Logical Structure Extraction from Software Requirements Documents

Rehan Rauf, Michał Antkiewicz, Krzysztof Czarnecki
Generative Software Development Lab
University of Waterloo, Waterloo, Canada
{rrauf,mantkiew,kczarnec}@gsd.uwaterloo.ca

***Abstract*—Software requirements documents (SRDs) are often authored in general-purpose rich-text editors, such as MS Word. SRDs contain instances of logical structures, such as *use case*, *business rule*, and *functional requirement*. Automated recognition and extraction of these instances enables advanced requirements management features, such as automated traceability, template conformance checking, guided editing, and interoperability with requirements management tools such as RequisitePro. The variability in content and physical representation of these instances poses challenges to their accurate recognition and extraction. To address these challenges, we present a framework allowing 1) the specification of logical structures in terms of their content, textual rendering, and variability and 2) the extraction of instances of such structures from rich-text documents. Our evaluation involves 36 different logical structures identified in 43 SRDs and shows that the intended content, style, and variability of these structures can be specified in the framework such that their instances can be extracted from the documents with high precision and recall, both close to 100%.**

***Index Terms*—Logical Structures; SRS; Requirements Extraction; Software Requirements Documents**

I. Introduction

Requirements management (RM) tools store and manipulate software requirements at the level of *logical structures* (LSs), such as *functional requirement* or *use case*. LSs are hierarchies of *logical components* (LCs), such as *actor* and *main flow* within use case. By knowing the LSs, these tools can offer advanced features, such as fine-grained traceability, guided editing, structured querying, and template conformance checking [1]. Despite the availability of such tools, many organizations still utilize general-purpose text editors, such as MS Word, to write software requirements. To bridge the gap between structured requirements and free text, RM tools such as RequisitePro [2] provide capability to import entire documents and link to requirements inside these documents. The import process requires users to specify either regular expressions or delimiter sequences for every structure to be extracted. Import capabilities are limited to identifying atomic requirements, in the form of sentences or paragraphs, and they do not identify the individual LCs within the structures.

We propose a framework for the specification of LSs as *templates* and the extraction of their instances from rich-text documents. The framework provides an opportunity for combining an open environment of generic and widely adopted text editors with the advanced features offered by RM tools.

The research was conducted in the following three phases:

1) We collected requirements documents from different sources, including industry.
2) We analyzed a randomly selected subset of documents from our collection to identify requirements for LS instance extraction and developed the framework.
3) We evaluated the framework on the entire collection of documents.

In the studied document sample, we observed that LS instances often follow common templates. Possible reasons for the similarities include the use of same requirements engineering method, the same author using the same template, and an organization prescribing a common template.

We hypothesize that LS instances can be extracted by finding elements in the documents matching a given template. However, many challenges are involved in doing so.

A template has two aspects—the *logical aspect* and the *physical aspect*. The logical aspect refers to the LCs of the LS and the rules related to these LCs. They are often prescribed by the software development process such as *Rational Unified Process* [3]. For example, LCs of a *use case* include *name*, *actor*s, and *main scenario*. The physical aspect refers to the physical representation of the LS in rich-text documents. For example, a use case may be written as a *table* and its name in the table's top left *cell* or as a *section* and the name as a part of the section's *title*.

Fig.1 shows instances of two different use case templates (left and right, respectively). The examples do not show actual data from collected SRDs—but depict the variability observed within and across the documents. The physical aspect of each of the templates is clearly different. Fig.1a shows use cases as sections of text; Fig.1b shows them as tables. The differences in LCs are also evident. The LCs in Fig.1a are *ID*, *Name*, *Flow*, and *Extensions*; the LCs in Fig.1b are *Name*, *Actor*, *Precondition*, and *Process Description*. These differences stem from different approaches to use cases, e.g., [3], [4].

Fig.1a and Fig.1b also depict examples of variations among the instances of the same template. These variations fall into two categories—*logical variations* and *physical variations*. The *logical variations* include differences in the number and type of LC instances across LS instances. For example, the LC *Extensions* is missing in UC10 (Fig.1a). The logical variations also include using different LCs to represent the same concept. For example, use cases in Fig. 1b describe the main scenario either as a list of steps or action-response pairs.

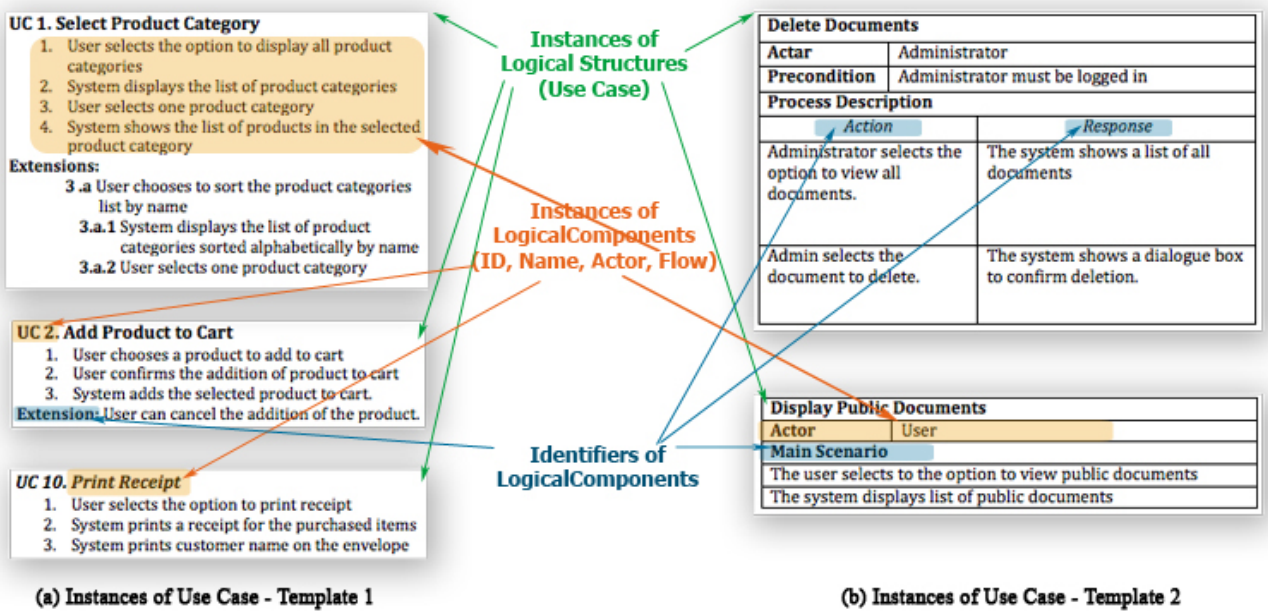


Fig. 1: Sample use cases

The *physical variations* can be *accidental* or *designed*. The accidental variations are minor mistakes introduced when writing LS instances. These include spelling mistakes—such as "Actar" instead of "Actor" in the second instance in Fig.1b—and mistakes in applying font style (bold or italic) or size—such as the unnecessary italicized section title in UC10. The designed variations are intended. For example, UC1 specifies extensions as a sub-section; UC2 specifies extension as a *text block*—a paragraph that begins with an identifier and a delimiter. The convention for writing IDs—either alpha-numeric or numeric only—is another designed variation.

The framework allows users to create a template for the LS of interest. The template specifies LCs, *physical components* (PCs)—such as *tables* and *lists*—used to represent instances of the LS, font style and size, and their variability. An *Extraction Tool* (ET) uses the template to extract the instances from documents.

To measure the applicability, effectiveness, and efficiency of the framework, we created templates for 36 LSs and used them to extract instances. The evaluation shows that the intended content, style, and variability of these LSs can be specified in the framework such that their instances can be extracted from the documents with high precision and recall, both close to 100%, and extraction times ranging from a few milliseconds to a few seconds.

The remainder of the paper is organized as follows. We discuss related work in Sec. II, explain the proposed framework in Sec. III, present the evaluation in Sec. IV, and conclude in Sec. VII.

II. Related Work

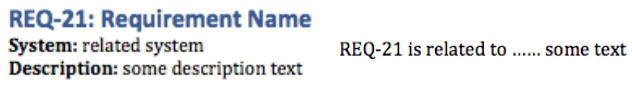
Automated natural language (NL) analyses have been used to validate requirements [5] [6] [7], disambiguate confusing statements [8], and extract ontology [9]. Most of these techniques work at the level of NL sentences without the knowledge of LS instances and LCs to which the sentences belong. Recently, tools are emerging that perform automatic inspection at the level of LSs, like use cases. For example, Text2Test [10] uses NL analysis to produce a semantic model of a use case and report problems in the model at edit time. The tool assumes use cases as input. Our framework does not use NL analyses, but it can offer a front end for deeper NL-based semantic analyses by locating LS instances within documents and thus providing additional context for analyzing sentences.

Document structure analysis includes two types of work. *Document physical layout analysis* recognizes hierarchies of physical components, such as pages, columns, paragraphs, and textlines, in scanned documents. *Document logical structure analysis* translates physical components into hierarchies of logical components, such as titles, authors, abstracts, lists, and sections (see [11] for a survey). Our framework is similar to the latter body of work, as it also uses tree-grammar-like specification to translate some physical hierarchies into logical ones; however, our LSs are domain-specific, such as use case or business object, and our PCs correspond to *document* logical structures, such as sections and lists. Thus, the LS template mechanism uses different physical components and properties than models for *document* logical structure.

Nojoumian and Lethbridge extract document logical structure—specifically, the hierarchy of headings—from the PDF document specifying the UML standard [12]. Their experience shows that the underlying physical representation contains a lot of noise; they succeed by relying on a special bookmark tag in PDF. Again, our LSs are domain-specific and

modeled declaratively.

We are not aware of any formal studies of LS instance extraction from software requirements documents. However, some commercial RM tools, such as Reqtify [13] and RequisitePro [2], allow users to extract and import requirements from rich-text documents. In Reqtify, an LS is a section with a title matching a specific regular expression. Further, users can specify a *single* LC nested in the LS using an additional regular expression; no further nesting is possible. Reqtify relies on predefined MS Word styles to recognize sections. In RequisitePro, users specify an LS either as a section using a particular MS Word style or set of keywords or as a text passage delimited by two regular expressions. Although both tools preserve section hierarchies during import, LS specifications have limited (Reqtify) or no nesting (RequisitePro).



(a) Requirement definition (b) Reference to a requirement

Fig. 2: Definition vs. reference

Because of the restrictions on LS specifications, documents should be written with the target RM tool in mind to be imported without time-consuming re-formatting. For example, both tools are good at matching individual requirements formatted as paragraphs or sections, but the recognition of LCs within these requirements is limited. For example, these tools cannot identify individual flow items of a use case unless they are prefixed by some identifier or formatted as sections or using a special Word style. In our framework, they are simply defined as a numbered list. Further, our framework supports LSs with multiple LCs, variability, and a rich repertoire of commonly found PCs, such as lists, text blocks, and tables, without relying on predefined Word styles.

Another limitation of existing RM tools relates to their use of regular expressions. For example, the regular expression `REQ-\d+*\\n\\n` would match each definition of a requirement of this form and each reference (of this form) to the requirement, as shown in Fig.2.

RequisitePro and Reqtify rely on using predefined Word styles (Headings) to disambiguate between definitions and references. The solution does not work when font styles and sizes are used to imitate the predefined Word styles or authors used a language different than English—we observed many such cases in industrial documents. Finally, both tools do not tolerate accidental physical variability—style mistakes or misspelled identifiers lead to missed instances. Our framework addresses all these problems.

The application of regular expressions for locating elements is widely found in information retrieval (IR). In IR literature, procedures for extracting relational content from web pages are called *wrappers* [14]. Wrappers use known patterns to locate and extract the required information. Automatic generation of wrappers from examples is called *wrapper induction* [15]. Several wrapper induction systems exist [14], [16], [17], [18], [19]; a tool survey [20] is also available.

Wrappers, which are sophisticated regular expressions, work well for generated web pages; however, many factors limit their practical use for LS instance extraction from manually written rich-text documents. First, wrappers are regular expressions over the underlying document representation, such as HTML tags. Since web pages are usually generated by scripts filling in data from databases, the generated HTML code is uniform for all instances. However, when writing requirements documents, authors often try to visually match the template, which usually results in different underlying representations. For example, similar looking MS Word documents may have different underlying Office Open XML [21] tags. Therefore, a wrapper trained on a set of instances may not be capable of extracting all instances from documents occurring in practice. Second, existing wrapper induction systems are limited in their capability to efficiently handle missing and out-of-order LC instances, and they require a large training set to capture all possible variants. One could potentially directly change the induced wrapper to take into account missing or out-of-order LC instances; however, the internal representations of induced wrappers we have examined are hard to modify since they are not intended for human editing.

The framework proposed in this paper was inspired by the work of Antkiewicz et al., which allows specifying concepts provided by application programming interfaces (APIs) of object-oriented frameworks and using these specifications to automatically extract their instances from application code [22]. The main difference between the two works is the type of artifacts they deal with. API concepts are specified in terms of code mappings requiring static program code analysis; LS templates are specified in terms of PCs of rich-text documents and require specialized document queries.

III. Framework for LS Instance Extraction

The analysis of sample SRDs and related work lead us to the following requirements on an LS instance extraction framework.

1) A single template should work with any document format, including Word, PDF, and HTML: An organization may use multiple document formats. Even if all documents visually conform to a single template, their underlying representations may be radically different. A template defined at the level of underlying representation would be different for each document format. To be shared across different document formats, templates should be defined at the logical level.

2) Templates should be specified in a human-readable form: Human-readable form allows for easy modification to take advantage of expert knowledge. For example, a user may know that preconditions and postconditions in use cases are optional LCs. Providing sample instances to capture all possible LC combinations and orders is impractical since the number of instances grows exponentially to the number of optional LCs.

3) The framework should tolerate minor errors like spelling mistakes and style inconsistencies: Such mistakes occur in

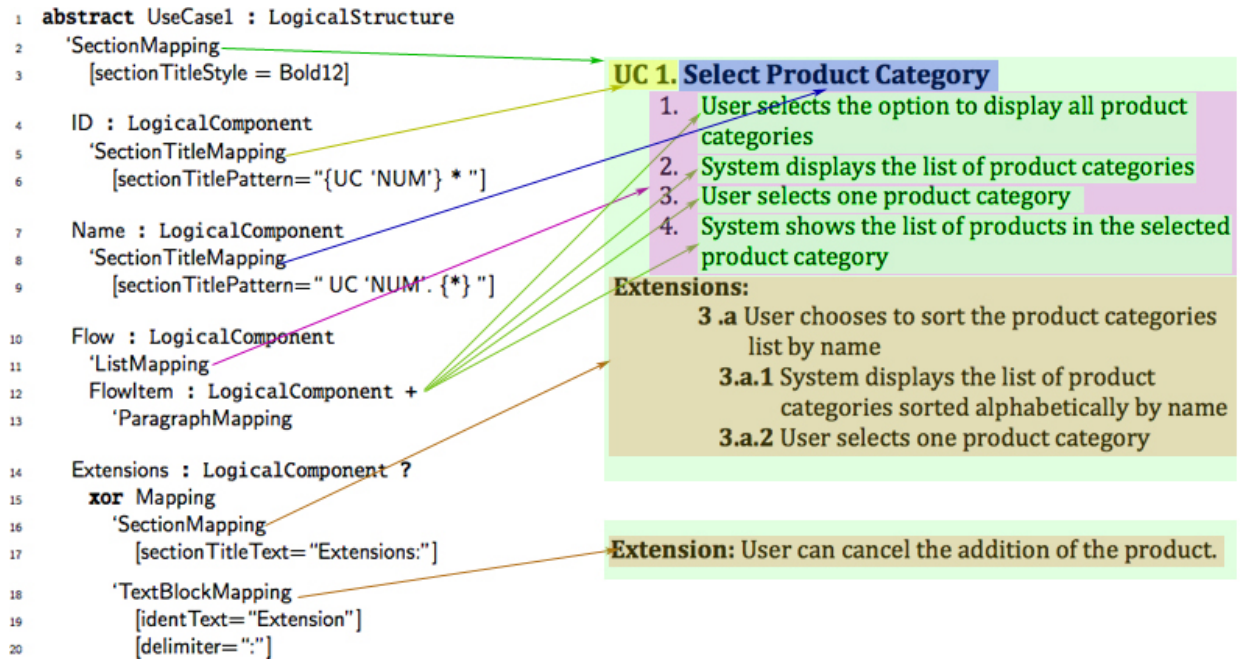


Fig. 3: A template for use case instances from Fig.1a with mappings to the instances

practice and would likely increase the number of false negatives during extraction. Capturing all possible variants of minor spelling errors within regular expression is not feasible.

4) The framework should be easily extensible with new LSs and presentations: Organizations and projects use many different LSs and should be able to easily add new LSs and new presentation variants.

The proposed framework consists of three parts: *template modeling*, *document queries*, and an *extraction tool* (ET). ET reads a template of an LS and uses pre-defined document queries to extract PCs from input documents; it returns parts of the documents that satisfy constraints specified in the template as instances of the LS. We describe each part in the following subsections.

A. Template Modeling

Fig. 3 and Fig. 4 present templates for extracting the use case instances from Fig. 1a,b, respectively. The templates are specified in *Clafer*, a powerful yet concise modeling language [23]. Clafer was successfully used for modeling 58 feature models, 5 metamodels (including UML2), and one feature-based model template. We refrain from explaining the language; instead, we explain how the sample templates specify the two LSs.

Line 1 in Fig. 3 declares an LS named `UseCase1`; `abstract` indicates that `UseCase1` is a template—as opposed to an instance. `ID`, `Name`, `Flow`, and `Extensions` are LCs nested under `UseCase1`; `FlowItem` (line 12) is an LC nested under `Flow`. Such representation directly corresponds to the hierarchical structure of the documents.

Every LS and LC has a *mapping*, specifying the PC that the LS or LC is represented by in the document. For example,

```
1  abstract UseCase2 : LogicalStructure
2    'TableMapping
3      Name : LogicalComponent
4        'CellMapping
5          [colIndex=1
6           rowIndex=1]
7      Actor : LogicalComponent
8        'HCellBlockMapping
9          [identText="Actor"]
10     Precondition : LogicalComponent ?
11       'HCellBlockMapping
12         [identText="Precondition"]
13     xor Flow
14       ProcDesc : LogicalComponent
15           'ColumnMapping
16              [colTitleText = "Process Description"]
17         Action : LogicalComponent
18           'ColumnMapping
19              [colTitleText = "Action"]
20         Response : LogicalComponent
21           'ColumnMapping
22              [colTitleText = "Response"]
23       MainScenario : LogicalComponent
24           'ColumnMapping
25              [colTitleText = "Main Scenario"]
```

Fig. 4: A template for use case instances in Fig.1b

`'SectionMapping` (line 2) indicates that each `UseCase1` instance maps to a section. Mappings have properties, which can be constrained within brackets. For example, line 3 specifies the font style for the title of the section representing a `UseCase1` instance. Fig. 5 gives the definition of `Bold12`, as an instance of the `LSStyle` template. Users can easily specify additional styles by instantiating `LSStyle` and constraining its properties. For example, `Bold12` requires bold, no italic, and a font size of 12. Font-size ranges can also be specified.

```
abstract LSStyle
    bold ?
    italic ?
    fontSize : Int
```
(a)

```
Bold12:LSStyle
    [bold]
    [~italic]
    [fontSize = 12]
```
(b)

Fig. 5: Definition of style `Bold12`

Templates specify variability using *cardinalities*. Optional LCs are marked by `?`; e.g., `Extensions` is optional (line 14). LCs representing one or more instances are marked by + (see `FlowItem` in line 12). As usual, `*` specifies zero or more instances. The *group cardinality* `xor` specifies alternatives. For example, the `xor` in line 15 states that `Extensions` are mapped to either a section (line 16) or a text block (line 18)—a designed physical variation. Further, `UseCase2` (Fig. 4) uses `xor` (line 13) to define two alternative LCs for the use case flow—a logical variation.

The framework assumes that every document consists of three kinds of *basic PCs*: *paragraph*, *cell* (table cell), and *graphical object*. All other PCs are *composite* and built from the basic ones. For example, a *list* is a collection of enumerated or bulleted paragraphs; a *table* is a collection of cells; a *column* of a table is an arrangement of cells in a specific order; and a *text block* is a paragraph with an identifier and a delimiter.

We identified a total of 15 PCs by choosing a random sample of 20 SRDs from our collection and manually inspecting how their LSs were represented. We further analyzed these PCs and defined a mapping for each of them, including their parameters (e.g., see Fig. 6). These 15 mappings sufficed to model all of the LSs we encountered in the evaluation of the framework (see [24] for the full list). We hypothesize that only a few more mappings will still need to be defined in the future.

```
abstract SectionMapping : Mapping
    sectionTitleText:String?
    sectionTitlePattern:String?
    sectionTitleStyle->LSStyle?
```

Fig. 6: Definition of `SectionMapping`

Some mappings have a text pattern property for specifying a regular expression that a PC must conform to. For example, `ID` and `Name` both map to the section title via `SectionTitleMapping` (Fig. 3, lines 5 and 8), and specify patterns for matching and extracting only the relevant part of the title using `sectionTitlePattern`. Similar to WHISK [16], the patterns consist of three regular expressions: `regex {regex} regex`. The sequence of the three expressions need to be matched by the PC as a whole; the expression within braces specifies the portion to which the LC maps to. To simplify writing common patterns, the framework provides predefined macros; e.g., 'NUM' expands into an expression matching numbers. Thus, `ID` maps to "UC" followed by a number, e.g., "UC 1", and `Name` maps to the name following the identifier, e.g., "Select Product Category".

Each mapping has a search scope. The default scope for LSs is the entire document repository. The default scope for LCs is that of its parent. Thus, the search scopes normally follow the nesting hierarchy, as illustrated on the right of Fig. 3. For example, the `Flow` (line 10) maps to a list of paragraphs via `ListMapping`. The list will be located within the scope defined by the mapping of the parent, i.e., `SectionMapping` of `UseCase1`. Similarly, `FlowItem` maps to a paragraph via `ParagraphMapping` in the scope defined by the mapping of `Flow`, i.e., each flow item will be a paragraph in the list.

```
1  abstract NonFuncReq : LogicalStructure
2    'TableMapping
3    'SectionMapping
4        [sectionTitlePattern = "*Appendix*"]
5    [scope = SectionMapping]
```

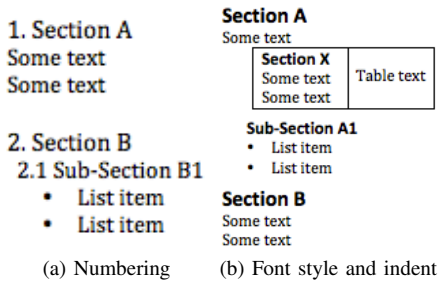
Fig. 7: Setting search scope

If needed, the search scope can also be specified explicitly, using the parameter `scope`. This parameter is useful if the instances of an LS must be written in specific sections of an SRD and therefore the search scope must be limited accordingly. Fig. 7 shows a fragment of a template for a non-functional requirement whose instances are always given as tables in appendices, i.e., sections containing the word `Appendix` in their title. Line 5 sets the search scope to `SectionMapping` defined on lines 3-4. The first mapping from the top (line 2) is assumed as the mapping for `NonFuncReq`.

By default, the LCs are unordered, i.e., the order in which they appear in the template does not influence the extraction process. If order is important to distinguish one LS from another, the `ordered` parameter of an LS or LC can be set to interpret its children LCs as ordered.

B. Document Queries

Every mapping has a corresponding document query for extracting the matching PC instances from input documents. Each query supports all parameters of its mapping. We implemented document queries for MS Word for the 15 mappings we identified. Basic PCs can be extracted directly using the Word API or by parsing the underlying OOXML. For performance reasons, we use the Word API for paragraphs and OOXML for tables.



(a) Numbering (b) Font style and indent

Fig. 8: Sample sections

We recognize composite PCs as arrangements of basic PCs. We analyzed the same 20 documents as for mappings to

develop heuristics for the recognition. The heuristics use properties of basic PCs, such as font style, font size, indentation, and cell coordinates

For example, a section is an arrangement of paragraphs, tables, and graphics in a specific hierarchy. Among others, section can be identified through paragraph numbering (Fig. 8a) or font style and sizes and indentation (Fig. 8b). As shown, sections can be embedded inside tables as well. The procedure for identifying sections traverses each paragraph serially, analyzing font style and sizes and numbering of each paragraph and building a hierarchy of sections and subsections.

New document formats, such as PDF and HTML, can be incorporated by providing mechanism for extracting only the basic PCs and their properties—the heuristics for complex elements remain the same and hence need not be re-implemented.

C. The Extraction Tool

ET interprets a given template and executes document queries for all documents in the given repository. It traverses the template in a depth-first manner and, for each LS and LC with a given mapping, it executes the corresponding document query with the parameters given in the mapping and the search scope as described in Sec. III-A. ET creates an instance of an LS or an LC for every PC instance that matches the mapping parameters. After recursively processing all children of the given instance, ET returns the instance only if all cardinality and other constraints, such as order, are satisfied; otherwise, the instance is discarded.

To overcome accidental variations, ET uses similarity-based matching for style parameters (e.g., `sectionTitleStyle`) and text-valued parameters (e.g., `sectionTitleText`). A template can specify a style-match threshold and a text-match threshold, each in the range of 0–100, to determine the exactness of style and text matching, respectively. For both thresholds, a value of 100 specifies that an exact match is required, and, if no value is specified, a default of 80 is assumed. This default worked worked well for most of the 43 documents in our evaluation—only 5 out of 36 templates specified explicit thresholds to address these few cases. Specifically, 2 templates use 90 for text matching and 3 templates used 90 for style matching.

ET's text matching uses bi-gram similarity [25], computed by breaking down the two strings being compared into bi-grams and calculating the Dice coefficient, given as $2|B \cap B'|/(|B| + |B'|)$, where B and B' are the sets of bi-grams in the first and the second string, respectively.

Style matching uses the following coefficient: $\frac{1}{3}((b = b') + (i = i') + \frac{maxF - |f - f'|}{maxF})$, where b, b' are bold parameters; i, i' are italic parameters; and f, f' are font sizes of the two styles; and $maxF$ is the maximum font size. Note that $x = x'$ yields 1 if both parameters are the same and 0 otherwise. We used $maxF$ of 20 for all of the experiments.

The range for both coefficients is [0,1] and is scaled to [0,100].

Selecting the threshold values depends on the style and text parameters used in the template and their role in uniquely identifying the LCs. If a template uses many *style* parameters without accompanying *text* parameters, then the template's style-match threshold should be set to a higher value. If a template relies on text identifiers, the text-match threshold has to be high enough to distinguish between them. For example, the text similarity between "preconditions" and "post-conditions" is 69; thus, a template using these identifiers for its LCs needs a text-match threshold of at least 70 to effectively distinguish instances of the two LCs.

IV. Evaluation

A. Analytical Evaluation

We now analyze the proposed framework in light of the requirements from the beginning of Section III.

1) A single template should work with any document format, including Word, PDF, HTML: The templates are independent of document formats. Only the infrastructure interpreting them is aware of the formats. Adding a new format involves implementing queries for the three basic PCs, as the queries for the remaining PCs are defined in terms of the basic ones.

2) Templates should be specified in a human-readable form: The templates are designed to be human-readable; both logical and designed physical variations can be expressed concisely.

3) The framework should tolerate minor errors like spelling mistakes and style inconsistencies: This requirement is achieved by similarity-based style and text matching.

4) The framework should be easily extensible with new LSs and presentations: The framework supports defining new templates and extending existing ones—directly or via inheritance.

B. Experimental Evaluation

The experimental evaluation aims at answering the following questions:

1) Can we specify LSs and extract their instances from real-world SRDs?
2) How efficient is the extraction?
3) How complex are the templates?
4) How do instances of an LS vary?
5) How does capturing variability affect the complexity of a template?
6) How critical is the need for human-editable templates?

Set-up of the experiments: We used a total of 43 documents—24 from three industrial partners, 7 from a use-case document repository [26], 6 student projects, and 6 downloaded from the Internet by searching with "Software Requirements Specification", "Software Requirements Documents", or "SRS" as keywords.

Our evaluation considers all LSs that had at least four instances within the 43 documents, giving us a total of 36 LSs. We classified instances as a single LS if they represented the same requirement concept like use case or functional requirement *and* followed the same template, as illustrated in Fig. 1. We usually found such instances within the same document. Some documents from a single source also shared templates, however. Nine industrial documents contained just a single use case instance each; all of them used the same template.

Four industrial documents contained multiple use cases each, using three different templates. Another eleven industrial documents contained system features (one document), functional and non-functional requirements (eight documents, using two templates for functional requirements), or performance requirements (two documents and one template). Eighteen non-industrial documents contained multiple use cases each; seven of these documents additionally contained functional and non-functional requirements and data objects. One non-industrial document contained functional requirements only. Each non-industrial document used separate templates.

We created 36 templates, one for each LS mentioned above. Our goal was to develop templates that capture the complete logical structure of each instance that is recognizable through document structure. For example, the actual template for the instances in Fig. 1b also recognized each cell within the "Action" and "Response" columns; deeper analysis, such as at the level of individual actions and responses, would require NL analysis techniques. We did not capture accidental physical variations in the templates; these variations are intended to be handled by the similarity-based matching.

UC 2. Add Product to Cart

1. User chooses a product to add to cart
2. User confirms the addition of product to cart
3. System adds the selected product to cart.

Extension: User can cancel the addition of the product.

Fig. 9: Highlighting of UC 2 (Fig. 1a) after extraction using template from Fig. 3

The development was iterative. For each LS, we selected a few instances and created an initial template for them. We ran ET and manually inspected the results to find false negatives and positives. ET outputs the results (instances, document name, and character locations) to an XML file. To aid the manual inspection, ET also highlights the instances in the original Word documents, using alternating colors for each LC instance (Fig. 9). Subsequently, we refined the initial template to include the false negatives and exclude the false positives. We continued the refinement until we reached the maximum recall and precision for each template.

ET and the document queries are implemented in C#. We ran all experiments on a laptop with a Core Duo 2 @2.26 GHz processor and 4GB of RAM, under Windows.

The following subsections discuss the experiments investigating the research questions and the results. We refer to the templates by their labels, T1–T36.

1) Can we specify LSs and extract their instances from real-world SRDs?: To answer this question, we measured the precision and recall for each of the 36 templates. For a template modeling a given LS, we measured two precisions: 1) $precision_{LS}$ is the fraction of retrieved instances that are actually instances of the given LS and 2) $precision_{LC}$ is the fraction of retrieved instances of the given LS that also had all their LCs recognized correctly. *Recall* is the fraction of all the instances of a given LS that were retrieved.

Using the 36 templates, ET extracted a total of 942 LS instances from the 43 documents. The average size of instances per template ranged approx. from 20 to 2,700 words. For each template, we ran the ET on a repository containing all the documents that used that template. The size of the *search space*—total number of words in the document repository—for each template ranged approx. from 1,500 to 26,800 words.

The recall was 100% for 33 templates and 97%, 95% and 83% for T21,19,24, respectively. One instance of T21 (business rule), was missed since, likely by mistake, parts of its body used 18pt font instead of the normal 11pt; this difference was beyond the style-match threshold. For T19 (use case), a single instance was missed because one of its mandatory LCs could not be recognized: the LC identifier was missing. A single instance was missed for T24 (use case) because the section could not be identified—the section heading did not use different font style or numbering.

$Precision_{LS}$ was 100% for all templates except one. T7 (functional requirement) had $precision_{LS}$ of 87% because a glossary item had the same style and parameters as a functional requirement.

$Precision_{LC}$ was 100% for 34 templates and 86% for T27 and T34. Both failures were due to PC recognition. Four instances of T27 (system feature) had one of their LCs (functional requirement) partially retrieved because of manual line wrapping, which introduced additional paragraphs. One instance of T34 (use case) had one LC (alternative flow) missing because the sub-section could not be recognized.

2) How efficient is the extraction?: The extraction time depends on many factors, including number of instances, number of LCs, search space, size of extracted instances (*extraction space*), and mappings used in the template; however, the search and extraction spaces affected the extraction time the most. Fig. 10 shows the extraction times on y-axis (excluding the time it takes to load documents into memory) for each template (lower x-axis) sorted by the sum of its search and extraction space sizes on x-axis. The extraction time for most

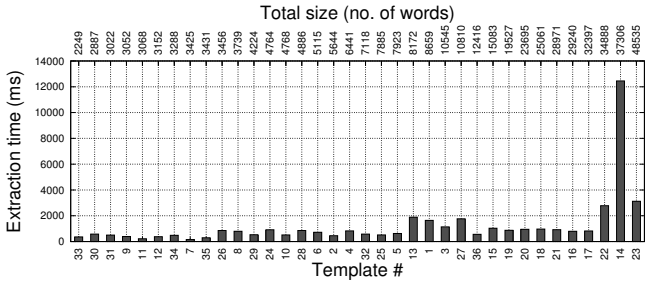


Fig. 10: Extraction times sorted by total size (search space + extraction space)

templates was less than 2 seconds, which is good for practical purposes. The longest time, 12 seconds, was for T14, whose 115 instances were spread across 5 documents. Another factor affecting the extraction time is the number of LCs: T14 has 13; T23 has only 7.

3) How complex are the templates?: We measure the complexity of a template by its length, as number of lines.

Many factors determine the template size, including number of LCs, amount of variability, extent to which details about the LS are modeled (depth, style), and kind of mappings and number of parameters used (assuming one parameter per line). Fig.11 shows the number of LCs (upper x-axis) and the template size (y-axis) for each template (lower x-axis).

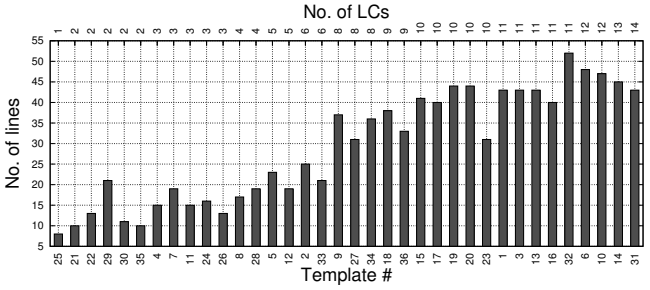


Fig. 11: Size of the templates sorted by no. of LCs

Fig.11 shows that template size is approximately proportional to the no. of its LCs. The maximum template size was 52 lines, still acceptable for human viewing.

4) How do instances of an LS vary?: We inspected each LS and their instances to analyze their variability. The variability falls into three categories: 1) LCs with variable cardinality—`?`, `+`, `*`; 2) LCs with designed physical variations—having alternative mappings or regular expressions or both; and 3) accidental variations found in the *instances* of LCs. The accidental variations include spelling and style differences relative to the spelling and style used in the majority of the instances. The 36 LSs had total of 254 LCs; 55 LCs had variable cardinality and 8 had designed physical variations—two using `xor`. Only one template used alternative LCs (flow, action-response description) to capture the same conceptual LC. Total of 24 LSs had at least one LC with variable cardinality, typically `?`. We also found 29 occurrences of accidental spelling or style differences, indicating the need to overcome such mistakes.

5) How does capturing variability affect the complexity of the template?: We conducted the following experiment to investigate this aspect. For each LS, we selected three instances: a random (using a random number generator) instance, most complete instance (with the most LCs present), and most incomplete instance (with the least LCs present). For each case (random, complete, incomplete), we created a template to capture that instance (the initial template) and compared it with the template that has maximum recall and precision (the final template); we recorded the changes in size and the performed refinements.

Fig.12 shows the difference in size (y-axis) of the initial and final template for the most complete, random, and most incomplete case for each LS (lower x-axis), sorted by the size change in the random case. The difference is shown as percentage of the size of the initial template (upper x-axis). LSs whose instances had no variability are omitted. The size-change is dependent on the type of modifications. The largest change occurred when the initial template did not contain

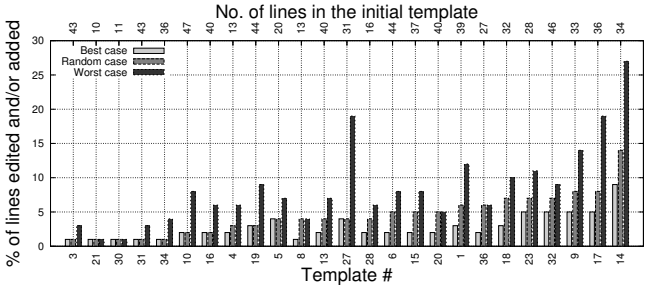


Fig. 12: Change in size of templates after refinement

optional LCs that had to be added during the refinement. In T14, 3 LCs had to be made optional (an addition of `?` to 3 lines), 2 LCs had to be added (an addition of 8 lines), and one alternating mapping was added (an addition of 3 lines) in the random case.

The nature of edits between the initial and final templates for the random case were i) `?` added: 34; ii) LC added: 18; iii) mapping modified: 5; iv) cardinality adjusted: 4; v) mapping added: 3. Most of these edits were an addition of the single character `?` to make an LC defined in the initial template optional. The second most-frequent edits were to add a missing LC.

Fig.12 and the nature of edits shows that the size of the templates did not change significantly when capturing the variability; most of the edits required to capture variability were easy to make.

6) How critical is the need for human-editable templates?: To motivate the case for human-editable template, we investigated how many sample instances would be required as input to an *induction system*—a system that induces templates from examples—to completely capture the variability for an LS. For a given LS, we created and subsequently refined a template for an increasing number of randomly chosen instances (starting from one instance) until the template reached the maximum precision and recall. We recorded the number of instances it took to get to the final template. The whole process was repeated five times for each LS in the set of 36.

Fig. 13 shows the median number of instances needed to get to the final template over the five repetitions (y-axis) for each LS (lower x-axis), sorted by the total number of instances of the given LS (upper x-axis).

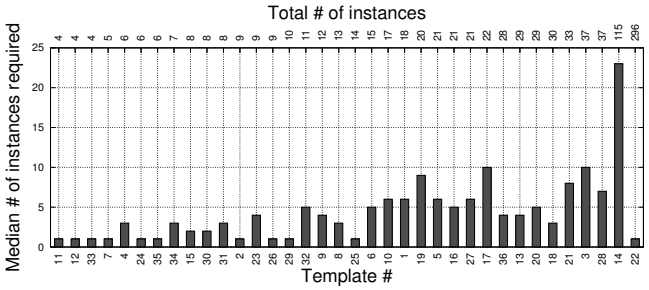


Fig. 13: Median # of instances required for induction

Some templates, e.g., T19, T17, T3, and T14, require a significant number of sample instances to fully capture the variability present in all of their instances. For T19 and T17, the number of sample instances constitutes almost 50% of the total number of instances, which makes template induction impractical. Given that most of the variability is due to optional LCs, which can be considered expert knowledge, it would be more practical to induce the templates from a few examples, and then edit it to add the variability.

V. Threats to Validity

We now discuss the threats to *internal* and *external* validity of the presented results and the measures taken to minimize such threats.

A. Threats to Internal Validity

Measuring precision and recall requires a reference of what constitutes a false positive and a false negative. We used our own judgment as a reference, which represents a potential bias. The judgement involved deciding the level of detail that should be recognized in an instance and whether a particular instance was properly recognized according to that level or not. We strived to recognize as much detail as explicit in the document structure and we carefully inspected the highlighted documents (and the XML output in cases of overlapping coloring for nested structures) to detect the false positives and false negatives. During the process, we found it relatively easy to judge with high confidence whether a particular LS or LC was correctly recognized. Additionally, we repeated the process, reliably obtaining the same results.

Template size depends on the modeling strategy, formatting, and line counting. A given LSs can be modeled in more than one way; for example, tables can be decomposed first by rows, then by columns, or vice versa. Also, templates may be formatted differently and their lines can be counted differently. To minimize this threat, we used consistent modeling strategy and formatting. Every LS, LC, mapping, and parameter constraint were defined in a single line. Furthermore, we excluded empty lines and comments from the calculation of template size.

B. Threats to External Validity

The sample SRDs may not be representative. On the other hand, over half of the SRDs were real-world documents from three of our industry partners. We also included documents from other sources. Overall, they contained quite a variety of LSs and presentation styles. We hypothesize that similar structures will be found across documents from other industrial and academic sources.

Another threat relates to the design of the study. Ideally, the data set for the study should be divided into two separate sets—the training set and the evaluation set. In the study, we used a randomly chosen subset of the data set to develop the approach and presented results over the entire data set, including the results for documents that were used for the initial development and those that were used later. The results show equal levels of precision and recall between the two, indicating that the identified mappings and the extraction process were equally applicable to the documents not used for development.

VI. Limitations and Future Work

The main assumption underlying the presented framework is that a consistent structure with slight variations is followed across the instances of an LS, which allows for specifying a template based on a few sample instances coupled with expert knowledge. The framework will bring little value when instances of a given LS are inconsistent in style and structure. However, such a case is unlikely in an industrial environment.

A. Document Queries

We assume that document queries can be written for all presentations. The accuracy of document queries depends heavily on the correctness of the identification of composite PCs. The heuristics for identification of composite PCs fail in some cases for two broad reasons: the document not having enough style/structure to distinguish between the different PCs and the applied style/structure not matching the parameters of heuristics.

The first case, when the document contains unstructured text, requires NL text-segmentation approaches to determine the boundaries of semantically related information [27]. This solution, which relies on a probabilistic model built on domain-specific examples to classify the passages, would not be as robust as relying on style/structure and it would require extensive training on different examples for different LSs.

In the second case, the identification heuristics for PCs that use the style/structure information can be machine learnt from a large set of sample PCs rather than being hand-coded, as in works on determining PCs in HTML documents that are not defined using proper tags [28], [29]. The probabilistic models learnt from examples are then used for structur extraction from text. Domain-specific vocabulary along with structural cues can also be used improve section recognition [30].

These techniques could be used in future to improve PC recognition of the presented framework.

B. Learning Templates from Sample Instances

Although Clafer provides an easy and powerful notation to write LS templates, it may not be feasible for non-technical users to write templates from scratch. Since inducing a complete template requires a large number of sample instances that cover all kinds of variations, we envision a tool that would induce an initial template based on a few instances, refine the template given more instances, and offer a natural interface for editing the template.

In the envisioned tool, the user would first declare a new LS and highlight LCs in the documents. The tool would automatically build a template with appropriate LC identifiers and mappings. An inspiration for such a system is Thresher [31], which is a browser plug-in for learning and extracting required information from arbitrary websites. The induction of the template from examples involves determining the LCs,

LC cardinality, alternate mapping groups, and inducing *pattern* parameters, such as `sectionTitlePattern`. Techniques similar to [32] can be used to determine the LCs and mapping groups. A wrapper induction system like WHISK can be used to identify the *pattern* parameters.

VII. Conclusion

We presented a framework for the specification of logical structures (LSs) and extraction of their instances from rich-text documents. The framework satisfies the requirements for a practical LS instance extraction framework and performs well in the experiments we conducted.

Applications. In general, the framework makes structured content of rich-text documents accessible for further automatic processing.

Rich-text document import. The framework lays a solid foundation for development of better import capabilities for requirements management tools, which are currently severely limited. To our knowledge, many organizations struggle with document import having to manually restructure and reformat their documents to enable automated import.

Template conformance checking. The framework can be used to implement document validators that can be used for template conformance checking and enforcement in organizations. For example, many organizations strive to unify the format and contents of software requirements documents across different projects.

Requirements management tools' features. The framework opens many possibilities for implementing typical features of RM tools, such as guided (structured) editing and def-use traceability recovery, directly in general-purpose rich-text editors. The ability to locate LS instances could also be used for referencing and navigating from code and tests.

Semantic annotation. LS instance extraction can be thought of as *semantic annotation* of rich-text documents, that is, assigning meaning in terms of the LSs and LCs to the matched parts of the documents. Such semantically annotated documents can be used as a source of knowledge in many knowledge extraction and management applications, including further context-sensitive semantic analysis of text.

Structured query. Directly related to semantic annotation is the ability to answer semantic queries directly from documents. For example, "select all use cases in which actor A participates and which refer to business rule B" or "select all use cases without a precondition".

Analysis of product line requirements. The framework can be used to implement document comparison at the LC level, which can be used for analysis of the commonality and variability in the requirements among products.

Acknowledgments Thanks to Dan Berry, Charlie Clarke, and Chrysanne DiMarco for valuable discussions. This research was supported in part by the Ontario Research Fund.

References

[1] R. Sud and J. Arthur, "Requirements management tools: A quantitative assessment," Virginia Tech, Tech. Rep. TR-03-10, 2003.

[2] P. Zielczynski, *Requirements management using IBM® Rational® RequisitePro®*. IBM Press, 2007.

[3] P. Kruchten, *The Rational Unified Process: An Introduction, Third Edition*. Addison-Wesley Professional, December 2003.

[4] A. Cockburn, *Writing Effective Use Cases*. Addison-Wesley, 2001.

[5] V. Gervasi and B. Nuseibeh, "Lightweight validation of natural language requirements: A case study," *RE*, pp. 140–148, 2000.

[6] F. Fabbrini, M. Fusani, S. Gnesi, and G. Lami, "An Automatic Quality Evaluation for Natural Language Requirements," *REFSQ*, 2001.

[7] S. Nanduri and S. Rugaber, "Requirements validation via automated natural language parsing," *HICSS*, pp. 362–368, 1995.

[8] W. Wilson, L. Rosenberg, and L. Hyatt, "Automated analysis of requirement specifications," *ICSE*, pp. 161–171, 1997.

[9] L. Mich, "NL-OOPS: from natural language to object oriented requirements using the natural language processing system LOLITA," *Nat. Lang. Eng.*, vol. 2, pp. 161–187, 1996.

[10] A. Sinha, S. Sutton Jr., and A. Paradkar, "Text2Test: automated inspection of natural language use cases," *ICST*, pp. 155–164, 2010.

[11] M. A. Song, R. A. Azriel, and K. B. Tapas, "Document structure analysis algorithms: A literature survey," *SPIE Electronic Imaging 5010*, pp. 197–207, 2003.

[12] M. Nojoumian and T. C. Lethbridge, "Extracting document structure to facilitate a knowledge base creation for the UML superstructure specification," *ITNG*, pp. 393–400, 2007.

[13] *Reqtify*, 2010 (accessed October 17, 2010). [Online]. Available: http://www.geensoft.com/en/article/reqtify

[14] N. Kushmerick, "Wrapper induction for information extraction," Ph.D. dissertation, University of Washington, 1997.

[15] ——, "Wrapper induction: Efficiency and expressiveness," *AI*, vol. 118, no. 1-2, pp. 15–68, 2000.

[16] S. Soderland, "Learning information extraction rules for semi-structured and free text," *Mach. Learn.*, vol. 34, no. 1-3, pp. 233–272, 1999.

[17] I. Muslea, S. Minton, and C. Knoblock, "A hierarchical approach to wrapper induction," *AGENTS*, pp. 190–197, 1999.

[18] S. Zheng, R. Song, J. Wen, and C. L. Giles, "Efficient record-level wrapper induction," *CIKM*, pp. 47–56, 2009.

[19] H. Garcia-Molina Cho, J. Hammer, H. Garcia-Molina, C. J., R. Aranha, and A. Crespo, "Extracting semistructured information from the web," *Wrkshp. on Mangmnt. of Semistruct. Data*, pp. 18–25, 1997.

[20] S. Kuhlins and R. Tredwell, "Toolkits for generating wrappers—a survey of software toolkits for automated data extraction from web sites," *LNCS*, pp. 184–198, 2003.

[21] F. Miller, A. Vandome, and J. McBrewster, *Office Open XML*. Alpha Press, 2009.

[22] M. Antkiewicz, K. Czarnecki, and M. Stephan, "Engineering of framework-specific modeling languages," *TSE*, vol. 35, pp. 795–824, 2009.

[23] K. Bak, K. Czarnecki, and A. Wasowski, "Feature and meta-models in Clafer: Mixed, specialized, and coupled," *SLE*, 2010.

[24] R. Rauf, "A framework for logical structure extraction from software requirements documents," Master's thesis, University of Waterloo, 2011. [Online]. Available: http://hdl.handle.net/10012/5710

[25] R. Angell, G. Freund, and P. Willett, "Automatic spelling correction using a trigram similarity measure," *Info. Proc. & Mangmt.*, vol. 19, pp. 255–261, 1983.

[26] *Use Cases Database (UCDB)*, 2010 (accessed October 17, 2010). [Online]. Available: http://www.se.cs.put.poznan.pl/knowledge-base/software-projects-database/use-cases-database-ucdb

[27] J. Yamron, I. Carp, L. Gillick, S. Lowe, and P. van Mulbregt, "A hidden Markov model approach to text segmentation and event tracking," *Acoustics, Speech & Signal Proc.*, vol. 1, pp. 333–336, 1998.

[28] K. Lerman, L. Getoor, S. Minton, and C. Knoblock, "Using the structure of web sites for automatic segmentation of tables," *SIGMOD*, pp. 119–130, 2004.

[29] M. Yoshida and K. Torisawa, "A method to integrate tables of the world wide web," *Web Doc. Analysis*, 2001.

[30] P. Cho, R. Taira, and H. Kangarloo, "Automatic section segmentation of medical reports," *AMIA Annual Symposium*, pp. 155–159, 2003.

[31] A. Hogue and D. Karger, "Thresher: Automating the unwrapping of semantic content from the World Wide Web," *WWWC*, pp. 86–95, 2005.

[32] K. Czarnecki, S. She, and A. Wasowski, "Sample spaces and feature models: There and back again," *SPLC*, pp. 22–31, 2008.