DOCTOR OF SCIENCES

Reverse Engineering Web Configurators

Abbasi, Ebrahim Khalil

Award date:
2014

Awarding institution:
University of Namur

PReCISE Research Centre

Faculty of Computer Science

University of Namur

Reverse Engineering Web Configurators

Ebrahim Khalil Abbasi

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

in the

PReCISE Research Centre
Faculty of Computer Science
University of Namur

March 2014

Jury

Prof. Pierre-Yves Schobbens, University of Namur, Belgium (chair)

Prof. Kim Mens, Catholic University of Louvain, Belgium

Prof. Mathieu Acher, University of Rennes, France

Prof. Anthony Cleve, University of Namur, Belgium

Prof. Patrick Heymans, University of Namur, Belgium (advisor)

Abstract

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

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

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

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

Résumé

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

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

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

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

Acknowledgements

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

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

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

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

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

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

Contents

List of Figures

List of Tables

Abbreviations

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

I dedicate this thesis to
my lovely Leila and my wonderful family
for their constant love and support.
I love you all dearly.

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 [Abbasi et al., 2013]) 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*.

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:

$$candidateCodeFragment[0].mappingIndexElement = td$$

```
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.

```
<table class="tagSelect">
    <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>
          </h6>
          <p>
              <span> A…Z 0…9 °"'*.#?$!:-+=()&,\/ @ </span>
                  © ★ † ♥
              <span> ♣ </span>
           </p>
      </td>
      <td></td>
  </tr>
  <tr>
      <td>
          <input type="radio"  id="dt-std-br" checked="checked" value="br" name="t1style"/>
      </td>
      <td>
          <label for="dt-std-br">
              <img title="Brass Dogtog " src="images/site/dt-std-br_sm.png">
          </label>
      </td>
      <td>
          <h6>
              <label for = "dt-std-br"> Brass Dogtog</label>
          </h6>
          <span>$8.99</span>
      </td>
      <td></td>
  </tr>
  <tr>
      <td>
        <div>
              <input type="radio"  id="dt-std-black" checked="checked" value="black" name="t1style"/>
        </div>
      </td>
      <td>
          <label for="dt-std-black">
              <img title="Black Dogtog " src="images/site/dt-std-black_sm.png">
          </label>
      </td>
      <td>
          <h6>
              <label for = "dt-std-black"> Black Dogtog</label>
          </h6>
          <p>
              <span> Cannot be debossed. </span>
           </p>
      </td>
      <td></td>
  </tr>
</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.

$$candidateCodeFragment[0].mappedIndexElement = td$$

$$candidateCodeFragment[1].mappingIndexElement = td$$

$$candidateCodeFragment[1].mappedIndexElement = td$$

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.

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.

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*

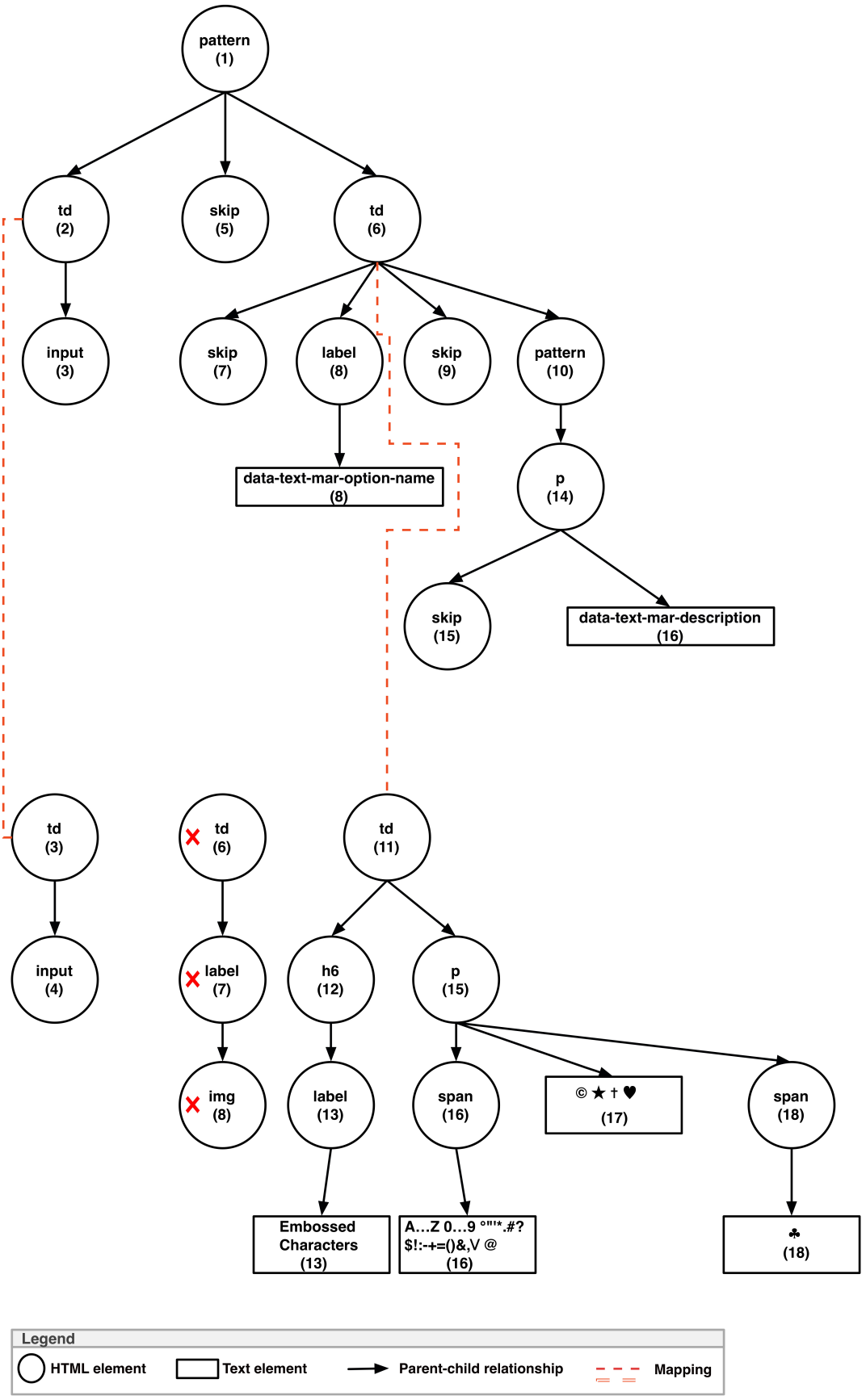

FIGURE 7.4: Tree representation (1).

```
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.

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 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).

```
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.

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, **findMatchingPattern** 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 **dataExtractionProcedure**, the data extraction process completes.

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

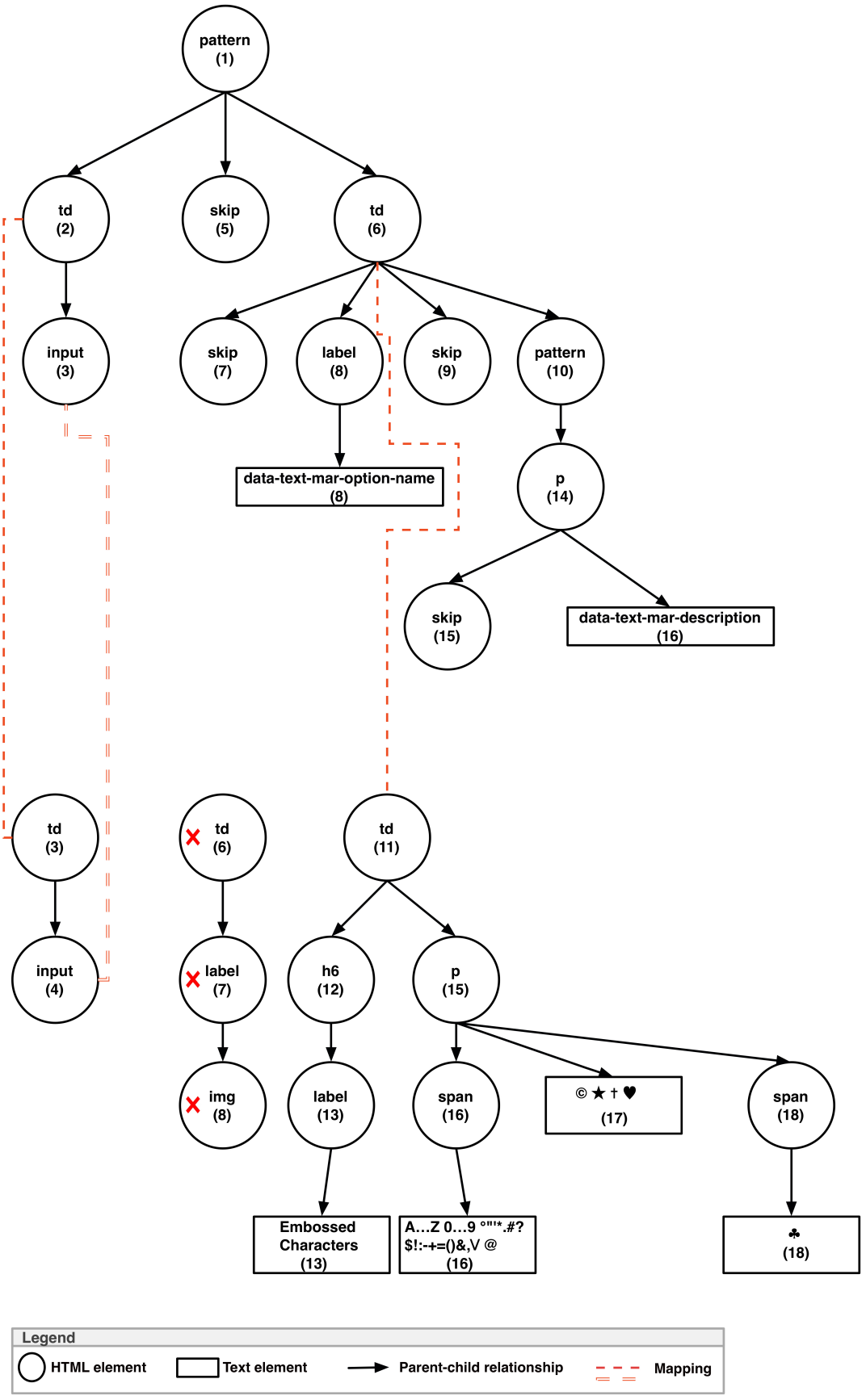

FIGURE 7.7: Tree representation (2).

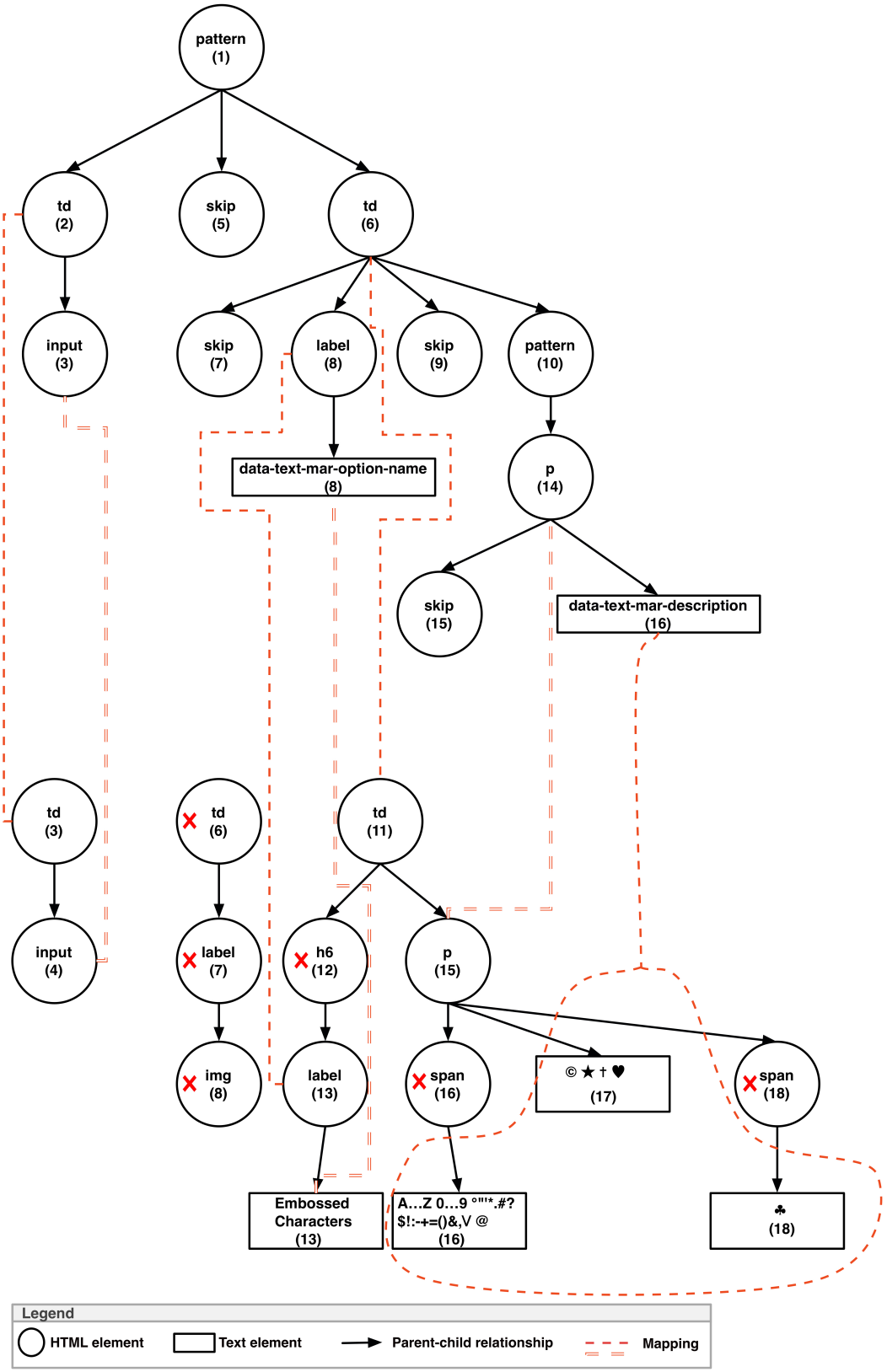

FIGURE 7.8: Tree representation (3).

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 an uppercase letter and can contain numbers as well as the underscore. If an option

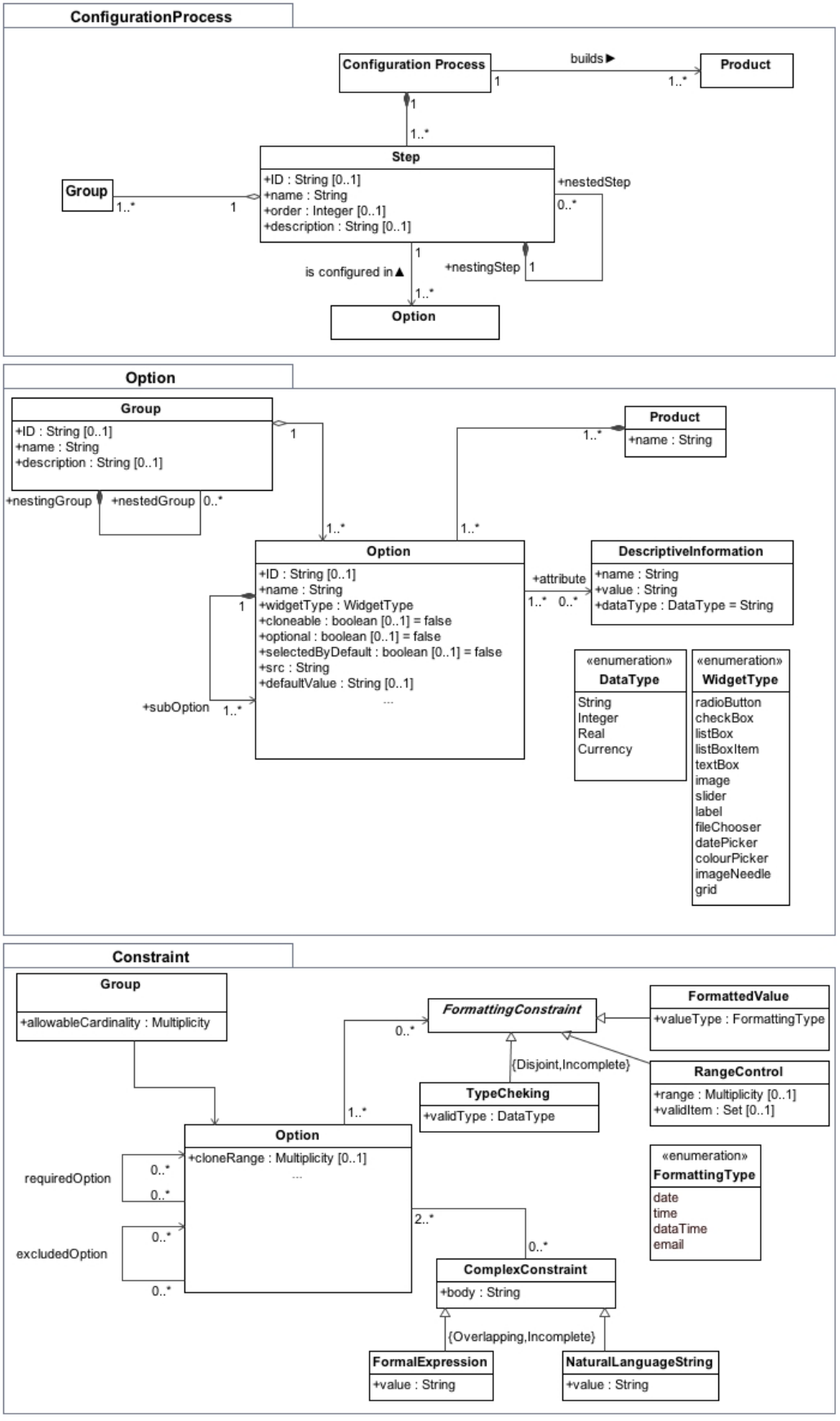

FIGURE 7.9: Schema of output data.

```
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.

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 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

[2] http://getfirebug.com/

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.

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.

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

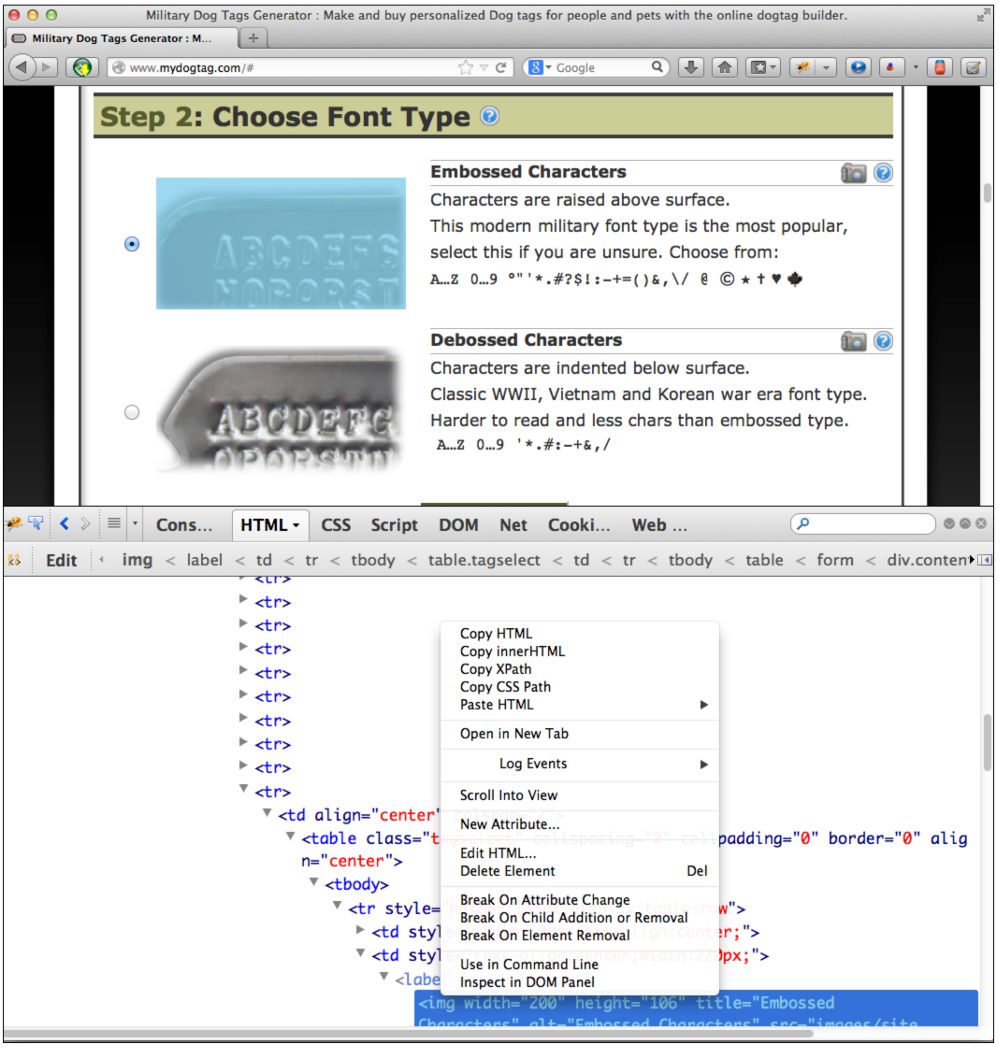

FIGURE 7.11: Firebug.

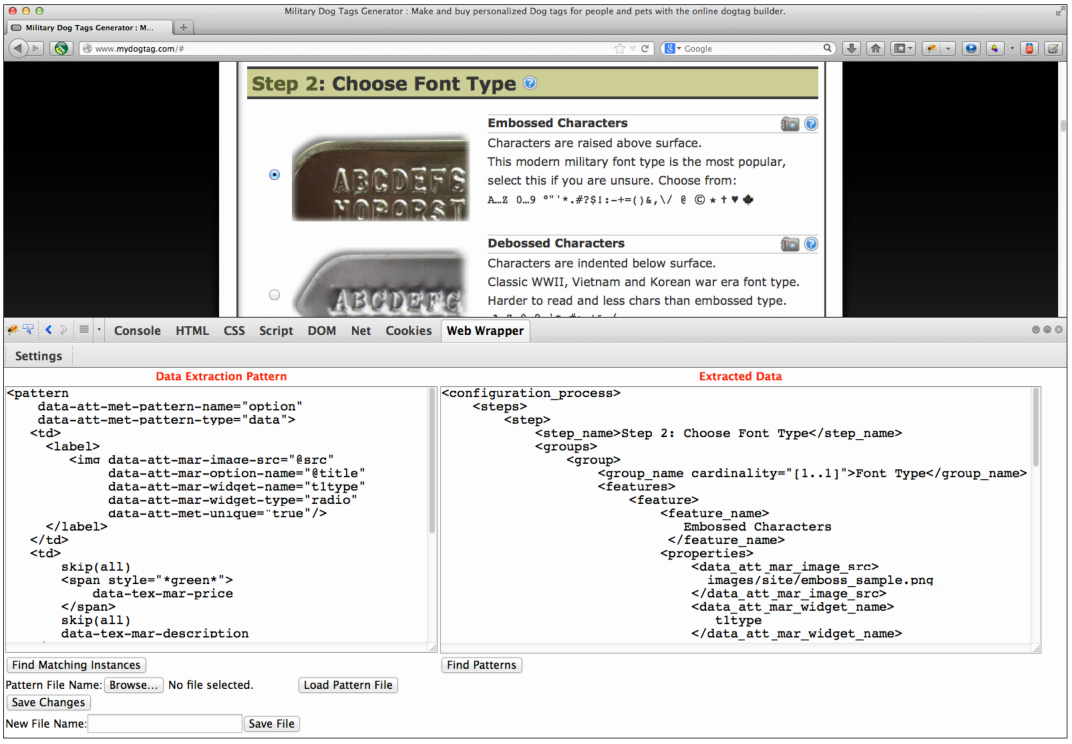

FIGURE 7.12: Web Wrapper extension.

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 8.3 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

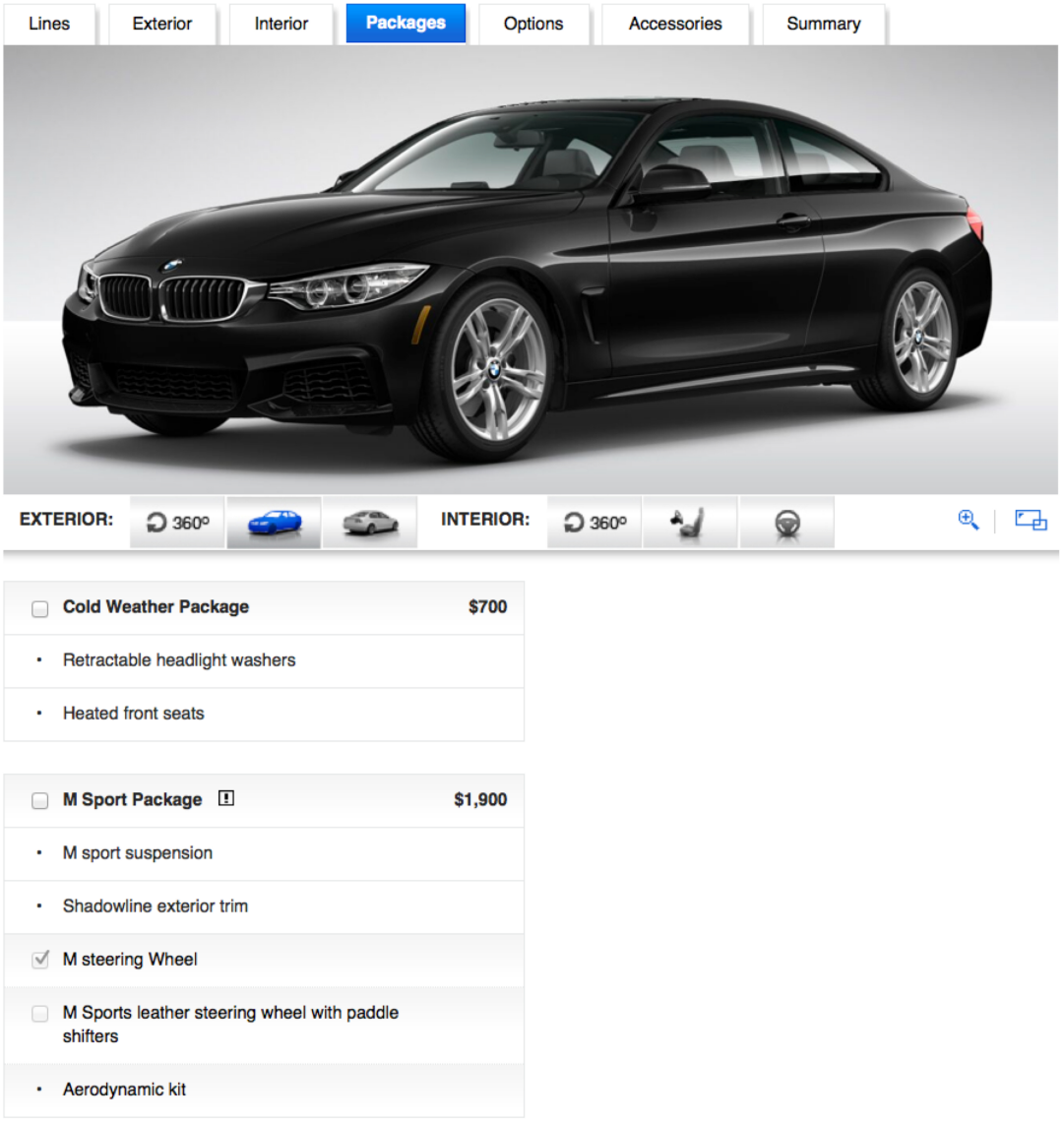

FIGURE 8.1: Parent-child relationship between objects (`http://www.bmwusa.com/`, October 22 2013).

respectively denote the *parentOptionData*, *childOptionData*, and *childOptionSelectableData* 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*.

```
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.

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).

```
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.

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).

```
<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.

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 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 generic and 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 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

```
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.

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. 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

(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.

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.

```
<?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>
            . . .
            . . .
          </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.

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 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 (!). 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 (! 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).

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* 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 maxlength formatting constraint defined for input elements in Figure 8.9.

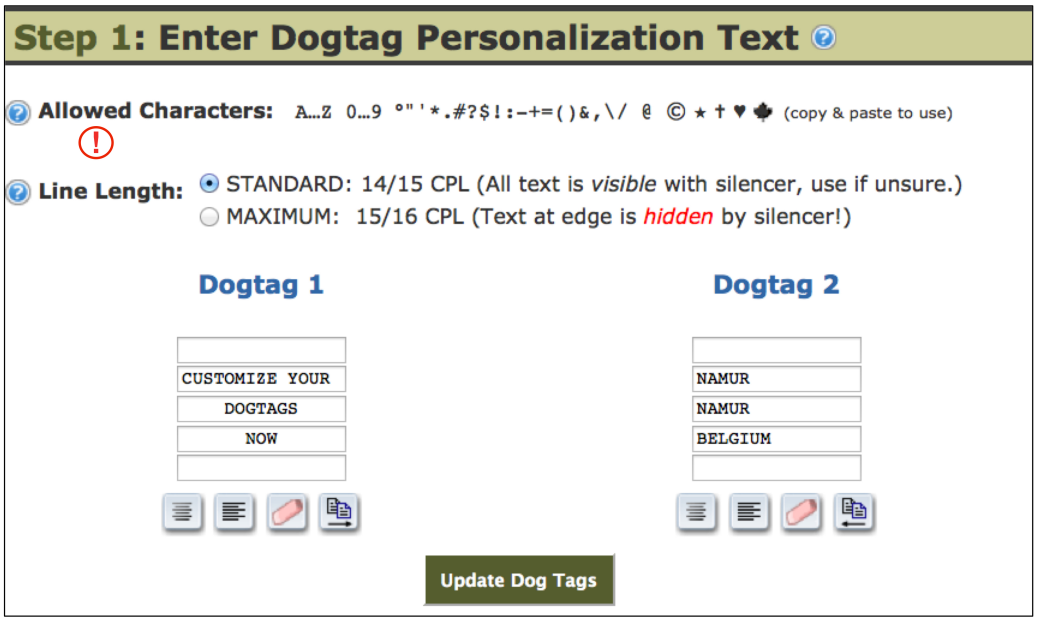

FIGURE 8.9: Textual formatting constraint (`http://www.mydogtag.com/`, June 13 2013).

```
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 constraints 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         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.

`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 (`http://www.bluenile.com`, February 24 2014).

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".

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.

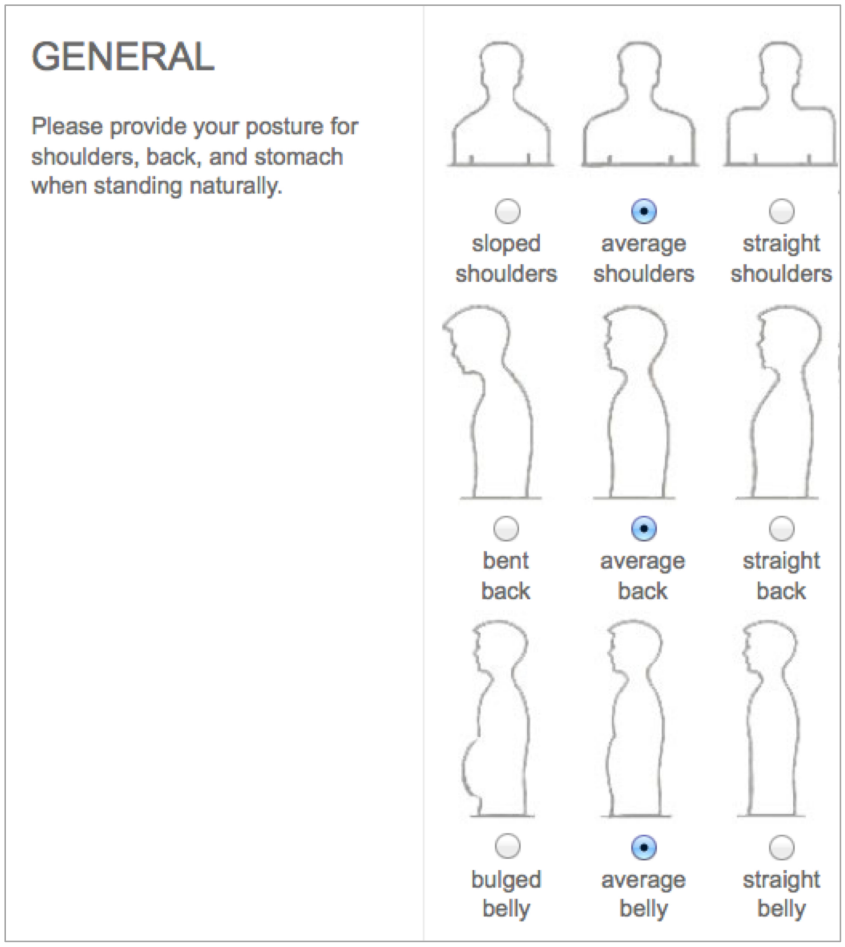

FIGURE 8.13: Three groups of options presented using radio buttons (`http://www.shirtsmyway.com/`, October 24 2013).

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.

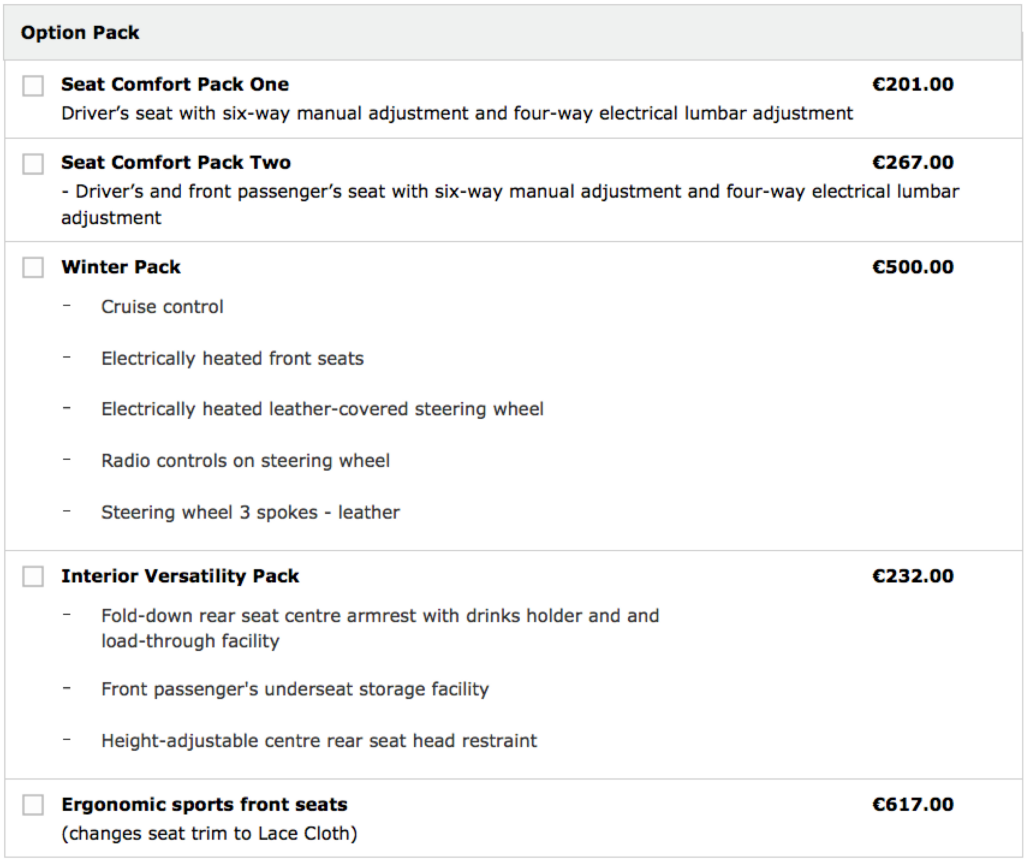

FIGURE 8.14: Cross-cutting constraints displayed in the GUI (`http://www.opel.ie/`, February 24 2014).

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 48 to 61 show that by selecting "Aragon 18" in the "Manufacturer" list box (line 50) the following items are loaded to the "Model" list box (lines 55-59): "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

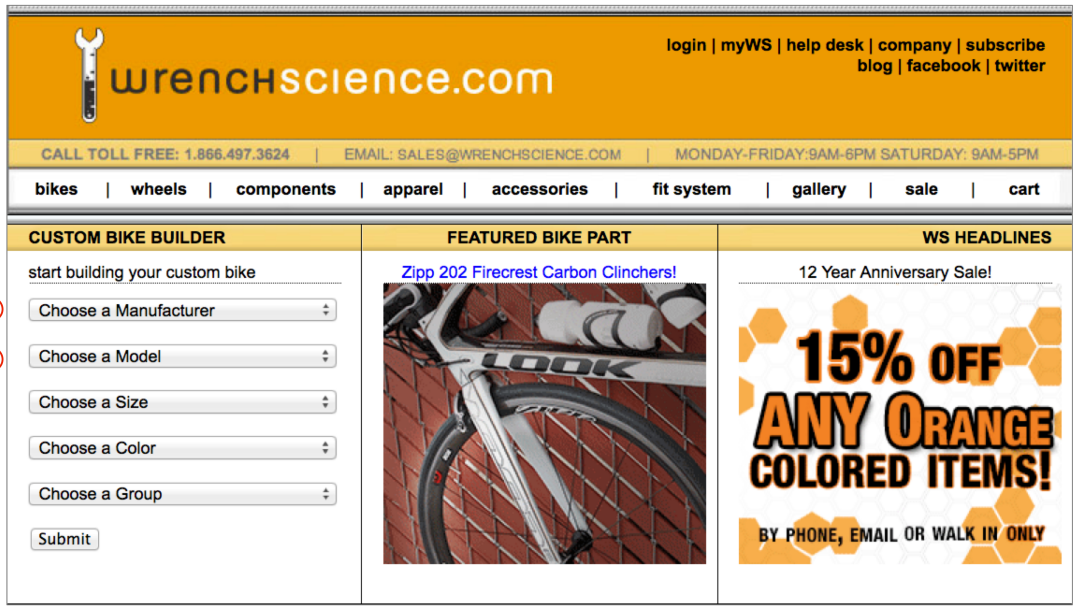

FIGURE 8.15: Independent and dependent options (`http://www.wrenchscience.com/`, February 24 2014).

```
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.

```
<?xml version="1.0" ?>
<configuration_process>
  <steps xmlns="http://www.mozilla.org/keymaster/gatekeeper/there.is.only.xul">
    <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>Liteville</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.

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 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.

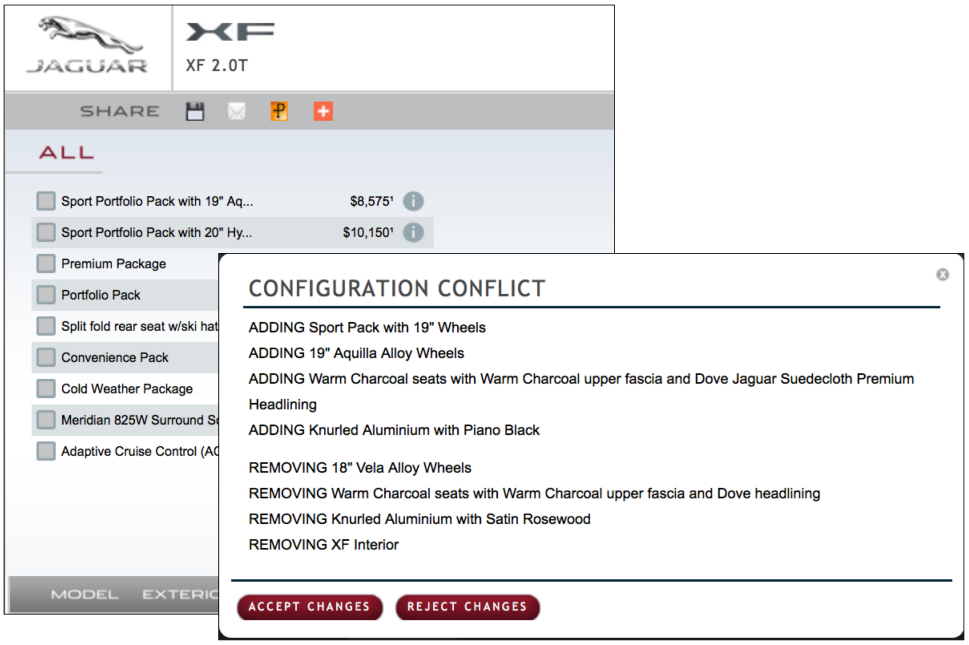

FIGURE 8.18: Controlled decision propagation (`http://www.jaguarusa.com/`, July 31 2013).

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 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 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).

```
<pattern data-att-met-pattern-type="region"      data-att-met-pattern-name="OptionsRegion"
      data-att-met-root-pattern="true"       data-att-met-dependent-pattern="conflictResolutionRegion"
      data-att-mar-step-name ="Options"          data-att-mar-group-name ="Options">
      <li class="listPanel">
            <pattern>
                  optionsData
            </pattern>
      </li>
</pattern>

<pattern data-att-met-pattern-name="optionsData" data-att-met-pattern-type="data">
      <a class="with|without" data-att-mar-full-name="@title" data-att-mar-widget-type="checkbox"
                  data-att-met-unique="true"     data-att-met-clickable="true">
        data-tex-mar-option-name
      </a>
      <span class="price">data-tex-mar-price</span>
</pattern>

<pattern data-att-met-pattern-type="region" data-att-met-pattern-name="conflictResolutionRegion">
<div id="sb-content">
            <pattern>
                  conflictResolutionData
            </pattern>
</div>
</pattern>

<pattern data-att-met-pattern-name="conflictResolutionData" data-att-met-pattern-type="data">
  <div class="conflict-resolution" data-att-met-unique="true">
    skip(all)
    <pattern>auxiliaryData</pattern>
  </div>
</pattern>

<pattern data-att-met-pattern-name="auxiliaryData" data-att-met-pattern-type="auxiliary">
        <li>
          data-tex-mar-constraint
        </li>
</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 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 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

```
<?xml version="1.0" ?>
<configuration_process>
  <steps xmlns="http://www.mozilla.org/keymaster/gatekeeper/there.is.only.xul">
    <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.

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., [Acher et al., 2013b; Haslinger et al., 2013]) 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. Moreover, in many cases it might simply be unrealistic to gather all valid configurations in an acceptable time since their number is in the order of 2^n where n is the number of options.

In this PhD, we tackle a simpler 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 solution 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).

```
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.

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 - buckle activated" 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.

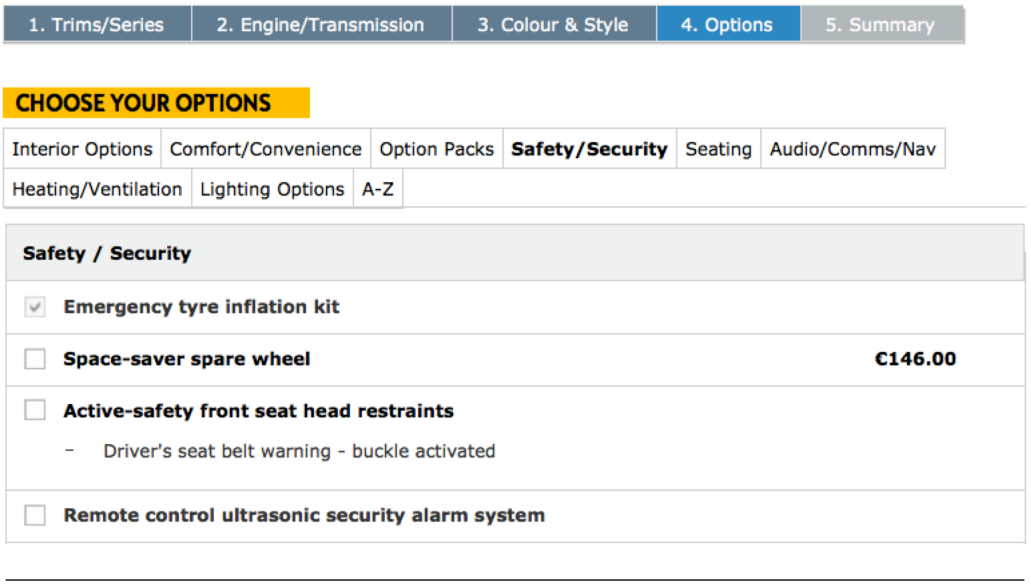

FIGURE 8.22: An example configuration environment (`http://www.opel.ie/`, August 3 2013).

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

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

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

```
<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">
      <form class="safety">
            <pattern>
                  independentData
            </pattern>
      </form>
</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">
      <form 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-configuration-state="@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.

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

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

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 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) 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". We plan to integrate our tool with FAMILIAR to more accurately infer constraints.

```
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 *generality* of our approach. By generality 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.
- *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.

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

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.

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

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

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	$A_{manual} \simeq 9\%$

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

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

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 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 ($M9$).

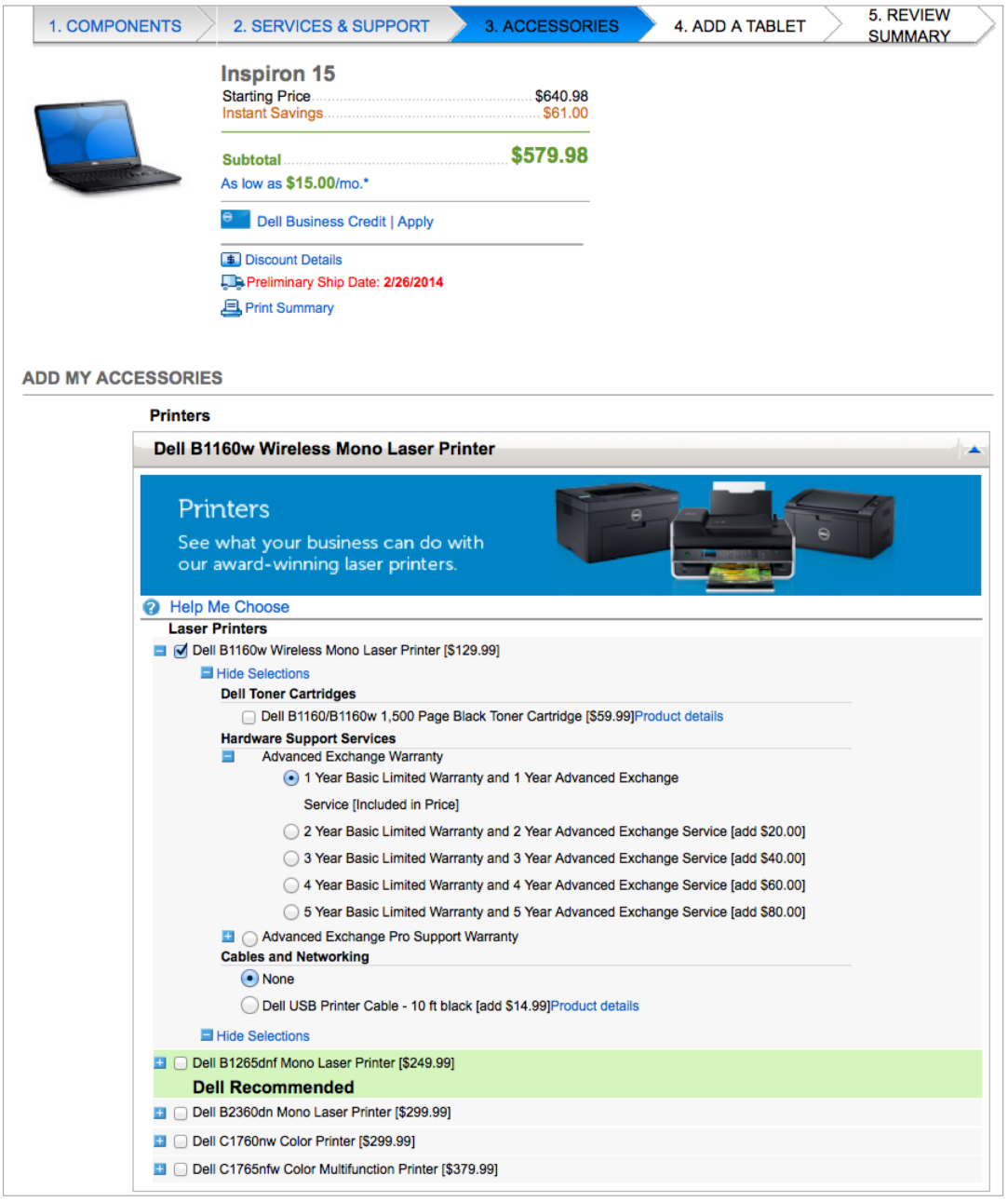

FIGURE 9.1: Dell's laptop configurator (`http://www.dell.com`, January 5 2014).

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 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.

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. 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.

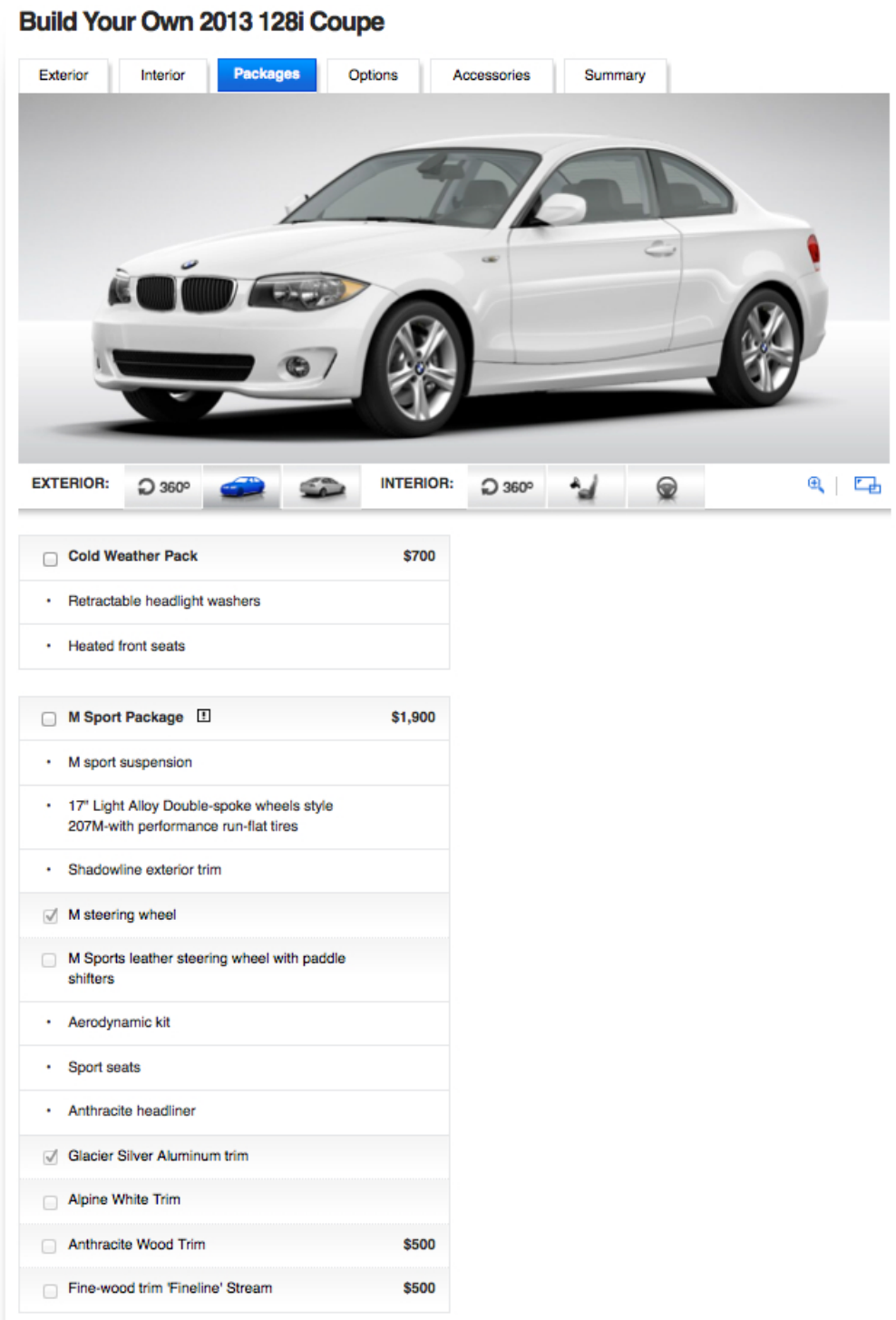

FIGURE 9.2: BMW's car configurator (`http://www.bmwusa.com`, January 5 2014).

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,

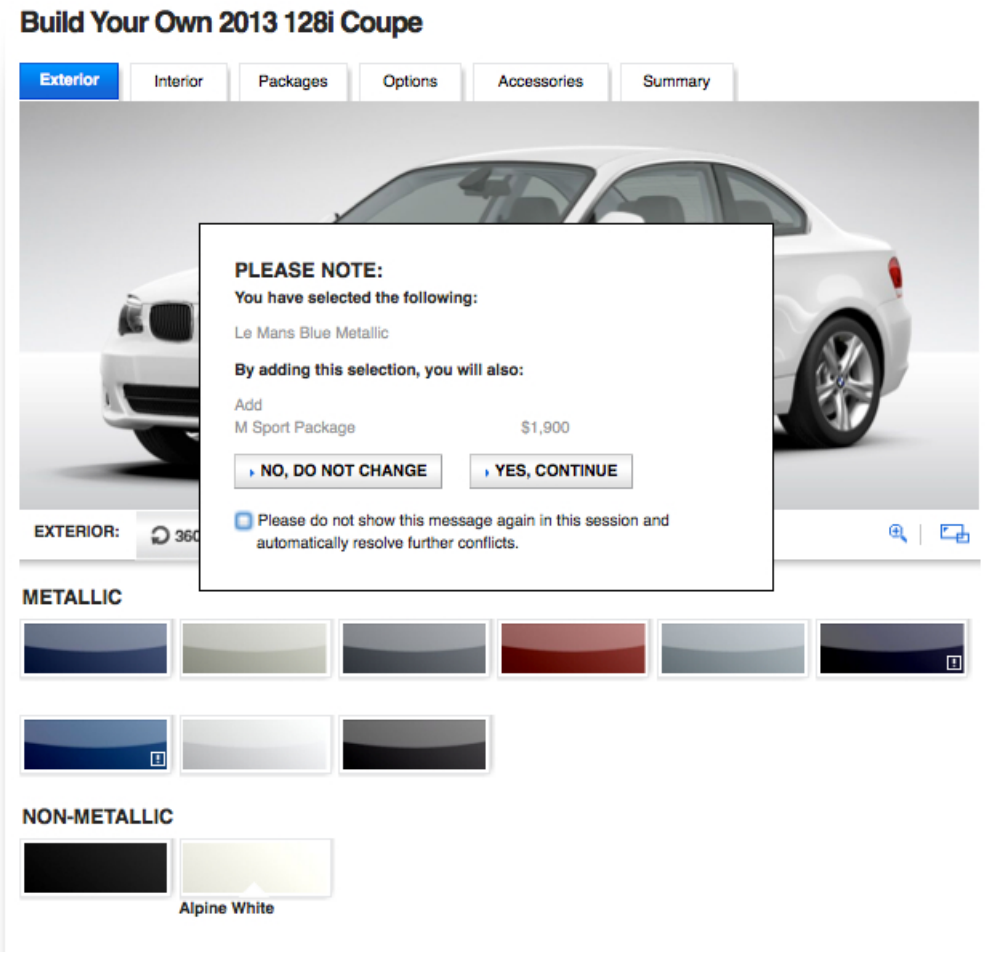

FIGURE 9.3: *controlled* decision propagation strategy in BMW's car configurator (`http://www.bmwusa.com`, January 5 2014).

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

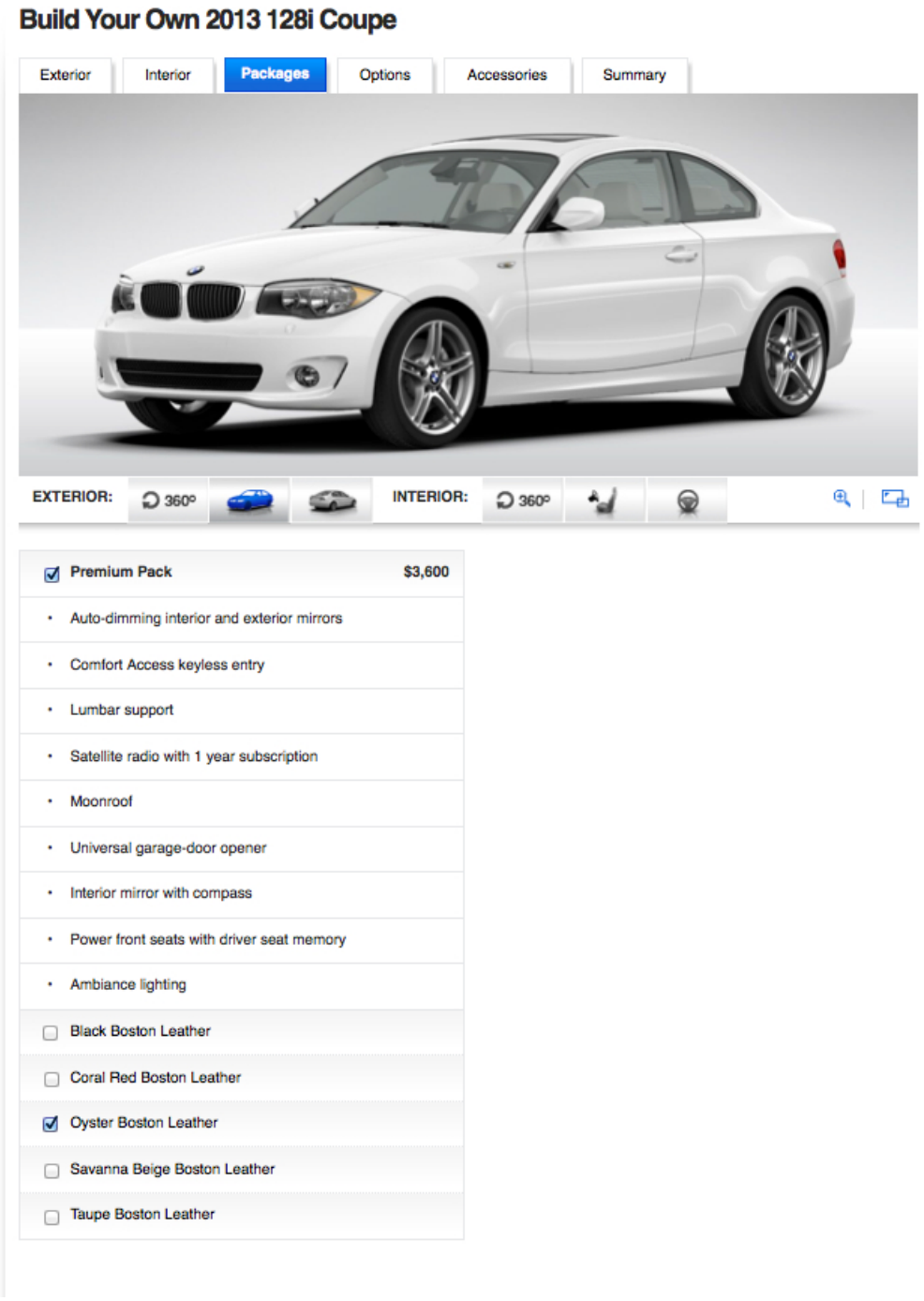

FIGURE 9.4: Configuration state changes in BMW's car configurator (`http://www.bmwusa.com`, January 5 2014).

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 (i.e., activating its corresponding menu) its contained ingredients are presented to the customer and other groups become hidden. We defined

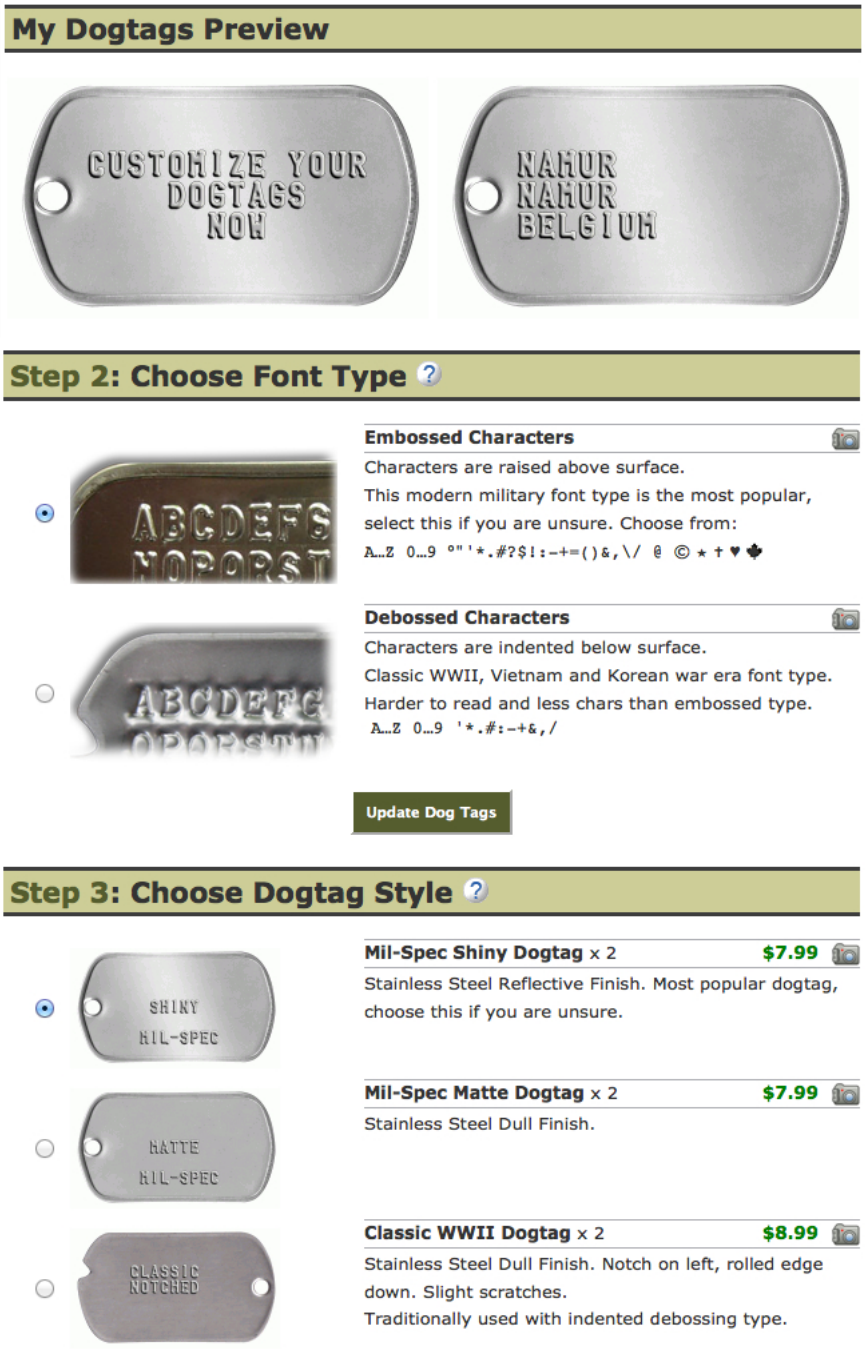

FIGURE 9.5: Dog-tag generator (`http://www.mydogtag.com`, January 5 2014).

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

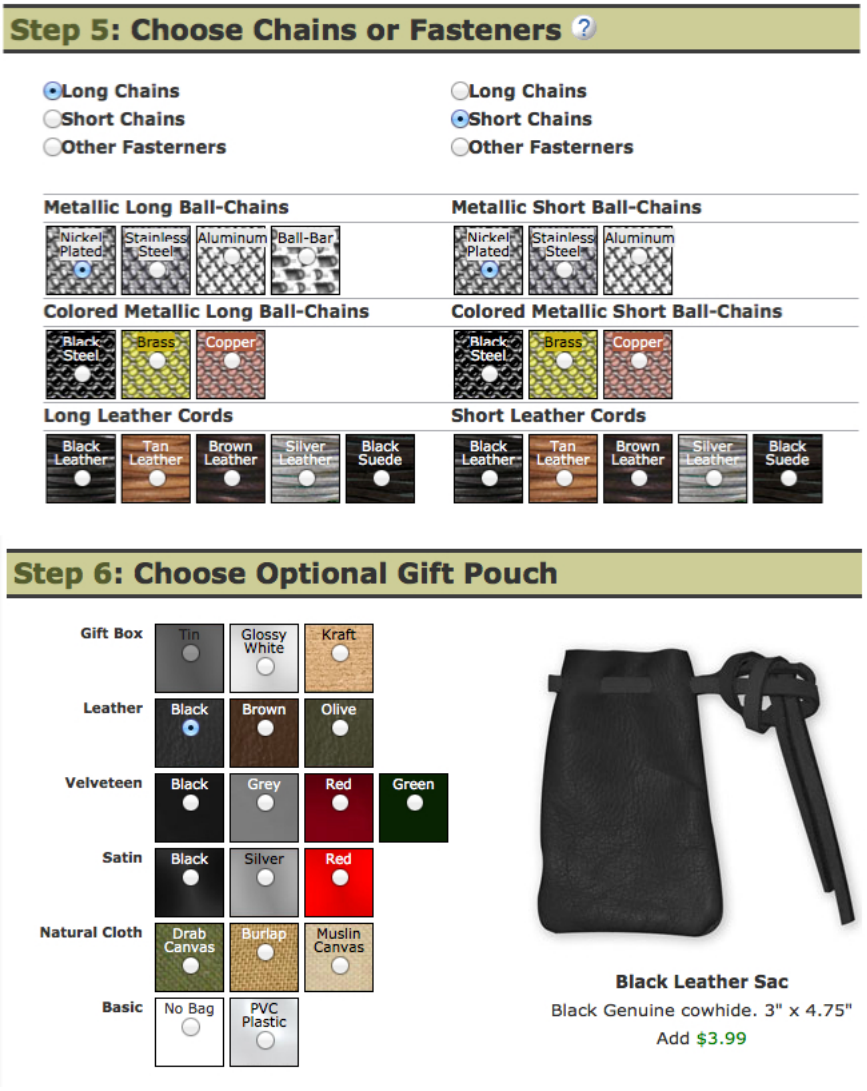

FIGURE 9.6: Dynamic data in Dog-tag generator (`http://www.mydogtag.com`, January 5 2014).

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) to extract groups and their contained options. However, we noticed

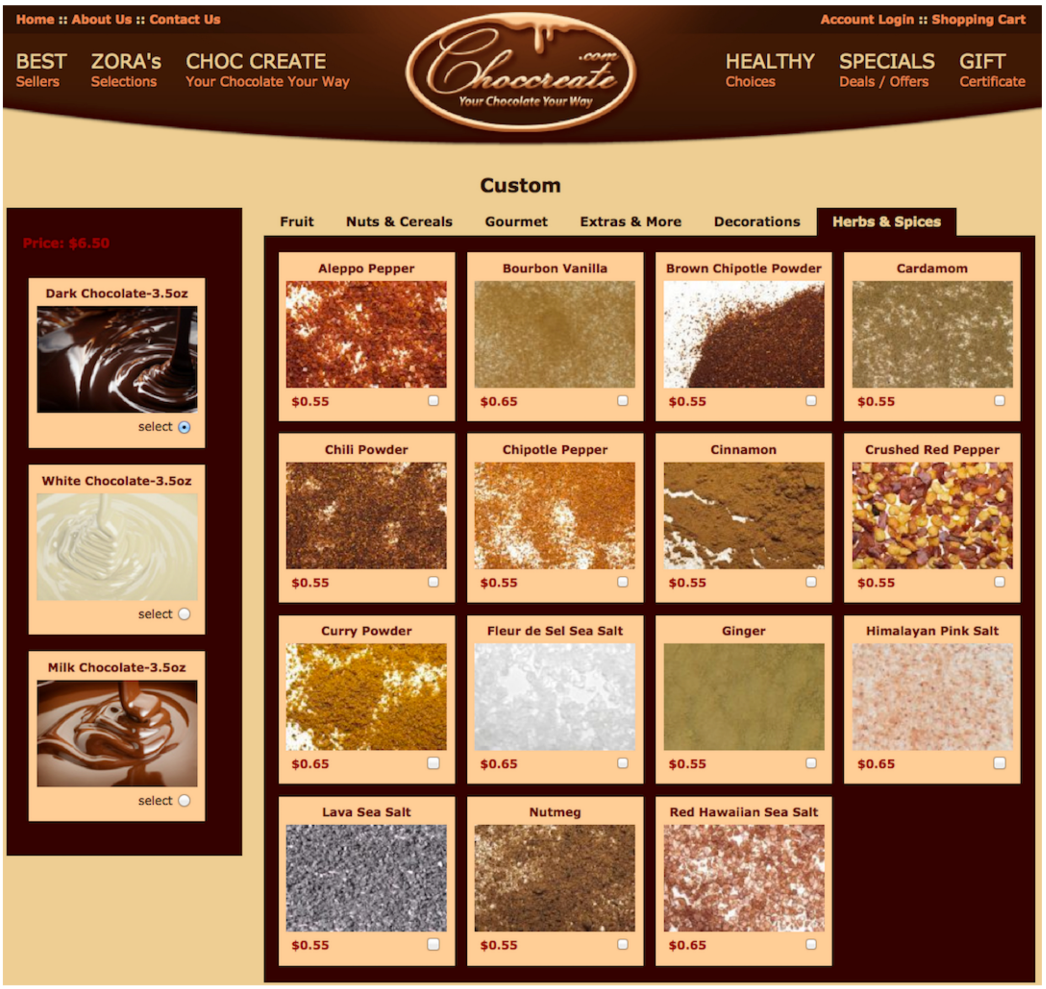

FIGURE 9.7: Chocolate maker (`http://www.choccreate.com`, January 5 2014).

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 ($M8$).

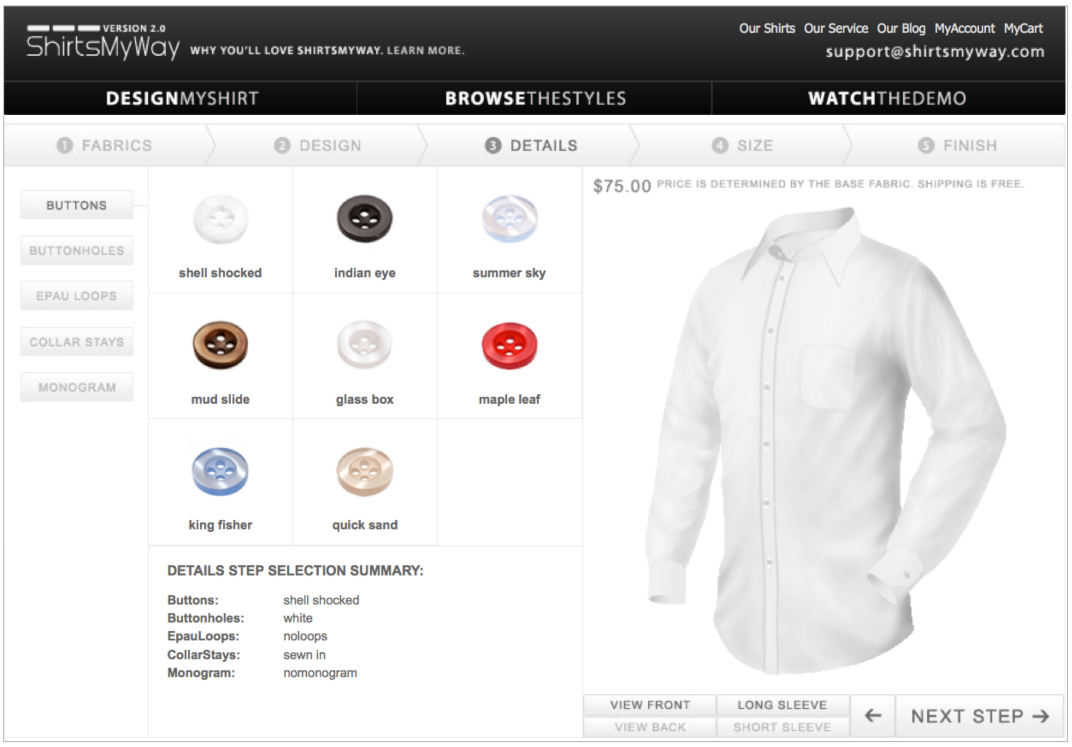

FIGURE 9.8: Shirt designer (`http://www.shirtsmyway.com`, January 6 2014).

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 [Acher et al., 2013a; Davril et al., 2013; She et al., 2011] 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 [Henard et al., 2013].

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.

TABLE 9.6: The time spent for writing patterns.

System	Time (mins)
S1	30
S2	100
S3	85
S4	25
S5	110
Total	350

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). Although this was not formally measured, we deem it important to indicate that the total amount of time spent by the author of the thesis to inspect the source code of the example configurators and write the required patterns was approximately 350 minutes (see Table 9.6 for details). The time needed for post-processing activities, e.g., adding missing data, removing noisy data, and generating TVL files is not included.

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 [Chang et al., 2006] 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 generality 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 explosion 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, i.e., 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 good practices we identified in our empirical study can be used in the forward-engineering process to improve the usability of the new configurators. Examples of such practices are: guided consistency checking, self-explanatory configuration process, stateful backward navigation, etc.

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 generic 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 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 support the extraction of 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 generality 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 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). We are working on a template-induction approach to deduce the template from which data objects in a page are generated.

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.

Only the template-based Web configurators are investigated. Our approach relies on the basic assumption that pages in a Web configurator are generated from a number of templates. There may be, however, Web pages that have unstructured data objects. In such cases, we cannot apply our approach.

The Cascading Style Sheets (CSS) language is not fully supported. We developed techniques that target static structure (i.e., the HTML code) and dynamic behaviour (handled by client scripting languages such as JavaScript and/or server-side technologies such as PHP) of Web configurators to find and extract configuration-specific data. CSS used to control the style and layout of a Web page, can be used for locating and extracting configuration-specific data as well. A notable example is a visibility constraint that determines when options are shown or hidden in the GUI. Consequently, by analysing the CSS *visibility* property of options we can deduce some cross-cutting constraints defined over those options.

The structural and data marking attributes we designed in our pattern specification language give us support to use *inline* style information (i.e., the style attribute in HTML elements) for the purpose of data extraction. However, it may be required to analyse the *internal* CSS code (i.e., the content of the `style` element in the `head` section of the page) and the *external* CSS files to infer configuration-specific data.

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 [Boucher et al., 2012c]. 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 [Boucher et al., 2012b]. 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

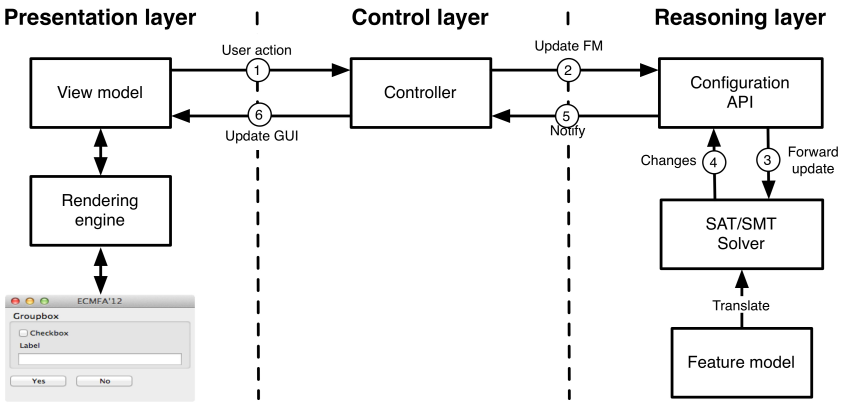

FIGURE 10.1: A MVC-like architecture for configurators [Boucher et al., 2012b].

used for verification purposes, e.g., checking the completeness and correctness of the configuration constraints.

We developed a tool that provides a lightweight environment for validation of feature models and verification of the configuration process [Abbasi et al., 2011a,b]. 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 [Hubaux et al., 2010, 2013] and *feature configuration workflows* (FCW) [Hubaux et al., 2009]:

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 [Mendonca et al., 2009][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

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

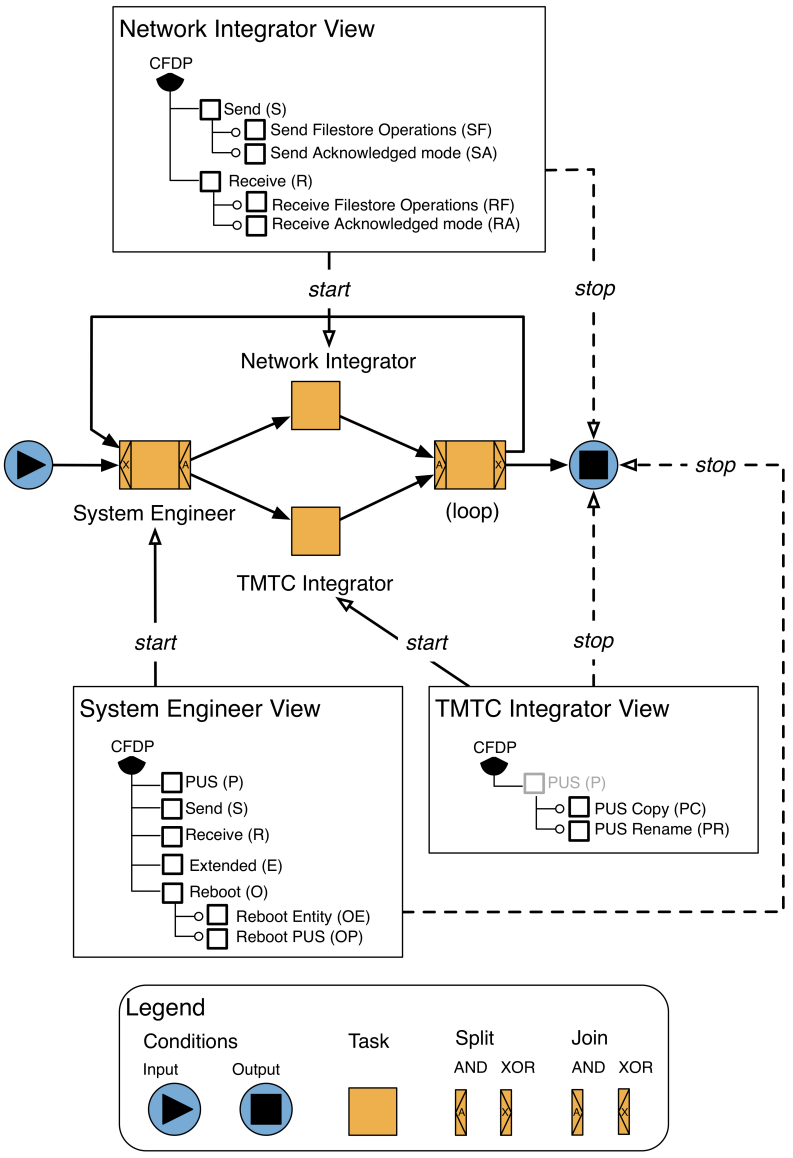

FIGURE 10.2: Example of FCW.

YAWL so as to trigger view-based configuration in SPLOT. We also implemented a new configuration environment, the *FCW engine*, which is 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 [Hubaux et al., 2010]. 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.

10.3.3 Other application scenarios

In this section, we present some further application scenarios that can exploit the results of this thesis.

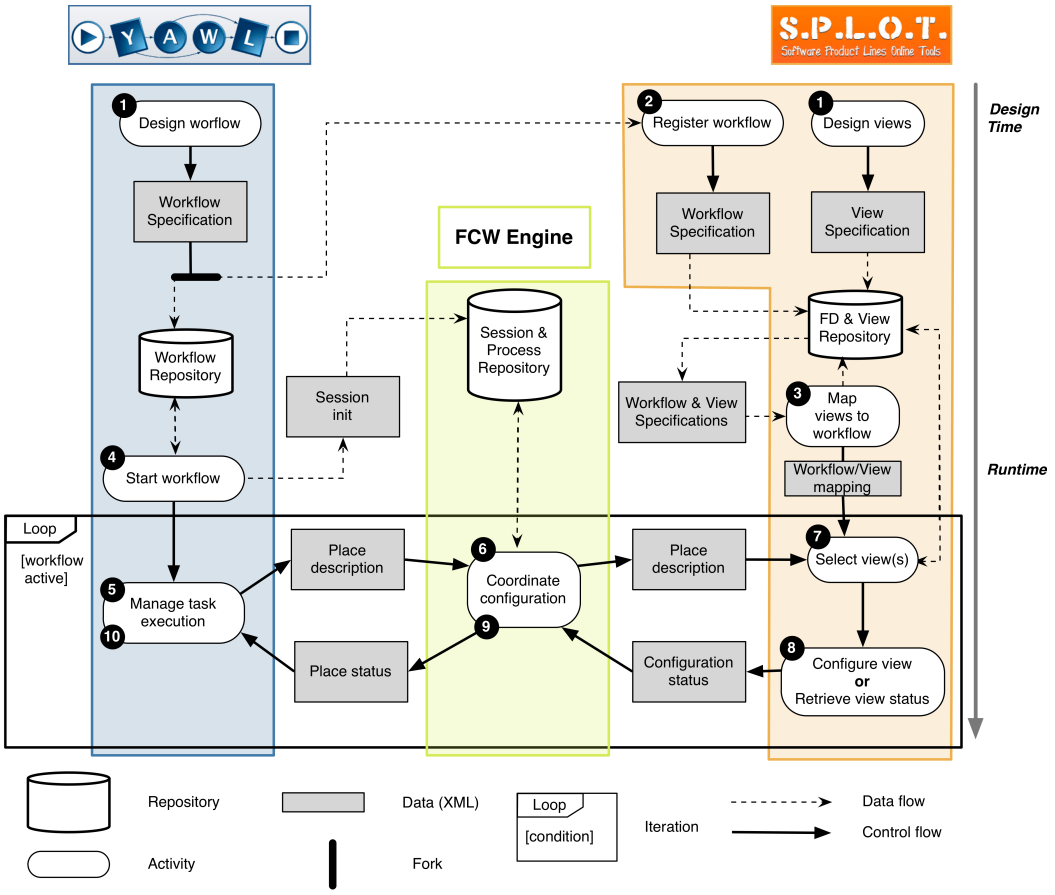

FIGURE 10.3: Overview of the essential components and typical use case scenario.

Identifying functionality related to user actions. We developed a Web Crawler to deal with dynamic behaviour of Web configurators. It is able to simulate some of the users' actions. By integrating this ability with Firebug's capability to track function calls, we are able to identify which functions (e.g., JavaScript functions) handle an action, which request is generated to be sent to the Web server, or even (in some cases) which server page is responsible for handling the request. Moreover, we can attach a new handler, and therefore a new functionality, to an action using jQuery at runtime.

These capabilities give us a lot of support for logging execution traces, testing, debugging, finding similar operational functionalities to be reused/replaced, improving the quality of existing Web configurators, etc. For instance, developers can add a new functionality to configuration actions so that when an option is configured, it registers itself to the list of configured options. Consequently, developers can build a trace log of user decisions that can be used to provide an undo operation for configurators that do not already offer it. Moreover, the configuration summary can be generated and presented to the customer.

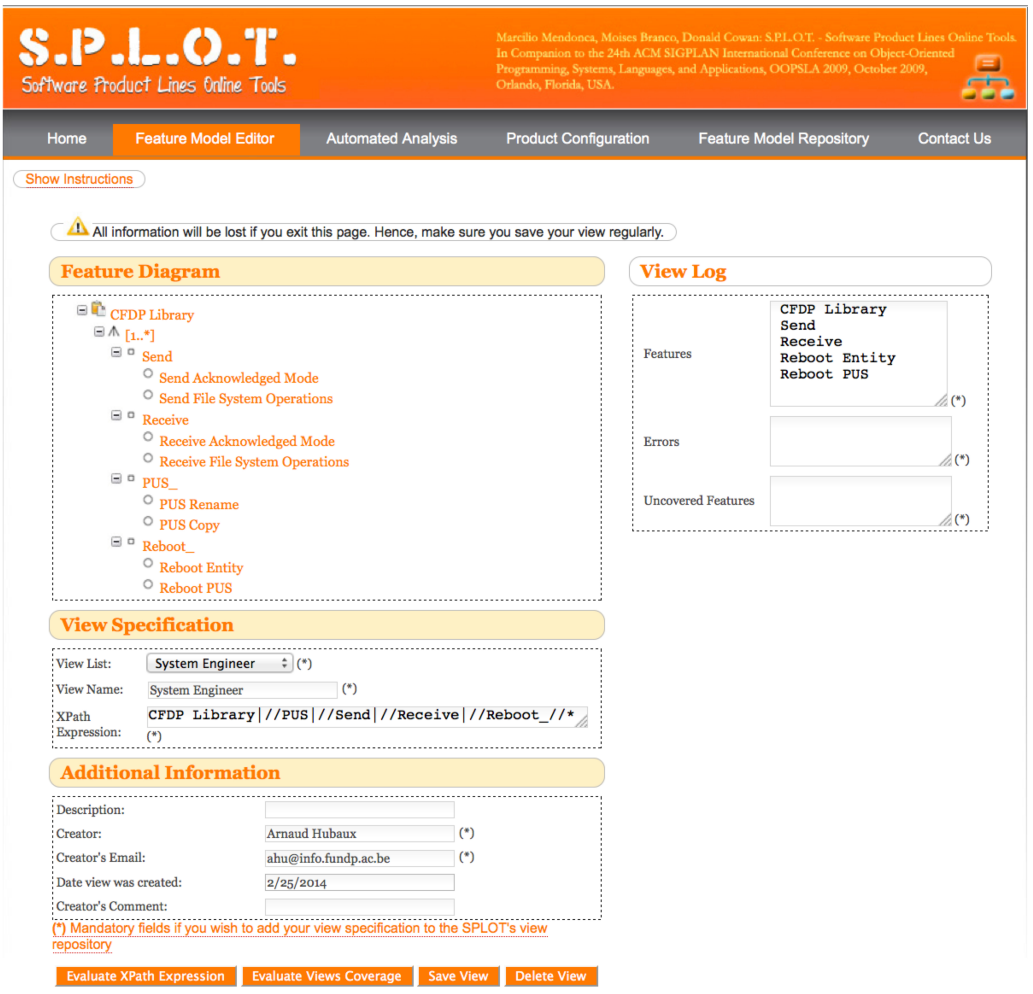

FIGURE 10.4: View creation menu.

More importantly, configurator developers can use our tool to verify that their development accurately implements the latest configuration options and constraints. It can be achieved by comparing the base feature model used for development of the configurator to the one that reverse-engineered by our approach.

Comparing the new and old configurator Websites. Once a new version of a Web configurator has been developed, it should be verified that the new configurator implements everything that must be kept from the old configurator, changes (e.g., options, constraints) made to the old version are identified and tested, etc. This can be to some extent achieved by comparing the reverse-engineered feature models of the old and new configurators.

Analysing competitors' Websites. Web configurators coming from the same industry sector most likely present products with similar characteristics (e.g., configuration options). Using our proposed Web data extraction techniques, we can acquire market

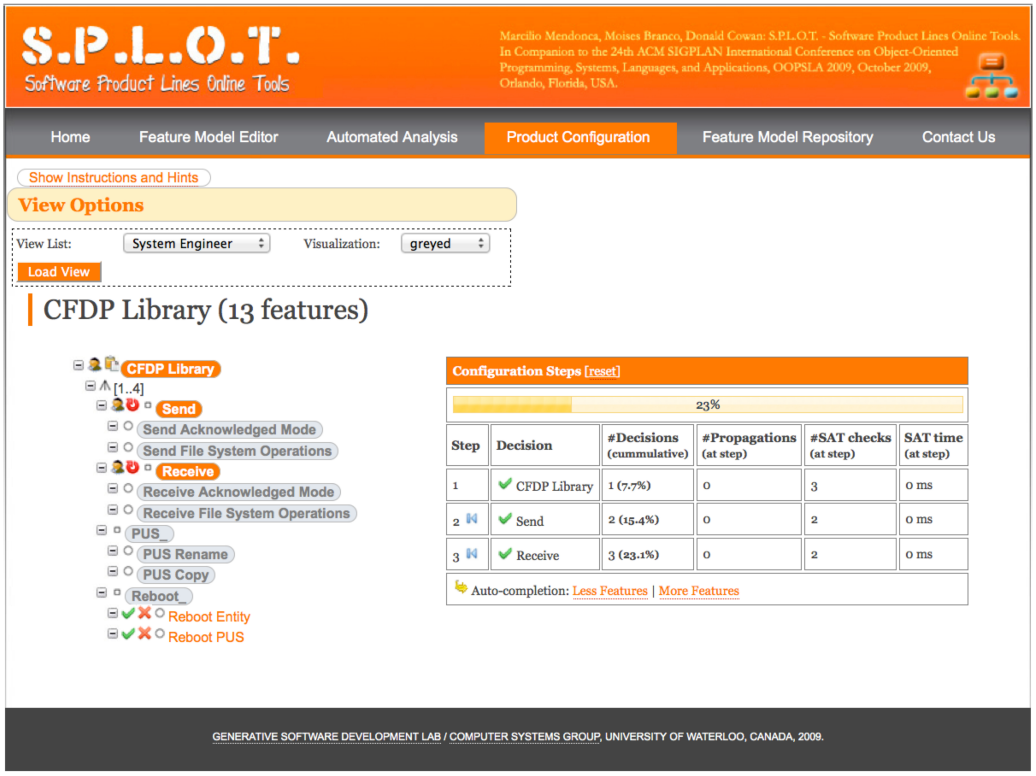

FIGURE 10.5: View configuration menu.

information from these competitors and compare their products (e.g., price comparison, option comparison). Baumgartner *et al.* [Baumgartner et al., 2007, 2005] presented that Web data extraction systems aim to provide services for acquiring market information, and should provide support for deep navigation and dynamic content pages as well. The Crawler can play a crucial role in exploring the configuration space and extracting dynamic data from different Web configurators.

Acquiring domain knowledge. Some configurators allow to customise several product categories. We can reverse engineer a feature model for each product and then merge them into a fully-fledged feature model. This new feature model represents all the variability of the set of products presented in a configurator. Similarly, we can merge the generated feature models of a family of configurators in a domain to build a body of knowledge for that domain.

Reverse engineering other kinds of applications. Although the reverse-engineering approach proposed in this thesis is used to reverse engineer feature models from Web configurators, there is the possibility of re-using it in other Web application domains as well. For instance:

- Online shopping systems (e.g., *amazon.com*) usually use templates to generate and present structured data [Arasu and Garcia-Molina, 2003].
- Conference registration applications have rather the same core design as the Web configurators we described in this thesis. Earlier user choices and decisions change what is presented to the user later.
- Student academic course planning applications guide the student through choices and offer courses based on prerequisite requirements. These applications have characteristics and organization similar to the Web configurators we studied.

Bibliography

Abbasi, E. K. (2013). Reverse engineering web sales configurators. In *2013 IEEE International Conference on Software Maintenance*, pages 586–589. IEEE Computer Society.

Abbasi, E. K., Acher, M., Heymans, P., and Cleve, A. (2014). Reverse Engineering Web Configurators. In *17th European Conference on Software Maintenance and Reengineering (CSMR)*, Antwerp, Belgique. IEEE.

Abbasi, E. K., Hubaux, A., Acher, M., Boucher, Q., and Heymans, P. (2013). The anatomy of a sales configurator: An empirical study of 111 cases. In Salinesi, C., Norrie, M., and Pastor, O., editors, *Advanced Information Systems Engineering*, volume 7908 of *Lecture Notes in Computer Science*, pages 162–177. Springer Berlin Heidelberg.

Abbasi, E. K., Hubaux, A., and Heymans, P. (2011a). 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. ACM.

Abbasi, E. K., Hubaux, A., and Heymans, P. (2011b). 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. IEEE Computer Society.

Acher, M., Cleve, A., Collet, P., Merle, P., Duchien, L., and Lahire, P. (2011). Reverse engineering architectural feature models. In *Proceedings of the 5th European conference on Software architecture*, ECSA'11, pages 220–235, Berlin, Heidelberg. Springer-Verlag.

Acher, M., Cleve, A., Collet, P., Merle, P., Duchien, L., and Lahire, P. (2013a). Extraction and evolution of architectural variability models in plugin-based systems. *Software and Systems Modeling*, pages 1–28.

Acher, M., Cleve, A., Perrouin, G., Heymans, P., Vanbeneden, C., Collet, P., and Lahire, P. (2012). 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. ACM.

Acher, M., Collet, P., Lahire, P., and France, R. B. (2013b). Familiar: A domain-specific language for large scale management of feature models. *Science of Computer Programming*, 78(6):657 – 681.

Adelberg, B. (1998). Nodose - a tool for semi-automatically extracting structured data and semi-structured data from text documents. *SIGMOD Record*, 27(2):283–294.

Álvarez, M., Pan, A., Raposo, J., Bellas, F., and Cacheda, F. (2010). Finding and extracting data records from web pages. *Signal Processing Systems*, 59(1):123–137.

Alves, V., Schwanninger, C., Barbosa, L., Rashid, A., Sawyer, P., Rayson, P., Pohl, C., and Rummler, A. (2008). 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. IEEE Computer Society.

Andersen, N., Czarnecki, K., She, S., and Wasowski, A. (2012). Efficient synthesis of feature models. In *Proceedings of the 16th International Software Product Line Conference - Volume 1*, SPLC '12, pages 106–115. ACM.

Apel, S., Lengauer, C., Möller, B., and Kästner, C. (2008). An algebra for features and feature composition. In Meseguer, J. and Roşu, G., editors, *Algebraic Methodology and Software Technology*, volume 5140 of *Lecture Notes in Computer Science*, pages 36–50. Springer Berlin Heidelberg.

Arasu, A. and Garcia-Molina, H. (2003). 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. ACM.

Arocena, G. O. and Mendelzon, A. O. (1999). Weboql: restructuring documents, databases, and webs. *Theory and Practice of Object Systems*, 5(3):127–141.

Bak, K., Czarnecki, K., and Wasowski, A. (2011). Feature and meta-models in clafer: Mixed, specialized, and coupled. In Malloy, B., Staab, S., and Brand, M., editors, *Software Language Engineering*, volume 6563 of *Lecture Notes in Computer Science*, pages 102–122. Springer Berlin Heidelberg.

Batory, D., Benavides, D., and Ruiz-Cortés, A. (2006). Automated analysis of feature models: Challenges ahead. *Communications of the ACM*, 49(12):45–47.

Baumgartner, R., Flesca, S., and Gottlob, G. (2001a). Declarative information extraction, web crawling, and recursive wrapping with lixto. In Eiter, T., Faber, W., and Truszczyński, M., editors, *Logic Programming and Nonmotonic Reasoning*, volume 2173 of *Lecture Notes in Computer Science*, pages 21–41. Springer Berlin Heidelberg.

Baumgartner, R., Flesca, S., and Gottlob, G. (2001b). 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. Morgan Kaufmann Publishers Inc.

Baumgartner, R., Frolich, O., and Gottlob, G. (2007). The lixto systems applications in business intelligence and semantic web. In *The Semantic Web: Research and Applications*, volume 4519 of *Lecture Notes in Computer Science*, pages 16–26. Springer Berlin Heidelberg.

Baumgartner, R., Frolich, O., Gottlob, G., Harz, P., Herzog, M., Lehmann, P., and Wien, T. (2005). Web data extraction for business intelligence: the lixto approach. In *Proceedings of BTW*, pages 48–65.

Baumgartner, R., Gatterbauer, W., and Gottlob, G. (2009). Web data extraction system. In *Encyclopedia of Database Systems*, pages 3465–3471. Springer US.

Becker, M., Geyer, L., Gilbert, A., and Becker, K. (2002). Comprehensive variability modelling to facilitate efficient variability treatment. In Linden, F., editor, *Software Product-Family Engineering*, volume 2290 of *Lecture Notes in Computer Science*, pages 294–303. Springer Berlin Heidelberg.

Bellucci, F., Ghiani, G., Paternò, F., and Porta, C. (2012). 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. ACM.

Benavides, D., Segura, S., and Ruiz-Cortés, A. (2010). Automated analysis of feature models 20 years later: A literature review. *Information Systems*, 35(6):615–636.

Benavides, D., Segura, S., Trinidad, P., and Ruiz-Cortés, A. (2007). 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.

Berger, T., She, S., Lotufo, R., Wasowski, A., and Czarnecki, K. (2010). Variability modeling in the real: a perspective from the operating systems domain. In *ASE'10*, pages 73–82. ACM.

Beuche, D. (2012). 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. ACM.

Blecker, T. and Abdelkafi, N. (2006). Mass customization: State-of-the-art and challenges. In Blecker, T. and Friedrich, G., editors, *Mass Customization: Challenges and Solutions*, volume 87 of *International Series in Operations Research and Management Science*, pages 1–25. Springer US.

Botterweck, Goetz, J. M. and Schneeweiss, D. (2009). A design of a configurable feature model configurator. In *3rd International Workshop on Variability Modelling of Software-Intensive Systems (VAMOS 2009)*.

Boucher, Q., Abbasi, E. K., Hubaux, A., Perrouin, G., Acher, M., and Heymans, P. (2012a). 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. IEEE Computer Society.

Boucher, Q., Perrouin, G., Acher, M., and Heymans, P. (2012b). Engineering configuration graphical user interfaces: A model-based perspective. Technical Report P-CS-TR ECMFA-000001, University of Namur.

Boucher, Q., Perrouin, G., and Heymans, P. (2012c). 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. ACM.

Bouillon, L. (2006). *Reverse Engineering of Declarative User Interfaces*. PhD thesis, Universite catholique de Louvain, Louvain-la-Neuve, Belgium.

Califf, M. E. and Mooney, R. J. (1999). 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. American Association for Artificial Intelligence.

Chang, C.-H., Kayed, M., Girgis, M. R., and Shaalan, K. F. (2006). A survey of web information extraction systems. *IEEE Transactions on Knowledge and Data Engineering*, 18(10):1411–1428.

Chang, C.-H. and Kuo, S.-C. (2004). Olera: Semisupervised web-data extraction with visual support. *IEEE Intelligent Systems*, 19(6):56–64.

Chang, C.-H. and Lui, S.-C. (2001). 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. ACM.

Chen, K., Zhang, W., Zhao, H., and Mei, H. (2005). 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. IEEE Computer Society.

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

Classen, A., Boucher, Q., Faber, P., and Heymans, P. (2010). The TVL specification. Technical Report P-CS-TR SPLBT-00000003, PReCISE Research Center, University of Namur, Namur, Belgium.

Classen, A., Boucher, Q., and Heymans, P. (2011a). A text-based approach to feature modelling: Syntax and semantics of tvl. *Science of Computer Programming*, 76(12):1130–1143.

Classen, A., Boucher, Q., and Heymans, P. (2011b). A text-based approach to feature modelling: Syntax and semantics of tvl. *Science of Computer Programming*, 76(12):1130–1143.

Classen, A., Heymans, P., and Schobbens, P.-Y. (2008). 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. Springer-Verlag.

Clements, P. and Northrop, L. (2002). *Software Product Lines: Practices and Patterns*. Addison Wesley.

Conallen, J. (1999). Modeling web application architectures with uml. *Communications of the ACM*, 42(10):63–70.

Crescenzi, V. and Mecca, G. (1998). Grammars have exceptions. *Information Systems*, 23(9):539–565.

Crescenzi, V., Mecca, G., and Merialdo, P. (2001). 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. Morgan Kaufmann Publishers Inc.

Cyledge (2014). Configurator database. http://www.configurator-database.com. [Online; accessed 16-February-2014].

Czarnecki, Krzysztof, H. S. and Eisenecker, U. (2005). Formalizing cardinality-based feature models and their specifications. *Software Process Improvement and Practice*, 10(1):7–29.

Czarnecki, K., Grünbacher, P., Rabiser, R., Schmid, K., and Wasowski, A. (2012). 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. ACM.

Czarnecki, K. and Kim, C. H. P. (2005). Cardinality-based feature modeling and constraints: A progress report. In *OOPSLA'05*.

Davril, J.-M., Delfosse, E., Hariri, N., Acher, M., Cleland-Huang, J., and Heymans, P. (2013). 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)*.

de Silva, C. E. B. e. M. (2010). *Reverse Engineering of Rich Internet Applications*. PhD thesis, University of Minho, Portugal.

Di Lucca, G. A., Fasolino, A. R., and Tramontana, P. (2004). Reverse engineering web applications: the WARE approach. *Software Maintenance and Evolution*, 16(1-2):71–101.

DOM (2014). What is the Document Object Model? http://www.w3.org/TR/DOM-Level-2-Core/introduction.html. [Online; accessed 16-February-2014].

Draheim, D., Lutteroth, C., and Weber, G. (2005). 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. IEEE Computer Society.

Embley, D. W., Campbell, D. M., Jiang, Y. S., Liddle, S. W., Lonsdale, D. W., Ng, Y.-K., and Smith, R. D. (1999). Conceptual-model-based data extraction from multiple-record web pages. *Data and Knowledge Engineering*, 31(3):227–251.

Ferrara, E., Meo, P. D., Fiumara, G., and Baumgartner, R. (2012). Web data extraction, applications and techniques: A survey. *CoRR*, abs/1207.0246.

Franke, N. and Piller, F. (2002). *Configuration Toolkits for Mass Customization: Setting a Research Agenda*. Technische Universität München.

Freitag, D. (1998). 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. American Association for Artificial Intelligence.

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

Gottschalk, F., Wagemakers, T. A. C., Jansen-Vullers, M. H., van der Aalst, W. M. P., and Rosa, M. L. (2009). Configurable process models: Experiences from a municipality case study. In *CAiSE*, pages 486–500.

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

Hammer, J., McHugh, J., and Garcia-Molin, H. (1997). 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. British Computer Society.

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

Haslinger, E. N., Lopez-Herrejon, R. E., and Egyed, A. (2013). 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. Springer-Verlag.

Hayes, R. H. and Wheelwright, S. G. (1979). The dynamics of process-product life cycles. *Harvard Business Review*.

Hedin, G., Ohlsson, L., and McKenna, J. (1998). Product configuration using object oriented grammars. In Magnusson, B., editor, *System Configuration Management*, volume 1439 of *Lecture Notes in Computer Science*, pages 107–126. Springer Berlin Heidelberg.

Henard, C., Papadakis, M., Perrouin, G., Klein, J., and Le Traon, Y. (2013). 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. IEEE Press.

Hogue, A. and Karger, D. (2005). 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. ACM.

Hsu, C.-N. and Dung, M.-T. (1998). Generating finite-state transducers for semi-structured data extraction from the web. *Information Systems*, 23(9):521–538.

HTML5 (2014). HTML5 Specification. http://www.w3.org/TR/html5/. [Online; accessed 16-February-2014].

Hubaux, A., Classen, A., and Heymans, P. (2009). Formal modelling of feature configuration workflows. In *Proceedings of the 13th International Software Product Line Conference*, SPLC '09, pages 221–230, Pittsburgh, PA, USA. Carnegie Mellon University.

Hubaux, A., Heymans, P., Schobbens, P.-Y., and Deridder, D. (2010). Towards multi-view feature-based configuration. In Wieringa, R. and Persson, A., editors, *Requirements Engineering: Foundation for Software Quality*, volume 6182 of *Lecture Notes in Computer Science*, pages 106–112. Springer Berlin Heidelberg.

Hubaux, A., Heymans, P., Schobbens, P.-Y., Deridder, D., and Abbasi, E. K. (2013). Supporting multiple perspectives in feature-based configuration. *Software and Systems Modeling*, 12(3):641–663.

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

Hvam, L., Mortensen, N. H., and Riis, J. (2008). *Product Customization*. Springer-Verlag Berlin Heidelberg.

Janota, M. (2010). *SAT Solving in Interactive Configuration*. PhD thesis, University College Dublin.

Janota, M., Botterweck, G., Grigore, R., and Marques-Silva, J. (2009). How to complete an interactive configuration process? *CoRR*, abs/0910.3913.

John, I. (2006). Capturing product line information from legacy user documentation. In Käköla, T. and Duenas, J., editors, *Software Product Lines*, pages 127–159. Springer Berlin Heidelberg.

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

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

Kushmerick, N. (2000). Wrapper induction: efficiency and expressiveness. *Artificial Intelligence*, 118(1-2):15–68.

Kushmerick, N., Weld, D. S., and Doorenbos, R. B. (1997). Wrapper induction for information extraction. In *IJCAI (1)*, pages 729–737. Morgan Kaufmann.

Laender, A. H. F., Ribeiro-Neto, B., and da Silva, A. S. (2002a). Debye - date extraction by example. *Data and Knowledge Engineering*, 40(2):121–154.

Laender, A. H. F., Ribeiro-Neto, B. A., da Silva, A. S., and Teixeira, J. S. (2002b). A brief survey of web data extraction tools. *SIGMOD Record*, 31(2):84–93.

Lerman, K., Getoor, L., Minton, S., and Knoblock, C. (2004). 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. ACM.

Liu, B., Grossman, R., and Zhai, Y. (2003). 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. ACM.

Liu, L., Pu, C., and Han, W. (2000). 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. IEEE Computer Society.

Lora-Michiels, A., Salinesi, C., and Mazo, R. (2010). 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.

Maras, J., Carlson, J., and Crnkovi, I. (2012). 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. ACM.

Mendonca, M., Branco, M., and Cowan, D. (2009). 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. ACM.

Meng, X., Hu, D., and Li, C. (2003). 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. ACM.

Mesbah, A., van Deursen, A., and Lenselink, S. (2012). Crawling ajax-based web applications through dynamic analysis of user interface state changes. *ACM Transactions on the Web*, 6(1):3:1–3:30.

Michel, R., Classen, A., Hubaux, A., and Boucher, Q. (2011a). 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. ACM.

Michel, R., Classen, A., Hubaux, A., and Boucher, Q. (2011b). A formal semantics for feature cardinalities in feature diagrams. In *VaMoS'11*, pages 82–89. ACM.

Muslea, I., Minton, S., and Knoblock, C. (1999). 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. ACM.

Muslea, I., Minton, S., and Knoblock, C. A. (2001). Hierarchical wrapper induction for semi-structured information sources. *Autonomous Agents and Multi-Agent Systems*, 4(1-2):93–114.

Nguyen, B., Robbins, B., Banerjee, I., and Memon, A. (2013). Guitar: an innovative tool for automated testing of gui-driven software. *Automated Software Engineering*, pages 1–41.

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

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

Patel, R., Coenen, F., Martin, R., and Archer, L. (2007). Reverse engineeing of web applications: A technical review.

Phan, X.-H., Horiguchi, S., and Ho, T.-B. (2005). Automated data extraction from the web with conditional models. *Business Intelligence and Data Mining*, 1(2):194–209.

Pine, B. J. (1993). *Mass Customization: The New Frontier in Business Competition*. Harvard Business School Press.

Pohl, K., Böckle, G., and van der Linden, F. J. (2005). *Software Product Line Engineering: Foundations, Principles and Techniques.* Springer-Verlag New York, Inc.

Reis, D. C., Golgher, P. B., Silva, A. S., and Laender, A. F. (2004). 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. ACM.

Ribeiro-Neto, B., Laender, A. H. F., and Silva, A. S. D. (1999). Extracting semi-structured data through examples. In *In Proceedings of the International Conference on Knowledge Management*, pages 94–101.

Ricca, F. and Tonella, P. (2001). Understanding and restructuring web sites with reweb. *IEEE MultiMedia*, 8(2):40–51.

Rogoll, T. and Piller, F. (2004). 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.*

Rosa, M. L., van der Aalst, W. M., Dumas, M., and ter Hofstede, A. H. (2008). Questionnaire-based variability modeling for system configuration. *Software and Systems Modeling*, 8(2):251–274.

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

Sahuguet, A. and Azavant, F. (1999). 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. Morgan Kaufmann Publishers Inc.

Sahuguet, A. and Azavant, F. (2001). Building intelligent web applications using lightweight wrappers. *Data and Knowledge Engineering*, 36(3):283–316.

Santoro, C. and Spano, L. D. (2009). MARIA: A universal, declarative, multiple abstraction-level language for service-oriented applications in ubiquitous environments. *ACM Transactions on Computer-human Interaction*, 16:1–30.

Schäler, M., Leich, T., Rosenmüller, M., and Saake, G. (2012). Building information system variants with tailored database schemas using features. In *CAiSE'12*, pages 597–612. Springer-Verlag.

Schobbens, P.-Y., Heymans, P., and Trigaux, J.-C. (2006). 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. IEEE Computer Society.

SEI (2014). Software Product Lines. http://www.sei.cmu.edu/productlines/. [Online; accessed 16-February-2014].

She, S., Lotufo, R., Berger, T., Wasowski, A., and Czarnecki, K. (2011). Reverse engineering feature models. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 461–470, New York, NY, USA. ACM.

Silva, J. C., Silva, C. E., Gonçalo, R. D., Saraiva, J., and Campos, J. C. (2010). 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.

Soderland, S. (1999). Learning information extraction rules for semi-structured and free text. *Machine Learning*, 34(1-3):233–272.

Stoiber, R. (2012). *A New Approach to Product Line Engineering in Model-Based Requirements Engineering*. PhD thesis, University of Zurich.

Streichsbier, C., Blazek, P., Faltin, F., and Fruhwirt, W. (2009). 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.

Streitferdt, D. (2004). *Family-oriented requirements engineering*. PhD thesis, Technical University of Ilmenau.

Thum, T., Kastner, C., Benduhn, F., Meinicke, J., Saake, G., and Leich, T. (2014). Featureide: An extensible framework for feature-oriented software development. *Science of Computer Programming*, 79(0):70 – 85.

Tilley, S. and Huang, S. (2001). 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. IEEE Computer Society.

Tip, F. (1995). A survey of program slicing techniques. *Programming Languages*, 3:121–189.

Tramontana, P. (2005). *Reverse Engineering Web Applications*. PhD thesis, University of Naples, Federico II, Italy.

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

Tseng, M. M. and Jiao, J. (2007). *Mass Customization*, pages 684–709. John Wiley and Sons, Inc.

Tseng, M. M. and Piller, F. (2003). The customer centric enterprise - advances in mass customization and personalization.

Vanderdonckt, J., Bouillon, L., and Souchon, N. (2001). 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. IEEE Computer Society.

Visser, E. (2004a). Program transformation with stratego/xt. In Lengauer, C., Batory, D., Consel, C., and Odersky, M., editors, *Domain-Specific Program Generation*, volume 3016 of *Lecture Notes in Computer Science*, pages 216–238. Springer Berlin Heidelberg.

Visser, Eelco, S. J. (2004b). Tutorial on strategic programming across programming paradigms. In *8th Brazilian Symposium on Programming Languages*.

von Hippel, E. (2001). Perspective: User toolkits for innovation. *Product Innovation Management*, 18(4):247 – 257.

von Hippel, E. and Katz, R. (2002). Shifting innovation to users via toolkits. *Management Science*, 48(7):821–833.

Wang, J. and Lochovsky, F. H. (2002). 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.

Wang, J. and Lochovsky, F. H. (2003). 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. ACM.

Weston, N., Chitchyan, R., and Rashid, A. (2009). 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. Carnegie Mellon University.

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

Zhai, Y. and Liu, B. (2005). 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. ACM.

Zheng, S., Song, R., Wen, J.-R., and Giles, C. L. (2009). 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. ACM.

Ziadi, T., Frias, L., da Silva, M. A. A., and Ziane, M. (2012). Feature identification from the source code of product variants. *2011 15th European Conference on Software Maintenance and Reengineering*, 0:417–422.