The 2016 AAAI Fall Symposium Series:
Cognitive Assistance in Government and Public Sector Applications
Technical Report FS-16-02

Smart Forms

Sudhir Agarwal, Abhijeet Mohapatra, and **Michael Genesereth**
Computer Science Department, Stanford University, USA
{sudhir,abhijeet,genesereth}@cs.stanford.edu

Abstract

We present Smart Forms, an innovative web forms technology for easy creation, maintenance, and evaluation of user-friendly web forms especially the ones that must implement complex laws, regulations, or business policies. In order to provide cognitive assistance to end users during form-filling, Smart Forms have built-in mechanisms for visual feedback, restriction of selectable values, and automatic form filling. Smart Forms can be created and maintained easily by declaratively configuring rather than procedurally programming these mechanisms. We also present the Smart Forms Editor which assists a Smart Form creator in creating data-driven form UI, editing, testing and verifying form rules, and testing and debugging a form.

Introduction

Many important tasks require people to fill forms, e.g., filing tax returns, applying for business permits, reporting financial compliance, and requesting health care data etc. Often, form data needs to be validated against certain laws and regulations by an authority. Forms may also contain instructions to assist people in supplying valid field values.

Often, the form-filling instructions as well as the laws and regulations against which the form data is validated are complex. For example, the fields 75 and 78 of the IRS Form 1040 for U.S. individual tax return depend directly or indirectly in complex ways on fields 44–74 as well as on external data, and the submitted form data needs to be validated and tax amount needs to be computed as per the tax rules. As a result, it is tedious for people to fill such forms, and it is hard for the authorities to validate and efficiently use form data.

Since the advent of the World Wide Web, more and more forms are offered as web forms in order to reduce the manual effort and infrastructure costs by performing validation automatically on the server-side and maintaining and processing the form data digitally. The subsequent introduction of JavaScript as a client-side scripting language also enables live feedback to the user during form filling (Flanagan 2011), which is typically achieved by automatically checking the form data against the form-filling instructions on the client-side when a user changes the value of a form field.

However, due to the lack of direct support for logical reasoning over complex conditions, it is difficult and error prone to procedurally program and maintain form behavior that is dependent on complex laws, regulations, or business policies. Often, field values depend on data external to a form. For example, an entered address is valid only if the entered combination of street, zip-code, city, state, and country actually exist. While more or more structured data is made available in the web, procedural programming languages generally also lack direct support for easily incorporating and efficiently querying remote structured data sources. This adds further to the difficulty in creating, maintaining, evaluating user-friendly web forms.

We present Smart Forms, an innovative web forms technology for easy creation, maintenance, and evaluation of user-friendly web forms especially the ones that must implement complex laws, regulations, or business policies. Smart Forms are meant to be created and maintained by domain experts themselves who are mostly not software programmers. Therefore, neither the creation and maintenance nor the evaluation of data of a smart form requires traditional procedural coding.

Smart Forms are created declaratively rather than procedurally by encoding as logical rules a form's immediate response to user's form-filling actions. The simple, compact and yet highly expressive syntax of the Smart Forms rule language also makes it easier to update Smart Forms when the constraints, laws, or regulations change. Automated procedures for evaluating form data against the constraints, laws and regulations enable live feedback to users at the time of form filling as well as efficient evaluation of form data after form submission. Smart Forms can use organization internal or publicly available integrated structured data to further enhance the usability of the form and correctness of the entered data.

We also present the browser-based Smart Forms Editor which makes it easy to import or connect structured data, design and generate the user interface in WYSIWYG and data-driven fashion, and edit form behavior rules.

Smart Forms have been successfully evaluated for many cases including undergraduate and masters program sheets within the Stanford Computer Science Department, IRS tax forms and FERPA compliance drafting of agreements between a service provider and a school district.

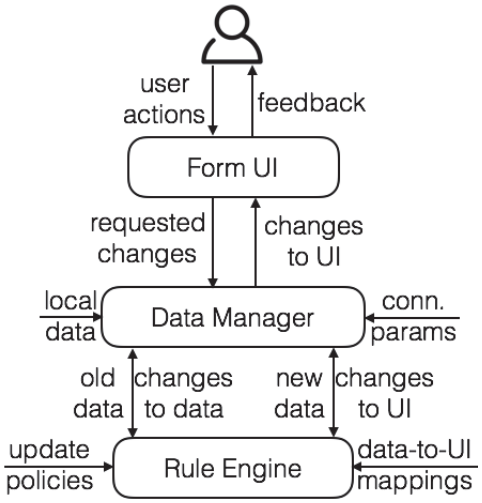

Figure 1: Smart Forms components involved in generating live feedback during form-filling

Smart Forms Architecture

In this section, we present an overview of the functioning of Smart Forms. We first describe the functioning of the real-time feedback mechanism of during form-filling and then the post-submission processing of a form's data.

Live Feedback during Form-Filling

Smart Forms can provide powerful and context-sensitive cognitive assistance to their users during form filling in form of auto-completion lists, disabled invalid options, instructions and warnings about constraint violations, and automatic filling fields with sensible values to satisfy complex constraints. A novelty of the Smart Forms technology is that it allows creation such user-friendly forms without having to write a single line of Javascript code.

Smart Forms are Logic Programming based implementation of the concept of Logical Spreadsheets (Kassoff 2011). Specifically, Smart Forms use extended Datalog with negation-as-failure, aggregates, built-ins, and update operations. Figure 1 shows the main components of Smart Forms that play a role in the real-time feedback to a user during form-filling. In this section we present the function and interplay of these components as well as the formalisms for representing their respective inputs and outputs.

Example 1 *Suppose the purpose of a Smart Form is to let a user choose at most three pizza toppings from a set* $T = \{t1, t2, t3, t4\}$ *of pizza toppings. Toppings* $t1$ *and* $t2$ *have 5g protein content each and toppings* $t3$ *and* $t4$ *have protein content 10g each. The pizza vendor wants to make it easy for the customer to select high protein topping combination while respecting customer's choice.*

Data Manager The Data Manager is responsible for evaluating queries over and managing changes to *form data*. Form data defines the state of the form at any time during the form-filling process and contains values of the form fields and any other data required for computing form fields values.

In Example 1, the data about toppings and their protein content is not large and can be provided to Data Manager as local data. The database of the Smart Form for Example 1 contains a unary relation *topping* to represent the set of toppings and a binary relation *protein* to represent the protein content of toppings (refer to (Genesereth 2015) for more information on the database syntax).

$$
\begin{aligned}
&topping(t1)\\
&topping(t2)\\
&topping(t3)\\
&topping(t4)\\
&protein(t1, 5)\\
&protein(t2, 5)\\
&protein(t3, 10)\\
&protein(t4, 10)
\end{aligned}
$$

In general, a form may require data that is too large to fit locally or resides in a large database. In order to support such cases, the Data Manager can also be provided with the connection parameters of remote databases. The Data Manager can manage data available to it locally as well as query remote data sources. In Example 1, instead of providing the data about set of toppings and their protein content as explicitly it could also provided by supplying the connection parameters to a database of nutrition facts.

Of particular interest is Jabberwocky [1], a web-based service that allows users to expressively query integrated structured data in the Web without themselves knowing anything about the location, vocabulary and schemas of the data sources (Agarwal, Mohapatra, and Genesereth 2016). Jabberwocky is intended as a general-purpose service that provides answers to questions (rather than a list of links to individual web pages) by examining the relevant databases itself and integrate the data in answering queries. Jabberwocky's schema is a conceptual schema and is thus easy to use for end-users. Currently, Jabberwocky's data graph contains over 3 Million facts, and its schema contains approx. 75 concepts and 500 attributes from multiple domains including US public data about territories, government (all levels and branches), companies, products, universities, and college sports.

Smart Forms use a special binary base relation *value* to represent the values of the fields of a Smart Form. The first argument of the *value* relation refers to a form field and the second argument is a value of the field. Note that form fields of type multi-select box or checkbox may have multiple values. Smart Forms use another special binary derived relation *view* to represent the computed values of the fields of a Smart Form. When there are multiple values for a single valued field, a Smart Form module selects one of the values. In Example 1, initially the base relation *value* is empty reflecting the state that no topping is selected.

[1] http://jabberwocky.stanford.edu

Form UI The Form UI is an HTML form that an end user interacts with in his or her browser. In Example 1, the UI of the Smart Form contains a checkbox for each topping. To keep the mapping simple, we use the object id of a topping as the id of the corresponding checkbox.

The Form UI intercepts a user's form-filling actions and generates a corresponding transaction request to the form database. In Example 1, if the user selects the topping $t1$, then the form's UI generates a transaction request $\delta^{+}value(t1, true)$. Similarly, when a selected topping $t1$ is unselected a transaction request $\delta^{-}value(t1, true)$ is generated.

Update Policies Smart Forms use *update policies* in order to efficiently compute a form's immediate response to a user's action. An update policy is a logic program (Kowalski 1988) with standard negation and four special operators δ^{+}, δ^{-}, Δ^{+} and Δ^{-}. An update policy defines a complete transaction to be performed on a database in response to a requested transaction) (Mohapatra, Agarwal, and Genesereth 2016). The declarative nature, simple syntax and high expressiveness of logic programs make it easy to express complex logical relationships among database relations. As a result, considering complex laws, regulations and policies in defining a form's response to a user's action becomes much easier than with a state of the art procedural programming language such as Javascript.

As mentioned above, Smart Form can automatically fill or unfill form fields with sensible values to assist a user during form-filling in entering values that satisfy possibly complex constraints. This can be achieved by defining an update policy rules for computing additions or deletions of corresponding tuples in the *value* relation. In Example 1, the pizza vendor defines the following update policy [2] for automatically unselecting the lowest protein topping when the customer selects fourth topping.

$$seltop(T) \text{ :- } topping(T),\ value(T, true)$$
$$nseltops(N) \text{ :- } countofall(T, seltop(T), N)$$
$$notmin(T1) \text{ :- } seltop(T1),\ seltop(T2),\ T1 \neq T2,$$
$$protein(T1, P1),\ protein(T2, P2),\ P1 > P2$$
$$\Delta^{-}value(F, true) \text{ :- } \delta^{+}value(T, true),\ nseltops(3),$$
$$setof(T1, (seltop(T1),\ \neg notmin(T1)), cons(F, R))$$

The first two rules define the views *seltop* and *nseltops* to represent the set and count of selected toppings respectively at any time during form-filling. In order to compute the selected topping with the least protein content, we first compute the view *notmin* to represent the set of selected toppings that do not have the least protein content. The last rule is a dynamic rule defining the change in response to the requested change of selecting fourth topping. The last atom (*setof*) of the rule builds a set of toppings that have the least protein content and returns the set as a pair (F, R) where F is an element of the set and R is the rest of the set. The head of the rule states the change that the checkbox with id F should be unchecked.

[2]Note that for the sake of better readability we use the infix notations $=$, $\neq$, $<$, and $>$ etc.

Rule Engine The Rule Engine evaluates the update policy whenever a user changes the value of a form field and computes changes to be performed in the database which included values of the form fields. In Example 1, selecting or unselecting a topping are the only actions a user can perform on the form. Therefore, the update policy is evaluated whenever a topping is selected or unselected. In Example 1, in the state with three selected toppings, the Rule Engine computes $\Delta^{-}value(T, true)$ where T is the unselected topping for the above presented example update policy. Then the Data Manager applies the computed changes to the form database.

Data to UI Mappings Form fields can be enabled or disabled by defining a rule with head *disabled*. In Example 1, suppose, instead of unselecting the checkbox corresponding to the topping with the lowest price when a user selects the fourth topping, the pizza vendor prefers to disable the fourth checkbox as soon as a user has selected three checkboxes. This can be achieved with the following rule (where definitions of *seltop* and *nseltop* are the same as above):

$$disabled(T, true) \text{ :- } nseltops(3),\ topping(T),\ \neg seltop(T)$$

Similarly, instructions and warnings can be dynamically generated by defining a rule with head *innerhtml*. Suppose, in Example 1, the pizza vendor prefers to also tell the user the reason for disabling the fourth checkbox. This can be achieved by the following rule where msg is the id of an HTML paragraph element:

$$innerhtml(msg, \textit{"max. 3 toppings allowed"}) \text{ :- } nseltops(3)$$

In addition to $disabled$ and $innerhtml$, Smart Forms also support the relations $display$ (for showing or hiding elements), $options$ (for setting the options of select elements), $color$ and $bgcolor$ (for setting the color and background color of elements), and $onclick$ and $cursor$ (for setting the onclick function and cursor for elements).

The Rule Engine then computes on the new database state the views corresponding to HTML attributes defined with the Data-to-UI-Mappings rules. The new computed HTML attributes are then applied to form UI. In Example 1, the Rule Engine computes $disabled(T, true)$ where T is the unselected topping and $innerhtml(msg, \textit{"max. 3 toppings allowed"})$. Applying these computed attributes to the form UI disables the fourth unselected checkbox and display the text "already selected 3 toppings" respectively.

Processing form data after Form Submission

In this section, we describe how form data is processed on the server once a Smart From has been submitted. There are mainly two tasks involved in the server-side processing of form data. First task involves re-validating the submitted form data and the second tasks involves storing the data, computing any results and performing steps to proceed further with the workflow.

Feedback generation and automatic filling of fields during form-filling can be done at the client-side or at the server-side. In either case, it is a good practice to re-check the validity of the submitted form data at server-side. The main

reasons for this are that the live feedback generation and the update policy for automatically filling of fields may be incomplete wrt. the constraints of the server-side database and that the client-side code can be potentially manipulated by the user.

The constraints of the server-side database can be modeled as views. In Example 1, the form data is the relation *seltop*. The constraint that at most three toppings may be selected can be modeled as

$$illegal \text{ :- } \textit{countofall}(T, \textit{seltop}(T), N),\ N > 3$$

If the server can compute *illegal*, then the server can reject the form data and send back to the client $\{illegal\}$. The client can then display an error message to the user. In general, there may be multiple constraints. In order to be able to show more meaningful error messages, the client needs to able to determine the violating constraints. This can be done either by using a different view for each constraint or using one view e.g. *illegal* with an argument to contain the error message as shown in the following rule.

$$illegal(\text{"At most 3 toppings may be selected"}) \text{ :-}$$
$$\textit{countofall}(T, \textit{seltop}(T), N),\ N > 3$$

If the submitted form data is valid, then the server can store the form data in the servers-side database and redirect client's browser to another page, possibly depending on the submitted form data. In Example 1, suppose the client's browser should be redirected to *url1* if the user has not selected any toppings and to *url2* otherwise. This behavior can be easily modeled with the following rules:

$$\textit{redirect(url1)} \text{ :- } \textit{countofall}(T, \textit{seltop}(T), 0)$$
$$\textit{redirect(url2)} \text{ :- } \textit{countofall}(T, \textit{seltop}(T), N),\ N > 0$$

The above rules demonstrate how a single user workflow can be declaratively defined and executed by the server. In general, a Smart Form may be part of a multi-user workflow and the server may need to consider data from multiple databases for form data. As a result, form data validation as well as determining which users should be shown which forms at what time often requires evaluation of complex queries of multiple databases. Since the databases may heterogeneous at the conceptual level, data integration techniques are required.

Our rule-based approach for defining the server functionality can seamlessly incorporate flexible data integration techniques such as LAV rules (Genesereth 2010) because of their common language semantics.

Creating Smart Forms

A smart form can be created as follows. The first step consists of creating a Web document e.g. HTML page. Then, the Smart Form library http://forms.stanford.edu/lib/smartform.js is added to the Web document. In order to model the constraints of the smart form, identifers are assigned to the relevant DOM elements. In addition, two new DOM elements, which are referenced using the identifiers *lambda* and *library*, are added to the Web document.

The contents of *lambda* and *library* characterize the underlying logic program. The DOM element *lambda* consists of the form data that is managed by the Data Manager. The DOM element *library* consists of view definitions and update policies that are evaluated by the Rule Engine. A subset of these views are used to update the smart form's UI. For example, the foreground color of the DOM elements is updated by evaluating rules of the form $\textit{color}(X, Y)$. A detailed description of the views that are used to updated a smart form's UI can be found in (Blundell and Mohapatra 2016).

Alternatively, a smart form can also be created using the Smart Form Editor. The Smart Form Editor consists of two components: a *rule editor*, and a *UI designer*.

Rule Editor. The rule editor serves as an efficient interface for authoring the smart form's rules and update policies. The rule editor supports the following features.

- Ability to plug-in or import, and export rules from local files or URLs.
- Ability to connect to, and query external structured data sources using the technique proposed in our previous work (Agarwal et al. 2015).
- Syntax highlighting, safety, and stratification checks to assist in the authoring of admissible rules.
- Smart auto-completion to make it convenient for a rule author to compose rules. When auto-completing a relation, the author is supplied with a list of options which include the reserved relations e.g. *value*, *view*, *color*, the update operators e.g. *pos*, *minus*, and other relations that have been defined by the author. A relation, say r, with arity k is auto-competed as the atom $r(X_1, X_2, \ldots, X_k)$. When auto-completing a term, the author is supplied with a list of options which includes the identifiers of the DOM elements, and other terms that have been previously defined by the author.
- Rule macros to automate the authoring of common, repetitive rules or rule patterns.
- Assisted authoring of update policies to ensure that the authored update policies do not violate the constraints that the form creators intend to enforce. To ensure that the authored policies are *sound*, our resolution-based algorithm Verify-Policy (Mohapatra, Agarwal, and Genesereth 2016) may be used. A complementary approach to the a-posteriori verification is to allow form creators to interactively author update policies from the constraints. Such an update policy authoring process can be facilitated by automatically generating all necessary and sufficient update policies (Orman 2001), and letting the smart form creator choose one policy, or combine multiple policies.
- Profiling tools to help optimize the rule base. These tools provide the support for defining breakpoints at different literals in a rule, fine-grained statistics about rule evaluation e.g. number of inferences, number of inferences or time taken to derive a supplied atom etc., the ability to trace the derivation of supplied atoms and to profile a subset of rules over a supplied workload (of facts).

- Rule organization (by predicates) and formatting to improve the readability of the authored rules.

UI Designer. The UI designer supports the creation of the smart form's UI in a WYSIWYG fashion using customizable widgets. The UI designer also allows the smart form's DOM to be declaratively specified through Data to UI mappings, allows new elements to be added or existing elements to be deleted from the DOM.

We have currently implemented an initial version of the Smart Form Editor. This implementation supports creation of smart forms with static DOMs, and is accessible at http://forms.stanford.edu/editor/. The rule editor in our implemented version does not currently support profiling tools, or macros.

Case Studies

Smart Forms have been tested and evaluated for many use cases. For some use case, Smart Forms have been deployed as part of the production system and used by hundreds of people regularly. In this section, we present two case studies that demonstrate the utility of Smart Forms for enforcing regulations within a University (in this case Stanford University) and enforcing a law at a district level respectively. The corresponding Smart Forms can be found at http://forms.stanford.edu.

Smart Forms in Stanford Engineering School

At Stanford University, students enrolled in the Master of Science in Computer Science (MSCS) Program are required to submit a MSCS program sheet by the end of their first quarter, and have it approved by their advisor and the MSCS program administrator. A MSCS program sheet is a plan that details the courses that the student is to take before graduation. Currently, there are 100 programsheets in the Stanford Computer Science Department 10 single depth and 90 dual-depth depth, all of which are implemented as smart forms.

Prior to the introduction of smart forms, all program sheets were paper forms, and the submitted forms were manually validated to check whether or not a student's plan satisfied the program requirements. Due to a large number of program requirements, the validation process was tedious, and error prone. This was remedied by capturing the program requirements formally, and implementing the program sheets as smart forms. We present an overview of a program sheet's UI, and the logical rules that capture the requirements for MSCS programs.

User Interface: In a program sheet's UI, courses are represented as checkboxes as shown in Figure 2. The course number is used as as identifier of the corresponding checkbox. Program requirements are represented as span elements. Violation of program requirements are indicated by coloring the corresponding span element red. In Figure 2, if no checkboxes are selected, then the text "Probability" is colored red to indicate a violation.

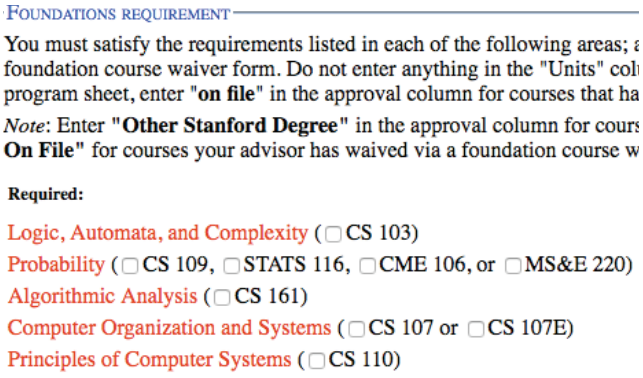

Figure 2: Program sheet User Interface

Program Requirements: In order to capture the program requirements, the following facts are included in the program sheet.

- Facts of the form $req(X)$ to indicate that X is a requirement e.g. $req(foundation\_req)$, $req(breadth\_req)$ etc.
- Facts of the form $foundation(X)$, $breadth(X)$, and $depth(X)$ to indicate whether X is a foundation course, breadth course, or a depth course respectively, e.g. $foundation(cs103)$, $breadth(cs140)$, $breadth(cs265)$.

In the following, we present examples of requirements that are formalized in the program sheet.

- The breadth requirement is satisfied by taking at least 3 breadth courses. The program sheet contains facts of the form $breadth(X)$, where X is the course number of breadth course e.g. *cs140*, *cs144*, *cs265* etc. The satisfiability of the breadth requirement is encoded using the following rules.

$$satisfied(breadth\_req) \text{ :- } countofall(X, breadth(X), N), \\ min(C, 3, 3)$$
$$breadthsel(X) \text{ :- } breadth(X),\ value(X, true)$$

  In the above rules, $breadth\_req$ is the identifier of the span element that corresponds to the breadth requirement "Take at least three Breadth courses".

- Violation of a program requirement is indicated by coloring the corresponding span element red. Otherwise, the program requirement is colored black. This behavior is encoded using the following rules.

$$color(X, red) \text{ :- } req(X),\ \neg satisfied(X)$$
$$color(X, black) \text{ :- } req(X),\ satisfied(X)$$

- Exactly one of *cs109*, *stats116*, *cme106*, or *mse220* must be selected to satisfy the probability requirement. This

behavior is encoded as follows.

$prob(cs109)$
$prob(stats116)$
$textitprob(cme106)$
$prob(mse220)$
$neg(value(Y, true))$:- $prob(X)$, $prob(Y)$,
$pluss(value(X, true))$, $value(Y, true)$, $X \neq Y$
$satisfied(prob\_req)$:- $prob(X)$, $value(X, true)$

The above update rule ensures that the selection of the checkboxes corresponding to the probability course is *mutually exclusive*. Therefore, at most one probability course may be selected at any time.

In addition to the MSCS program sheets, the Undergraduate program sheets http://logic.stanford.edu/ugps, room reservations https://gin.stanford.edu/showschedule.php in the Computer Science Department, and interactive LTI exercises in the *Introduction to Logic* MOOC https://www.coursera.org/learn/logic-introduction/ are also implemented as smart forms.

FERPA-compliant Agreement Drafting

This Smart Form assists school districts and parties interested in acquiring data about school children in drafting a FERPA [3]-compliant agreement on the types and usage of shared data.

The Smart Form does so by enabling interactive formation of an agreement between an information service provider and a district as well as analysis of multiple agreements. In the Smart Form, a contacting party can fill in the details such as which student's data is to be shared, the potential use of the data by the provider, age/grades level of the students etc. The Smart Form then checks the validity of the agreement as per FERPA, COPPA and SOPIPA and displays the violations and obligations if any.

Below we present an excerpt of the database and the rules that we have modeled for this case study. The complete set of rules is visible in the 'Law' tab of the prototype.

Data and Views The following view definition defines that student's name is a personally identifiable information (PII). Other PIIs such as student's date and place of birth, student's SSN, student's mother's maiden name etc., are modeled similarly. Categories *district_data_non_pii*, *additional_data_pii* and *additional_data_non_pii* are defined analogous to the definition of the category *district_data_pii*.

$$district\_data\_pii(D, district\_student\_name) \text{ :-}$$
$$district\_data(D, district\_student\_name)$$

The following view definition states a provider is under direct control of district if the provider can amend terms with consent. Other views can be modeled analogously.

$$provider\_under\_direct\_control\_of\_district(D) \text{ :-}$$
$$provider\_can\_amend\_terms\_with\_consent(D)$$

[3]Family Educational Rights and Privacy Act, Children's Online Privacy Protection Act

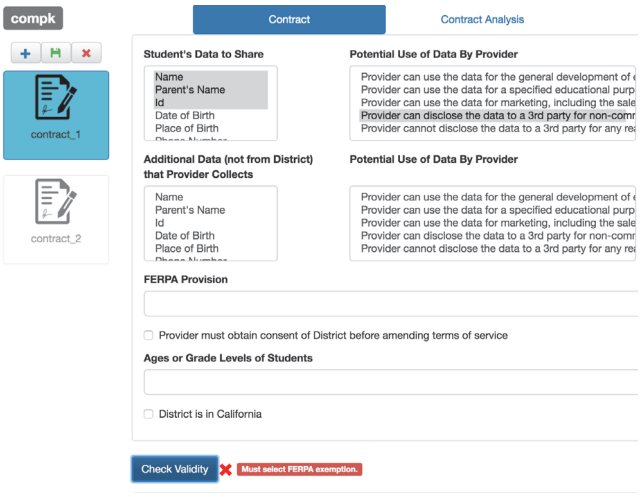

Figure 3: Smart Form for FERPA compliant agreement formation showing that the agreement draft is invalid as well as the reason that a FERPA exemption must be selected

Constraints The following rule states the provider must select a FERPA provision.

$illegal($*"Must select FERPA exemption."*$)$:-
$district\_data\_pii(D, A)$, $\neg ferpa\_provision(D, dir\_exempt)$,
$\neg ferpa\_provision(D, actual\_par\_consent)$,
$\neg ferpa\_provision(D, school\_off\_exempt)$

The following rule states that commercial use of data is prohibited under school official exemption.

$illegal($*"Under School Official Exemption,*
commercial use of data is prohibited."$)$:-
$ferpa\_provision(D, school\_off\_exempt)$,
$district\_potential\_use\_by\_provider(D, district\_4aiii)$

Based on the validity rules, the Smart Form can automatically check whether a given agreement draft is valid according to FERPA by evaluating the relation *illegal* (see also Figure 3).

Rights and Obligations Below the modeling of the consequence that if FERPA provision is directory exemption, then the district must allow opportunity for parent to opt-out of the disclosure of student's data.

$consequence(D,$ *"District must allow opportunity for*
parents to opt-out of the disclosure of student data."$)$:-
$district\_data\_pii(D, A)$,
$ferpa\_provision(D, directory\_exemption)$

In addition to computing whether a contract is valid or not, the Smart Form can also automatically output the reason for the invalidity, any rights, limitations and obligations that parties have if the contract is valid. For example, if the chosen exemption is "Actual Parental Consent", then our Smart Form can automatically compute that the district may disclose the data only according to the terms of the parental consent.

Hypothetical Analysis Our Smart Form also supports hypothetical reasoning over a set of (valid) contracts. Typically, an information service provider enters into multiple contracts, one for each district, to be able to achieve broader coverage for his/her service. Given a set of such contracts, an information provider is often faced with the problem of deciding whether he/she may use certain data artifact for a particular use. In order to obtain the answer to such a question with our prototype, the information service provider would formulate his question as a query in the 'Contract Analysis' tab. Our prototype then analyses all the existing contracts of the information service provider and produces the answer to the question. For example, if an information service provider wishes to know which data he/she may share with a third party, he/she would pose the query *provider_may_share(District,Data,3rd_party,Use)*. The answer to this query will contain all (district, data artifact, usage) tuples that the provider may share with a 3rd party.

Related Work

The following limitations of Google Forms (Google Inc. 2016) restrict their use in many professional scenarios. (1) Google Forms' data validation constraints neither support logical connectors nor field inter-relationships. To the best of our knowledge, Google Forms do not allow form creators to extend the data validation support. (2) Google Forms API requires programming knowledge in JavaScript which is not a viable option for most professionals who are not software programmers, e.g., officials of a city administration. Furthermore, a programmatically created Google Form is not synchronized with the data that was used to generate it. Furthermore, form elements cannot be added or removed dynamically based on a user's interactions with a Google Form.

SurveyMonkey (SurveyMonkey 2016) as the name suggests specializes in creating survey forms. SurveyMonkey forms typically consist of a list of multiple choice questions. SurveyMonkey forms can be created only manually and do not support further constraints on field values. Typeform (Typeform 2016) is similar to Surver Monkey except that Typeform focuses more on attractive design by providing form templates and ease of use by allowing only one question per form. Typeform, Gravity Forms (Gravity Forms 2016b), NinjaForms (Ninja Forms 2016), Formstack (Formstack 2016) and a few more use Conditional Logic for showing/hiding a field or an entire section based on user's input in another field. Conditional Logic allows you configuration of forms to show or hide fields, sections, forms or even the submit button based on user selections (Gravity Forms 2016a). Conditional Logic allows showing or hiding form fields, form sections, and complete form as well as enable or disable a form's submit button based on a previous user selection. Conditional Logic can support only simple conditions involving only field values since it does not support variables and data other than field values.

Conclusion and Outlook

Forms remain essential to the delivery of a wide range of government services. To gather information and implement government policy, government departments and agencies issue and receive back millions of paper forms per year. Efficient issuing and processing forms can have significant implications in shaping both the cost efficiency of departments and agencies, and how citizens perceive the public services.

We have presented Smart Forms, an innovative web forms technology for easy creation, maintenance and evaluation of user-friendly web forms especially the ones that must implement complex laws, regulations or business policies. Smart Forms have built-in mechanisms for providing context-sensitive feedback and automatically filling fields to assist end users in form-filling. These mechanisms can be configured by Smart Form creators without writing a single line of traditional procedural programming code.

Smart Forms is a core technology toward realizing the vision of a Citizens' Dashboard, a one stop service where citizens and organizations can easily find, fill and submit relevant government forms as well as manage, analyze and efficiently use their data.

References

Agarwal, S.; Mohapatra, A.; Genesereth, M. R.; and Boley, H. 2015. Rule-based exploration of structured data in the browser. In *Proceedings of 9th International Symposium (RuleML 2015)*, 161–175. Springer.

Agarwal, S.; Mohapatra, A.; and Genesereth, M. 2016. Jabberwocky. http://logic.stanford.edu/~sudhir/jabberwocky.pdf. Tool Demonstration at 3rd Symposium on Computation + Journalism, Stanford University, Stanford, USA.

Blundell, H., and Mohapatra, A. 2016. Smart Form User Manual. http://forms.stanford.edu/editor/help.html. Accessed: 02-09-2016.

Flanagan, D. 2011. *JavaScript: The Definitive Guide*. O'Reilly & Associates, 6th edition.

Formstack. 2016. Formstack. http://www.formstack.com. Accessed: 18-08-2016.

Genesereth, M. R. 2010. *Data Integration: The Relational Logic Approach*. Synthesis Lectures on Artificial Intelligence and Machine Learning. Morgan & Claypool Publishers.

Genesereth, M. R. 2015. A brief introduction to deductive databases. http://logic.stanford.edu/jarvis/complaw/ddb.html. Accessed: 03-09-2016.

Google Inc. 2016. Google Forms. http://www.google.com/forms/. Accessed: 18-08-2016.

Gravity Forms. 2016a. Conditional Logic. http://www.gravityforms.com/features/conditional-logic/. Accessed: 18-08-2016.

Gravity Forms. 2016b. Gravity Forms. http://www.gravityforms.com. Accessed: 18-08-2016.

Kassoff, M. 2011. *Logical Spreadsheets*. Ph.D. Dissertation, Computer Science Dept., Stanford University.

Kowalski, R. A. 1988. The early years of logic programming. *Commun. ACM* 31(1):38–43.

Mohapatra, A.; Agarwal, S.; and Genesereth, M. 2016. Update policies. Technical report. Submitted to 29th Australasian Joint Conference on Artificial Intelligence (AI 2016), Hobart, Australia. http://logic.stanford.edu/~sudhir/update-policies.pdf.

Ninja Forms. 2016. Ninja Forms. http://www.ninjaforms.com. Accessed: 18-08-2016.

Orman, L. V. 2001. Transaction repair for integrity enforcement. *IEEE Trans. Knowl. Data Eng.* 13(6):996–1009.

SurveyMonkey. 2016. SurveyMonkey. https://www.surveymonkey.com. Accessed: 18-08-2016.

Typeform. 2016. Typeform. http://www.typeform.com/. Accessed: 18-08-2016.