-met-pattern-type="data" data-att-met-pattern-name="options">
2       <input  type="checkbox"/>
3       <label>
4           <span class="SectionText"> data-tex-mar-option-name
5               <pattern data-att-met-multiplicity="[0..1]">  sub-options </pattern>
6           </span>
7           <span class="SectionPrice">data-tex-mar-price </span>
8       </label>
9   </pattern>

10  <pattern data-att-met-pattern-type="auxiliary" data-att-met-pattern-name="sub-options">
11      <ul>
12          <li data-att-met-multiplicity="[1..*]"> data-tex-mar-require-option </li>
13      </ul>
14  </pattern>
```

Code Fragment

```
1   <td>
2       <input  type="checkbox"/>
3       <label>
4           <span class="sectionText"> Renault i.d. Metallic Paint </span>
5           <span class="SectionPrice"> £595.00 </span>
6       </label>
7   </td>

8   <td>
9       <input  type="checkbox"/>
10      <label>
11          <span class="SectionText"> Climate Pack
12              <ul>
13                  <li> Automatic lights and wipers </li>
14                  <li> Dual zone climate control </li>
15              </ul>
16          </span>
17          <span class="SectionPrice"> £500.00 </span>
18      </label>
19  </td>
```

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

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

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).

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. 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

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).

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`.

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).

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).

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.

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).

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:

```
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.

- 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`:

- 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"`.
- 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`.

- 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`.
- 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`.
- 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"`.
- 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.

Chapter 7

Data Extraction Procedure

The user uses *vde* patterns to specify the structure of data objects of interest and to mark data items to be extracted from those objects. The Web Wrapper, given a set of patterns and a Web page, seeks and finds matching data objects in the page and extracts their data items. The extracted data is represented in an XML format in a predefined data model.

In this chapter, we present the data extraction procedure used by the Wrapper to find and extract variability data specified by the user using the *vde* patterns. We specifically explain our proposed algorithm implemented by the Wrapper to find code fragments (representing data objects of interest) in the source code that structurally match the given patterns (Section 7.1). We then present the data model based on which the extracted data is hierarchically organized and represented (Section 7.2). We also briefly demonstrate the tool we developed to support the Wrapper[1] (Section 7.3).

7.1 Data Extraction

7.1.1 Setting up a configuration file

The input for the data extraction process is a target Web page and a configuration file that contains the required *vde* patterns specified to extract data from that Web page. The user inspects the source code of the page, identifies templates from which the

[1]This tool also gives support for the Crawler (see Chapter 8).

page is generated, specifies *data* and *auxiliary* patterns defining the structure of those templates, and marks the required data in those patterns. She also identifies the target data region in the page and specifies a *region* pattern denoting that region. Once the configuration file is created, the Wrapper starts the data extraction process.

7.1.2 Pattern matching algorithm

Given a configuration file and a Web page, the Wrapper operates within the block of the source code identified by the region pattern and looks for code fragments that structurally match the data pattern (and auxiliary patterns used in the specification of the data pattern). The Wrapper uses a *data extraction procedure* to find matching code fragments. Our proposed algorithm for the data extraction procedure provides a two-step solution to find matching code fragments: (1) first *finding candidate code fragments* that may match the given data pattern and then (2) *traversing each candidate code fragment to find out if it is exactly matching the pattern.* The algorithm seeks to find mappings between elements of a code fragment and the given patterns using their tree representations. During the mapping, if a conflict is detected the target code fragment is excluded from the data extraction process. During the traversal of a code fragment, its data items are also extracted.

7.1.2.1 Finding candidate code fragments

The data extraction procedure parses the source code of a Web page to extract data. It ignores as much as possible the source lines of code that do not have any data of interest, and considers all those may have. The region pattern delimits the extraction procedure to operate within a specific portion of the source code. Consequently, many irrelevant lines of code are automatically discarded by the extraction procedure.

We offer to divide the source code within the given region (identified by the region pattern) into a number of candidate code fragments, each of which may match the given data pattern. Clearly, those lines of code that are not covered by the candidate code fragments are ignored by the data extraction procedure as well.

We now present the algorithm for finding candidate code fragments. The algorithm works on the HTML tag tree of the given patterns (the content of the configuration file)

and DOM of the target Web page. The proposed algorithm is based on the following three key assumptions:

- A code fragment that matches the given data pattern has a unique element, meaning that there is one and only one instance of that element in the code fragment.
- All elements in the path from the unique element up to the root element of the data pattern (excluding the root `pattern` element in the data pattern) are predefined HTML elements. This path defines the *signature* of the unique element. The *length* of a signature is the number of its included elements.
- The elements included in the signature of the unique element in the pattern specification must not have the `skip(all)` element neither as an immediate, nor as an indirect sibling element.

Our empirical observation (Chapter 3 and [AHA+13]) shows that the first expectation is true. For example, each option is represented using a widget and the element implementing the widget is a unique element in the code fragment of the option. Note that the element tag name together with its attributes may make an element unique. The user uses the `data-att-met-unique = "true"` attribute to mark the unique element in the data pattern specification. This underlying assumption acts as the first parameter to find candidate code fragments.

The second assumption ensures that the unique element must be included and marked in the data pattern. If it is contained in an auxiliary pattern, so the `pattern` element of the auxiliary pattern will be part of the signature of the unique element, this will violate the assumption.

`skip(all)` does not preserve the parent-child relationships between elements in a pattern, and therefore makes it impossible to accurately compute the signature of the unique element. For this reason, the third assumption is made.

Candidate Code Fragment. A candidate code fragment of a given data pattern is a code snippet in the source code of the given page such that:

- there is a one-to-one mapping between its first-level HTML elements and those of the data pattern,

- it contains an element that is identical to the unique element marked in the data pattern specification, and
- the identical element in the code fragment has the same signature (with the calculated length) as the unique element in the data pattern.

The algorithm uses a *bottom-up* tree traversing to find candidate code fragments. It first finds all HTML elements in the source code that are identical to the unique element and have the same signature as the unique element. For each found identical element, the algorithm then walks l steps up, in which l is the length of the signature. At the end of the bottom-up traversing, the algorithm stops on an HTML element, called the *index* element – in the source code it corresponds to a first-level HTML element in the data pattern. The algorithm then propagates the mappings between the siblings of the index element in the source code and the data pattern. When the algorithm draws an analogy between the first-level elements of the data pattern and a code fragment in the source code, it records that code fragment as a candidate code fragment. Each candidate code fragment is identified with its first-level elements.

The algorithm for detecting and recording the candidate code fragments is given in Figure 7.1. **getCandidateCodeFragments** is a function that takes as input the HTML tag tree of a configuration file (*configFile*) and the DOM of the source code of the given page (*pageSource*). We use the source code shown in Figure 7.2 and the configuration file presented in Figure 7.3 as inputs to describe the algorithm. To make it easy for the reader to find the code lines from the text, we optionally use *al:*, *sc:*, and *cf:*before the line number(s) to respectively refer to Figure 7.1, Figure 7.2, and Figure 7.3.

The algorithm first finds the unique element of the data pattern (line *al:*2). If more than one element is marked as unique, the algorithm returns the first matching element. In the given configuration file, the `input` element (line *cf:*3) is returned as *uniqueElement*. Then, the algorithm finds the element identifying the data region in the source code (line *al:*3). Since the <`table class="tagselect"`> element is defined to be the root element of the data region (line *cf:*20) in the configuration file, the algorithm looks for an identical element in the source code. It finds the element in line *sc:*1 and returns the element as *regionElement*.

```
1 getCandidateCodeFragments (configFile, pageSource) {
2   let uniqueElement be the unique element in the data pattern
3   let regionElement be the element highlighting the region in the source code
4   i = 0
5   uniqueElementSignature = getSignature (configFile, uniqueElement)
6   identicalElements[] = getIdenticalElements (pageSource, uniqueElement, regionElement)
7   for each (element in identicalElements) {
8           elementSignature = getSignature (pageSource, element)
9           if (areIdenticalSignatures (uniqueElementSignature, elementSignature )) {
10              candidateCodeFragment[i].mappingIndexElement = getRoot (uniqueElementSignature)
11              candidateCodeFragment[i].mappedIndexElement = getRoot (elementSignature)
12              i++
13          }
14  }
15  for each (codeFragment in candidateCodeFragment) {
16      mappingElement = codeFragment.mappingIndexElement
17      mappedElement = codeFragment.mappedIndexElement
18      i = 0
19      while (mappingElement = getPreviousSibling (configFile, mappingElement)) {
20          if (mappingElement instanceOf skipSiblingElement) {
21              n = getMultiplicityValue (mappingElement)
22              mappedElement = getPreviousSibling (codeFragment, mappedElement, n)
23              mappingElement = getPreviousSibling (configFile, mappingElement)
24          }else{
25              mappedElement = getPreviousSibling (codeFragment, mappedElement)
26          }
27          if (areIdenticalElements (mappingElement, mappedElement)) {
28              codeFragment.previousMappingElement[i] = mappingElement
29              codeFragment.previousMappedElement[i] = mappedElement
30              i++
31          }else{
32              codeFragment. mappingIndexElement = null
33              break
34          }
35      }
36  }
37  for each (codeFragment in candidateCodeFragment) {
38      if (codeFragment. mappingIndexElement == null) {
39          break
40      }
41      mappingElement = codeFragment.mappingIndexElement
42      mappedElement = codeFragment.mappedIndexElement
43      i = 0
44      while (mappingElement = getNextSibling (configFile, mappingElement)) {
45          if (mappingElement instanceOf skipSiblingElement) {
46              n = getMultiplicityValue (mappingElement)
47              mappedElement = getNextSibling (codeFragment, mappedElement, n)
48              mappingElement = getNextSibling (configFile , mappingElement)
49          }else{
50              mappedElement = getNextSibling (codeFragment, mappedElement)
51          }
52          if (areIdenticalElements (mappingElement, mappedElement)) {
53              codeFragment.nextMappingElement[i] = mappingElement
54              codeFragment.nextMappedElement[i] = mappedElement
55              i++
56          }else{
57              codeFragment. mappingIndexElement = null
58              break
59          }
60      }
61  }
62  return candidateCodeFragment
63 }
```

FIGURE 7.1: The algorithm for finding candidate code fragments.

```
1  <table class="tagSelect">
2    <tr>
3      <td>
4        <input type="radio"  id="embossed" checked="checked" value="em" name="t1type"/>
5      </td>
6      <td>
7        <label for="embossed">
8          <img title="Embossed Characters " src="images/site/emboss_sample.png">
9        </label>
10     </td>
11     <td>
12       <h6>
13         <label for = "embossed"> Embossed Characters</label>
14       </h6>
15       <p>
16         <span> A…Z 0…9 °"'*.#?$!:-+=()&,\/ @ </span>
17           © ★ † ♥
18         <span> ♣ </span>
19       </p>
20     </td>
21     <td></td>
22   </tr>
23   <tr>
24     <td>
25       <input type="radio"  id="dt-std-br" checked="checked" value="br" name="t1style"/>
26     </td>
27     <td>
28       <label for="dt-std-br">
29         <img title="Brass Dogtog " src="images/site/dt-std-br_sm.png">
30       </label>
31     </td>
32     <td>
33       <h6>
34         <label for = "dt-std-br"> Brass Dogtog</label>
35       </h6>
36       <span>$8.99</span>
37     </td>
38     <td></td>
39   </tr>
40   <tr>
41     <td>
42       <div>
43         <input type="radio"  id="dt-std-black" checked="checked" value="black" name="t1style"/>
44       </div>
45     </td>
46     <td>
47       <label for="dt-std-black">
48         <img title="Black Dogtog " src="images/site/dt-std-black_sm.png">
49       </label>
50     </td>
51     <td>
52       <h6>
53         <label for = "dt-std-black"> Black Dogtog</label>
54       </h6>
55       <p>
56         <span> Cannot be debossed. </span>
57       </p>
58     </td>
59     <td></td>
60   </tr>
61 </table>
```

FIGURE 7.2: An example source code (`http://www.mydogtag.com/`, May 14 2013).

```
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 7.3: Pattern configuration file.

getSignature (line *al:*5) computes the signature of *uniqueElement*, i.e., `td` and assigns it to *uniqueElementSignature*. Next, the **getIdenticalElements** function (line *al:*6) finds all elements within *regionElement* that are identical to *uniqueElement*. Consequently, the `input` elements in lines *sc:*4, 25, and 43 of the source code are identified by the function and stored in the *identicalElements* array.

Lines *al:*7-14 iterate for each identified identical element and exclude those elements whose signature does not conform to the signature of *uniqueElement*. In this step, the `input` element in line *sc:*43 is excluded, because its signature is `div` that is distinct from that of *uniqueElement*. Note that since the length of the signature of *uniqueElement* is *1*, then, for each identical element a signature with the length of *1* is computed. For each identical element that has the same signature as *uniqueElement*, an instance of *candidateCodeFragment* is created. So, two instances of this type are created for the example source code given in Figure 7.2. This object has two data members. *mappingIndexElement* that holds the root element of the signature of *uniqueElement* (line *al:*10),

i.e., the `td` element (line *cf:*2) in the configuration file. *mappedIndexElement* stores the root element of the signature of the identical element (line *al:*11). Consequently, when the algorithm reaches line *al:*15, the objects have the following states:

candidateCodeFragment[0].mappingIndexElement = td

candidateCodeFragment[0].mappedIndexElement = td

candidateCodeFragment[1].mappingIndexElement = td

candidateCodeFragment[1].mappedIndexElement = td

It means that the `td` elements in lines *sc:*3 and 24 of the source code are mapped onto the `td` element of the data pattern (line *cf:*2). In fact, *mappingIndexElement* is the first-level element of the data pattern containing *uniqueElement*, and *mappedIndexElement* is an element in a candidate code fragment that maps to *mappingIndexElement*.

Lines *al:*15-36 attempt to find mappings between the previous (left) sibling elements of *mappingIndexElement* in the data pattern and *mappedIndexElement* in a code fragment. It first takes *mappingElement* as the previous immediate sibling element of *mappingIndexElement* (line *al:*19). If this element is an instance of *skipSiblingElement* (line *al:*20), i.e., `skip(sibling, Multiplicity)`, the algorithm skips the n previous sibling elements (in which n is the multiplicity value, line *al:*21) of *mappedIndexElement* in the code fragment and returns the $n+1$th previous sibling element as *mappedElement* (line *al:*22). Now that *skipSiblingElement* is resolved, its previous element is assigned to *mappingElement* (line *al:*23). By definition, previous (left) and next (right) immediate sibling elements of *skipSiblingElement* is an HTML element, thus *mappingElement* is an HTML element. If *mappingElement* is not an instance of *skipSiblingElement*, the algorithm returns the previous immediate sibling element of *mappedIndexElement* in the code fragment as *mappedElement* (line *al:*25). If *mappingElement* and *mappedElement* are identical elements (line *al:*27), then they are captured in the *codeFragment* object. *codeFragment* has two *previousMappingElement* and *previousMappedElement* arrays. *previousMappingElement[i]* holds *mappingElement* of the data pattern and *previousMappedElement[i]* its corresponding mapped element, i.e., *mappingElement*, of the code fragment (lines *al:*28 and 29). During the mapping, if a conflict is detected (both mapping and mapped elements are not identical), *mappingIndexElement* is set to *null*

(line *al:*32), meaning that the code fragment no longer matches the data pattern and must not be further processed (e.g., see lines *al:*38-40). The aforementioned steps are iterated for all previous sibling elements of *mappingIndexElement* (line *al:*19). In our example in Figure 7.3 since there is no previous sibling element for *mappingIndexElement* (line *cf:*2), lines *al:*15-36 are not executed.

Lines *al:*37-61 perform the same steps as lines *al:*15-36, but for the *next* (right) sibling elements of *mappingIndexElement* in the data pattern and *mappedIndexElement* in the code fragment. Consider the example source code in Figure 7.2 and the configuration file in Figure 7.3. The `td` element (*mappedIndexElement*) in the code fragment (line *sc:*3) is mapped to the `td` element (*mappingIndexElement*) in the data pattern (line *cf:*2). The next sibling element of *mappingIndexElement* is `skip(sibling,[1])` (line *cf:*5), thus one next immediate sibling element of *mappedIndexElement* must be discarded, i.e., the `td` element in line *sc:*6. Consequently, when the algorithm reaches line *al:*52, *mappingElement* is the `td` element (in line *cf:*6) and *mappedElement* is the `td` element (in line *sc:*11). Since *mappingElement* and *mappedElement* are identical, the mapping is captured in the *codeFragment* object (lines *al:*53 and 54). Note that in the next iteration of lines *al:*37-61, the `td` element in line *sc:*32 of the source code is mapped onto the `td` element of the data pattern in line *cf:*6.

When the **getCandidateCodeFragments** function terminates for this example, it returns two candidate code fragments (an array of *candidateCodeFragment* objects) from the source code that match the data pattern: one code fragment starts at line *sc:*3 and ends at line *sc:*20 (*candidateCodeFragment[0]*), and another starts at line *sc:*24 and ends at line *sc:*37 (*candidateCodeFragment[1]*). A candidate code fragment is identified by its first-level elements, each of which maps into one first-level HTML element of the data pattern.

Figure 7.4 depicts the tree representations of the data pattern (lines *cf:*1-12) at the top, and the code fragment (lines *sc:*3-20) at the bottom. For each element, the corresponding line number is enclosed in parentheses. The dashed line shows elements mapped together. Note that the `td` element in line *sc:*6 is mapped into the `skip(sibling,[1])` element (but not into an HTML element) in the pattern and therefore it and its children (marked with ×) are discarded and not included in the recorded candidate code fragment.

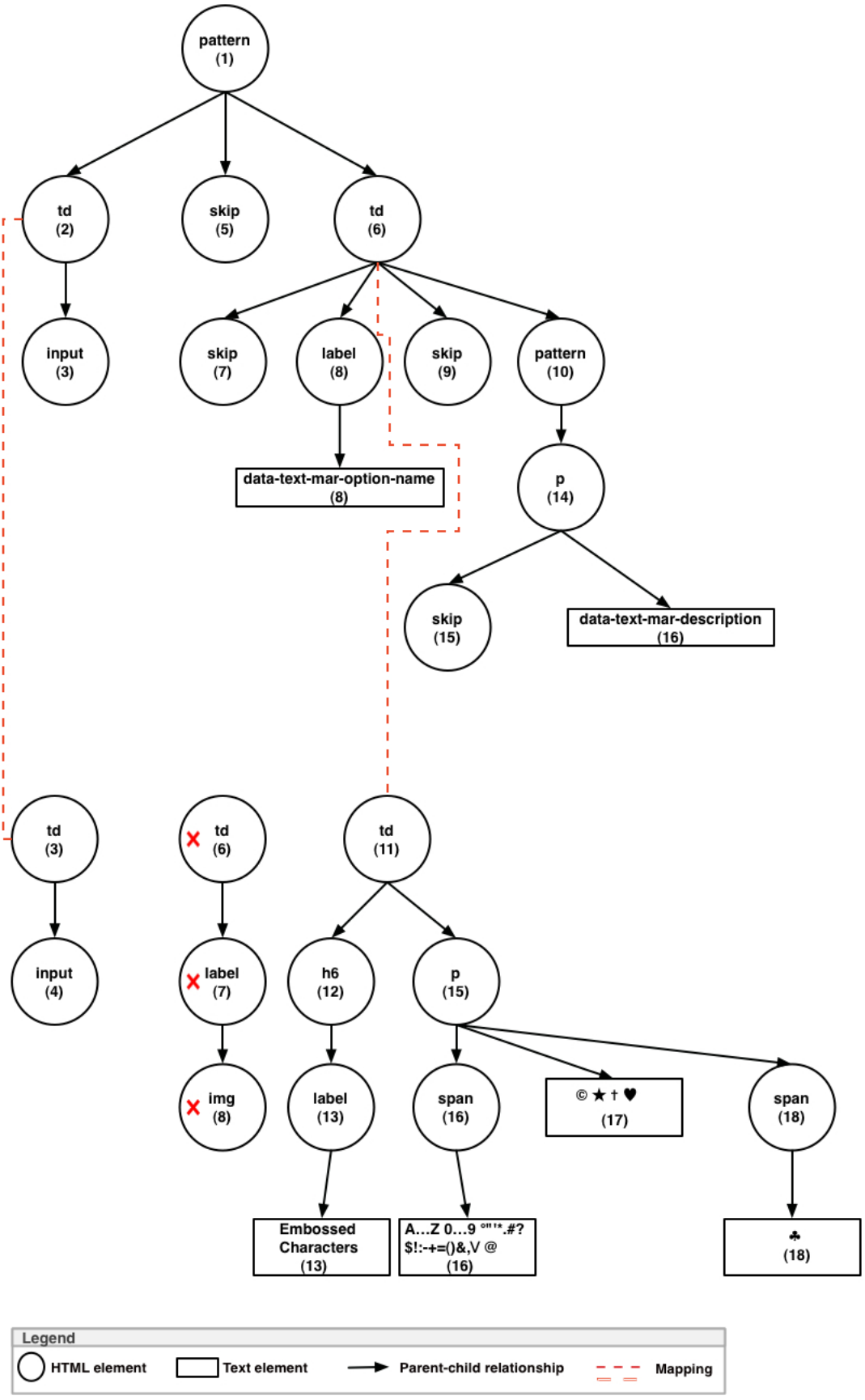


FIGURE 7.4: Tree representation (1).

7.1.2.2 Traversing the candidate code fragments

Figure 7.5 presents **dataExtractionProcedure**, the main entry procedure of our data extraction system. It first checks the configuration file (*configFile*) for syntax and some semantic errors (line 2). **validConfigFile** parses the configuration file to an HTML tag tree and ensures that pattern-specific attributes and elements are properly defined, and match predefined naming conventions. Moreover, it validates that patterns are well-formed and all the referenced patterns exist in the configuration file.

```
1 dataExtractionProcedure (configFile, pageSource) {
2   if (!validConfigFile(configFile))
3       return
4   candidateCodeFragments[] = getCandidateCodeFragments (configFile, pageSource)
5   for each ((codeFragment in candidateCodeFragments) and (codeFragment.mappingIndexElement != null)) {
6       outputData = new outputData()
7       mappingIndexElement = codeFragment.mappingIndexElement
8       for (i=0; i< codeFragment.previousMappingElement.length ) {
9           patternElement = codeFragment.previousMappingElement[i]
10          mappedElement =  codeFragment.previousMappedElement[i]
11          traverseTree (mappingIndexElement, patternElement, mappedElement, configFile, pageSource, outputData)
12          if (mappingIndexElement == null)
13              goToLine (8)
14      }
15      patternElement = codeFragment.codeFragment.mappingIndexElement
16      mappedElement = codeFragment.codeFragment.mappedIndexElement
17      traverseTree (mappingIndexElement , patternElement, mappedElement, configFile, pageSource, outputData)
18      if (mappingIndexElement == null)
19          goToLine (8)
20      for (i=0; i< codeFragment.nextMappingElement.length ) {
21          patternElement = codeFragment.nextMappingElement[i]
22          mappedElement =  codeFragment.nextMappedElement[i]
23          traverseTree (mappingIndexElement, patternElement, mappedElement, configFile, pageSource, outputData)
24          if (mappingIndexElement == null)
25              goToLine (8)
26      }
27      if (validData (outputData))
28          addToExtractedData (outputData)
29  }
30 }
```

FIGURE 7.5: Data extraction procedure.

In line 4, the procedure calls **getCandidateCodeFragments** which returns back the candidate code fragments. Lines 5-29 iterate for each candidate code fragment and extract their data according to the data pattern (and the auxiliary patterns if they are used in the specification of the data pattern). Data extracted from a code fragment is stored in the *outputData* object (line 6). We later describe in this chapter the schema (data model) of *outputData* (see Section 7.2). Note that **getCandidateCodeFragments** only finds elements from a code fragment that map to the first-level children of

the data pattern (see Figure 7.4). To find all other mappings and extract data, we use the **traverseTree** procedure (Figure 7.6). This procedure uses the tree representations of both the data pattern and the code fragment, traverses them in parallel and tries to find mappings between their elements. It uses a mixture of both *depth-first* and *breadth-first* traversals to trace the tag trees. **traverseTree** is called for every two mapped elements (one from the data pattern and one form the code fragment). Lines 8-14 call the procedure for the mapped elements preceding the index elements, line 17 calls it for index elements, i.e., *mappingIndexElement* and *mappedIndexElement*, and finally, lines 20-26 call the procedure for the mapped elements following the index element. Note that everywhere during traversal of the code fragment, if a conflict is detected, meaning that the code fragment no longer matches the data pattern, the code fragment should not be further processed. To implement this, *mappingIndexElement* is passed to **traverseTree** and within the procedure in the case of a conflict, this parameter is set to *null*. When the procedure terminates, *mappingIndexElement* is checked (lines $12-13$, $18-19$, and $24-25$), and if it is *null*, the current code fragment is discarded and the next code fragment is considered. When the code fragment is successfully traversed, *outputData* is validated (line 27), ensuring that a valid data is extracted. **validData** checks, for instance, that all the mandatory data items marked in the patterns are extracted. If the data is valid, it is reported out (line 28).

Tree traversing

TABLE 7.1: Element instances.

Element instance	**Tag/element**
htmlElement	any valid HTML tag
patternElement	<pattern>pattern-name</pattern>
relationElement	<relation>or</relation>
skipAllElement	skip(all)
dataMarkingTextElement	a data marking text element
skipSiblingElement	skip(sibling, Multiplicity)
skipTextElement	skip(A_STRING_VALUE)

traverseTree is a recursively defined procedure that takes as input *mappingIndexElement* of the target code fragment, two mapping elements *patternElement* from the data

pattern and *targetElement* from the target code fragment that is mapped to *patternElement*, the tree representations of the configuration file (*configFile*) and the target code fragment (*pageSource*), as well as the *ouputData* object which gathers the extracted data as the code fragment is progressively processed.

Line 2 checks that the target code fragment is still valid, otherwise returns back. Line 3 ensures that *patternElement* is not *null*. If *patternElement* is not *null*, but *targetElement* is (line 4), it means that there is no mapping and a mismatch between the code fragment and the data pattern is discovered. In that case, the process must not be continued. Line 5 validates that *patternElement* and *targetElement* are identical, otherwise there is a mismatch and the process is aborted. **extractAttributeData** traces *patternElement* and if it contains data marking attributes, extracts their corresponding data items from *targetElement* and adds them to *outputData*.

In each execution of **treeTraversing**, the procedure traverses the first-level children of *patternElement*, and accordingly *targetElement*, meaning that the procedure uses a breadth-first order for traversing (lines 11-90). Everywhere during this traversing, if two HTML elements (one from the data pattern and one from the target code fragment) are structurally mapped, a new call of **treeTraversing** is made for these two elements. This recursive calling of the **treeTraversing** procedure constitutes a depth-first traversal of the data pattern and the target code fragment. To start, the first immediate child of the data pattern (*currentPatternElement*, line 7) and the code fragment (*currentTargetElement*, line 8) are taken. If *currentPatternElement* is null, it means that *patternElement* is a leaf node and there is no more node to be processed (line 9). If *patternElement* exists, however, there is no mapped element from the code fragment, a mismatch is detected and the process is terminated (line 10).

Lines 11-90 iterate until all first-level children of *patternElement* are visited or a conflict is detected. We explain the algorithm and present how it operates when it visits different types of elements (see Table 7.1) in the data pattern. Note that we mostly present the true conditions and so blocks of code to be executed when a condition is not true (which may lead to terminate the process) are usually omitted.

***currentPatternElement* is an instance of *htmlElement*.** Lines 12-40 show the block of code to be executed if *currentPatternElement* is a predefined HTML element (not a pattern-specific element). If the element is assigned a multiplicity attribute (line

```
traverseTree (mappingIndexElement, patternElement, targetElement, configFile, pageSource, outputData) {
    if (mappingIndexElement == null) return
    if (patternElement == null) return
    if (targetElement == null) {  mappingIndexElement = null return}
    if (! areIdenticalElements (patternElement, targetElement)) { mappingIndexElement = null return }
    extractAttributeData(patternElement, targetElement, configFile, pageSource, outputData)
    currentPatternElement = getFirstChild (configFile, patternElement)
    currentTargetElement = getFirstChild (pageSource, targetElement)
    if (currentPatternElement == null) return
    if (currentTargetElement == null) { mappingIndexElement = null return }
    do {
        if (currentPatternElement instanceOf htmlElement) {
            if (hasMultiplicityAttribute (currentPatternElement )) {
                lowerValue = getMultiplicityLowerValue (currentPatternElement)
                upperValue = getMultiplicityUpperValue (currentPatternElement)
                if (lowerValue == upperValue == '*') {
                    while ((currentTargetElement != null) and (areIdenticalElements (currentPatternElement, currentTargetElement ))) {
                        traverseTree (mappingIndexElement, currentPatternElement, currentTargetElement, configFile, pageSource, outputData)
                        currentTargetElement = getNextSibling (pageSource, currentTargetElement)
                    }
                    currentPatternElement = getNextSibling (configFile, currentPatternElement)
                }else if ((lowerValue == upperValue) and (lowerValue instanceOf integerNumber )) {
                    while ((currentTargetElement != null) and (areIdenticalElements (currentPatternElement, currentTargetElement )) and (lowerValue>0)) {
                        traverseTree (mappingIndexElement, currentPatternElement, currentTargetElement, configFile,  pageSource, outputData)
                        currentTargetElement = getNextSibling (pageSource, currentTargetElement)
                        lowerValue--
                    }
                    currentPatternElement = getNextSibling (configFile, currentPatternElement)
                } else if ((lowerValue == 0) and (upperValue == 1)) {
                    if (areIdenticalElements (currentPatternElement,  currentTargetElement )) {
                        traverseTree (mappingIndexElement, currentPatternElement, currentTargetElement, configFile,  pageSource, outputData)
                        currentPatternElement = getNextSibling (configFile, currentPatternElement)
                        currentTargetElement = getNextSibling (pageSource, currentTargetElement)
                    }else currentPatternElement = getNextSibling (configFile, currentPatternElement)
                }
            }else {
                traverseTree (mappingIndexElement, currentPatternElement, currentTargetElement, configFile, pageSource, outputData)
                currentPatternElement = getNextSibling (configFile, currentPatternElement)
                currentTargetElement = getNextSibling (pageSource, currentTargetElement)
            }
        }else if (currentPatternElement instanceOf patternElement) {
            if (getNextSibling (currentPatternElement) instanceOf relationElement) {
                nextPatternElement = getNextSibling (configFile, getNextSibling (configFile ,currentPatternElement))
                currentTargetElement = handleOrRelation (currentPatternElement, nextPatternElement, currentTargetElement, configFile, pageSource, outputData)
                currentPatternElement = getNextSibling (configFile, nextPatternElement)
            }else {
                currentTargetElement = handlePattern (currentPatternElement, currentTargetElement, configFile, pageSource, outputData)
                currentPatternElement = getNextSibling (configFile, currentPatternElement)
            }
        }else if (currentPatternElement instanceOf skipAllElement) {
            nextPatternElement =  getNextSibling (configFile, currentPatternElement)
            if  (nextPatternElement instanceOf  htmlElement)
                findMatchingElement (nextPatternElement, targetElement, configFile, pageSource, outputData)
            else if (nextPatternElement instanceOf  patternElement)
                findMatchingPattern (nextPatternElement, targetElement, configFile, pageSource, outputData)
            else if (nextPatternElement instanceOf  dataMarkingTextElement)  {
                data = getWholeInnerText (targetElement, pageSource)
                addDataToOutput (nextPatternElement, data, configFile, pageSource, outputData)
                if (existsAnotherInstanceOfDataMarkingTextElement (targetElement, nextPatternElement, configFile)) {mappingIndexElement = null return}
            }
            currentPatternElement = getNextSibling (configFile, nextPatternElement)
            if (currentPatternElement instanceOf skipAllElement) continue
            else { mappingIndexElement = null return }
        }else if (currentPatternElement instanceOf dataMarkingTextElement) {
            data = getFirstLevelText (targetElement, pageSource)
            addDataToOutput (currentPatternElement, data, configFile, pageSource, outputData)
            if (existsAnotherInstanceOfDataMarkingTextElement (targetElement, currentPatternElement, configFile)) {mappingIndexElement = null return}
             currentPatternElement = getNextSibling (configFile, currentPatternElement)
        }else if (currentPatternElement instanceOf skipSiblingElement) {
            nextPatternElement = getNextSibling (configFile, currentPatternElement)
            if (nextPatternElement instanceOf htmlElement)
                currentTargetElement  = handleMatchingElement (nextPatternElement, currentTargetElement, configFile, pageSource, outputData)
            else if (nextPatternElement instanceOf patternElement)
                currentTargetElement = handleMatchingPattern (nextPatternElement, currentTargetElement, configFile, pageSource, outputData)
            currentPatternElement = getNextSibling (configFile, nextPatternElement)
        }else if (currentPatternElement instanceOf skipTextElement) {
            skippedTextValue  =  getValue (currentPatternElement)
            if (getFirstLevelText (targetElement, pageSource).contains(skippedTextValue)) {
                nextPatternElement = getNextSibling (configFile, currentPatternElement)
                if (nextPatternElement instanceOf dataMarkingTextElement) {
                    data = getFirstLevelText (targetElement, pageSource)
                    data = data.replace (skippedTextValue,"")
                  addDataToOutput (nextPatternElement, data, configFile, pageSource, outputData)
                  if (existsAnotherInstanceOfDataMarkingTextElement (targetElement, nextPatternElement, configFile)) {mappingIndexElement = null return}
                }
            }else{ mappingIndexElement = null  return}
            if (existsAnotherInstanceOfSkippedTextElement (targetElement, currentPatternElement, configFile)) {mappingIndexElement = null return}
             currentPatternElement = getNextSibling (configFile, nextPatternElement)
        }
    } while (currentPatternElement)
}
```

FIGURE 7.6: Tree traversing.

13), multiple HTML elements from the code fragment will be mapped to *currentPatternElement*. The algorithm first retrieves *lowerValue* and *upperValue*, respectively, as the lower- and upper-bound values of the multiplicity (lines 14-15).

If *lowerValue* and *upperValue* are both set to the wildcard symbol (*) (line 16), it means that all adjacent HTML elements starting from *currentTargetElement* that are identical to *currentPatternElement* must be mapped to it. Consequently, the algorithm examines *currentTargetElement* and its next sibling elements (lines 17-20) and for every two identical elements calls **treeTraversing**. The iteration continues until a dissimilar HTML element form the code fragment is visited or no HTML element is left. After the iteration, the current pattern element is processed, its next sibling element is taken (line 21) and the program logically continues again from line 11.

If *lowerValue* and *upperValue* are both set to an integer number (line 22), it means that a predefined number of adjacent HTML elements starting from *currentTargetElement* that are identical to *currentPatternElement* should be mapped to it. *lowerValue* (and also *upperValue*) specifies up to how many HTML elements from the code fragment will be mapped to *currentPatternElement*.

If *lowerValue* is 0 and *upperValue* is 1 (line 29), it means that *currentPatternElement* is an optional element and there might not exist a matching element from the code fragment for this pattern element. If such an element is found (line 30), a new call for **treeTraversing** is made to process the two mapping elements.

If *currentPatternElement* does not have a multiplicity attribute (line 36), it means that only one HTML element form the code fragment is mapped to it. *currentPatternElement* and *currentTargetElement* are passed to **treeTraversing** (line 37) and their next sibling elements are considered (lines 38-39).

***currentPatternElement* is an instance of *patternElement*.** If *currentPatternElement* is a *patternElement* (line 41), the procedure also checks its next sibling element (line 42). If the next sibling element of *currentPatternElement* is an instance of *relationElement*, it means that there exist an or-relationship between two patterns. According to the grammar, these two patterns are the left and the right sibling elements of an element of type *relationElement*. *currentPatternElement* and *nextPatternElement* (line 43) both are instances of *patternElement*, meaning that their child text element is

the name of an auxiliary pattern. These two elements with *currentTargetElement* are passed two **handleOrRelation** (line 44). The main task of this procedure is to find the matching pattern for *currentTargetElement* and then to continue traversing within the chosen pattern. We recall that an auxiliary pattern has only one first-level child. **handleOrRelation** queries the first-level child of the two patterns, the one is identical to *currentTargetElement*, its owning pattern is chosen. **handleOrRelation** finally returns back the current target element (line 44).

If the next sibling element of *currentPatternElement* is not an instance of *relationElement* (line 46), *currentPatternElement* is a pattern that should be individually processed. **handlePattern** takes the responsibility for checking if *currentTargetElement* is identical to the first-level child element of the auxiliary pattern referenced in *currentPatternElement*. If true, it continues traversing within the pattern.

***currentPatternElement* is an instance of *skipAllElement*.** If *currentPatternElement* is the `skip(all)` element (line 50), then its next sibling element might be an instance of *htmlElement*, *patternElement*, or *dataMarkingTextElement*. The procedure first queries the next sibling element of `skip(all)` in *nextPatternElement* (line 51). If *nextPatternElement* is an instance of *htmlElement* (line 52), **findMatchingElement** finds all HTML elements in the target code fragment whose (immediate or indirect) parent element is *targetElement* and extracts their data items (line 53). If *nextPatternElement* is an instance of *patternElement* (line 54), **findMatchingPattern** looks for instances of the auxiliary pattern (whose name is referenced in *nextPatternElement*) within the *targetElement* element (line 55). If *nextPatternElement* is an instance of *dataMarkingTextElement* (line 56), **getWholeInnerText** collects the values of all text elements of children and grandchildren of *targetElement* (line 57), and **addDataToOutput** appends them as a single value to *outputData* (line 58). The name of the output data is specified with *nextPatternElement*. Remember that an HTML element can have only one child data marking text element. Therefore, as soon as the procedure processed one data marking text element, it checks if there exists another instance. If it finds it, it means that a conflict is observed and the process must be left (line 59). Also note that according to the grammar, if there exists a `skip(all)` element within the first-level children of an HTML element, every first-level child element of that HTML element must be led by a `skip(all)` element. To implement this restriction, once the `skip(all)` element and its following element are processed, the next element must also be an instance of

skipAllElement (lines 61-62). If not, there is a mismatch: the target code fragment no longer matches the data pattern, and the process is aborted (line 63).

***currentPatternElement* is an instance of *dataMarkingTextElement*.** If the current pattern element is a data marking text element (line 64), the values of all the first-level children text elements of *targetElement* are gathered and appended to *outputData*. The name of the output data is specified with *currentPatternElement* (lines 65-66). Again, the procedure ensures that only one data marking text element exists within the first-level children of *targetElement* (line 67).

***currentPatternElement* is an instance of *skipSiblingElement*.** If the current element is an instance of *skipSiblingElement* (line 69), then its next sibling element must be an instance of *htmlElement* or *patternElement*. The multiplicity value of *currentPatternElement* defines how many elements must be skipped after the last visited element of the target code fragment. The procedure first queries the next sibling element of *currentPatternElement* in *nextPatternElement* (line 70). If *nextPatternElement* is an instance of *htmlElement* (line 71), **handleMatchingElement** skips some sibling elements of *currentTargetElement* according to the multiplicity value, and finds the next HTML element that matches *nextPatternElement* and continues traversing (line 72). Once done, **handleMatchingElement** returns back the next HTML element of the target code fragment to be considered. If *nextPatternElement* is an instance of *patternElement* (line 73), it means that *nextPatternElement* is a reference to an auxiliary pattern. In this case, **handleMatchingPattern** finds the only child HTML element of the pattern and moves ahead the process with this HTML element and *currentTargetElement* (line 74). Similarly, **handleMatchingPattern** returns back the next HTML element of the target code fragment to be considered.

***currentPatternElement* is an instance of *skipTextElement*.** If *currentPatternElement* is an instance of *skipTextElement* (line 76), it means that *currentPatternElement* specifies a value that must be contained within the value of the first-level children text elements of *targetElement*. **getValue** first returns back the value specified in *currentPatternElement* (line 77) and then it is checked if the value is contained within the value of the first-level children text elements of *targetElement* (line 78). If true, the algorithm then visits the next sibling element of *currentPatternElement*. If this element is an instance of *dataMarkingTextElement* (line 80) the procedure collects the value of

all children text elements of *targetElement* (line 81). Since the value defined in an instance of *skipTextElement* must be visited but not extracted, it is removed from the gathered data (line 82), and data is added to *outputData* (line 83). If the value specified in *currentPatternElement* is not found within *targetElement*, then a conflict is observed (line 86). We would recall that an HTML element can only have one instance of a *skipTextElement* element. If *targetElement* has more, there is a mismatch between the target code fragment and the data pattern and the process must be aborted (line 87).

If **treeTraversing** is called for the tag tree representations of the data pattern and the target code fragment shown in Figure 7.4, the `td(2)` and `td(3)` elements are assigned to *patternElement* and *targetElement*, respectively. As the procedure is executing, the `input(3)` element is considered as the first child of *patternElement* and assigned to *currentPatternElement* (line 7). Accordingly, the `input(4)` is assigned to *currentTargetElement*. Since *currentPatternElement* is an HTML element and has no multiplicity attribute (see *cf:*3), line 37 is executed and a new call of **treeTraversing** for these two elements is made. Note that the `input(3)` and `input(4)` does not have sibling elements. Therefore, when line 38 is executed, *currentPatternElement* is set to *null*, the condition in line 84 becomes false, and consequently, the procedure terminates.

When **treeTraversing** is called for the `input(3)` and `input(4)` elements, **extractAttributeData** (line 6) is executed. For the `input` element in the configuration file (line *cf:*3), the `data-att-mar-type` data marking attribute is defined and its value is the value of the `type` attribute of the matching element from the target code fragment, i.e., `radio` (line *sc:*4). So, `radio` is assigned to `data-att-mar-type` and added to *outputData*. Since `input(3)` has no children, the condition in line 9 is true, **treeTraversing** terminates, and the control is returned back to the calling procedure. Figure 7.7 presents the current state of the mapping elements.

Figure 7.8 shows the state of the mapping elements when **treeTraversing** is called for `td(6)` and `td(11)` as *patternElement* and *targetElement*, respectively. The first child element of `td(6)` is `skip(all)` which is an instance of *skipAllElement* and its next sibling element is `label(8)` which is an HTML element. So, **treeTraversing** calls the **findMatchingELement** procedure (line 53). **findMatchingELement** skips all children and grandchildren of `td(11)` (thus `h6(12)` is skipped) and stops at `label(13)` in the code fragment. Then, a new **treeTraversing** is called for `label(8)` and `label(13)`

respectively as *patternElement* and *targetElement*. The first child of `label(8)` is a data marking text element, named `data-text-mar-option-name` (line *cf:*8). Consequently, the value of the child text element of `label(13)` is extracted, i.e. "Embossed Characters", assigned to `data-text-mar-option-name`, and appended to *outputData* (lines 65-66).

Backing to **treeTraversing** called for `td(6)` and `td(11)`, the next pattern element to be considered is `skip(all)` in line *cf:*9. This element is followed by the the *description* auxiliary pattern with the multiplicity value of 1 (line *cf:*10). **findMatchingPattern** (line 55) first finds the immediate child element of the pattern, i.e., the `p(14)` element. It then skips all children and grandchildren of `td(11)` in the code fragment and looks for a `p` element. **findMatchingPattern** finds an identical `p(15)` element in the code fragment and calls a new **treeTraversing** for `p(14)` and `p(15)`. The first child element of `p(14)` is `skip(all)` in line *cf:*15, and its next sibling element is the `data-tex-mar-description` data marking text element. Therefore, **getWholeInnerText** (line 57) skips all children and grandchildren of `p(15)` in the code fragment (`span(16)` and `span(18)` are discarded) and collects the value of all children and grandchildren text elements of the `p(15)`. As a result, the values in lines *sc:*16, 17, and 18 are gathered as a single string value, assigned to `data-tex-mar-description`, and added to *outputData* (line 58).

Now that all elements of the code fragment are mapped to the elements of the data pattern, then all calls of **treeTraversing** successfully terminate, the control is returned back to line 27 of the **dataExtractionProcedure** (Figure 7.5). **validData** validates *outputData* and since the data is valid, it is printed out.

The next candidate code fragment is the block lines *sc:*24-37. **treeTraversing** finds a conflict at line 55, because when **findMatchingPattern** is called, it queries the child element of the *description* pattern, i.e. the `p(14)` element. However, **findMatching-Pattern** can not find any `p` element in the children and grandchildren of `td` (line *sc:*32) in the code fragment, and since the *description* pattern is not an optional pattern, a mismatch is observed and the code fragment in lines *sc:*24-37 is ignored.

Now that no candidate code fragment is left to be considered by **dataExtractionPro-cedure**, the data extraction process completes.

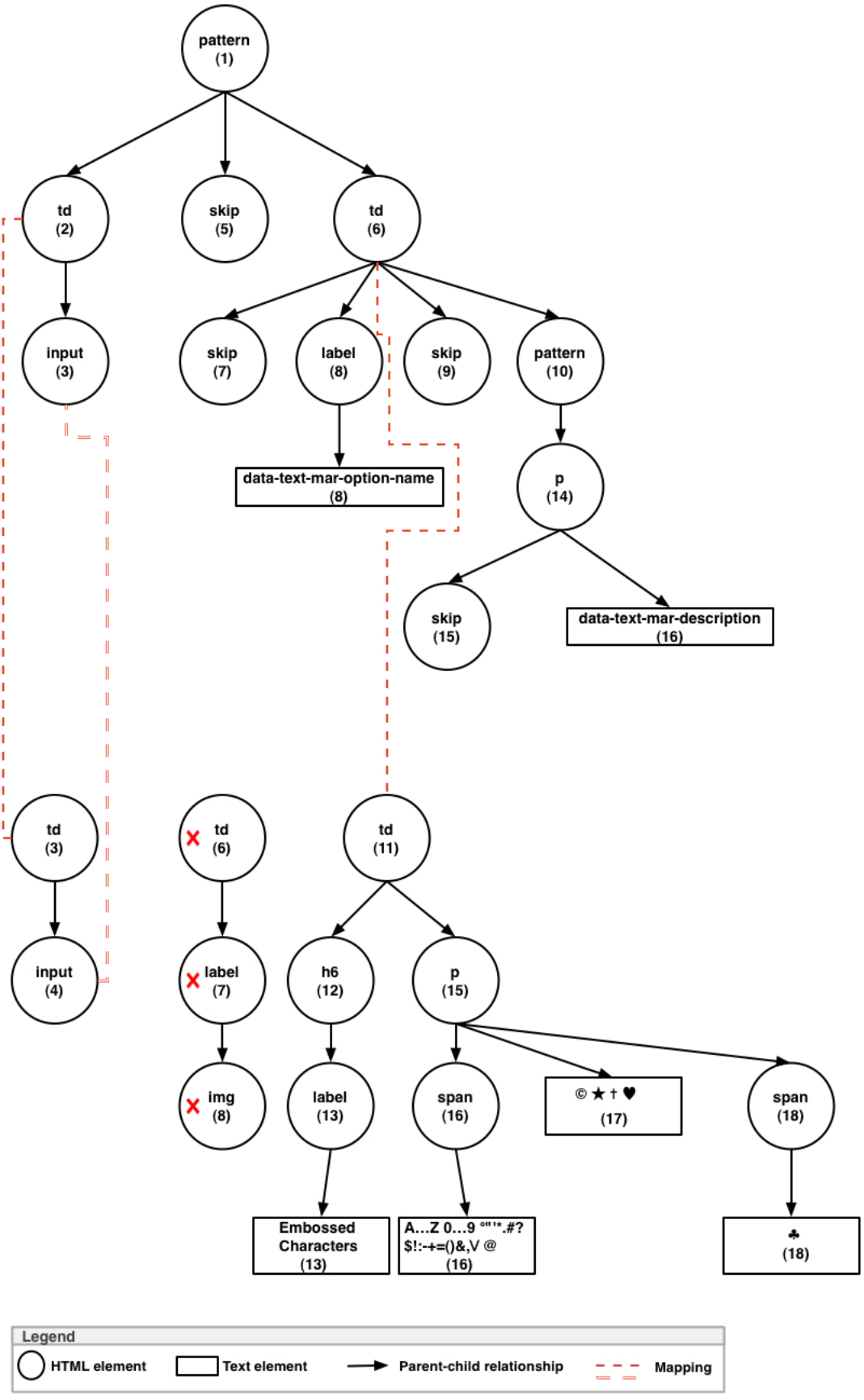


FIGURE 7.7: Tree representation (2).

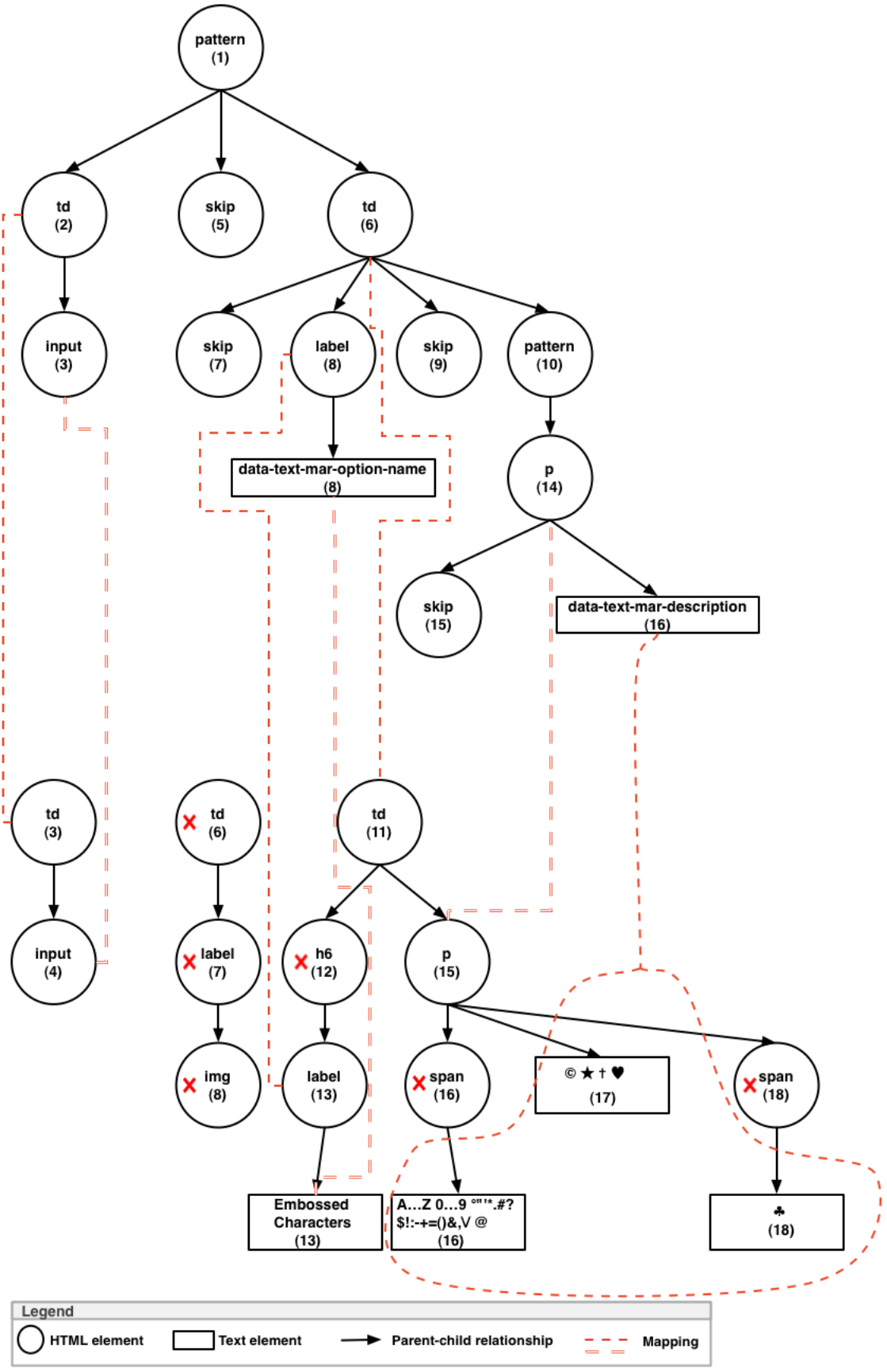


FIGURE 7.8: Tree representation (3).

7.2 Data Presentation

The data extracted by the Wrapper (in fact, **dataExtractionProcedure**) is hierarchically organized according to a predefined data model (Section 7.2.1) and serialized using an XML format (Section 7.2.2). The produced XML file is processed to add missing data, remove noisy data, etc. The clean XML file is then transformed into a TVL model (Section 7.2.3).

7.2.1 Data model

Figure 7.9 shows our proposed data model to structure the output data. The data model is divided into three packages: *configuration process*, *option*, and *constraint*.

Configuration process

A **Configuration Process** is constituted of a sequence of **Step**s and optionally nested steps. Users follow these steps to complete the configuration of a **Product**. Each step includes a subset of **Options** which are chosen by users to be included in the final product. Within a step, options are organized in different **Groups** and nested groups. We recall that a group describes logical dependencies between options, while a step denotes a part of the configuration process.

In our data model, the configuration process in a Web configurator must have at least one step and each step must have at least one group. If not, the user should define them in the pattern specification (the `data-att-mar-step-name` and `data-att-mar-group-name` attributes). A step is identified with its *name* and can optionally have a unique *ID*, an integer value denoting its *order* among other steps of the configuration process, and a *description*.

Option

A **Product** is characterized by a set of **Option**s. Each option is contained in a **Group** and is configured in a **Step**. A group is identified with a *name* and may be hierarchically

organized as *nested* groups. An option can be contained in one and only one group or nested group.

An option must have a *name* and is represented by a *widgetType*. If more than one instance of an option can be included in the final product, the option is *cloneable*. If the user does not have to configure an option, meaning that the option is *optional*. If an option is configured at the beginning of the configuration process, then its *selectedByDefault* attribute is set to *true*.

If the widget type of an option is *image* or its widget is combined with an image, then the *src* attribute stores the (absolute or relative) URL of the image. If, to configure an option, the user has to enter a value (in a *textBox*, *fileChooser*, etc.) the option might have a *defaultValue*.

An option may have *subOption*s. For instance, we consider a list box as an option and its items as suboptions. An option can also optionally have one or more **DescriptiveInformation**s associated to the option. A description information of an option is identified with a *name*, its *value*, and the *dataType* of the value.

Constraint

A **Constraint** determines valid combinations of options or imposes restrictions on values that can be entered for that option. We consider three types of constraints in our data model: *formatting*, *group*, and *cross-cutting* constraints.

Formatting constraint. A formatting constraint ensures that a valid value is set by the user. A **TypeChecking** constraint forces the user to enter a strongly typed value; a **RangeControl** constraint defines upper and lower bounds or defines the set of valid values to be entered; and a **FormattedValue** constraint controls that the value that is entered is in a special format.

Group constraint. A **group** constraint defines the number of options that can be selected from a group of options. The allowable number of options to be selected is defined in the *allowableCardinality* attribute of a group. If an option is cloneable, *cloneRange* specifies the minimum and maximum number of instances of the option to be included in the final product.

Cross-cutting constraint. A cross-cutting constraint is defined over two or more options. The selection of an option may *require*/*exclude* the selection of other options. More **complex constraint**s may exist that can be described in a formal way or in natural language.

7.2.2 Output XML file

The extracted data is serialized in an XML file whose structure conforms to the data model described in Section 7.2.1. The names of the data marking text elements and attributes in the data and auxiliary patterns are the tag names of XML elements representing their corresponding data items in the XML file. Moreover, a list of predefined and built-in tag names is used by the Wrapper to create/rename elements in the XML file.

For the source code in Figure 7.2 and the configuration file in Figure 7.3, the Wrapper created the XML file presented in Figure 7.10. Except for the tag names in lines 13 and 14, all other tag names are predefined in our system. By definition, the `step_name` element represents the step name, and `group_name` has the group name as its child text element. The option name marked in the configuration file as `data-tex-mar-option-name` (line *cf:*8) is documented as the `feature_name` element (line 11) in the XML file. The `properties` element presents the descriptive information extracted for an option (lines 12-15). `data-att-mar-type` and `data-tex-mar-description` data marking elements are used as tag names in the XML file (lines 13 and 14, respectively), with a minor change: the hyphen sign (-) in the names of the data marking elements in the configuration file is replaced with the underscore sign (_) in the output XML file. The reason for this replacement is that some software may think of the hyphen sign as the subtract operator.

7.2.3 TVL model

The final step of our reverse engineering process is to transform the XML file into a TVL model. This is performed by a module written in Java. The transformation may require some changes in the extracted data to produce valid content in TVL. For instance, in TVL a feature name (that corresponds to an extracted option name) has to start with

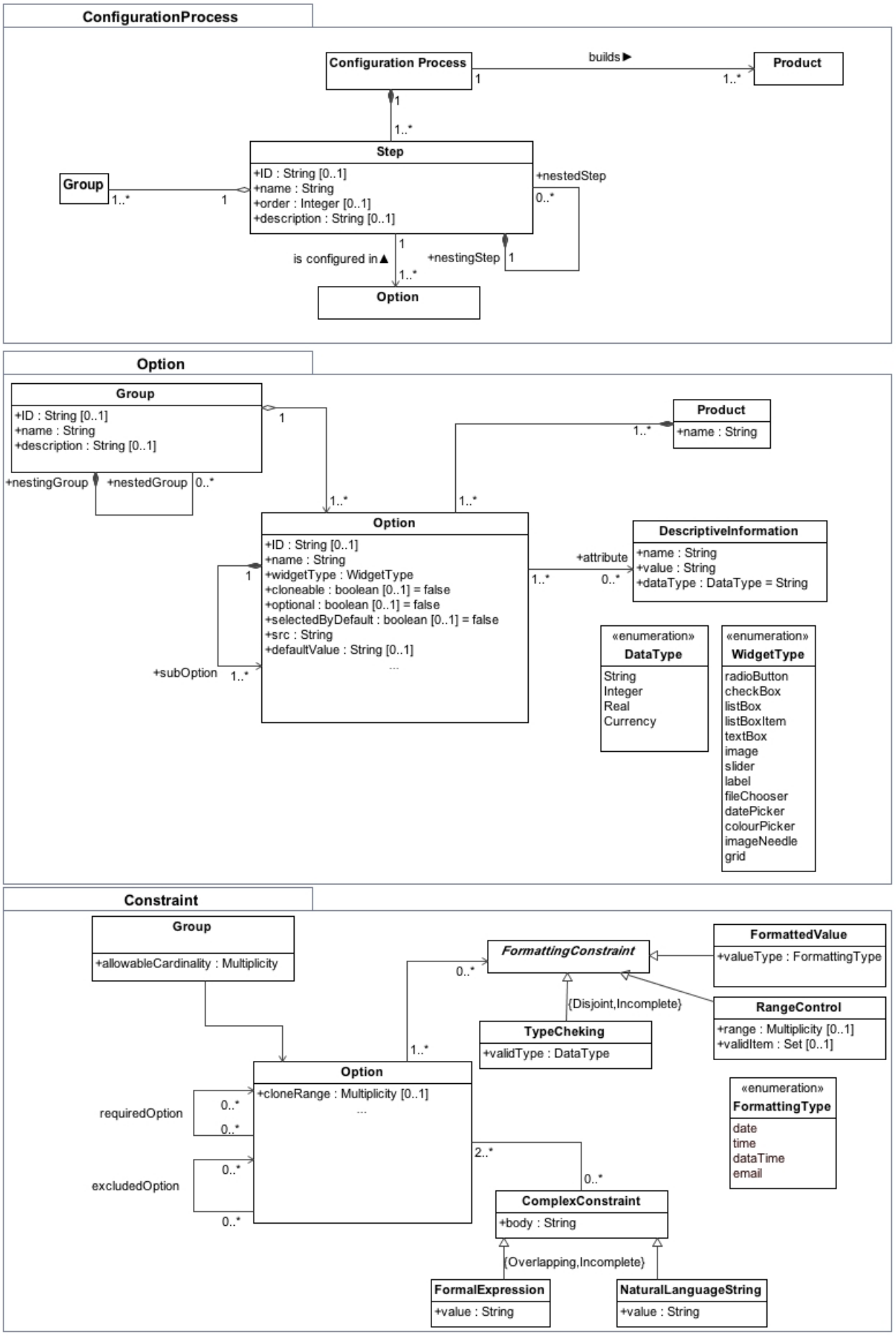


FIGURE 7.9: Schema of output data.

```
1 <?xml version="1.0" ?>
2 <configuration_process>
3   <steps>
4     <step>
5       <step_name>Step2</step_name>
6       <groups>
7         <group>
8           <group_name>Fonts</group_name>
9           <features>
10            <feature>
11              <feature_name>Embossed Characters</feature_name>
12              <properties>
13                <data_att_mar_widget_type>radio</data_att_mar_widget_type>
14                <data_tex_mar_description>A...Z 0...9 °"'*.#?$!:-+=()&,V @ © ★ † ♥ ♣</data_tex_mar_description>
15              </properties>
16            </feature>
17          </features>
18        </group>
19      </groups>
20    </step>
21  </steps>
22 </configuration_process>
```

FIGURE 7.10: An example output XML file.

an uppercase letter and can contain numbers as well as the underscore. If an option name extracted by the Wrapper breaks one of these rules, a conflict will be raised when generating the TVL model. To resolve these transformation conflicts, the user has to configure the transformation module to handle different types of conflicts. For instance, for each invalid character that may appear in the names of options, she should define replacement character that is valid in TVL. We designed an XML configuration file that the user can use to set up the transformation module.

7.3 Tool implementation

To implement the algorithms discussed in this chapter (and those that will be discussed in Chapter 8 as well) we developed a *Firebug*[2] extension (3 KLOC, 2 person-month). Firebug is a powerful Web development tool that allows to inspect and modify the source code of Web pages in real-time (Figure 7.11). The user can select an element on the Web page and inspect its source code in a panel. Moreover, she can edit an element (e.g., add, edit, or delete an attribute) or delete an element and inspect its impact on the Web page. Firebug is compatible with all major browsers, but our extension is tested only

[2] http://getfirebug.com/

for Firefox. It is installed on the browser as an add-on and has access to the source code of the page currently loaded in the browser as well as its DOM.

Firebug provides a set of APIs that can be used to add new features. The extension we developed is built on top of Firebug. It adds a new panel called *Web Wrapper* to the existing panels of Firebug. The *Web Wrapper* panel (see Figure 7.12) provides a GUI using which the user can define/load/update a configuration file, run the extraction procedure, and inspect the output XML file. If needed, the user can define a data region by easily adding a new element to the Web page or editing an existing element.

The algorithms are implemented in JavaScript. We used built-in JavaScript functions and *jQuery* APIs [3] to traverse and manipulate the HTML tag tree of the configuration file and DOM of the given Web page.

Once the XML file is produced by our extension, the user may need to edit the file. There are many XML processing tools to edit the XML file, e.g., *XMLSpear*[4], a free XML editor with real-time validation.

We also implemented a Java module (550 LOC, 1 person-week) to transform an XML file to a TVL model. It takes as input an XML file and a configuration file for resolving transformation conflicts, and generates as output a TVL file.

The delivered tools are available at `http://info.fundp.ac.be/~eab/result.html`.

7.4 Chapter Summary

In this chapter, we explained our proposed algorithm and illustrated its behaviour to find and extract data from code fragments that structurally match a given data pattern. The algorithm provides a two-step solution to find matching code fragments. It first attempts to find candidate code fragments by finding mappings between their first-level elements and those of the data pattern. Once the candidate code fragments are identified, the algorithm traverses each code fragment to find other mappings between it 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.

[3] `http://jquery.com/`
[4] `http://www.donkeydevelopment.com/`

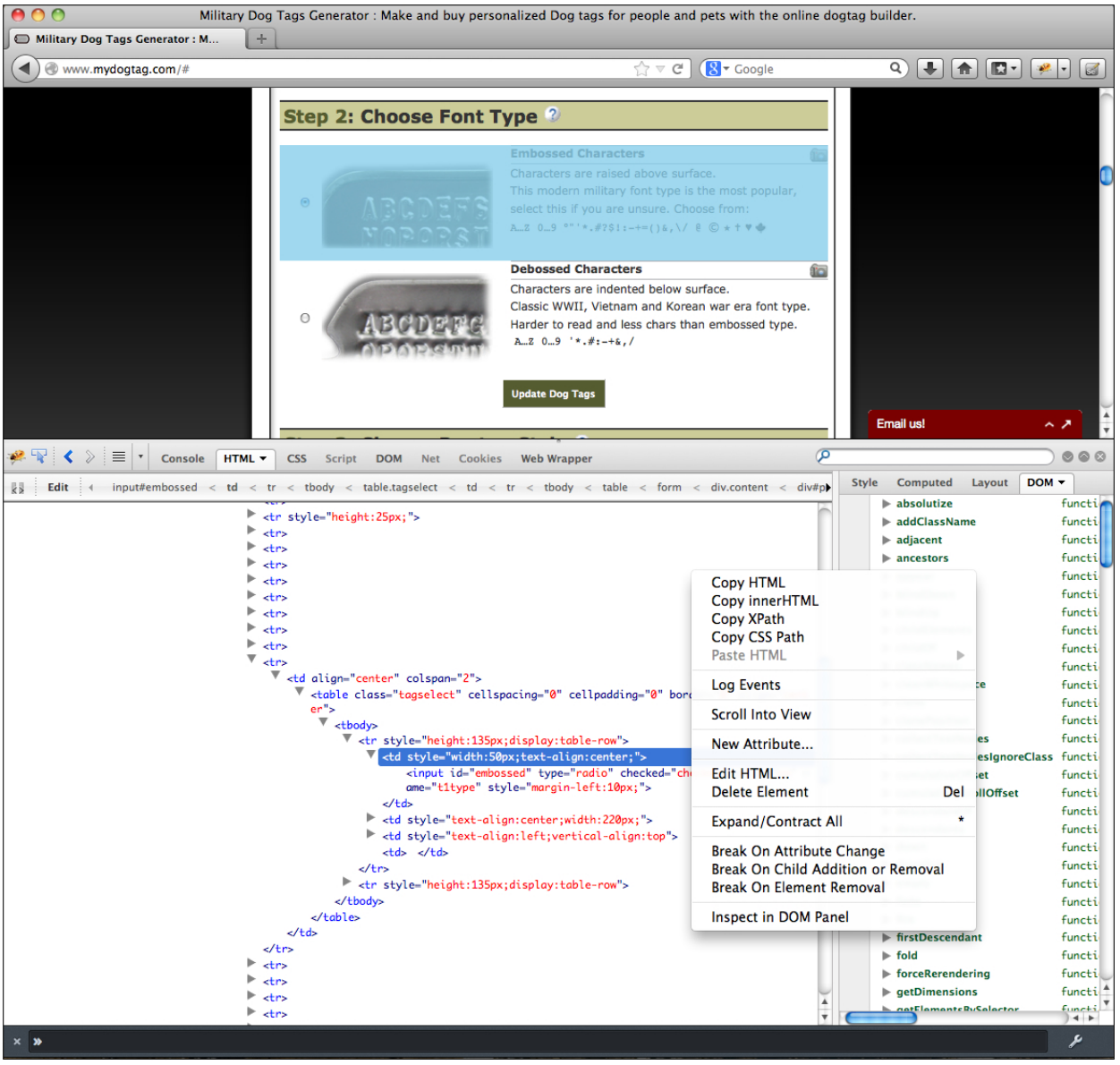


FIGURE 7.11: Firebug.

We also presented the data model based on which the extracted data is hierarchically organized, the XML format of the output data, and the transformation into TVL. We finally described the tools we developed to support the data extraction procedure.

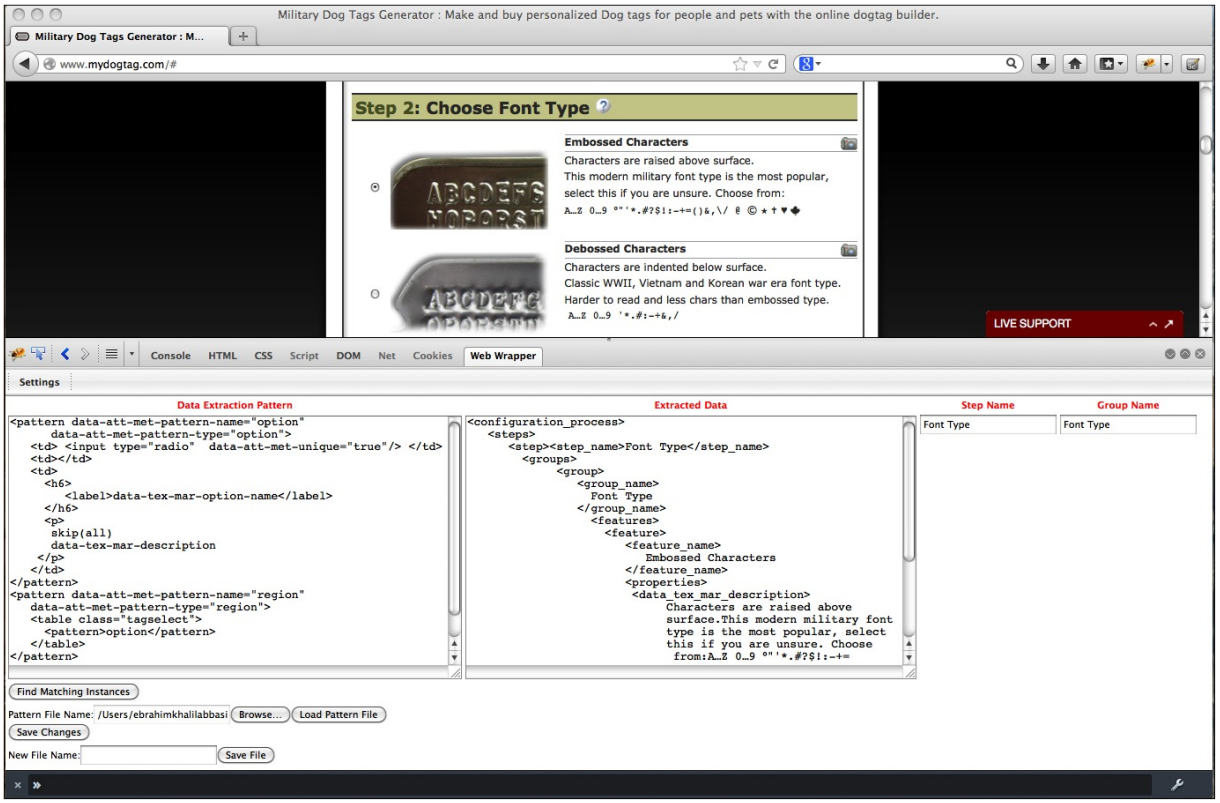


FIGURE 7.12: Web Wrapper extension.

Chapter 8

Extracting Dynamic Variability Data

Web configurators are highly interactive applications and as they are executing, new content may be automatically added to the page, and existing content may be removed or changed. The exploration of the configuration space (e.g., navigation through the configuration steps) and the configuration of options are two common actions that may change the content of the page.

This chapter presents our solution to extract dynamic variability data. We first introduce the notion of *dependency* between patterns, the main foundation on top of which our solution is developed (Section 8.1). We then present our crawling technique to automatically explore the configuration space and configure options (Section 8.2). We specifically explain how the Wrapper and the Crawler collaborate together to simulate the users' actions to systematically generate new content or alter the existing content, and to deduce and extract configuration-specific data. We finally explain the methods we offer to trigger and to extract constraints defined over options (Section 8.3).

8.1 Dependencies between *vde* patterns

A motivating example. Figure 8.1 shows a configuration step that contains a set of *packages* which can be selected to be included in the product (a car in this example). Each package is represented by a *parent option* (which is presented through a selectable

check box) and a set of *child options* (which are presented through a "•" or a disabled check box). Figure 8.2 presents the code fragments that implement "M Sport Package" shown in Figure 8.1. The code fragment in lines 2-10 corresponds to the parent option and the code fragments in lines 11-38 implement the child options. We identified three templates that are used to generate the structure of a package: one to generate the parent option, one for the child options presented using a dot ("•"), and one for the child options presented through a disabled check box. To extract packages, one may think of specifying a data pattern to extract the parent options and two auxiliary patterns to extract their child options. Besides being a complicated data pattern, this specification does not document the parent-child relationship between a parent option and its child options. The documentation of parent-child relationships between options facilitates the creation of feature hierarchy when generating the TVL model of the target configurator.

In addition to the parent-child relationship, we observed other types of relationships between data objects in a Web page that should be documented, because they exploit configuration-specific data. For instance, consider a case in which by selecting an option its implied options are dynamically loaded in the page. It means that there are underlying constraints (a kind of relationship) between the chosen option and the newly added options. These constraints should be identified and extracted as well.

To deal with these issues, we define the notion of *dependency* between *vde* patterns.

Definition: Dependency between *vde* patterns Let P_1 and P_2 be two data patterns. A *dependency* is a relationship that semantically relates the set of code fragments that match P_2 to a code fragment that matches P_1. We respectively call P_1 and P_2 as *independent* and *dependent* patterns.

To define the dependency, we use the `data-att-met-dependent-pattern` meta attribute. This attribute is specified in the region pattern (independent region pattern) that denotes the independent data pattern. The value of the attribute is a comma-separated list of region patterns (dependent region patterns) that indicate the dependent data patterns.

In the configuration file that contains two or more region patterns, it should be explicitly indicated which of those patterns is the first pattern to be processed by the Wrapper. We use the `data-att-met-root-pattern = "true"` attribute to denote the starting pattern in the configuration file.

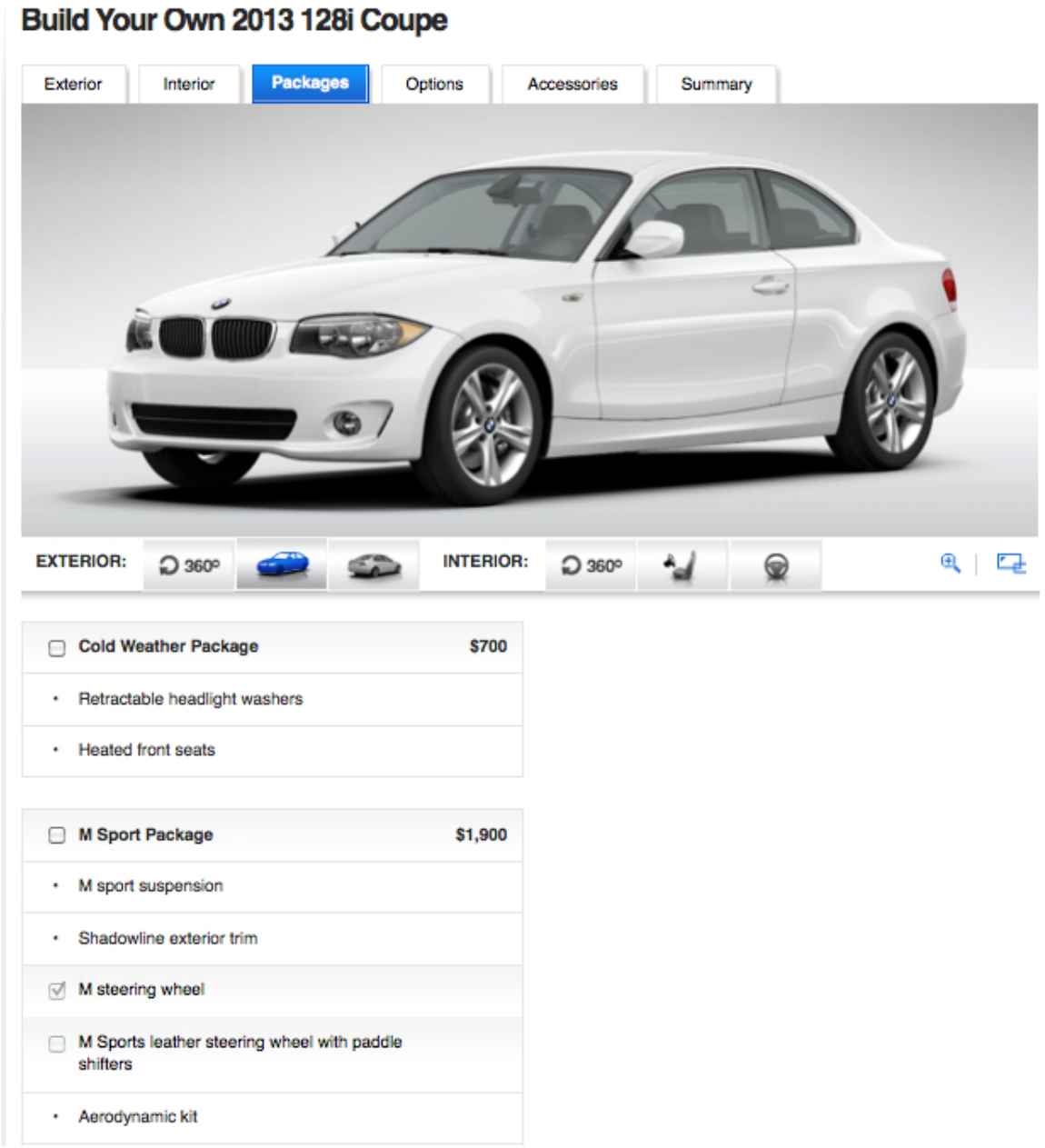


FIGURE 8.1: Parent-child relationship between objects (`http://www.bmwusa.com/`, October 22 2013).

Figure 8.3 presents the patterns specified to extract packages from the page shown in Figure 8.1. *parentOptionData* (lines 8-16), *childOptionData* (lines 22-27), and *childOptionSelectableData* (lines 33-40) are three data patterns specified to extract the parent options, the child options presented using a dot, and the child options presented using a disabled check box, respectively. *parentOptionRegion* (lines 1-7), *childOptionRegion* (lines 17-21), and *childOptionSelectableRegion* (lines 28-32) are region patterns that

```
1  <div class="packageGroupContainer">

2    <div class="packageOptionParent">
3      <div class="packageCheckBoxContainer">
4        <input  type="checkbox">
5      </div>
6      <div class="optionInfo">
7        <h4  class="optionName">M Sport Package</h4>
8      </div>
9      <div class="optionPrice">$1,900</div>
10   </div>

11   <div class="packageOptionChild">
12     <span class="packageCheckBoxContainer">•</span>
13     <span class="optionInfo">M sport suspension</span>
14   </div>

15   <div class="packageOptionChild">
16     <span class="packageCheckBoxContainer">•</span>
17     <span class="optionInfo">Shadowline exterior trim</span>
18    </div>

19    <div class="packageOptionChildSelectable">
20      <div class="packageCheckBoxContainer">
21        <input  type="checkbox" checked="true" disabled="disabled">
22      </div>
23      <div class="optionInfo">
24        <a class="optionName">M steering wheel</a>
25      </div>
26    </div>

27    <div class="packageOptionChildSelectableLast">
28      <div class="packageCheckBoxContainer">
29        <input type="checkbox"  disabled="disabled">
30      </div>
31      <div class="optionInfo">
32        <a class="optionName">M Sports leather steering wheel with paddle shifters</a>
33      </div>
34    </div>

35   <div class="packageOptionChild">
36     <span class="packageCheckBoxContainer" style="float:left;">•</span>
37     <span id="subOptionName_2_4" class="optionInfo">Aerodynamic kit</span>
38   </div>

39 </div>
```

FIGURE 8.2: The code fragments for "M Sport Package" shown in Figure 8.1.

respectively denote the *parentOptionData*, *childOptionData*, and *childOptionSelectable-Data* patterns. Note that all the region patterns point to the same region in the page (lines 4,18, and 29).

In Figure 8.3, a dependency is defined between *parentOptionData* as the independent pattern and *childOptionData* and *childOptionSelectableData* as the dependent patterns (line 2). The Wrapper starts the data extraction process with the *parentOptionData*

pattern (line 3). It first extracts data from a code fragment that structurally matches *parentOptionData* and then seeks to find and extract data from all code fragments in the indicated region that match *childOptionData* or *childOptionSelectableData*.

The data object extracted with respect to a dependent data pattern is documented as the child data object of the object extracted with respect to the corresponding independent data pattern. For instance, in Figure 8.4 that represents the data extracted from "M Sport Package" shown in Figure 8.1 given the configuration file presented in Figure 8.3, "M Sport Package" is the parent object (line 2) and all other objects are documented as the child objects. In the XML file, each child object has a <parent_feature> element that contains the name of the parent object (lines 10, 14, 18, 22, and 31).

Using the notion of dependency between patterns, we offer a technique to automatically crawl the configuraion space (Section 8.2) and a method to trigger and extract constraints defined over options (Section 8.3).

8.2 Crawling the Configuration Space

In a configurator, the whole configuration space, i.e., configuration steps, groups, and configuration-specific objects, is not presented in the currently loaded page. It may be distributed over multiple pages each having a unique URL and including a subset of configuration-specific objects (*multi-page* user interface paradigm), or all the configuration-specific objects are contained in a page (*single-page* user interface paradigm). These paradigms are not mutually exclusive and a configurator can implement both.

For configurators that follow the single-page paradigm, we observed two common patterns to present options:

- Once the page is initially loaded in the browser, it presents all the configuration-specific objects to the user.
- When the page is initially loaded in the browser, not all content is represented at once, but instead, as the application is executing, new configuration-specific content may be automatically added to the page, and existing content may be removed or changed. For instance, in the configurator appearing in Figure 8.6, the

```
1  <pattern  data-att-met-pattern-name="parentOptionRegion"  data-att-met-pattern-type="region"
2            data-att-met-dependent-pattern="childOptionRegion, childOptionSelectableRegion"
3            data-att-met-root-pattern="true">
4     <div class="packageGroupContainer">
5        <pattern> parentOptionData </pattern>
6     </div>
7  </pattern>

8  <pattern data-att-met-pattern-name="parentOptionData" data-att-met-pattern-type="data">
9     <div class="packageOptionParent" data-att-met-unique="true">
10       skip(all)
11       <input type="checkbox" data-att-mar-widget-type="@type" data-att-mar-checked="@checked">
12       skip(all)
13       <h4>data-tex-mar-option-name</h4>
14       skip(all)
15       <div class="optionPrice">data-tex-mar-price</div>
16 </pattern>

17 <pattern data-att-met-pattern-name="childOptionRegion" data-att-met-pattern-type="region" >
18    <div class="packageGroupContainer">
19       <pattern> childOptionData </pattern>
20    </div>
21 </pattern>

22 <pattern data-att-met-pattern-name="childOptionData" data-att-met-pattern-type="data">
23    <div class="packageOptionChild" data-att-met-unique="true">
24       skip(all)
25       <span class="optionInfo">data-tex-mar-option-name</span>
26    </div>
27 </pattern>

28 <pattern data-att-met-pattern-name="childOptionSelectableRegion" data-att-met-pattern-type="region">
29    <div class="packageGroupContainer">
30       <pattern> childOptionSelectableData </pattern>
31    </div>
32 </pattern>

33 <pattern data-att-met-pattern-name="childOptionSelectableData" data-att-met-pattern-type="data">
34    <div class="packageOptionChildSelectable*" data-att-met-unique="true">
35       skip(all)
36       <input data-att-mar-widget-type="@type" data-att-mar-checked="@checked" data-att-mar-disabled="@disabled"/>
37       skip(all)
38       <a class="optionName">data-tex-mar-option-name</a>
39    </div>
40 </pattern>
```

FIGURE 8.3: The configuration file to extract options shown in Figure 8.1.

selection of an option from the "Model line" group loads its consistent options to the "Body style" group.

To extract all the configuration-specific content, the whole configuration space should be explored. It means that all the pages of a configurator containing configuration-specific content should be navigated and all the actions that may generate dynamic configuration-specific content should be performed. Due to the diversity of patterns used by configurators to generate and present data objects, developing a scalable and

```
<feature>
    <feature_name>M Sport Package</feature_name>
    <properties>
        <data_att_mar_widget_type>checkbox</data_att_mar_widget_type>
        <data_tex_mar_pricr>$1,900</data_tex_mar_pricr>
    </properties>
</feature>
<feature>
    <feature_name>M sport suspension</feature_name>
    <parent_feature>M Sport Package</parent_feature>
</feature>
<feature>
    <feature_name>Shadowline exterior trim</feature_name>
    <parent_feature>M Sport Package</parent_feature>
</feature>
<feature>
    <feature_name>Aerodynamic kit</feature_name>
    <parent_feature>M Sport Package</parent_feature>
</feature>
<feature>
    <feature_name>M steering wheel</feature_name>
    <parent_feature>M Sport Package</parent_feature>
    <properties>
        <data_att_mar_widget_type>checkbox</data_att_mar_widget_type>
        <data_att_mar_checked>true</data_att_mar_checked>
        <data_att_mar_disabled>disabled</data_att_mar_disabled>
    </properties>
</feature>
<feature>
    <feature_name>M Sports leather steering wheel with paddle shifters</feature_name>
    <parent_feature>M Sport Package</parent_feature>
    <properties>
        <data_att_mar_widget_type>checkbox</data_att_mar_widget_type>
        <data_att_mar_disabled>disabled</data_att_mar_disabled>
    </properties>
</feature>
```

FIGURE 8.4: An excerpt of the XML file representing the extracted data for "M Sport Package" shown in Figure 8.1.

automatic technique to crawl the configuration space is rather complicated (if not impossible). We observed that configurators following the multi-page paradigm usually consist of a relatively small set of pages containing configuration-specific content. The user can manually explore them and run the data extraction procedure for each page individually. In this PhD, we offer a semi-automatic approach to crawl the configuration space in a Web page and extract dynamic configuration-specific content.

Crawling the configuration space for the purpose of dynamic data generation and extraction requires (1) the *exploration* of the configuration space, i.e., activating all containers in a page that may include configuration-specific content, and (2) the *configuration* of options, i.e., giving new value to options. For instance, activation of a configuration step makes available/visible its contained options in the page and makes unavailable/hidden those of other steps.

Automatically crawling the configuration space in a Web page requires (1) the simulation of users' exploration and configuration actions to systematically generate new content or alter existing content of the page, and then (2) the analysis of the changes made to the page to extract or deduce configuration-specific data. We implemented a *Web Crawler*

that simulates some of the users' actions to generate new data. The newly generated data is then extracted by the Wrapper.

8.2.1 Simulating Users' Actions

In a Web page, the exploration and configuration actions are clicking on clickable widgets (e.g., menu, button, check box, radio button, image), selecting an item from a list box, and entering a value in text inputs. Simulating the action of entering a value in a text input is a way to trigger the corresponding input-validation function and then to deduce from that the formatting and cross-cutting constraints defined over the text input (see Section 8.3.1). At present, the Crawler provides no support for the simulation of entering input values.

In the pattern specification, the element to be clicked by the Crawler is identified by the `data-att-met-clickable = "true"` attribute in the data pattern. If this element is a list box, the Crawler selects all the items of the list box one by one, otherwise it clicks on the element.

8.2.2 Analysing Page State Changes

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 must be analysed to identify the newly added content and to deduce from that the configuration-specific data. We observed that when an exploration or configuration action is performed by the user, a few identifiable regions on the page are impacted and their content may be changed. Consequently, rather than analysing the whole page, only those regions should be investigated. Based on this observation, we divide the configuration space into two groups: *independent* and *dependent* regions. When an action is performed on a configuration-specific object in an independent region, new objects are added to the dependent regions or existing ones are changed.

Using the notion of dependency between patterns we formulate this observation. In fact, the region pattern owning the independent pattern denotes the region of clickable elements, and the region patterns owning the dependent data patterns indicate the regions of added/changed objects. This formulation, for instance, allows to specify a

relationship between a pattern specified to extract configuration steps (the independent pattern) and patterns specified to extract options included in each step (the dependent patterns).

Figure 8.5 presents the data extraction procedure followed by the Web Wrapper and the Web Crawler. This algorithm is the modified version of the algorithm shown in Figure 7.5 in Chapter 7. The algorithm is improved to be used for the crawling purpose. Comparing with the previous version, **dataExtractionProcedure** requires two more input parameters. First, *currentRegionPatternName* denotes that among all the region patterns defined in the configuration file (*configFile*) which one should be currently processed by the Wrapper and the Crawler. Since each region pattern has only one data pattern, having the name of a region pattern is enough to identify the data pattern to be processed. Second, *parentObjectName* is the name of the data object that the data objects extracted from the currently processing region pattern have dependency with that data object. For instance, *parentObjectName* can be the name of a configuration step and *currentRegionPatternName* the name of the region pattern that indicates the region containing options of that step. Or, *parentObjectName* can be the name of an option and *currentRegionPatternName* the name of the region pattern that denotes the region of options loaded in the page as the result of selecting that option. Before calling **dataExtractionProcedure** for the first time, the name of the region pattern with which the extraction process starts is identified (line 3) and then the procedure is called (line 4). Note that *parentObjectName* is **null**.

For each candidate code fragment, the Web Wrapper first extracts its data (lines 8-30). If *parentObjectName* is not *null* (line 9), it is also added to the output data (line 10). When the extraction process is completed by the Wrapper for the current code fragment, the Crawler starts the crawling process. It identifies the *clickableElement* in the code fragment (line 31) with respect to the specification of the data pattern. The element with the `data-att-met-clickable = "true"` attribute in the data pattern is used to identify the clickable element in the code fragment. The Crawler also finds out the name of the extracted data object (line 32). This name, in fact, is the the extracted value in *outputData* marked with either the `data-att-mar-option-name` attribute or the `data-tex-mar-option-name` text element in the data pattern.

If *clickableElement* exists (line 33), the Crawler retrieves the list of the dependent region

patterns from the `data-att-met-dependent-pattern` attribute of the current region pattern (line 34). If *clickableElement* is a list box (line 35), the Crawler first gets the list of items of the list box (line 36) and then iterates over each item (lines 37-43). The parent object name to be assigned to the dependent pattern consists of the name of the list box and the name of the currently selected item of the list box, separated by a dot (line 38). For each item, the Crawler chooses it as the selected item (line 39) and then calls the **dataExtractionProcedure** (relaunches the Web Wrapper) for each dependent region pattern (lines 40-42). If *clickableElement* is not a list box (it may be a radio button, check box, button, image, etc.), the Crawler simulates its click event (line 45) and calls **dataExtractionProcedure** (lines 46-48). When the Crawler simulates the selection of an item from a list box or the click event on an element, it in fact programmatically makes changes to the regions denoted by the dependent region patterns, therefore the Wrapper is recalled to extract data from those regions (lines 41 and 47).

Example 8.1: Figure 8.6 presents three option groups in a configurator, namely "Model line", "Body style", and "Model". Because of underlying cross-cutting constraints between options included in these groups, the selection of an option from "Model line" adds its consistent options to "Body style", and in turn, the selection of an option from "Body style" loads its consistent options into "Model". We use the dependency between patterns to crawl the whole configuration space of these three groups.

Figure 8.7 shows the patterns specified to extract variability data from the page shown in Figure 8.6. *dataPattern* (lines 1-6) is a data pattern defined to extract options. The region patterns *modelLineRegion* (lines 7-12), *bodyStyleRegion* (lines 13-18), and *modelRegion* (lines 19-24) denote respectively the "Model line", "Body style", and "Model" groups.

The Wrapper starts the process from the "Model line" group (`data-att-met-root-pattern = "true"` – line 8). It extracts data from the first code fragment that matches *dataPattern*. Consequently, the option "Audi A1" is extracted. When data for the "Audi A1" option is extracted, the Crawler selects this option by clicking on the widget representing it. This selection loads the new options "3 door" and "Sportback" to the "Body style" group. From the specification of the *modelLineRegion* pattern, the Crawler finds that the next region pattern to be investigated is *bodyStyleRegion* (`data-att-met-dependent-pattern = "bodyStyleRegion"` – line 8). It thus calls the Wrapper to process this region pattern.

```
1  if (!validConfigFile(configFile))
2   return
3  startingRegionPatternName = getStartingRegionPattern (configFile)
4  dataExtractionProcedure (configFile, startingRegionPatternName, null, pageSource)

5 dataExtractionProcedure (configFile, currentRegionPatternName, parentObjectName, pageSource) {
6     candidateCodeFragments[] = getCandidateCodeFragments (configFile, currentRegionPattern, pageSource)
7     for each ((codeFragment in candidateCodeFragments) and (codeFragment.mappingIndexElement != null)) {
8         outputData = new outputData()
9         if (parentObjectName != null)
10              addParentObject (outputData, parentObjectName )
11        mappingIndexElement = codeFragment.mappingIndexElement
12        for (i=0; i< codeFragment.previousMappingElement.length ) {
13            patternElement = codeFragment.previousMappingElement[i]
14            mappedElement = codeFragment.previousMappedElement[i]
15            traverseTree (mappingIndexElement, patternElement, mappedElement, configFile, pageSource, outputData)
16            if (mappingIndexElement == null)
17                goToLine (8)
18        }
19        patternElement = codeFragment.codeFragment.mappingIndexElement
20        mappedElement = codeFragment.codeFragment.mappedIndexElement
21        traverseTree (mappingIndexElement , patternElement, mappedElement, configFile, pageSource, outputData)
22        if (mappingIndexElement == null)
23            goToLine (8)
24        for (i = 0; i< codeFragment.nextMappingElement.length ) {
25            patternElement = codeFragment.nextMappingElement[i]
26            mappedElement = codeFragment.nextMappedElement[i]
27            traverseTree (mappingIndexElement, patternElement, mappedElement, configFile, pageSource, outputData)
28            if (mappingIndexElement == null)
29                goToLine (8)
30        }
31        clickableElement = getClickableElement (configFile, currentRegionPatternName, codeFragment)
32        objectName = getObjectName (outputData)
33        if (clickableElement != null) {
34            dependentPatternList = getDependentPatternList (configFile, currentRegionPatternName)
35            if (clickableElement instanceOf selectElement) {
36                elementItems = getElementItems (clickableElement)
37                for (j = 0; j < elementItems.length; j++) {
38                    objectNameFullText = objectName+ "." + elementItems[j].text
39                    selectItem (clickableElement, elementItems[j])
40                    for (k = 0; k < dependentPatternList.length; k++) {
41                        dataExtractionProcedure (configFile, dependentPatternList[k], objectNameFullText, pageSource)
42                    }
43                }
44            } else {
45                    clickElement (clickableElement)
46                    for (k = 0; k < dependentPatternList.length; k++) {
47                        dataExtractionProcedure (configFile, dependentPatternList[k], objectName, pageSource)
48                    }
49            }
50        }
51        if (validData (outputData))
52            addToExtractedData (outputData)
53    }
54 }
```

FIGURE 8.5: The adopted data extraction procedure for the purpose of crawling.

The Wrapper starts the process for the "Body style" group (denoted by the *bodyStyleRegion* pattern) and extracts the option "3 door". Then, the Crawler selects this option which loads "A1" to the "Model" group (Figure 8.6(a)). Similarly, by analysing the *bodyStyleRegion* pattern, the Crawler finds out that *modelRegion* is the next region pattern to be examined (`data-att-met-dependent-pattern = "modelRegion"` – line 14). It therefore calls the Wrapper for this pattern. The Wrapper starts extracting data from the "Model" group (denoted by the *modelRegion* pattern) and extracts data for the option "A1". Once done, the Crawler selects "A1" but since there is no dependent region pattern defined for *modelRegion*, the Crawler stops. The Wrapper finds no more data to be extracted from the "Model" group, comes one step back, and considers the "Sportback" option in the "Body style" group. The Wrapper extracts data for "Sportback". The Crawler selects the option which loads "A1 Sportback" to the "Model" group (Figure 8.6(b)). Then, the Wrapper extracts data for "A1 Sportback" and the Crawler selects it. Once that all the options in the "Body style" and the "Model" groups are extracted, the Wrapper returns back to the "Model line" group and takes the "Audi A3" option as the next option to be processed. This process iterates until all the options in "Model line", and accordingly in "Body style" and "Model", are extracted.

Note that data objects extracted from the independent and dependent regions inherit their dependency. For instance, there is a dependency between "Audi A1" (the independent option) and "3 door" and "Sportback" (the dependent options). When generating the TVL model, these dependencies are interpreted and documented.

Figure 8.8 presents an excerpt of the output XML file generated for the options shown in Figure 8.6. Note that for each option in the "Body style" and "Model" groups a parent option (represented through the `parent_feature` XML element) is also documented that is its corresponding parent option respectively in the "Model line" and "Body style" groups. For example, the parent option for the "3 door" and "Sportback" options is the option "Audi A1". It means that by selecting "Audi A1" in the "Model line" group, "3 door" and "Sportback" are loaded in the "Body style" group.

(a) Options *Audi A1* and *3 door* are selected.

(b) Options *Audi A1* and *Sportback* are selected.

FIGURE 8.6: Dynamic Content (`http://configurator.audi.co.uk/`, July 3 2013).

```
1  <pattern data-att-met-pattern-name="dataPattern" data-att-met-pattern-type="data">
2     <div class = "radioButtonl*checked*"  data-att-met-unique="true" data-att-met-clickable="true" data-att-mar-widget-type="@class">
3        <span class="jqRadioButton"></span>
4        <span class ="label">data-tex-mar-option-name </span>
5     </div>
6  </pattern>

7  <pattern data-att-met-pattern-type="region" data-att-met-pattern-name="modelLineRegion" data-att-mar-step-name="Model"
8           data-att-mar-group-name="Model line" data-att-met-root-pattern="true" data-att-met-dependent-pattern="bodyStyleRegion">
9     <div class="*gridLeft">
10       <pattern> dataPattern </pattern>
11    </div>
12 </pattern>

13 <pattern data-att-met-pattern-type="region" data-att-met-pattern-name="bodyStyleRegion" data-att-mar-step-name="Model"
14          data-att-mar-group-name="Body Style" data-att-met-dependent-pattern="modelRegion">
15    <div class="*gridCenter">
16       <pattern> dataPattern </pattern>
17    </div>
18 </pattern>

19 <pattern data-att-met-pattern-type="region" data-att-met-pattern-name="modelRegion" data-att-mar-step-name="Model"
20          data-att-mar-group-name="Model">
21    <div class="*gridRight">
22       <pattern> dataPattern </pattern>
23    </div>
24 </pattern>
```

FIGURE 8.7: The patterns specified to crawl the page shown in Figure 8.6.

8.3 Extracting Constraints

Extracting constraints defined over configuration options is a challenging issue because of different strategies used by Web configurators to implement them. In some cases, a combination of crawling scenarios with the data extraction approach is needed in order

```
<?xml version="1.0" ?>
<configuration_process>
    <steps xmlns="http://www.mozilla.org/keymaster/gatekeeper/there.is.only.xul">
        <step>
            <step_name>Model</step_name>
            <groups>
                <group>
                    <group_name cardinality="[1..1]">Model line</group_name>
                    <features>
                        <feature>
                            <feature_name cardinality="[1..1]">Audi A1</feature_name>
                            <properties>
                                <data_att_mar_widget_type>radioButton</data_att_mar_widget_type>
                            </properties>
                        </feature>
                        <feature>
                            <feature_name cardinality="[1..1]">Audi A3</feature_name>
                            <properties>
                                <data_att_mar_widget_type>radioButton</data_att_mar_widget_type>
                            </properties>
                        </feature>
                        . . .
                    </features>
                </group>
                <group>
                    <group_name cardinality="[1..1]">Body Style</group_name>
                    <features>
                        <feature>
                            <feature_name cardinality="[1..1]">3 door</feature_name>
                            <parent_feature>Audi A1</parent_feature>
                            <properties>
                                <data_att_mar_widget_type>radioButton</data_att_mar_widget_type>
                            </properties>
                        </feature>
                        <feature>
                            <feature_name cardinality="[1..1]">Sportback</feature_name>
                            <parent_feature>Audi A1</parent_feature>
                            <properties>
                                <data_att_mar_widget_type>radioButton</data_att_mar_widget_type>
                            </properties>
                        </feature>
                        . . .
                        <feature>
                    </features>
                </group>
                <group>
                    <group_name cardinality="[1..1]">Model</group_name>
                    <features>
                        <feature>
                            <feature_name>A1</feature_name>
                            <parent_feature>3 door</parent_feature>
                            <properties>
                                <data_att_mar_widget_type>radioButton</data_att_mar_widget_type>
                            </properties>
                        </feature>
                        <feature>
                            <feature_name>A1 Sportback</feature_name>
                            <parent_feature>3 door</parent_feature>
                            <properties>
                                <data_att_mar_widget_type>radioButton</data_att_mar_widget_type>
                            </properties>
                        </feature>
                        . . .
                    </features>
                </group>
            </groups>
        </step>
    </steps>
</configuration_process>
```

FIGURE 8.8: Output XML file for the page shown in Figure 8.6.

to systematically trigger and extract constraints. This section presents our solutions to extract formatting, group, and cross-cutting constraints.

8.3.1 Formatting Constraints

A formatting constraint ensures that values set in input elements (e.g., text boxes) are valid. In this PhD thesis, we target those formatting constraints that are encoded/presented in the page.

Textual formatting constraints. Some configurators describe formatting constraints in the GUI with textual explanation. Such constraints can be extracted by marking such text in the *vde* patterns.

Example 8.2: Figure 8.9 presents a configuration step containing text boxes whose values should be set by the user. Allowed characters that can be used to enter values are shown to the user (①). The patterns appearing in Figure 8.10 are specified to extract text boxes (as options) and the formatting constraint shown in Figure 8.9. Note that the constraint is presented generally and not attached to a specific option. Therefore, we define a dependency (`data-att-met-dependent-pattern = "constraintRegion"` – line 2) between the pattern specified to extract options (*optionData* – lines 7-12) and the one specified to extract the constraint (*constraintData* – lines 18-23). `data-tex-mar-constraint-valid-characters` (line 21) is a data marking element that captures the allowed characters (① in Figure 8.9). Figure 8.11 represents an excerpt of the produced XML file for applying the patterns (Figure 8.10) to extract data from the given page (Figure 8.9).

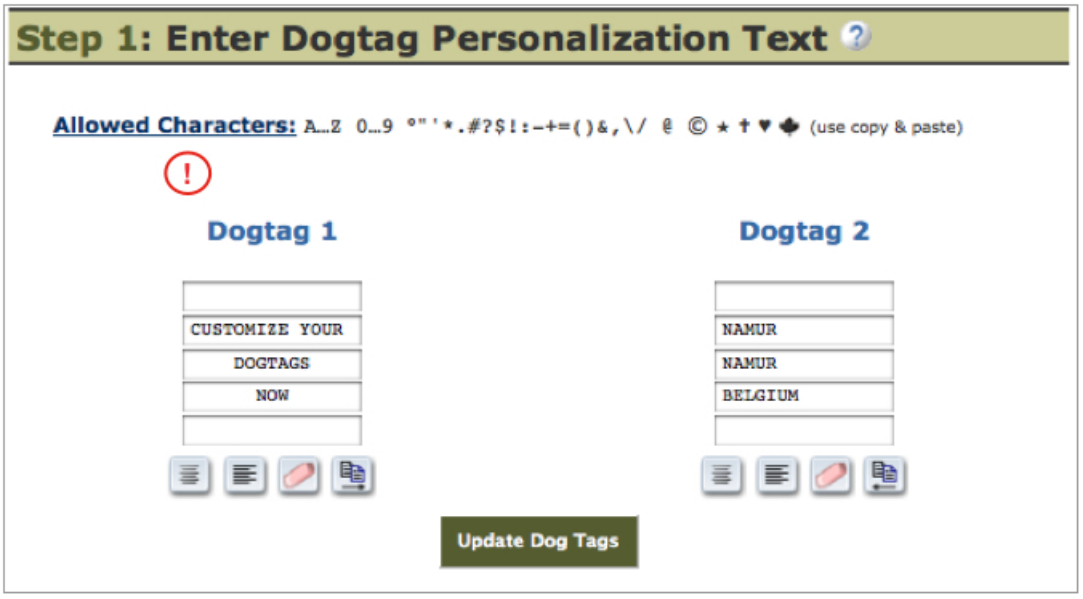


FIGURE 8.9: Textual formatting constraint (`http://www.mydogtag.com/`, June 13 2013).

Formatting constraints defined in tag attributes. We observed that some formatting constraints can be extracted from tag attributes of an input element. For instance, a common formatting constraint is the constraint that defines up to how many characters can be entered by the user in an input element. The attribute *maxlength*

```
1  <pattern data-att-met-pattern-name="optionRegion" data-att-met-pattern-type="region"
2     data-att-met-root-pattern="true"  data-att-met-dependent-pattern="constraintRegion">
3     <table class="options">
4        <pattern>optionData</pattern>
5     </table>
6  </pattern>

7  <pattern data-att-met-pattern-name="optionData" data-att-met-pattern-type="data">
8     <input class="tagfield"   type="text"
9        data-att-mar-constraint-max-length="@maxlength" data-att-mar-default-value="@value"
10       data-att-mar-option-name="@name"                 data-att-mar-widget-type="@type"
11       data-att-met-unique="true"                        data-att-met-clickable="true"/>
12    </pattern>

13    <pattern data-att-met-pattern-name="constraintRegion"  data-att-met-pattern-type="region">
14       <td class="constraint">
15          <pattern>constraintData</pattern>
16       </td>
17    </pattern>

18    <pattern data-att-met-pattern-name="constraintData" data-att-met-pattern-type="data">
19       <p data-att-met-unique="true">
20          skip(all)
21          data-tex-mar-constraint-valid-characters
22       </p>
23    </pattern>
```

FIGURE 8.10: Patterns specified to extract text boxes and their formatting constraint shown in Figure 8.9.

```
1  <?xml version="1.0" ?>
2     ...
3  <feature>
4     <feature_name>t1r3</feature_name>
5     <constraints>
6        <data_att_mar_constraint_max-length>15</data_att_mar_constraint_max-length>
7     </constraints>
8     <properties>
9        <data_att_mar_default_value>DOGTAGS</data_att_mar_default_value>
10       <data_att_mar_widget_type>text</data_att_mar_widget_type>
11    </properties>
12 </feature>
13 <feature>
14    <parent_feature>t1r3</parent_feature>
15    <constraints>
16       <data_tex_mar_constraint_valid_characters>
17          Allowed Characters:  A…Z 0…9 °"'*.#?$!:-+=()&amp;amp;,\/ @ © ★ † ♥ ♣  (use copy &amp;amp; paste)
18       </data_tex_mar_constraint_valid_characters>
19    </constraints>
20 </feature>
21    ...
```

FIGURE 8.11: The XML file produced for options shown in Figure 8.9.

is usually used to specify the maximum number of allowed characters. In fact, by extracting the value of these attributes we can extract such formatting constraints. The `data-att-mar-constraint-length` in Figure 8.10 (line 9) is defined to extract the max-length formatting constraint defined for input elements in Figure 8.9.

`data_att_mar_constraint_max_length` in the output XML file documents this constraint (Figure 8.11 – line 6).

Formatting constraints controlling bounds of a slider. A slider element lets the user either enter a value bounded by a minimum and maximum value or move its handle to select a value from a predefined domain. We extract the lower and upper bounds (Figure 8.12 – Ⓐ) of a slider as formatting constraints.

FIGURE 8.12: Formatting constraints controlling bounds of sliders.

Deduce formatting constraints from the context data. Our analysis of Web configurators reveals that in some cases there are valuable clues in the option name or other attached descriptive information that can be used to deduce formatting constraints. For instance, the words such as *size*, *length*, *width*, *inches*... in the option name are signs showing that the valid value to be set is an *integer* or *real* number. Using a *natural language processing-based approach* we can detect and extract such formatting constraints. We leave this approach for future work.

Extracting formatting constraints by dynamic analysis. Our practical experience with Web configurators shows that using the aforestated approaches we can extract a large number of constraints. However, we observed a few cases where detection and extraction of formatting constraints requires dynamic analysis and simulation strategies. Since this requires the simulation of entering values in input elements. At present the Wrapper and the Crawler do not support this.

8.3.2 Group Constraints

A group constraint defines the number of options that can be selected from a group of options. Widgets used to implement groups directly handle these constraints. Therefore, to detect and extract group constraints we need to analyse the types of widgets used to represent the grouped options. When extracting each option, the Wrapper also documents its widget. `data-att-mar-widget-type` and `data-att-mar-sub-widget-type` are used in the pattern specification to record the widget type of every extracted option. Once done, this data is analysed to deduce the applied group constraint. The constraint defined on a group is documented in the `cardinality` attribute of the relevant element in the output XML file.

The `cardinality` attribute is assigned to the XML element that denotes the name of the group containing the extracted options. In two situations an option can be also assigned the `cardinality` attribute. First, if the option contains sub-options. A common example is a list box that contains a number of items. In this case, the `cardinality` attribute is assigned to the XML element that contains the list box name. Second, if there is a dependency relationship between an independent option and a set of dependent options. We assume that the dependent options build an implicit group. The cardinality of this implicit group is defined in its corresponding independent option.

Grouped options are represented using radio buttons. For the options represented using radio buttons, in addition to their widget type, we also need to extract the value of their `name` attributes. Options represented through radio buttons and having the same value for their `name` attributes belong to a group. They in fact implement an *alternative* group. An *alternative* constraint is documented with the `cardinality = "[1..1]"` attribute in the appropriate element in the output XML file. We would note that configurators may use custom attributes to encode the type, the name, etc. of widgets. For these cases, we mark and extract these attributes and then analyse them to deduce the group constraints.

Example 8.3: In the pattern specification appearing in Figure 8.7, `data-att-mar-widget-type = "@class"` (line 2) records the widget type of options that match the given data pattern. The value of the attribute `class` represents the widget type of the corresponding option. The element `data_att_mar_widget_type` in the output XML file (Figure 8.8) contains this extracted data (i.e., `radioButton`). Since all the

options are represented using radio buttons (see Figure 8.6), the cardinality of the "Model line", "Body style", and "Model" groups is defined to be `1..1` in the XML file. Also note that, an option in the "Model line" group ("Body style" group, respectively) has dependent options in the "Body style" group ("Model" group). Therefore it is assigned the `cardinality` attribute as well.

Figure 8.13 presents a set of options which are rendered through radio buttons. Although all options are contained in one group, i.e., "GENERAL", they are semantically organized into three different alternative groups. In these cases, the Wrapper creates and adds a dummy group in the XML file and categorizes options within this group. The name of the dummy group is the value of the `name` attribute of the contained options. For this example, the Wrapper creates three dummy groups, namely "generalShoulders", "generalBack", and "generalBelly".

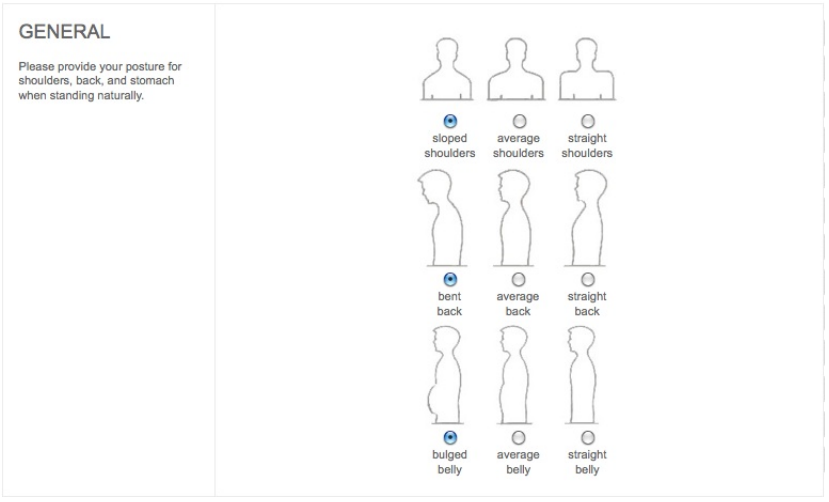


FIGURE 8.13: Three groups of options presented using radio buttons (`http://www.shirtsmyway.com/`, October 24 2013).

A list box represents an option. When an option is represented using a single-selection list box, we assume its items being the sub-options. These sub-options constitute an implicit *alternative* group. Therefore, the Wrapper assigns the `cardinality = "[1..1]"` to the element that denotes the option name in the output XML file.

Grouped options represented using images. If all the extracted grouped options are represented through images, they implement an *alternative* group.

Grouped options represented using check boxes. If all the extracted grouped options are represented using check boxes, they implement a *multiple choice* group. A

multiple choice constraint is documented with the `cardinality = "1..*"` attribute in the appropriate element in the output XML file. In very rare cases we observed that some exclusive options are implemented by (non exclusive) check boxes. We leave the automatic detection of these cases for future work.

Grouped options represented using text boxes. If all the extracted grouped options are represented using text boxes, they implement a *multiple choice* group.

A slider with a discrete domain represents an option. When the domain of a slider representing an option consists of a finite number of discrete values (Figure 8.12 – Ⓑ), the Wrapper considers such values as the sub-options of the slider. In this case, the Wrapper assigns an *alternative* group constraint defined as `cardinality = "[1..1]"` to the element that denotes the option name in the output XML file.

8.3.3 Cross-cutting constraints

A cross-cutting constraint is defined over two or more options regardless of their inclusion in a group. Extracting cross-cutting constraints is a challenging issue in reverse engineering of Web configurators because they follow different strategies to implement and handle cross-cutting constraints (Chapter 3). This makes it hard and likely impossible to implement a generic approach to deal with this variation in the implementation of cross-cutting constraints. In this thesis, we nevertheless propose a number of approaches to tackle the problem of extracting cross-cutting constraints.

8.3.3.1 Cross-cutting constraints displayed in the GUI

In some Web configurators, cross-cutting constraints are documented as annotations to the corresponding options. The constraint may be described with a textual explanation or be a list of *required* or *excluded* options attached to an option. For these cases, cross-cutting constraints can be marked in the *vde* pattern specification and be extracted like other data.

Example 8.4: Figure 8.14 shows an excerpt of the configuration environment of a Web configurator. Each option is annotated with a list of *required* options. It means that there is a *required* cross-cutting constraint between the parent option and the listed child

options. A cross-cutting constraint is also attached to the "Ergonomic sports front seats" option with a textual explanation, i.e., "changes seat trim to Lace Cloth". Cross-cutting constraints displayed in the GUI can be treated and extracted like other data presented in the page.

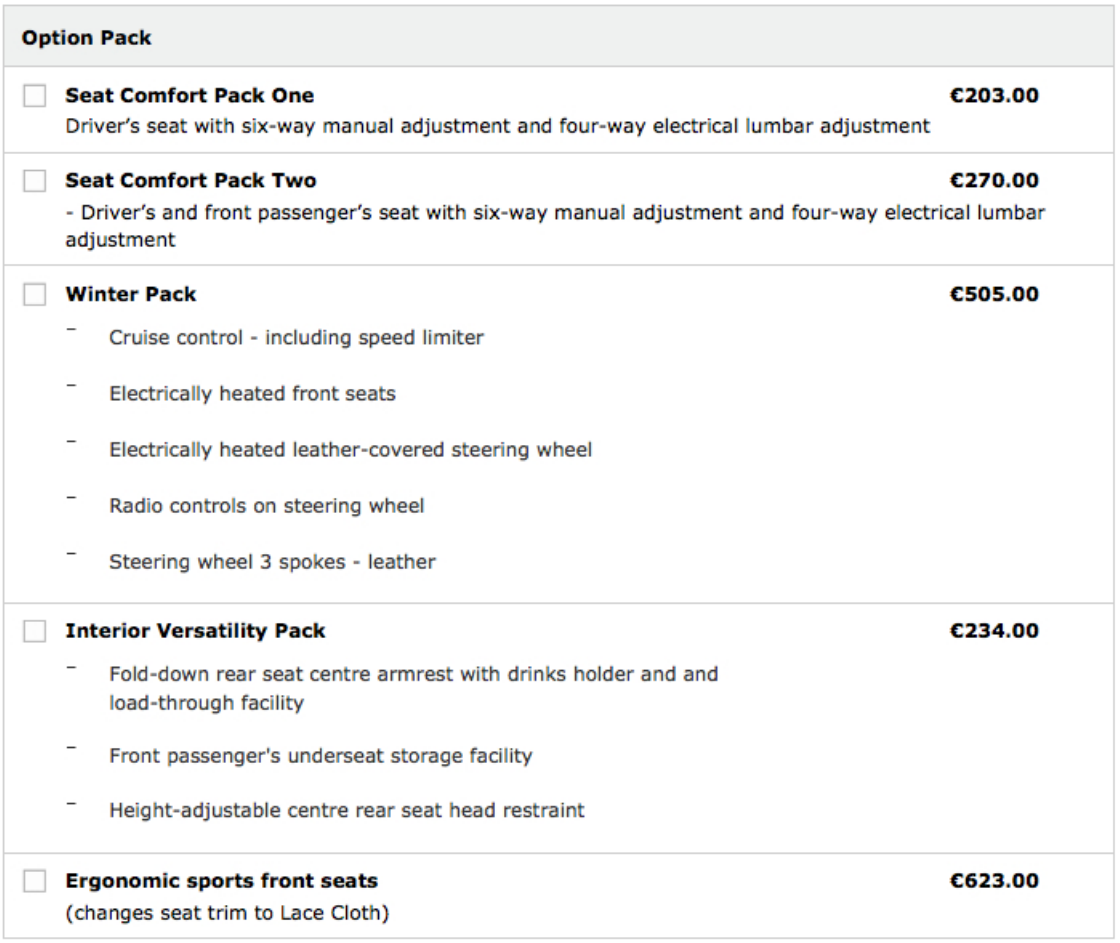


FIGURE 8.14: Cross-cutting constraints displayed in the GUI (`http://www.opel.ie/`, July 31 2013).

8.3.3.2 Cross-cutting constraints defined between independent and dependent options

A very common scenario followed by configurators is the loading of new consistent dependent options to the page upon the selection of an existing independent option. Here, in fact, there is a cross-cutting constraint between the independent option and the dependent options. To extract such cross-cutting constraints, we use the notion of dependency between *vde* patterns. The independent and dependent patterns are

respectively defined to extract the independent and dependent options. By crawling the configuration space, i.e., selecting all independent options and recording their dependent options, we can extract the underlying cross-cutting constraints as well. We should indicate that in the XML file where we document the dependency between independent and dependent options. The appropriate cross-cutting constraints are deduced and recorded when generating the TVL model from the XML file. Note also that as it is discussed in Section 8.3.2, in some cases, we also assume a group constraint defined over the dependent options.

Example 8.5: Figure 8.15 presents the configuration environment of a configurator in which options are represented using list boxes. There are cross-cutting constraints between items included in the "Manufacturer" (Ⓐ) and the "Model" (Ⓑ) list boxes, so that by selecting an item from the former (the independent option) new items are automatically added to the latter (the dependent options). Figure 8.16 shows patterns specified to extract data from the "Manufacturer" (lines 1-14) and the "Model" (lines 16-28) list boxes. Note that "Manufacturer" is the independent option and "Model" the dependent option (`data-att-met-dependent-pattern = "frameModelRegion"` – line 2). The Wrapper first extracts all the items of the "Manufacturer" list box, and then the Crawler selects its items one by one. The modification of an item in "Manufacturer" automatically changes the items in the "Model" list box which are recorded by the Wrapper. The output XML file is presented in Figure 8.17. For example, lines 47 to 60 show that by selecting "Aragon 18" in the "Manufacturer" list box (line 49) the following items are loaded to the "Model" list box (lines 54-58): "Choose a Model", "——", "Krypton", "Gallium", and "Gallium Pro".

8.3.3.3 Cross-cutting constraints shown in popup windows

In some configurators, when an option is given a new value and one or more constraints apply, the configurator asks the user to confirm or discard a decision before altering other options (*controlled* decision propagation pattern). In some cases, the configurator requires the user to resolve a conflict before proceeding with the configuration process. In this case, the configurator presents the conflict and a choice to the user, and the user makes the required decisions (*guided* decision propagation pattern). Configurators that follow these decision propagation patterns present a popup window to the user and display that which options are affected and how. Therefore, the contents of these

FIGURE 8.15: Independent and dependent options (`http://www.wrenchscience.com/`, July 31 2013).

```
1  <pattern data-att-met-pattern-name="manufacturerRegion"     data-att-met-pattern-type="region"
2           data-att-met-dependent-pattern="frameModelRegion"  data-att-met-root-pattern="true">
3     <div class="box1">
4        <pattern>manufacturerData</pattern>
5     </div>
6  </pattern>
7
8  <pattern data-att-met-pattern-name="manufacturerData"       data-att-met-pattern-type="data">
9     <select data-att-mar-option-name="@name"                 data-att-mar-widget-type="select"
10           data-att-met-clickable="true"                    data-att-met-unique="true"
11           id="ctl00_pageContentRegion_frameManufacturer">
12       <option data-att-met-multiplicity="[*]">data-tex-mar-sub-option-name</option>
13    </select>
14 </pattern>
15
16 <pattern data-att-met-pattern-name="frameModelRegion"      data-att-met-pattern-type="region">
17      <div class="box1">
18         <pattern>frameModelData</pattern>
19      </div>
20  </pattern>
21
22  <pattern data-att-met-pattern-name="frameModelData"       data-att-met-pattern-type="data">
23      <select data-att-mar-option-name="@id"                data-att-mar-widget-type="select"
24         data-att-met-clickable="true"                      data-att-met-unique="true"
25         id="frameModel">
26         <option data-att-met-multiplicity="[*]">data-tex-mar-sub-option-name</option>
27      </select>
28  </pattern>
```

FIGURE 8.16: Specified *vde* patterns to extract data from the page shown in Figure 8.15.

windows carry valuable data about the cross-cutting constraints. By analysing these contents we can extract such constraints. Our empirical analysis confirms that these windows are template-generated objects and, consequently, *vde* patterns can be specified to extract their content, i.e., cross-cutting constraints defined over options.

Example 8.6: Figure 8.18 presents an example of a configurator that follows the *controlled* decision propagation pattern. It shows the situation where the option "Sport

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration_process>
    <steps>
        <step>
            <step_name>no-step</step_name>
            <groups>
                <group>
                    <group_name>no-group</group_name>
                    <features>
                        <feature>
                            <feature_name cardinality="[1..1]">ctl00$pageContentRegion$frameManufacturer</feature_name>
                            <properties>
                                <data_att_mar_widget_type>select</data_att_mar_widget_type>
                            </properties>
                            <sub_features>
                                <sub_feature_name>Choose a Manufacturer</sub_feature_name>
                                <sub_feature_name>Argon 18</sub_feature_name>
                                <sub_feature_name>BMC</sub_feature_name>
                                <sub_feature_name>Cinelli</sub_feature_name>
                                <sub_feature_name>Colnago</sub_feature_name>
                                <sub_feature_name>De Rosa</sub_feature_name>
                                <sub_feature_name>Eddy Merckx</sub_feature_name>
                                <sub_feature_name>Ellsworth</sub_feature_name>
                                <sub_feature_name>Ibis</sub_feature_name>
                                <sub_feature_name>Intense</sub_feature_name>
                                <sub_feature_name>Knolly</sub_feature_name>
                                <sub_feature_name>Litespeed</sub_feature_name>
                                <sub_feature_name>Look</sub_feature_name>
                                <sub_feature_name>Moots</sub_feature_name>
                                <sub_feature_name>Niner</sub_feature_name>
                                <sub_feature_name>Parlee</sub_feature_name>
                                <sub_feature_name>Pinarello</sub_feature_name>
                                <sub_feature_name>Time</sub_feature_name>
                                <sub_feature_name>Yeti</sub_feature_name>
                            </sub_features>
                        </feature>
                        <feature>
                            <feature_name cardinality="[1..1]">frameModel</feature_name>
                            <parent_feature>ctl00$pageContentRegion$frameManufacturer.Choose a Manufacturer</parent_feature>
                            <properties>
                                <data_att_mar_widget_type>select</data_att_mar_widget_type>
                            </properties>
                            <sub_features>
                                <sub_feature_name>Choose a Model</sub_feature_name>
                            </sub_features>
                        </feature>
                        <feature>
                            <feature_name cardinality="[1..1]">frameModel</feature_name>
                            <parent_feature>ctl00$pageContentRegion$frameManufacturer.Argon 18</parent_feature>
                            <properties>
                                <data_att_mar_widget_type>select</data_att_mar_widget_type>
                            </properties>
                            <sub_features>
                                <sub_feature_name>Choose a Model</sub_feature_name>
                                <sub_feature_name>--------------------</sub_feature_name>
                                <sub_feature_name>Krypton</sub_feature_name>
                                <sub_feature_name>Gallium</sub_feature_name>
                                <sub_feature_name>Gallium Pro</sub_feature_name>
                            </sub_features>
                        </feature>
                        <feature>
                            <feature_name cardinality="[1..1]">frameModel</feature_name>
                            <parent_feature>ctl00$pageContentRegion$frameManufacturer.BMC</parent_feature>
                            <properties>
                                <data_att_mar_widget_type>select</data_att_mar_widget_type>
                            </properties>
                            <sub_features>
                                <sub_feature_name>Choose a Model</sub_feature_name>
                                <sub_feature_name>--------------------</sub_feature_name>
                                <sub_feature_name>impec</sub_feature_name>
                                <sub_feature_name>timemachine TMR01</sub_feature_name>
                                <sub_feature_name>teammachine SLR01</sub_feature_name>
                                <sub_feature_name>granfondo GF01</sub_feature_name>
                            </sub_features>
                        </feature>
                        ...
```

FIGURE 8.17: The output XML file produced for the page shown in Figure 8.15 and the patterns given in Figure 8.16.

Portfolio Pack with 19" Aq" is selected and a popup window appeared which tells that the selection of this option will lead to *ADDING* and *REMOVING* other options, meaning that there are *required* and *excluded* cross-cutting constraints between them. By extracting the content of this window we can extract constraints defined over options. The popup window is encoded in a region of the page and becomes visible when needed. The *vde* patterns to crawl and to extract data from this page are displayed in Figure 8.19. Each option represented in the page is an independent option and the popup window that

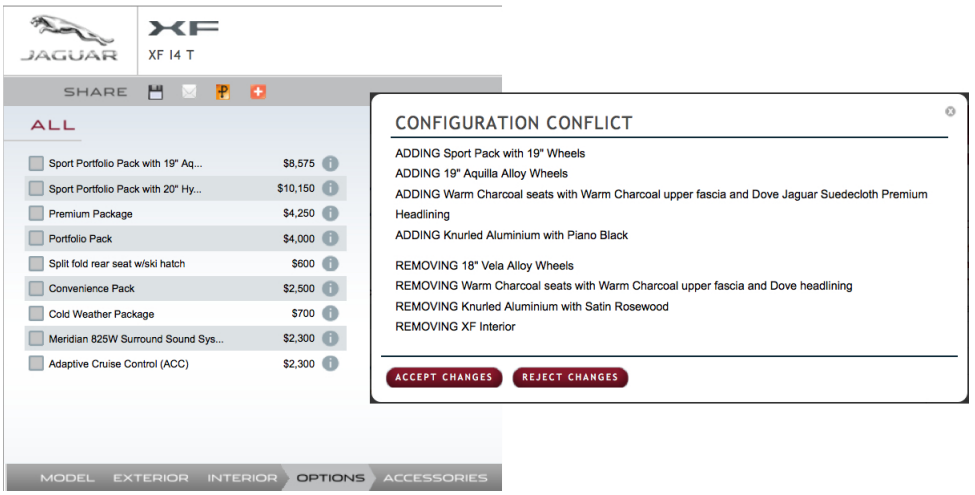


FIGURE 8.18: Controlled decision propagation (`http://www.jaguarusa.com/`, July 31 2013).

appears after the selection of the option is a dependent object. The *conflictResolutionRegion* pattern denotes the region of the window in the page, and *conflictResolutionData* is a data pattern that specifies the template from which the window is generated. Figure 8.20 shows the produced output XML file. The constraints are documented in the `constraints` XML element (lines 20 and 45).

```
1  <pattern data-att-met-pattern-type="region" data-att-met-pattern-name="OptionsRegion"
2           data-att-met-root-pattern="true"   data-att-met-dependent-pattern="conflictResolutionRegion"
3           data-att-mar-step-name ="Options"  data-att-mar-group-name ="Options">
4     <li class="listPanel">
5        <pattern>
6           optionsData
7        </pattern>
8     </li>
9  </pattern>
10
11 <pattern data-att-met-pattern-name="optionsData" data-att-met-pattern-type="data">
12    <a class="with|without" data-att-mar-full-name="@title" data-att-mar-widget-type="checkbox"
13                            data-att-met-unique="true" data-att-met-clickable="true">
14       data-tex-mar-option-name
15    </a>
16    <span class="price">data-tex-mar-price</span>
17 </pattern>
18
19 <pattern data-att-met-pattern-type="region" data-att-met-pattern-name="conflictResolutionRegion">
20    <div id="sb-content">
21       <pattern>
22          conflictResolutionData
23       </pattern>
24    </div>
25 </pattern>
26
27 <pattern data-att-met-pattern-name="conflictResolutionData" data-att-met-pattern-type="data">
28    <div class="conflict-resolution" data-att-met-unique="true">
29       skip(all)
30       <pattern>auxiliaryData</pattern>
31    </div>
32 </pattern>
33
34 <pattern data-att-met-pattern-name="auxiliaryData" data-att-met-pattern-type="auxiliary">
35    <li>
36       data-tex-mar-constraint
37    </li>
38 </pattern>
```

FIGURE 8.19: *vde* patterns specified to extract data from the page shown in Figure 8.18.

It is worth pointing out that the window in Figure 8.18 is not a modal window. Consequently, when it is shown to the user it does not block all the other workflows in the

```xml
<?xml version="1.0" ?>
<configuration_process>
    <steps>
        <step>
            <step_name>Options</step_name>
            <groups>
                <group>
                    <group_name cardinality="[0..*]">Options</group_name>
                    <features>
                        <feature>
                            <feature_name>Sport Portfolio Pack with 19" Aq</feature_name>
                            <properties>
                                <data_att_mar_full_name>Sport Portfolio Pack with 19" Aquilla</data_att_mar_full_name>
                                <data_att_mar_widget_type>checkbox</data_att_mar_widget_type>
                                <data_tex_mar_price>$8,575</data_tex_mar_price>
                            </properties>
                        </feature>
                        <feature>
                            <parent_feature>Sport Portfolio Pack with 19" Aq</parent_feature>
                            <constraints>
                                <data_tex_mar_constraint>ADDING Sport Pack with 19" Wheels</data_tex_mar_constraint>
                                <data_tex_mar_constraint>ADDING 19" Aquilla Alloy Wheels</data_tex_mar_constraint>
                                <data_tex_mar_constraint>ADDING Warm Charcoal seats with Warm Charcoal upper fascia and
                                   Dove Jaguar Suedecloth Premium Headlining
                                </data_tex_mar_constraint>
                                <data_tex_mar_constraint>ADDING Knurled Aluminium with Piano Black</data_tex_mar_constraint>
                                <data_tex_mar_constraint>REMOVING 18" Vela Alloy Wheels</data_tex_mar_constraint>
                                <data_tex_mar_constraint>REMOVING Warm Charcoal seats with Warm Charcoal upper
                                   fascia and Dove headlining
                                </data_tex_mar_constraint>
                                <data_tex_mar_constraint>REMOVING Knurled Aluminium with Satin Rosewood</data_tex_mar_constraint>
                                <data_tex_mar_constraint>REMOVING XF Interior</data_tex_mar_constraint>
                            </constraints>
                        </feature>
                        <feature>
                            <feature_name>Sport Portfolio Pack with 20" Hy</feature_name>
                            <properties>
                                <data_att_mar_full_name>Sport Portfolio Pack with 20" Hydra</data_att_mar_full_name>
                                <data_att_mar_widget_type>checkbox</data_att_mar_widget_type>
                                <data_tex_mar_price>$10,150</data_tex_mar_price>
                            </properties>
                        </feature>
                        <feature>
                            <parent_feature>Sport Portfolio Pack with 20" Hy</parent_feature>
                            <constraints>
                                <data_tex_mar_constraint>ADDING 20" Hydra Alloy Wheels</data_tex_mar_constraint>
                                <data_tex_mar_constraint>ADDING Sport Pack with 20" Wheels</data_tex_mar_constraint>
                                <data_tex_mar_constraint>ADDING Warm Charcoal seats with Warm Charcoal upper fascia
                                   and Dove Jaguar Suedecloth Premium Headlining
                                </data_tex_mar_constraint>
                                <data_tex_mar_constraint>ADDING Knurled Aluminium with Piano Black</data_tex_mar_constraint>
                                <data_tex_mar_constraint>REMOVING 18" Vela Alloy Wheels</data_tex_mar_constraint>
                                <data_tex_mar_constraint>REMOVING Warm Charcoal seats with Warm Charcoal upper
                                   fascia and Dove headlining
                                </data_tex_mar_constraint>
                                <data_tex_mar_constraint>REMOVING Knurled Aluminium with Satin Rosewood</data_tex_mar_constraint>
                                <data_tex_mar_constraint>REMOVING XF Interior</data_tex_mar_constraint>
                            </constraints>
                        </feature>
                        ...
```

FIGURE 8.20: The output XML file produced for the page shown in Figure 8.18 and the patterns given in 8.19.

application. It means that the Wrapper and the Crawler can progress with the extraction process without waiting for the user to interact with the window and close it. However, it may be the case that a modal window is used, thus preventing the extraction process. In these cases, the window can be programmatically closed. If it can not, the user has to manually close the window to resume the extraction process.

8.3.3.4 Deducing cross-cutting constraints from configuration state changes

In some cases, when an option is configured and one or more cross-cutting constraints apply, the configurator automatically propagates the required changes to all the impacted options in the page and alters their configuration states (*automatic* decision propagation strategy). Since an option is represented using a widget, when its configuration state is changed, the user-interface state (selected, deselected, unavailable, etc.) of the corresponding widget will be changed accordingly. As a consequence, by analysing the

user-interface-state changes of widgets representing options we can find out which options are impacted and how, and then we will be able to deduce the applied cross-cutting constraints.

Usually an attribute of the widget (e.g., the *checked* attribute in a check box or the *selected* attribute in a list box) or another HTML element in the code fragment representing an option denotes the state of the option. By documenting and then analysing the changes made for this *state attribute* during a crawling process we can deduce cross-cutting constraints.

The extraction of constraints from configuration state changes requires addressing two challenges. First, the set of configurations must be recorded. A configuration presents options and their states. Second, the state changes made to options in different configurations should be studied to infer the cross-cutting constraints.

To deduce all cross-cutting constraints, we must collect all valid configurations, and then use a feature model synthesis algorithm (e.g., [ACLF13, HLHE13]) to infer the constraints. Collecting all valid configurations requires that the Crawler navigates the whole configuration space, investigates all possible option combinations, and records the state of all options. Due to the following limitations of our approach, it can not collect all valid configurations:

- Our approach does not provide support for automatically crawling the Web pages of configurators that follow the multi-page paradigm. Consequently, it can not identify constraints across pages.
- For options included in a Web page, the Crawler selects/deselects them one by one. It can not simulate different combinations of options. It means that, the Crawler is only able to collect a subset of all valid configurations.

We believe that it is a hard problem to develop a generic approach to gather all valid configurations from the client-side of Web configurators because of variations in their implementation. In this PhD, we tackle a simple case of the problem in which: there are no dependencies between options included in the target page and those that are included in the other pages of the configurator. We now explain our solutions for collecting the set of configurations and inferring the cross-cutting constraints.

Producing the set of configurations. To systematically generate the configuration state changes of options in a page, we again rely on the dependency between *vde* patterns. An independent pattern denotes options to be automatically configured and one (or more) dependent pattern indicates those options whose state changes should be documented. The independent and dependent patterns may point to the same set of options in the page.

When the configuration state of an option is changed by the Crawler and the changes made to the states of other options are documented by the Wrapper, the Crawler may either keep the configuration of the option in the new state or may change it back to the previous state. The `data-att-met-reset-state = "false"` in the independent data pattern tells the Crawler to follow the former scenario. This is a way to supervise the crawling strategy.

Given the independent and dependent patterns, the crawling process is implemented using the algorithm shown in Figure 8.5. We recall the main steps in Figure 8.21 for generating the set of configurations. For convenience, we use one independent and one dependent patterns to describe the algorithm. Before starting the crawling process, the current configuration state of all options (all options denoted by the independent and dependent patterns), called *index configuration state*, is extracted and documented by the Wrapper (lines 2 and 3). *independentPattern* and *resetState* that are respectively the independent region pattern and the value of the `data-att-met-reset-state` attribute are taken (lines 4 and 5). Then, for each option in the region defined by the independent region pattern (lines 6-12), the Crawler changes the configuration state of the option (line 7) and the Wrapper extracts the current configuration state of all the options (including options in both the independent and the dependent regions – line 8). The extracted data is added to the output XML file (line 9). Note that the option clicked by the Crawler (line 7) is also documented in the XML file. If `data-att-met-reset-state = "true"` (line 10), the Crawler undoes the change made to the option (line 11) and takes the next option into consideration (line 6).

Example 8.7: Figure 8.22 displays an excerpt of a configuration environment in which options are represented using check boxes. In Figure 8.23, the index configuration states of these options are given (in the `data_att_mar_configuration_state` elements). The "Emergency tyre inflation kit" option is checked and disabled. The "Driver's seat belt warning

```
1 generateConfigurations (configFile, pageSource) {
2   indexConfigurationState = getConfigurationState (configFile, pageSource)
3   addToExtractedData (indexConfigurationState)
4   independentPattern = getIndependentPattern (configFile)
5   resetState  = getResetStateAttributeValue (independentPattern)
6   for each (option in independentPattern) {
7     changeConfigurationState (option, pageSource)
8     currentConfigurationState = getConfigurationState (configFile, pageSource)
9     addToExtractedData (currentConfigurationState)
10    if (resetState)
11      changeConfigurationState (option, pageSource)
12  }
13 }
```

FIGURE 8.21: Algorithm for generating the configuration set.

seat head restraints" option is disabled. Other options are undecided. Figure 8.24 presents the *vde* patterns specified to crawl and to extract options (`data-tex-mar-option-name` – lines 15 and 32) and their configuration states (`data-att-mar-configuration-state` which records the value of the `class` attribute of the `span` element in the code fragment representing an option – lines 14 and 31). Since to define the dependency between patterns at least two patterns should be involved, the *independentOptionsRegion* (lines 1-8) and *dependentOptionsRegion* (lines 19-26) region patterns are defined, but both patterns denote the same region of the page (lines 3 and 21). The Crawler uses the *independentData* pattern (lines 10-17) to identify options to be automatically configured and the Wrapper uses the *dependentData* pattern (lines 28-34) to identify options to be extracted. Again note that both data patterns point to the same set of options because their corresponding region patterns point to the same region of the page. In Figure 8.25, an excerpt of the output XML file is shown. The text value of all the `parent_feature` elements is "Space-saver spare wheel". It means that the selection of the "Space-saver spare wheel" option ("ui-checkbox-state-checked" – line 13) generated the configuration.

Analysing configuration state changes. The algorithm presented in Figure 8.21 generates the set of configurations. This set is then analysed to deduce cross-cutting constraints. Figure 8.26 shows the algorithm (the **deduceConstraints** procedure) we proposed to detect constraints from a given set of configurations. Since the state changes of an option in all the recorded configurations should be studied, the instances of all the options in all the configurations are taken (line 2). Then, for each option in the list (lines 3-22), its state in the previous configuration (line 4) and in the current configuration (line 5) are extracted. In addition, the option whose selection/deselection generated

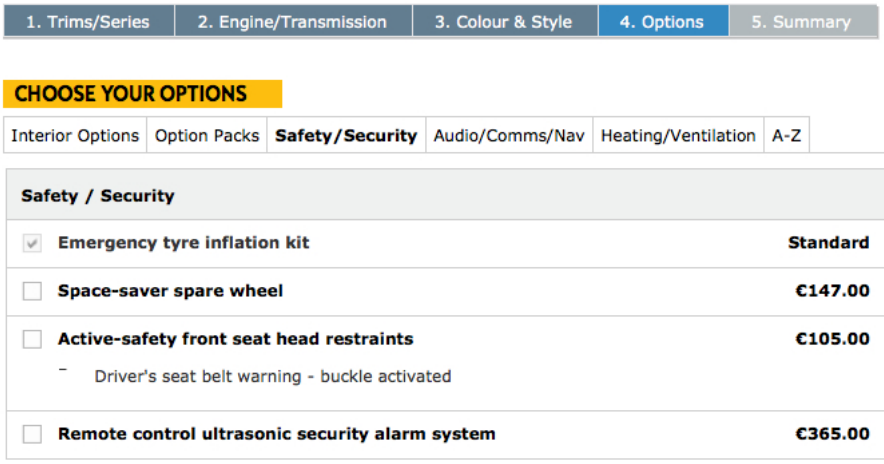


FIGURE 8.22: An example configuration environment (`http://www.opel.ie/`, August 3 2013).

```
1  <features>
2      <feature>
3          <feature_name>Emergency tyre inflation kit</feature_name>
4          <properties>
5              <data_att_mar_configuration_state>ui-checkbox-state-checked-disabled</data_att_mar_configuration_state>
6          </properties>
7      </feature>
8      <feature>
9          <feature_name>Space-saver spare wheel</feature_name>
10         <properties>
11             <data_att_mar_configuration_state>ui-checkbox</data_att_mar_configuration_state>
12         </properties>
13     </feature>
14     <feature>
15         <feature_name>Active-safety front seat head restraints</feature_name>
16         <properties>
17             <data_att_mar_configuration_state>ui-checkbox</data_att_mar_configuration_state>
18         </properties>
19     </feature>
20     <feature>
21         <feature_name>Driver's seat belt warning - buckle activated</feature_name>
22         <properties>
23             <data_att_mar_configuration_state>ui-checkbox-state-disabled</data_att_mar_configuration_state>
24         </properties>
25     </feature>
26     <feature>
27         <feature_name>Remote control ultrasonic security alarm system</feature_name>
28         <properties>
29             <data_att_mar_configuration_state>ui-checkbox</data_att_mar_configuration_state>
30         </properties>
31     </feature>
32 </features>
```

FIGURE 8.23: Index configuration state for the options shown in Figure 8.22.

the current configuration (line 6) and its state in the current configuration (line 7) are identified. The configured option is highlighted by the `parent_feature` element in the extracted data (Figure 8.25).

Considering the condition that the previous and current states of the target option are respectively *checked* and *undecided* (line 8), if the current state of the configured option is *checked* (line 9) then the configured option *excludes* the target option (line 10), otherwise the configured option *requires* the option (line 12).

If the previous and current states of the target option are respectively *undecided* and *checked* (line 15), and if the current state of the configured option is *checked* (line 16)

```
<pattern data-att-met-pattern-type="region" data-att-met-pattern-name="independentOptionsRegion"
         data-att-met-root-pattern="true"   data-att-met-dependent-pattern="dependentOptionsRegion">
   <fieldset class="safety">
      <pattern>
         independentData
      </pattern>
   </fieldset>
</pattern>

<pattern data-att-met-pattern-name="independentData" data-att-met-pattern-type="data">
   <li>
      <input type="checkbox"            data-att-met-clickable="true"
             data-att-met-unique="true" data-att-met-reset-state="true"/>
      <span  data-att-mar-configuration-state="@class"></span>
      <label>data-tex-mar-option-name</label>
   </li>
</pattern>

<pattern data-att-met-pattern-type="region" data-att-met-pattern-name="dependentOptionsRegion"
         data-att-met-root-pattern="true">
   <fieldset class="safety">
      <pattern>
         dependentData
      </pattern>
   </fieldset>
</pattern>

<pattern data-att-met-pattern-name="dependentData" data-att-met-pattern-type="data">
   <li>
      <input type="checkbox" data-att-met-unique="true"/>
      <span data-att-mar-selection-status="@class"></span>
      <label>data-tex-mar-option-name</label>
   </li>
</pattern>
```

FIGURE 8.24: *vde* patterns specified to crawl the options shown in Figure 8.22.

```
<features>
    <feature>
        <feature_name>Emergency tyre inflation kit</feature_name>
        <parent_feature>Space-saver spare wheel</parent_feature>
        <properties>
            <data_att_mar_selection_status>ui-checkbox-state-disabled</data_att_mar_selection_status>
        </properties>
    </feature>
    <feature>
        <feature_name>Space-saver spare wheel</feature_name>
        <parent_feature>Space-saver spare wheel</parent_feature>
        <properties>
            <data_att_mar_selection_status>ui-checkbox-state-checked</data_att_mar_selection_status>
        </properties>
    </feature>
    <feature>
        <feature_name>Active-safety front seat head restraints</feature_name>
        <parent_feature>Space-saver spare wheel</parent_feature>
        <properties>
            <data_att_mar_selection_status>ui-checkbox</data_att_mar_selection_status>
        </properties>
    </feature>
    <feature>
        <feature_name>Driver's seat belt warning - buckle activated</feature_name>
        <parent_feature>Space-saver spare wheel</parent_feature>
        <properties>
            <data_att_mar_selection_status>ui-checkbox-state-disabled</data_att_mar_selection_status>
        </properties>
    </feature>
    <feature>
        <feature_name>Remote control ultrasonic security alarm system</feature_name>
        <parent_feature>Space-saver spare wheel</parent_feature>
        <properties>
            <data_att_mar_selection_status>ui-checkbox</data_att_mar_selection_status>
        </properties>
    </feature>
</features>
```

FIGURE 8.25: The output XML file – the option "Space-saver spare wheel" is selected by the Crawler in Figure 8.22.

then the configured option *requires* the target option (line 17). Otherwise, the configured option *excludes* the target option (line 19).

Our constraint synthesis algorithm can correctly detect constraints of the forms:

- f *requires* F
- f *excludes* F

in which f is an option and F is a set of options. This algorithm will detect false positives of the aforementioned forms. For example, it will detect "f_a *requires* f_b", while the true constraint is "f_a *AND* f_x *requires* f_b".

```
1 deduceConstraints (Configurations) {
2   optionList = getOptionList (Configurations)
3   for each (option in optionList) {
4       optionPreviousState  = getOptionPreviousState (option, Configurations)
5       optionCurrentState  = getOptionCurrentState (option, Configurations)
6       configuredOption  = getConfiguredOption (option, Configurations)
7       configuredOptionCurrentState  = getOptionCurrentState (configuredOption, Configurations)
8       if  ((optionPreviousState == "checked")  and  (optionCurrentState == "undecided")) {
9           if  (configuredOptionCurrentState == "checked") {
10              addToExtractedData (configuredOption excluded  option)
11          } else {
12              addToExtractedData (configuredOption requires  option)
13          }
14      }
15      if  ((optionPreviousState == "undecided")  and  (optionCurrentState == "checked")) {
16          if  (configuredOptionCurrentState == "checked") {
17              addToExtractedData (configuredOption requires  option)
18          } else {
19              addToExtractedData (configuredOption excludes  option)
20          }
21      }
22  }
23 }
```

FIGURE 8.26: Algorithm for deducing constraints from the state changes.

In Figure 8.25, by comparing the current state of each option with its previous state (i.e., its index configuration state – Figure 8.23), the **deduceConstraints** procedure detected that the configuration state of the "Emergency tyre inflation kit" is changed from the *checked* ("ui-checkbox-state-checked-disabled" – line 5 in Figure 8.23) to the *undecided* ("ui-checkbox-state-disabled" – line 6 in Figure 8.25). Since the current state of the configured option, i.e., "Space-saver spare wheel", is *checked* (line 13 in Figure 8.25), there is an *exclusion* cross-cutting constraint between the "Space-saver spare wheel" and the "Emergency tyre inflation kit" options. The states of other options remained unchanged.

8.4 Chapter Summary

In this chapter, we introduced the notion of dependency between patterns to formulate the logical relationship between objects in a Web page. Based on this formulation,

we then presented our solution to crawl the configuration space, i.e., automatic exploration and configuration of options in a page: the independent pattern denotes clickable Web objects (e.g., widgets representing options, configuration steps) to be automatically clicked by the Crawler, and the dependent pattern indicates data objects (e.g., options dynamically loaded in the page as the result of the selection of an option, options made available as the result of activating a configuration step) to be extracted by the Wrapper.

We also described our proposed methods to extract formatting, group, and cross-cutting constraints. Using the dependency between patterns, we are able to trigger and then extract cross-cutting constraints defined over options.

Chapter 9

Evaluation

In this chapter, we present the experiment we set up to evaluate the proposed reverse-engineering process. We describe our evaluation model, i.e., goals, questions, and metrics (Section 9.1) and report on our experiment and the results (Section 9.2). We then discuss the results and present the qualitative observations (Section 9.3), and finally explain the threats to validity (Section 9.4).

9.1 Experimental Setup

Goal and scope. We aim to evaluate the application of our approach to reverse engineer feature models from Web configurators. The first criterion to be examined is the *accuracy* of the extracted data, i.e., the extracted data is the *right* data and the reverse-engineered models are *complete* models. We neither have base models to which we could compare our generated models nor have access to the developers of the studied configurators who can validate our models. Therefore, we have not been able to compute and report on *recall* to indicate which fraction of all configuration-specific objects has been recognized. As to the *precision*, our goal is to specify a minimum set of patterns to extract all options (100% coverage, if possible) presented in the page. For automatic constraint extraction, we plan to apply the methods presented in Section 8.3 and assess the correctness of the extracted constraints.

The second criterion to be evaluated is the *scalability* of our approach. By scalability we mean (1) the *expressiveness* of the *vde* patterns to deal with variations in presentation and implementation of configuration-specific entities in Web pages of different configurators, and (2) the *ability* of our crawling approach to extract dynamic data.

Since our approach is supervised and semi-automatic, the third criterion to be measured is the users' *manual effort* required to perform the extraction. It should be noted that the usability of the tool was not a high-priority requirement as this is a research prototype.

Questions and metrics. We address the following main questions (Q):

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

The underlying metrics (M) are formulated as follows:

- *M1.* The number of patterns required to extract data.
- *M2.* The number of pattern dependencies specified in order to either crawl the configuration space or document relationships between objects.
- *M3.* The number of lines of code (LOC) of all patterns written to extract data.
- *M4.* The number of times the data extraction procedure is executed.
- *M5.* The number of automatically extracted options.
- *M6.* The number of object not presented in the page but identified and extracted by the crawling technique (dynamic data objects).
- *M7.* The number of manually added options.
- *M8.* The number of constraints automatically identified and extracted.
- *M9.* The number of constraints manually added.

TABLE 9.1: Questions and Metrics

Questions	Q1	Q2	Q3	Q4
Metrics	M5, M8	M1, M5	M2, M6, M8	M1, M3, M4, M7, M9, M10

TABLE 9.2: Example Web configurators chosen for evaluation.

Name	**URL**	**System**
Dell's laptop configurator	http://www.dell.com	S1
BMW's car configurator	http://www.bmwusa.com	S2
Dog-tag generator	http://www.mydogtag.com	S3
Chocolate maker	http://www.choccreate.com	S4
Shirt designer	http://www.shirtsmyway.com	S5

- *M10.* The manual activities performed to organize and refine the extracted data, add the missing data, etc.

Table 9.1 presents the questions and their associated metrics.

If a pattern is used in two or more configuration files, we reported it once. We also consider two patterns similar if and only if their structures exactly match. To compare the structure of two patterns, we take into account their tag names, tag positions, structural text elements, and structural attributes.

To count the number of lines of code for each pattern, we put one and only one element in each line. For example, the following pattern has eight lines of code.

```
<pattern data-att-met-pattern-type="data"  data-att-met-pattern-name="option">
  <div>
    skip(all)
    <div>
      data-tex-mar-option-name
    </div>
  </div>
</pattern>
```

Data set. We took the five example configurators S1-S5 listed in Table 9.2, chosen from the sample set of configurators we know from our empirical study (Chapter 3). S1 is Dell's laptop configurator. We took the *"Inspiration 15"* model in this experiment. S2 is the car configurator of BMW. For this study, we chose the *"2013 128i Coupe"* model. S3 is a dog-tag generator. In S4 the customer can choose her chocolate and create its masterpiece and ingredients. S5 is a configurator that allows customers to design their shirts.

Execution. The author of the thesis supervised the reverse-engineering process. For each Web page of the target configurator, we first inspected its source code using the Firebug's *HTML panel* to find out which templates are used to generate the page and then specified the required patterns with respect to these templates to extract data objects of interest. Our goal was to specify a minimum set of patterns to extract all options (100% coverage, if possible) presented in the page. For each option, we extracted its name, widget type, image source (for image options), and other attached descriptive information (e.g., price). After running the Wrapper for the given patterns, we manually compared the extracted data with that presented in the page to find out the missing or noisy data. We either altered the existing patterns or specified new ones to achieve 100% coverage for the extracted options.

Using the Web Crawler we simulated the click event on every option to recognize whether it creates and adds new data objects to the page or triggers a constraint. If yes, we then applied the crawling approach to extract this data. To know whether a change has been made to the page, we used the *Firediff*[1] add-on. Firediff is an extension to track changes in Firebug. It implements a change monitor and records all of the changes made by Firebug and the application itself to CSS and the DOM.

If all the extracted options from a group are presented through images, the Wrapper considers them in an alternative group and extracts an alternative constraint for these grouped options. If all the extracted options from a group are represented using check boxes, they implement a multiple choice group and therefore the Wrapper reports for them a multiple choice constraint. We observed that image options that are visually grouped together may implement a multiple choice group. In very rare cases we also observed that some exclusive options are implemented by (non exclusive) check boxes. Therefore, for these two cases, after extracting group constraints by the Wrapper, we also manually checked them to ensure that the identified group constraints are true.

To denote the target region within which the Wrapper and the Crawler should work (the specification of the region pattern) we tried to find an HTML element that points exactly to the region we need. If we could not find such element, we either edited the attribute value of an existing element, added a new attribute, or added a new indicator HTML element to the page.

[1] https://addons.mozilla.org/En-us/firefox/addon/firediff/

9.2 Experiment and results

We now report on our experience and results for each configurator. The tools and the complete set of data are available at `http://info.fundp.ac.be/~eab/result.html`.

Table 9.3 presents the experimental data, Table 9.4 displays different pattern-specific elements we used in the pattern specifications, and in Table 9.5 the number of lines of code of the generated TVL files for each configurator is given.

S1: Dell's laptop configurator. In S1 (Figure 9.1), options are presented using radio buttons and check boxes. We specified only one data pattern and one region pattern for extracting all options ($M1$). S1 provides a four-step configuration process such that by activating each step the page is reloaded so as to contain the step's options. So, we had to manually activate each step and then run the extraction procedure for that. It explains why we did not specify a dependency (between an independent pattern that denotes the step names and the dependent one that indicates options) and we did not use the crawling technique to extract all options in one execution, instead of four ($M4 = 4$). Given the specified patterns, the Wrapper could extract all 233 options ($M5$) presented in the pages as well as group names for options categorized in alternative and multiple choice groups. To categorize options extracted from each step in a group, we manually added three group names ($M7$) to the extracted data. Note that we also count group names as options in this experiment.

The Wrapper could correctly identify and extract 49 option groups and group constraints defined on these groups ($M8$). In TVL, a group is represented with a parent feature (i.e., the group name), its decomposition or constraint type (i.e., *and*-, *xor*-, *or*-decompositions, or a cardinality), and its sub-features (i.e., options included in the group). For 46 of the identified groups, the Wrapper could not extract a group name (because either it is not specified in the data pattern to extract group names or there is no explicit group name specified in the page for some groups). However, we noticed that options grouped together have the same value for the `name` attribute of their widget elements. Therefore, we extracted the value of this attribute for all options and when transforming from the XML file to TVL, the transformation module used these values to create and label the parent feature of the grouped sub-features. Values of the `name` attributes are not meaningful identifiers in S1, they are constituted of abbreviations, numbers, and symbols. To give meaningful names for automatically added parent features, we manually renamed these feature names in the generated TVL files ($M10$).

In S1, the `table` element is used as the basic building block for the page layout. In this implementation, we could not recognize a clear template from which an option and its sub-options are generated. Consequently, we could not extract and document the hierarchical relationships between options. To build a feature hierarchy in TVL that reflects the hierarchy of options

TABLE 9.3: Experimental results.

PATTERN SPECIFICATION					
System	**Pattern** (M1)	**Dependency** (M2)	**LOC** (M3)	**Executions** (M4)	**Manual Work** (M10)
S1	Data 1 Region 1		24	4	Rename 46 Replace 12
S2	Data 5 Region 3	5	90	14	Highlight 10
S3	Data 5 Region 4	2	82	8	Highlight 7 Remove 126
S4	Data 2 Region 2	1	40	2	
S5	Data 6 Region 3		86	11	Rename 5 Remove 8
Total	Data 19 Region 13	8	322	39	Rename 51 Remove 134 Highlight 17 Replace 12
DATA					
System	**Options** (M5)	**Dynamic objects** (M6)	**Manual objects** (M7)	**Constraints** (M8)	**Manual constraints** (M9)
S1	233		3	Group 49	Group 12
S2	97		7	Group 7 False group 1 Cross-cutting 30	Group 4
S3	137	Option 44 Data 19	8	Group 14 Formatting 10	Formatting 1
S4	24	95	2	Group 7	
S5	233 False positive 6 Redundant 2		Group name 11 Text input 3	Group 26 Formatting 40	
Total	True 724 False positive 6 Redundant 2	158	34	Group 103 Cross-cutting 30 Formatting 50 False group 1	Group 16 Formatting 1
PRECISION					
Precision	Option t = 863 N = 905	P $\simeq$ 95%		Constraint t = 183 N = 201	P $\simeq$ 91%
AVERAGE (DYNAMIC OBJECTS)					
Average	Option d = 158 N = 882	$A_{dynamic} \simeq 18\%$		Constraint d = 30 N = 183	$A_{dynamic} \simeq 16\%$
AVERAGE (MANUAL OBJECTS)					
Average	Option m = 34 N = 916	$A_{manual} \simeq 4\%$		Constraint m = 17 N = 200	$A_{manual} \simeq 9\%$

presented in the Web pages of S1, we manually replaced 12 feature and group blocks in the generated TVL files ($M10$) and added 12 decomposition types ($M9$).

TABLE 9.4: Pattern-specific elements.

System	skip(all)	skip(STRING)	Multiplicity	Structural Attribute	Or Operator	Wildcard
S1	✓		✓	✓	✓	
S2			✓	✓		✓
S3	✓	✓		✓		✓
S4		✓		✓	✓	✓
S5	✓	✓	✓	✓		✓
Total	3	3	3	5	2	4

TABLE 9.5: LOC of the generated TVL files.

System	LOC
S1	1212
S2	428
S3	1127
S4	598
S5	1113
Total	4478

S2: BMW's car configurator. S2 presents options using images and check boxes. We specified a data pattern to extract image options and one for those represented using check boxes. For some check box options, there is a list of attached sub-options, which are either presented using check boxes or labels (Figure 9.2). We defined two data patterns to extract these sub-options. When an option is given a new value and one or more cross-cutting constraints apply, the configurator asks the user to confirm or discard the decision before propagating the required changes to all the impacted options (*controlled* decision propagation strategy – Figure 9.3). It presents a *conflict window* and lists the names of the impacted options. The content of this window is generated using a template. We therefore specified a data pattern to extract the content of this window. Overall we defined five data patterns to extract data from S2. We also defined three region patterns to point to the regions we needed ($M1$).

We specified five dependencies between patterns ($M2$). Two dependencies are defined between the patterns that denote options and the pattern that indicates the conflict window (Figure 9.3). These dependencies are used to crawl the configuration space in order to trigger and extract cross-cutting constraints. Two other dependencies are specified to document the parent-child relationships between the parent options (i.e., options have a list of attached sub-options) and their sub-options (Figure 9.2). One dependency is also defined to produce and record the set of configurations in a step (which is used to deduce cross-cutting constraints – Figure 9.4).

We could extract all 97 options presented in the pages of S2 ($M5$) using the specified patterns. We also manually added seven options to the extracted data ($M7$). These are group names used to categorize the extracted options. In some cases, we had to edit an element in the page

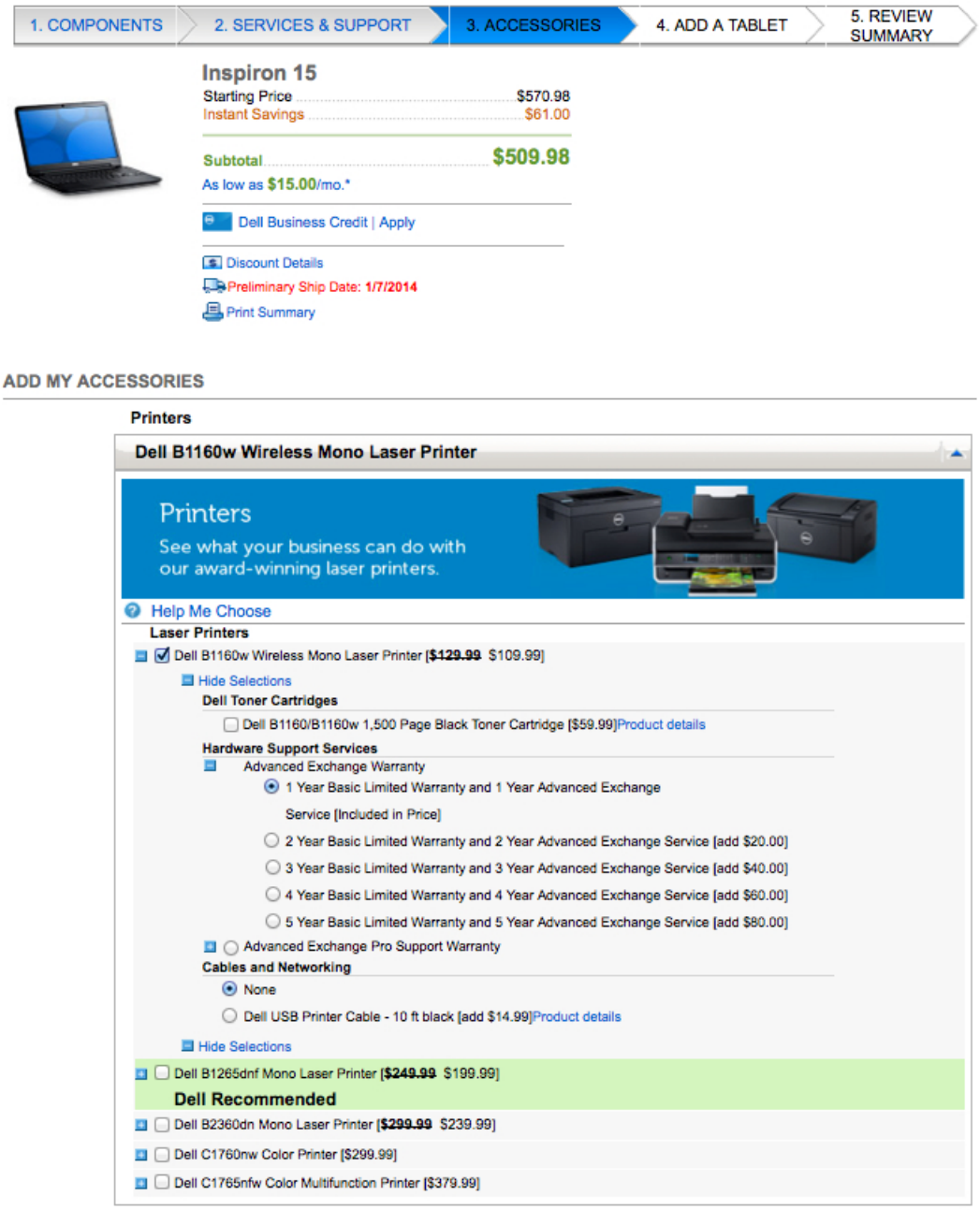


FIGURE 9.1: Dell's laptop configurator (`http://www.dell.com`, January 5 2014).

to highlight the portion of the page within which the Wrapper and the Crawler should operate ($M10$). This element is used in the specification of the region patterns.

As to group constraints, the Wrapper identified and extracted seven group constraints ($M8$) and we manually added four more ($M9$). One of the group constraints reported by the Wrapper was incorrect. In this case, all the options extracted from a group were image options. Therefore, the Wrapper considered the group as an alternative group. However, manual testing of these options revealed that they semantically implement two different alternative groups (Figure 9.3).

To trigger and extract cross-cutting constraints, we used three different strategies depending on their implementations in S2. First, in some cases, by configuring an option a conflict window is presented showing which options are impacted and how (Figure 9.3). Using the Crawler, we simulated configuration actions on options and if the conflict window was displayed, then the Wrapper extracted its content and analysed the content to deduce the cross-cutting constraints. Second, in one case, by selecting an option in a step (i.e., the "Packages" step – Figure 9.4) its implied options are selected in the following step (i.e., the "Options" step). For this case, we selected each option from the "Packages" step and recorded the selected options in the "Options" step. When generating the TVL model, the transformation module documented them as *requires* constraints. The third strategy we used to identify the cross-cutting constraints in S2 is the analysis of the configuration state changes (Figure 9.4). We recorded all configurations for a step and applied our algorithm discussed in Section 8.3.3.4 to deduce constraints from this set of configurations. Using these three strategies, we could identify and extract 30 cross-cutting constraints ($M8$).

We executed the data extraction procedure 14 times ($M4$) in total to extract options and constraints from S2.

S3: Dog-tag generator. Options in S3 are presented using either text boxes or radio buttons (some combined with images – Figures 9.5 and 9.6). We specified four regions patterns and five data patterns ($M1$) and executed them eight times ($M4$) to extract options. We specified one data pattern for text options. We also identified three different templates from which radio button options are generated. Therefore, we had to specify three data patterns to extract these options. The reason for using different templates for radio button options is that they present different attached descriptive information (Figures 9.5 and 9.6). For some options, only the option's short name is presented and when the user clicks on the option, its full name, size, and price are dynamically added to the page (Figures 9.6 – *Step 6*). We so defined a data pattern to denote these dynamically-generated data objects.

The Crawler detected two cases that require the crawling scenario to extract dynamic data objects. In one case, the selection of an option loads its implied options in the page and hides the irrelevant options (Figures 9.6 – *Step 5*). We defined a dependency between the corresponding patterns ($M2$) to dynamically generate and extract these data objects by the Crawler. The Crawler identified 44 dynamically-generated options for this case ($M6$). In the second case, by clicking an option its additional descriptive information (i.e., full name, size, and price) is dynamically added to the page (Figures 9.6 – *Step 6*). We defined another dependency ($M2$) here to extract this data. The Crawler collected 19 data objects for this case ($M6$). We extracted 181 options (137 options presented in the page plus 44 dynamically-generated options) from S3 and manually added eight group names ($M7$) to the extracted data.

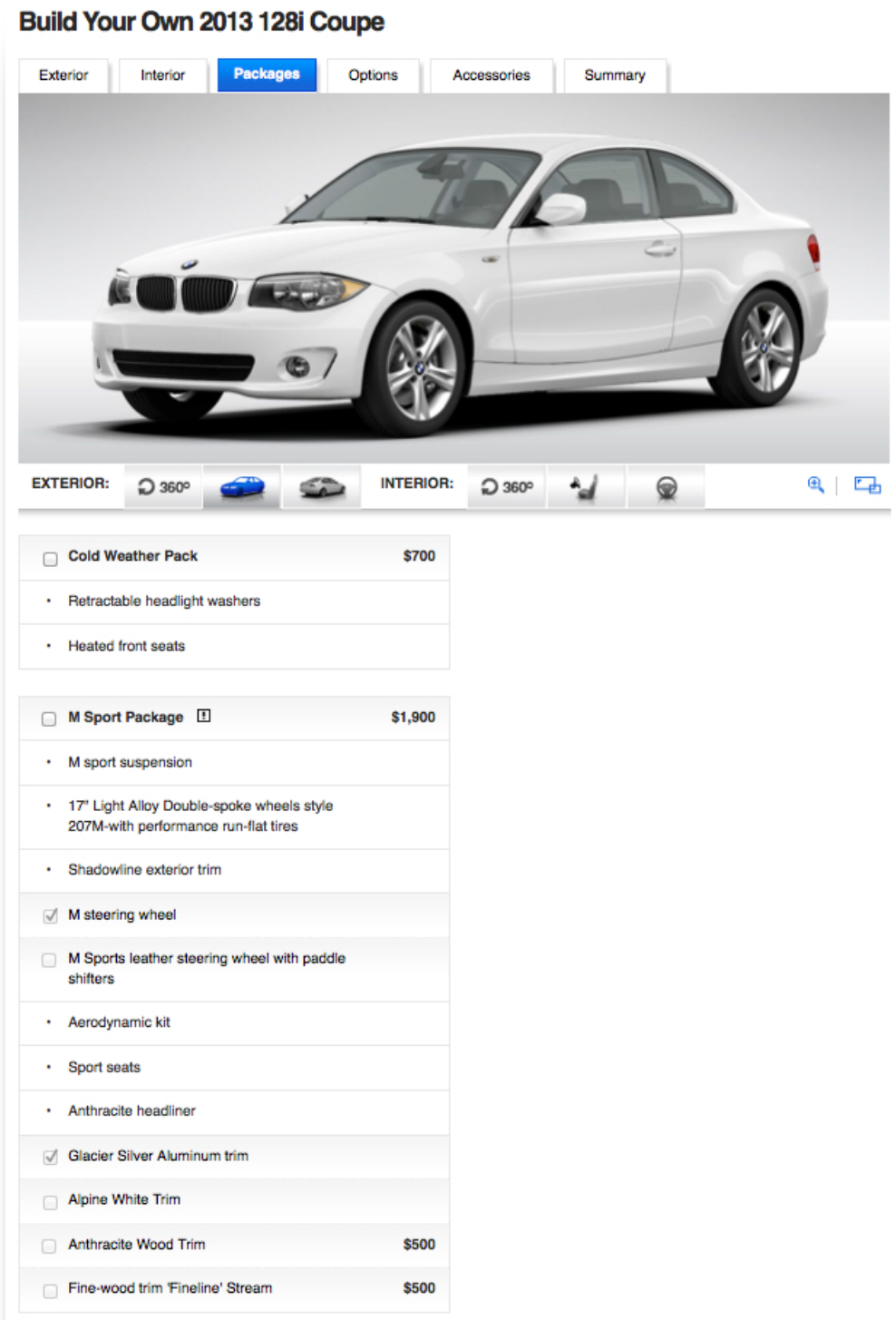


FIGURE 9.2: BMW's car configurator (`http://www.bmwusa.com`, January 5 2014).

The Wrapper also extracted 14 group constrains and ten formatting constraints ($M8$). These formatting constraints control the maximum length of strings that the user can enter in text options. We also manually added a formatting constraint to the extracted data. This constraint specifies the set of allowed characters that the user can use to enter strings in text options.

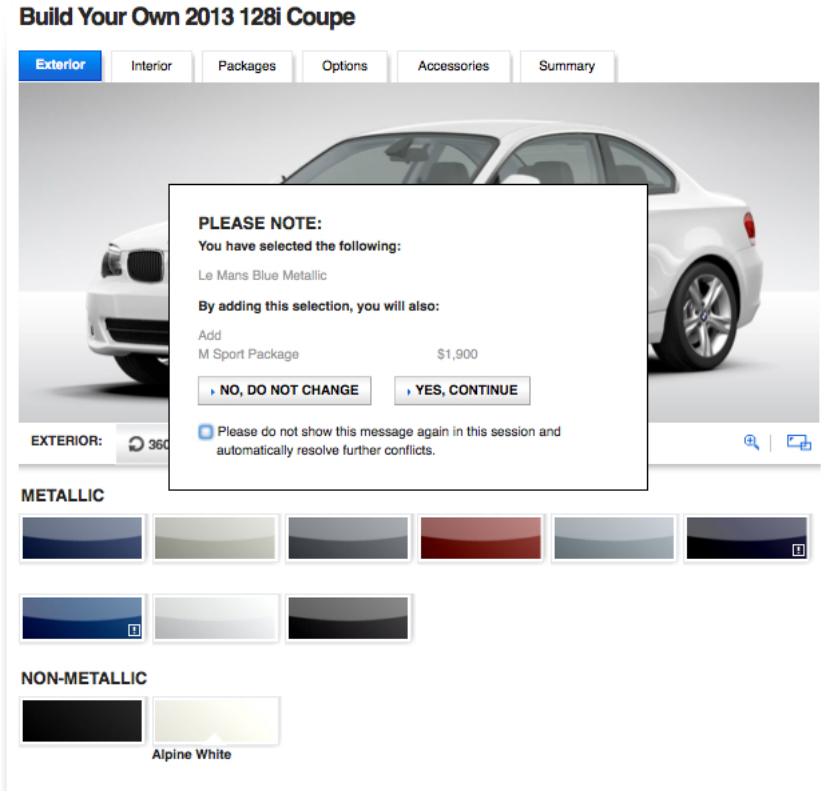


FIGURE 9.3: *controlled* decision propagation strategy in BMW's car configurator (`http://www.bmwusa.com`, January 5 2014).

We could define a pattern to extract this data object, but manually adding this object in the extracted data is much quicker than writing new code.

In S3, some radio buttons representing options are combined with images (Figures 9.5 and 9.6). For these options we also needed to extract the image's URL. We noticed that for these cases, the URL of the image is located in the JavaScript code of the `onclick` event handler of the <`input type="radio"`> element. Therefore, to extract the URLs, we extracted the JavaScript code of these radio button options (126 options in total) and then manually removed the noisy lines of code ($M10$). We then kept the line that presents the URL.

To denote regions that are used in the specification of region patterns, we also had to manually edit attributes of seven HTML elements in the page ($M10$).

S4: Chocolate maker. S4 provides a two-step configuration process (Figure 9.7). In the first step, the customer chooses her chocolate type and then in the second step, she selects ingredients. The chocolate types are presented using radio buttons and ingredients are check box options. We specified a data pattern to extract both radio button and check box options. The ingredients are also categorized into a set of groups, each of which is presented with a menu. We specified

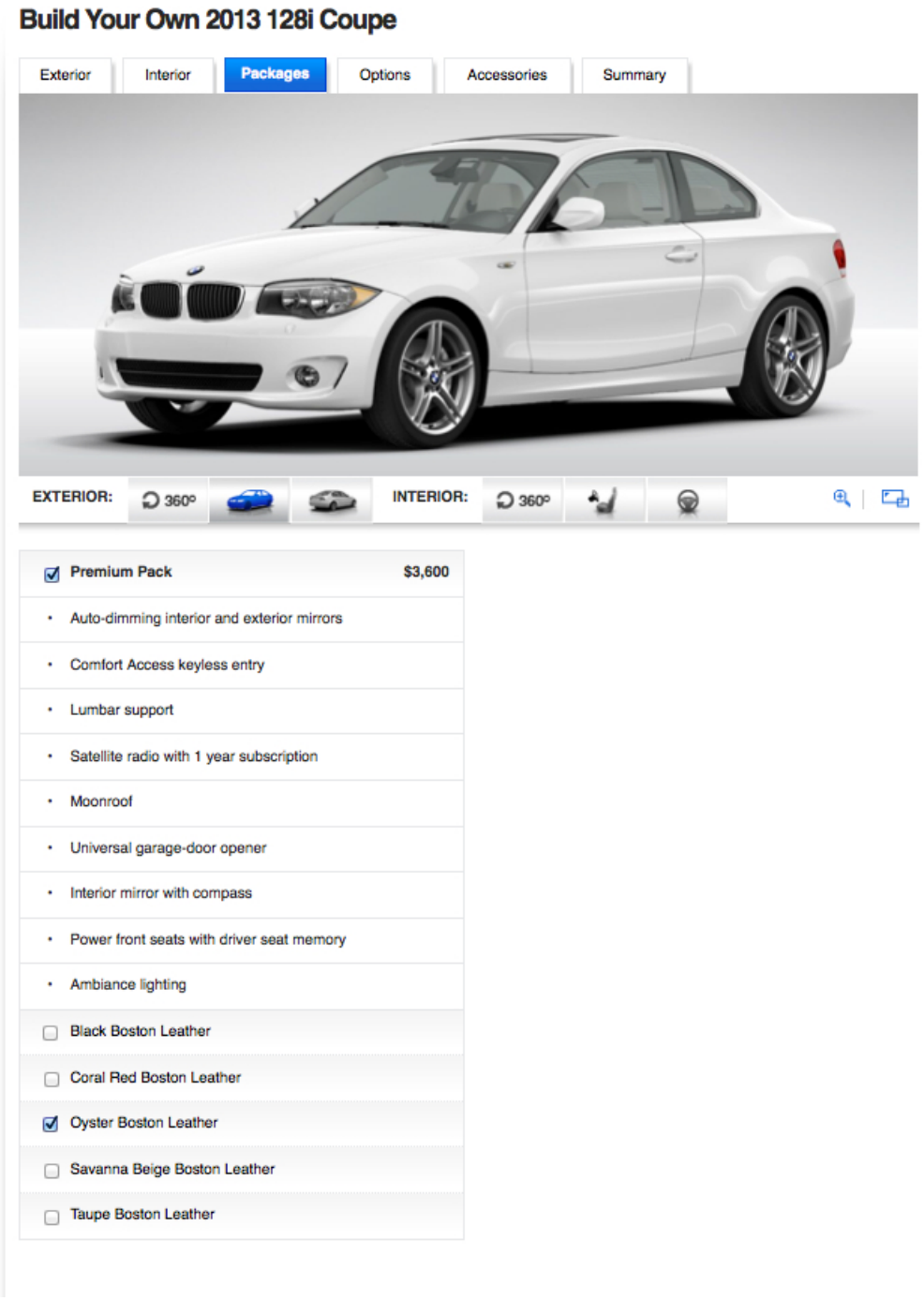


FIGURE 9.4: Configuration state changes in BMW's car configurator (`http://www.bmwusa.com`, January 5 2014).

another data pattern to extract the name of these groups. We also specified two region patterns ($M1$).

When the page is initially loaded, the radio button options (three objects), the group names (six objects), and the ingredients of one group (15 objects) are presented to the customer. Given the specified patterns, the Wrapper could extract all these 24 options ($M5$). By activating a group

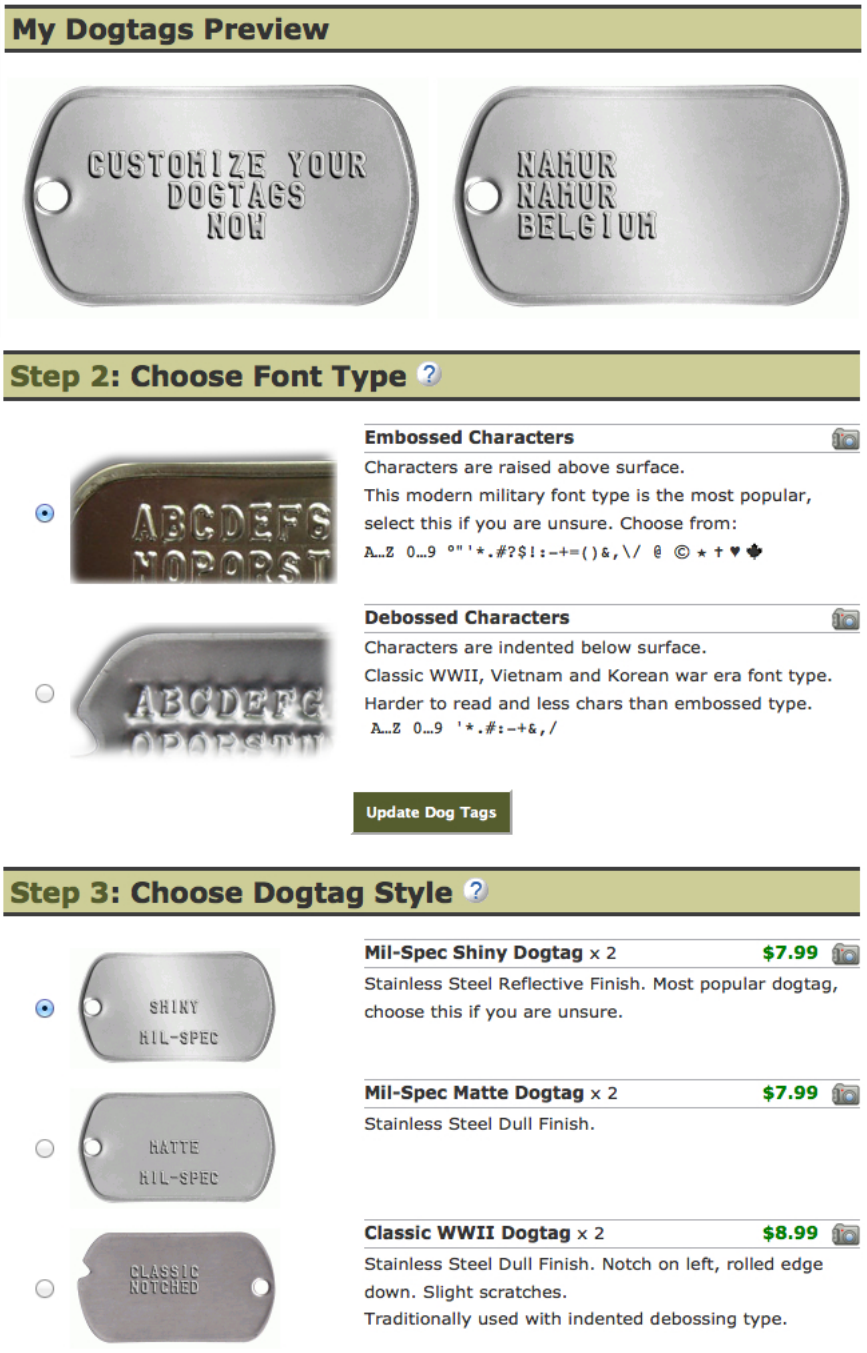


FIGURE 9.5: Dog-tag generator (`http://www.mydogtag.com`, January 5 2014).

(i.e., activating its corresponding menu) its contained ingredients are presented to the customer and other groups become hidden. We defined a dependency ($M2$) between the pattern that denotes the groups and the one that indicates the ingredients (check box options) and ran the Crawler. The Crawler detected 95 dynamically-generated options ($M6$) which are then extracted by the Wrapper. By running the Wrapper and the Crawler twice ($M4$), we extracted 119 options from S4 ($24 + 95$). In addition, the Wrapper identified seven group constraints ($M8$). We also

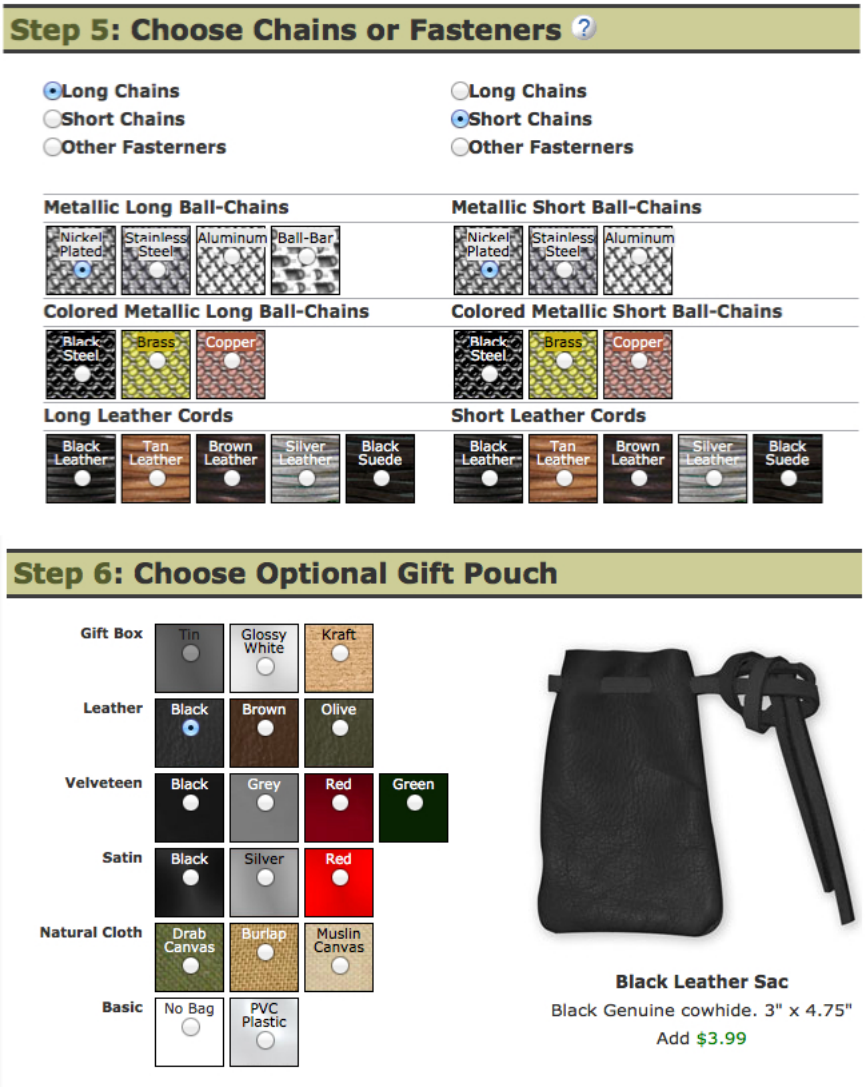


FIGURE 9.6: Dynamic data in Dog-tag generator (`http://www.mydogtag.com`, January 5 2014).

manually added two group names to the extracted data ($M7$) to categorize options extracted from each step.

S5: Shirt designer. S5 uses different types of widgets to present options. We specified two data patterns to extract image options and four data patterns to extract options presented using text boxes, radio buttons, and list boxes. Moreover, three region patterns are specified ($M1$). We called the data extraction procedure 11 times ($M4$) to extract data objects of interest.

In some steps in S5, options are categorized in different groups and each group is represented using a button. The user has to click on each button to make its relevant options visible (Figure 9.8). We could specify a dependency between a pattern that denotes the buttons (as the independent pattern) and the pattern for the contained options (as the dependent pattern)

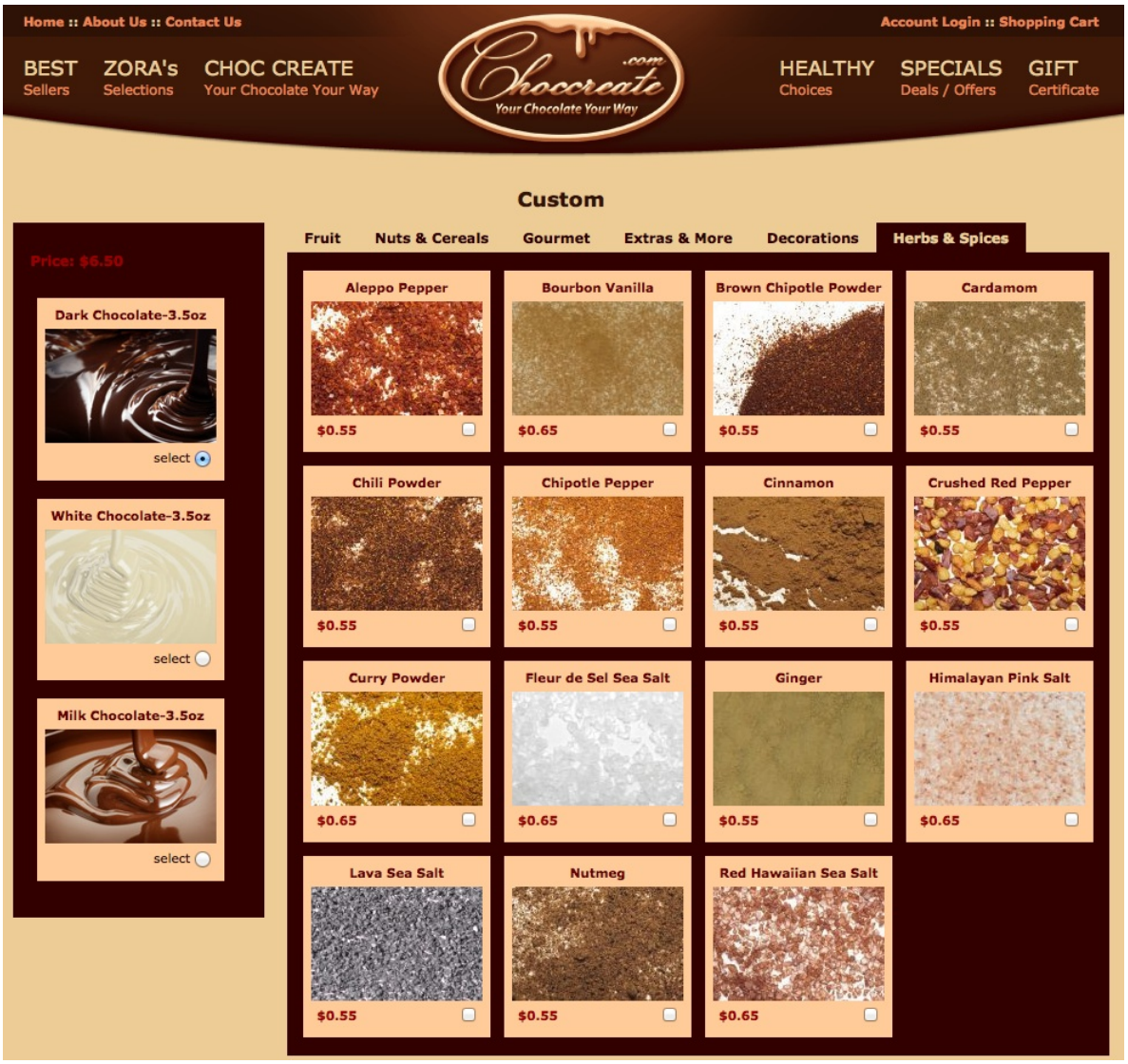


FIGURE 9.7: Chocolate maker (`http://www.choccreate.com`, January 5 2014).

to extract groups and their contained options. However, we noticed that all these options are encoded in the source code and they just become visible/hidden when needed. Therefore, we did not use the crawling scenario here. We used the Wrapper and extracted all these options. We also observed that options contained in the same group have the same value for the *name* attribute of their widget elements. We used this attribute to identify the relevant group names. Overall we extracted 233 true positive options from S5 ($M5$). Six options are also incorrectly identified. The reason for these false positives is that we did not consider the visibility of options and extracted all options encoded in the source code of the page. Using the crawling scenario we could avoid all these false positives. We also noticed that two extracted image options are redundant. The reason for this redundancy is that when the user selects an image option, the configurator creates another data object for this selected option and displays it in the page as

well. The Wrapper in fact extracted both these data objects. We had to manually remove false positive and redundant options from the extracted data ($M10$).

In some cases, all options presented in a region are represented using the same widgets, except one option that is represented using a different widget. In these cases, writing a specific pattern only for this option and using the Wrapper to extract that single option does not pay off. The engineer can manually add this option to the extracted data. We observed three such cases in S5 and manually added text options to the extracted options. In addition, we manually added 11 group names in the generated XML file ($M7$). When generating the TVL model from the XML file, the transformation module detected that five options have the same names. Therefore, we renamed these options to generate consistent TVL models ($M10$).

In S5, the Wrapper could identify 26 group constraints and 40 formatting constraints ($M8$).

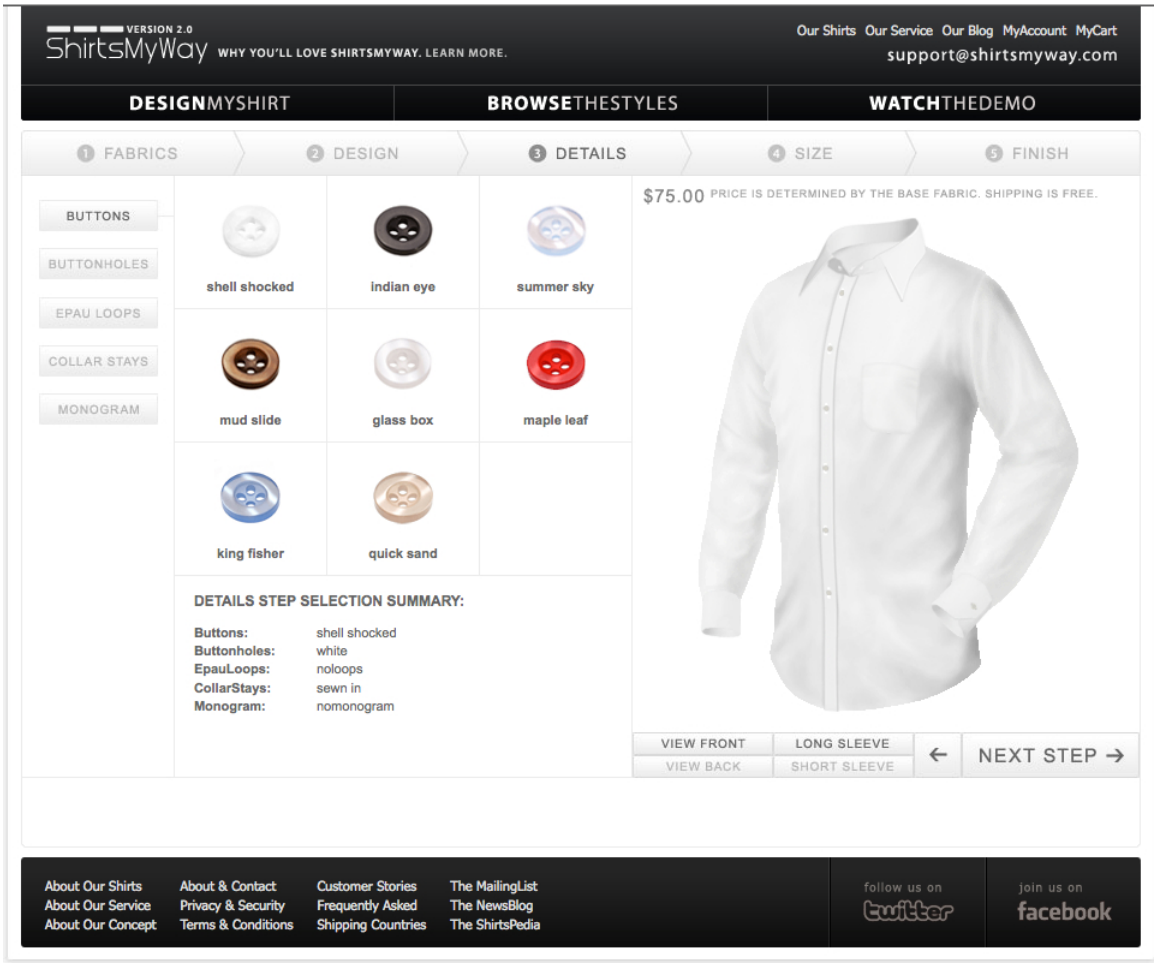


FIGURE 9.8: Shirt designer (`http://www.shirtsmyway.com`, January 6 2014).

9.3 Discussion

9.3.1 Evaluation results

We now discuss the key results of the application of our approach on the Web configurators studied in this experiment.

Accuracy of the extracted data (Q1). Overall, the accuracy of the extracted data is promising. Hundreds of configuration options and their associated descriptive information are extracted. In addition to options, our approach could also identify and record the logical relationships between options, i.e., constraints defined over options.

Let N be the total number of all the reported objects (i.e., the sum of the automatically extracted true positive objects, false positives, redundant objects, and manually added objects) for all the five studied configurators, and t be the total number of true positive objects automatically extracted by the Wrapper (in collaboration with the Crawler). Then, we define the precision P by: $P = \frac{t}{N} \times 100$.

As to options, our approach could extract all 724 options presented in the pages, plus 139 options identified by the crawling technique ($M5$, 44 in S3 plus 95 in S4). We only had six false positive and two redundant extracted options. 34 options had to be manually added to the extracted data. So, overall, we achieved 95% precision for option extraction.

Considering constraints, we cannot claim that our approach could detect and extract *all* constraints implemented by the configurators. However, we state that almost *all* the extracted constraints are true (except for a group constraint in S2). Overall, the proposed approach identified and collected 103 group constraints, 30 cross-cutting constraints, and 50 formatting constraints ($M8$). 16 group constraints and one formatting constraint were manually added. One of the automatically extracted group constraints was incorrect. The calculated precision for constraint extraction is 91%.

It is worth to mention other experiences in reverse engineering contexts [DDH+13, ACC+13, SLB+11] that also obtain incomplete feature models, and thus call for intervention of the user or any kind of knowledge/artefact to further refine the model [HPP+13].

Expressiveness of the proposed pattern language (Q2). We could specify patterns to cover all code fragments that encode configuration-specific objects in the subject systems ($M1$). Given these patterns, the Wrapper extracted all options presented in the pages as well as those that are dynamically generated by the Crawler ($M5$).

Pattern-specific elements and operators that we designed in our language gave us a lot of support for the studied configurators. We used the wildcard operator in pattern specification of four

configurators. The `skip(all)`, `skip(STRING)`, and `skip(sibling, MULTIPLICITY)` elements were used for three configurators. The *or* operator was used for two configurators (Table 9.4).

Applicability of the notion of dependency between patterns (Q3). The notion of dependency between patterns addresses the dynamic nature of the configuration process. We gain numerous additional configuration options with the crawling approach, which is implemented based on the pattern dependency.

In S3, by specifying two dependencies ($M2$), the Crawler could identify and extract 63 dynamic objects ($M6$). The use of this technique in S4 ($M2 = 1$) leads to extracting 95 dynamic options, which is almost three times more than options extracted without crawling ($M5 = 24$ and $M6 = 95$).

We also used the crawling technique to trigger and extract 30 cross-cutting constraints in S2 ($M8$). Moreover, dependency between patterns allowed us to document the parent-child relationships between options in S2.

Let N be the total number of true positive data objects and d be the total number of true positive *dynamic* data objects, both extracted from the five target Web configurators. Then, we define the average of dynamically extracted data objects by: $A = \frac{d}{N} \times 100$. The average of dynamic objects and constraints extracted by the crawling approach are 18% and 16%, respectively.

Nevertheless, we cannot claim that the crawling technique can detect and extract *all* objects that may be dynamically generated at runtime. It also cannot automatically generate the complete set of configurations that is required to deduce all cross-cutting constraints defined over options.

The manual effort required to perform the extraction process (Q4). In this experiment, we specified in total 13 region patterns and 19 data patterns ($M1$), wrote 322 lines of code for these patterns ($M3$), and executed them 39 times ($M4$) to extract all data. We also manually added 34 options ($M7$), 16 group constraints and one formatting constraint ($M9$) to the extracted data. We also had to remove 134 noisy data items and rename 51 extracted options. In addition, to denote the regions within which the Wrapper and the Crawler should operate, we manually edited 17 HTML elements in the source code of the pages. To reflect the hierarchical relationships that exist between the presented options in the pages, we manually changed the position of 12 feature blocks in the generated TVL files ($M10$).

Let N be the total number of true positive data objects either extracted automatically or added manually and m be the total number of true positive data objects that are manually added to the extracted data objects in the five studied Web configurators. Then, we define the average of manually added data objects by: $A = \frac{m}{N} \times 100$. The average of objects and constraints manually added by the user are 4% and 9%, respectively.

We believe that our semi-automatic and supervised approach provides a realistic mix of manual and automated work. It acts as an interesting starting point for re-engineering a configurator while mining the same amount of information manually is clearly daunting and error-prone. The manual writing of 322 lines of code to specify the required patterns in this experiment leads to generating TVL models with a total 4478 lines of code (see Table 9.5).

9.3.2 Qualitative observations

Specification of patterns

As it is expected, when the templates from which the configuration-specific objects are generated are structurally similar, a small set of patterns is required. We specifically observed that the use of wildcard (*) and *or* (|) operators in the attributes, and the `skip(sibling, Multiplicity)` and `skip(all)` elements are very useful in minimizing the set of required patterns and lines of code for each pattern. For instance in S4, the templates for options represented using radio buttons and those represented using check boxes are the same, except for the `type` attribute of the `input` elements, which is "radio" and "checkbox" respectively for radio buttons and check boxes. Using the *or* operator between values of the `type` attribute we could specify only one data pattern for both templates: <`input type="radio|checkbox"`>.

We also found the notion of multiplicity of an element (the `data-att-met-multiplicity` meta attribute) very practical in this experiment. For instance, the list of impacted options in the conflict window in S2 and the items of list boxes in S5 are examples of multi-instantiated elements that we could model in the patterns. In addition, in S1 we used the multiplicity element to denote the optional elements. The use of the `skip(all)` element in S3 also allowed us to shorten 13 lines of code in a pattern specification.

We found the use of structural attributes in the specification of patterns extremely useful. They provide significant measurement information for the Wrapper in finding code fragments that match a given pattern (in the pattern matching algorithm). As it is presented in Table 9.4, we used structural attributes in patterns of all the studied systems. In some cases, structural attributes are the only means to find objects of interest and ignore the irrelevant data objects. For instance in S2, some options are presented using images. In addition to these image options, there are a lot of image objects in the page that do not present configuration-specific objects and must be ignored by the Wrapper. If we use only the <`img`> element in the pattern specification, the extracted data will contain a lot of noisy data objects. However, we noticed that the `img` elements that present options have the `class="byoColorChipImage"` attribute. Therefore, we used the <`img class="byoColorChipImage"`> element in the pattern specification to extract image options.

Our Web data extraction approach provides a *tag-level* encoding [CKGS06] and considers any text string between two HTML tags or the value of a tag attribute as a token. It cannot treat each word of a string as a token. Consequently, we cannot automatically extract configuration-specific data if it is a substring of a text string. In S3, for instance, to extract the URL of the image options, we had to extract the whole string of a tag attribute and then manually extract the substring that presents the URL. The only tokenization capability of our pattern language is the `skip(STRING)` element that tells the Wrapper to visit the `STRING` value in the target text element but not extract it. We used this element in S3 to skip the word "Add" before the price data items, in S4 to ignore the word "select" before option names, and in S5 to remove the ":" symbol after option names.

Crawling the configuration space

For S1, we specified only one data pattern and one region pattern to extract options from all steps. However, we cannot directly apply the crawling approach to automatically explore the steps and extract all options in one execution because it uses the multi-page interface paradigm. We can envision to adapt our tool to support the multi-page interface paradigm, i.e., when different pages with different URLs are used.

For S2, S3, and S4, since some steps use different templates, and therefore different data patterns are required, we manually activated each step and did not use the crawling approach to automatically explore the steps. An interesting lesson learned is as follows: there is a tradeoff to find between spending time/effort in specifying patterns and manually helping the tool to navigate in the configuration space.

In S2, there are cross-cutting constraints defined over options contained in two consecutive steps "Packages" and "Options": by selecting an option in the former step, its *required* options are selected in the later step. To deduce these cross-cutting constraints, we specified a dependency between the data pattern defined for options in the "Packages" step and the data pattern that denotes the options in the "Options" step. We aimed to simulate the selection of options in "Packages" and the extraction of the selected options in "Options". For this special case, the Crawler failed to trigger the *click* event of options in "Packages". Due to this technical issue, we had to manually select options in "Packages" and then run the Wrapper to extract options from "Options". It explains why we had relatively high execution rate for S2.

Mining of constraints

To automatically deduce group constraints, we rely on the widget types and attributes such as `name` used to represent options. This experiment shows that this method perfectly works: only

one out of 104 extracted group constraints is incorrect.

The runtime performance of the pattern-matching algorithm

All the data extraction algorithms are implemented in JavaScript. This gives us a high-performance execution time when looking for code fragments of matching objects. However, when using the crawling technique, we have to wait for the application response (e.g., load new options in the page, display the conflict window, change the configuration state of options, etc.). Even for these cases, we observed that the response time is fast enough and the data extraction process provides almost real-time responses.

The extraction of other configuration-specific data

In some cases, in addition to options and constraints, we also could extract other configuration-specific data. For instance in S3, we recorded the *default configuration*, i.e., the string values of the text options and the list of the selected options when the configuration space is initially loaded in the page.

9.4 Threats to Validity

The main *external* threat to validity is the sample set of the subject systems involved in our evaluation. We only chosen samples from five sectors. A larger-scale evaluation is needed to further confirm the scalability of the approach.

An *internal* threat to validity is that our approach is supervised and the technical knowledge of the user running the extraction process, her choices, and interpretations can influence the results. This experiment was conducted by the author of the thesis. He proposed and developed the notion of *vde* patterns and implemented the tools. He already knows the chosen configurators and how they work. First, his choices on what data objects to be extracted from each Web configurator influenced the number of required data patterns. For instance in S3, if a user intends to extract only the name of options (and to exclude their widget type and price) four (instead of five) different data patterns are required. Second, in this experiment we considered the group names as well as (some of) the menus as configuration-specific objects. If another user does not make this assumption, the number of extracted objects will decrease. In a real reverse engineering context an engineer would have to be familiar with the configurators and be trained in using the reverse-engineering tools as well.

Another internal threat to validity is related to deducing cross-cutting constraints. To detect all cross-cutting constraints, all possible option combinations must be investigated but combinatorial blow-up precludes it. The impact this has on the completeness of the extracted constraints is hard to predict.

Chapter 10

Conclusion and Future work

10.1 Contributions

Nowadays, mass customization has been embraced by a large portion of the industry. As a result, the Web abounds with configurators that assist customers in customizing all kinds of products and services to their specific needs. A Web configurator is an online product configuration environment that presents hundreds of configuration options to customers who gradually select the options to be included in the final product. In many cases, Web configurators have become the single entry point for placing customer orders. As such, they are key assets for companies and act as a privileged interface between customers and companies.

Despite the high importance of Web configurators, a consistent body of knowledge dedicated to their engineering is still missing. To tackle this problem, empirical data on the current practice is required. In particular, we needed to understand the intrinsic nature of Web configurators. We therefore set out to answer the first research question in this PhD study:

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

To get a better grasp of main characteristics of Web configurators, we conducted an empirical and systematic study of 111 configurators from different industry sectors. We notably investigated their three essential dimensions: rendering of configuration options, constraint handling, and configuration process support. Based on this, we highlighted good and bad practices in engineering Web configurators.

Our empirical study exposed that configurators are complex systems: a diversity of Web widgets used to represent configuration options in different layouts, numerous kinds of constraints

govern the options, the configuration process can be multi-step and non linear, and advanced capabilities are provided to check consistency, propagate user decisions, etc. This study also revealed that although such applications have specific common characteristics, they are developed in an unspecific way, that is, like any other Web application. The absence of specific, adapted, and rigorous methods in engineering Web configurators leads to maintainability, usability, and reliability issues. Specifically, we identified a number of bad practices in the configurators: incomplete reasoning, counter-intuitive representation of options, losing of all decisions when navigating backward, etc.

The empirical study provided enough evidence that for Web configurators qualities like usability and correctness are not convincingly satisfied. This opened avenues for re-engineering support and methodologies to migrate legacy Web configurators to more reliable, efficient, and maintainable solutions. We offered to first systematically *reverse engineer* a variability model, *viz.*, a feature model (in TVL), from a legacy Web configurator and then to use this model to *forward engineer* a new improved configurator that has a customized and easily maintainable user interface as well as an underlying reliable reasoning engine.

The major difficulty in reverse engineering Web configurators is that, despite having a common goal and similar features, they vary significantly. A first notable variation lies in the way variability data are implemented and presented in the Web pages (or in a multiplicity of intermediate panels/pages): they use a variety of Web objects to visually represent variability data; a page can contain various kinds of data objects with different structures; the data objects can have complex structures, etc. Web configurators also vary in the way they load data in the pages, handle different kinds of constraints, and control the configuration process. They are highly interactive and dynamic applications. As they are executing, new content may be created and automatically added to the page, and existing content may be removed or changed.

In the second part of this PhD research, we were concerned with the reverse-engineering of Web configurators. We planed to develop a consistent set of methods, languages and tools to extract variability data from a Web configurator. More precisely, our main research questions is:

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

Our survey of the state of the art in reverse engineering Web applications, Web data extraction, and synthesis of feature models shows that the problem of extracting feature models from Web configurators had not been studied. Existing approaches do not consider specific properties of such applications. They are either too general or designed for a different application domain.

To answer **RQ2**, we first investigated the client-side of a sample set of Web configurators to understand how configuration-specific objects are implemented. We observed that objects presenting 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 the page. In a template, text elements and tag attributes are data slots filled by data items when generating the page. We thought that we could use these templates to extract data of interest. In order to achieve this aim, we proposed the notion of *variability data extraction pattern* (*vde* pattern). The user specifies a pattern, expressed in an HTML-like language, to define the structure of objects of interest and to mark data items to be extracted from these objects. A pattern, in fact, represents a number of templates from which the objects of interest are generated.

The specified patterns are given to a data extraction procedure, called a Web Wrapper, that tries to locate in a Web page code fragments (presenting objects) that structurally conform to input patterns, and extracts as output data items from those code fragments corresponding to the marked data in the patterns. We also proposed and implemented a novel pattern matching algorithm using which the Wrapper finds the matching code fragments.

RQ2 addressed the problem of extracting structured variability data by static analysis of the source code of a Web page. However, a static analysis is clearly not sufficient in general. It does not account for the dynamic nature of the configuration process and the runtime behaviour of Web configurators. We therefore formulated the third research question of this PhD thesis as follows:

- **RQ3** *How to (semi-)automatically extract the dynamic variability content from the Web pages of a configurator?*

To address **RQ3**, we developed a crawling technique, provided by a Web Crawler. The Crawler automatically explores the configuration space (e.g., navigates through the configuration steps) and simulates users' configuration actions. The exploration and configuration actions usually add new data objects to the page or change existing data objects. The Wrapper then analyses the content of the page to identify and extract newly added data, or deduce variability data from the changes made to the data objects. We also implemented an approach that uses the crawling technique to trigger and extract constraints defined over options.

The extracted data is hierarchically organized and serialized using an XML format. We implemented a Java module to transform the generated XML file into a feature model represented in TVL.

Once we developed our tool-supported and supervised reverse-engineering process, we then set up an experiment and evaluated our approach. In particular, we evaluated the scalability of the

approach, the accuracy of the extracted data, and the users' manual effort required to run the reverse-engineering process. Experimental results on five existing Web configurators show that the specification of a few patterns allows to identify hundreds of options and constraints.

10.2 Major limitations

This section presents the limitations of the proposed approach and the future work that is needed to better address these problems.

Only the client-side of configurators is considered. In the empirical study of Web configurators, we considered their client sides because a lot of valuable information can be extracted: GUI data, constraint management, configuration process, etc. We recognized that the server-side can be also considered for some aspects of the study (e.g., to determine how reasoning operations are implemented) and gaining additional insights.

Building a complete feature model requires, ideally, analysing both the client and server sides of a configurator. It typically also requires consulting other sources such as documentation, expert knowledge, etc. We investigated here the visible parts of configurators, i.e., the GUI and the Web client because it is the entry point for customer orders and most of the variability data is somehow represented in Web pages. Moreover, analysing the server-side will require the use of completely different reverse-engineering techniques, on various kinds of artefacts. Furthermore, server code is much more difficult to obtain, even for research purpose.

The method relies on the expert's manual effort. At present, the expert inspects the source code of the page, identifies templates from which the data objects of interest are generated, and then specifies the appropriate patterns for these templates. This activity can be cumbersome in some cases. A possible improvement for this limitation is to integrate our approach with methods that can infer templates used to generate a given page. From these identified templates, the expert can choose those which are used to generate configuration-specific objects. It can decrease the practitioners' effort when re-engineering their (legacy) configurators. A problem with these methods is that they make many assumptions on how the page is formatted and the data is represented (e.g., data records are presented in a list).

The crawling technique provides no support for multi-page Web applications. The Crawler is able to explore the configuration space in a page and unable to navigate through the pages of a Website. For this reason, the approach does not support constraints across pages. We intend to integrate our approach with Web crawling approaches aiming to explore pages in Web applications that follow multi-page user interface paradigm.

10.3 Perspectives

10.3.1 Forward engineering

The main motivation for reverse engineering feature models from Web configurators is to use them for forward engineering more reliable and maintainable configurators. Boucher *et al.* have already studied the problem of deriving user-friendly configuration interfaces from feature models [BPH12]. They discussed the main challenges and possible solutions, from the visual and behavioural perspectives. The authors then presented a generic model-driven method to use a feature model for the generation of configuration GUIs [BPAH12]. In this approach, the GUI is supported by an underlying reasoning engine to control and update the GUI elements. Figure 10.1 presents the model-view-controller (MVC) architecture proposed by Boucher *et al.* to design configurators. In this architecture:

- The **model** is a feature model presented in TVL. The feature model is connected to a reasoning engine (SAT/SMT solver), which is responsible for controlling the interactive configuration through a generic API.
- The **view** contains a description of the GUI to be displayed to the user. This description is generated from the feature model using the XML User Interface Language (XUL).
- The **controller**, as the central component of the architecture, listens to user actions, updates the feature model (selected features, attribute values, etc.) and interacts with the reasoning engine to determine the list of changes to be propagated to the GUI. Once done, it updates the GUI model by hiding, making visible or updating elements affected by the changes.

The combination of a model-based approach to produce customized GUIs from a feature model with a reliable engine to reason about this feature model would provide an easily maintainable user interface and correct configurations.

10.3.2 Configuration verification

In addition to a feature model, our proposed reverse-engineering process also produces a process model. The process model is a description of configuration steps, options contained in each step, and optionally the steps' order (i.e., the workflow of the configuration process). The reverse-engineered feature models and process models can be used for verification purposes, e.g., checking the completeness and correctness of the configuration constraints.

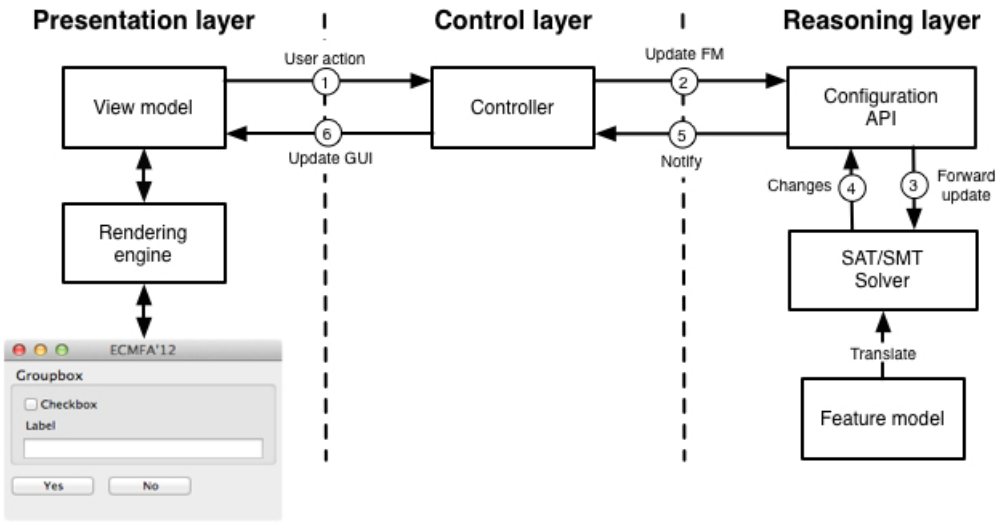


FIGURE 10.1: A MVC-like architecture for Configurators [BPAH12].

We developed a tool that provides a lightweight environment for validation of feature models and verification of the configuration process [AHH11b, AHH11a]. This tool can be used by the developers of the configurators to test their models before using them in development of actual Websites. The conceptual foundations for this tool are laid on the notions of *multi-view* feature models [HHSD10, HHS+13] and *feature configuration workflows* (FCW) [HCH09]:

Multi-view feature model. A view is defined on a feature model as a subset of its features. Several views allow to divide the feature model into smaller, more manageable parts. Views can be defined for specific stakeholders, roles, configuration steps, or particular combinations of these elements.

Feature configuration workflow (FCW). FCW is a formalism that proposes to use a workflow to drive the configuration of views. The workflow defines the configuration process and each view on the feature model is assigned to a task in the workflow. A view is configured when the corresponding workflow task is executed (Figure 10.2).

Support for FCW has been implemented by extending and integrating two third-party tools: SPLOT [MBC09][1] and YAWL[2]. SPLOT supports feature modelling and configuration. To provide efficient interactive configuration, SPLOT relies on a SAT solver (SAT4J[3]) and a BDD solver (JavaBDD[4]). We extended SPLOT to support view creation, configuration, and view-to-workflow mapping. Workflow design, execution, analysis and user management is provided by YAWL. Interactive services were added to YAWL so as to trigger view-based configuration in SPLOT. We also implemented a new configuration environment, the *FCW engine*, which is

[1] http://www.splot-research.org/
[2] http://www.yawlfoundation.org/
[3] http://www.sat4j.org/
[4] http://javabdd.sourceforge.net/

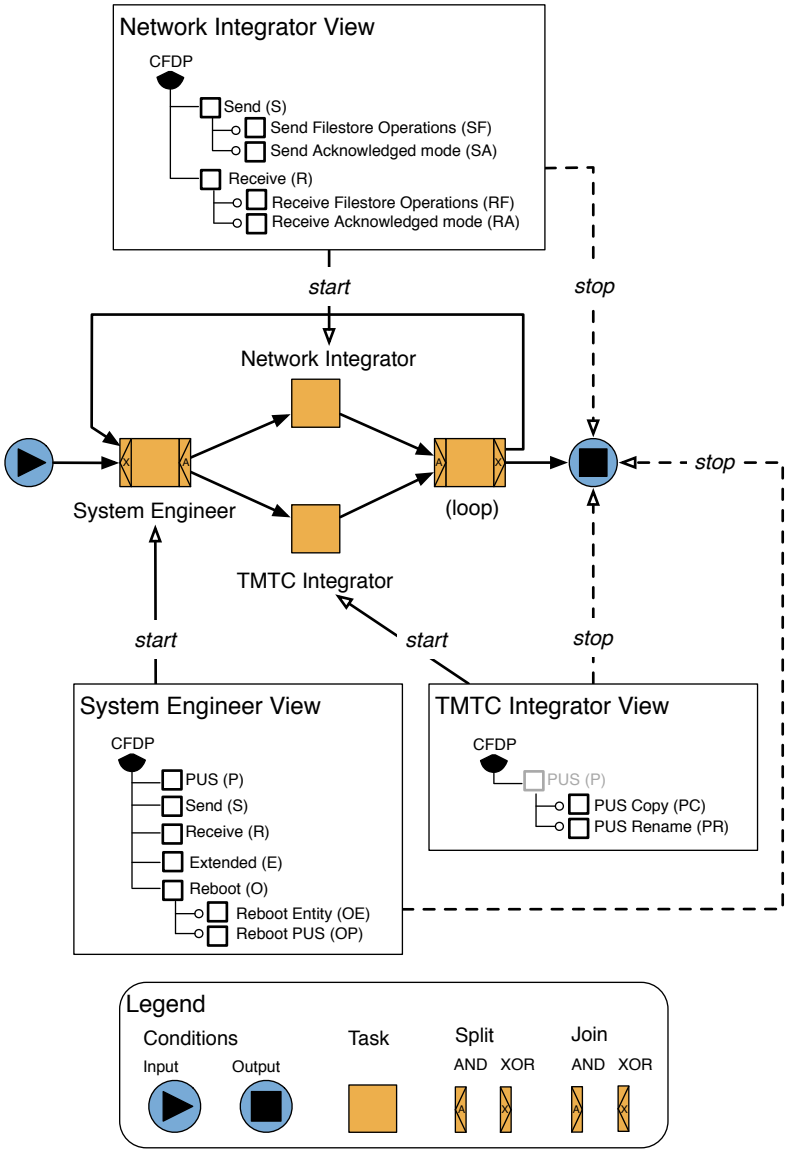


FIGURE 10.2: Example of FCW.

responsible for managing configuration sessions, conveying the information between YAWL and SPLOT, and monitoring the whole configuration process.

Figure 10.3 shows the essential components of our integrated tool as well as a typical usage scenario.

Design time: Configuration preparation. At design time, the user defines and stores views in SPLOT (❶). A view is defined with an XPath-like expression [HHSD10]. The XPath expression specifies paths to features in the feature model that should be part of the view. A coverage test is run to verify that the whole feature model can be configured through the defined views, i.e., that no feature can be left undecided after the views have been configured (Figure 10.4). The user also designs the workflow and stores it in the repository in YAWL (❶). Once created and

checked, the workflow is uploaded and registered in SPLOT (❷). Once the required views and workflow are made available, the mapping of the views to tasks of the workflow is performed in SPLOT (❸). A view not only has to be mapped to a task that triggers its configuration, but also to a *stop* that tells when it should be fully configured. The stop is materialized in the workflow by a condition. The mapping is correct and complete when (1) all the views are mapped to exactly one task and one stop, and (2) the coverage of the mapped views is complete.

Runtime: Product configuration. At runtime, the product configuration process starts in YAWL (❹). The user executes a task (❺), which calls the associated Web service. When an element is activated in YAWL, the Web service sends its name, its type (task or condition) and the session information to the FCW engine. The *Coordinate configuration* service in the engine handles messages received from YAWL and SPLOT (❻). The FCW engine controls the status of tasks and conditions. The status of a task can be *Ready*, *Configured*, or *Completed*. Similarly, the status of the stops can be *Ready* or *Completed*. The FCW engine initiates either a view configuration request if the element is a task, or a configuration status if it is a condition. If it is a configuration request, SPLOT loads the corresponding view (❼). In the interactive configuration form, the user performs the configuration by selecting/deselecting the features (❽). SPLOT controls the configuration process to guarantee that only valid decisions are made (Figure 10.5). We extended SPLOT to support partial and complete configuration, persistency and recovery, and decision logging.

When the configuration of the view is terminated, the FCW engine updates the status of the task (❾), and the user can mark the task as complete in YAWL (❿). If the place is a condition, the FCW engine requests the list of views attached to the stop (❼). SPLOT returns the status of each of the views to the FCW engine (❽), which then checks whether the stop is satisfied, i.e., whether all the views are completely configured (❾). When the final condition is reached, the configuration stops, and the resulting product can be retrieved from the repository in SPLOT.

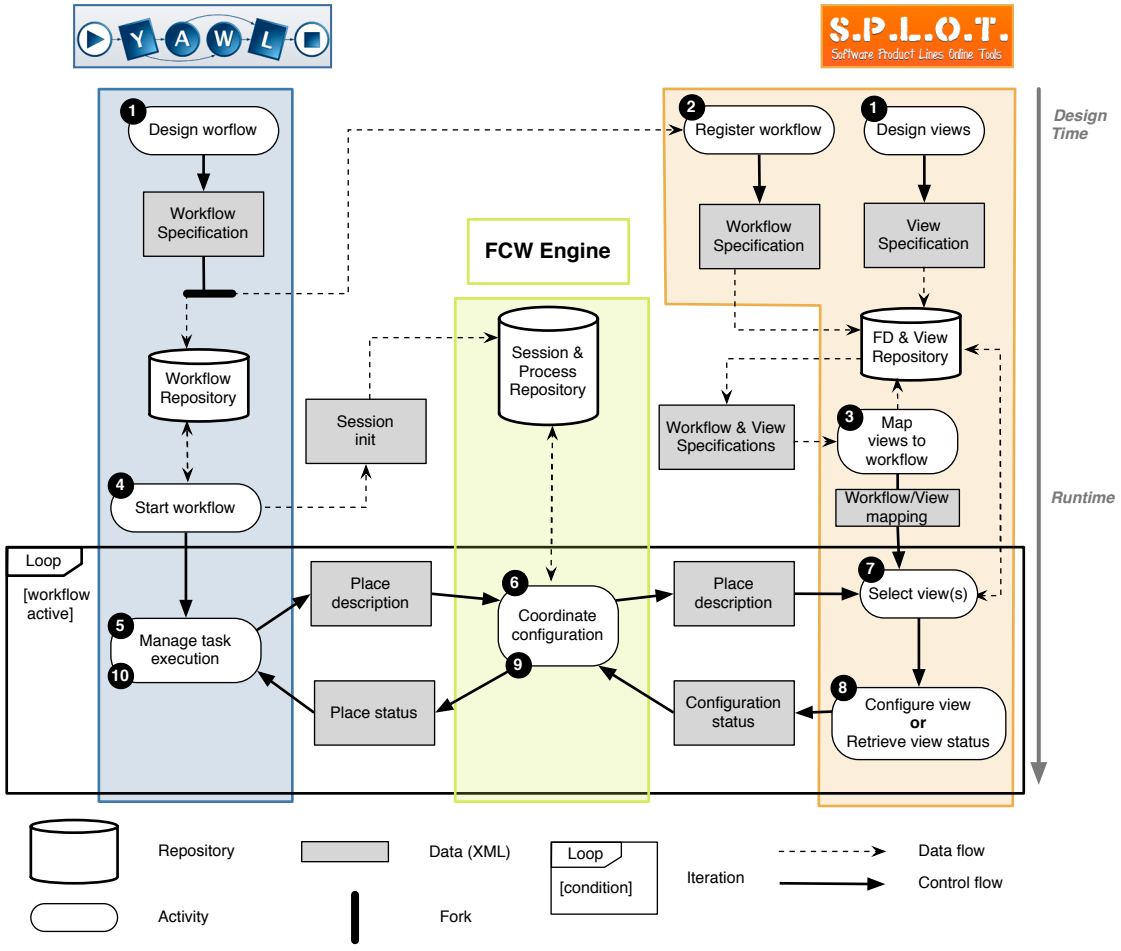


FIGURE 10.3: Overview of the essential components and typical use case scenario.

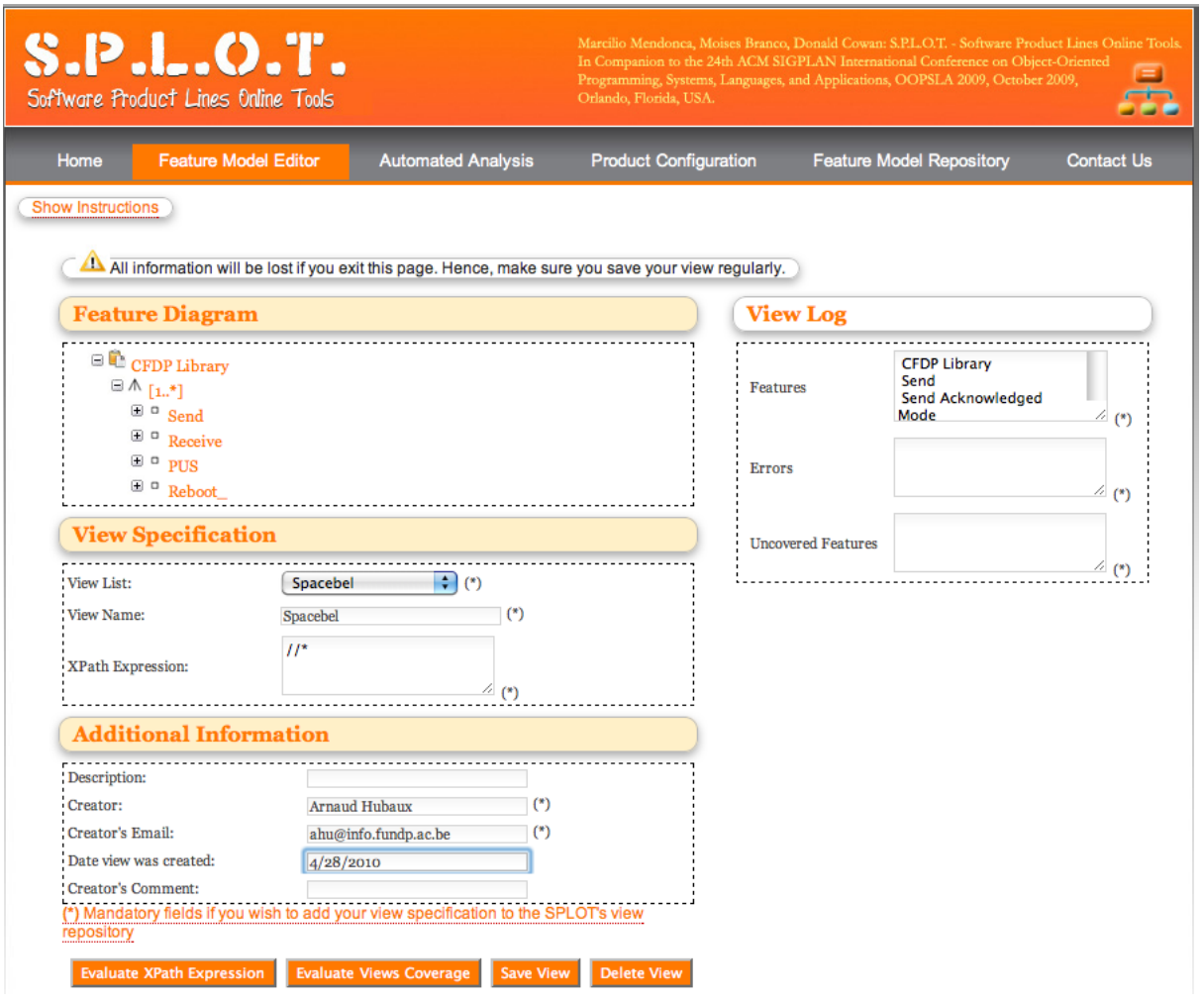


FIGURE 10.4: View creation menu.

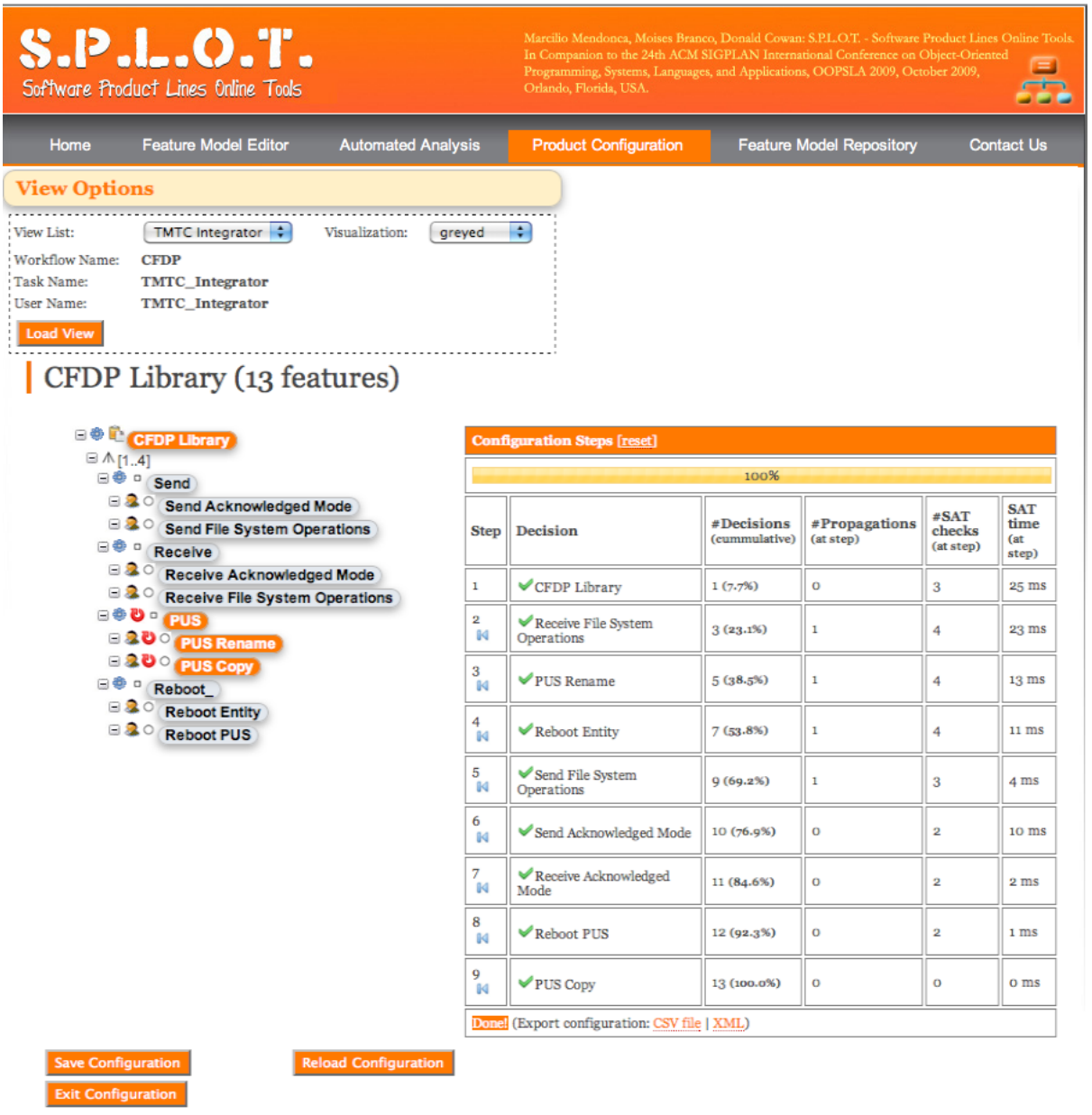


FIGURE 10.5: View configuration menu.

Bibliography

[ACC+11] Mathieu Acher, Anthony Cleve, Philippe Collet, Philippe Merle, Laurence Duchien, and Philippe Lahire. Reverse engineering architectural feature models. In *Proceedings of the 5th European conference on Software architecture*, ECSA'11, pages 220–235, Berlin, Heidelberg, 2011. Springer-Verlag.

[ACC+13] Mathieu Acher, Anthony Cleve, Philippe Collet, Philippe Merle, Laurence Duchien, and Philippe Lahire. Extraction and evolution of architectural variability models in plugin-based systems. *Software and Systems Modeling*, pages 1–28, 2013.

[ACLF13] Mathieu Acher, Philippe Collet, Philippe Lahire, and Robert B. France. Familiar: A domain-specific language for large scale management of feature models. *Science of Computer Programming*, 78(6):657 – 681, 2013.

[ACP+12] Mathieu Acher, Anthony Cleve, Gilles Perrouin, Patrick Heymans, Charles Vanbeneden, Philippe Collet, and Philippe Lahire. On extracting feature models from product descriptions. In *Proceedings of the Sixth International Workshop on Variability Modeling of Software-Intensive Systems*, VaMoS '12, pages 45–54, New York, NY, USA, 2012. ACM.

[ACSW12] Nele Andersen, Krzysztof Czarnecki, Steven She, and Andrzej Wasowski. Efficient synthesis of feature models. In *Proceedings of the 16th International Software Product Line Conference - Volume 1*, SPLC '12, pages 106–115. ACM, 2012.

[Ade98] Brad Adelberg. Nodose - a tool for semi-automatically extracting structured data and semi-structured data from text documents. *SIGMOD Rec.*, 27(2):283–294, June 1998.

[AGM03] Arvind Arasu and Hector Garcia-Molina. Extracting structured data from web pages. In *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, SIGMOD '03, pages 337–348, New York, NY, USA, 2003. ACM.

[AHA+13] Ebrahim Khalil Abbasi, Arnaud Hubaux, Mathieu Acher, Quentin Boucher, and Patrick Heymans. The anatomy of a sales configurator: An empirical study of 111 cases. In Camille Salinesi, MoiraC. Norrie, and scar Pastor, editors, *Advanced Information Systems Engineering*, volume 7908 of *Lecture Notes in Computer Science*, pages 162–177. Springer Berlin Heidelberg, 2013.

[AHH11a] Ebrahim Khalil Abbasi, Arnaud Hubaux, and Patrick Heymans. An interactive multi-perspective toolset for non-linear product configuration processes. In *Proceedings of the 15th International Software Product Line Conference, Volume 2*, SPLC '11, pages 50:1–50:1, New York, NY, USA, 2011. ACM.

[AHH11b] Ebrahim Khalil Abbasi, Arnaud Hubaux, and Patrick Heymans. A toolset for feature-based configuration workflows. In *Proceedings of the 2011 15th International Software Product Line Conference*, SPLC '11, pages 65–69, Washington, DC, USA, 2011. IEEE Computer Society.

[ALMK08] Sven Apel, Christian Lengauer, Bernhard Mller, and Christian Kstner. An algebra for features and feature composition. In Jos Meseguer and Grigore Rou, editors, *Algebraic Methodology and Software Technology*, volume 5140 of *Lecture Notes in Computer Science*, pages 36–50. Springer Berlin Heidelberg, 2008.

[AM99] Gustavo O. Arocena and Alberto O. Mendelzon. Weboql: restructuring documents, databases, and webs. *Theor. Pract. Object Syst.*, 5(3):127–141, August 1999.

[APR+10] Manuel Álvarez, Alberto Pan, Juan Raposo, Fernando Bellas, and Fidel Cacheda. Finding and extracting data records from web pages. *J. Signal Process. Syst.*, 59(1):123–137, April 2010.

[ASB+08] Vander Alves, Christa Schwanninger, Luciano Barbosa, Awais Rashid, Peter Sawyer, Paul Rayson, Christoph Pohl, and Andreas Rummler. An exploratory study of information retrieval techniques in domain analysis. In *Proceedings of the 2008 12th International Software Product Line Conference*, SPLC '08, pages 67–76, Washington, DC, USA, 2008. IEEE Computer Society.

[BA06] Thorsten Blecker and Nizar Abdelkafi. Mass customization: State-of-the-art and challenges. In Thorsten Blecker and Gerhard Friedrich, editors, *Mass Customization: Challenges and Solutions*, volume 87 of *International Series in Operations Research and Management Science*, pages 1–25. Springer US, 2006.

[BAH+12] Q. Boucher, E. 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.

[BBRC06] Don Batory, David Benavides, and Antonio Ruiz-Cortes. Automated analysis of feature models: Challenges ahead. *Commun. ACM*, 49(12):45–47, December 2006.

[BCW11] Kacper Bk, Krzysztof Czarnecki, and Andrzej Wsowski. Feature and meta-models in clafer: Mixed, specialized, and coupled. In Brian Malloy, Steffen Staab, and Mark Brand, editors, *Software Language Engineering*, volume 6563 of *Lecture Notes in Computer Science*, pages 102–122. Springer Berlin Heidelberg, 2011.

[Beu12] Danilo Beuche. Modeling and building software product lines with pure::variants. In *Proceedings of the 16th International Software Product Line Conference - Volume 2*, SPLC '12, pages 255–255, New York, NY, USA, 2012. ACM.

[BFG01a] Robert Baumgartner, Sergio Flesca, and Georg Gottlob. Declarative information extraction, web crawling, and recursive wrapping with lixto. In Thomas Eiter, Wolfgang Faber, and Miroslaw Truszczyski, editors, *Logic Programming and Nonmotonic Reasoning*, volume 2173 of *Lecture Notes in Computer Science*, pages 21–41. Springer Berlin Heidelberg, 2001.

[BFG01b] Robert Baumgartner, Sergio Flesca, and Georg Gottlob. Visual web information extraction with lixto. In *Proceedings of the 27th International Conference on Very Large Data Bases*, VLDB '01, pages 119–128, San Francisco, CA, USA, 2001. Morgan Kaufmann Publishers Inc.

[BGG09] Robert Baumgartner, Wolfgang Gatterbauer, and Georg Gottlob. Web data extraction system, 2009.

[BGGB02] Martin Becker, Lars Geyer, Andreas Gilbert, and Karsten Becker. Comprehensive variability modelling to facilitate efficient variability treatment. In Frank Linden, editor, *Software Product-Family Engineering*, volume 2290 of *Lecture Notes in Computer Science*, pages 294–303. Springer Berlin Heidelberg, 2002.

[BGPP12] Federico Bellucci, Giuseppe Ghiani, Fabio Paternò, and Claudio Porta. Automatic reverse engineering of interactive dynamic web applications to support adaptation across platforms. In *Proceedings of the 2012 ACM international conference on Intelligent User Interfaces*, IUI '12, pages 217–226, New York, NY, USA, 2012. ACM.

[Bou06] Laurent Bouillon. *Reverse Engineering of Declarative User Interfaces*. PhD thesis, Universite catholique de Louvain, Louvain-la-Neuve, Belgium, 2006.

[BPAH12] Quentin Boucher, Gilles Perrouin, Mathieu Acher, and Patrick Heymans. Engineering configuration graphical user interfaces: A model-based perspective. Technical Report P-CS-TR ECMFA-000001, University of Namur, 2012.

[BPH12] Quentin Boucher, Gilles Perrouin, and Patrick Heymans. Deriving configuration interfaces from feature models: a vision paper. In *Proceedings of the Sixth International Workshop on Variability Modeling of Software-Intensive Systems*, VaMoS '12, pages 37–44, New York, NY, USA, 2012. ACM.

[BS09] Janota Mikolas Botterweck, Goetz and Denny Schneeweiss. A design of a configurable feature model configurator. In *3rd International Workshop on Variability Modelling of Software-Intensive Systems (VAMOS 2009)*. 2009.

[BSL+10] Thorsten Berger, Steven She, Rafael Lotufo, Andrzej Wasowski, and Krzysztof Czarnecki. Variability modeling in the real: a perspective from the operating systems domain. In *ASE'10*, pages 73–82. ACM, 2010.

[BSRC10] David Benavides, Sergio Segura, and Antonio Ruiz-Cortés. Automated analysis of feature models 20 years later: A literature review. *Inf. Syst.*, 35(6):615–636, September 2010.

[BSTRc07] David Benavides, Sergio Segura, Pablo Trinidad, and Antonio Ruiz-corts. Fama: Tooling a framework for the automated analysis of feature models. In *In Proceeding of the First International Workshop on Variability Modelling of Softwareintensive Systems (VAMOS*, pages 129–134, 2007.

[CBFH10] Andreas Classen, Quentin Boucher, Paul Faber, and Patrick Heymans. The TVL specification. Technical Report P-CS-TR SPLBT-00000003, PReCISE Research Center, University of Namur, Namur, Belgium, 2010.

[CBH11a] Andreas Classen, Quentin Boucher, and Patrick Heymans. A text-based approach to feature modelling: Syntax and semantics of tvl. *Sci. Comput. Program.*, 76(12):1130–1143, December 2011.

[CBH11b] Andreas Classen, Quentin Boucher, and Patrick Heymans. A text-based approach to feature modelling: Syntax and semantics of tvl. *Sci. Comput. Program.*, 76(12):1130–1143, December 2011.

[CCI90] Elliot J. Chikofsky and James H. Cross II. Reverse engineering and design recovery: a taxonomy. *Software, IEEE*, 7(1):13–17, 1990.

[CE05] Helsen Simon Czarnecki, Krzysztof and Ulrich Eisenecker. Formalizing cardinality-based feature models and their specifications. *Software Process Improvement and Practice*, 10(1):7–29, 2005.

[CGR+12] Krzysztof Czarnecki, Paul Grünbacher, Rick Rabiser, Klaus Schmid, and Andrzej Wasowski. Cool features and tough decisions: a comparison of variability modeling approaches. In *Proceedings of the Sixth International Workshop on Variability Modeling of Software-Intensive Systems*, VaMoS '12, pages 173–182, New York, NY, USA, 2012. ACM.

[CHS08] Andreas Classen, Patrick Heymans, and Pierre-Yves Schobbens. What's in a feature: A requirements engineering perspective. In *Proceedings of the Theory and Practice of Software, 11th International Conference on Fundamental Approaches to Software Engineering*, FASE'08/ETAPS'08, pages 16–30, Berlin, Heidelberg, 2008. Springer-Verlag.

[CK04] Chia-Hui Chang and Shih-Chien Kuo. Olera: Semisupervised web-data extraction with visual support. *IEEE Intelligent Systems*, 19(6):56–64, November 2004.

[CK05] Krzysztof Czarnecki and Chang Hwan Peter Kim. Cardinality-based feature modeling and constraints: A progress report. In *OOPSLA'05*, 2005.

[CKGS06] Chia-Hui Chang, Mohammed Kayed, Moheb Ramzy Girgis, and Khaled F. Shaalan. A survey of web information extraction systems. *IEEE Trans. on Knowl. and Data Eng.*, 18(10):1411–1428, October 2006.

[CL01] Chia-Hui Chang and Shao-Chen Lui. Iepad: information extraction based on pattern discovery. In *Proceedings of the 10th international conference on World Wide Web*, WWW '01, pages 681–688, New York, NY, USA, 2001. ACM.

[CM98] Valter Crescenzi and Giansalvatore Mecca. Grammars have exceptions. *Inf. Syst.*, 23(9):539–565, December 1998.

[CM99] Mary Elaine Califf and Raymond J. Mooney. Relational learning of pattern-match rules for information extraction. In *Proceedings of the sixteenth national conference on Artificial intelligence and the eleventh Innovative applications of artificial intelligence conference innovative applications of artificial intelligence*, AAAI '99/IAAI '99, pages 328–334, Menlo Park, CA, USA, 1999. American Association for Artificial Intelligence.

[CMM01] Valter Crescenzi, Giansalvatore Mecca, and Paolo Merialdo. Roadrunner: Towards automatic data extraction from large web sites. In *Proceedings of the 27th*

International Conference on Very Large Data Bases, VLDB '01, pages 109–118, San Francisco, CA, USA, 2001. Morgan Kaufmann Publishers Inc.

[CN02] Paul Clements and Linda Northrop. *Software Product Lines: Practices and Patterns*. Addison Wesley, 2002.

[Con99] Jim Conallen. Modeling web application architectures with uml. *Commun. ACM*, 42(10):63–70, October 1999.

[CZZM05] Kun Chen, Wei Zhang, Haiyan Zhao, and Hong Mei. An approach to constructing feature models based on requirements clustering. In *Proceedings of the 13th IEEE International Conference on Requirements Engineering*, RE '05, pages 31–40, Washington, DC, USA, 2005. IEEE Computer Society.

[DDH+13] Jean-Marc Davril, Edouard Delfosse, Negar Hariri, Mathieu Acher, Jane Cleland-Huang, and Patrick Heymans. Feature model extraction from large collections of informal product descriptions. In *European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE'13)*, 2013.

[DLFT04] Giuseppe Antonio Di Lucca, Anna Rita Fasolino, and Porfirio Tramontana. Reverse engineering web applications: the WARE approach. *J. Softw. Maint. Evol.*, 16(1-2):71–101, January 2004.

[DLW05] Dirk Draheim, Christof Lutteroth, and Gerald Weber. A source code independent reverse engineering tool for dynamic web sites. In *Proceedings of the Ninth European Conference on Software Maintenance and Reengineering*, CSMR '05, pages 168–177, Washington, DC, USA, 2005. IEEE Computer Society.

[dS10] Carlos Eduardo Bastos e Marques de Silva. *Reverse Engineering of Rich Internet Applications*. PhD thesis, University of Minho, Portugal, 2010.

[ECJ+99] D. W. Embley, D. M. Campbell, Y. S. Jiang, S. W. Liddle, D. W. Lonsdale, Y.-K. Ng, and R. D. Smith. Conceptual-model-based data extraction from multiple-record web pages. *Data Knowl. Eng.*, 31(3):227–251, November 1999.

[FMFB12] Emilio Ferrara, Pasquale De Meo, Giacomo Fiumara, and Robert Baumgartner. Web data extraction, applications and techniques: A survey. *CoRR*, abs/1207.0246, 2012.

[FP02] N. Franke and F.T. Piller. *Configuration Toolkits for Mass Customization: Setting a Research Agenda*. Technische Universität München, 2002.

[Fre98] Dayne Freitag. Information extraction from html: application of a general machine learning approach. In *Proceedings of the fifteenth national/tenth conference on Artificial intelligence/Innovative applications of artificial intelligence*, AAAI '98/IAAI '98, pages 517–523, Menlo Park, CA, USA, 1998. American Association for Artificial Intelligence.

[Fre00] Dayne Freitag. Machine learning for information extraction in informal domains. *Machine Learning*, 39(2-3):169–202, 2000.

[GFA98] M. L. Griss, J. Favaro, and M. d' Alessandro. Integrating feature modeling with the rseb. In *Proceedings of the 5th International Conference on Software Reuse*, ICSR '98, pages 76–, Washington, DC, USA, 1998. IEEE Computer Society.

[GWJV+09] Florian Gottschalk, Teun A. C. Wagemakers, Monique H. Jansen-Vullers, Wil M. P. van der Aalst, and Marcello La Rosa. Configurable process models: Experiences from a municipality case study. In *CAiSE*, pages 486–500, 2009.

[HCH09] Arnaud Hubaux, Andreas Classen, and Patrick Heymans. Formal modelling of feature configuration workflows. In *Proceedings of the 13th International Software Product Line Conference*, SPLC '09, pages 221–230, Pittsburgh, PA, USA, 2009. Carnegie Mellon University.

[HD98] Chun-Nan Hsu and Ming-Tzung Dung. Generating finite-state transducers for semi-structured data extraction from the web. *Inf. Syst.*, 23(9):521–538, December 1998.

[hd11] http://www.configurator database.com, 2011.

[HHS+13] Arnaud Hubaux, Patrick Heymans, Pierre-Yves Schobbens, Dirk Deridder, and Ebrahim Khalil Abbasi. Supporting multiple perspectives in feature-based configuration. *Softw. Syst. Model.*, 12(3):641–663, July 2013.

[HHSD10] Arnaud Hubaux, Patrick Heymans, Pierre-Yves Schobbens, and Dirk Deridder. Towards multi-view feature-based configuration. In Roel Wieringa and Anne Persson, editors, *Requirements Engineering: Foundation for Software Quality*, volume 6182 of *Lecture Notes in Computer Science*, pages 106–112. Springer Berlin Heidelberg, 2010.

[HK05] Andrew Hogue and David Karger. Thresher: automating the unwrapping of semantic content from the world wide web. In *Proceedings of the 14th international conference on World Wide Web*, WWW '05, pages 86–95, New York, NY, USA, 2005. ACM.

[hLC12] http://www.w3.org/TR/DOM Level-2-Core/introduction.html. W3C, what is the Document Object Model?, 2012.

[HLHE11] E.N. Haslinger, R.E. Lopez-Herrejon, and A. Egyed. Reverse engineering feature models from programs' feature sets. In *18th Working Conference on Reverse Engineering (WCRE), 2011*, pages 308–312, 2011.

[HLHE13] Evelyn Nicole Haslinger, Roberto Erick Lopez-Herrejon, and Alexander Egyed. On extracting feature models from sets of valid feature combinations. In *Proceedings of the 16th International Conference on Fundamental Approaches to Software Engineering*, FASE'13, pages 53–67, Berlin, Heidelberg, 2013. Springer-Verlag.

[HMGM97] Joachim Hammer, Jason McHugh, and Hector Garcia-Molin. Semistructured data: the tsimmis experience. In *Proceedings of the First East-European conference on Advances in Databases and Information systems*, ADBIS'97, pages 22–22, Swinton, UK, UK, 1997. British Computer Society.

[HMR08] L. Hvam, N. Henrik Mortensen, and J. Riis. *Product Customization*. Springer-Verlag Berlin Heidelberg, 2008.

[HOM98] Grel Hedin, Lennart Ohlsson, and John McKenna. Product configuration using object oriented grammars. In Boris Magnusson, editor, *System Configuration Management*, volume 1439 of *Lecture Notes in Computer Science*, pages 107–126. Springer Berlin Heidelberg, 1998.

[HPP+13] Christopher Henard, Mike Papadakis, Gilles Perrouin, Jacques Klein, and Yves Le Traon. Towards automated testing and fixing of re-engineered feature models. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 1245–1248, Piscataway, NJ, USA, 2013. IEEE Press.

[htt12] http://www.w3.org/TR/html5/. W3C, HTML5 Specification, 2012.

[HW79] Robert H. Hayes and Steven G. Wheelwright. The dynamics of process-product life cycles. *Harvard Business Review*, 1979.

[HXC12] A. Hubaux, Y. Xiong, and K. Czarnecki. A survey of configuration challenges in linux and ecos. In *VaMoS'12*, pages 149–155. ACM Press, 2012.

[Ins13] Software Engineering Institute. Software Product Lines. http://www.sei.cmu.edu/productlines/, 2013. [Online; accessed 18-August-2013].

[Jan10] Mikoláš Janota. *SAT Solving in Interactive Configuration*. PhD thesis, University College Dublin, November 2010.

[JBGMS09] Mikolás Janota, Goetz Botterweck, Radu Grigore, and João Marques-Silva. How to complete an interactive configuration process? *CoRR*, abs/0910.3913, 2009.

[Joh06] Isabel John. Capturing product line information from legacy user documentation. In Timo Kkla and JuanCarlos Duenas, editors, *Software Product Lines*, pages 127–159. Springer Berlin Heidelberg, 2006.

[JS10] R. Goncalo J. Saraiva J.C. Campos J.C. Silva, C. Silva. The guisurfer tool: towards a language independent approach to reverse engineering gui code. In *Proceedings of the 2nd ACM SIGCHI Symposium on Engineering interactive computing systems*, pages 181–186. ACM, 2010. ISBN: 978-1-4503-0083-4.

[KCH+90] K. Kang, S. Cohen, J. Hess, W. Nowak, and S. Peterson. Feature-oriented domain analysis (foda) feasibility study. Technical report, Software Engineering Institute, 1990.

[Kot95] Suresh Kotha. Mass customization: Implementing the emerging paradigm for competitive advantage. *Strategic Management Journal*, 16(S1):21–42, 1995.

[Kus00] Nicholas Kushmerick. Wrapper induction: efficiency and expressiveness. *Artif. Intell.*, 118(1-2):15–68, April 2000.

[KWD97] Nicholas Kushmerick, Daniel S. Weld, and Robert B. Doorenbos. Wrapper induction for information extraction. In *IJCAI (1)*, pages 729–737. Morgan Kaufmann, 1997.

[LGMK04] Kristina Lerman, Lise Getoor, Steven Minton, and Craig Knoblock. Using the structure of web sites for automatic segmentation of tables. In *Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, SIGMOD '04, pages 119–130, New York, NY, USA, 2004. ACM.

[LGZ03] Bing Liu, Robert Grossman, and Yanhong Zhai. Mining data records in web pages. In *Proceedings of the ninth ACM SIGKDD international conference on Knowledge discovery and data mining*, KDD '03, pages 601–606, New York, NY, USA, 2003. ACM.

[LMSM10] Alberto Lora-Michiels, Camille Salinesi, and Raúl Mazo. A Method Based on Association Rules to Construct Product Line Models. In *Proceedings of the Fourth International Workshop on Variability Modelling of Software-intensive Systems*, VaMoS '10, pages 147–150, 2010.

[LPH00] Ling Liu, C. Pu, and W. Han. Xwrap: An xml-enabled wrapper construction system for web information sources. In *Proceedings of the 16th International*

Conference on Data Engineering, ICDE '00, pages 611–, Washington, DC, USA, 2000. IEEE Computer Society.

[LRNdS02] Alberto H. F. Laender, Berthier Ribeiro-Neto, and Altigran S. da Silva. Debye - date extraction by example. *Data Knowl. Eng.*, 40(2):121–154, February 2002.

[LRNdST02] Alberto H. F. Laender, Berthier A. Ribeiro-Neto, Altigran S. da Silva, and Juliana S. Teixeira. A brief survey of web data extraction tools. *SIGMOD Rec.*, 31(2):84–93, June 2002.

[MBC09] Marcilio Mendonca, Moises Branco, and Donald Cowan. S.p.l.o.t.: software product lines online tools. In *Proceedings of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications*, OOPSLA '09, pages 761–762, New York, NY, USA, 2009. ACM.

[MCC12] Josip Maras, Jan Carlson, and Ivica Crnkovi. Extracting client-side web application code. In *Proceedings of the 21st international conference on World Wide Web*, WWW '12, pages 819–828, New York, NY, USA, 2012. ACM.

[MCHB11a] Raphael Michel, Andreas Classen, Arnaud Hubaux, and Quentin Boucher. A formal semantics for feature cardinalities in feature diagrams. In *Proceedings of the 5th Workshop on Variability Modeling of Software-Intensive Systems*, VaMoS '11, pages 82–89, New York, NY, USA, 2011. ACM.

[MCHB11b] Raphael Michel, Andreas Classen, Arnaud Hubaux, and Quentin Boucher. A formal semantics for feature cardinalities in feature diagrams. In *VaMoS'11*, pages 82–89. ACM, 2011.

[MHL03] Xiaofeng Meng, Dongdong Hu, and Chen Li. Schema-guided wrapper maintenance for web-data extraction. In *Proceedings of the 5th ACM international workshop on Web information and data management*, WIDM '03, pages 1–8, New York, NY, USA, 2003. ACM.

[MMK99] Ion Muslea, Steve Minton, and Craig Knoblock. A hierarchical approach to wrapper induction. In *Proceedings of the third annual conference on Autonomous Agents*, AGENTS '99, pages 190–197, New York, NY, USA, 1999. ACM.

[MMK01] Ion Muslea, Steven Minton, and Craig A. Knoblock. Hierarchical wrapper induction for semi-structured information sources. *Autonomous Agents and Multi-Agent Systems*, 4(1-2):93–114, March 2001.

[MvDL12] Ali Mesbah, Arie van Deursen, and Stefan Lenselink. Crawling ajax-based web applications through dynamic analysis of user interface state changes. *ACM Trans. Web*, 6(1):3:1–3:30, March 2012.

[NJ02] Andrew Nierman and H. V. Jagadish. Evaluating Structural Similarity in XML Documents. In *Proceedings of the Fifth International Workshop on the Web and Databases (WebDB 2002)*, 2002.

[NRBM13] Bao Nguyen, Bryan Robbins, Ishan Banerjee, and Atif Memon. Guitar: an innovative tool for automated testing of gui-driven software. *Automated Software Engineering*, pages 1–41, 2013.

[OLN06] S. K. Ong, Q. Lin, and A. Y. C. Nee. Web-based configuration design system for product customization. *International Journal of Production Research*, 44(2):351–382, 2006.

[PBvdL05] Klaus Pohl, Günter Böckle, and Frank J. van der Linden. *Software Product Line Engineering: Foundations, Principles and Techniques.* Springer-Verlag New York, Inc., 2005.

[PCMA07] Reshma Patel, Frans Coenen, Russell Martin, and Lawson Archer. Reverse engineering of web applications: A technical review, 2007.

[PHH05] Xuan-Hieu Phan, Susumu Horiguchi, and Tu-Bao Ho. Automated data extraction from the web with conditional models. *Int. J. Bus. Intell. Data Min.*, 1(2):194–209, December 2005.

[Pin93] B. Joseph Pine. *Mass Customization: The New Frontier in Business Competition.* Harvard Business School Press, 1993.

[RGSL04] D. C. Reis, P. B. Golgher, A. S. Silva, and A. F. Laender. Automatic web news extraction using tree edit distance. In *Proceedings of the 13th international conference on World Wide Web*, WWW '04, pages 502–511, New York, NY, USA, 2004. ACM.

[RNLS99] Berthier Ribeiro-Neto, Alberto H. F. Laender, and Altigran S. Da Silva. Extracting semi-structured data through examples. In *In Proceedings of the International Conference on Knowledge Management*, pages 94–101, 1999.

[RP04] Timm Rogoll and Franck Piller. Product configuration from the customer's perspective: A comparison of configuration systems in the apparel industry. In *International Conference on Economic, Technical and Organisational aspects of Product Configuration Systems*, 2004.

[RT01] Filippo Ricca and Paolo Tonella. Understanding and restructuring web sites with reweb. *IEEE MultiMedia*, 8(2):40–51, April 2001.

[RvdADtH08] Marcello La Rosa, Wil M.P. van der Aalst, Marlon Dumas, and Arthur H.M. ter Hofstede. Questionnaire-based variability modeling for system configuration. *Software and Systems Modeling*, 8(2):251–274, 2008.

[SA99] Arnaud Sahuguet and Fabien Azavant. Building light-weight wrappers for legacy web data-sources using w4f. In *Proceedings of the 25th International Conference on Very Large Data Bases*, VLDB '99, pages 738–741, San Francisco, CA, USA, 1999. Morgan Kaufmann Publishers Inc.

[SA01] Arnaud Sahuguet and Fabien Azavant. Building intelligent web applications using lightweight wrappers. *Data Knowl. Eng.*, 36(3):283–316, March 2001.

[SBFF09] C. Streichsbier, P. Blazek, F. Faltin, and W. Fruhwirt. Are *de facto* Standards a Useful Guide for Designing Human-Computer Interaction Processes? The Case of User Interface Design for Web-based B2C Product Configurators. In *HICSS '09. 42nd Hawaii International Conference on System Sciences*, pages 1–7, 2009.

[SHT06] Pierre-Yves Schobbens, Patrick Heymans, and Jean-Christophe Trigaux. Feature diagrams: A survey and a formal semantics. In *Proceedings of the 14th IEEE International Requirements Engineering Conference*, RE '06, pages 136–145, Washington, DC, USA, 2006. IEEE Computer Society.

[SLB+11] Steven She, Rafael Lotufo, Thorsten Berger, Andrzej Wasowski, and Krzysztof Czarnecki. Reverse engineering feature models. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 461–470, New York, NY, USA, 2011. ACM.

[SLRS12] Martin Schäler, Thomas Leich, Marko Rosenmüller, and Gunter Saake. Building information system variants with tailored database schemas using features. In *CAiSE'12*, pages 597–612. Springer-Verlag, 2012.

[Sod99] Stephen Soderland. Learning information extraction rules for semi-structured and free text. *Mach. Learn.*, 34(1-3):233–272, February 1999.

[SS09] Carmen Santoro and Lucio Davide Spano. MARIA: A universal, declarative, multiple abstraction-level language for service-oriented applications in ubiquitous environments. *ACM Transactions on Computer-human Interaction*, 16:1–30, 2009.

[Sto12] Reinhard Stoiber. *A New Approach to Product Line Engineering in Model-Based Requirements Engineering.* PhD thesis, University of Zurich, 2012.

[Str04] Detlef Streitferdt. *Family-oriented requirements engineering.* PhD thesis, Technical University of Ilmenau, 2004.

[SW98] D. Sabin and R. Weigel. Product configuration frameworks-a survey. *Intelligent Systems and their Applications, IEEE*, 13(4):42–49, 1998.

[TH01] Scott Tilley and Shihong Huang. Evaluating the reverse engineering capabilities of web tools for understanding site content and structure: a case study. In *Proceedings of the 23rd International Conference on Software Engineering*, ICSE '01, pages 514–523, Washington, DC, USA, 2001. IEEE Computer Society.

[Tip95] F. Tip. A survey of program slicing techniques. *JOURNAL OF PROGRAMMING LANGUAGES*, 3:121–189, 1995.

[TJ07] Mitchell M. Tseng and Jianxin Jiao. *Mass Customization*, pages 684–709. John Wiley and Sons, Inc., 2007.

[TKB+12] Thomas Thm, Christian Kstner, Fabian Benduhn, Jens Meinicke, Gunter Saake, and Thomas Leich. Featureide: An extensible framework for feature-oriented software development. *Science of Computer Programming*, (0):–, 2012.

[TP03] Mitchell M. Tseng and Frank Piller. The customer centric enterprise - advances in mass customization and personalization, 2003.

[TPF12] Alessio Trentin, Elisa Perin, and Cipriano Forza. Sales configurator capabilities to prevent product variety from backfiring. In *Workshop on Configuration (ConfWS)*, 2012.

[Tra05] Porfirio Tramontana. *Reverse Engineering Web Applications*. PhD thesis, University of Naples, Federico II, Italy, 2005.

[VBS01] Jean Vanderdonckt, Laurent Bouillon, and Nathalie Souchon. Flexible reverse engineering of web pages with vaquista. In *Proceedings of the Eighth Working Conference on Reverse Engineering (WCRE'01)*, WCRE '01, pages 241–, Washington, DC, USA, 2001. IEEE Computer Society.

[vH01] Eric von Hippel. User toolkits for innovation. *Product Innovation Management*, 2001.

[vHK02] Eric von Hippel and Ralph Katz. Shifting innovation to users via toolkits. *Management Science*, 48(7):821–833, 2002.

[Vis04a] Eelco Visser. Program transformation with stratego/xt. In Christian Lengauer, Don Batory, Charles Consel, and Martin Odersky, editors, *Domain-Specific Program Generation*, volume 3016 of *Lecture Notes in Computer Science*, pages 216–238. Springer Berlin Heidelberg, 2004.

[Vis04b] Saraiva Joao Visser, Eelco. Tutorial on strategic programming across programming paradigms. In *8th Brazilian Symposium on Programming Languages*, 2004.

[WCR09] Nathan Weston, Ruzanna Chitchyan, and Awais Rashid. A framework for constructing semantically composable feature models from natural language requirements. In *Proceedings of the 13th International Software Product Line Conference*, SPLC '09, pages 211–220, Pittsburgh, PA, USA, 2009. Carnegie Mellon University.

[WL02] Jiying Wang and Frederick H. Lochovsky. Wrapper induction based on nested pattern discovery. Technical Report HKUST-CS-27-02, Department of Computer Science, University of Science and Technology Clear Water Bay, Kowloon Hong Kong, 2002.

[WL03] Jiying Wang and Fred H. Lochovsky. Data extraction and label assignment for web databases. In *Proceedings of the 12th international conference on World Wide Web*, WWW '03, pages 187–196, New York, NY, USA, 2003. ACM.

[XHK05] H. Xie *, P. Henderson, and M. Kernahan. Modelling and solving engineering product configuration problems by constraint satisfaction. *International Journal of Production Research*, 43(20):4455–4469, 2005.

[ZFdSZ12] Tewfik Ziadi, Luz Frias, Marcos Aurelio Almeida da Silva, and Mikal Ziane. Feature identification from the source code of product variants. *2011 15th European Conference on Software Maintenance and Reengineering*, 0:417–422, 2012.

[ZL05] Yanhong Zhai and Bing Liu. Web data extraction based on partial tree alignment. In *Proceedings of the 14th international conference on World Wide Web*, WWW '05, pages 76–85, New York, NY, USA, 2005. ACM.

[ZSWG09] Shuyi Zheng, Ruihua Song, Ji-Rong Wen, and C. Lee Giles. Efficient record-level wrapper induction. In *Proceedings of the 18th ACM conference on Information and knowledge management*, CIKM '09, pages 47–56, New York, NY, USA, 2009. ACM.