CAWIML: A Computer Assisted Web Interviewing Mark-up Language

Jose Miguel Lloret Perez

A thesis submitted in partial fulfilment of the
requirements of the
Robert Gordon University
for the degree of Master of Research

This research programme was carried out
in collaboration with Pexel Research Services

October 2016

Abstract

Computer-Assisted Web Interviewing (CAWI) is the new mode of conducting surveys through web browsers. This on-line solution extends the traditional paper questionnaire with functionality to inform the order of questions, the logic to guide question relevance and inconsistency checks to validate responses. Large scale international surveys are typically conducted by research agencies in multiple countries using CAWI systems. However, these demand for non-proprietary and platform independent questionnaire definitions that work throughout multiple survey systems.

In this thesis, we conduct a comparative analysis at two levels: one for the different Extensible Markup Language (XML) authoring solutions that capture questionnaire features; and another to explore the architecture styles for the most popular CAWI solutions. The popular hierarchical model, employed to manage the questionnaire flow, is not semantically intuitive to domain experts and lacks flexibility to allow for questionnaire design refinements. An analysis of system architectures suggests that the commonly adopted multi-page paradigm to build web pages, neither reduces the server burden nor addresses the responsiveness requirements expected from survey systems.

Accordingly to address the language shortcomings we introduce a Computer Assisted Web Interviewing Markup Language (CAWIML) that uses two schema languages to validate vocabulary, structures and relationships among XML constructs and adopts a state-transition model to manage the routing and flow of questions. CAWIML serves our Representational State Transfer (REST) system to drive the design and collection stages through a single-page web build. We present our language results from testing CAWIML on a comprehensive set of real-world surveys from Pexel Research Services and use the distribution of CAWIML's vocabulary on this sample to demonstrate its coverage of questionnaire features and effective routing support. In order to evaluate our platform, we computationally simulate both the stress test for parallel processing of requests and interviewee behaviour in terms of different user interaction response configuration levels. Results suggest that both the parallelism and variation in user behaviour can be handled within acceptable levels of usability thresholds.

Keywords: survey, questionnaire, xml, mark-up, xml schema, cai, cawi, state-transition, rest, single-page.

Declaration of Authorship

I declare that I am the sole author of this thesis and that all verbatim extracts contained in the thesis have been identified as such and all sources of information have been specifically acknowledged in the bibliography. Parts of the work presented in this thesis have appeared in the following publication:

- Lloret, J. and Wiratunga, N. (2015). Survey state model (SSM). XML Authoring of Electronic Questionnaires. In XML Prague. A conference on XML, pages 159-177. (Chapter 4)

Acknowledgements

Firstly, I would like to express my sincere gratitude to my supervisor Dr. Nirmalie Wiratunga for the continuous support of my MRes study and related research, for her patience, motivation, and immense knowledge. Her guidance has helped me in all the time of research and writing of this thesis.

Besides my supervisor, I would like to thank Pexel Research Services and Robert Gordon University for giving me the opportunity to undertake a Knowledge Transfer Partnerships (KTP) project with the possibility to study for a postgraduate qualification. In particular, I am grateful to Bruce Leslie for his insightful comments and expertise at every stage of this research, but also for his task of selecting and testing the real-world survey samples through my experiments.

On a more personal level, I must thank my patient and understanding girlfriend Daria for all her love and support. She has not only accepted me on all my bad days but also has encouraged me many times to put all my best for this research. I have also to mention perhaps the most faithful companion, my dog Hugo, for all the days and nights that he has been sat next to me.

Last but not least, I thank my family, and in particular my mum, who has been always proud of all the efforts I have made in order to look for better career opportunities.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Internet is the new medium for conducting surveys [Bethlehem and Biffignandi, 2012]. The accessibility to a large group of respondents, the low cost of distribution and the flexibility for the respondents to choose the right time to answer a questionnaire are factors that have made the Computer-Assisted Web Interviewing (CAWI) the most popular Computer-Assisted Interviewing (CAI) solution today. Although, benefits such as manually less demanding on interviews, improved data collection quality and fewer transcribing errors are already experienced with other CAI modes (e.g. Computer-Assisted Telephone Interviewing (CATI), Computer-Assisted Personal Interviewing (CAPI)), the use of CAWI is more attractive as it eliminates the need for interviewers and consequent time restrictions to interview completion. Furthermore, it is a convenient solution requiring only a computer equipped with a browser and an Internet connection in order to conduct the survey.

With the increasing popularity of CAI systems, the traditional questionnaires, composed of questions and instructions, have been extended to now include features that automatically decide the order of the questions, the conditions under which they have to be asked or inconsistency checks to be applied to responses. In order to ease the task of defining questionnaire specifications, several domain specific languages have been developed such as Blaise [Netherlands, 2015], from the Statistics of Netherlands, or Computer-Assisted Survey Execution System (CASES) by the University of California [of California, 2015]. These systems are aimed at capturing the requirements for question design, specifications for conditions and checks needed to route or traverse the questions space. In particular this routing requires that the inherent logical relationships between responses provided during the course of responding to a questionnaire is captured accurately. Similarly, different CAWI solutions have emerged providing visual interfaces for defining questionnaires with the purpose of reducing or eliminating the necessity of programmers for describing complex questionnaire's logic (e.g. SurveyMonkey, Qualtrics or NEBU).

Although a CAWI system offers several advantages to its predecessors [Bethlehem and Biffignandi, 2012], its choice comes at a price, since special attention must be paid to ensure all stages of functionality are managed by smart interfaces, i.e. the absence of interviewers to collect data or the task of authoring questionnaire's logic by designers, requires for richer and maintainable interfaces that quickly adapt to user's requirements. Accordingly, to address these demands, a clear separation of the presentation of the questionnaire constructs from the code that process questions and instructions is needed.

1.1 Research Motivation

The *correctness* of a questionnaire specification involves two tasks: one is checking that every construct conforms to a vocabulary, structure and content, i.e. the grammatical aspect, and the other consist of validating integrity relations and business specific constraints, i.e. the validation rules. Authoring language solutions such as Blaise and CASES sufficiently address both levels of correctness, however the problem arises when large surveys (e.g. European Community Household Panel, European Union Labour Force Survey or European Health Interview Survey among others) have to be conducted through several research agencies across different countries. Since these companies usually have different CAI solutions to support the life-cycle of survey research, the use of specifications that are only understandable by specific software solutions, raises the need for having a language solution that is *non-proprietary* and *platform independent*.

Two exchangeable data formats exist today to address the requirements of reusable questionnaire specifications: one is Extensible Markup Language (XML), widely used to exchange data among organisations and best known for its declarative and extensible properties; and the other is JavaScript Object Notation (JSON), increasingly used to communicate modern web applications because of its direct support by JavaScript. Intuitively, the JSON choice is more attractive due to its less verbose expressive descriptions as well as its faster performance when compared to XML [Nurseitov et al., 2009]. However, it lacks support from standards such as World Wide Web Consortium (W3C) or International Organization for Standardization (ISO) in terms of including rules that can validate correctness of documents. Accordingly for this research we make use of XML as an authoring solution making use of its different schema solutions to express constraints.

The *routing structure* of a questionnaire is another aspect that has to be carefully considered. When the flow of a questionnaire is defined, designers or social researchers commonly use skip patterns to filter one or several questions. This enables the smooth movement from one part of the questionnaire to another by removing irrelevant questions from an interview. Despite the fact that using unstructured patterns such as GOTO to express logic in programming was stated by Djisktra [Dijkstra, 1968], it is widely accepted by the software community as best avoided. In contrast, social researchers, who are typically the designers of questionnaires, have no qualms about designing questionnaires with complex logic that involve multiple GOTO constructs. Therefore, a solution to find a modelling structure that describes questionnaires in a structured manner while allowing designers to express their logic without the need for programmers is needed [Madsen, 2009].

In addition to the issues related to the specification of questionnaires, there exists different architectural properties that should also be considered when designing a CAWI solution:

- *Scalability*, known as the capacity to work under different workloads. For instance, when an interview is carried out, keeping the status of the last operation performed for each respondent is needed. As such, this is typically solved by storing interviewee's status through a session variable making it impossible to free server resources until an interview is completed. This stateful communication impacts negatively over the infrastructure making it hard to replicate and synchronise an architecture when multiple servers can handle requests from the same respondent.

- *Simplicity*, which is achieved through the separation of functionalities such as the user interfaces into a separated component within the server [Fielding, 2000]. For instance, popular CAWI solutions such as SurveyMonkey use the Model-View-Controller (MVC) design pattern for building multiple web pages on the server. However, with the introduction of the second generation of World Wide Web (WWW), these tasks can be moved to the client, i.e the browser, and help to reduce not only the time to complete requests but also simplify the server burden.

- *Portability*, that permits a software solution running on different platforms. This architectural property has particular significance for web application allocation where it is important to frequently utilise services from large data centres. For that purpose, it is not desired that a CAWI solution is tied to a particular operating system or more specifically to have a database solution that is platform dependent.

- *Reliability*, which is determined by the capacity of a system to replace a component under any failure. This property constitutes one of the most important aspects to consider since a CAWI system has to offer flexibility to allow questionnaire completion when it suits the user.

In order to address the issues discussed above in relation to the specification of questionnaires as well as the architectural properties of CAWI systems, this thesis explores the following research questions:

1. Are the current state-of-art XML authoring languages able to validate the correctness of questionnaire constructs using standard XML schema languages?

2. How can we represent the flow of questionnaires using structured patterns adapted to facilitate the questionnaire logic for routing purposes?

3. Do the popular CAWI solutions consider the architectural properties of scalability, flexibility, portability and reliability adequately?

1.2 Research Aim and Objectives

The aim of this thesis is to design and implement a new CAWI solution that provides web interfaces closer to native desktop applications. Specifically, this study is focused on the design and collection stages of surveys but also extends its functionality to cover some aspects of the management, analysis and reporting stages. The system solution has to have a clear separation of concerns, i.e. an authoring XML solution should be utilised to create questionnaire specifications through a visual design interface and similarly these descriptions should aid to automatically drive the routing logic for survey response data collection.

As such, we intend to achieve our research aim with the following six objectives:

1. Conduct a comparative analysis of the state-of-art XML language solutions that cover questionnaire definitions with a focus on the coverage of constructs and the capacity to validate correctness with standard XML schema formalisms.

2. Critically appraise current modelling approaches in terms of their ability to manage questionnaire flow definitions for the purposes of routing.

3. Develop a new XML authoring solution to better address the correctness, together with the state-transition structures necessary for routing.

4. Analyse the architecture of different CAWI solutions in order to determine

whether the necessary and sufficient properties for a CAWI system solution are induced or not.

5. Implement an architecture based on Representational State Transfer (REST) to better handle architectural properties such as scalability, simplicity, portability or reliability.

6. Conduct an evaluation at two levels, one for the coverage of questionnaire constructs and another to evaluate the capacity of the proposed architecture to work under different workloads.

1.3 Significance of Research Contributions

This study has been carried out as part of the Knowledge Transfer Partnerships (KTP) project between Robert Gordon University and Pexel Research Services [1]. This company, which is based on Glasgow, conducts telephone interviewing through its own infrastructure of units and is considered one of the largest in Europe. With the emergence of the new on-line survey mode, they are keen to consider the design and development of a new hybrid solution that permits cost reductions of training their staff to use complex interfaces.

The first significant contribution of this work has been to design a novel XML solution that combines two approaches for expressing constraints in XML. The validation of correctness, conducted in a two-step process, permits reducing or eliminating the need to rely on general purpose programming languages for checking complex rules. Moreover, as the XML language proposes the state-transition paradigm to express questionnaire's routing, not only is it better able to adapt to changes in specifications but also the use of structured patterns allows for adaptability that can support and mimic the requirements that are important to designers of questionnaires (such as social researchers).

The second significant contribution has been the system architecture that better induces the properties of: scalability, through a stateless communication among the parts; simplicity, by transferring the web pages building to the client-side; portability, through a platform independent programming language combined with a cross platform database choice; and reliability, through loose coupling components that are easy to change under any failure.

Secondary contributions of this research include the possibility to incorporate a searchable central repository of survey definitions according to our XML solution. This has

[1] http://www.pexel.co.uk/

the potential to help researchers, who often want to know what questions can be used for a particular topic [Corbett, 2011], to reuse past questions when creating new questionnaires.

1.4 Ethical Issues

The XML solution that we have designed has been evaluated against several real questionnaires provided by third parties to Pexel. These paper questionnaires remain under the company premises and cannot be disclosed. In the Appendix A.1, we have provided links to these real questionnaires implemented in our authoring language, however, we have replaced and eliminated any sensitive information in order to anonymise the third parties behind each survey. In respect to the survey used to explain the different features that a questionnaire may have (see Figure 2.1), this instance constitutes a simulated paper questionnaire and consequently does not contain any confidential information from any third party.

The evaluation of our new CAWI architecture for the gathering of responses for questionnaires presented in Chapter 6 has used randomly generated data, thus having no ethical implications. Other experiments that have been conducted during the course of this on-line survey solution such as usability testing in which the system was evaluated by Pexel's employees, may contain personal details or handling of commercially sensitive information but these are adequately described by ISO 20252 with which the company complies.

1.5 Thesis Overview

The rest of this thesis is outlined as follows: Chapter 2 introduces the different questionnaire features that can be used to specify surveys. We also discuss XML as the exchangeable data format to create reusable questionnaire definitions together with an extensive comparison of the relevant XML schema languages to validate and check correctness at grammatical and semantic levels.

In Chapter 3 we conduct a comparative analysis of the state-of-art XML languages for questionnaire design with focus on routing and personalisation aspects, notation adopted to define questionnaire expressions, schema language utilised to define constraints as well as flow modelling for question's sequencing. Additionally, we explore different architecture styles adopted by the most relevant CAWI systems in order to determine how well they support properties such as scalability, simplicity, portability and reliability.

Chapter 4 presents the Computer Assisted Web Interviewing Markup Language (CAWIML), a contribution of this thesis to address the validation of questionnaire specifications using XML schema formalisms. Here we explain the state-transition paradigm adopted to model questionnaire flow and present the Reverse Polish Notation (RPN) notation formalism to define expressions that permit filtering, computing or personalising questions and their contents. The use of CAWIML's main features are discussed using a running example of a questionnaire.

Chapter 5 presents a REST based architectural solution that implements the expected CAWI system properties. Different layers for communicating client and server, the business objects that address the survey life-cycle stages as well as its non-relational persistence solution to address high demands of data are presented. Additionally, the Single Page Application (SPA) paradigm adopted to build the client interfaces directly in browsers is discussed.

Evaluation is in Chapter 6 with results organised under two themes: coverage of questionnaire constructs by CAWIML analysed on fifteen real questionnaires; and simulated load testing of increasing numbers of concurrent users with focus on different response time thresholds.

We conclude this thesis in Chapter 7 with a summary of our main contributions and desirable extensions for future work.

Chapter 2

Background

This chapter is intended to introduce the reader to the terminology and concepts in the context of CAI systems. We have focused our efforts on XML as the standard for exchange and representation of electronic questionnaires. Although we could have considered JSON as a more attractive representation format due to its direct mapping to JavaScript objects, its simplicity and its less verbosity for using across a web application, we have discarded it because as far as we know, there is no standard schema formalism that permits the validation of correctness without relying on general purpose programming languages.

The rest of this Chapter is structured as follows: Section 2.1 uses an example questionnaire to describe the different features that are used in surveys. The Section 2.2 explains the origin of XML as well as the requirements needed to consider a file as XML well-formed. Section 2.3 discusses the different schema paradigms that can be used to create constraints over well-formed XML documents and finally, Section 2.4 introduces the reader to XPath query language since it is widely used for specifying sophisticated constraints.

2.1 Electronic Questionnaires

Questionnaires are defined as instruments to collect data. Typically, they are composed of questions and instructions to guide the flow through an interview. With the introduction of CAI systems, these have been extended with additional features. There are two studies that explore the different constructs required to specify questionnaires. The first approach, proposed by Katz [Katz et al., 1997], describes the different tasks involved in creating specifications for questionnaires whereas the second proposal, introduced by Bethlehem [Bethlehem, 2000], uses an object-oriented paradigm to better understand all the features of an electronic questionnaire.

In order to describe all the different constructs that may be used to construct questionnaires, we present in Figure 2.1 a questionnaire intended to unify the efforts from Katz and Bethlehem. The most common types of questions are *single-response*, *multiple-response* and *open-ended* (e.g. Q1, Q3 and Q5 respectively) whilst *grid* (e.g. Q4) unlike the common types is naturally more cognitively demanding on the interviewee since more than one question in the form of rows is asked [Bock, 2013].

Usually questionnaires are divided into *sections* where *intro* statements (e.g. INF1, INF2, END) become helpful to establish the context and introduction to a part of the questionnaire. For instance, in this example, there are two sections used to organise the set of questions, the outer section for INF1, Q1, Q2, Q3, Q4, Q5, INF2 and END and an inner for Q6a which can be asked multiple times.

The *instructions*, in bold font, are normally included to manage questionnaire routing according to interviewee responses. There are three such routing constructs describing the flow through the questionnaire in our example:

- *skip* feature allows the interviewee to skip over questions to move on to another question. This can be an *unconditional* skip, as in the case of skips associated with responses in Q1 or Q2; or a *conditional* skip, presented in Q5.
- *filter* constructs are based on a logical expression involving the responses to one or more questions. They are described as if-then-else statements, for instance the instructions attached over Q5 or INF2 and also known as complex branching.
- *loop* feature permits repeated execution of a questionnaire sub-part. For example, the instruction over Q6a defines a loop that iterates a maximum of four iterations over the selected responses from Q2 and Q3.

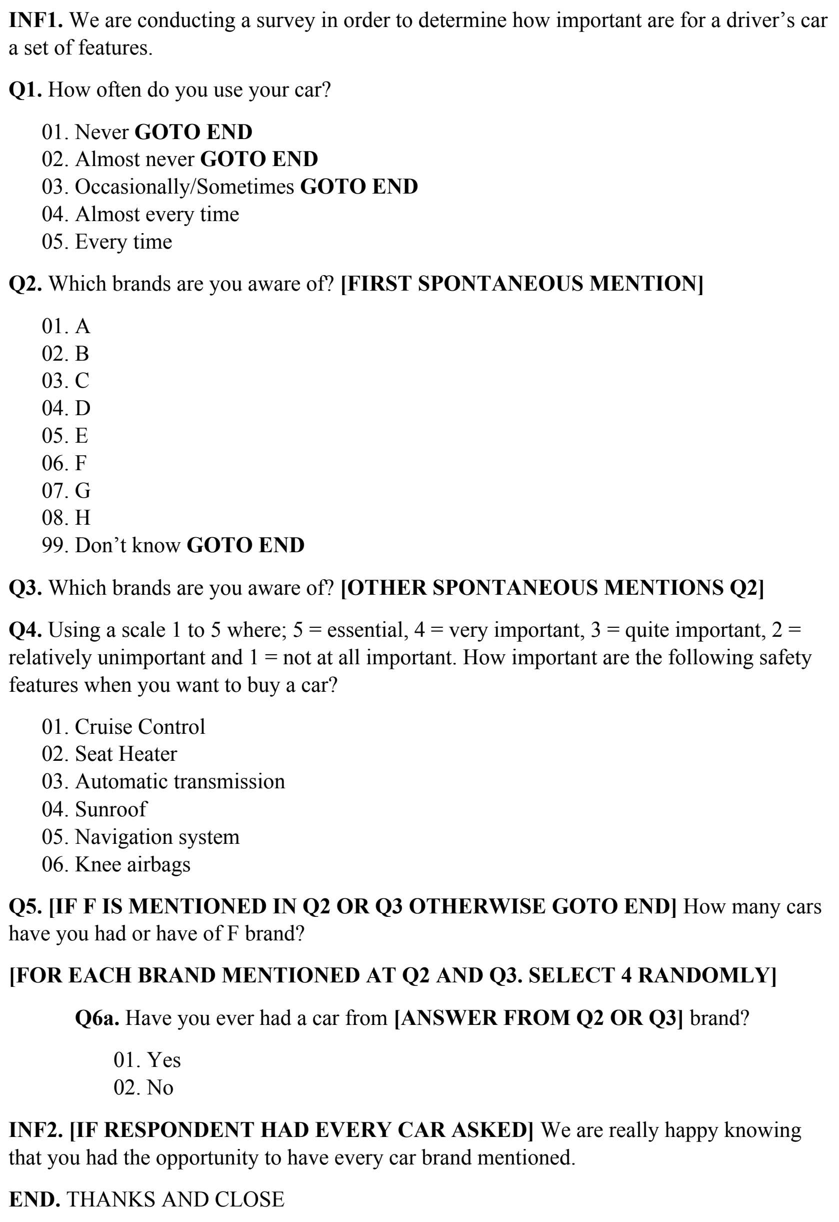
In addition to the constructs listed above, there are further features which are less frequent but nevertheless are also featured in questionnaire design.

- *Piping* which allows retrieving responses from one or more previous questions as part of the text for another (e.g. question text for Q6a) or generating a set of responses based on an expression (e.g. Q3 responses are generated automatically according to those responses not mentioned in Q2).
- *Randomising* or *Rotating* features which reduce bias evasive responses by altering the data order presented to the user [Warner, 1965] (e.g. the random presentation

of the question Q6a).

- *Computation* constitutes the execution of an arithmetical expression and its assignment over a reference variable. Typically it is used to communicate responses among sections. For example, after Q6a a stateful variable is needed to capture all the affirmative responses for Q6a. This global variable is used later as part of the filter condition associated with INF2.

- *Check* allows the functionality to establish whether or not a logical expression is being satisfied and thereafter to notify the respondent about any inconsistencies. These constructs are helpful for a post-validation of constraints. For instance, a case in which a respondent answered that she is a non-smoker and later responds to a question as spending money on buying cigarettes. In this example, this construct could warn the respondent about such as inconsistency.



INF1. We are conducting a survey in order to determine how important are for a driver's car a set of features.

Q1. How often do you use your car?

01. Never **GOTO END**
02. Almost never **GOTO END**
03. Occasionally/Sometimes **GOTO END**
04. Almost every time
05. Every time

Q2. Which brands are you aware of? **[FIRST SPONTANEOUS MENTION]**

01. A
02. B
03. C
04. D
05. E
06. F
07. G
08. H
99. Don't know **GOTO END**

Q3. Which brands are you aware of? **[OTHER SPONTANEOUS MENTIONS Q2]**

Q4. Using a scale 1 to 5 where; 5 = essential, 4 = very important, 3 = quite important, 2 = relatively unimportant and 1 = not at all important. How important are the following safety features when you want to buy a car?

01. Cruise Control
02. Seat Heater
03. Automatic transmission
04. Sunroof
05. Navigation system
06. Knee airbags

Q5. [IF F IS MENTIONED IN Q2 OR Q3 OTHERWISE GOTO END] How many cars have you had or have of F brand?

[FOR EACH BRAND MENTIONED AT Q2 AND Q3. SELECT 4 RANDOMLY]

Q6a. Have you ever had a car from **[ANSWER FROM Q2 OR Q3]** brand?

01. Yes
02. No

INF2. [IF RESPONDENT HAD EVERY CAR ASKED] We are really happy knowing that you had the opportunity to have every car brand mentioned.

END. THANKS AND CLOSE

Figure 2.1: Fully functional paper questionnaire instance

2.2 Extensible Mark-up Language (XML)

The web browsers use HyperText Markup Language (HTML) to build the presentation of information. They interpret each HTML tag in order to display words, images or videos. However, this language is neither able to capture the description of contents, i.e. semantics, nor permits extending its mark-up with tags that are application-specific. In order to address these limitations, W3C developed XML. This meta-language, not only permits representing semi-structured data [Bray et al., 2008a] but also serves as a medium of communication that is widely used across Internet applications.

XML files are composed of elements, attributes and relationships [Varde et al., 2010]. Listings 2.1 shows an XML document example that describes a small questionnaire with two sections and their respective routing. Specifically, an *element* is used to represent an entity which is enclosed through a start and end tag (e.g. survey, section, intro, single, multiple, routing and variable). An element may contain character data, other elements or a mixture of both within. Also, this may have *attributes* whose presence is limited to the start-tag exclusively (e.g. id or ref).

```xml
<survey>
    <section id="section1">
        <multiple id="Q1"/>
        <single id="Q2"/>
        <intro id="Q0"/>
    </section>
    <section id="section2">
        <intro id="Q3"/>
    </section>
    <routing ref="section1">
        <variable ref="Q0"/>
        <variable ref="Q1"/>
        <variable ref="Q2"/>
    </routing>
    <routing ref="section2">
        <variable ref="Q3"/>
    </routing>
</survey>
```

Listing 2.1: XML example case

The *relationship* captures the nesting of elements and permit creating for simple to complex structures. For instance, the section element has 'section1' as id attribute and contains multiple, single and intro elements as children. Similarly, a more complex structure is the survey root element that contains section and routing elements within.

XML only requires for documents to be *well-formed*, i.e a valid XML document according to W3C must have:

- a unique single root element in which every other element is contained within,
- properly nesting of all the elements,
- presence of start and end tag for every element,
- and value for attributes enclosed with quotes.

However, XML by itself is merely a standard notation which does not restrict the elements and attributes permitted or the structures and content allowed. Therefore, in order to differentiate well-formed documents from those that a valid according to an XML authoring language, it is needed to formally define a schema by using an XML schema language.

2.3 Schemas Languages

XML schema languages are formal languages used to define schemas. A schema represents the definition of the *syntax* and *semantics* that is allowed for an XML authoring language [Mller and Schwartzbach, 2006]. The syntax consist of defining the vocabulary of the language as well as the different structures in which the elements and attributes are permitted. In contrast, the semantics determines whether or not the syntax expressed in an XML document is meaningful.

The use of a schema language to formalise an XML authoring solution permits obtaining instance specifications that are non-ambiguous and benefits the software program that parses the instances since the number of errors can be reduced or eliminated. In order to check whether an XML document conforms to a specific XML language, a schema processor, which is an implementation program for an XML schema language, takes two arguments: the XML document or instance; and the schema (see Figure 2.2). This processor, automatically decides the validity of a document and in addition, for those documents that are not valid, it provides a report document explaining the reasons of its failure.

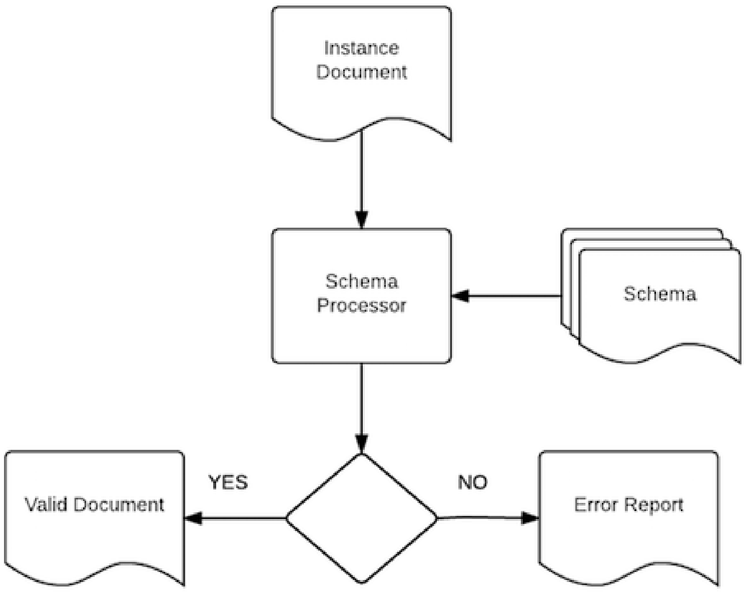


Figure 2.2: XML validation process

During the process of validating XML documents against an XML language, four different levels of validation [Sthrenberg and Christian, 2010] are checked in order to ensure the correctness of an XML specification:

1. *Structure*: This level validates the mark-up introduced as well as the order and occurrences of elements and attributes.

2. *Data-types*: At this stage it is determined whether the content defined for elements and attributes conforms to the data-types defined in the schema.

3. *Integrity constraints*: This level verifies uniqueness of identifiers as well as checks that any reference points to existing keys. Usually, the keys and references to these, are values that are set for attributes.

4. *Business rules*: At this level, any additional data constraints [Van der Vlist, 2006] that cannot be categorised in any of the above mentioned levels is checked. This level is very specific to the application domain in which the authoring XML language was designed for.

The XML schema languages are divided into two approaches: grammar-based and rule-based [Sthrenberg, 2013]. The grammar-based schema languages specify the mark-up, structure and data-types expected for XML documents. For instance, they restrict the presence of attributes and elements, the type expected for values, the order in which elements or attributes may appear in the document or the minimum and maximum number of occurrences allowed. In contrast, the rule-based schema languages mainly ensure that the relationships among elements and attributes is consistent. However, they can also constraint XML documents by incorporating business specific rules that make specifications meaningful for the application domain.

Different schema languages exist today aimed to express constraints that are checked at any of the above mentioned validation levels. The three most popular grammar-based schema languages are Document Type Definition (DTD), XML Schema Definition (XSD) and Regular Expression Language for XML New Generation (RELAX NG). Regarding the rule-based language formalisms, Schematron (SCH) is the only candidate representative. Table 2.1 summarises the validation levels that these schema languages support. Throughout the next following subsections, the schema languages most relevant to ensure the correctness of XML specifications are explored.

	DTD	**XSD**	**RELAX NG**	**SCH**
Structures	Basic	Yes	Yes	No
Data-types	Limited	44 built-in	Not directly	Not directly
Integrity Constraints	ID IDREF	unique key key-ref	Not directly	Yes
Business rules	No	No	No	Yes

Table 2.1: Validation levels supported by the most popular XML schema languages

2.3.1 Document Type Definition (DTD)

DTD is the built-in schema language since the first working draft of XML [Bray et al., 2008b]. There are many XML authoring languages that are specified using DTD such as Extensible HyperText Markup Language (XHTML) or Survey Interchange Standard (Triple-S). The last mentioned, is known as the standard XML language for describing questionnaires. DTD has basic support to define syntax for documents since it is not possible to constraint the order and occurrences of elements when character data and elements are allowed within an element structure. Regarding the data-types, it is neither capable of constraining character data (e.g. id attribute for section starts with section followed by any number) nor allows frequent used types such as integer, date or Uniform Resource Identifier (URI). In regard to the integrity constraints, it provides ID and IDREF for uniqueness and referential key respectively. However, any defined ID works for the entire document not being able to specify a more restricted scope (e.g. the question id cannot be duplicated across different sections). Moreover, compound keys, i.e. those that combine different attributes to form a key is not supported either. With regards to business rules, there is no mechanism that permits validating semantics over syntactically valid XML instances.

DTD was designed before the name-space mechanism was introduced. This feature, which permits reusing other schema specifications to create a new XML language, is not provided and consequently it is not only hard to create modular schemas but also it is difficult to evolve them. Furthermore, a schema defined through DTD does not use XML notation so checking that a specification of an XML language conforms to DTD requires the use of non-standard XML tools to verify its validity.

2.3.2 XML Schema Definition (XSD)

XSD is the recommendation schema language for W3C [Sperberg-McQueen and Thompson, 2012]. It was designed to be more expressive than DTD and introduces name-spaces, data-types and an XML syntax for restricting well-formed XML files. As such, it permits importing structures of elements through the URI mechanism, uses XML syntax to express constraints, which eliminates the need to learn a new proprietary syntax and provides a richer built-in data-types system. Additionally, to strengthen the data-types set, it allows the definition of customised types through restriction (e.g. Uniform Resource Locator (URL) type is a subset of the string base type) and by extension (e.g. an element single question extends from question element by adding a set of open-closed response elements).

This schema language unlike its counterparts, provides a better mechanism to express integrity constraints for XML documents. XSD overcomes the limitations of ID, IDREF that are present in DTD by using a subset of XPath Query Language (see Section 2.4) to define unique values for elements or attributes and to reference these through *key* and *key-ref* features respectively. For instance, Listings 2.2 defines a key to ensure the uniqueness of a question (e.g. intro, single, multiple or open) in the context of a section from a survey through a *selector* element. Moreover, to determine what attribute of a question element is used to check the uniqueness, the *field* element is utilised. Similarly, in that example, the key-ref uses the selector to select a variable within a routing element in which the attribute ref is used to hold the reference to a unique question key. Note that key-ref also defines a refer attribute which points at a key element previously defined in the schema for an authoring language.

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" elementFormDefault="qualified ↩
    ">
  <xs:element name="survey">
```

```
        ...
        <xs:key name="questionKey">
            <xs:selector xpath="section/intro | section/single | section/multiple | ↩
                section/open"></xs:selector>
            <xs:field xpath="@id"></xs:field>
        </xs:key>
        ...
        <xs:keyref name="questionKeyRef" refer="questionKey">
            <xs:selector xpath="routing/variable"></xs:selector>
            <xs:field xpath="@ref"></xs:field>
        </xs:keyref>
    </xs:element>
    ...
</xs:schema>
```

Listing 2.2: Key and references in XSD

The above mentioned integrity constraints could be used to verify that the document from Listings 2.1 does not actually introduce duplicate keys for question ids neither specifies a reference to a non-existent question id within the routing.

XSD is one of the most used schema language for defining XML languages, mainly because supports enough expressiveness to define structure, data-types and integrity constrains for XML documents. However, it is hard to learn due to the amount of features that provide and does not introduce any mechanism to express business rules. [1]

2.3.3 Regular Expression Language for XML New Generation (RELAX NG)

RELAX NG is a schema language developed within the Organization for the Advancement of Structured Information Standards (OASIS). This language was built with the design principles of simplicity and expressiveness. As such, its syntax to express constraints for XML documents is closer to plain English instructions [Van der Vlist, 2003]. RELAX NG is part of the ISO/International Electrotechnical Commission (IEC) 19757. Specifically, it is the candidate schema language for describing structure and content of XML documents.

The data-types and integrity constraints are not directly supported by the language

[1]The version 1.1. introduces assertions to express business rules through a subset of XPath expressions (attributes, children and descendants of an element) [Gao et al., 2012]. However other existing relationships among XML documents such as parent, ancestors or siblings are unable to express which restrict its expressiveness and usage.

so it relies on external schema languages to address these tasks. Specifically, the data-types for elements and attributes are usually imported from XSD whereas the integrity constraints are weakly supported by having a special compatibility data-type library that uses ID and IDREF from DTD. Regarding the business rules, as this language is categorised as grammar-based, it does not directly offers features for specifying additional constraints.

2.3.4 Schematron (SCH)

SCH is a schema language designed by Rick Jelliffe [Dodds, 2001]. This language differs from grammar-based XML schema languages in that it is rules-based, i.e. defines assertions within a rule context that are evaluated over XML instance documents. This language, likewise RELAX NG, also takes part from the ISO/IEC 19757 and constitutes the standard reference to ensure that documents match the rules defined through a schema. In SCH, the constraints are specified via XPath query language (see Section 2.4). The use of these expressive queries, permits defining any kind of tree relationships (e.g. child, parent, ancestor) that are inherited in XML documents making this schema language unique in terms of expressing semantics.

The core constructs of Schematron are patterns, rules and assertions. Listings 2.3 describes an integrity constraint that check the uniqueness of section identifiers. The pattern element is used to group different rules. A rule requires a context attribute which constitutes the path used to evaluate one or more asserts (e.g. this rule is fired for every section within the survey element). An assert is specified through the test attribute and addressed to check a constraint within the context rule described. For instance, the assert example counts the preceding section's identifiers that are equals to the identifier set through the place holder $id (e.g. let element) in order to determine whether is zero or not. A false result for the assert, displays the customised message within that element.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<schema xmlns="http://purl.oclc.org/dsdl/schematron" queryBinding="xslt2">
   <pattern id="integrity_constraints" >
      <rule context="/survey/section">
         <let name="id" value="@id"/>
         <assert test="count(preceding-sibling::*[@id = $id]) = 0">
            duplicate key found for section id <value-of select="$id"/>
         </assert>
      </rule>
   </pattern>
</schema>
```

Listing 2.3: Integrity constraint expressed in SCH

The validation of XML documents against a SCH schema can be carried out either by using an Extensible Stylesheet Language Transformations (XSLT) [Clark, 1999] processor or by using an XPath implementation. The XSLT approach, represented in Figure 2.3 requires performing two steps:

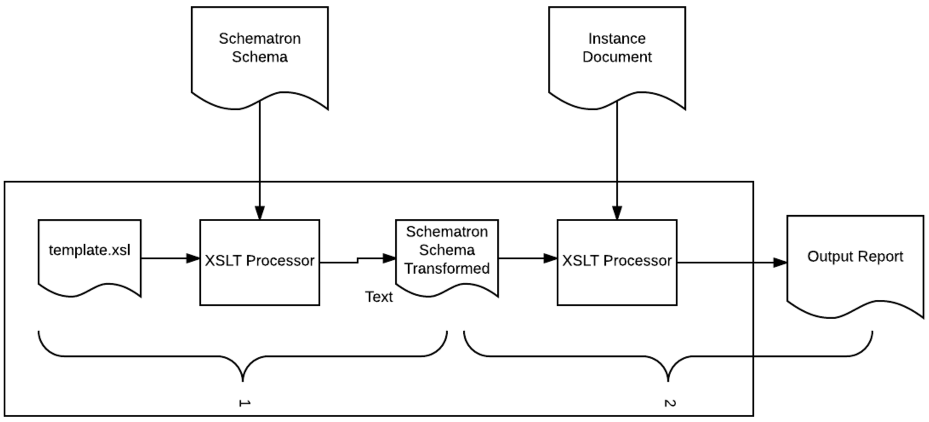


Figure 2.3: Schematron XSLT validation process

- First, an XSLT processor gets a SCH schema together with a pre-defined SCH template, provided by Academia Sinica Computing Centre [2], in order to produce a transformed SCH schema understandable by the XSLT processor.
- Second, the transformed schema, together with the XML document is passed to the XSLT processor to finally produce the output report based on the rules and assertions in the original SCH schema defined.

The XPath implementation, unlike its counterpart, is faster for validating XML documents since the transformation step is not needed. However, it has less functionality due to XSLT-specific functions such as document() or key() which are not supported. For instance, *document* function permits validating rules among XML documents.

The high expressiveness of SCH not only makes this schema language the best for representing any integrity constraint [Murata et al., 2005] but also extends its capacity to define any sort of business rule. Regarding the structure, although it is able to describe many instances of grammars [Jellife, 2007], this constraining level is best addressed through grammar-based languages. In regard to the data-types, SCH does not

[2]https://www.ascc.sinica.edu.tw/en/about/overview.html

directly provide any built-in type. However, these may be simulated using the XPath functions.

2.4 XPath Query Language

XPath is a query language that uses path expressions to navigate through XML documents. It is the recommended query language for XML documents by W3C. SCH was the first schema language that used XPath for expressing rules and after XSD borrowed this idea to specify its integrity constraints. The syntax used in this query language is based on a location path, i.e. a similar concept used in file systems, which consist of a sequence of location steps:

- axis to direct the navigation with the relationships parent, children, siblings, ancestors or descendants [Clark and DeRose, 2015] since this language treats the XML files as trees of nodes,
- node test that permits filtering the path to a specific node or element and
- predicates which allow selecting only those nodes with specific properties or attributes.

To better understand how this syntax works, we show some examples using the XML document from Listings 2.1. For instance, in order to select the section 1 element and children, we could express /survey/section[@id='section1'] with the node test /survey/section and the predicate specified in brackets, but we could be more specific about retrieving only the ids of the questions /survey/section[@id='section1']/child::node()/@id. Note, that the first example obviates the axis, i.e. the expression has been defined using the *abbreviated mode*, which is more compact. In contrast, the second one explicitly uses the axis to guide the location deeper in the sub tree /survey/section and it is known as *full syntax mode*.

2.5 Conclusion

In this Chapter we have explained questionnaires through a representative example that covers the spectrum of questionnaire constructs. These instruments, composed of questions and instructions, guide an interview's flow and are separated into three categories: *content* to describe questions grouped into sections; *routing* to decide question's sequence; and *personalisation* to adapt the questionnaire structural and sequencing properties to an interviewee given their real time responses.

Our study of different XML schemas has shown that grammar-based schema languages are adequate to cover correctness levels such as structure and data-types, however they fail to address integrity and business constraints. In contrast, the rule-based schema formalisms and in particular SCH is best suited to specify semantics.

Chapter 3

Literature

Several authoring languages have been proposed to specify questionnaires. The two most popular solutions are CASES, that provides structured and unstructured patterns for routing and Blaise, which promotes the design of skip free logic in favour of loops, conditional branches and modules. Although these languages offer adequate construct coverage, their adoption remains restricted due the proprietary nature of the language and lack of being a standard interchange format.

As part of the aim to design and implement a new CAWI solution for Pexel, we have done research at two levels: one to seek responses in terms of correctness of questionnaire specifications. For that purpose, we have conducted a comparative analysis of the state-of-art XML languages in terms of routing and personalisation constructs, notation style adopted for expressions, schema language formalism utilised, underlying flow modelling for routing or the survey stages that they support; and the other to determine the properties induced by the architectural style adopted for the most relevant CAWI solutions.

The rest of this Chapter is structured as follows: Section 3.1 introduces the XML authoring language solutions for questionnaires before comparing them in Section 3.2. Finally, Section 3.3, reviews the architecture styles of Blaise and SurveyMonkey CAWI systems and introduces the methodology and testing methods useful for evaluating our new CAWI solution proposed.

3.1 The XML Languages

The most relevant XML languages addressed to cover questionnaire constructs are Triple-S, Simple Survey System (SSS), Questionnaire Definition Language (QDL) and Data Documentation Initiative (DDI). The following sections provide a brief description of each authoring solution before conducting a comparative analysis.

3.1.1 Survey Interchange Standard (Triple-S)

Triple-S is aimed to represent the content aspects of surveys [Gerrard et al., 2011] in order to make it easier to transfer data and meta data among CAI systems or any analysis software package. This language is considered the standard for representing social surveys and at least fifty registered implementers may be found on its website [Gerrard et al., 2015]. Although its focus has been to provide a comprehensive coverage of survey functionality, we have found one case study where Triple-S has been adapted to describe a business decision making tool [Wright, 2007].

3.1.2 Questionnaire Definition Language (QDL)

QDL is a language built for Tool for Analysis and Documentation of Electronic Questionnaires (TADEQ) project whose aim is at building a software tool that represents questionnaire specifications in a human-readable format [Bethlehem, 2000]. This tool can operate in either textual or graphical mode. It is suited to designers who want detailed information of the constructs and interviewers who need documentation to help them when they are conducting interviews. Although this project has some implementers like Blaise that has its own converter, this project has been abandoned and there is no longer support for this language.

3.1.3 Data Documentation Initiative (DDI)

DDI emerged in 1995 as an international project to create standardised meta data to document social science datasets [Rasmussen and Blank, 2007]. It emerged to solve the problems of documenting datasets and introduces XML as the exchangeable data format to be both, machine-readable and human-understandable. It has an important value for analysis and archiving since permits describing data at two levels: *variable* level, which consist of describing the different variables involved on the research; and *study level*, since it allow describing the population that links to the stored information.

In its third version was introduced the description of social surveys. Since then, the Austrian Bureau of Statistics (ABS) has been experimenting with this exchangeable format for their design tool for questionnaires. Similarly, National Institute of Statistics and Economics Studies (INSEE), that uses Blaise CAI system for collecting data has shown interest on using DDI as a standard to communicate different disciplines in the data collection field. For that purpose, a Proof of Concept (POF) was created by de Bolster to convert from Blaise language specifications to DDI 3.1. From that experiment, it was concluded that neither DDI instances are human readable nor compatibility between different versions is considered. For instance, when valid DDI instances for questionnaires were verified against the schemas of the newer version at least 365 errors occurred [de Bolster, 2013].

3.1.4 Simple Survey System (SSS)

SSS is a CAPI system solution developed for Research Triangle Institute (RTI) which has its own XML language to address content, routing and personalisation features of surveys [Bethke, 2008]. This language unlike Triple-S, QDL or DDI has a robust schema to represent the logical and arithmetical expressions based on functional programming style.

3.2 Comparative Analysis of XML Languages

In this section we detail a comparative analysis of the XML authoring languages used to define questionnaire specifications. For that purpose, we explore the coverage of routing and personalisation constructs in Section 3.2.1 and 3.2.2 respectively. The relevant flow paradigms used to capture the routing sequence for questionnaires is explained in Section 3.2.3. The constraining level coverage of the schema languages used to define these XML languages is addressed in Section 3.2.4. Section 3.2.5 studies the three notation to describe logical and arithmetical expressions through an example from a real questionnaire and finally the survey life-cycle stages in which every XML language is focused on is detailed in Section 3.2.6. Table 3.1 summarises the above mentioned aspects that will be treated on the next sections.

3.2.1 Routing Constructs

The filter construct is well represented by SSS, QDL and DDI whereas it is partially described by Triple-S since this language is only able to represent simple logic involving

Category	*Feature*	*Triple-S*	*SSS*	*QDL*	*DDI*
Routing	**Skip**	No	No	Yes	No
	Filter	Partial	Yes	Yes	Yes
	Loop	No	Yes	Partial	Yes
	Check	No	No	Yes	No
	Computation	No	Yes	Yes	Yes
Personalisation	**Piping - text-fill**	No	Yes	No	Yes
	Piping - carry-forward	No	No	No	No
	Randomising	No	No	No	Partial
	Rotating	No	No	No	Partial
Other	**Flow Paradigm**	N/A	Hierarchical	Hierarchical	Hierarchical
	XML Schema Language	DTD	XSD 1.0	DTD	XSD 1.0
	Expressions Notation	N/A	PreFix	Infix	Infix
	Survey Stage	Analysis/Reporting	Design/Collection	Design	Analysis/Reporting

Table 3.1: Comparison of XML languages for electronic questionnaires

only one variable.

In contrast, the skip logic is only defined by QDL which considers the fact that for questionnaire designers it is difficult to express conditional statements [Katz et al., 1997]. However, as far as we know, the tool that implements this language does not offer support since it is difficult to implement conditions without having restrictions over the type of jumps allowed [Bethlehem and Hundepool, 2004]. The other languages do not cover this construct either because the skips can be reversed and use filters instead [Bethke, 2008] or since surveys designed without unstructured patterns are easy to modify, share and understand [Spencer, 2012].

The loop construct is well represented in SSS and DDI. Despite the fact that DDI offers three types of loops (RepeatUntil, RepeatWhile and range loop) [Thomas et al., 2009], we consider that SSS is more flexible than DDI to define expressions since it embodies a simple functional programming style through the use of XML tags. In spite of QDL, it only offers support for range loop but does not consider iterations over lists (see the instruction after Q5 in Figure 2.1). Regarding Triple-S, it does not directly define any mechanism to iterate, although offers an interesting feature to relate data collected from hierarchical surveys (e.g. survey responses for a household survey followed by responses of the property members defined in another survey) [Wright, 2007].

The check construct is only featured in QDL and this is likely to be motivated by the need to describe multi-item constraints to check consistencies among answers to related questions [Katz et al., 1997] or alternatively because this XML language is strongly related to Blaise system [Bethlehem, 2000]. Regarding the computation feature, other than Triple-S, all others languages support its representation.

3.2.2 Personalisation Constructs

Although SSS and DDI offer mechanisms to describe *text-fill* aspects, the other piping feature, i.e. the *carry-forward* has not been considered. This construct, which permits describing operations to retrieve selected on unselected responses from previous questions as part of the responses for other questions (see Q3 from Figure 2.1), may help to better adapt surveys for each respondent. Accordingly, we consider that a specification language cannot leave out this construct. For instance, the popular SurveyMonkey CAWI system, we have observed that it implements this personalisation feature as part of its interface functionalities.

Regarding the randomising and rotating features, these constructs are only partially covered by DDI permitting to change the order for content constructs such as single and/or multiple question responses. However, more sophisticated patterns like selecting a specific number of responses after randomising/rotating or reordering only a subset of responses is not taken into account. For instance, the instruction above Q6a from Figure 2.1 not only specifies to randomise the elements selected but also requires to iterate a maximum of four times.

3.2.3 Routing Flow Paradigms

The questionnaire's flow captures the sequence for the question constructs defined in a questionnaire. The designers or social researchers commonly use skip patterns that can be interchanged with filters to express the logical order of elements in a questionnaire. We have explored the directed graphs, Petri Nets and the hierarchical paradigms in order to establish the suitability for representing sequence logic.

The *directed graph*, was explored in the past by Fagan et al. in order to analyse the use of skip constructs on surveys. These graphs based on *nodes* and *arcs* are able to represent question and response choices respectively. When additional information such as the conditions that determine the ordering of questions are added to these graphs, they become flowcharts which result in tools to document and understand questionnaire's flow [Jabine, 1985]. The directed graphs, permit validating the correctness of data gathered from surveys by detecting whether or not a question response is missed or is not applicable. These verifications are conducted applying different graph properties [Fagan and Greenberg, 1988]. Although the properties from graphs can be used to model the flow of questionnaires, as far as we know, this paradigm has not been used to formally define questionnaire specifications in XML.

In more recent work, Petri Nets have been applied to visualise and analyse complex questionnaires [Rolke, 2010]. Figure 3.1 illustrates the Petri Net representation on part of the questionnaire from Figure 2.1 in relation to question Q1 and the skip construct that permit navigating either to Q2 or END. Here a *place* (e.g. Q1, Q2 and END) captures the static information from a question whereas a transition (e.g. 01, 02, 03, 04, 05) represents a response choice. Additionally exist *arcs* to connect places to transitions (e.g. arrow connecting Q1 to 01) and vice versa (e.g. 01, 02 and 03 to END). A Petri Net permits validating the *reachability* of places, i.e. checking whether or not a set of responses given for questions correspond to a valid state of the net. However, this modelling approach is only limited to finite domain, i.e. it is only applicable for single and multiple questions, but becomes hard to manage open questions since an arc connecting places to/from transitions has to be specified for any possible value expected.

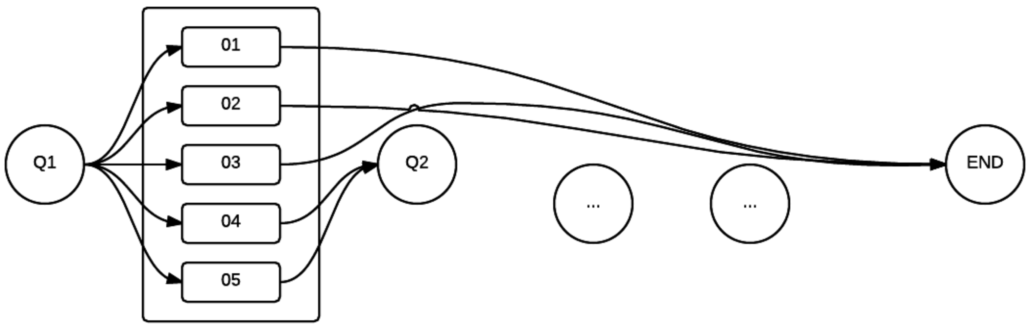


Figure 3.1: Petri Net instance

SSS, DDI and QDL adopt a hierarchical modelling approach where the use of skip pattern is avoided in favour of structured constructs (if-then-else, loops and sequences) as advocated by Dijkstra [Dijkstra, 1968]. As such, the routing structure of surveys can be seen as trees permitting not only the identification of the path followed but also determining all the circumstances under which a question may be triggered [Spencer, 2012]. Algorithm 1 describes in pseudo-code the routing of the questionnaire from Figure 2.1 according to the hierarchical modelling. Note how the nesting of constructs is used to avoid skips. However, this nesting is not naturally well suited to reusability.

The benefits of eliminating skip patterns are universally acknowledged by high-level programme languages architects, however designers or social researchers still specify questionnaires through documents using skips to navigate from one question to another [Katz et al., 1997], not only because they do not have programming skills but also because the final client interested in the survey, demands plain English instructions. As the hierarchical approach replaces the use of skip constructs, there is an evident *gap* between what the social researchers design versus the *code* that a CAWI system uses to execute a survey. Costigan states that having two sources of specification, i.e. the code and the document requires duplication of effort and it is prone to error [Costigan and Elder, 2003]. To unify these two sources, Spencer proposes that researchers should be trained to use structured patterns [Spencer, 2012], i.e. avoid the use of skip constructs. However, in practice, motivating social researchers to adopt this practice remains a problem [Costigan and Elder, 2003].

The ability to introduce changes to questionnaires is another factor to consider since clients often demand the modification of survey logics when the data collection is already taking place. This involves having a modelling approach that addresses the *adaptability* appropriately. It is evident from the above hierarchical representation that strong coupling between the inner and outer filters increases with increasing skip

Algorithm 1: Hierarchical modelling example

```
INF1;
Q1;
if NOT(Q1 IS_SEL '01' OR Q1 IS_SEL '02' OR Q1 IS_SEL '03') then
    Q2;
    if NOT(Q2 IS_SEL '99') then
        Q3;
        if NOT(Q3 IS_SEL '99') then
            Q4;
            if Q2 IS_SEL '06' then
                Q5;
                for each Q2 SEL do
                    Q6a;
                end
                if HAD_CAR >1 then
                    INF2;
                end
            end
        end
    end
end
END;
```

constructs. As dependencies among constructs are like to impact negatively to changes to questionnaire's flow, there is a need to explore a less intrusive model.

3.2.4 Schema Languages

The different XML languages reviewed are formally defined through an XML schema. Specifically, they use grammar-based schemas to define the vocabulary, structure and data-types expected for instances defining a questionnaire. Throughout this section, the XML example from Listings 2.1 will be used to discuss features supported by DTD and XSD.

QDL and Triple-S are defined using DTD (see Section 2.3.1). This schema formalism has two weaknesses: inability to express complex structures for elements; and limited number of data-types whereby common types such as number or date are not supported.With regards to the integrity constraints, the ID and IDREF mechanisms, offered for describing uniqueness of elements and references to valid identifiers respectively, are not robust enough for expressing semantic constraints over XML documents. Specifically, the lexical space of an identifier is global to the entire document (e.g.

the question id cannot be duplicated across different sections) and as Fan and Simeon state, this is a very strong restriction for a schema language [Fan and Simon, 2003]. Moreover, the IDREF is not able to point to a specific key identifier (e.g. it cannot be described such that the ref attribute for a routing element links to the identifier attribute of a section). Regarding the business rules level, there is no such feature to constraint the additional semantics for XML documents.

SSS and DDI use a more expressive schema language that was built to address the limitations of DTD. Most structures are supported in XSD. Its very rich set of data-types goes farther than simply supporting only common type such as string, boolean, decimal, integer or date to permitting the definition of any customised type through regular expressions. Regarding the integrity constraints, although a more expressive mechanism using key and key-ref through XPath expressions is provided (see Section 2.3.2), not every possible relationship existing in XML documents can be defined [Gao et al., 2012]. Specifically, in XSD the XPath expression for a *xs:selector* can only use children and descendants of the element in which it is defined. In addition, the *xs:field* restricts the XPath expressions to only select attributes [Van der Vlist, 2006]. For instance, it is not possible to constraint the variables Q0, Q1 or Q2 such that they can point to questions defined in 'section1'. With respect to the business rules, only XSD 1.1, which is not used neither in SSS nor DDI, supports assertions to express additional semantics for XML documents. However, the XPath expressions are equally limited to attributes, children and descendants of the node where the assertion has to be checked.

The grammar-based schema languages are adequate to specify mark-up and syntax for XML documents, however they are insufficient to express integrity constraints or business rules. Accordingly to address these issues it is best to use a rule-based schema languages such as SCH which has no restrictions on XPath expression definitions. Therefore, if the well defined patterns from grammar-based languages are combined with the expressiveness of rule-based schemas [Van der Vlist, 2006] [Costello and Simmons, 2015] [Dongwon and Chu, 2000], it is possible to create an XML authoring solution that is better suited to handle the different validation stages, i.e. a language that ensures the correctness of questionnaire specifications without the necessity of relying on programming languages to validate complex semantic constraints.

3.2.5 Expressions Notation

The routing constructs contain logical and arithmetical expressions applicable for filters, loops, checks and computations. Typically these expressions are defined using infix style. For instance, questionnaire languages such as QDL and DDI use this mode. Consider the following infix notation example, that follows the normalised convention proposed by Hughes [Huhges, 2007] as an attempt for describing expressions in paper questionnaires:

ASK IF: [QB = 'Male' AND (QA = 'Scotland' OR QA = 'Wales') AND (Q5 = 'Less' OR Q5 = '1-3')], *[Ask males in Scotland and Wales who eat less than or equal to 3 portions of vegetables per week]* [1]

Firstly, the filter construct (ASK IF) as described, secondly an algebraic expression is defined and finally a plain English instruction is provided in order to be more readable. Although this notation constitutes the natural way of thinking, the inclusion of brackets becomes a necessity in order to override the operators precedence.

Unlike the infix notation, with prefix and postfix notations where the operators are written before and after their operands respectively. Here the operators are no longer ambiguous with respect to the operands and consequently the parentheses may be obviated. Accordingly our example in prefix notation can be stated as follows:

AND AND = QB 'Male' OR = QA 'Scotland' = QA 'Wales' OR = Q5 'Less' = Q5 '1-3'

And the postfix representation is:

QB 'Male' = QA 'Scotland' = QA 'Wales' = OR AND Q5 'Less' = Q5 '1-3' = OR AND.

SSS uses the prefix mode through a functional programming style inspired by Lisp Language which is a more desirable approach to choose. However, when prefix mode is compared with postfix notation, it is less efficient because the order that the operators have to be evaluated does not strictly follow the left-to-right order, i.e. the operators placed on the left, must wait until the intermediate operations on the right part are solved (e.g. the first and second AND operators). This involves moving backward and forward through the structural representation of the expression, increasing the number of operations. With respect to postfix mode, there is no need for operands to wait (e.g. the last two operators OR, AND) have the intermediate results solved before being evaluated and consequently it is computationally most effective.

[1]Expression extracted from survey 08 of the Appendix A.1

3.2.6 Survey Stages

Survey research is a process that goes beyond asking a set of questions [Corporation, 2012]. As part of this research process, the survey may be divided into five stages (see Figure 3.2). These stages are *design*, that involves the formulation of questions needed for the achievement of the aim and objectives. The *collection*, where an instrument such as a questionnaire is used in order to obtain responses to the formulated questions. The *management*, aimed at monitoring the results gathered and helpful to determine the presence of any problematic question. The *analysis*, that consist of conducting different statistics to study the data collected and finally the *reporting* where different artefacts such as documents, tables, graphs or charts are used to present the data gathered in order to help decision making. This stage may serve also as a mechanism to export data and meta data into different formats such as Comma-separated Values (CSV) that may be utilised by more sophisticated software vendors such as SPSS to conduct advanced analytics.

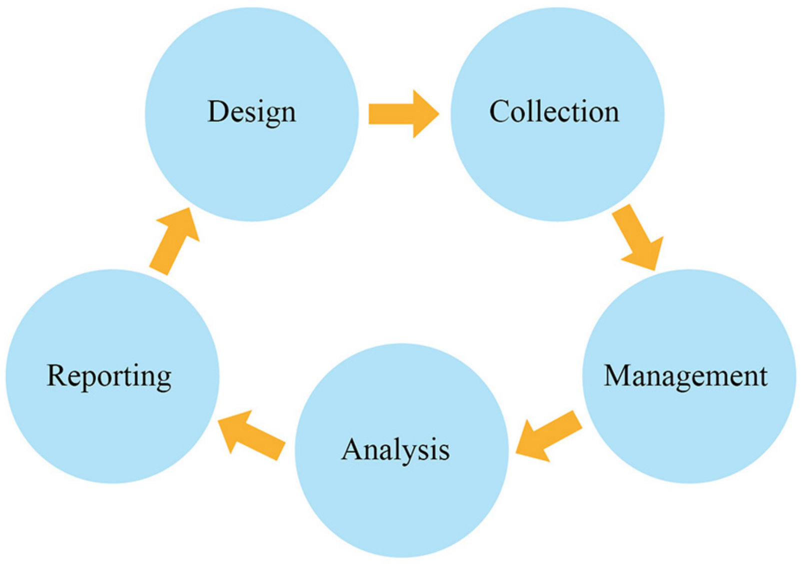


Figure 3.2: The 5 stages of survey research

The authoring languages studied are focused on one or more stages of the survey research cycle. For instance, Triple-S and DDI pay special attention to analysis and reporting stages since they provide structures that permit exporting the data collected for a questionnaire as well as its associated meta data. In contrast, QDL is best suited to address the design stage since it is built as part of a project to develop a tool for documenting questionnaire specifications [Bethlehem and Hundepool, 2002].

Whilst SSS, although was built to facilitate the creation of intuitive interfaces for supporting the design stage, it is more adequate to drive the collection stage given its formalism to describe expressions which offers more advantages than its counterparts and permits reducing the number of validations that have to be carried out before executing a questionnaire.

3.3 CAWI Systems

CAWI systems are based on a Client-Server (CS) architecture style that uses Hypertext Transfer Protocol (HTTP) to communicate between client and server. Several refinements for this basic architecture have been made over the years such as the *distributed objects* style (e.g. Common Object Request Broker Architecture (CORBA)), which uses the object-oriented paradigm, for client server communication by encapsulating data and behaviour together [Overdick, 2007]. This approach is not appropriate for distributed environments since it asserts too much responsibility on the client which has to manage the life-cycle of objects, i.e. operations such as create, copy, move or destroy, and the server that has to rely on these operations performed.

A more modern architecture style is Service Oriented Architecture (SOA) that defines services to address the different functionalities of a system. In this approach, there are two agents involved: the *provider*, which implements a defined business function that operates independently of any other service provider; and the *consumer* which uses the service [MacKenzie et al.,]. The interactions among the agents are performed through different communication protocols such as Simple Object Access Protocol (SOAP) and using standard exchangeable formats like XML.

As Pexel company demands the design and development of a new CAWI solution that can offer potential advantages over competitors, we consider that it is important to evaluate different architectural properties for the existing CAWI systems in order to determine how they address the simplicity, portability, reliability and scalability. Similarly, as the software system is network-based, it is crucial to review different test strategies for performance since these may help to estimate response times that in term, impact usability.

The rest of this section is structured as follow: Section 3.3.1 reviews the architectural style of different CAWI systems and Section 3.3.2 explores different testing methods and parameters used to simulate scenarios for performance testing of CAWI systems.

3.3.1 CAWI System Architectures

This Section explores Blaise and SurveyMonkey architectures in order to determine the architectural properties being adopted by CAWI systems. However, other software solutions such as CASES, have not been explored due to the absence of publicly available documentation.

Blaise

The architecture style of Blaise is Windows Communication Foundation (WCF), which is an implementation of SOA. In this style, the interactions among components is carried out by sending data through asynchronous messages that can be either XML formatted or using complex streams of binary data. The CAWI solution offered by Blaise, separates the functionalities into different roles (see Figure 3.3) [Segel et al., 2015]. The most important server roles are: *survey manager*, that holds the creation and publishing of surveys; *data entry server*, used to validate input and routing logic; *data server*, that performs read/write operations into the databases; *session server*, that stores and controls the active interviews; and *web server* used to host and serve web pages to the end users.

The WCF architecture style of Blaise induces simplicity by making a clear separation of concerns that leads to have services less complex and interdependent. Additionally, as this approach defines interfaces to communicate the different roles, any change at any component should not impact negatively into its consumers. However, the portability, reliability and scalability are not carefully considered. Specifically, the portability is not present due to the platform-dependent architecture of WCF that only works under Microsoft environments. Regarding the reliability, it may be affected by the fact that the data server role makes the entire system vulnerable under any failure due to its inability to be set up with multiple physical or virtual machines [Volguine, 2013]. In respect to the scalability, the presence of a session server role to keep the state of every interview ongoing, not only prevents the server to free resources but also makes it harder to manage, replicate and synchronise state changes under a multi-server configuration.

SurveyMonkey

SurveyMonkey is the world's largest survey company [Groom, 2014]. Its CAWI solution is written in Python and its core features are separated into different services. Most of the services communicate through a JSON web Application Program Interface (API)

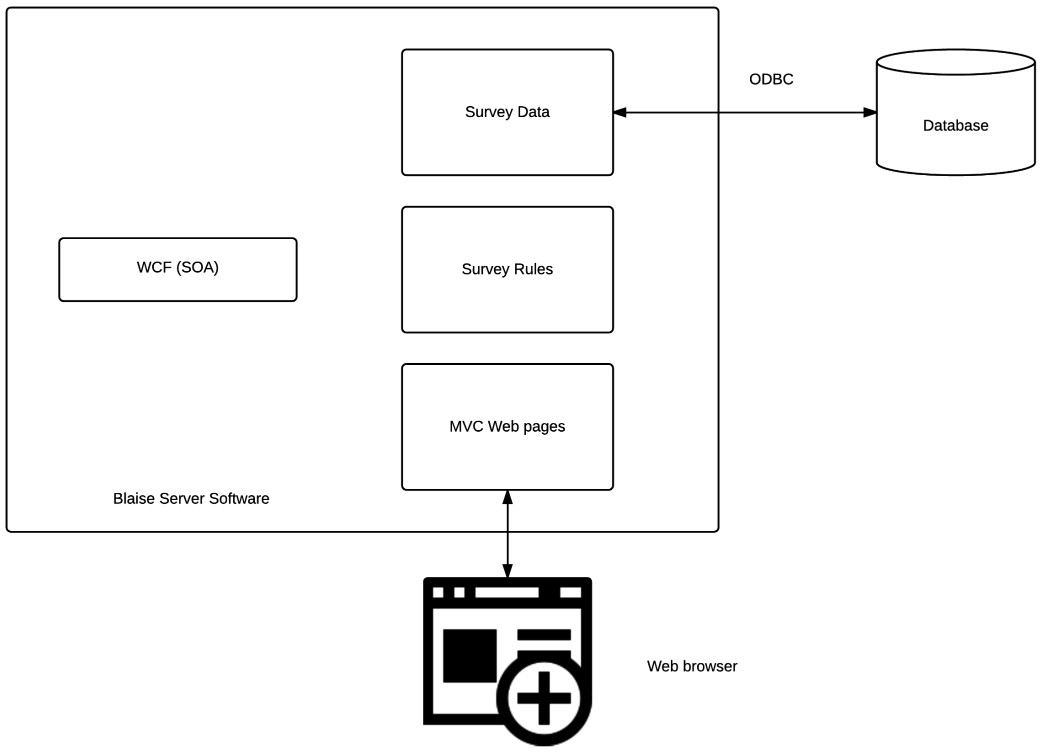


Figure 3.3: Blaise Architecture

over HTTP/HTTPS. SurveyMonkey implements SOA through Web Server Gateway Interface (WSGI) (see Figure 3.4). In this style, the web server is set up to receive client's requests and return responses back. The web server itself, does not directly creates a response but invokes the web application that produces a response based on the URL requested and pass it back to the web server. The server ultimately sends to the client. The WSGI specifies the rules that need to be implemented by both sides, i.e. the web server and MVC Framework. SurveyMonkey utilises Pyramid as the web application framework to produce many of its services.

The WSGI architecture style of SurveyMonkey induces simplicity and scalability. The simplicity is achieved through the use of pyramid MVC web framework which permits loose coupling due to the separation of concerns. This software pattern, promotes parallel development (e.g. developers may focus on models, controllers or views). Regarding the scalability, unlike Blaise, there is no session persisted on the server, i.e. its stateless configuration based on a token-based authentication, promotes flexibility to scale. In respect to the portability, the application code is written in a cross-platform language, however we have not found enough information to determine whether or not the reliability or portability are induced.

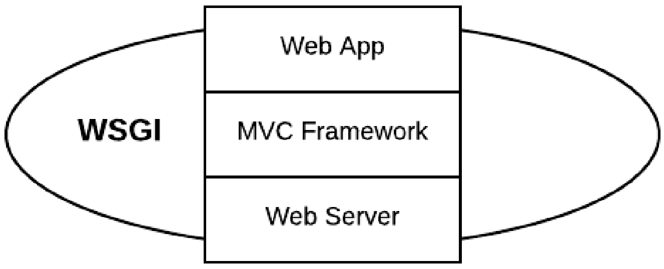


Figure 3.4: SurveyMonkey Architecture

3.3.2 Performance Testing

CAWI systems are addressed for large group of respondents that can access simultaneously to complete a questionnaire at any time. As these systems are network-based applications, it is desired to conduct performance testing in order to determine the capacity of the system to work under different configurations.

The performance testing consist of creating different test plans by varying parameters such as number of concurrent users or the complexity of the survey selected. Typically, the methodology plan to execute performance testing consist of:

- Selecting a survey, where the length and complexity helps to predict the performance of the system to deal with questionnaires with similar constructs.
- Designing a specific scenario either by randomly responding to questions or by taking the most common sequence of questions answered by survey respondents [Volguine, 2013].
- Varying the number of concurrent respondents and finally running the scenario under the parameters chosen.

Segel et al. discuss a *stress test* strategy in order to verify the capacity of a web deployment of Blaise system [Segel et al., 2013]. In that experiment, they vary the number of concurrent respondents and introduce the *thinking time* concept consisting of setting up a random distributed time that simulates the amount of time that takes respondents to answer questions. Additionally, they explain the importance of selecting an adequate time in which the maximum number of concurrent users will be reached in order to avoid unrealistic simultaneous accesses.

A more recent study carried out by Volguine pursues a stable and responsive on-line survey respondent experience through the introduction of three more test strategies a part from stress testing. These are: *normal capacity* consisting in monitoring the system for two hours under an average load level for a day, *peak* during two hours under a maximum load expected for day and *endurance* with a significant load level during eight hours [Volguine, 2013]. Although Volguine offers a full suite of tests to assess performance, reliability and responsiveness of Blaise, it is not clear whether the thinking time is included or not. Therefore, this can lead to a biased testing situation that is not close to a real scenario in which an interviewee thinks before responding to a question. Moreover, the use of normal, peak and endurance tests strategies assumes that the tester knows what are the system level usages which is not always known, specially for CAWI systems that have never been in the market.

3.4 Conclusions

We have critically analysed four XML authoring languages to determine the coverage of questionnaire constructs. The routing features, are only fully addressed by DTD, hardly addressed by Triple-S and addressed with only structured patterns by SSS and DDI. Regarding the personalisation constructs, none of the languages explored offers support for carry-forward functionality, well represented in interfaces of CAWI solutions such as SurveyMonkey. In respect to the survey stages supported, Triple-S or DDI best suit the export of survey data and meta data to other social disciplines. In contrast, SSS offers a more robust expression notation that reduces the validations needed to execute a questionnaire. However, although this prefix notation eliminates the use of parentheses, it is less efficient when compared to the postfix notation mode.

The inability of grammar-based schema languages to address the correctness of semantics for questionnaire specifications, is evident in all the authoring solutions reviewed. Particularly, the DTD schema formalism existing in Triple-S and QDL, is very limited in its ability to express integrity constraints. Similarly, the use of XSD through languages such as SSS and DDI does not permit expressing any kind of relationship existing in XML files, raising the need for using rule-based standard formalisms or general-purpose programming languages to address the semantic.

Regarding the representation of question sequence, the hierarchical modelling adopted by QDL, SSS and DDI does not adequately facilitate the questionnaire logic for routing purposes. Specifically, this paradigm does not only become unsatisfactory to unify the paper specification of a questionnaire against the code produced but is also difficult to make changes when skip patterns have to be reversed, which impacts negatively over the adaptability principle.

The study of different architectural styles for CAWI systems, shows for instance that Blaise does not support the scalability, portability and reliability properties adequately. In contrast, the stateless configuration adopted by SurveyMonkey promotes flexibility to scale systems. Respecting the simplicity, although it is sufficiently covered in both systems by separating the tasks into different services, we consider that building of web pages could be transferred to the client using the responsive SPA paradigm. Finally, from all the testing methods reviewed for CAWI system evaluation, we find that the stress testing is best to determine the capacity of a system when different parameters such as number of concurrent interviewees or think time metrics varies.

Chapter 4

CAWI Mark-up Language

This Chapter presents CAWIML as an alternative authoring language to specify questionnaires only using standard XML schema languages. Particularly, it uses a state-transition paradigm for question's sequence and is intended to facilitate the questionnaire routing logic more adequately than the popular hierarchical model. RPN, is the expression formalism utilised for describing routing and personalisation constructs indistinctly.

The rest of this Chapter is structured as follows: Section 4.1 introduces the state-transition routing structure. Section 4.2 explains the postfix notation mode as the formalism for questionnaire expressions, followed by Section 4.3 that explains our XML authoring solution. Finally, XML details for content, routing and personalisation constructs expressed in CAWIML are presented in Section 4.4.

4.1 The State-Transition Modelling Solution

The state-transition modelling approach is our proposal to better address the routing requirements involving design-code equivalence and adaptability criteria (see Section 3.2.3). This model, widely used for the specification of reactive systems [Androutsopoulos et al., 2008], is inspired by the Extended Finite State Machine (EFSM) [Alagar and Periyasamy, 2011].

An example of the state-transition model, represented in Figure 4.1, describes the questionnaire presented in Figure 2.1. It contains various types of states, represented by ellipses, that are linked through transitions to form state models (e.g. the Outer and Inner rectangles).

Each state model contains variables that reference questions defined in a section (e.g. INF1, Q1, Q2, Q3, Q4, Q5, INF2 and END for the Outer section or Q6a for the Inner). The scope of these is local to a state model and therefore their references do not exist outside. In order to reference variables through any state model, this paradigm defines global place holders, known as fields, that permit sharing data across different parts of a questionnaire (e.g. HAD_CAR describing an integer number of cars that the respondent had).

The states are addressed to perform single operations and these may be categorised as follows:

- *Simple* states are used to present the questions to the respondent and to store responses in variables (e.g. every state prefixed with 's').
- *Composite* states refer to a defined state model (e.g. c0 and c1 for Outer and Inner state models respectively) and are useful for reducing coupling among questions in a survey.
- Pseudo states are normally used to take routing path decisions through the evaluation of boolean expressions. Specifically, there are *if* and *for* states to decide the conditions under which questions are asked and *check* states to validate inconsistent responses. Additionally, the *computation* state unlike its counterparts is utilised to update place holder variables through *arithmetic* expressions.

The transitions connect a state source with a state target to create a questionnaire's flow (e.g. arrows connect ellipses). If a transition does not define a boolean expression, it is assumed true whenever the state source of the transition is reached. In contrast, if a boolean expression is defined, this means that the expression has to be evaluated in order to determine its truth (e.g. Q1 '01' IS_SEL Q1 '02' IS_SEL OR Q1 '03' IS_SEL OR). Here we propose the use of postfix notation mode as the expression formalism for all the expressions used in the design of a questionnaire.

The state models have an initial state that determines what state is executed first (e.g. s0 and s8 for Outer and Inner respectively) as well as one or more ending states (e.g. 'sink0', 'sink1' or 'sink2'). Similarly, the state-transition model requires a state model that marks the beginning, i.e. an entry point for the model to capture a questionnaire's flow (e.g. 'c0' and 'sink0' states).

Accordingly, we formalise the state-transition model that is applied to questionnaire routing as follows:

$$M = \langle Q, V, T, I, E \rangle \tag{4.1}$$

where,

1. $Q(\neq \emptyset)$ is a finite set of states.
2. V is the set of state variables. Every variable $x \in V$ may be accessed at every state $q \in Q$.
3. T is a finite set of transitions. A transition $t \in T$ is represented as $q \xrightarrow{[c]} q'$, where $\{q, q'\} \in Q$ and c is a boolean expression involving variables of V defined in pre-state q. The absence of c is interpreted to true.
4. $I \subset Q$ is the set of initial states. Every composite state has an initial state and consequently there is a set of initial states if the model contains composite states.
5. $E \subset Q$ is the set of end states. These states may be *sink* addressed to finish a state model or *terminate* to interrupt and finish the execution of a state-transition model.

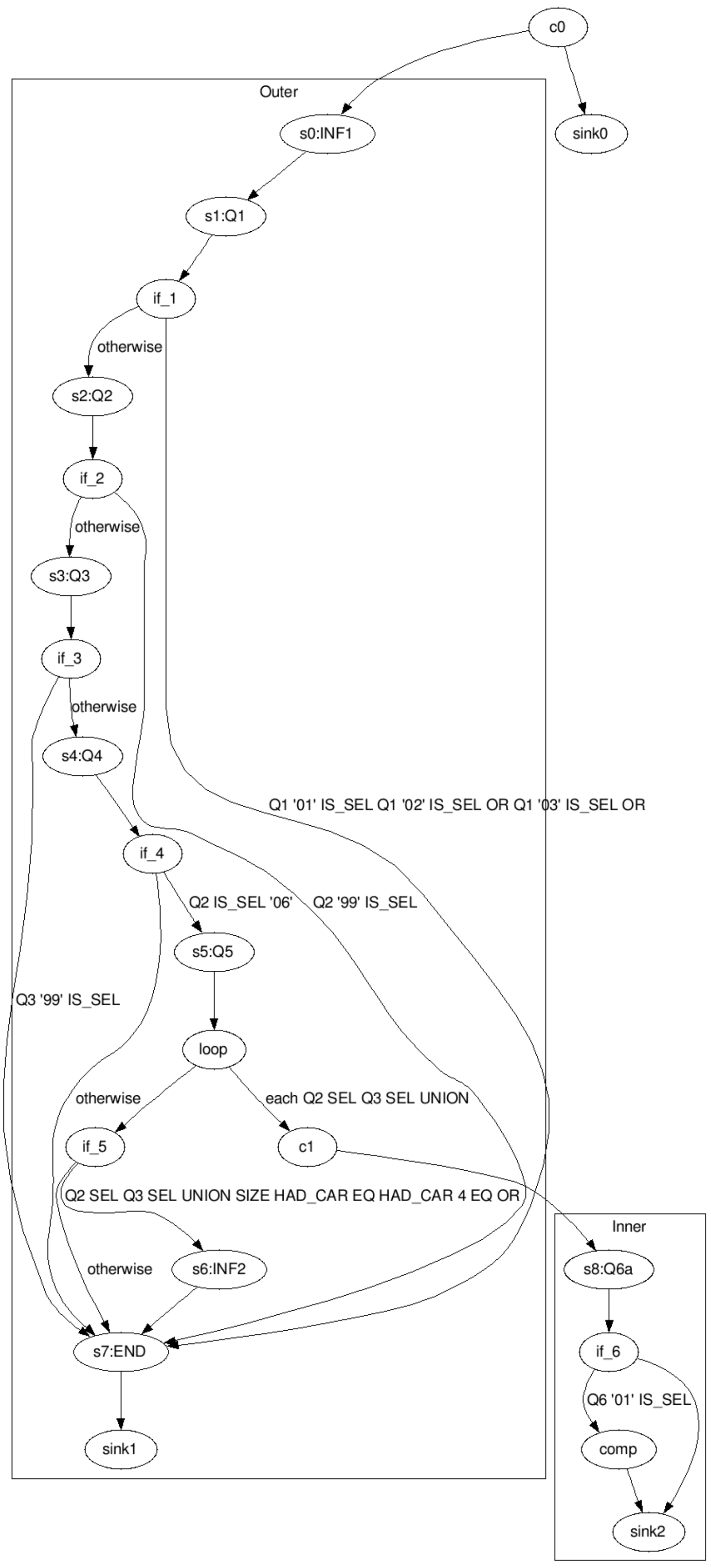


Figure 4.1: State-Transition for the paper questionnaire in Figure 2.1

Name	Operand 1	Result
POS	Integer/Decimal	Integer/Decimal
NEG	Integer/Decimal	Integer/Decimal
INC	Integer	Integer
DEC	Integer	Integer
NOT	Boolean	Boolean
EMPTY	String/List	Boolean
SIZE	String/List	Integer
SEL	List	List
UNSEL	List	List
ALL	List	List
VALUEOF	String	String
	Integer	
	Decimal	
	List	

Table 4.1: Unary Operators of CAWIML

4.2 The RPN Notation

The postfix notation, also known as RPN, is the notation formalism that we have adopted to define the logical and arithmetical expressions applicable not only for routing constructs such as filters, loops, checks and computations but also for describing text-fill and carry-forward personalisation constructs. Its simplicity to evaluate any kind of expression, the non-ambiguity for operators precedence and its efficiency in terms of number of operations to perform, make this formalism significantly better than infix or prefix modes (see Section 3.2.5). The RPN formalism has two types of operations: *unary*, that expect one and only one operand; and *binary* which require two operands. By combining these two categories, it is possible to express from simple to complex questionnaire logic constructs.

Table 4.1 lists the set of unary operators that CAWIML provides to express typical questionnaire constructs. The last four operators (e.g. *SEL*, *UNSEL*, *ALL* and *VALUEOF*) are particularly useful for operations carried out through piping constructs. For instance, Figure 2.1 specifies a carry-forward piping to populate the unselected answers for Q2 as part of responses for Q3 (e.g. Q2 UNSEL). Similarly, the example questionnaire describes a text-fill construct for the Q6a text. This piping feature, which may be formally expressed as ITERATOR VALUEOF, describes the current loop iterator value since Q6a may be executed multiple times during the process of conducting an interview.

The set of binary operator constructs are listed in Table 4.2 and differentiates the operations into four subtypes:

- *equality and relational*, used for conditional statements such as filter, loop or check;
- *conditional*, utilised to join two boolean expressions;
- *arithmetical*, for operations such as addition, subtraction, multiplication or division; and
- *list* to perform operations like UNION or INTERSECTION of sets.

The commonly used binary operator *IS_SEL*, checks whether or not a response from a single, multiple or grid question has been chosen. Operators such as UNION or INTERSECTION are crucial to express personalisation features such as complex carry-forward constructs as these permit the join of selected, unselected or all responses from different question types (like single or multiple).

4.3 The XML Language Solution

The limitations for all grammar-based XML language solutions reviewed (see Section 3.2.4) require the use of programming languages to validate semantics for questionnaire specifications. These restrictions have led us to design and implement a new authoring language that is non-proprietary, platform independent and use standard formalisms to define structure, data-types, integrity constraints and business rules. This language, uses the state-transition model for routing logic (see Section 4.1) together with RPN notation to define expressions either for routing or personalisation constructs (see Section 4.2).

CAWIML uses XSD to define structure and data types together with SCH to express integrity constraints and business rules. We have chosen XSD due to its well-defined patterns to express vocabulary and structures, its rich set of data-types, its use of XML to define constraints and because it is the recommendation schema formalism proposed by W3C (see Section 2.3.2). SCH adheres to standard ISO/IEC 19757 and is also the only rule-based schema language known to address the semantics limitations that grammar-based languages have through XPath query language (see Section 2.3.4). Therefore, to ensure the correctness of XML questionnaire instances, we employ a two step process to integrate four different levels of validation. Figure 4.2 describes how this process is carried out.

Type	Name	Operand 1	Operand 2	Result
Equality and Relational	**EQ**	String	String	Boolean
		Integer/Decimal	Integer/Decimal	Boolean
	NE	String	String	Boolean
		Integer/Decimal	Integer/Decimal	Boolean
	LT	Integer/Decimal	Integer/Decimal	Boolean
	LE	Integer/Decimal	Integer/Decimal	Boolean
	GT	Integer/Decimal	Integer/Decimal	Boolean
Conditional	**OR**	Boolean	Boolean	Boolean
	AND	Boolean	Boolean	Boolean
Arithmetical	**ADD**	Integer/Decimal	Integer/Decimal	Integer/Decimal
	SUB	Integer/Decimal	Integer/Decimal	Integer/Decimal
	MUL	Integer/Decimal	Integer/Decimal	Integer/Decimal
	DIV	Integer/Decimal	Integer/Decimal	Integer/Decimal
	MOD	Integer	Integer	Integer
List	**IS_SEL**	List	String	Boolean
	UNION	List	List	List
	INTERSECTION	List	List	List

Table 4.2: Binary Operators of CAWIML

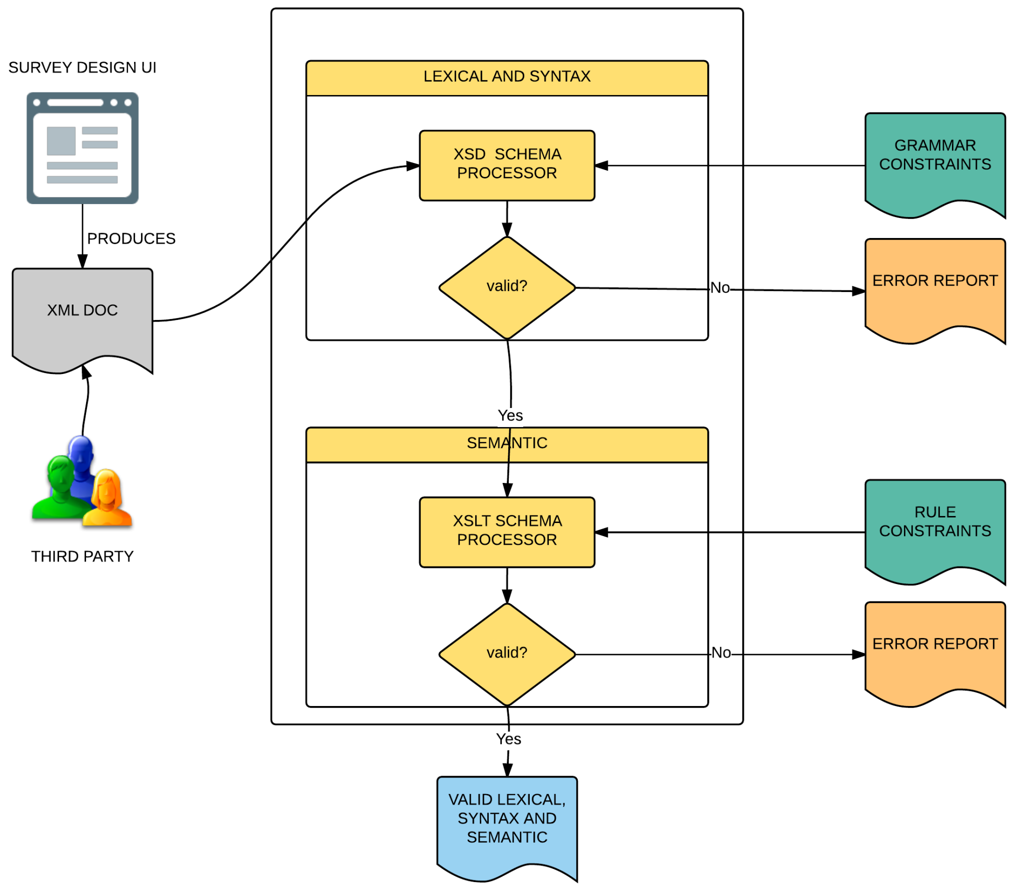


Figure 4.2: Validation process of CAWIML

An XML document, that is presented either from our interface for designing questionnaires or from any third party, is passed to the validation component. In the first instance, the XML schema processor takes our grammar constraints and an XML document to verify whether the vocabulary, structures and data-types are valid. Two types of errors can arise from this operation, *non-recoverable* errors that halt the process, i.e. the document is not well-formed (see Section 2.2) or *recoverable* which are queued without interrupting the XML processing.

In the second instance, an XSLT processor takes our SCH rules, previously converted to the valid format accepted from this processor (see Section 2.3.4), and the XML document to determine whether or not the relationships and domain specific rules are correct. The absence of any error at this stage confirms that the XML document is valid according to our structure, data-types, integrity constraints and business rules and it is consequently ready to be parsed in our CAWI system for conducting the on-line survey.

4.4 CAWIML Language Details

CAWIML addresses the content, routing and personalisation constructs of surveys by structuring XML documents into 5 categories:

1. *Survey* element defines global information relative to a questionnaire and contains information such as name, description and date.

2. *Content* specifies questions grouped through sections. It provides support for the common question types such as intro, single, multiple, open and grid questions.

3. *Field* element defines place holders variables needed to share information across different questionnaire sections. String, integer, decimal and list are the types supported currently.

4. *Routing* captures questionnaire's flow using the state-transition structure proposed in Section 4.1.

5. *Personalisation* defines constructs to adapt questionnaires for an interviewee through features such as text-fill, carry-forward, randomising and rotating.

Listings 4.1 represents the necessary elements to create a specification in CAWIML. The following sections, showcase the content, routing and personalisation categories.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<ssm>
    <survey>
        <name>...</name>
        <description>...</description>
        <date>...</date>
    </survey>
    <content>
        <section id="Section1">
            ...
        </section>
        ...
    </content>
    <field>
        ...
    </field>
    <routing>
        <statemodel ref="Section1">
            ...
        </statemodel>
        ...
        <entrypoint>
        ...
      </entrypoint>
    </routing>
    <personalisation>
      <piping ref="Section1">
        ...
      </piping>
      ...
    </personalisation>
</ssm>
```

Listing 4.1: CAWIML global structure

4.4.1 The Content Constructs

The content category of CAWIML is structured in sections. A section may contain one or more questions within and it is referenced through a state model (see Section 4.4.2). Our authoring language permits describing the most common question types for questionnaires (e.g. intro, single, multiple, open and grid). Listings 4.2 defines two sections (e.g. Outer and Inner) within the content element to group the questions specified in Figure 2.1. Essentially the use of a section serves as a container that defines questions through which state models reference and decide question's sequence.

```
<?xml version="1.0" encoding="UTF-8"?>
<ssm xmlns="https://github.com/jollopre/ssm"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="https://github.com/jollopre/ssm ../schema/ssm.xsd">
    ...
    <content>
      <section id="Outer">
        <label lang="en">Outer</label>
        ...
      </section>
      <section id="Inner">
        <label lang="en">Inner</label>
        ...
      </section>
    </content>
    ...
</ssm>
```

Listing 4.2: Content category

In the outer section, there are eight questions (e.g INF1, Q1, Q2, Q3, Q4, Q5, INF2, END). For instance, an open-ended integer question (e.g. Q5), defined in Listings

4.3, permits asking the number of cars of brand F that the interviewee has had. The open-ended integer question offers the possibility to define boundaries (e.g. min and max elements) in which the number responded must be within. It also allows setting a default value for the first time the question is shown to the interviewee (e.g. value element).

```xml
<?xml version="1.0" encoding="UTF-8"?>
<ssm xmlns="https://github.com/jollopre/ssm"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="https://github.com/jollopre/ssm ../schema/ssm.xsd">
    ...
    <content>
        <section id="Outer">
            <label lang="en">Outer</label>
            ...
            <open name="Q5">
                <label lang="en">How many cars have you had or have of F brand?</ ↩
                     label>
                <integer>
                    <min></min>
                    <max></max>
                    <value></value>
                </integer>
            </open>
            ...
        </section>
        ...
    </content>
    ...
</ssm>
```

Listing 4.3: Open-ended question

The inner section unlike the outer, only defines one question (e.g. Q6a) whose type is single. A single question (see Listings 4.4) is addressed to capture one and only one response from a set. The example provided, asks through the label element whether or not the respondent had a car from a particular brand. Note that there is a text-fill pipe reference defined within that label (see Section 4.4.3) which will be replaced with the specific brand once the questionnaire is executing. This single question also defines two closed responses with codes 01 and 02 for yes and no respectively.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<ssm xmlns="https://github.com/jollopre/ssm"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="https://github.com/jollopre/ssm ../schema/ssm.xsd">
    ...
    <content>
        ...
        <section id="Inner">
            <label lang="en">Inner</label>
            <single name="Q6a">
                <label lang="en">Have you ever had a car from <pipe ref="pipe0"/> ↩
                    brand?</label>
                <close code="01">
                    <label lang="en">Yes</label>
                </close>
                <close code="02">
                    <label lang="en">No</label>
                </close>
            </single>
        </section>
    </content>
</ssm>
```

Listing 4.4: Single question

In order to see other question types, please refer to Appendix A.2 containing the definition for the questionnaire from Figure 2.1 in CAWIML.

4.4.2 The Routing Constructs

The routing in CAWIML is composed of different state models that reference to sections defined in the content part of the language. Throughout this section, the questionnaire from Figure 2.1 will be used to discuss the features supported by CAWIML. In Listings 4.5, there are two state models (e.g. Outer and Inner) to capture the question's sequence from those sections. In addition, there is an entry point state model element that marks the beginning of the questionnaire's flow.

Every state model defines an initial state that determines what is the first state to execute through a source element (e.g. Outer defines INF1). Similarly, it requires at least one state that describes the end of a question's sequence through *sink* element. The different states supported in our state-transition solution for questionnaires are detailed in the following subsections.

```
<?xml version="1.0" encoding="UTF-8"?>
<ssm xmlns="https://github.com/jollopre/ssm"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="https://github.com/jollopre/ssm ../schema/ssm.xsd">
    ...
    <routing>
```

```
        <statemodel ref="Outer">
            <source id="INF1"/>
            ...
            <state id="c1">
                <include statemodel="Inner"/>
            </state>
            ...
        </statemodel>
        <statemodel ref="Inner">
            <source id="Q6a"/>
            ...
        </statemodel>
        <entrypoint>
            <source id="c0"/>
            <state id="c0">
                <include statemodel="Outer"/>
                <transition target="sink0"/>
            </state>
            <state id="sink0">
                <sink/>
            </state>
        </entrypoint>
    </routing>
</ssm>
```

Listing 4.5: State-transition routing

Sink

Sink state is aimed at describing the ending of a state model, i.e. it marks the end of a section. For instance, Listings 4.6 describes a sink with id 'sink0'. When this state is reached, the state model Outer finishes and consequently the questionnaire terminates. As the reader may appreciate, there are no outgoing transitions from this state type.

```
<routing>
    ...
    <statemodel ref="Outer">
        ...
        <state id="sink0">
            <sink/>
        </state>
        ...
    </statemodel>
    ...
</routing>
```

Listing 4.6: Sink state

Terminate

Terminate state unlike Sink states, are addressed to interrupt the entire routing, i.e. when this state is reached not only finishes the state model under which is defined but also terminates the questionnaire's flow even if there are more states defined in the entry point state model. Listings 4.7 describes this state type with an id 'terminate0'.

```
<routing>
    ...
    <statemodel ref="Inner">
        ...
        <state id="terminate0">
            <terminate/>
        </state>
        ...
    </statemodel>
    ...
</routing>
```

Listing 4.7: Sink state

Simple

Simple state is responsible for retrieving variables, i.e. the definition of a question and its associated responses, if any. It has capabilities to define one or more variables, i.e. every variable referenced here must be presented on the same screen of the CAWI collection stage that supports CAWIML. Listings 4.8 describes two simple states (e.g. INF1 and Q1) that contain references to variables (e.g. INF1 and Q1 for intro and single question respectively). For instance, when an interviewee decides to move forward through the questionnaire after reading INF1, the state Q1 is reached given the transition target defined within the INF1 state.

```
<routing>
    ...
    <statemodel ref="Outer">
        ...
        <state id="INF1">
            <variable ref="INF1"/>
            <transition target="Q1"/>
        </state>
        <state id="Q1">
            <variable ref="Q1"/>
            <transition target="p0"/>
        </state>
        ...
    </statemodel>
    ...
</routing>
```

Listing 4.8: Simple state

Composite

Composite state permits switching to another state model. For instance, the Inner state model referenced under the 'c1' state (see Listings 4.9) describes the sequence of questions for the Inner section of the paper questionnaire that corresponds to Q6a (see Figure 2.1). Note that this state has been defined within the state model Outer, i.e. the Inner state model will be only reached under the sequence defined for the Outer state model.

```
<routing>
    ...
    <statemodel ref="Outer">
        ...
        <state id="c1">
          <include statemodel="Inner"/>
        </state>
        ...
    </statemodel>
    ...
</routing>
```

Listing 4.9: Composite state

If-then-else

If state represents filter and skip constructs indistinctly. It is composed of a boolean expression in RPN notation and describes two transitions, then and else, for true and false result of the expression respectively. For instance, Listings 4.10 specifies the skip features attached over Q1, i.e. if response 01, 02 or 03 is selected, the 'sink0' state has to be reached, otherwise the interviewee will see question Q2 on the screen.

```
<routing>
    ...
    <statemodel ref="Outer">
        ...
        <state id="p0">
            <if>
                <condition>
                    <variable ref="Q1"/>
                    <constant type="string" value="01"/>
                    <operator name="IS_SEL"/>
                    <variable ref="Q1"/>
                    <constant type="string" value="02"/>
                    <operator name="IS_SEL"/>
                    <operator name="OR"/>
                    <variable ref="Q1"/>
                    <constant type="string" value="03"/>
                    <operator name="IS_SEL"/>
                    <operator name="OR"/>
                </condition>
                <then>
                    <transition target="sink0"/>
                </then>
                <else>
                    <transition target="Q2"/>
                </else>
            </if>
        </state>
        ...
    </statemodel>
    ...
</routing>
```

Listing 4.10: If-then-else state

Check

Check state defines a boolean expression that validates the presence of an inconsistency. There are two types supported: *warning*, that alerts the interviewee but permits her to continue the section sequence; and *error*, that stops the execution of the state model until the conflict is solved. Listings 4.11 describes a case where the questionnaire's flow is interrupted if the response was not selected. This example is useful to validate whether or not people younger than eighteen have never been married. It should described together with an if-then-else state to filter those interviewees with age under eighteen.

```
<routing>
    ...
    <statemodel ref="X">
        ...
        <state id="x1">
            <check type="error">
                <condition>
                    <variable ref="Qx"/>
                    <constant type="string" value="never"/>
                    <operator name="IS_SEL">
                </condition>
                <label lang="en">
                    People younger than 18 have never been married
                </label>
                <transition target="x2" />
            </check>
        </state>
        ...
    </statemodel>
    ...
</routing>
```

Listing 4.11: Check state

For

For state captures the loop construct of surveys and similar to the if-then-else state, has two transitions one for executing the loop body and another that is reached whenever the boolean expression is not met. There are three loop types: *range*, that iterates numbers by specifying start, end and step expressions (e.g. start at 0, end at 5 and step 1 would iterate from 0 to 4); *List* mode, that iterates all the elements of a list defined in the field section; and *expr_list* that iterates a list returned by RPN expression.

Listings 4.12 describes the instruction specified over Q6a through expr_list loop mode. This state contains: *field* element, that references a global variable (e.g. 'p4_iterator'), updated every time the iterator changes; and two transitions, one for switching to another state model (e.g. target 'c1') and the other that is reached when the loop condition is not met (e.g. target 'p5'). Note that this example includes a randomising construct (see Section 4.4.3) that alters the iterator order and ensures that at maximum four times the loop is executed.

```
<routing>
    ...
    <statemodel ref="Outer">
        ...
        <state id="p4">
            <for>
                <field ref="p4_iterator"/>
                <in>
                    <expr_list>
                        <variable ref="Q2"/>
                        <operator name="SEL"/>
                        <variable ref="Q3"/>
                        <operator name="SEL"/>
                        <operator name="UNION"/>
                    </expr_list>
                    <randomising>
                        <all present="4"/>
                    </randomising>
                </in>
                <transition target="c1"/>
            </for>
            <transition target="p5"/>
        </state>
        ...
    </statemodel>
    ...
</routing>
```

Listing 4.12: For state

Computation

Computation state is used to update place holder variables. These variables are typically used to share data across sections. For instance, Listings 4.13 permits aggregating data from different state models through the global variable HAD_CAR. This construct, implicitly defined within the paper questionnaire must be used together with an if-then-else to decide whether or not the interviewee responded *yes* to the Q6a. Note that this question may be repeated multiple times for each brand mentioned at Q2 or Q3 and therefore through HAD_CAR is captured the number of cars that the interviewee had.

```
<routing>
    ...
    <statemodel ref="Inner">
        ...
        <state id="p1">
            <computation ref="HAD_CAR">
                <assignment>
                    <variable ref="HAD_CAR"/>
                    <constant type="integer" value="1"/>
                    <operator name="ADD"/>
                </assignment>
            </computation>
            <transition target="sink0"/>
        </state>
        ...
    </statemodel>
    ...
</routing>
```

Listing 4.13: Computation state

4.4.3 The Personalisation Constructs

The personalisation constructs in CAWIML define the dynamic behaviour for surveys. These features, that may serve to adapt the survey for each respondent, are defined in this questionnaire language through elements such as pipe, randomising and rotating. The following subsections detail the personalisation constructs through features extracted From Figure 2.1.

Text-fill piping

Text-fill piping describes the behaviour of retrieving responses from previous questions as part of the text for another. For instance, in Listings 4.14 a reference to a pipe (e.g. pipe0) is described as part of the text for Q6a. This pipe, defined within a piping element points at the Inner state model and has a RPN expression describing the current loop iterator value. Note that this value may be one of the entire set of response codes from Q2 or Q3 (e.g. A, B, C, D, E, F, G, H). For instance, if the response was A for question Q2 and B,C for question Q3, the Q6a would be repeated three times by changing the pipe value to A, B or C in its question label.

```
<?xml version="1.0" encoding="UTF-8"?>
<ssm xmlns="https://github.com/jollopre/ssm"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="https://github.com/jollopre/ssm ../schema/ssm.xsd">
    ...
    <content>
        ...
        <section id="Inner">
            <label lang="en">Inner</label>
            <single name="Q6a">
                <label lang="en">Have you ever had a car from <pipe ref="pipe0"/> ↩
                    brand?</label>
                <close code="01">
                    <label lang="en">Yes</label>
                </close>
                <close code="02">
                    <label lang="en">No</label>
                </close>
            </single>
        </section>
        ...
    </content>
    ...
    <personalisation>
        ...
        <piping ref="Inner">
            ...
            <pipe id="pipe0">
                <variable ref="p4_iterator"/>
                <operator name="VALUEOF"/>
            </pipe>
            ...
        </piping>
        ...
    </personalisation>
</ssm>
```

Listing 4.14: Text-fill

Carry forward piping

Carry-forward unlike its counterpart, is intended to capture the behaviour of populating responses for a question based on a RPN expression that returns a list. That list commonly represents the responses selected/unselected from previous questions. For instance, Listings 4.15 has a multiple question (e.g. Q3) that contains a pipe reference (e.g. 'pipe0'). This pipe describes the unselected responses from Q2. The CAWI system must retrieve on real-time those unselected responses to automatically populate them as part of the responses for Q3 when the gathering of survey responses takes place. For instance, if the interviewee responded A for Q2, Q3 would be automatically populated with the responses B, C, D, E, F, G, H and Don't know, that represent those non-selected at Q2.

```
<?xml version="1.0" encoding="UTF-8"?>
```

```
<ssm xmlns="https://github.com/jollopre/ssm"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="https://github.com/jollopre/ssm ../schema/ssm.xsd">
    ...
    <content>
        ...
        <section id="Outer">
            <label lang="en">Outer</label>
            <multiple name="Q3">
                <label lang="en">Which brands are you aware of? [OTHER SPONTANEOUS ↩
                    MENTIONS Q2]</label>
                <pipe ref="pipe0"/>
            </multiple>
        </section>
        ...
    </content>
    ...
    <personalisation>
        ...
        <piping ref="Outer">
            <pipe id="pipe0">
                <variable ref="Q2"/>
                <operator name="UNSEL"/>
            </pipe>
        </piping>
        ...
    </personalisation>
</ssm>
```

Listing 4.15: Carry-forward

Randomising/Rotating

Randomising and rotating constructs are used to alter the data order presented to the respondent. In CAWIML these features are usually defined when a question is specified but they can also be utilised for reordering loops. There are two modes of specifying data order to both randomising and rotating:

- *All* that performs ordering of the entire set of responses and contains an attribute *present* to determine the number of elements to show. For instance, the for state (see section 4.4.2) describes this construct to alter the iterator order and determine the maximum number of times that this loop should be repeated.
- *Subset* which selects the elements to be randomised or rotated. Listings 4.16 defines a subset of codes (e.g. 01, 02, 03, 04, 05, 06, 07 and 08) that must

be reordered randomly. Note that response 09 is not included in that set and therefore this response must appear as last choice to select for Q2.

The importance of having two modes arises due to the fact that sometimes there can be responses in which their order should not be modified. For instance, it is frequent to offer responses such as don't know or not applicable at last in order to capture those interviewees that really do not know enough to have a formed opinion. For that purpose, the subset construct ensures that those responses out of the subset will remain unaltered.

```
<?xml version="1.0" encoding="UTF-8"?>
<ssm xmlns="https://github.com/jollopre/ssm"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="https://github.com/jollopre/ssm ../schema/ssm.xsd">
   ...
   <content>
      <section id="Outer">
         <single name="Q2">
            <label lang="en">Which brands are you aware of? [FIRST SPONTANEOUS ↩
                MENTION]</label>
            <randomising>
               <subset>
                  <code ref="01"/>
                  <code ref="02"/>
                  <code ref="03"/>
                  <code ref="04"/>
                  <code ref="05"/>
                  <code ref="06"/>
                  <code ref="07"/>
                  <code ref="08"/>
               </subset>
            </randomising>
            <close code="01">
               <label lang="en">A</label>
            </close>
            <close code="02">
               <label lang="en">B</label>
            </close>
            <close code="03">
               <label lang="en">C</label>
            </close>
            <close code="04">
               <label lang="en">D</label>
            </close>
            <close code="05">
               <label lang="en">E</label>
            </close>
```

```
                <close code="06">
                    <label lang="en">F</label>
                </close>
                <close code="07">
                    <label lang="en">G</label>
                </close>
                <close code="08">
                    <label lang="en">H</label>
                </close>
                <close code="99">
                    <label lang="en">Don't know</label>
                    <exclusive value="true"/>
                </close>
            </single>
        </section>
        ...
    </content>
    ...
</ssm>
```

Listing 4.16: Randomising

4.5 Conclusions

In order to ensure correctness of questionnaire specification through XML, we have presented in this Chapter CAWIML that combines XSD together with SCH to define structure, data-types, integrity constraints and business rules using a two step validation process. This language, described through standard and platform independent schema languages, permits reduce or eliminate the necessity of using programming languages to validate survey specifications. Furthermore, as it is non-proprietary, survey agencies may benefit of sharing questionnaire specifications when surveys have to be conducted across different CAWI systems.

CAWIML abstracts the questionnaire's routing with the state-transition paradigm. This approach integrates skip and filter constructs through an if-then-else statement and better address the conceptual gap between what survey designers want versus the code needed to produce this functionality. In particular unlike the hierarchical approach, the state-transition model avoids the necessity to reverse logics when skip patterns are present. Also, as the states perform single operations, if-then-else nested structures are non-existent thus helping to reduce coupling and improving adaptability. In addition, the unification of RPN notation to express piping and routing constructs, not only eliminates ambiguities for complex expressions but also is faster to evaluate when compared to the prefix notation mode.

Chapter 5

CAWI System

This Chapter introduces our CAWI system based on REST architectural constraints to support survey life-cycle through the specification of questionnaires using CAWIML. We have proposed a multi-layer architecture that uses XML and JSON for client server communication. In this infrastructure, the client and server implementations are separated, so changes made at any side do not impact negatively into the other which helps them to evolve independently. The client side, which adopts the novel SPA paradigm to build the HTML code directly in the browser through JavaScript, has reduced the server burden and consequently improved responsiveness.

The rest of this Chapter is structured as follows: The system architecture is presented in Section 5.1 where its different layers are explained. Thereafter Section 5.2 provides implementation details for server and client respectively.

5.1 Architecture of the System

Our proposed CAWI system uses the REST architecture [Fielding, 2000] to consider architectural principles such as scalability, simplicity or reliability. Through REST, the system is constrained to produce uniform interfaces using the HTTP verbs (e.g. GET, POST, PUT and DELETE). This restriction permits client and server to *evolve* independently. For instance, changes at any service implementation reduce or eliminate the impact produced on the client. The stateless constraint, also from REST, requires to have a server solution that avoids keeping session state for each client. For that restriction, we have implemented a token based identifier that makes it easier to *scale* our solution without needing to replicate any session data across a multi-server configuration. Reliability is improved because it eases the task of recovering from partial failures within components, connectors or data [Fielding, 2000].

The system is structured to separate the CAWI services from the user interfaces (see Figure 5.1) using JSON and XML standard exchange data formats for communication. The server side is structured into different layers:

- the *API layer* (e.g. RESTful API), that uses the HTTP methods defined by RFC 2616 [Fielding et al., 1999] creates the communication interfaces and is the only component that is directly accessible by the client;
- the *business layer*, used by the API, separates the different functionalities of the system through Java packages (e.g. content, routing, personalisation) in order to promote reusability; and
- the *data access layer*, which serves to abstract the data storage solution from the business layer, makes the system easier to maintain since this connector is centralised in one place.

This layered server structure induces *portability*, thanks to the use of Java as the general purpose programming language that works across any platform.

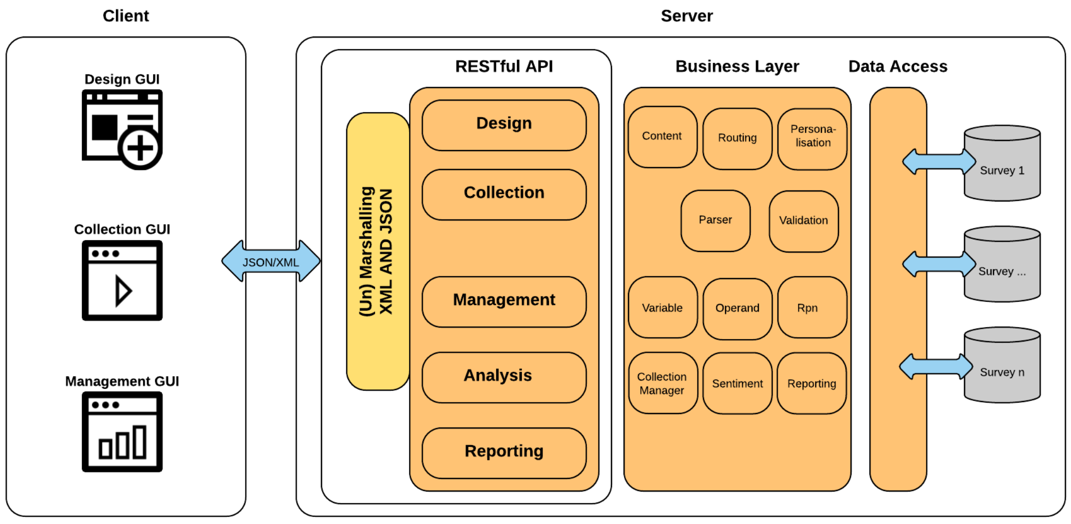


Figure 5.1: REST architecture

The client side of this architecture makes a clear separation by building and rendering every HTML code entirely on the client using the novel SPA paradigm (see Section 5.1.4). This approach when compared to the multi-page paradigm, that Blaise or SurveyMonkey adopt, is more consistent with the *simplicity* property, since it reduces the amount of data transferred from the server, improves user experience and reduces the burden on the server.

5.1.1 RESTful API

The process of conducting survey research is separated into five stages (see Section 3.2.6). In order to address these separated steps, we have implemented a REST API that permits client communication between these five services (e.g. design, collection, management, analysis and reporting). The business objects, represented as Java classes, are converted to JSON or XML every time the server responds to a request through a process named *marshalling*. Similarly, when data sent from client is received, an *unmarshalling* process transforms it to the adequate business object class instance.

The design service, seen as the only stage that uses XML exchange format to communicate the parts, deals with the definition of questionnaire instances according to CAWIML. The rest of the services, only accept JSON because it offers a direct mapping to JavaScript objects in which our client interfaces are built and also it is less verbose than XML which helps to reduce the amount of data transferred across client and server.

The collection service uses GET, POST and DELETE HTTP verbs to produce uniform interfaces for communication. The GET provides services for retrieving general information of any survey (e.g. title, description or date they were created) or to fetch incomplete interviews either because the interviewer decided to postpone or because the browser was closed. The POST verb is used for actions such as starting an interview or to store questionnaire responses when moving forward and backward through a survey. The DELETE is used throughout this service for activities like marking the ending of an interview, i.e. to remove the possibility of a valid client token to modify a questionnaire that is completed.

5.1.2 Business Layer

The business layer, organised into different modules, is built through an object-oriented paradigm and it is used by the REST API to retrieve or update state and behaviour. For instance, the content, routing and personalisation packages are built according to the grammatical rules that CAWIML defines in its XSD schema. These three packages are widely used across the survey stages and following sections provide a high-level view of the different modules that are involved in each of the client stages of design, collection and management analytics.

The design stage

The Validation and Parser packages are crucial at the questionnaire design stage as these carry out the validation, correctness and translation of XML constructs into artefacts that are ready for data gathering. Particularly, the parser package reads data from valid XML specifications to produce objects according to the content, routing and personalisation modules. It uses an event-driven approach that handles a variety of small to large questionnaire specifications appropriately. The event-driven style when compared to the popular Document Object Model (DOM) parsing, uses significantly less resources since there is no need to create a tree of objects in memory representing the XML file under process.

The collection stage

The collection stage uses the state-transition model expressed in a CAWIML instance to conduct interviews. The class diagram from Figure 5.2 represents an overview of the most important classes involved. A state model consisting of states and transitions, holds variables that represent the questions from a section and their responses. Additionally, it keeps references to pipes that together with the pseudo states, use the RPN formalism to execute logical and arithmetical expressions. The routing class, contains a list of state models and an entry point to mark the state model execution entry point. It also holds the set of place holder variables (e.g. Field class) that are shared across state models.

Each interviewee (e.g. User class) has a Manager object that registers the set of state models defined in a state-transition and controls which state model is active at any time. The state modelRunnable class, adds behaviour to a state model by including methods such as hasPrevious(), hasNext(), previousVisible() and nextVisible() that permit navigating backward and forward through an interview. It contains interesting properties like *created*, that registers the date and time in which the state model was generated, and *enabled*, that determines whether or not the state model variables should be considered when data is exported or visualised on client interfaces. The instances of this class are typically identified by a state model and state (e.g. IdStatemodelRunnable) but also may contain an index if the state model is part of a loop. For instance, the inner state model from Figure 4.1, that represents the questionnaire from Figure 2.1, can have up to eight different state modelRunnable instances identified by outer and 'c1' for the state model and the state respectively. Also an index that varies from 01 to 08 according to the set of response codes from Q2 or Q3 is set to identify each state model copy uniquely.

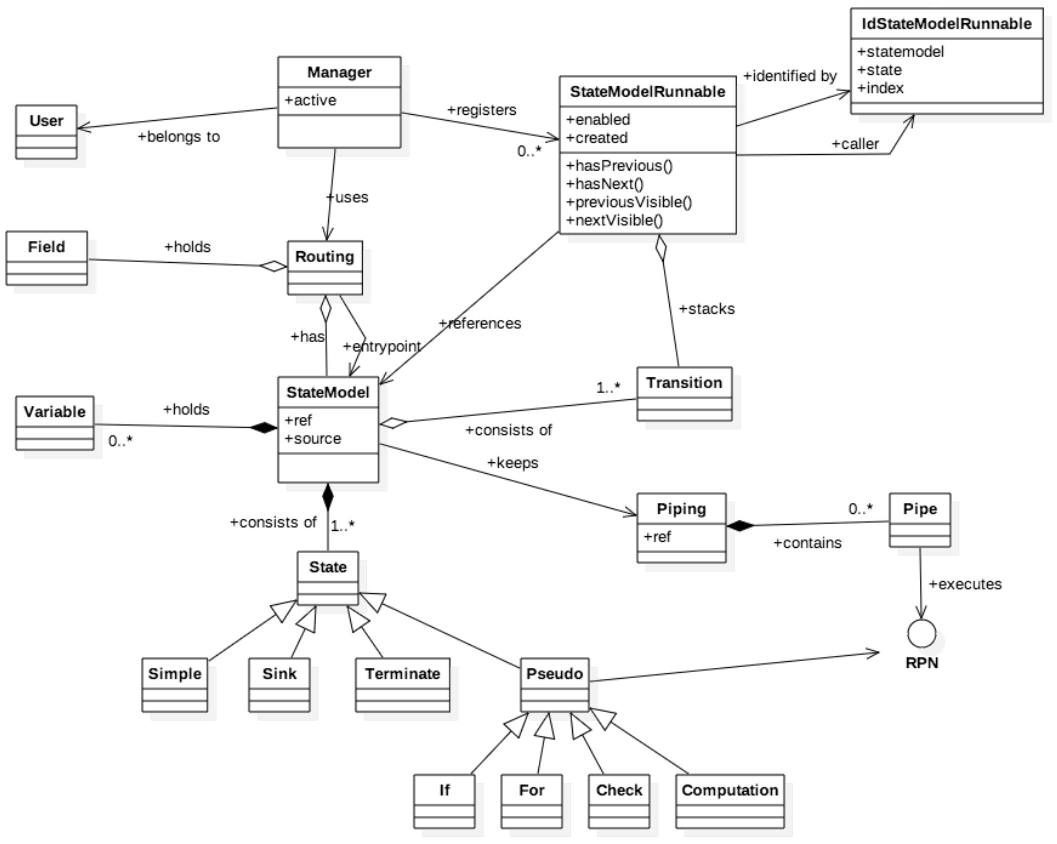


Figure 5.2: Class diagram for the Collection Manager

The management, analysis and reporting stages

The management stage, aimed at monitoring the data gathered in real time, fetches questionnaire objects by conducting multiple queries against the persistence layer. For instance, status objects are sent to the client in order to inform the percentage of completion of a questionnaire or to determine accurately the current question in an interview session.

The analysis stage incorporate aggregate queries to study the data collection for questionnaires for each type of question. For instance, an attractive feature that calculates the degree of positivity for open string questions, automatically separates sentences into one of six categories (e.g. strong positive, positive, weak positive, weak negative, negative or strong negative) proposed by Haque and Tamjid [Haque and Rahman, 2014] by using the word scores from SentiWordNet [Esuli and Sebastiani, 2006].

The reporting stage offers mechanisms to export survey data and meta data into CSV format currently. This process uses a variation of the Topological Sort algorithm proposed by Kahn [Kahn, 1962] to retrieve the order of the questions as they are defined in the routing of the CAWIML specification and presents the survey meta data first followed by the data for each interviewee on each successive line.

5.1.3 Database Solution

The database solution that we have chosen to persist and retrieve survey data and meta data is a document-based No Structured Query Language (NoSQL) approach. Through this style the database design, organised in collection of documents, offers a *direct mapping* to the business objects of our CAWI solution. Additionally, as this database style weakly references data from different collections, helps to reduce the complexity of data synchronisation when information allocated in different servers has to be combined. We have carefully considered the separation of surveys into different databases, i.e. for each questionnaire provide separate archives for the data and meta data. This separation facilitates easy isolation of problem causes when dealing with the questionnaire life-cycle.

The capacity of NoSQL solutions for horizontal scaling, consisting of connecting multiple physical or virtual machines, is not only more affordable than vertical, that is focused on empowering a server with more CPU or RAM as the relational databases systems do, but also it is significantly easier to set up inducing to a more scalable database solution. The schema-free feature permits the addition or removal of properties from data representations without the need for running migration scripts [Padhy et al., 2011] and therefore provides a more flexible persistence solution. Finally, as this approach best suits data intensive applications [Gyorodi et al., 2015], it also improves with timely handling of client requests.

5.1.4 Single Page Application

The client-side architecture follows the novel SPA paradigm. This approach proposes the creation of components to address a specific functionality of an interface. These components, based on a MVC design pattern or any of its variations, are loaded dynamically when the user interacts with the Web application. Figure 5.3, represents the different parts that form a component (e.g. model, view and controller). The DOM, which is an object representation of the HTML produced, is the place where the user sends any event such as clicking forward or backward buttons through an interview. These events are captured by controllers that perform tasks like keeping synchronised models and views. The models, used to represent the state of a component are best allocated to a single area since multiple views may initiate changes to them. Finally, the views reflect model data changes and contain the HTML code that is rendered on the browser. In order to abstract the DOM manipulation every time a view changes, we use the popular AngularJS [1] framework since it automatically handles the DOM control [Kozlowski and Bacon Darwin, 2013] and reduces the burden of testing components due to the different DOM implementations that browsers still have (e.g. Mozilla versus Internet Explorer).

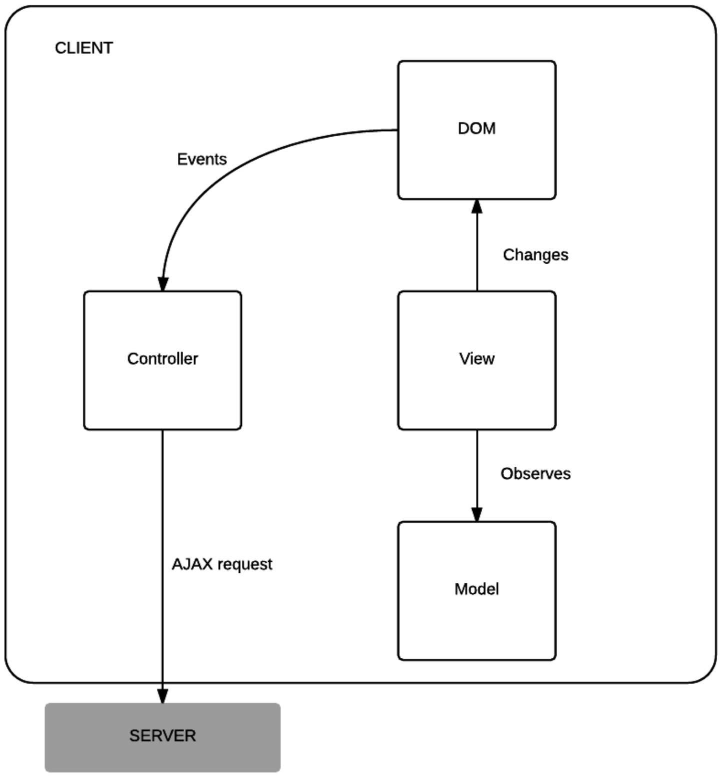


Figure 5.3: Client architecture

Through the use of SPA paradigm, our server burden has been simplified since the task of building pages has been transferred to the client which has permitted obtaining rich interactive interfaces that do not need to be reloaded at every request. Moreover, the system has gained better responsiveness because the data transferred rather than being HTML is JSON, known for its compact syntax. Therefore, this solution stands out when compared to the multi-page paradigm that Blaise or SurveyMonkey have in which the entire interface is refreshed on every request [Mesbah and van Deursen, 2007] impacting

[1] `https://angularjs.org`

not only over the system performance but also offering poorer user interactivity.

5.2 Implementation Details

Our CAWI system solution, clearly separates client and server functionalities to obtain an architecture which offers flexibility and evolvability architectural principles. Section 5.2.1 explains implementation details for the multi-layered server whereas Section 5.2.2 introduces the reader to the three interfaces that give support to survey life-cycle.

5.2.1 The Server Side

The server side of our CAWI system is separated into three layers, i.e. RESTful API, business layer and data access layer. Section 5.2.1 explains the implementation of REST principles through an example that overviews the HTTP methods utilised for the collection stage. Regarding the state-transition used for gathering survey data, we have focused on explaining the RPN algorithm that evaluates expressions for routing and personalisation constructs on real-time in Section 5.2.1.

RESTful API

The API layer used to communicate client and server has been implemented through Jersey, a standard open source framework used to develop RESTful Web Services [2] in Java. Throughout this section, a Java class that captures the requirements for the collection stage of surveys (see Listings 5.1) will be used to discuss the API implemented to communicate the parts. This class defines methods (e.g. public void postToken(...)) to implement functionalities such as resume, forward and backward through an interview. Also, it specifies annotations (e.g. @POST), that permit an easy mapping between a Java class and a web resource.

```
@Path("/collection")
public class Collection{
  /* ... */
  @Path("/token/{id}")
  @POST
  public void postToken(@PathParam("id") String id) throws IOException{
    /* Creates an instance of the survey id for an interviewee */
  }
  @Path("/token/{id}")
```

[2] https://jersey.java.net/index.html

```
  @DELETE
  public void deleteToken(@PathParam("id") String id) throws IOException{
    /*  Disables the survey id for an interviewee */
  }
  @Path("/surveyinfo/{id}")
  @GET
  @Produces(MediaType.APPLICATION_JSON)
  public Survey getSurveyInfo(@PathParam("id") String id) throws IOException{
    /* Retrieves the information from the survey id (e.g. name, description and ↩
        datetime that it was created). */
  }
  @Path("/resume/{id}")
  @GET
  @Produces(MediaType.APPLICATION_JSON)
  public RoutingStatus getResume(@PathParam("id") String id) throws IOException{
    /* Retrieves the last state reached for an interviewee. */
  }
  @Path("/previous/{id}")
  @POST
  @Consumes(MediaType.APPLICATION_JSON)
  @Produces(MediaType.APPLICATION_JSON)
  public RoutingStatus postPrevious(@PathParam("id") String id, Vector<Response> ↩
      clientResponse) throws IOException{
    /* Gets the next state of the survey for an interviewee */
  }
  @Path("/next/{id}")
  @POST
  @Consumes(MediaType.APPLICATION_JSON)
  @Produces(MediaType.APPLICATION_JSON)
  public RoutingStatus postNext(@PathParam("id") String id, Vector<Response> ↩
      clientResponse) throws IOException{
    /* Gets the previousmark the interviewee has finish the questionnaire and the ↩
        cookie have to deleted from the browser. state of the survey for an ↩
        interviewee */
  }
  /* ... */
}
```

Listing 5.1: Collection resource implementation details

@Path annotation describes a relative URI path to a resource. We have defined this annotation at two levels: at the class level, that permits creating a resource identifier (e.g. */collection*); and at method level, that specifies a sub resource for a given resource (e.g. */token*{*id*} that is reachable by clients as */collection/token*{*id*}). Every request sent to the server requires a survey id for which the interview is conducted (e.g. @PathParam). @Consumes and @Produces define the exchangeable data format used to communicate client and server (e.g. MediaType.APPLICATION_JSON) for requests and responses respectively. In addition, although it not present on the example provided, every sub resource that consumes or produces survey data and meta data requires a token header containing the interviewee's identifier. Through this header, we have achieved a stateless communication that helps for an easy scaling.

Our REST API uses GET, POST and DELETE HTTP methods. With @GET annotation it is possible to access to sub resources such as */surveyinfo*, that retrieves the meta data of a survey like its title, description and data or */resume*, that gives flexibility for the respondents to choose the right time to continue an interview.

The modification of sub resources is carried with @POST and @DELETE annotations. @POST, it is used for operations such as moving forward and backward through a questionnaire (e.g. postPrevious(...) or postNext(...)) and for creating a database record for a newer interviewee (e.g. postToken(...)). With respect to the @DELETE, that is used to mark a questionnaire as completed, helps to prevent the modification of survey data by interviewees.

The RPN Evaluation

The evaluation of a RPN expressions is conducted for every interview that takes place. Every time a variable is updated, the system automatically re-evaluates those RPN expressions in which a reference to that variable exists. The algorithm implemented to execute questionnaire expressions in our CAWI system (see Listings 5.2) only requires two arrays: *expression*, that contains an operand or operator in each position; and *stack*, to simulate push and pop operations through software [Brown, 2001] for intermediate operations that are carried out while executing an expression.

```
public Operand evaluate() throws OperandException,RpnException{
    LOCK = true;
    if(expression == null){ reset(); throw new RpnException("RPN expression is NULL" ↩
        );}
    stack.clear();
    Token first,second;
    for(int i=0;i<expression.size();i++){
      if(expression.get(i) instanceof Variable){
        Variable var = (Variable)expression.get(i);
        stack.add(var.getOperand());
      }
      else if(expression.get(i) instanceof Operand){
        stack.push(expression.get(i));
      }
      else if(expression.get(i) instanceof Operator){
```

```
15          Operator o = (Operator)expression.get(i);
16          if(o.isBinary()){ //BINARY OPERATION
17            if(stack.isEmpty()){ reset(); throw new RpnException("Empty stack trying to ↩
                   retrieve second operand for binary operation");}
18            second = stack.pop();
19            if(stack.isEmpty()){ reset(); throw new RpnException("Empty stack trying to ↩
                   retrieve first operand for binary operation");}
20            first = stack.pop();
21            result = ((Operand)first).resolve((Operand) second, o);
22            stack.add(result);
23          }
24          else{ //UNARY OPERATION
25            if(stack.isEmpty()){ reset(); throw new RpnException("Empty stack trying to ↩
                   retrieve the operand for unary operation");}
26            first = stack.pop();
27            result = ((Operand)first).resolve(o);
28            stack.add(result);
29          }
30        }
31        else{
32          reset(); throw new RpnException("Unknown token processing the RPN expression" ↩
                );
33        }
34      }
35      if(stack.isEmpty()){reset(); throw new RpnException("Not enough arguments in the ↩
             RPN expression");}
36      first = stack.pop();
37      if(!stack.isEmpty()){reset(); throw new RpnException("Malformed RPN expression") ↩
            ;}
38      LOCK=false;
39      result=(Operand)first;
40      return result;
41    }
```

Listing 5.2: The RPN evaluation

The expression array, is read from left to right (e.g. Line 6). If a variable or constant is found (e.g. Lines 7 and 11), it is pushed in the stack. In contrast, the presence of an operator performs one or more operations depending on its type. For binary operations, two operands are popped from the stack whereas for unary operations, only one operand is required. In both cases, if there are still positions to read in the expression array, the intermediate result is pushed in the stack (see Lines 22 and 28). The algorithm terminates when every position of the expression array has been read and its computational complexity cost is linear. Exceptions may be thrown at three different levels:

- At the beginning, in cases where the expression is NULL.
- During the loop execution, either because the stack is unexpectedly empty trying to retrieve an operand for unary or binary operation or because the expression contains a token element unknown for the algorithm.
- At the end, when the loop has been completed, the stack must have only one element to pop, otherwise a malformed RPN expression was passed.

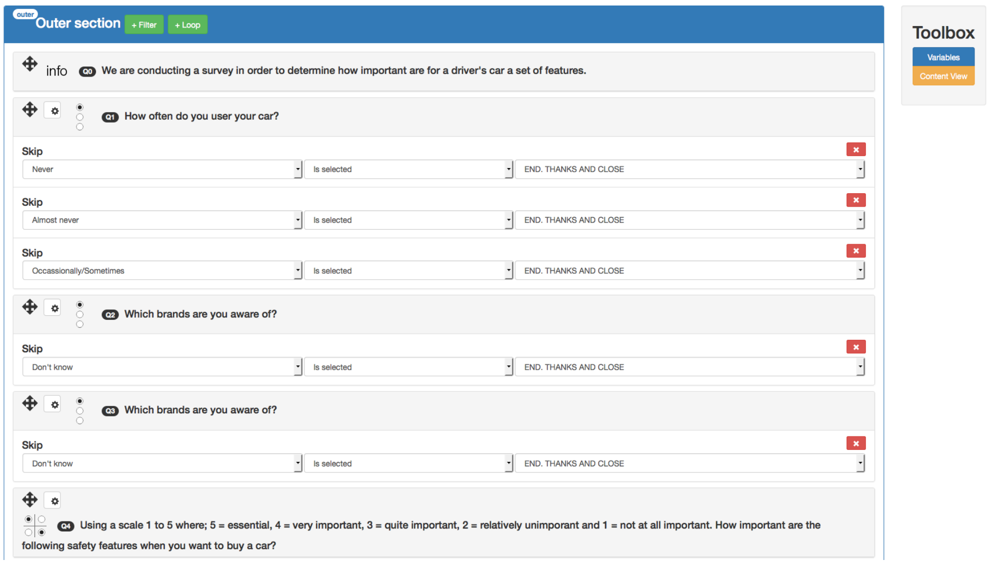
5.2.2 The Client Side

The client side of our CAWI solution organises the interface into three separated applications in order to address the survey life-cycle process. In these interfaces we have used HTML, Cascading Style Sheets (CSS) and JavaScript for structure, presentation and behaviour respectively. The following subsections detail design, collection and other interfaces that have been implemented using the SPA paradigm.

The Design Interface

The interface for designing questionnaires provides high-level visual tools to facilitate the creation of survey features without needing to write any XML code. This permits designers, that usually do not have programming skills, to create simple to complex questionnaire specifications. The interface has two modes: one for describing the content and personalisation features; and the other for specifying the questionnaire's routing.

Figure 5.4 represents the look and feel of the authoring tool for questionnaire specifications. Specifically, this screen shot depicts questions INF1, Q1, Q2, Q3 and Q4 from Figure 2.1 with a top bottom sequence order. At the top, there are two functionalities that are only applicable for a section. These are, *filter* that permits describing a logical expression and *loop* that allows specifying any of the three iteration modes (e.g. range, list and expr_list explained in the Section 4.4.2). The right side has a toolbox to access to the global variables (e.g. Variable button) as well as to switch to the content screen (e.g. content button). Next to each question there is a button (e.g. wheel) that allows selecting any routing feature. For instance, the example provided describes three skip constructs (e.g. under Q1, Q2 and Q3) where three combo boxes must be specified: the first combo, it is used to select a response from a question given (e.g. response never from Q1); the second requires to choose a binary operator (e.g. IS_SEL); and the third expects a destination question (e.g. END). In order to avoid circular dependencies, this interface provides mechanisms that prevents questionnaire designers to define skips to questions defined above this construct on the screen.


outer Outer section + Filter + Loop

info Q0 We are conducting a survey in order to determine how important are for a driver's car a set of features.

Q1 How often do you user your car?

Skip
Never | Is selected | END. THANKS AND CLOSE

Skip
Almost never | Is selected | END. THANKS AND CLOSE

Skip
Occassionally/Sometimes | Is selected | END. THANKS AND CLOSE

Q2 Which brands are you aware of?

Skip
Don't know | Is selected | END. THANKS AND CLOSE

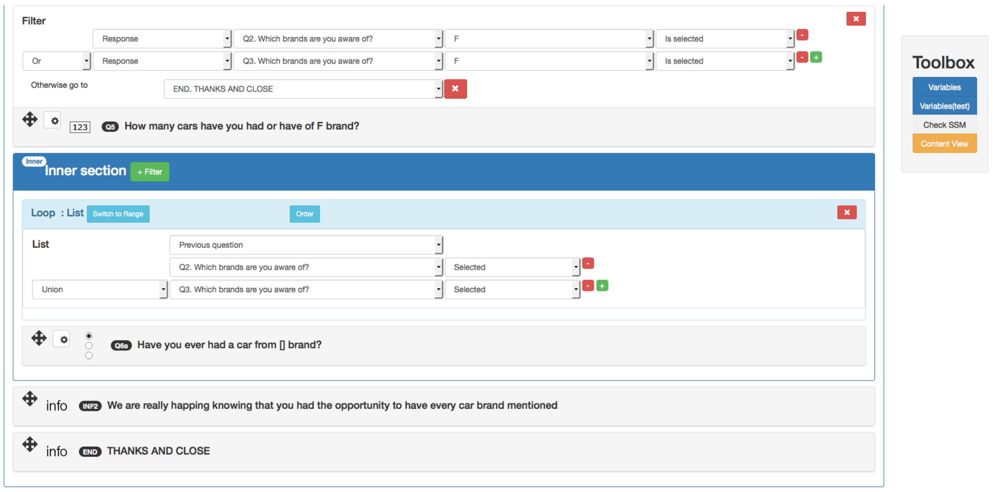
Q3 Which brands are you aware of?

Skip
Don't know | Is selected | END. THANKS AND CLOSE

Q4 Using a scale 1 to 5 where; 5 = essential, 4 = very important, 3 = quite important, 2 = relatively unimporant and 1 = not at all important. How important are the following safety features when you want to buy a car?

Figure 5.4: Routing design interface

Figure 5.5 represents the routing from Q4 till END for the questionnaire Figure 2.1. Two constructs are defined; a filter (e.g. over Q5) and an expr_list loop (e.g. above Q6a). The routing constructs, although are presented in the interface through infix notation mode, are automatically translated to postfix by using the Shunting-yard algorithm [3]. This is mainly due to the fact that CAWIML only accepts RPN expressions for logical and arithmetical operations.


Filter
Response | Q2. Which brands are you aware of? | F | Is selected
Or | Response | Q3. Which brands are you aware of? | F | Is selected
Otherwise go to | END. THANKS AND CLOSE

123 Q5 How many cars have you had or have of F brand?

Inner Inner section + Filter

Loop : List Switch to Range Order

List
Previous question
Q2. Which brands are you aware of? | Selected
Union | Q3. Which brands are you aware of? | Selected

Q6a Have you ever had a car from [] brand?

info INF2 We are really happing knowing that you had the opportunity to have every car brand mentioned

info END THANKS AND CLOSE

Figure 5.5: Content/Personalisation design interface

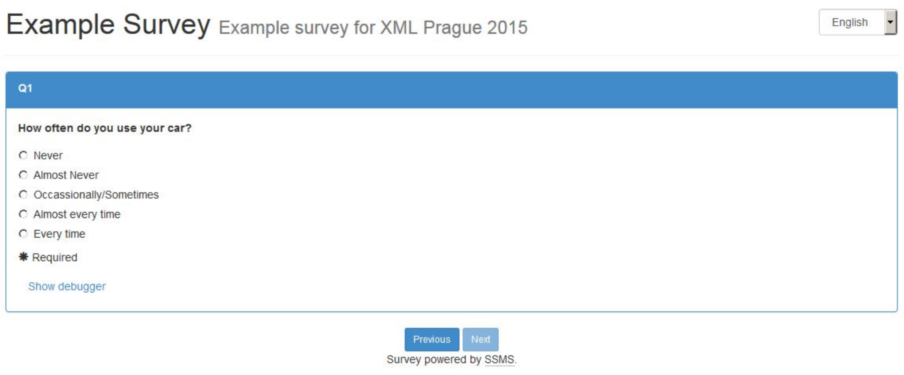
[3]http://rosettacode.org/wiki/Parsing/Shunting-yard_algorithm#JavaScript

The Collection Interface

The interface for collecting survey data separates states such as simple, check, sink or terminate on different template views. The data validation is entirely performed in the client (e.g. checking that the number of chosen responses is between a range, ensuring that every row from a grid question contains at least one response selected or verifying data-types). These validations help to reduce the amount of operations to carry out between server and client. Additionally, as this interface reacts to events immediately, the user experience is greatly improved.

The stateless restriction from our REST architecture requires for each new interview carried out in a browser to persist the interviewee's identifier either by using cookies or through the browser's data storage. Similarly, when the questionnaire is completed, it is the responsibility of the client to delete this persisted id. Every time the navigation is updated, i.e. next or previous action is requested, the server responds with the new state reached and the client redraws the part of the interface that needs to be changed without the need for page reloading.

Figure 5.6 represents an example of an interview demo that was presented at the XML Prague conference 2015. Firstly the display of the example question provides a language translation option in order to change to other languages. Next, it displays an example of a single question (e.g. Q1) with multiple responses represented through radio buttons. Finally, at the bottom, the navigation options (e.g. previous and next) are presented. Note that the next action appears disabled but it becomes enabled whenever the data entered for a question successfully passes the client validations.


Example Survey Example survey for XML Prague 2015

Q1

How often do you use your car?

- Never
- Almost Never
- Occassionally/Sometimes
- Almost every time
- Every time

✱ Required

Show debugger

Survey powered by SSMS.

Figure 5.6: Collection interface

The Management, Analysis and Reporting Interface

This interface unifies the management, analysis and reporting stages in one single interface. The most attractive functionalities from this interface is the visualisation of real-time survey responses while interviews are taking place. For that purpose, this interface provides different chart types for each type of question.

For instance, Figure 5.7 represents for each interviewee (e.g. y axis), her response to an open-ended question enclosed in one of the six categories (e.g. strong positive, positive, weak positive, weak negative, negative or strong negative) through x axis. This functionality could help to quickly analyse whether or not the responses obtained for an open question match with the survey requirements.

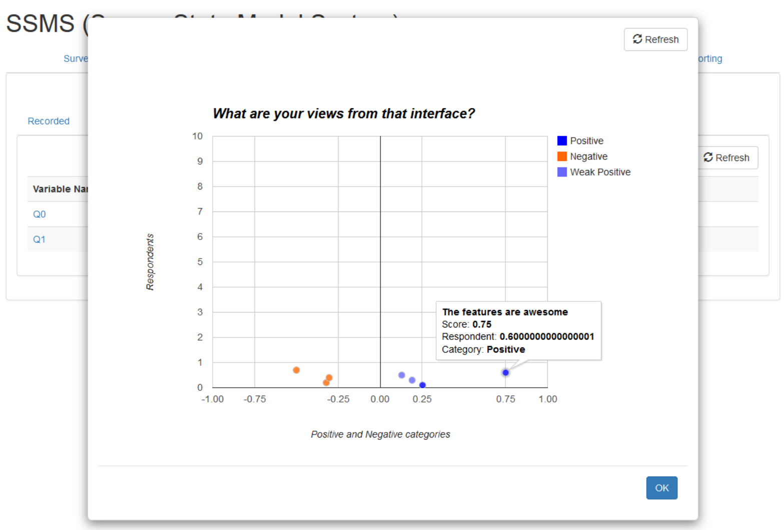


Figure 5.7: Analysis interface

5.3 Conclusions

Our CAWI system solution for survey life-cycle adopts REST constraints to adhere to scalability, simplicity and reliability architectural properties expected for CAWI systems. Particularly, the REST API, which conforms to the HTTP protocol, has been implemented with the standard reference library for Java. This multi-layered solution is highly portable and uses a NoSQL database solution which permits a more flexible persistence solution when compared to the traditional relational databases.

The client side, based on the SPA not only improves the responsiveness of a distributed system but also helps to produce richer interactive interfaces. This paradigm when compared to the multi-page approach that CAWI systems such as Blaise or SurveyMokey adopt, is more attractive. Particularly, with the choice of AngularJS as the framework for building pages, we have gained a simplified cross browser testing of functionalities, reduced the server burden by transferring interface logics to the client and promoted a parallel development of design and collection interfaces.

Chapter 6

Evaluation

In this Chapter we have conducted experiments at two levels. Firstly, we have encoded fifteen real questionnaires into CAWIML in order to determine the coverage of constructs provided. And secondly, we have performed stress testing of our state-transition for conducting interviews by simulating a scenario that is executed with increasing loads.

The rest of this Chapter is structured as follows: Section 6.1 provides the frequencies for questionnaire's features in our XML solution and analyses the results obtained. Section 6.2 introduces the methodology adopted for determining system's capacity. In this procedure parameters such as number of interviewees connected or metrics like response times have been considered in order to analyse results for interviews with simulated data.

6.1 XML Language Evaluation

In order to test the coverage of questionnaire features through CAWIML language and to determine typical frequency of constructs, we have randomly chosen a set of fifteen real questionnaires provided by Pexel. Particularly, we have studied the frequency distribution of CAWIML vocabulary over this sample of surveys. Figure 6.1 shows the total number of occurrences for each construct sorted by decreasing order of frequency. The bar chart separates the content, routing and personalisation features with purple, green and blue colours respectively. A more detailed view for this dataset, presented in Table 6.1, represents frequency of questionnaire constructs separated for each survey.

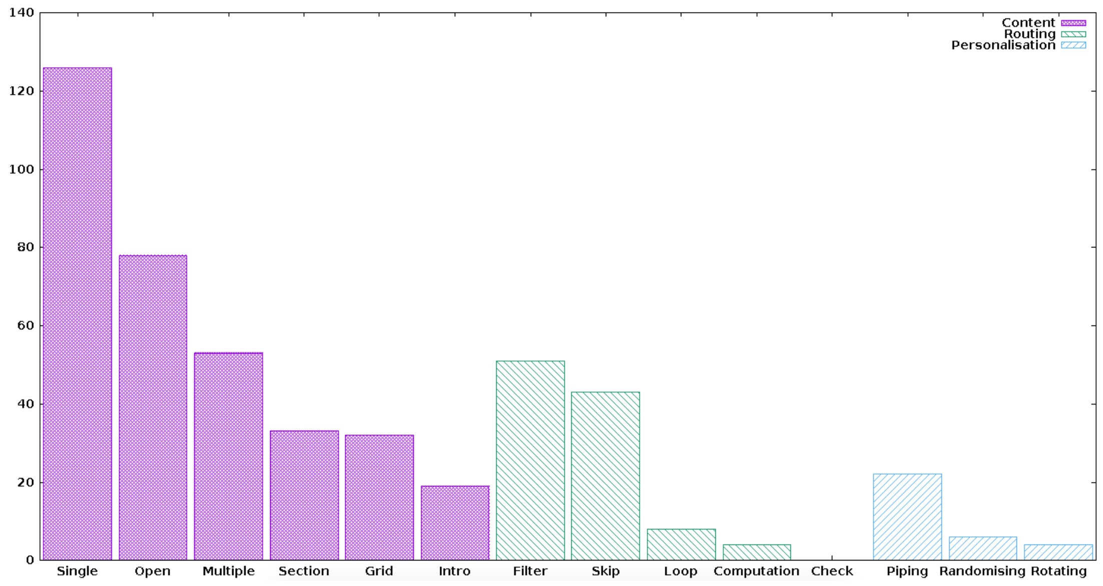


Figure 6.1: Total construct frequencies for fifteen surveys

Firstly, we can confirm that all constructs in our sample of questionnaires were encodable by CAWIML. Next, in terms of frequency of content, we can observe that single-response questions are most frequent whereas intro constructs were least frequent. Intro question whose purpose is introducing a part of a questionnaire, does not appears once at the beginning instead with each of the sections in the sampled surveys (e.g. questionnaire 04 and 10 with seven and five sections respectively, have only one intro for the entire survey).

Secondly, for the routing classification, filter construct has obtained the highest number of occurrences, however it does not appear in every questionnaire (e.g. 02, 04, 07, 12). In contrast, the skip feature, although slightly less frequent than filter, is popular amongst the surveys. Therefore, we conclude that skip logic remains an important feature of questionnaires and so should ideally be facilitated by the underlying language of surveys. Computation constructs, become essential when data from one section is needed for another section, but has only been found four times in the questionnaire 13. The absence of check constructs is not surprising since it is a rare feature that tends to appear in less well-designed questionnaires.

Thirdly, for the personalisation grouping, the three features are presented in the sample tested and piping appears to be the most common construct, specially for populating responses automatically through carry-forward mode. Regarding randomising and rotating constructs, they are less frequent.

	Feature	01	02	03	04	05	06	07	08	09	10	11	12	13	14	15	TOTAL
Content	Single	8	10	7	16	11	8	10	9	2	9	3	9	4	9	11	126
	Open-ended	11	1	5	0	11	6	5	1	6	0	12	3	6	9	2	78
	Multiple	3	0	2	1	1	2	1	6	8	3	8	2	1	7	8	53
	Section	1	1	1	7	1	3	2	1	3	5	3	2	1	1	1	33
	Grid	0	0	5	0	1	3	0	1	1	2	0	5	6	7	1	32
	Intro	1	2	1	1	2	1	1	2	1	1	3	0	1	1	1	19
Routing	Filter	1	0	2	0	9	2	0	3	9	3	6	0	5	5	6	51
	Skip logic	3	4	1	4	1	5	4	2	2	4	2	4	1	1	5	43
	Loop	0	0	0	0	0	0	0	0	2	0	6	0	0	0	0	8
	Computation	0	0	0	0	0	0	0	0	0	0	0	0	4	0	0	4
	Check	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Personalisation	Piping	0	0	3	0	0	0	0	4	9	0	0	0	1	0	5	22
	Randomising	0	0	0	0	0	0	0	2	1	0	0	0	0	0	3	6
	Rotating	0	0	0	0	0	0	0	2	0	0	0	0	0	0	2	4

Table 6.1: Frequency of questionnaire constructs separated by survey

6.2 Collection Stage Evaluation

6.2.1 Methodology

In order to evaluate the capacity of our CAWI system for collecting survey data, we have chosen the stress testing strategy with the following server configuration:

- 2.4 GHz Intel Core i5 CPU,
- 1GB RAM dedicated exclusively for the CAWI system,
- and a maximum pool size at 100 to satisfy high number of accesses to the database for update/retrieve state-transition objects.

This strategy has been executed with the methodology plan presented in Section 3.3.2 in which first a questionnaire is selected, second a scenario is designed and thirdly different parameters are varied.

Survey Choice

We have chosen questionnaire 09 (see Appendix A.1) since it has an adequate distribution of survey features. Specifically, for the content features, there are: one intro, one single-grid, two single, eight multiple and six open-ended questions. In respect to the routing constructs, there are nine filters, two skip logics and two loops that iterate over a list of responses with sizes nine and thirteen respectively. Regarding the personalisation features, nine pipes are defined either as *text-fill* or *carry forward responses* as well as a randomising construct that changes the element's order for one of the loops.

Scenario Design

The scenario designed to simulate the gathering of survey data, represented in Algorithm 2, starts with the following statements: first, a new survey instance is created; second, general information of the questionnaire such as title or description is requested; and third, the first state of the survey is called. After that sequence, an iterative process is carried out until the end of the questionnaire is reached.

The loop body firstly executes a random distributed time in order to simulate the amount of time that takes respondents to answer one or more questions and secondly, decides whether executing forward or backward requests for 70% or 30% respectively for the total of iterations. We have chosen a higher value for forward requests in order

Algorithm 2: Scenario for stress testing

```
POST token;
GET surveyinfo;
GET resume;
while $stateClass \neq Sink$ AND $stateClass \neq Terminate$ do
    Random Thinking Time from 500ms to 3000ms;
    70% of the iterations;
    if hasNext then
        Answer question(s) randomly;
        POST next;
    end
    30% of the iterations;
    if hasPrevious then
        POST previous;
    end
end
DELETE token;
```

to ensure that every simulated interview is completed. When the forward action is requested, a JavaScript functionality that randomly chooses the response for a question according to its type, is performed. Finally, when the loop is completed, the task of deleting the client token is requested, i.e. a successful response will prevent the virtual respondent to modify her survey responses after that.

Parameters Varied

Two parameters have been considered when the scenario is executed: the *virtual users*, that is the number of concurrent users that are responding to the questionnaire. This constant varies from 50 to 300 by increments of 50; and the *thinking time*, that is randomly distributed from 500 ms to 3000 ms. This parameter has served us to adequately emulate the interviewee's thinking behaviour before responding to a question.

6.2.2 Metrics

We have collected average response time, peak load and error rate for each testing level executed. The *average*, that corresponds to the statistical mean of the population, has been measured in milliseconds. Regarding the *peak load*, that captures the highest response time obtained for all the requests sent, has permitted us to identify bottlenecks in resources. In respect to the *error rate*, that is the percentage of problematic requests from every testing level executed.

In order to determine the responsiveness of the system we have compared the response times obtained for average and peak load metrics against the following three categories: *100* ms, that is the threshold in which users feel that a system reacts instantaneously to their requests; *1* second, that constitutes the limit for feeling a seamless flow; and *10* seconds, which is the limit to keep user's attention. Any time above that threshold leads to less ideal situations [Nielsen, 2010].

6.2.3 Results

The results obtained after executing stress testing at different load levels of concurrent users are presented in Table 6.2. This table captures for each level the sub resource, number of samples, average response time, minimum response time request, peak load or maximum response time request and the percentage of error. These sub resources correspond to the sub resources exposed to the client in our CAWI system (see Section 5.2.1).

The average response times for each sub resource remained under 100 milliseconds for 50, 100, 150 number of virtual users, i.e. for the interviews carried out at these levels, the respondents will feel that the system reacts instantaneously to their requests. POST token, GET survey info and DELETE token have extended this average response time threshold until 200 and GET survey info has obtained the greatest average performance until 250 simultaneous virtual interviewees. On average, none of the sub resources has exceeded 10 seconds threshold at any level in which the user's attention is difficult to handle. Not surprisingly, POST next and POST previous average response times are significantly higher than the others. In particular, POST next has obtained the highest average response time which is explained due to the fact that one or more RPN expressions can be executed when moving from one state to another in the questionnaire's flow.

In respect to the peak load, every request for GET survey info and DELETE token at number of users 50 and 100 has been enclosed under the threshold that the system reacts instantaneously. We have found bottlenecks for POST next at 250 and 300 (e.g. 8095 and 11025 respectively) and for POST previous at 300 (e.g. 11566). We consider that these values above the user's attention threshold are directly related to the number of database connections available when requesting data, i.e. having set up 100 pooling connections has lead the users to wait until a database connection is free for usage.

Regarding the percentage of errors, it has been encouraging to us that no errors were raised at any configuration level evaluated. This can be explained by the setting of an optimum ramp-up time of 10 users per second entering in the system. This value is sufficiently high to avoid saturating the server with unrealistic requests (e.g. all user at once) at the beginning of each test plan and low enough to prevent any interview to be completed before all the users are concurrently active.

No. Users	**Sub resource**	**Samples**	**Avg.**	**Min**	**Max**	**Error %**
50	POST token	50	38	19	180	0.00%
	GET surveyinfo	50	5	3	10	0.00%
	GET resume	50	32	20	62	0.00%
	POST next	764	42	12	829	0.00%
	POST previous	314	46	11	727	0.00%
	DELETE token	50	2	2	5	0.00%
100	POST token	100	30	19	132	0.00%
	GET surveyinfo	100	6	3	104	0.00%
	GET resume	100	28	19	121	0.00%
	POST next	1563	55	11	330	0.00%
	POST previous	641	52	12	298	0.00%
	DELETE token	100	3	2	18	0.00%
150	POST token	150	35	18	155	0.00%
	GET surveyinfo	150	5	3	45	0.00%
	GET resume	150	31	19	147	0.00%
	POST next	2288	89	11	1089	0.00%
	POST previous	938	93	12	1055	0.00%
	DELETE token	150	6	2	196	0.00%
200	POST token	200	54	18	396	0.00%
	GET surveyinfo	200	11	3	157	0.00%
	GET resume	200	55	19	479	0.00%
	POST next	3146	215	12	3871	0.00%
	POST previous	1288	195	11	3163	0.00%
	DELETE token	200	6	2	218	0.00%
250	POST token	250	114	18	1822	0.00%
	GET surveyinfo	250	38	3	842	0.00%
	GET resume	250	168	19	1623	0.00%
	POST next	4122	872	12	8095	0.00%
	POST previous	1689	924	11	8406	0.00%
	DELETE token	250	123	1	4037	0.00%
300	POST token	300	243	18	2898	0.00%
	GET surveyinfo	300	108	3	3566	0.00%
	GET resume	300	504	19	6133	0.00%
	POST next	4602	1737	13	11025	0.00%
	POST previous	1886	1606	13	11566	0.00%
	DELETE token	300	175	2	3929	0.00%

Table 6.2: Stress test results varying No. Users

The average response times obtained can be misleading if other aspects such as the standard deviation are not considered. For that purpose, we present in Table 6.3 the overall average response time for each configuration level, its standard deviation as well as the confidence value. The standard deviation has resulted higher than its mean at every level, i.e. it indicates that the response times are spread out over the wide range of values. This variety of data can be explained due to the fact that for every interview executed different paths can be taken which result in different durations when moving forward or backward through the questionnaire's flow.

No. Users	**Samples**	**Avg.**	**Std. Dev.**	**95% Confidence**
50	1277	40.10893417	53.30459906	2.923595761
100	2603	49.09646426	49.82751791	1.914168922
150	3825	79.93279289	98.4293415	3.119298604
200	5233	182.2417813	327.6495887	8.877329721
250	6809	774.011604	1237.580114	29.39542518
300	7687	1474.736274	2013.843128	45.01894237

Table 6.3: Stress test totals for each configuration level

In order to determine whether or not the average response times obtained are a good estimation of the statistical mean when this scenario is repeated indefinitely, we have also studied the confidence value. As we have collected a large number of independent requests to the server with a finite mean and variance, this parameter permits determining how likely are these overall averages to fluctuate. For instance, for levels 50, 100 and 150, with 95% confidence, the average response time of the state-transition collection service is 40.11±2.93, 49.10±1.92 and 79.94±3.12 respectively. Similarly, for levels 200, 250, 300, the mean should fluctuate around 182.25±8.88, 774.02±29.40 and 1474.74±45.02 respectively, however at these levels the confidence intervals are significantly higher, which can be directly related with the instability of the system due the high number of simultaneous users connected to the limited resources of the server such as database connections or memory size.

6.3 Conclusions

The implementation of the features from fifteen real paper questionnaires into CAW-IML has been successfully covered. In this experiment, we have analysed the constructs frequencies in order to determine the most popular features used at any questionnaire's category. Specifically, we have identified for content that single question is the most frequent feature utilised. Respecting the routing, the slightly differences in frequency between skip and filter constructs have led to determine that they are indistinctly used for representing conditional logic. Additionally, the checks absence at any questionnaire has confirmed its rarity across questionnaire specifications. In respect to the personalisation, we have identified that piping construct to be the most popular construct either for dynamic question labelling or for automatic population of responses.

The stress test methodology used to evaluate our state-transition at runtime has emulated adequately the interviewee's behaviour when responding to a questionnaire. Specifically, this scenario has considered the interviewee's thinking time before responding as well as it has ensured the completion for any questionnaire initiated.

In terms of the average response times obtained, we can conclude that the system reacts instantaneously for user levels such as 50, 100 and 150 and with seamless flow for the rest of the levels. Regarding the peak load, the system only has experienced three isolated cases at high number of users but on average the responsive limit thresholds have been met. In respect to the error rate, the use of adequate server resources together with time periods that ensure all the users do not access the system simultaneously, have permitted the CAWI system to obtain server responses free of errors. Finally, we have demonstrated that with 95% confidence that acceptable performance is achievable with real-life questionnaires that share similar features using the proposed architecture and CAWIML solution.

Chapter 7

Conclusion

In this thesis, we addressed the problem of validating correctness of questionnaire specifications through the use of standard XML schema languages. We compared the different XML solutions in terms of routing or personalisation constructs to conclude with the necessary and sufficient constructs to address lexical, syntactic and semantic validation levels.

Additionally, as part of the KTP programme we developed a CAWI system that uses CAWIML with consideration of different architectural properties expected for distributed systems. In this chapter, we revisit our objectives proposed and discuss desirable future extensions to our work.

7.1 Objectives Revisited

1. **Conduct a comparative analysis of the state-of-art XML language solutions that cover questionnaire definitions with a focus on the coverage of constructs and the capacity to validate correctness with standard XML schema formalisms.** In Chapter 3 we critically analysed four XML authoring languages for representing questionnaire constructs. For routing, we concluded that these solutions normally replace the use of skip constructs in favour of structured patterns. Regarding personalisation features, existing languages do not provide sufficient mechanisms to express piping features such as carry-forward. In regard to their ability to validate correctness of specifications only using XML schemas, we explored their underlying schema language and found that there is generally an inability to address semantics for questionnaire specifications.

2. **Critically appraise current modelling approaches in terms of their ability to manage questionnaire flow definitions for the purposes of routing.** In Chapter 3, we explored modelling solutions such as directed graphs, useful for analysing skip patterns, or Petri Nets, that help to analyse complex questionnaire paths. We focused on how the hierarchical modelling is being adopted by almost every authoring language solution explored. In particular, although this tree representation is able to quickly identify the circumstances under which a question is reached, the hierarchical modelling introduces a conceptual gap between language of the designers and the language of computation.

3. **Develop a new XML authoring solution to better address the correctness, together with the state-transition structures necessary for routing.** In Chapter 4 we presented CAWIML as a novel authoring language to specify questionnaire constructs using grammar and rule-based schema languages to address correctness. This language is able to describe skip and filter constructs indistinctly, introduces RPN expressions and is faster at processing expressions for routing and personalisation constructs than infix or prefix modes.

   The state-transition adopted by CAWIML for its routing, helps to close the conceptual gap between what designers specify versus the code produced since it does not require any skip logic reversing. Additionally, the single operation on its states, prevents defining nested structures and consequently improves the maintenance.

4. **Analyse the architecture of different CAWI solutions in order to determine whether the necessary and sufficient properties for a CAWI system solution are induced or not.** In Chapter 3 we reviewed Blaise and SurveyMonkey. Our analysis suggests that SurveyMonkey is better than Blaise in terms of scalability due to its stateless communication and portability because its cross platform language. In regard to the simplicity property, although it is well considered in these systems through a separation of functionalities into different components, its MVC multi-pages paradigm to build web pages neither reduces the server burden nor addresses the responsiveness effectively.

5. **Implement an architecture based on REST to better handle architectural properties such as scalability, simplicity, portability or reliability.** Our CAWI solution, presented in Chapter 5, adopts REST constraints such as stateless communication or separation of concerns to induce scalability and simplicity respectively, utilises Java to build a multi-layer cross platform solution and uses a non-relational persistence platform suitable for data intensive applications.

Moreover, the adoption of the SPA paradigm not only offers an improved user experience but also frees the server of tasks such as building web pages to induce more adequately the simplicity architectural principle.

6. **Conduct an evaluation at two levels, one for the coverage of questionnaire constructs and another to evaluate the capacity of the proposed architecture to work under different workloads.** The evaluation of CAWIML with fifteen real questionnaires suggests that the state-transition model is able to represent a wider range of questionnaire constructs as well as to address the routing task challenge.

   The stress testing methodology proposed to evaluate our architecture considers the response time in relation to usability thresholds. Moreover, the absence of errors at the different configuration levels let us conclude that the system is able to adequately use server resources.

7.2 Future Work

In this section we highlight some desirable future extensions that we would like to carry out. The first extension consist of conducting user testing for our different interfaces. At Pexel, they frequently conduct training sessions for newer employees. These sessions help them to understand the survey life-cycle as well as to get familiarised with the system. Although, we have tested our collection interface during two sessions, further experiments would permit us to better assess system responsiveness. Similarly, the design interface requires different user testing evaluation that was postponed due to difficulties to arrange time with Pexel designers as well as company willingness to focus on implementing functionalities at other survey stages (e.g. management, analysis and reporting).

The second extension consist of empowering the security of our CAWI solution. For instance, our interfaces are able to validate data entered against the XML rules specified, however we acknowledge that the API layer requires additional efforts to prevent non-valid data entered. In addition, it is also planned to encrypt sensitive data on client-server communication through Internet Protocol Security (IPsec) or Secure Socket Layer (SSL) as well as to introduce mechanisms of auditing to diagnose problems and observe intrusion signs.

The third extension consist of capturing additional data through our on-line interviewing system. This process, known as para-data, may help to improve the overall survey life-cycle and specifically the survey data quality. As such, our management stage may be enhanced with features that permit:

- identifying respondent response patterns;
- monitoring the data collection progress (e.g. number of times an interview is started and suspended or at what question a survey is suspended or abandoned);
- examining actions performed in each page (e.g. capturing whether or not the questions are answered in the order presented, number of keystrokes or mouse motions); or
- determining time required to complete a questionnaire, section of a questionnaire or individual questions.

The new management features when analysed adequately may enhance the design of questionnaires since it should be easier to identify problematic areas of the questionnaire. Therefore, CAWIML can be augmented with constructs at different levels:

- *system level*, to collect features such as software, database version or operative system used;
- *client level*, with mechanisms to identify type of device, operative system, browser utilised or screen size;
- *interview level*, to capture features such as language utilised, time zone, geographic identifiers or questionnaire completion time; and
- *question level* with mechanisms to track time entered and exit from a question, number of times a question is revisited or order chosen in multiple response questions.

The fourth extension consist of designing and implementing a question recommender aimed at improving the design stage survey life-cycle. Since our survey XML definitions can be stored in a searchable central repository, CAWIML questions can be extended to include information relative to the topic of a question, discipline in which it is applied, or what population is best suited to ask. These new XML constructs together with the question level para-data collected may benefit our CAWI system to make useful question recommendations at the questionnaire design stage.

Bibliography

[Alagar and Periyasamy, 2011] Alagar, V. S. and Periyasamy, K. (2011). *Specification of Software Systems*. Springer London, London.

[Androutsopoulos et al., 2008] Androutsopoulos, K., Binkley, D., Clark, D., and Harman, M. (2008). Slicing extended finite state machines. Technical report, King's College London. Department of Computer Science.

[Bethke, 2008] Bethke, A. D. (2008). Representing procedural logic in xml. *Journal of Software*, 3(2).

[Bethlehem, 2000] Bethlehem, J. (2000). The routing structure of questionnaires. In *ASC 1999. Leading Survey and Statistical Computing into the New Millennium*, pages 1–14.

[Bethlehem and Biffignandi, 2012] Bethlehem, J. and Biffignandi, S. (2012). *Handbook of web surveys*. John Wiley and SONS.

[Bethlehem and Hundepool, 2002] Bethlehem, J. and Hundepool, A. (2002). *TADEQ prototype version 1.5 User Manual*.

[Bethlehem and Hundepool, 2004] Bethlehem, J. and Hundepool, A. (2004). Tadeq: A tool for the documentation and analysis of electronic questionnaires. *Journal of Official Statistics*, 20(2):233–264.

[Bock, 2013] Bock, T. (2013). Market research.

[Bray et al., 2008a] Bray, T., Paoli, J., Sperberg-McQueen, C., Maler, E., and Yergeau, F. (2008a). Extensible markup language (xml) 1.0 (fifth edition). https://www.w3.org/TR/2008/REC-xml-20081126/.

[Bray et al., 2008b] Bray, T., Paoli, J., Sperberg-McQueen, C., Maler, E., and Yergeau, F. (2008b). Prolog and document type declaration. https://www.w3.org/TR/2008/REC-xml-20081126/#sec-prolog-dtd.

[Brown, 2001] Brown, B. (2001). Postfix notation mini-lecture.

[Clark, 1999] Clark, J. (1999). Xsl transformations (xslt) version 1.0. https://www.w3.org/TR/xslt.

[Clark and DeRose, 2015] Clark, J. and DeRose, S. (2015). Location path - axes. https://www.w3.org/TR/xpath/#axes. Xpath Axes.

[Corbett, 2011] Corbett, J. (2011). Incorporating metadata standards into existing working practices - how can it be done? In *Proceedings of the Fifth International Conference of the Association for Survey Computing*, pages 177–184.

[Corporation, 2012] Corporation, I. B. M. I. (2012). The hows and whys of survey research.

[Costello and Simmons, 2015] Costello, R. L. and Simmons, R. A. (2015). Two types of xml schema languages. http://xfront.com/schematron/Two-types-of-XML-Schema-Language.html.

[Costigan and Elder, 2003] Costigan, P. and Elder, S. (2003). Does the questionnaire implements the specification? who knows? In *ASC 2003. The Impact of Technology on the Survey Process*, pages 85–95.

[de Bolster, 2013] de Bolster, G. (2013). Generating blaise from ddi. In *International Blaise Users Conference*, pages 96–102.

[Dijkstra, 1968] Dijkstra, E. W. (1968). Go-to statement considered harmful. *ACM 11, 3: 147-148.*

[Dodds, 2001] Dodds, L. (2001). Schematron: validating xml using xslt. http://www.ldodds.com/papers/schematron_xsltuk.html. Retrieved May, 2016.

[Dongwon and Chu, 2000] Dongwon, L. and Chu, W. W. (2000). Comparative analysis of six xml schema languages. *AGM SIGMOD*, 29(3).

[Esuli and Sebastiani, 2006] Esuli, A. and Sebastiani, F. (2006). Sentiwordnet: A publicly available lexical resource for opinion mining. In *In Proceedings of the 5th Conference on Language Resources and Evaluation (LREC06*, pages 417–422.

[Fagan and Greenberg, 1988] Fagan, J. and Greenberg, B. (1988). Using graph theory to analyze skip patterns in questionnaires. Technical report, Statistical Research Division Bureau of the Census Washington, D.C.

[Fan and Simon, 2003] Fan, W. and Simon, J. (2003). Integrity constraints for {XML}. *Journal of Computer and System Sciences*, 66(1):254 – 291. Special Issue on {PODS} 2000.

[Fielding, 2000] Fielding, R. T. (2000). *Architectural Styles and the Design of Network-based Software Architectures*. Phd thesis, University of California.

[Fielding et al., 1999] Fielding, R. T. et al. (1999). Hypertext transfer protocol – http/1.1. https://tools.ietf.org/rfc/rfc2616.txt.

[Gao et al., 2012] Gao, S., Sperberg-McQueen, C., and Thompson, H. (2012). W3c xml schema definition language (xsd) 1.1 part 1: Structures. https://www.w3.org/TR/xmlschema11-1/#assertion-validation. Retrieved May, 2016.

[Gerrard et al., 2011] Gerrard, L., Hughes, K., Jenkins, S., Ross, E., and Wright, G. (2011). *The Survey Interchange Standard*. The Asociation for Survey Computing (ASC).

[Gerrard et al., 2015] Gerrard, L., Hughes, K., Jenkins, S., Ross, E., Wright, G., and Molloy, P. (2015). Triple-s implementations november 2015. http://www.triple-s.org/ssssoft.htm.

[Groom, 2014] Groom, C. (2014). Surveymonkey speeded up with python. Last updated Apr 13, 2014, 12:21 PM.

[Gyorodi et al., 2015] Gyorodi, C., Gyorodi, R., Pecherle, G., and Olah, A. (2015). A comparative study: Mongodb vs. mysql. In *Engineering of Modern Electric Systems (EMES), 2015 13th International Conference on*, pages 1–6.

[Haque and Rahman, 2014] Haque, A. and Rahman, T. (2014). Sentiment analysis by using fuzzy logic. *International Journal of Computer Science, Engineering and Information Technology*, 4:33–48.

[Huhges, 2007] Huhges, N. G. (2007). The difficulty of understanding social survey questionnaires from the published documentation. In *Proceedings of the Fifth International Conference of the Association for Survey Computing*, pages 213–222.

[Jabine, 1985] Jabine, T. B. (1985). Flow charts: A tool for developing and understanding survey questionnaires. *Journal of Official Statistics*, 1(2):189–207.

[Jellife, 2007] Jellife, R. (2007). Converting xml schemas to schematron. http://archive.oreilly.com/pub/post/converting_xml_schemas_to_sche.html. Retrieved May, 2016.

[Kahn, 1962] Kahn, A. B. (1962). Topological sorting of large networks. *Commun. ACM*, 5(11):558–562.

[Katz et al., 1997] Katz, Irvin an Mason, G., Stinson, L., and Conrad, F. (1997). Questionnaire designers versus instrument authors: Bottlenecks in the development of computer-administered questionnaires. In *Proceedings of the Survey Research Methods Section*, pages 1029–1034.

[Kozlowski and Bacon Darwin, 2013] Kozlowski, P. and Bacon Darwin, P. (2013). *Mastering Web Aplication Development with AngularJS*. Packt Publishing Ltd.

[MacKenzie et al.,] MacKenzie, C. M., Laskey, K., McCabe, F., Brown, P. F., and Metz, R. Reference model for service oriented architecture 1.0. Technical report committee specification 1, OASIS.

[Madsen, 2009] Madsen, L. B. (2009). Definition of validation rules for xml data for electronic questionnaires. Thesis for master of information technology in software development, IT University of Copenhagen.

[Mesbah and van Deursen, 2007] Mesbah, A. and van Deursen, A. (2007). Migrating multi-page web applications to single-page ajax interfaces. In *Software Maintenance and Reengineering, 2007. CSMR '07. 11th European Conference on*, pages 181–190.

[Murata et al., 2005] Murata, M., Lee, D., Mani, M., and Kawaguchi, K. (2005). Taxonomy of xml schema languages using formal language theory. *ACM Trans. Internet Technol.*, 5(4):660–704.

[Mller and Schwartzbach, 2006] Mller, A. and Schwartzbach, M. I. (2006). *An Introduction to XML and Web Technologies*. Addison-Wesley.

[Netherlands, 2015] Netherlands, S. (2015). Blaise, survey software for professionals.

[Nielsen, 2010] Nielsen, J. (2010). Website response times. https://www.nngroup.com/articles/website-response-times/.

[Nurseitov et al., 2009] Nurseitov, N., Paulson, M., Reynolds, R., and Izurieta, C. (2009). Comparison of json and xml data interchange formats: A case study. *Scenario*, 59715:157–162.

[of California, 2015] of California, U. (2015). Computer-assisted survey execution system.

[Overdick, 2007] Overdick, H. (2007). The resource-oriented architecture. In *Services, 2007 IEEE Congress on*, pages 340–347.

[Padhy et al., 2011] Padhy, R. P., Patra, M. R., and Satapathy, S. C. (2011). Rdbms to nosql: Reviewing some next-generation non-relational database's. *International Journal of Advanced Engineering sciences and technologies*, 11:15–30.

[Rasmussen and Blank, 2007] Rasmussen, K. and Blank, G. (2007). The data documentation initiative: a preservation standard for research. *Archival Science*, 7:55–71.

[Rolke, 2010] Rolke, H. (2010). Automata and preti net models for visualizing and analyzing complex questionnaires - a case study. In *Proceedings of the Workshops of the 31st International Conference on Application and Theory of Petri Nets and Other Models of Concurrency (PETRI NETS 2010)*, pages 317–329.

[Segel et al., 2015] Segel, P., Subramanian, M., Frey, R., and Snowden, R. (2015). Blaise 5 server configuration for web surveys. In *International Blaise Users Conference*, pages 161–168.

[Segel et al., 2013] Segel, P., Subramanian, M., Snowden, R., Frey, R., and Mike, F. (2013). Implementing blaise 5 in a production environment. In *International Blaise Users Conference*, pages 243–258.

[Spencer, 2012] Spencer, S. (2012). A case against the skip statement. http://bit.ly/CaseAgainstSkip.

[Sperberg-McQueen and Thompson, 2012] Sperberg-McQueen, C. and Thompson, H. (2012). W3c xml schema. https://www.w3.org/XML/Schema. Retrieved May, 2016.

[Sthrenberg, 2013] Sthrenberg, M. (2013). Quo vadis xml? In *XML Prague. A conference on XML*, pages 141–162.

[Sthrenberg and Christian, 2010] Sthrenberg, M. and Christian, W. (2010). Refining the taxonomy of xml schema languages. a new approach for categorizing xml schema languages in terms of processing complexity. In *Proceedings of Balisage: The Markup Conference 2010*.

[Thomas et al., 2009] Thomas, W., Arofan, G., and Gager, J. (2009). *User Guide for Data Documentation Initiative Version 3.1*.

[Van der Vlist, 2003] Van der Vlist, E. (2003). *RELAX NG*. O'Reilly Media.

[Van der Vlist, 2006] Van der Vlist, E. (2006). Xml schema languages compared. In *XML Prague. A conference on XML*, pages 9–28.

[Varde et al., 2010] Varde, A., Rundensteiner, E., and Fahrenholz, S. (2010). *Web-based Support Systems*, chapter XML Based Markup Languages for Specific Domains, pages 215–238. Springer London, London.

[Volguine, 2013] Volguine, O. (2013). Blaise 4.8.4 web form load and performance testing. In *International Blaise Users Conference*, pages 64–95.

[Warner, 1965] Warner, S. L. (1965). Randomized response: A survey technique for eliminating evasive answer bias. *Journal of the American Statxistical Association*, 60(309).

[Wright, 2007] Wright, G. (2007). Triple-s: The broader horizon. In *ASC 2007. The Challenges of a Changing World: Developments in the Survey Process*, pages 241–246.

Appendix A

Appendix

A.1 Real Questionnaires Sample

The following list represents the fifteen real questionnaires provided by Pexel Research Services that have been implemented in CAWIML:

1. https://github.com/jollopre/ssm/blob/master/real_surveys/01.xml
2. https://github.com/jollopre/ssm/blob/master/real_surveys/02.xml
3. https://github.com/jollopre/ssm/blob/master/real_surveys/03.xml
4. https://github.com/jollopre/ssm/blob/master/real_surveys/04.xml
5. https://github.com/jollopre/ssm/blob/master/real_surveys/05.xml
6. https://github.com/jollopre/ssm/blob/master/real_surveys/06.xml
7. https://github.com/jollopre/ssm/blob/master/real_surveys/07.xml
8. https://github.com/jollopre/ssm/blob/master/real_surveys/08.xml
9. https://github.com/jollopre/ssm/blob/master/real_surveys/09.xml
10. https://github.com/jollopre/ssm/blob/master/real_surveys/10.xml
11. https://github.com/jollopre/ssm/blob/master/real_surveys/11.xml
12. https://github.com/jollopre/ssm/blob/master/real_surveys/12.xml
13. https://github.com/jollopre/ssm/blob/master/real_surveys/13.xml
14. https://github.com/jollopre/ssm/blob/master/real_surveys/14.xml

15. https://github.com/jollopre/ssm/blob/master/real_surveys/15.xml

A.2 CAWIML Instance

Listings A.1 captures the paper questionnaire provided in Figure 2.1. This instance has been defined using CAWIML language.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<ssm xmlns="https://github.com/jollopre/ssm"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="https://github.com/jollopre/ssm ../schema/ssm.xsd">
   <survey>
      <name>Paper questionnaire instance</name>
      <description>Example survey for MRes thesis</description>
      <date>2016-03-01</date>
   </survey>
   <content>
      <section id="Outer">
         <label lang="en">Outer</label>
         <intro name="INF1">
            <label lang="en">We are conducting a survey in order to determine how ↩
                 important are for a drivers car
               a set of features.</label>
         </intro>
         <single name="Q1">
            <label lang="en">How often do you use your car?</label>
            <close code="01">
               <label lang="en">Never</label>
            </close>
            <close code="02">
               <label lang="en">Almost Never</label>
            </close>
            <close code="03">
               <label lang="en">Occassionally/Sometimes</label>
            </close>
            <close code="04">
               <label lang="en">Almost every time</label>
            </close>
            <close code="05">
               <label lang="en">Every time</label>
            </close>
         </single>
         <single name="Q2">
            <label lang="en">Which brands are you aware of? [FIRST SPONTANEOUS  ↩
                MENTION]</label>
            <close code="01">
```

```
                <label lang="en">A</label>
            </close>
            <close code="02">
                <label lang="en">B</label>
            </close>
            <close code="03">
                <label lang="en">C</label>
            </close>
            <close code="04">
                <label lang="en">D</label>
            </close>
            <close code="05">
                <label lang="en">E</label>
            </close>
            <close code="06">
                <label lang="en">F</label>
            </close>
            <close code="07">
                <label lang="en">G</label>
            </close>
            <close code="08">
                <label lang="en">H</label>
            </close>
            <close code="99">
                <label lang="en">Don't know</label>
                <exclusive value="true"/>
            </close>
        </single>
        <multiple name="Q3" refused="true">
            <label lang="en">Which brands are you aware of? [OTHER SPONTANEOUS MENTIONS Q2]</label>
            <pipe ref="pipe0"/>
        </multiple>
        <grid name="Q4">
            <label lang="en">Using a scale 1 to 5 where; 5=essential, 4=very important, 3=quite important,
            2=relatively unimportant, 1=not at all important. How important are the following safety features
            when you want to buy a car?</label>
            <single>
                <rows>
                    <close code="01">
                        <label lang="en">Cruise Control</label>
                    </close>
                    <close code="02">
                        <label lang="en">Seat Heater</label>
                    </close>
                    <close code="03">
                        <label lang="en">Automatic transmision</label>
                    </close>
                    <close code="04">
                        <label lang="en">Sunroof</label>
                    </close>
                    <close code="05">
                        <label lang="en">Navigation system</label>
                    </close>
                    <close code="06">
                        <label lang="en">Knee airbags</label>
                    </close>
                </rows>
                <columns>
                    <close code="05">
                        <label lang="en">Essential</label>
                    </close>
                    <close code="04">
                        <label lang="en">Very important</label>
                    </close>
                    <close code="03">
                        <label lang="en">Quite important</label>
                    </close>
                    <close code="02">
                        <label lang="en">Relatively unimportant</label>
                    </close>
                    <close code="01">
                        <label lang="en">Not at all important</label>
                    </close>
                </columns>
            </single>
        </grid>
        <open name="Q5">
            <label lang="en">How many cars have you had or have of F brand?</ ↩
                label>
            <integer>
                <min></min>
                <max></max>
                <value></value>
            </integer>
        </open>
        <intro name="INF2">
            <label lang="en">We are really happy knowing that you had the  ↩
                opportunity to have every car brand mentioned</label>
        </intro>
        <intro name="END">
            <label lang="en">THANKS AND CLOSE</label>
        </intro>
    </section>
    <section id="Inner">
        <label lang="en">Inner</label>
        <single name="Q6a">
            <label lang="en">Have you ever had a car from <pipe ref="pipe0"/>  ↩
                brand?</label>
            <close code="01">
                <label lang="en">Yes</label>
            </close>
            <close code="02">
                <label lang="en">No</label>
            </close>
        </single>
    </section>
</content>
<field>
    <integer id="HAD_CAR" value="0"/>
    <iterator id="p4_iterator"/>
</field>
<routing>
    <statemodel ref="Outer">
        <source id="INF1"/>
        <state id="INF1">
            <variable ref="INF1"/>
            <transition target="Q1"/>
        </state>
        <state id="Q1">
            <variable ref="Q1"/>
            <transition target="p0"/>
        </state>
        <state id="p0">
            <if>
                <condition>
                    <variable ref="Q1"/>
                    <constant type="string" value="01"/>
                    <operator name="IS_SEL"/>
                    <variable ref="Q1"/>
                    <constant type="string" value="02"/>
                    <operator name="IS_SEL"/>
                    <operator name="OR"/>
                    <variable ref="Q1"/>
                    <constant type="string" value="03"/>
                    <operator name="IS_SEL"/>
                    <operator name="OR"/>
                </condition>
                <then>
```

```
                        <transition target="sink0"/>
                    </then>
                    <else>
                        <transition target="Q2"/>
                    </else>
                </if>
            </state>
            <state id="sink0">
                <sink/>
            </state>
            <state id="Q2">
                <variable ref="Q2"/>
                <transition target="p1"/>
            </state>
            <state id="p1">
                <if>
                    <condition>
                        <variable ref="Q2"/>
                        <constant type="string" value="99"/>
                        <operator name="IS_SEL"/>
                    </condition>
                    <then>
                        <transition target="sink0"/>
                    </then>
                    <else>
                        <transition target="Q3"/>
                    </else>
                </if>
            </state>
            <state id="Q3">
                <variable ref="Q3"/>
                <transition target="p2"/>
            </state>
            <state id="p2">
                <if>
                    <condition>
                        <variable ref="Q3"/>
                        <constant type="string" value="99"/>
                        <operator name="IS_SEL"/>
                    </condition>
                    <then>
                        <transition target="sink0"/>
                    </then>
                    <else>
                        <transition target="Q4"/>
                    </else>
                </if>
            </state>
            <state id="Q4">
                <variable ref="Q4"/>
                <transition target="p3"/>
            </state>
            <state id="p3">
                <if>
                    <condition>
                        <variable ref="Q2"/>
                        <constant type="string" value="06"/>
                        <operator name="IS_SEL"/>
                        <variable ref="Q3"/>
                        <constant type="string" value="06"/>
                        <operator name="IS_SEL"/>
                        <operator name="OR"/>
                    </condition>
                    <then>
                        <transition target="Q5"/>
                    </then>
                    <else>
                        <transition target="sink0"/>
                    </else>
                </if>
            </state>
            <state id="Q5">
                <variable ref="Q5"/>
                <transition target="p4"/>
            </state>
            <state id="p4">
                <for>
                    <field ref="p4_iterator"/>
                    <in>
                        <expr_list>
                            <variable ref="Q2"/>
                            <operator name="SEL"/>
                            <variable ref="Q3"/>
                            <operator name="SEL"/>
                            <operator name="UNION"/>
                        </expr_list>
                        <randomising>
                            <all present="4"/>
                        </randomising>
                    </in>
                    <transition target="c1"/>
                </for>
                <transition target="p5"/>
            </state>
            <state id="c1">
                <include statemodel="Inner"/>
            </state>
            <state id="p5">
                <if>
                    <condition>
                        <variable ref="Q2"/>
                        <operator name="SEL"/>
                        <variable ref="Q3"/>
                        <operator name="SEL"/>
                        <operator name="UNION"/>
                        <operator name="SIZE"/>
                        <variable ref="HAD_CAR"/>
                        <operator name="EQ"/>
                        <variable ref="HAD_CAR"/>
                        <constant type="integer" value="4"/>
                        <operator name="EQ"/>
                        <operator name="OR"/>
                    </condition>
                    <then>
                        <transition target="INF2"/>
                    </then>
                    <else>
                        <transition target="END"/>
                    </else>
                </if>
            </state>
            <state id="INF2">
                <variable ref="INF2"/>
                <transition target="END"/>
            </state>
            <state id="END">
                <variable ref="END"/>
                <transition target="sink0"/>
            </state>
        </statemodel>
        <statemodel ref="Inner">
            <source id="Q6a"/>
            <state id="Q6a">
                <variable ref="Q6a"/>
                <transition target="p0"/>
            </state>
            <state id="p0">
                <if>
                    <condition>
                        <variable ref="Q6a"/>
                        <constant type="string" value="01"/>
                        <operator name="IS_SEL"/>
                    </condition>
                    <then>
                        <transition target="p1"/>
                    </then>
                    <else>
                        <transition target="sink0"/>
                    </else>
                </if>
            </state>
            <state id="p1">
                <computation ref="HAD_CAR">
                    <assignment>
                        <variable ref="HAD_CAR"/>
                        <constant type="integer" value="1"/>
                        <operator name="ADD"/>
                    </assignment>
                </computation>
                <transition target="sink0"/>
            </state>
            <state id="sink0">
                <sink/>
            </state>
        </statemodel>
        <entrypoint>
            <source id="c0"/>
            <state id="c0">
                <include statemodel="Outer"/>
                <transition target="sink0"/>
            </state>
            <state id="sink0">
                <sink/>
            </state>
        </entrypoint>
    </routing>
    <personalisation>
        <piping ref="Outer">
            <pipe id="pipe0">
                <variable ref="Q2"/>
                <operator name="UNSEL"/>
            </pipe>
        </piping>
        <piping ref="Inner">
            <pipe id="pipe0">
                <variable ref="p4_iterator"/>
                <operator name="VALUEOF"/>
            </pipe>
        </piping>
    </personalisation>
</ssm>
```

Listing A.1: CAWIML instance