INVESTIGACIÓN & DESARROLLO, No. **13**, Vol. **1**: 68 – 84 (2013)
ISSN 2518-4431

SEA: A PLATFORM FOR DESIGNING, CAPTURING AND PROCESSING LARGE-SCALE SURVEYS

SEA: UNA PLATAFORMA PARA DISEÑAR, RECOLECTAR Y PROCESAR ENCUESTAS A GRAN ESCALA

Alex Villazón*, Vladimir Calderón and Ivan Krsul****
**Centro de Investigaciones de Nuevas Tecnologías Informáticas – CINTI*
Universidad Privada Boliviana, Bolivia
***Artexacta*
avillazon@upb.edu
(Recibido el 10 julio 2013, aceptado para publicación el 20 de agosto 2013)

ABSTRACT

Collecting data through surveys is widely used by governments, international and non-governmental organizations to gather up-to-date and useful statistical information. Unfortunately, this process is tedious and often performed using paper support or customized software not allowing on-the-fly validations to avoid transcription errors at the early stage of the data collection. In this article, we describe SEA, a new web-based platform allowing sophisticated and large-scale surveys to be designed, captured and processed. SEA allows the survey designers to express complex input constraints through formulas for data validation. We describe the constraint language and the internals of the SEA engine, which is implemented using compilation, code generation and interpretation techniques.

RESUMEN

Una forma ampliamente utilizada para obtener datos estadísticos actualizados y relevantes es por medio del despliegue de encuestas (surveys), cuyos resultados pueden servir a organismos gubernamentales, internacionales y no-gubernamentales. Lamentablemente, dicho proceso es tedioso y suele ser realizado utilizando formularios de papel, o programas de software a medida, que no permiten la validación de los datos de manera inmediata (on-the-fly) durante las etapas más tempranas de la recopilación de los datos. En este artículo presentamos SEA, una nueva plataforma basada en tecnología web, que permite el diseño, la recolección y el procesamiento de encuestas sofisticadas a gran escala. SEA permite a los diseñadores de encuestas la posibilidad de expresar restricciones en los datos de entrada (input) por medio de fórmulas de validación. Presentamos el lenguaje de restricciones (constraint language) que se ha desarrollado para SEA, y describimos el funcionamiento interno de la plataforma cuya implementación está basada en técnicas de compilación, generación de código e interpretación.

Keywords: Code Generation, Compilation Techniques, Constraint Validation
Palabras Clave: Generación de Código, Técnicas de Compilación, Validación de Restricciones

1. INTRODUCTION

Traditionally, collecting survey information is done using hard copies (paper) of questionnaires, and doing data transcription to an electronic support after data collection. Unfortunately, such an approach is tedious and error-prone. Errors can be introduced in the questionnaires by the enumerators (i.e. people in charge of filling questionnaires), thus requiring validation steps by supervisors. Also, the transcription of the collected data into a computer system can be a factor of error. Even though there are some technical means to avoid manual transcription (e.g. using reading devices), in some cases where the collected data has numbers or text fields, it is not possible to perform automatize the collection of data into the system.

A common solution is to use a simple electronic support to reduce the potential data capture errors. For example, it is not rare to see large surveys using Microsoft Office tools (Excel/Access with some in-house VisualBasic solution). Unfortunately, this requires IT support, in order to develop the surveys, as there is no standard survey designer tool and gathering data is not simple (often it is necessary to develop data import/export tools to upload captured data into a central database). On-line survey tools have been developed recently, enabling data collection [1], [2], [3] and [4]. However, these tools target small surveys, with a rather limited set of question types and validation mechanisms.

In this article, we present SEA (Survey Engine & Analysis) a platform and set of tools enabling large-scale surveys. SEA provides a survey builder module, which allows defining complex surveys including those requiring definition of constraints on the values, thus reducing the possible errors during capturing. SEA also provides a simple yet powerful *constraint language*, allowing non-computer scientists to describe constraints based on a rich set of question types. A full survey management module allows defining different users with roles (enumerators, validators and managers), so as

DOI: 10.23881/idupbo.013.1-6i

to follow at any moment the evolution of a survey. SEA is implemented in the .Net Framework [5] using code generation techniques to dynamically generate *JavaScript* code in the *C#* programming language.
SEA supports several basic and advanced types of questions in a survey with basic and advance validations. Basic validation refers to verifying if a mandatory answer is set or not, and if the type of answered question is of the expected type. Advanced validation, refers to the user defining complex *constraints* (i.e. boolean expressions), which restricts the possible valid values of answers. Constraints are defined using *user-defined formulas or expressions* based on question identifiers combined with operators and functions that are intuitive and easy to use. Also, the survey designer allows the definition of *variables* and *computed values* with complex formulas, which are calculated on-the-fly, allowing the user to validate values during the capture, and therefore prevent wrong data to be collected. SEA allows also the definition of *Conditional Display* that are associated to groups of questions to be shown or hidden according to answer values during the capture. This allows hiding questions, which are not to be filled during data capture, and also prevents wrong data to be captured.

Because each survey has a different set of questions with different types, it is not possible to define a single 'language' to validate formulas or expressions entered by the users. We applied techniques of *dynamic code generation* (a) to generate a grammar and compilers for each survey in order to validate on-the-fly the expressions introduced by users during survey design (on server side), and (b) to generate validation code (JavaScript) to dynamically validate the answers during survey capture (on client side). Because formulas create complex dependencies between questions, the system performs a dependency analysis to correctly validate data, compute expressions and evaluate conditional displaying.

This article is structured as follows. In Section 2 we describe the architecture of the SEA platform. Section 3 presents the survey definition and depicts the approach to code generation and presents the SEA Library for validation. Section 4 presents the constraint language definition and how variables and computed values are evaluated. Section 5 describes how Conditional Display is handled. Section 6 presents the dependency analysis for on-the-fly validation. Finally, Section 8 concludes the article.

2. SEA ARCHITECTURE

The architecture of the SEA is depicted in Figure 1. It consists of the client side and the server side. The server has several components that enable Survey Design and the Questionnaires Capture. Most of the components are accessible through Web Services to communicate with the client or can communicate internally. All the survey definitions and the collected data are stored into a database.

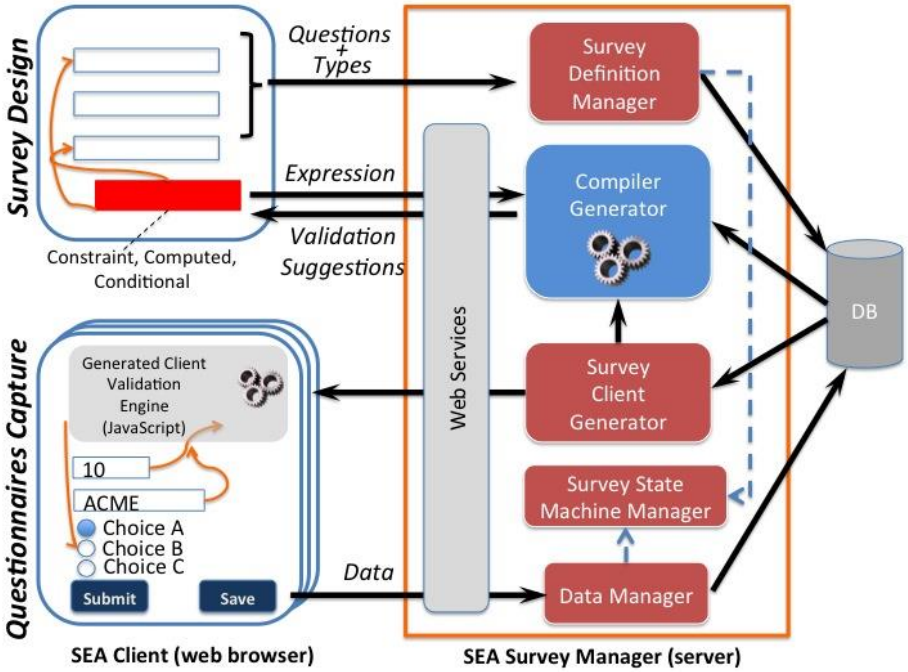

Figure 1 - Architecture of the SEA Platform.

2.1. Survey Definition

The components responsible for the survey definition are shown on the top part of Figure 1. During survey design, the user adds, modify and remove questions, which are sent to the Survey Definition Manager module on the server. This module translates the defined questions with their type to a flat representation in the SEA database (DB). When a constraint, computed value, variable or conditional display expression is added to the survey, the expression is sent to the Compiler Generator module, which reads the full definition of the complete survey and *dynamically generates the grammar* (i.e. the language definition) and the compiler for that specific grammar. The module validates the user-defined expression and sends back a validation code or, in case the expression is invalid, all the information about where the expression may be wrong, and a list of possible valid question identifiers or operators. Thus, the user can interactively and intuitively build valid expressions without having to learn complex syntax to express formulas. The system also performs a full validation (consistency check) of the survey to verify if the question identifiers specified in the expressions are valid. This is important notably when a question is removed, which may invalidate several expressions associated to it.

2.2. Questionnaire Capture

Data capturing is shown in the left bottom part of Figure 1. Data capturing is done through survey instances called Questionnaires. One survey can therefore have several Questionnaires associated to it. When a user starts a capture, the Survey Client Generator module will read the survey definition from the database and will dynamically generate all the input controls specified in the survey to generate a Questionnaire page. In addition, for every expression specified in constraints, variables, computed values and conditionals, the Compiler Generator component is asked to generate the valid grammar and compiler, and it also generates all the client-side code (in *JavaScript*) that is necessary to validate the data entered by the users when filling the questionnaire. The code is a *generated validation engine* that runs in the client-side web browser that is specific to a given survey. The necessary *JavaScript* code is also attached to the question input controls to trigger validation callbacks, so as to always verify the consistency of the data that is entered. During the capture, the user will see immediately (on-the-fly) the validations that performed by the engine through messages indicating where wrong data is introduced and calculating expressions and constraints, guiding the user to complete the questionnaire.

SEA exposes several Web Services (WS), which interact with the client during survey design and data capture. These Web Services interact with the client through Ajax operations to enable interactivity. For example, during survey design, when the user types an expression, a web service is invoked which passes the expression to validate. The Web Service creates the grammar and the compiler associated to the survey, and returns a validation code in case the expression is valid. If the expression is invalid, the HTML and JavaScript code is generated to indicate in which part of the expression there is a potential error, and a list of valid expression identifiers and operators. The user can rapidly fix the expression error interactively.

2.3. Data Management

Another Web Service is used for Data Management when submitting and saving questionnaires. On the client side, a *JavaScript* function generates a data structure containing the value for every input control in the Questionnaire. This data is serialized in an optimized format (JSON[1]), and is sent to the Data Management Web Service. The Web Service de-serializes the data and saves it in the database. The serialization and deserialization takes care of correctly handing the data according to the question types. For some data types, only one value is saved (e.g. integer, decimal, text values), however for types having multiple answers (e.g. multi-choices), special handing is required. Similar procedure is required when restoring saved data in a Questionnaire that was saved beforehand. The saved values together with the question identifiers are serialized, and the data is set according to their type (several input fields might be associated to a single question). During data capturing, the user can save data explicitly, or automatic saving can be configured, which creates a background activity that saves the questionnaire every configurable amount of time.

2.4. State Management

Survey and Questionnaire states are handled by the Survey State Machine Manager component, which is used to verify the states during survey definition and capture. The state of the questionnaire changed during submission of data saving accordingly.

Every survey has a finite state machine associated to it. The Survey State Machine Manager component handles the following survey states:

[1] JSON stands for JavaScript Object Notation, a lightweight data-interchange format based in a subset of JavaScript.

- `CREATED`: The survey has been created and can be edited but not captured.
- `STARTED`: The survey has been started and can be captured but not edited.
- `DISABLED`: The survey has been disabled after been started. It can be edited but cannot be captured.
- `FINISHED`: The survey is finished and cannot be captured or edited anymore.

During the survey definition process, it is not possible to capture data (only preview is allowed). The survey is in the `CREATED` state. Once the survey definition is done, the survey owner can start the survey to enable data capturing, which puts the survey in the `STARTED` state. At any moment, the survey manager can put the survey in the `DISABLED` state, so as to allow modification in the questions. The modification of questions may put the associated questionnaires (which have stored answers in the system), in an inconsistent state, e.g. if questions are removed or constraints are changed, which requires a re-validation of questionnaires[2]. Finally, when the survey is finished, it is put in the `FINISHED` state, which prevents any further modification.

Similar to Surveys, Questionnaires have different states, which are also handled by the Survey State Machine Manager:

- `INPROGRESS`: The default state after creation. The user can fill the Questionnaire.
- `SAVED`: The user has explicitly saved data. The user can continue working in the questionnaire later on.
- `SUBMITTED`: The user has successfully transcribed the questionnaire. Depending on the survey options, the questionnaire must be validated by a transcription validator.
- `QUARANTINED`: If one or more constraints are broken, but a justification is provided, the questionnaire will require manual validation. This state is only possible with *weak* and *flexible constraints*, which will be explained later.
- `FINALIZED`: The questionnaire has been approved as valid, and cannot be modified anymore.

Before being able to submit the final completed Questionnaire, a complete validation is performed on the client-side. Only if the full data is valid and consistent (according to the specified constraints) the questionnaire can be submitted.

3. SURVEY DEFINITION

SEA enables the definition of questions, which are grouped in Sections. It is possible to associate constraint expressions to any question, so as to link the validation of a group of questions according to the expression. The expressions are formed using question identifiers, together with operators (according to the different question types). Also, it is possible to use user-defined variables or computed values, which are evaluated to numeric values. Finally, Boolean expressions can be associated to the different questions to define conditional displaying, which allow to show or to hide questions.

3.1. Question Types and Basic Validation

SEA allows user to design surveys using different question types. The questions are basic types and advanced types. The following basic types are supported:

- *Decimal:* A decimal value
- *Integer:* An integer value
- *Multiple Choice (inclusive):* Several choices among a list of choices
- *Multiple Choice (exclusive):* Only one choice among a list of choices
- *Rating Scale:* Number between a range
- *Percentage:* A percentage (by default 0-100%, user can define ranges)
- *Boolean:* True/False or Yes/No value
- *Year:* A year
- *Date:* A valid date in ISO format YYYY-MM-DD
- *Time:* A valid time in format HH:MM
- *Currency:* An amount in a given currency
- *Currency Type:* A given currency
- *Country:* A given country
- *Text:* A free text

In addition, the following advanced types are supported:

[2] We will discuss this issue later, showing how SEA performs on-server validation.

- *Location (Geotag)*: A location coordinates using Google Maps ,
- *Image/File*: An image or file in different file formats (`.jpg`, `.png`, `.gif`, `.pdf`, `.docx`, etc.)

Every question in a survey has a ***unique identifier***, which is valid through all the survey (independently of the section where it is defined). This unique identifier (together with the type of the question) is used to define expressions in constraints, computed values, variables or conditional displaying. For example, the identifier "`Age`" can be defined of type Integer. Basic validation is performed on the question type, i.e. SEA verifies, during questionnaire capture, if the user has input data of the correct type. For example, a non-numeric value for the "`Age`" question will trigger an invalid type message.

The survey designer can specify if the question is mandatory or not. If a question is mandatory, the SEA forces the user to input a valid value. If a question is not mandatory, the user can specify three types of optional values, which are exclusive between them:

- Do not know (`.d`)
- Not applicable (`.n`)
- Refuses to answer (`.r`)

SEA adapts the input control according to the type. For example, if the input control is a text box (e.g. integer, decimal, etc.), it generates validation functions to take into account these new options. In the case of an input with an explicit list (e.g. Boolean checkboxes, list of countries, multiple choices), the new options are added and the necessary code to validate these values is generated.

3.2. Expression, Constraints and Compiler Generation

Constraints are restrictions that can be applied to questions. A constraint is a *Boolean formula* that is associated to any question and validates its value (additionally to basic validations). For example, if we need to limit the "`Age`" question, so as to restrict the possible values to be bigger than 18, a constraint can be added with the intuitive constraint formula "`Age > 18`". If several constraints are associated to a question, they are combined with the `AND` Boolean operator.

SEA supports three types of Constraints: *hard*, *flexible* and *weak* constraints. Hard constraints are constraints that must be evaluated to true in order to pass validation. Flexible constraints can be evaluated to false, and are accepted if and only if a justification is given by the user (i.e. when a flexible constraint is invalid, an additional text box for a textual justification is added). An invalid flexible constraint (with justification) puts the questionnaire in Quarantine state (see Section 2.4 for details of State Management) and requires a person with the role of quarantine verifier to accept or not the justification. Finally, weak constraints are constraints that can be evaluated to true or false. Only a warning message is shown to the user if evaluated to false, but the validation passes.

In general, validation input values that are somehow related between them is not trivial. Even though ASP.Net provides basic validation capabilities (e.g. type validation, regular expression validation or custom validation with user-defined code), they typically target validation of a single input control. Actually, there are three main difficulties in SEA: First, because the input control is generated dynamically in a per-survey basis, it is not possible to gather the actual identifier of the control[3]. Second, because of this dynamic generation of controls, performing the validation on the server may require a *full interpreter* of the SEA constraint language. Also, because constraint formulas may create complex dependencies between questions, dependency analysis becomes very complex. Finally, actually there is no single SEA constraint language. In fact, the constraint language is survey-specific (depending on the user-defined question identifiers and types), requiring several different interpreters.

In order to solve these issues, we leverage the *JavaScript* interpreter on the client side. That is, rather than developing constraint language interpreters, we generate *JavaScript* expressions directly from the different survey-specific constraint languages, which can be natively interpreted in the web browser[4]. For this, we use the powerful *jQuery* library [6], which allows flexible manipulation of document elements and event handling, so as to refer to the correct input controls, using the question identifiers only. We implemented the SEA Library containing all necessary survey validations. We use the *jquery.validate* plugin library [7], which allows defining validation classes and method. Validation classes can be added or removed dynamically to inputs (i.e. adding/removing a css style sheet 'class' to an input). Validation rules described as methods set a valid state (return true) or an invalid state (return false). In SEA, we

[3] Input controls in ASP.Net are uniquely defined in a Page. Their identifiers are generated when the `runat='server'` attribute is used. The `ClientID` attribute can be used in static JavaScript code in the control definition (e.g. using `<%= AControl.ClientID %>`, which could be put in embedded code for a custom validation) but this value is not available during page generation.

[4] All major web browsers support *JavaScript* by default.

developed pre-defined validation methods in the SEA Library and we also dynamically generate validation methods (we will see different examples of validation methods later). Validations methods can also be added or removed dynamically from inputs as rules and can be triggered explicitly to verify if some input is valid or not.

Figure 2 shows part of the SEA Library defining several generic functions. Examples of these functions is the `RegisterInputAccess()` function (lines 2-5), which is used to initialize the mapping of the actual value holders, and the map for the QuestionID for each question. The SEA Library includes functions, e.g. to Save and Restore data through Ajax (not shown).

We can also see the initialization code for all the required structures (lines 10-12). The `$.IDMap` object is used to store the mapping between question identifiers (e.g. 'Age') with the first part of the actual ASP.Net generated identifier. The `$.IDHolderMap` object is used to identify the second part of the name, which points to the input holder (each type question has a different holder). The combination of these entries (i.e. `$.IDMap` and `$.IDHolderMap`) allows to identify the actual input for every question. For example `$.IDMap['Age'] + $.IDHolderMap['Age']` gives the actual input name of the dynamically generated control associated to the '`Age`' question.

```
1  // SEA Library
2  function RegisterInputAccess(key, holder, questionid) {
3      $.IDHolderMap[key] = '_' + holder;
4      $.QuestionIDMap[key] = questionid;
5  };
6  ...
7
8  $(document).ready(function () {
9
10     $.IDMap = new Object(); // map for Question IDs with ASP generated unique IDs
11     $.IDHolderMap = new Object();// map for the actual value holder
12     $.QuestionIDMap = new Object(); // map for the QuestionID
13
14     // Obtains all the IDMap entries for all the questions in the questionnaire
15     $(document.body).find('[name$=idhf]').each(function () {
16         $.IDMap[$(this).val()] = this.id.substring(0, this.id.lastIndexOf('_'));
17     });
18
19     ...
20
21     $("#form1").validate({
22       ... // validator initialization options (not shown)
23       ... // initializes the $.validator object
24     });
25     ...
26     $.validator.addMethod('integer', function (value, element) {
27         return this.optional(element) || /^-?\d+$/.test(value + ''); // (positive or negatives)
28     }
29     , 'Integer value only'); // Message to show when wrong value is entered
30 }
```

Figure 2 - Excerpts of the SEA Library.

The initialization of the `$.IDMap` object is done in the SEA Library and is valid for any Questionnaire. For this, every ASP.Net control corresponding to a Question must define a hidden field as follows: `<asp:HiddenField ID="idhf" runat="server"/>` (where "`idhf`" stands for Identifier Hidden Field). During dynamic instantiation of the control, the corresponding Question identifier (e.g. '`Age`') is set to the hidden field as value. Thus, the code in lines 15-17 in Figure **2** gathers the first part of the name of the corresponding control. For example, for a hidden field with name "`cp_ctl07_idhf`", for the '`Age`' question, then `$.IDMap['Age']` will have '`cp_ctl07`' as value.

During the generation of the Questionnaire page, every user control is instantiated and initialized. Additionally, *JavaScript* code is generated to validate constraints. Figure 3 shows part of the resulting Questionnaire page with the generated input for an integer "`Age`" question, and the code generated to validate the simple constraint expression "`Age > 18`" (we will discuss later the constraint language in detail). We can observe in line 5 of Figure 3 the actual input with the generated ASP.Net identifier. Also, line 7 has the hidden field used to obtain the first part of the identifier's value (i.e. "`cp_ctl07`").

Lines 12-36 of **Figure 3** contain the generated *JavaScript* code for initialization and validation of the constraint. This code is executed once when the Questionnaire page is loaded. First, line 13 has a call to the `RegisterInputAccess(…)` function, passing the question identifier ("`Age`"), the holder identifier corresponding to an "integer survey text box"

("`istb`") and the QuestionID (100). In line 14, a variable is set with the complete input identifier name that can be used by *jQuery* to access the input[5].

```
1  ...
2  // Dynamically Generated Control
3  <div class="surveyQuestion">
4      <span id="cp_ctl07_lqt" title="Age" class="label">Age</span>
5      <input name="ctl00$cp$ctl07$istb" type="text" id="cp_ctl07_istb" class="integerQuestion" />
6  </div>
7  <input type="hidden" name="ctl00$cp$ctl07$idhf" id="cp_ctl07_idhf" value="Age" />
8  <input type="hidden" name="ctl00$cp$ctl07$typehf" id="cp_ctl07_typehf" value="INTEGER" />
9
10 ...
11 // Generated code for validation
12 $(document).ready(function() {
13   RegisterInputAccess('Age', 'istb', 100); // Initializes the Maps
14   var ageQ = '#'+ $.IDMap['Age'] + $.IDHolderMap['Age'];
15
16   $(ageQ).addClass('required'); // The question is MANDATORY
17   $(ageQ).rules('add', { integer : true }); // The question is of type integer
18
19   $(ageQ).addClass('Constraint_10_100'); // Register the constraint function
20
21   // Constraint function definition
22   $.validator.addMethod('Constraint_10_100', function(value, element) {
23
24     var ISEMPTY = 0;
25     // GENERATED EXPRESSION from 'Age > 18' CONSTRAINT
26     if(parseFloat(($(ageQ).val() == '' ?(ISEMPTY = NaN): $(ageQ).val())) > 18 ){
27       return true;
28     }
29     if(isNaN(ISEMPTY)) {
30       return true;
31     }
32     return false;
33   }, "Must be an adult");
34   ...
35
36 });
```

Figure 3 - Excerpts of a Generated Questionnaire.

Line 16 of Figure 3 adds basic validation to the input using *jquery.validation* library. The '`required`' class is used for mandatory inputs in the *jquery.validation* library. In line 17, a new validation rule is added to the input, which validates if the input value is an integer number[6]. The definition of this validation rule is shown in Figure 2 in the SEA Library in lines 26-29. The validation rule is triggered every time that the user enters a value in the input. If the validation is evaluated to false, the error message in line 29 of Figure 2 is displayed.

The code in line 19 of Figure 3 associates a generated constraint to the "`Age`" question input. The constraint code is shown in lines 22-33. Similar to the validation of the integer value, the '`Constraint_10_100`' generated validation method is evaluated every time that the user enters a value on the associated input field. The input remains invalid when it returns false (and the message in line 33 is displayed), and passes validation when it returns true. Note that the constraint function's name contains the ID of the constraint (10) and the ID of the question to which it is associated (100), which makes the name unique.

The constraint expression "`Age > 18`" written by the Survey designer is compiled into the *JavaScript* expression of line 26 of Figure 3. Note that the `ISEMPTY` local variable is used to validate empty values, which otherwise will fail the conversion to a numeric float value. If a `NaN` (not a number) is set in the conditional, then the whole conditional is evaluated to false. The standard `isNaN()` *JavaScript* function checks if the `ISEMPTY` numeric value has been set to `NaN` within the generated expression, and returns true if it is the case. This technique is applied for all generated *JavaScript* expressions for constraints, computed values, variables and conditional displaying.

[5] In *jQuery*, `$("#theid")` allows to access to an element with an identifier equal to "`theid`".

[6] Note that several basic type validation methods are defined in the jquery.validation library. However, the default 'integer' rule in *jquery.validation* is incorrect (does not support negative values), and we had to define a corrected version on the SEA Library. Of course validation methods for several question types supported by SEA are also defined in the SEA Library.

3.3. SEA Code Generation

The expressions in constraints, variables, computed values and conditional displaying are survey-specific and depends on the question identifiers and types. Thus, it is not possible to define a single grammar or compiler for all surveys. Our approach is to generate the grammar for the survey each time that an expression is required to be validated (e.g. during survey design) or to generate the corresponding *JavaScript* expression when generating the Questionnaire.

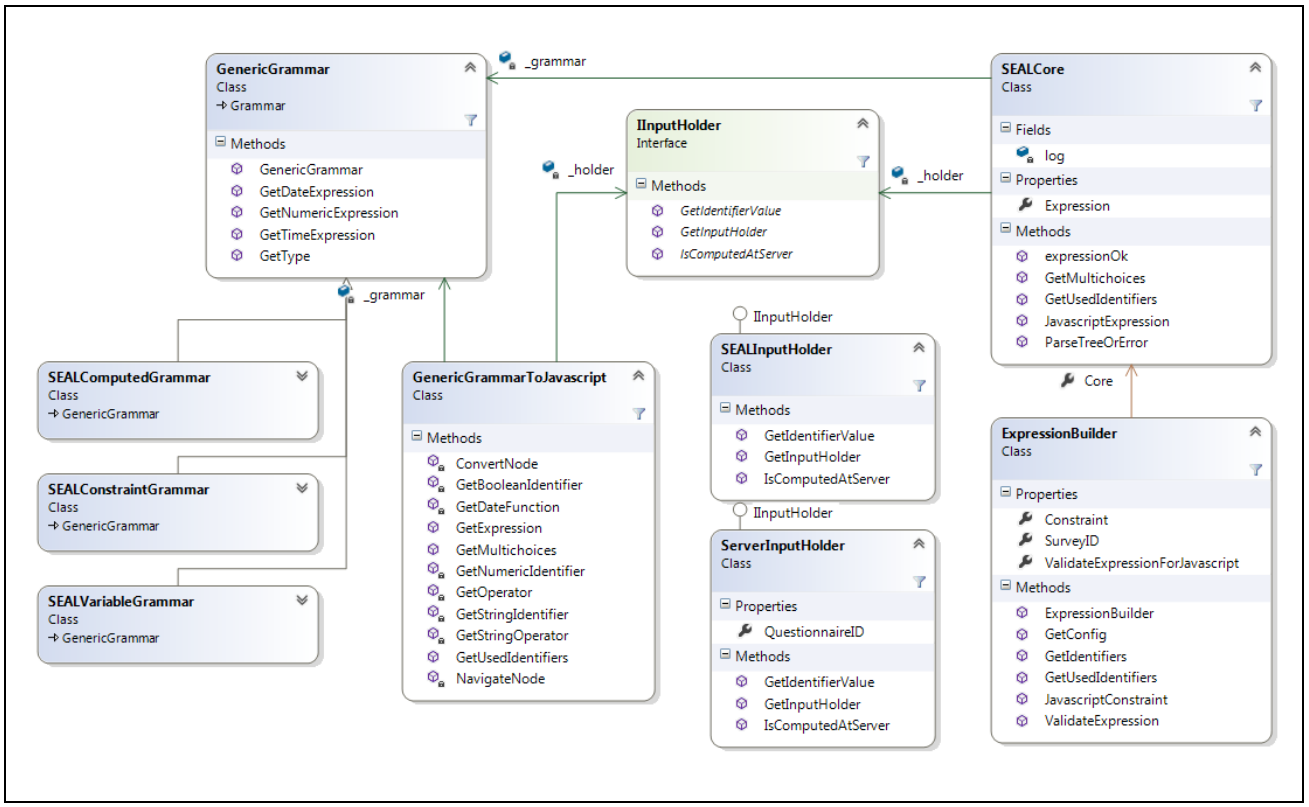

Figure 4 - Class Diagram of SEA Constraint Language Code Generator**.**

Figure 4 shows part of the class diagram of the Compiler Generator component of SEAL, our SEA Constraint Language. The `GenericGrammar` class defines the common compiler construction elements (grammar terminals, non-terminals and Backus-Naur Form (BNF) rules [8]). Several subclasses of this class specialize the generic grammar (e.g. the `SEALConstraintGrammar` is for *boolean expression* for constraints, whereas the `SEALComputedGrammar` is for customized *numerical expressions* for computed values). Our implementation is based on the *Irony .Net Language Implementation Kit* [9] to build SEAL Grammars.

The `ExpressionBuilder` is the main class for generating the corresponding grammar to validate an expression and generate *JavaScript* code. It instantiates a `SEALCore` class specifying the correct grammar. The `SurveyID` is used to read the questions identifiers and types for a given survey from the database. These identifiers and types are used to create the corresponding grammar for that particular survey. Then, a parser is instantiated to validate the expressions against the specific grammar.

The `ExpressionBuilder` is used also to generate the *JavaScript* code corresponding to a valid expression. For this, it instantiates the `GenericGrammarToJavascript` class. Its `NavigateNode(…)` method is used to traverse recursively the grammar tree (starting from the root node) for a given expression. The expression is therefore transformed from SEAL to *JavaScript* and returned to the caller. The `IInputHolder` interface is used to customize the generation of the *JavaScript* expressions, depending if the code will be evaluated on the client or on the server. For example, the `SEALInputHolder` will generate *JavaScript* with *jQuery* expressions to access the values of the input elements as shown in line 26 of Figure 3 (i.e. the `GetIdentifierValue(…)` method generates the *JavaScript* + *jQuery* code to access the input on the Questionnaire page). The `ServerInputHolder` on the other hand, is not meant for generating *JavaScript* to validate interactive user input, but rather to validate answers to questionnaires stored on the database. Thus, the generated *JavaScript* code will depend on the Questionnaire rather than on the Survey. For example, for a given Questionnaire, if the user has answered the value `20` for the question '`Age`', then for the constraint expression "`Age > 18`", the `ServerInputHolder` will help to generate the *JavaScript* code "`parseFloat(20) > 18`" which can be directly

evaluated (i.e. the `GetIdentifierValue(…)` for question "`Age`" reads the database answer value and returns "`parseFloat(20)`").

Note that the evaluation on server is used to re-validate answers on questionnaires that are already submitted, in quarantine or finished, for example, when the survey manager decides to modify constraints after a survey has been started. The evaluation of *JavaScript* on the server is done using the *Jint JavaScript interpreter for .Net* [10]. This choice was to avoid implementing SEAL specific interpreters, as the *JavaScript* generated expressions are just simple numerical and string operations.

4. SEAL: THE SEA CONSTRAINT LANGUAGE

One of the main design goals of SEA was to allow users to write constraint expressions intuitively, without having to learn any complex programming language with complex syntax. SEAL, the SEA Constraint Language is based on Question identifiers, simple operators and predefined functions to express constraints.

4.1. SEAL Grammar and definitions

Terminals:

The following *Boolean Terminals* are accepted:

- "`true`" or "`TRUE`" : true
- "`false`" or "`FALSE`" : false

A *Number Terminal* is positive or negative number (e.g. `10`, `10.12`, `-100`, `-1.2`)
A *String Terminal* is a string between simple quotes (e.g. `'a string'`)

Type Identifiers:

SEAL supports several question types as described in Section 3. The language supports the following types of identifiers: Boolean, Numeric, Date, Time, String, Multichoice Inclusive, Multichoice Exclusive, Variable identifiers grouped as follows:

- *BooleanIdentifier:* Boolean question type
- *NumericIdentifier:* Integer, Decimal, Percentage, Rating Scale, Year, Currency and Computed question types
- *DateIdentifier:* Date question type
- *TimeIdentifier:* Time question type
- *StringIdentifier:* Text, Currency Type, Multichoice Inclusive, Multichoice Exclusive, and Country question types.
- *MultichoiceIdentifier:* Multihoice Inclusive or Exclusive question types.

These types of identifiers are used to allow operations and functions. Note that the Multichoice types can be used as strings and also with operations specific to Multichoice selections.
Basic Operators and expressions:

We define the following basic *Numeric Boolean operators*:

- "`>`" : Greater than
- "`<`" : Smaller than
- "`>=`" : Greater of equal than
- "`<=`" : Smaller or equal than
- "`=`" or "`==`" : Equal to
- "`!=`" : Different than

We define the following common *Numeric operators*: "`+`", "`-`", "`*`", "`/`" (addition, subtraction, multiplication and division).

The following *Boolean operators* are supported:

- "`&`" or "`&&`" or "`and`" or "`AND`" : The AND logic operator
- "`|`" or "`||`" or "`or`" or "`OR`": The OR logic operator
- "`not`" or "`NOT`" or "`!`" : Negation operator

The following Boolean String, Boolean Date and Boolean Time operators are accepted:

- **"="** or **"=="** **:** Equal to
- **"!="** **:** Not equal to

Note that SEAL expressions can be grouped with parenthesis as commonly found in any programming language.

Advanced operators and functions:

The "`isSelected()`" *Boolean Multichoice* operator is defined as follows:

MutichoiceIdentifier + "`.isSelected(`" + `TerminalNumber` + "`)`" **:** A multichoice identifier, followed by the "`.isSelected(`" operator, followed by the index of the selection to test, followed by ")".

This operator allows testing if the user has selected a given choice in a Multichoice question. Note that the index of the selection starts at index 0.

For example, assuming we have the following questions in a Survey:

- 'Married' of type Boolean
- 'Children' of type Multichoice Exclusive with the following choices:
  - "more" : Three children of more
  - "two" : Two children
  - "one" : One children
  - "none" : No children
- 'AnnualIncome' of type Integer

We can write a constraint testing if a person is married and his annual income is smaller than 10.000 or has more than 3 children, as follows:

```
"Married AND (AnnualIncome < 10000 OR Children.isSelected(0))"
```

Note that the 'Married' question is Boolean, so its value will be true or false. Also, note that it is also possible to use the value of the choice for the 'Children' question (as the multichoice questions support also string operators as mentioned before). Thus, the following expression is equivalent:

```
"(Married = TRUE) & (AnnualIncome < 10000 | Children = 'more')"
```

For questions of type Date, the following operators and functions are available:

- **"`>`"** : Date greater than
- "`<`" : Date smaller than
- **"`>=`"** : Date greater of equal than
- "`<=`" : Date smaller or equal than
- **"`=`"** or **"`==`"** : Date equal to
- "`!=`" : Date different than
- "`@day(`" + *DateIndentifier* + "`)`" : Gets the day numeric value from DateIdentifier
- "`@month(`" + *DateIdentifier* + "`)`" : Gets the month numeric value of DateIdenitifier
- "`@year(`" + *DateIdentifier* + "`)`" : Gets the year numeric value of DateIdentifier

Dates can be written in the form "`[`" + *Year* + "`-`" + *Month* + "`-`" + *Day* + "`]`"
For example, if we have two questions of type date ("`StartDate`" and "`EndDate`") and a question of type year ("`StartYear`"), we can write a constraint saying that the year of the `StartDate` must be equal to the `StartYear` and the `EndDate` must be smaller than 2013-01-01, as follows:

```
(@year(StartDate) = StartYear) AND (EndDate < [2013-01-01])
```

For questions of type Time, similar operators as for Date type are available.

Times can be written in the form "`{`" + Hour + "`:`" + Minutes + "`}`". Also the "`@hour(`" + *TimeIdentifier* +"`)`" and "`@minute(`" + *TimeIdentifier* + "`)`" functions are available.

For example, a valid constraint with a "`SubmissionTime`" question of type Time will be:

```
SubmissionTime < {12:00}
```

or numeric values it is also possible to evaluate the `Min` and `Max` values, as follows:

"`Min(`" + *NumericExpression* + "`,`" + *NumericExpression* + "`)`"
and
"`Max(`" + *NumericExpression* + "`,`" + *NumericExpression* + "`)`"

The *NumericExpression* can therefore be any expression evaluated to a numeric value. For example, taking the previously defined questions, the following expression is valid:

```
Min(@day(StartDate), 25) + Max(10, @hour(SubmissionTime)) != 100
```

4.2. Variables and computed values

SEA also allows defining 'opaque' variables that can be used in any expression. Variables are value holders that store input values of numeric types. Only numeric types are allowed, because it is not possible to infer the actual type of a variable for validating an expression. For example, assuming the question "`Age`", and we define a variable "`Average`", we can define the constraint expression "`Age + Delta < 30`". Thus, "`Delta`" can be used to hold any evaluated numerical value. For example, "`Delta`" can be defined with an expression, such as "`(@year(StartDate) + @year(EndDate))/2)`" or whatever valid numerical expression. Enabling variables to hold other types (e.g. a text type) would mean that SEA should infer the type of "`Delta`" and invalidate the expression (because we cannot use the "`+`" operator between an integer and a text), which is of course out of the scope of SEA.

Similar to variables, SEA allows the use of Computed values, which have numerical expressions associate to them. Note that both Variables and Computed values have identifiers, and can be used in any SEAL expression (constraints, or other).

The first difficulty with Computed values is that the evaluation of the computed value expression needs to be triggered every time that one of the values of the question identifiers used in the expression change. Let's assume that we have two questions of type Year ("`StartYear`" and "`EndYear`") and we want to calculate a Computed value for the "`Duration`" with the expression "`EndYear - StartYear`". Thus, we have to trigger the evaluation of the expression each time that any of the inputs associated to the question identifiers in the expression change. For this, we define in the SEA Library a jquery.validation method called '`ComputedVarValidator`' (see lines 31-36 in Figure 5).

```
1  // SEA Library
2  // Calls all the functions associated to a computed value
3  function computedVarValidate(theElement) {
4      var fctarray = $.ComputedVar[theElement];
5      if (fctarray != undefined) {
6          for (var i = 0; i < fctarray.length; i++) {
7              var fct = fctarray[i];
8              fct(); // call the function !!!
9          }
10
11     }
12 };
13 // Sets the computed value to the corresponding input, or the message
14 // if expression is evaluated to NaN
15 function EvalComputedVar(expression, identifier, msg) {
16    if(isNaN(expression)) {
17        $('#' + $.IDMap[identifier] $.IDHolderMap[identifier]).val(msg);
18    }else{
19     $('#'+ $.IDMap[identifier] + $.IDHolderMap[identifier]).val(expression);
20    }
21 };
22
23 ...
24
25 $(document).ready(function () {
26
27     ...
```

```
28      // The array holding validation functions
29      $.ComputedVar = new Array();
30       ...
31      $.validator.addMethod('ComputedVarValidator', function (value, element) {
32           // get the id from the Hidden Field (after the div)
33           var theId = $(element).parent().next().val();
34           computedVarValidate(theId); // evaluate the computed value
35           return true; // always return true, as it's not a on-the-fly validation
36       }, 'Should never happen');
37  }
```

Figure 5 - Excerpts of the SEA Library for Computed Values evaluation.

The role of this method is to trigger the evaluation of the generated *JavaScript* expression associated to any of the questions whose identifiers are in the expression. Therefore, we register this validation method to the inputs of the "`StartYear`" and "`EndYear`" questions (see the generated code to register the inputs with the validation method in lines 11 and 17 in Figure 6).

```
1   ...
2   // Dynamically Generated Controls (not shown)...
3   ...
4   // Generated code for validation
5   $(document).ready(function() {
6     RegisterInputAccess('Duration', 'cstb', 300); // The textbox to show computed value
7
8     RegisterInputAccess('StarYear', 'ystb', 200); // The Start Year input textbox
9     var startYearQ = '#'+ $.IDMap['StartYear'] + $.IDHolderMap['StartYear'];
10
11    $(startYearQ).addClass('ComputedVarValidator'); // Should trigger evaluation on change
12
13    ...
14    RegisterInputAccess('EndYear', 'ystb', 201); // The End Year input textbox
15    var endYearQ = '#'+ $.IDMap['EndYear'] + $.IDHolderMap['EndYear'];
16
17    $(endYearQ).addClass('ComputedVarValidator'); // Should trigger evaluation on change
18
19    ...
20
21
22     if ($.ComputedVar['EndYear'] == undefined) { $.ComputedVar['EndYear'] = new Array(); }
23      // Register function to evaluate for 'EndYear' input
24     $.ComputedVar['EndYear'].push(function () {
25          EvalComputedVar(
26            // Generated expression for 'EndYear - StartYear'
27            parseFloat(($(endYearQ).val() == '' ? (ISEMPTY = NaN) : $(endYearQ).val()))
28            -
29            parseFloat(($(startYearQ).val() == '' ? (ISEMPTY = NaN) : $(startYearQ).val()))
30          , 'Duration', '...calculating...');
31        }
32      );
33     if ($.sangComputedVar['StartYear'] == undefined) { $.sangComputedVar['StartYear'] = new Array(); }
34     // Register function to evaluate for 'StartYear' input
35     $.ComputedVar['StartYear'].push(function () {
36          EvalComputedVar(
37            // Generated expression for 'EndYear - StartYear'
38            parseFloat(($(endYearQ).val() == '' ? (ISEMPTY = NaN) : $(endYearQ).val()))
39            -
40            parseFloat(($(startYearQ).val() == '' ? (ISEMPTY = NaN) : $(startYearQ).val()))
41          , 'Duration', '...calculating...');
42        }
43     );
44  ...
45  });
46
```

Figure 6 - Excerpts of the generated code for Computed Values evaluation.

Because the evaluation must be done on-the-fly, we use *JavaScript*'s feature to manipulate functions as first-class objects. Therefore, for each question identifier that is used in any computed value or variable, we generate the function that evaluates the expression. The SEA Library defines the `$.ComputedVar` array to hold functions associated to

question identifiers used in Computed values expressions (see line 29 of Figure 5). In our example, we need to generate the function to evaluate the expression "`EndYear - StartYear`", and register this function to both question identifiers, because we need to calculate the expression when any of these inputs are changed. Note that because a question identifier can be used in arbitrary number of expressions for Computed values or variables, several functions can be associated to every question identifier. For this reason, every entry in the `$.ComputedVar` array, will store an array of functions. Lines 22 and 33 of Figure 6 show the initialization of the arrays to hold the generated functions for the '`EndYear`' and '`StartYear`' respectively. These generated functions are then pushed in the array (see lines 24 and 35 of Figure 6) for later evaluation. The functions bodies are actually an invocation to the `EvalComputedVal(...)` function defined in the SEA Library (see lines 15-21 in Figure 5), which basically receives as first argument the generated *JavaScript* expression to evaluate, then the identifier of the Computed value where to put the computed value (i.e. the 'text box' associated to the '`Duration`' computed value), and a message in case the expression cannot be evaluated (for this we use the same mechanism using `NaN` to handle empty inputs as described before).

The evaluation of the functions associated to the inputs of an expression is done con the `computedVarValidate(…)` function of the SEA Library (see lines 3-12 in Figure 5). This function is called when the validation method `ComputedVarValidator` is triggered on a given input value, which identifier is passed as argument. All the associated functions are then evaluated dynamically (see line 6-9 in Figure 5).

5. CONDITIONAL DISPLAY

SEA provides a mechanism called "Conditional Display" that allows to evaluate arbitrary Boolean expressions, so as to define conditions that enable or disable the displaying of a given set of questions. Thus, the user can associate conditional displaying to any question, specifying the expression, and selecting a list of questions to be shown (in case that the condition is evaluated to true) and a list of questions to hide (in the opposite case).

For each Conditional Display, a validation method is generated and associated to the corresponding input question. Then, the *JavaScript* expression is generated to test the conditional. If the conditional is true, for each question in the list of questions to show, we generate the necessary code to: (a) enable the input (b) put (back) the validation rules, and (c) mark the question to bypass validation (we an array defined in the SEA Library, that stores for each question a flag indicating if validation should be bypassed or not). For each question in the list of questions to hide, we generate the necessary code in opposite logic, to: (a) disable the input, (b) remove the validation rules, and (c) mark the question to skip validation bypass. If the conditional is false, the code with opposite logic for both show and hide is generated.

In Figure 7 the generated pseudo-code for a Conditional Display is shown.

```
1  // Generated code for Conditional Display
2  $(document).ready(function() {
3  ...
4  $.validator.addMethod('ConditionalDisplay_1', function (value, element) {
5     if (! (generatedJavaScriptExpression)) {
6        // For each Question in the "To Show List"
7         enable_input
8         add_validation_rules
9         $.ConditionalBypass['QuestionID'] = false;
10        // For each Question in the "To Hide List"
11        disable_input
12        remove_validation_rules
13        $.ConditionalBypass['QuestionID'] = true;
14        return true;
15    }
16    // For each Question in the To Show List
17    disable_input
18    remove_validation_rules
19    $.ConditionalBypass['QuestionID'] = true;
20    // For each Question in the "To Hide List"
21    enable_input
22    add_validation_rules
23    $.ConditionalBypass['QuestionID'] = false;
24    return true;
25 }, 'Should never happen');
26 });
27 ...
```

Figure 7 - Generated code to handle Conditional Display.

The enabling/disabling input is important, because the user should not be allowed to put any input in the questions that are hidden. Also, removing the validation rules is important, because hidden inputs must not be considered in the global validation. Otherwise, inputs that are supposed not to be taken into account may invalidate the Questionnaire. Finally, the entries in the `$.ConditionalBypass` array have two goals: The first one is to skip the generation of data to be stored in the database. That is, for every input that is hidden, the function to serialize answers (save or auto-save procedure) will skip those inputs. The second goal is to skip constraint validations, i.e. all the inputs that are hidden are also skipped. In the first case, the code to skip data generation is defined in the SEA Library, whereas the code to skip constraint validation is generated at the beginning of every constraint validation method (e.g. see line 22 in Figure 3), before the evaluation of the generated *JavaScript* expression, so as to skip the constraint validation.

6. DEPENDENCY ANALYSIS FOR ON-THE-FLY VALIDATION

Because of the flexibility of the SEA Constraint Language, which allows to write expressions that connects questions identifiers in any part of the survey, the validation of the whole questionnaire becomes complex. The main issue is that the validation methods have to be triggered according to the order in which the questions are connected. The basic jquery.validation framework does not have a predefined validation order, and it can happen that the registered order is not consistent. In addition, SEA allows performing on-the-fly validation, which helps the user to see valid or invalid input immediately while entering data during questionnaire capture. This feature is important, because all the validation methods that affect constraints, computed values, variables, and conditional displaying need to be triggered every time a value is modified. Triggering a full validation is not an option, because validation of the full Questionnaire every time that the user enters some data harms performance even for medium-size surveys[7]. We therefore apply a dependency analysis that limits the set of question inputs to validate.

The dependency analysis builds for each question identifier or computed value identifier, a list of identifiers that are related to it through any type of expression. The dependency analysis is done in two passes. The first pass creates direct dependencies for every question identifier, and the second pass expands indirect dependencies using the dependency information created in the first pass. Note that the goal of this dependency analysis is to generate a list of identifiers to trigger fast validation when an input is modified, and therefore there is no verification of circular dependencies.

Figure 8 shows the pseudo-code of the two-passes dependency analysis.

```
1  var dependencyDictionary = new Dictionary<string, List<string>>();
2  var finalDependencyDictionary = new Dictionary<string, List<string>>();
3  // First Direct dependency creation
4  var directDependents, varDictionary = new Dictionary<string, List<string>>();
5  varDictionary = for each variable id creates the list of used identifiers in expression;
6  foreach question q in Survey do
7     if q is COMPUTED then
8       var usedIdentifiers = get used identifiers from computed expression
9       foreach id in usedIdentifiers do
10        if id is in varDictionnary then
11           add uniquely the list from varDictionary in directDependents with key q
12        else
13           add uniquely id in directDependents with key q
14      endforeach
15    endif
16    foreach contraint c associated to q do
17        var constraintUsedIdentifiers = get used identifiers from constraint expression
18        foreach id in constraintUsedIdentifiers do
19           if id is in varDictionary then
20             add uniquely the list from varDictionnary to directDependents with key q
21           else
22             add uniquely id in directDependents for key q
23       endforeach
24    endforeach
25    foreach conditionalDisplay cd associated to q do
26        var condUsedIdentifiers = get used identifiers from conditional expression
27        foreach id in condUsedIdentifiers do
28           if id in varDictionary then
29             add uniquely the list from varDictionary to directDependents with key q
30          else
31             add uniquely id in directDependents
32        add uniquely list of questions to show from cd in directDependents with key q
```

[7] Depending on the number of constraints and number of questions, a full validation can take more than 2 seconds. Letting the user wait two seconds every time that it enters one data makes the system unpractical.

```
33          add uniquely list of questions to hide from cd in directDependents with key q
34      endforeach
35  endforeach
36
37  // Final dependency creation
38  foreach element in directDependents do
39      var dependents = all keys in directDependents where element.key is in directDependents.values
40      var newElements = new List<string>();
41      foreach d in dependents do
42          foreach depElem in directDependents[d] do
43              add uniquely depElem in newElements
44          endforeach
45          add uniquely d in newElements // adds itself
46      endforeach
47      foreach depElem in directDependents[element] do
48          add uniquely depElem in newElements
49      endforeach
50      add newElements in finalDependencyDictionary with key element.key
51   endforeach
52  // Generates the dependency list
53  foreach dep in finalDependencyDictionary do
        generate JavaScript "$.ValidateList[' + dep.key '] = new Array(' + expanded dep.values + ');"
    endforeach
```

Figure 8 - Pseudo-code of the dependency analysis.

First, for every variable defined in the survey (if any), we build a list of all the identifiers that are used in the variable expression (lines 4-5). Note that the list contains a single instance of every identifier as expressions can refer several times to the same identifier. We gather all the question identifiers of the full survey and for every question identifier; we check if the identifier is an Computed value and get the list of used identifiers from the associated expression. If any of the identifiers refers to a variable, then we add the related identifiers from the variables dependencies (lines 7-15). Then, we gather all the Constraints associated to the current question, and do the same for every referred variable (lines 16-24). For every Conditional Display associated to the current question, we gather the list of used identifiers in the expression and add the identifiers to the list also checking if any variable was used (lines 25-31). Additionally, we gather the list of questions to show and questions to hide, and add them to the dependency list (lines 32-33). At the end of this first pass we have in the `dependencyDictionary`, for every question identifier (as key), an explicit list of direct dependency identifiers (as value).

The dependency analysis performs a second pass, to expand the list of dependent question using the first pass dictionary. That is, for every entry in the dictionary, it gathers the dependent list by checking all the entries in the `dependencyDictionary` where the identifier is present in any of the value list (line 39). Then, it creates a new list by putting all the direct dependents of every of those identifiers including the identifier itself (lines 41-46). Finally, it takes the original direct dependent elements, and adds the list to the new dependency list (lines 47-49). The resulting new expanded dependency list is added to the `finalDependencyDictionary` (as value) for the current question identifier (as key). For each element in this dictionary, we then generate the validation dependency list (`$.ValidateList` defined in the SEA Library) which is used to validate the elements every time that the input associate to an element is modified.

Figure 9 shows the `OnTheFlyValidation(…)` function that receives the input element that triggered the call.

```
1   // In SEA Library
2   function OnTheFlyValidation(elem) {
3       var parent = $(elem).parent();
4       while (!$(parent).hasClass('surveyQuestion')) {
5           var newParent = $(parent).parent();
6           parent = newParent;
7       }
8       var theID = $(parent).next().val();
9       var vallist = $.ValidateList[theID]; // Gather the dependency list
10
11      for (var i = 0; i < vallist.length; i++) {
12          var theDepElem = FindInputAccess(vallist[i]);
13          if (theDepElem != null && theDepElem.length > 0) {
14              $.validator.element(theDepElem); // Trigger validation
15          }
16      }
17  };
```

Figure 9 - On-the-fly validation through dependency list.

Using this element, we gather the actual question identifier (lines 3-8) and using this identifier we gather the dependency list from `$.ValidateList` (line 9). For each identifier in the list, we gather the actual input element (line 12), and trigger the validation of all the validation methods associated to it (line 14). This allows trigger evaluation of constraints, computed values, variables and conditional display of all the dependent questions through all the survey, independently where they are placed.

Figure 9 shows the `OnTheFlyValidation(…)` function that receives the input element that triggered the call. Using this element, we gather the actual question identifier (lines 3-8) and using this identifier we gather the dependency list from `$.ValidateList` (line 9). For each identifier in the list, we gather the actual input element (line 12), and trigger the validation of all the validation methods associated to it (line 14). This allows trigger evaluation of constraints, computed values, variables and conditional display of all the dependent questions through all the survey, independently where they are placed.

7. LIMITATIONS AND FUTURE WORK

One of the main limitations of the current SEA platform is the code bloat in large surveys[8] due to the complex code generation. Even though the system allows to validate surveys with complex dependencies and constraints with reasonable performance (thanks to the dependency analysis), the generated questionnaire can have large amount of JavaScript code associated to validation methods and complex logic handling of conditional display. We are implementing a caching mechanism in order to reduce the performance impact and code bloat on the client side.

Because of the heavy use of validation methods of the *jquery.validation* library, we had to adapt the framework to simplify some functions in order to cope with performance issues. We also implemented caching mechanism in the server side (notably to access de database), so as to reduce the performance impact on generating large questionnaires.

We are integrating to SEA, a module for data analysis, which will allow users to simplify visualization and filtering of survey data to perform sophisticated data analysis in an intuitive way.

8. CONCLUSION

In this article we presented SEA, a new web-based platform to design, capturing and processing large-scale surveys. In opposite to existing survey systems, SEA focus on data reliability, reducing the number of errors during data capture thanks to user-defined formulas, using an intuitive and simple constraint language. We showed compilation and code generation techniques enabling the handling of large surveys with complex dependencies. Our system uses standard web technology, including Web Services, Ajax and JavaScript enabling simple on-line survey design and capturing.

9. ACKNOWLEDGEMENTS

We would like to thank all the Artexacta Development Team for their effort on developing the SEA platform.

10. REFERENCES

[1] Google Inc. “Google Consumer Survey.” Internet: http://www.google.com/insights/consumersurveys/ 2010 [Accessed May 2013).
[2] Qualtrics. “Qualtrics Research Suite.” Internet: http://www.qualtrics.com/research-suite/. 2013 [Accessed June 2013].
[3] SurveyMonkey. “SurveyMonkey.” Internet: http://www.surveymonkey.com/. 1999. [Accessed June 2013].
[4] Widgix, LLC. “SurveyGizmo.” Internet: http://www.surveygizmo.com/ . 2005. [Accessed June 2013].
[5] Microsoft Corporation. “The .NET Framework.” Internet: http://www.microsoft.com/net/. 2002. [Accessed January 2012].
[6] J. Resig and the jQuery Foundation. “jQuery: The write less, do more, JavaScript library.” Internet: http://www.jquery.com/. 2006. [Accessed January 2012].
[7] jQuery validation plug-in. “Form validation with jQuery.” Internet: http://www.jqueryvalidation.org/. 2004. [Accessed January 2012].

[8] With more than 1000 questions.

[8] A. Aho, R. Sethi, J. Ullman. *Compilers, principles, techniques, and tools*. Addison-Wesley Longman Publishing Co. Inc. Boston, MA. USA, 1986.
[9] Irony. "Irony - .Net Language Implementation Kit." Internet: http://irony.codeplex.com/. 2007. [Accessed January 2012].
[10] Jint. "Jint – Javascript Interpreter for .Net." Internet: http://jint.codeplex.com/. 2010. [Accessed July 2013].