Peer-to-Peer Form Based Web Information Systems

Stijn Dekeyser[1] **Jan Hidders**[2] **Richard Watson**[1] **Ron Addie**[1]

[1] **University of Southern Queensland, Australia**
[2]**University of Antwerp, Belgium**

Abstract

The World Wide Web revolutionized the use of forms in everyday private and business life by allowing a move away from paper forms to easily accessible digital forms. Data captured using such HTML forms could be processed using relational databases or other applications that enforce and apply business logic. Lately XForms has been introduced, offering a logical evolution of digital data capture and dissemination using Internet and document technology.

This paper introduces two important new ideas. The first one is the main focus of the paper: a novel type of peer-to-peer web information system where forms are first-class citizens containing extended access rules of very fine granularity which govern read and update rights to data objects associated to the forms. The second idea, which we explore in a preliminary section, forms a powerful motivation for the use of such systems: the automatic and dynamic derivation of workflow processes from the access rules contained in forms.

As such, the proposed system leverages current forms and Internet technology to liberate the creation and use of forms and reports, facilitating the capture and dissemination of data, while allowing dynamic management of work flows within organizations.

1 Introduction

The background for the theory developed in this paper is a real-life problem. The Department of Mathematics & Computing at USQ would like its staff to easily capture data from colleagues and students through web-based forms, efficiently store that data, and re-use the data captured by others, without compromising security and access rights, and without staff having to script their own web pages with database functionality. In addition, while giving everyone the opportunity to create complex interactive data-driven applications, the system which allows this must be able to communicate with other such systems on different peers, and show end-users the various actions which make up the workflow represented by a form. The workflow is not defined a priori; instead, addition of new forms may add or alter individual steps.

Let us consider a brief example. Suppose the secretary to the Dean creates a Leave Application form to be filled in by staff members prior to going on leave. The Dean must approve the application, after which it is sent to human resources and cannot be altered anymore. Prior to the Dean's approval, the applicant may change the dates of the application, but he cannot do so after approval is given. Months later, it is decided that the Head of Department (HoD) must give a separate approval, prior to the Dean's. A simple change of the form's definition must add this new information and alter the original access rules, which changes the workflow graph.

Another example in which data is re-used by different peers is given by the Publications Form. Suppose staff member John makes a form available to all staff in which they can enter and alter details of publications of which they are an author, and let others use parts of the entered information. Now suppose another staff member, Jill, wants to extend this form. She might create a form definition that re-uses John's data objects, but extends them with her own. End-users may enter data in either form, after which the basic publication data as defined in John's form is available to everyone, as long as the access rights are satisfied.

These two scenarios constitute a basis for proposing a new paradigm for web based information systems in which forms are first-class citizens representing complex, distributed instances and in which workflows are dynamically built up from the access rules present in form definitions.

Motivation. To the best of our knowledge, a system that allows all these functions does not yet exist. Currently, parts of the problem can be solved using various techniques and tools. For example, capturing data can be done by a distributed database, where users create their own tables and re-use information by using views[1] defined by others. Electronic forms can be generated using HTML and special purpose scripts, or the recent XForms [14] recommendation may be used depending on available implementations. In the latter case, access rights to data elements, and a concept of workflow, still need to be coded separately. Finally, commercial workflow systems (e.g. [7, 22]) require a complex design phase and implementation performed usually by specialists outside of the organization, after which adaptations in the business actions require a new cycle of design and implementation.

Hence, the two main motivations for this research are as follows. Firstly, we want to ultimately create truly enabling software that allows individuals a fairly easy way to create electronic forms, capture data with them that will be stored efficiently, and generate reports of data captured by their own forms but also those of other, distributed, users so long as this is allowed by access rules.

Secondly, as forms defined in the system include

[1]This then raises the problem of updating relational views.

Copyright ©2006, Australian Computer Society, Inc. This paper appeared at The Seventeenth Australasian Database Conference (ADC2006), Hobart, Tasmania. Conferences in Research and Practice in Information Technology, Vol. 17. Gillian Dobbie, Ed. Reproduction for academic, not-for profit purposes permitted provided this text is included.

access rights, a workflow process is automatically associated with them. There is no need for a complex design phase for constructing a workflow, and any updates to the access rules (or new ones in a new form) incrementally modify the associated workflow. In addition to this significant benefit, we want to give creators of forms an easy method to check if the desired end states of the form can indeed be reached, and want to inform users of a form how the data is used (if this is allowed by the creator). These are properties that can be derived once the graph representing the workflow is constructed.

Contribution. The twin motivations listed above naturally translate in two research goals. The main contribution of this paper, however, lies with the formalization of the form-based peer-to-peer web information system. Specifically we formalize schemas for forms, and define the access rules language.

A secondary contribution lies in the exploratory fifth section by describing research questions associated to the derivation of workflow processes implied by access rules in forms.

Organization. This paper is organized as follows. In Section 2 we discuss both technical and theoretical work related to ours. In Section 3 we present the formal model for form-based peer-to-peer web information systems. We present the access rule language in Section 4 which will also be used to infer workflows, as explored in Section 5. Finally, in Section 6 we give a brief conclusion, discuss implementations, and outline the next steps in our ongoing research of deciding workflow processes in electronic form systems.

2 Related Work

The examples briefly described in the introduction point to both a variety of tools and systems that can be used to implement such a system, and also a variety of fundamental topics and concepts that are being drawn upon. In the first category, we may list tools such as XForms and server-side scripting languages (e.g. PHP) which facilitate communication with database servers (e.g. PostgreSQL). On the theory side, clearly all of the following are relevant: workflow theory, data and schema integration, distributed databases, views, peer-to-peer information systems, and security.

2.1 XForms

Many practical problems associated with electronic forms as implemented by HTML have recently been solved by the introduction of the XForms recommendation by the World Wide Web consortium [14]. In our work, we use XForms as but one of the tools to solve our technical problem; indeed, we use its strengths such as its Model-View-Controller design pattern, its client-side validation, construction of XML output, and so on. Importantly, however, we add many desirable fundamental features, thus suggesting new avenues of study in the context of future versions of XForms.

Looking at some important practical differences between our form-based information system and XForms, we list the following three issues:

- *Database connection.* In XForms, it is possible to read data from and write data to a database. But the tables must already exist, requiring form designers to know the schema and provide the correct SQL expressions.
  In contrast, in our system form designers need only to focus on creating the schema of their form, possibly re-using other form schemas; read and write access to and from the database, as well as preceding data-definition statements, are handled automatically.
- *Access rules.* In XForms, there are no rules to regulate access to data stored in the XML instance or a database. It is assumed that all data in the XML source is accessible, or that the database handles access rights. In the latter case, the form designer may not have full control over these constraints.
  In our proposed system, access rules are an explicit part of the form's definition, under control of the form's designer, and enforced by the forms server.
- *Workflow modelling.* There is also no notion of a workflow process attached to XForms; fields may be entered in a random order, although some values are calculated from others, and constraints may reference other fields.
  The access rules we require in a form's definition implicitly impose an order in which fields may be assigned values. Hence it is possible to infer a workflow process corresponding to a form.

Turning to some more theoretical issues w.r.t. XForms, we note that the recommendation is very complex owing to the fact that users have a very expressive language in which to describe forms. Not only does XForms use the full power of XML Schema's type system, it also introduces a rich constraint language. This expressivity precludes finding decidable problems such as completion. In our ongoing work, we will take a different approach, limiting expressiveness to allow the study of decidability problems.

2.2 Workflow Processes

The secondary aim of this paper is to investigate under which conditions (in the form of a data model and access rules language) it is possible to automatically construct a workflow graph corresponding to a form definition.

Research in the area of workflow modelling [3] has been active since the late eighties and has led to the commercial development of various Workflow Management Systems (WfMS) [18]. The main perspectives have traditionally been (1) *control flow* (or process), (2) *resource* (or organization), (3) *data* (or information), (4) *task* (or function), and (5) *operation* (or application) perspectives [2]. Often the aim has been to extend modelling concepts to better capture various subtle details of these perspectives. Dynamic derivation of workflow processes has not yet received attention, and constitutes a very significant motivation for using form-based information systems, which are the main contribution of this paper. The most relevant perspective relating to workflow research in our setting is the data perspective, as electronic forms record data and do this progressively on availability of other data previously entered.

In contrast, in Workflow Patterns [5] control flow (constraints on order of processing, synchronization, etc) is more important than data-flow. In our case, we focus on the flow of data and the operations performed on them; control-flow more or less implicitly follows from the data-flow.

Hence, a workflow *case* in the context of this paper is an instance over a certain form's schema, an *action* corresponds to the entry of data in a part of a form, and the workflow *process* is the sequence of actions that can be executed to arrive at a correctly

completed form as defined by the access rules over the data provided by the form's designer.

Workflow Mining. Another area in workflow research recently has been the mining of workflow processes from diverse information sources such as transaction and event logs [4, 8]. In this case, as in our work, workflow processes are not modelled ahead of time by experts. However, the focus is significantly different from ours, and the two methods are completely independent.

Finally, in our own previous work [16, 17], we have discussed formal methods to decide when two workflow processes are the same, and have also presented non-destructive methods to integrate form-based views in workflow systems.

2.3 Databases, Modelling, Integration, Distribution, and Views

The largest area of research relevant to this paper is most obviously that of databases. Several topics are especially relevant. Firstly, our form definitions contain a schema strongly based on *entity-relationship modelling*. Instances over the schemas correspond in fact to *nested relations*, a concept widely studied in the seventies of the previous century. We represent the instances as trees and will normally serialize them as *XML documents*, another popular, if much more recent, database research area. Likewise, the schemas will be expressed in a language based on XML Schema. In addition, our access rules language is based on a subset of XPath corresponding to first order logic restricted to two variables (FO^2).

Distributed Databases and Peer-to-Peer Information Systems. A clear design decision for our forms system has been to use the powerful notion of peer-to-peer information systems. Rather than describing one centralized forms server, we assume groups of users each have their own server which communicates with peers to access forms, obtain data stored at other locations, and add to that data. This aspect opens up many interesting topics already studied in the context of distributed databases. We shall assume solutions from that field rather than re-invent the wheel. However, two related issues deserve some additional attention: views and data & schema integration.

Data and Schema Integration. Both within a single peer as between various peers, it is possible and desirable to reuse schemas introduced by different forms. Schema integration is a very complex and widely studied topic in the context of databases systems [10, 11, 20, 21]. In our case, some of the complexities are irrelevant, while other results are very much applicable. However, the main difference, from a data integration perspective, is that we don't limit integration to read-only data. Indeed, we must allow users to read information from various peers representing an integrated schema, but we must also enable them to update this data. This further complicates integration considerably, however, the description of our form-based information system will adequately support a simple and efficient procedure for reading and writing data from various sources.

Views. Our data model stores a collection of logically related data structured in accordance to the schemas present in forms. In a real sense, each form schema plus its access rules acts as a view on the underlying data model. The `isa` constraints that we define in Section 3 are able to *project* parts from different entities. The *join* operation is present in the modelling of relations in the schema. And some form of *selection* is done when applying access rules. Hence, creators of forms have an implicit view language at their disposal, and the problem of view updatability and maintenance appears.

The data model can be implemented in various types of database systems; usually, a relational DBMS will be used because our nested relations-like instances can be translated easily to this model. Alternatively, native or XML-enabled databases can also be used. When a relational system is chosen, results from research into relational views can be used [12]. Research into updating XML views is currently underway [19].

2.4 Security and Authentication

Forms systems collect, store, retrieve and disseminate possibly sensitive information. It is essential that such systems provide secure access to data when required. The area of security has been and continues to be thoroughly studied. Formal languages for expressing security relationships have also been defined [15, 24]. In our work, we take a rather database-oriented approach to security; in Section 4 we describe a rule language that enables a forms creator to define precisely who can create, read and delete data corresponding to forms. These rules will be part of the data dictionary of the forms system.

A reliable authentication process together with data encryption, typically based on public key cryptography and digital certificates, is needed in addition to the access rules to build a truly secure system. It is assumed that these authentication and encryption procedures are available as a service to the form-based information system, and so is outside the scope of the current work.

Authentication on single server systems is a straightforward operation. Extending authentication seamlessly to systems where data is shared across multiple peers is not so simple. We certainly wish to avoid the situation were a user is required to identify herself to every peer that is involved in a forms transaction. Instead, authentication should be carried out at the primary forms server, and inter-peer negotiation should propagate appropriate access rights to data on other servers. This problem as yet requires additional research.

3 FormWIS Data Model

We proceed with describing what a form-based web information system (FormWIS) is, and define the data model that it uses.

Form-based Web Information System. A FormWIS is a cooperative information system that presents all data to the user in electronic web forms. As such it offers a view on the underlying data model that can be updated through data entry in the form. Users can perform all manipulations of the presented data if this is allowed by the access rules that are part of the definition of a form[2].

Users can add new forms that may or may not share information with previously defined forms. The underlying data model is then automatically extended with the extra information in this new form definition. The access rules associated to the new data are determined by the user who defined the form.

[2]This implies that users may have to identify themselves by supplying, for example, a password. We assume such an authentication mechanism is present.

A FormWIS will cooperate with peers over a network, making data sharing between disparate organizations possible. What is special and desirable about FormWISs, is that they allow a more natural evolution of data capturing and liberal reuse of information sources both within an organization and with third parties, while maintaining strict rules about who has which type of access to which data. In addition, as will be discussed in Section 5, they allow for automatic and dynamic modelling of a workflow process.

3.1 Formal Definitions

To foster a clear understanding of the FormWIS data model, we will first define schemas and instances over schemas, before turning to two different semantics of instances.

3.1.1 Schemas and Instances

A FormWIS stores definitions of forms and instances that belong to a form. We will defer the formal definition of a *form* to Section 4, but one important part of a form is its schema. In turn, a schema consists of several parts. We first define a frame.

Definition 1 (Frame) *A* frame *is a tuple $F = (C, R, s, t)$ where*

- *C is a set of class names partitioned into C^e, the entity classes, and C^v, the value classes*
- *R a set of relation names*
- *$s : R \rightarrow C^e$ and $t : R \rightarrow C$ giving the source and target classes of a relation.*

A frame is said to be a forest frame *if the corresponding multi-graph is a forest.*

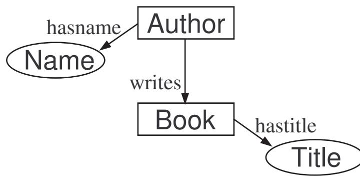

Figure 1: An example of a frame.

Example 1 *Consider the following frame F which will be used in the context of a form capturing book publication details for authors. $F = (C, R, t, s)$ where $C = C^e \cup C^v$ with $C^e = \{Author, Book\}$ and $C^v = \{Name, Title\}$. $R = \{writes, hasname, hastitle\}$, and $s(writes) = \{Author: dekeyser\} = s(hasname)$. Furthermore, $s(hastitle) = \{Book\} = t(writes)$, $t(hasname) = \{Name: \}$, and $t(hastitle) = \{Title\}$. A graphical representation of this frame is shown in Figure 1.*

One of the central ideas in our work is that we argue that forms are usually hierarchical in nature. It therefore seems natural to use an hierarchical data model like nested relations or XML. However, we will want to model *many-to-many* relations such as the relationship between the classes *Book* and *Author*, which is more easily done in a graph-based model such as the Entity-Relationship Model (ER Model) or the Object Exchange Model (OEM).

Definition 2 (Instance of frame) *An* instance *of a frame $F = (C, R, s, t)$ is a tuple (O, I_C, I_R) where*

- *O a set of objects partitioned into O^e, entities, and O^v, values*
- *$I_C : C \rightarrow 2^O$ the class interpretation function such that $I_C(c) \subseteq O^e$ if $c \in C^e$ and $I_C(c') \subseteq O^v$ if $c' \in C^v$*
- *$I_R : R \rightarrow 2^{O \times O}$ the relation interpretation function such that for all $(o_1, o_2) \in I_R(r)$ it holds that $o_1 \in I_C(s(r))$ and $o_2 \in I_C(t(r))$.*

Example 2 *Consider the following instance I of frame F presented in Example 1: $I = (O, I_C, I_R)$ where $O = O^e \cup O^v$ with $O^e = \{a_1, a_2, a_3, b_1, b_2\}$ and $O^v = \{knuth, date, widom, programming, databases\}$. Furthermore, $I_C(Author: dekeyser\ ) = \{a_1, a_2, a_3\}$ and $I_C(Book) = \{b_1, b_2\}$. Finally, $I_R(writes) = \{(a_1, b_1), (a_2, b_1), (a_2, b_2), (a_3, b_2)\}$, $I_R(hasname) = \{(a_1, knuth), (a_2, date), (a_3, widom)\}$, and $I_R(hastitle) = \{(b_1, programming), (b_2, databases)\}$.*

An instance of a frame can also be thought of as a labelled graph where the nodes are labelled with sets of classes (meaning that a node is an object in each of those classes) and edges are labelled with the name of a relationship.

Definition 3 (Instance graph) *The* graph of an instance *(O, I_C, I_R) of a frame (C, R, s, t) is the tuple (O, E, λ) where*

- *O is the set of nodes,*
- *$E = \{(o_1, r, o_2) | (o_1, o_2) \in I_R(r)\}$ is the set of labelled edges*
- *$\lambda : O \rightarrow 2^C$ the node labelling function such that $\lambda(o) = \{c \in C | o \in I_C(c)\}$.*

Example 3 *The instance formulated in Example 2 can be presented as the graph shown in Figure 2.*

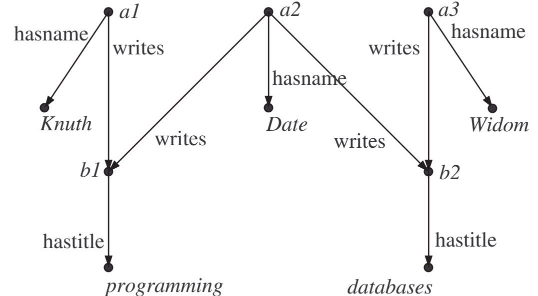

Figure 2: An instance graph for Frame F presented in Example 1, with the λ labelling function omitted for clarity.

We are now ready to define the schema of a form.

Definition 4 (Schema) *A* schema *is a tuple $S = (F, K)$ where F is a forest frame and K a set of* constraints *over F. These can be any type of constraint but we will assume here that they are of the following forms: $c_1\ \mathsf{isa}\ c_2$, $r_1\ \mathsf{isa}^{\downarrow}\ r_2$ and $r_1\ \mathsf{isa}^{\uparrow}\ r_2$ with $c_1, c_2 \in C$ and $r_1, r_2 \in R$.*

This definition allows us to create one conceptual schema from a variety of schemas each belonging to different forms. Thus, a form's schema is a forest frame consisting of one tree plus isa constraints. These constraints can refer to classes or relationships

within the form's frame, but also to other forms' frames. It is only when taking the schemas of all forms together that one, purely conceptual, schema emerges; this *super schema* is a forest of trees with isa 'vines' between them. An example is given in Figure 3.

Note that it is the inclusion of the isa constraints in our theoretical data model that allows implementations to become peer-to-peer, hence providing the functionality alluded to in the title of this paper. Indeed, in Figure 3 the separate schemas that make up the conceptual super schema may be present on different peers.

3.1.2 Graph and Tree-based Semantics

Whereas a form's schema is a tree, a corresponding instance is actually a graph. Clearly the *writes* relation used in the previous examples is many-to-many, meaning that an author has written several books and a book may be written by several authors. Thus, the corresponding instance is a graph, as shown in Figure 2.

However, we would like instances to be trees, for a variety of reasons. First, because a form's schema is hierarchical, we would like to render its instances as trees on the users' screens. Exactly how the rendering should indicate that two objects shown is actually one and the same object is a GUI issue that we don't discuss here. An instance being a tree also allows data to be serialized as XML documents (perhaps using attribute references), which can then be further manipulated using languages such as XSLT and XQuery. The main reason, however, will become clear in Section 5: without hierarchical instances it is not possible to describe individual states in the state diagram corresponding to the form's workflow model.

To allow users to specify many-to-many relationships in the presence of both *hierarchical* schemas *and* instances, we will require them to use isa constraints. Consider that we have a second form whose schema $S' = (F', K)$. The frame F' is similar to F: $F' = (C', R', s', t')$, $C' = \{\text{Book'}, \text{Author'}\}$, $R' = \{\text{written}\}$, $s'(\text{written}) = \{\text{Book'}\}$, and $t'(\text{written}) = \{\text{Author'}\}$. In addition, the set K has the following isa constraints: $Book'$ isa $Book$, $Author'$ isa $Author$, and $written$ $\mathsf{isa}^{\uparrow}$ $writes$. The conceptual super schema is given in Figure 3.

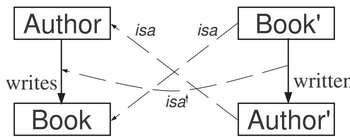

Figure 3: An example of a conceptual super schema obtained from schemas S and S'.

In the presence of such isa constraints, we must define the semantics of instances. We first turn to graph instances.

Definition 5 (Graph Instance) *A* graph instance *of a schema $S = (F, K)$ is an instance (O, I_C, I_R) of F that satisfies the constraints in K:*

1. *if c_1 isa $c_2 \in K$ then $I_C(c_1) \subseteq I_C(c_2)$*
2. *if r_1 $\mathsf{isa}^{\downarrow}$ $r_2 \in K$ then $I_R(r_1) \subseteq I_R(r_2)$*
3. *if r_1 $\mathsf{isa}^{\uparrow}$ $r_2 \in K$ then $I_R(r_1)^{-1} \subseteq I_R(r_2)$*

and the following two general constraints:

1. disjointness *if $o \in I_C(c_1) \cap I_C(c_2)$ then there is a $c_3 \in C$ such there are isa paths from c_3 to both c_1 an c_2 and $o \in I_C(c_3)$*
2. surjectivity *if $o \in I_C(t(r))$ then there is an o' such that $(o', o) \in I_R(r)$*

Note 1 *We make the following remarks about this definition:*

- *The disjointness constraint we use means that if an object belongs to two distinct classes, then these classes have a common subclass of which the object is also a member. This constraint is more liberal than requiring for two classes to be disjoint there should be no directed path of isa edges between them.*
- *There can be implicit isa constraints that are logically implied by the set K. In previous work, we have shown how they can be derived. We assume in this paper that this derivation happens automatically when a form is submitted.*
- *To have a simple notion of "location" of data we will assume that there are no cycles of isa edges and there is no multiple inheritance. Under these conditions there is always a unique highest class for an object, which might be considered as the true storage location of the data.*

Example 4 *The instance graph shown in Figure 2 is a graph instance since for the preceding examples the set of isa constraints K is empty, and the disjointness and surjectivity constraints are satisfied.*

As we will not be using graph instances, but hierarchical instances, we now turn to the tree-based semantics.

Definition 6 (Tree Instance) *A* tree instance *of a schema $S = (F, K)$ is an instance (O, I_C, I_R) of frame F plus an equivalence relation $\equiv \subseteq O \times O$ such that the graph of the instance is a forest and nodes are labelled by λ with at most one class, and moreover satisfies the constraints in K under $\equiv$:*

1. *if c_1 isa $c_2 \in K$ then $I_C^{\equiv}(c_1) \subseteq I_C^{\equiv}(c_2)$ where $I_C^{\equiv}(c) = \{[o]^{\equiv} | o \in I_C(c)\}$*
2. *if r_1 $\mathsf{isa}^{\downarrow}$ $r_2 \in K$ then $I_R^{\equiv}(r_1) \subseteq I_R^{\equiv}(r_2)$ where $I_R^{\equiv}(r) = \{([o_1]^{\equiv}, [o_2]^{\equiv}) | (o_1, o_2) \in I_R(r)\}$*
3. *if r_1 $\mathsf{isa}^{\uparrow}$ $r_2 \in K$ then $I_R^{\equiv}(r_1)^{-1} \subseteq I_R^{\equiv}(r_2)$*

and the following four general constraints:

1. disjointness *if $o_1 \in I_C(c_1)$, $o_2 \in I_C(c_2)$ and $o_1 \equiv o_2$ then there is a $c_3 \in C$ and $o_3 \in I_C(c_3)$ such $o_2 \equiv o_3$ and there are isa paths from c_3 to both c_1 an c_2 and $o \in I_C(c_3)$*
2. surjectivity *if $o \in I_C(t(r))$ then there is an o' such that $(o', o) \in I_R(r)$*
3. duplicate-free attributes[3] *if $(o_1, o_2) \in I_R(r)$ and $(o_1, o_3) \in I_R(r)$ then $o_2 \not\equiv o_3$*
4. equivalent common attributes[4] *if $o_1, o_2 \in I_C(s(r))$, $o_1 \equiv o_2$ and $(o_1, o_3) \in I_R(r)$ then there is an $o_4 \in O$ such that $o_3 \equiv o_4$ and $(o_1, o_4) \in I_R(r)$*

It is important to understand (1) why the graph semantics and the tree semantics are not equivalent, and (2) why this is not important in the context of this paper. However, the reasons for this can only be given when we have defined the access rules.

[3] Every equivalence class appears only once in attribute.

[4] Equivalent nodes have the same set of equivalence classes in common attributes.

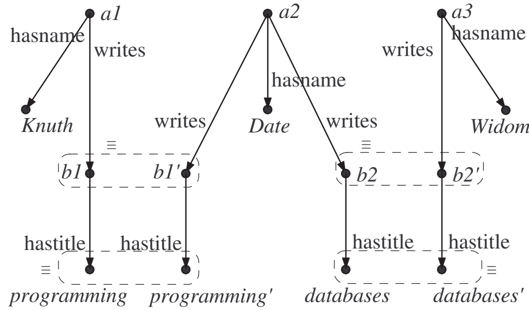

Figure 4: A tree instance corresponding to the graph instance shown in Figure 2.

4 Access Rules

We have now defined the data model for the forms system. Up to now a form has a hierarchical schema to which a number of hierarchical instances correspond. This means that users accessing a form can now be shown instances of the form, and can also make changes to the instance. However, we need to be able to specify access rules to each object in the instance. We will describe such rules in this section.

4.1 Actions on Data

Users of the forms system will want to do a number of things with it. Firstly, they want to open a form to see the instances that are associated with it. Clearly, it should be possible to restrict users to see only instances they are entitled to see, as defined by the form's designer. Secondly, they will want to update information in the form, either by changing values in an existing instance, or by creating a new instance all together.

A good example is the paper submission form present in our department. Users of the form will want to see those details of all papers that they are entitled to see. They will also want to be able to change the title of a paper, but only if it's a paper of which they are an author. Likewise, they should be able to add a new paper as long as they are an author of it.

In our system, we will allow the designer of a form to specify access rights tied to the schema of the form. The access rules in the schema will be evaluated over the corresponding instances, and only those objects in the instance where the rules are satisfied will be accessible.

CRUD. Of the usual CRUD (Create, Read, Update, Delete) rules used for accessing data objects, only the C, R, and D rights will be needed. This is because our rules will be tied to edges in the data model, and edges have no properties of their own, making an update of an edge meaningless. Updating the value of a property of a class involves removing an edge and creating a new one. Furthermore, as in previous work [13], we will only consider leaf operations: edges can be created or deleted if they appear as leafs in the tree. Larger operations on the tree (such as a move) can be simulated by a sequence of leaf operations.

Note that there is no need for propagation of updates between different peers, as each individual data item is stored at a unique location and peers that use it must obtain it from this location[1].

Handling updates on form schemas when they are used across different peers is another matter. We assume that in such instances the system will notify affected users.

4.2 Access Rule Language and Forms

As mentioned in Section 2, our access rule language is based on a limited subset of XPath corresponding to FO^2. Specifically, we use XPath's surface syntax, including conditions containing path expressions, but excluding the descendent-or-self axis (denoted as `//`). As our instances resemble nested relations more than semi-structured data, the nesting depth is always fixed in the schema, thus making this axis unnecessary.

The path expressions can be used on their own, meaning that existence of the end-node is checked, or in a comparison to a constant or one of three system variables `userid`, `date`, and `time`. In addition, path expressions can be combined using *and*, *or*, and *not*.

Definition 7 (Access Rule) *Given a frame $F = (C, R, s, t)$, an access rule is a tuple (e, o, r) with edge $e \in R$, o a* create, read, *or* delete *operation, and r an access rule expression.*

Thus, access rules are attached to edges in the frame of a schema and indicate that the operation in question may be performed when the access rule expression evaluates to true. An access rule is evaluated over each instance tree of the forest of instances belonging to a schema. Evaluation of a path expression starts from the node in the tree that corresponds to the class from which edge e departs in the schema.

Note that this definition of an access rule allow very fine-grained security provisions. Indeed, access is regulated to the level of individual attributes, giving the designer of a form full control over how data captured through her form, but also other forms that re-use part of her schema, is used.

Definition 8 (Form) *A form is a tuple (S, A) where S is a schema with a frame F that is a single tree, and A is a set of access rules.*

The notion of a form is the central idea in this paper: a form represents a tree-like data model with `isa` constraints and a set of access rules, and corresponds with a forest of tree instances over its schema. Users may access and edit the instances as long as the access rules are satisfied. Individual objects in the instances may be shared over various forms.

A Comprehensive Example. We now show how the real-life example described in the Introduction can be solved in our forms-based information system. Figure 5 gives a graphical representation of the schema of the *Leave Application* form. A non-exhaustive list of corresponding access rules is given in Figure 6. Note that we use the abbreviation U as a shorthand for both *create* and *delete* rules.

Rule (1) means that users can only create new leave applications for themselves. Rule (2) means that users can only see their own applications, except for the Dean and the Head of Department, who can see all applications. The begin and end dates for leave applications can only be changed by the user whose application it is, as stated by rules (3) and (4), and only if the Head of Department has not already approved the application. Rule (5) indicates that only

[1] Of course, for efficiency reasons, an actual implementation may choose to propagate updates instead; our model doesn't necessitate this but does allow it.

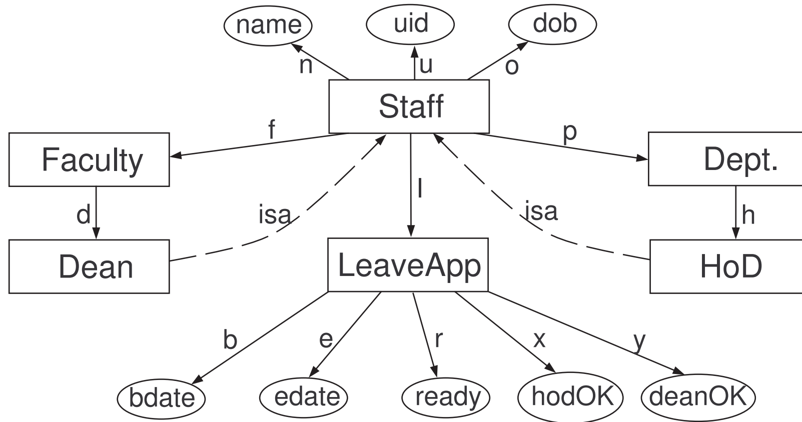

Figure 5: The Schema of the *Leave Application* Form.

$$(l,\ C,\ ./\texttt{uid} = \text{userid}) \tag{1}$$
$$(l,\ R,\ ./\texttt{uid} = \text{userid}\ \vee ./\texttt{f}/\texttt{d}/\texttt{uid} = \text{userid}\ \vee ./\texttt{p}/\texttt{h}/\texttt{uid} = \text{userid}) \tag{2}$$
$$(b,\ U,\ ../\texttt{uid} = \text{userid}\ \wedge\ \text{not}(./\texttt{hodOK})) \tag{3}$$
$$(e,\ U,\ ../\texttt{uid} = \text{userid}\ \wedge\ \text{not}(./\texttt{hodOK})) \tag{4}$$
$$(x,\ U,\ ./\texttt{ready}\ \wedge ../\texttt{p}/\texttt{h}/\texttt{uid} = \text{userid}) \tag{5}$$
$$(y,\ U,\ ./\texttt{hodOK}\ \wedge ../\texttt{f}/\texttt{d}/\texttt{uid} = \text{userid}) \tag{6}$$

Figure 6: Some access rules for the *Leave Application* Form.

the Head of Department can set the `hodOK` attribute to true, and only if the user has indicated that her application is ready. Hence, the system can automatically notify the Head of Department that his input is sought, when rule (5) is satisfied. This illustrates how control-flow is derived from the access rules. Finally, rule (6) says that the Dean can approve the application when the Head of Department has already done so.

The definition of a form indicates that it is the form's creator (or owner) that sets up the access rules for individual data items described in the form's schema. When another person creates a new form that re-uses all or part of the original form's schema, the original access rules still apply, in addition to any new access rules defined by the new form. To give an example, suppose the above Leave Application form was created by the Faculty, but the Head of Department wants to capture additional data if his staff are applying for leave (e.g. he wants them to supply a reason). His new form will re-use the original form's schema *and* access rules, and in addition he can add rules. All rules must be satisfied before the operation can proceed. If the HoD wants the original rules to be modified, he will need to negotiate with the owner of the form that first defined the rule. We argue that this precisely captures real-life dynamics within an organization, making significantly liberated capture and re-use of data possible while maintaining the highest level of security.

4.3 Information Leakage

While the access rules language we presented provides a very powerful yet elegant method to constrain access, the method is not water tight. Consider the following scenario: form designer *Bob* creates a class C_1 and specifies that only he can read it. Now suppose a second person *Alice* creates a form with a class C_2 and specifies that the class C_2 may be read if $C_1.a = x$ (where a is some attribute of C_1 and x is a value for a). Alice (and others) can now infer the the value of $C_1.a$ by attempting to modify C_2, which is against the access rule specified by Bob.

A simple access rule evaluator will hence allow information leakage in certain cases. A somewhat naive solution would be to decree that access rules may only be evaluated over parts of the instance tree that the user may see. However, this is circular as now visibility may become dependent on visibility.

It seems that information leakage, however, is a decidable property of a set of access rules. Hence, a practical solution to this potential security concern is to check newly submitted forms with their access rules and reject those that test positive for the information leakage property.

5 Workflow Processes

It is clear that the access rules not only regulate who can see and update which part of the instance, but also that these rules impose an order on updates. Hence, it would be very useful if our system could automatically derive the workflow process associated to this form. We investigate this in a preliminary manner in this section.

The reason why we include this rather informal discussion in this paper is twofold: firstly, automatic derivation of workflow processes is one of the main motivations for introducing form-based information systems. Secondly, the section will show that there are some highly interesting, non-trivial research problems to be found in this topic. This is the secondary contribution of this paper, and may perhaps inform the direction of continued research associated to XForms.

The leave application form detailed in the previous

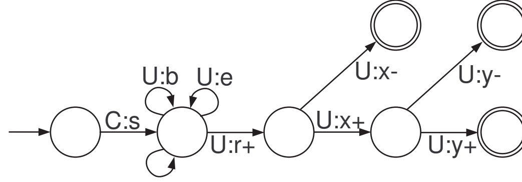

Figure 7: The *Leave Application* Workflow Process.

section illustrates that a form corresponds to a workflow process. In workflow literature, processes are usually modelled using finite state machines or more often using Petri nets [1, 23]. We will first examine finite state machines, attempting to establish whether one can be derived automatically for a form. Figure 7 shows a finite state machine (excluding some transitions for simplicity) modelling the workflow process of the leave application form shown in Figure 5.

States correspond to separate steps (or *actions*) in the workflow process. They represent an instance of the form at a certain time. A specific update of the instance represents a transition to another state.

The transitions are labelled with an abbreviation of the operation performed on the instance represented by the starting state of the transition. For example, `U:x+` means an update of edge x with value *true* (minus meaning *false*), while `C:s` means the creation of a new edge s to a new object of the LeaveApp class.

Importantly, the FSM in Figure 7 shows that different end states may exist. For this purpose, we extend the definition of a Form to include a *completed form formula*, expressed in the access rule language, which describes 'completion' end states; i.e., states that the creator of the form has indicated are acceptable completions of the form.

An interesting research question can now be posed: can we represent a form's workflow process using finite state machines such that reachability of completion states is decidable?

5.1 Canonical Instances

The first problem we need to solve is that of finding a finite representation of the infinite number of instances that may correspond to a form. The number of instances is infinite even when only structure is concerned and specific values are disregarded, since a relationship between two classes in a schema may have a many-to-many participation constraint. Figure 8 shows some instances for a simple frame involving relationships a and b.

Figure 8 also illustrates that it is possible to partition the set of infinite instances into a finite number of equivalence classes, which we call *canonical instances*. From the perspective of the access rule language (at least when considering only unary path expressions that check whether a node exists but does not compare values) each member of a canonical instance is indistinguishable from another member.

In the example, canonical instance I represents all instances in which no a edge exists (and hence no b edge under it). Canonical instance II represents all instances in which at least one a exists, but none of them have a b child. The third canonical instance represents those that have at least one a edge, and each of them has a b edge. Finally, canonical instance IV contains instances that have at least one a, and at least one of those has a b child.

Theorem 1 (Canonical Instances) *The set of all tree instances of a form can be partitioned into a finite number of canonical instances.*

Clearly it is very important that we now have a finite number of canonical instances, since finding a finite state machine representing a form's workflow process involves finding a finite set of states for the automaton.

Note that we can only create canonical instances for tree instances, not for graph instances. That is why we presented both in Section 3, and stated that we have to use the tree instances instead of the graph instances.

Unfortunately, a finite set of states for the FSM is not sufficient. Consider that transitions between instances going left to right in Figure 8's top row represent addition operations that add an a edge, while transitions in the opposite direction represent deletions of such edges. The problem is that the transition from canonical instance II to I also involves a deletion of the 'last' a remaining in II. We require these cases to be in two different canonical instances, because our XPath-based access rule language can distinguish between the two (e.g. `/a`). Hence, we require *counting* to determine if a transition stays within a canonical instance, or results in another canonical instance.

The conclusion is that we cannot use canonical instances as the states and update operations as the transitions of the FSM that is to represent the workflow process of a form. Therefore we briefly considered using Petri nets because they have the ability to do some counting using different *tokens* inside *places*. Unfortunately, use of *inhibitor* arcs [9] proved necessary to perform the counting we need, thus making reachability undecidable.

5.2 Decidability

The problems outlined above indicate that checking reachability of completion states is undecidable.

Theorem 2 (Undecidability) *Given a form with a schema, access rules, and a completed form formula, it is undecidable whether a completion state can be reached.*

The proof involves reduction to the two-counter automaton, a FSM with two registers that contain whole numbers, and that can check whether the registers contain 0. Transitions increment or decrement the registers. It is well known that two-counters are Turing-complete. We can simulate the registers using edges in instances, and check for 0 by expressing for example $not(a)$.

One positive result so far, involves dead-end states, or instances of a form that cannot be changed anymore. Dead-ends are usually interpreted in a negative way: as states that should not be reached because the

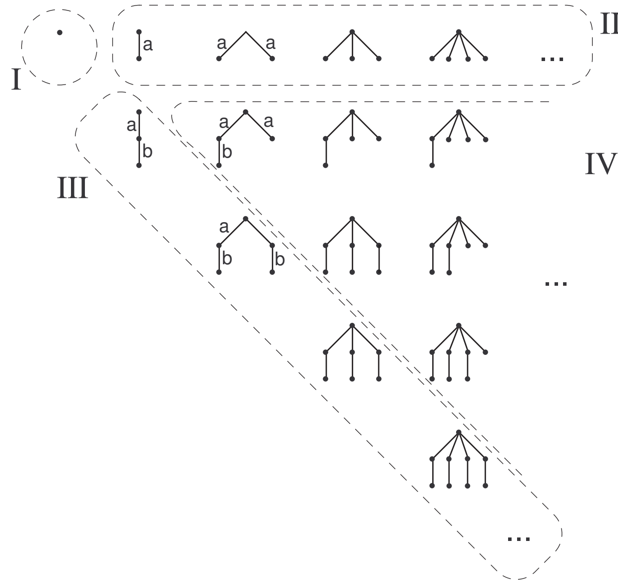

Figure 8: (Canonical) Instances over a simple schema. Some edge labels omitted for clarity.

system then enters a deadlock. However, it may be the case that a *completed form formula* describes a situation in which data in a form should not be updatable when some flag has been set. Not updatable means that it's a dead-end state, but one that represents a valid, correctly completed form. An example is where the Dean has approved a leave application, after which nothing in it can be changed anymore. In this sense, determining dead-ends is very worthwhile, helping the designer of the form to verify the rules he supplied with the form.

Theorem 3 (Dead-Ends) *Given a form including a schema and access rules, and an instance tree over the schema, it is decidable whether the instance can be updated.*

This result is due to our access rules language conforming to FO^2, which is a decidable subset of First Order logic using only two variables.

Unfortunately, because of the undecidability of general reachibility, we cannot prove that the dead-ends we found were reachable in the first place, making this result somewhat less practical.

To end our examination of the secondary aim of this work, i.e., under which circumstances would it be possible to automatically construct a workflow process from a form definition, we offer the conjecture that a positive result may be found when a slightly weaker access rules language is used.

Conjecture 1 (Decidability) *Given a form with a schema, access rules over the proposed XPath subset* without *negation, and a completed form formula, it is decidable whether a completion state can be reached.*

6 Conclusion, Implementations, and Future Work

Conclusion. We have presented a formal model for a form-based peer-ro-peer web information system. The model includes a definition for forms that incorporate a schema extended with access rules. A schema may be constructed by re-using elements from other schemas, both on the same peer and on other peers. This re-use is done via `isa` relationships. With a schema corresponds an instance in the form of a graph. To allow the access rules to traverse upwards to just one parent of a node, the instance is shredded into a forest of trees. The access rules impose an implicit order for data entry in the corresponding form, enabling us to check whether a workflow graph can be constructed, and to find specific states in the workflow.

Future Work

We are currently refining the data model we presented in this paper, and are investigating, given that data model, what subsets of the rule language do allow decidability while still maintaining a practical level of expressiveness.

We will also construct a rigorous proof, based on our current sketch, that the information leakage property we presented in Section 4.3 is decidable.

Implementation

On a practical level, we have already implemented a first prototype of very limited abilities [6] and are starting work on a second prototype that implements most of the ideas presented in this paper. Many practical issues, such as user interface design, will not be dealt with in this second prototype and will require additional research.

The second prototype is built around a substantially altered version of the Document Object Model as implemented in Java by the Xerces Apache project. It is currently a stand-alone forms server that accepts requests and updates and returns a form instance restricted by applying the relevant access rights. Extending it to enable peer-to-peer communication is being planned.

Derivation of workflow processes is not yet considered in the second prototype. This is the main goal of our third prototype, concurrently being planned. Here data is stored in a relational back-end, and a very restricted access rules language is offered. A key design

issue is how to push the checking of access rules to the relational database.

Acknowledgement

We would like to thank Toon Calders for sharing his insights in the decidability problem described in this paper, and Hua Wang for valuable discussions about security issues.

References

[1] W. van der Aalst: Petri-net-based Workflow Management Software. In Proceedings of the NFS Workshop on Workflow and Process Automation in Information Systems, pp 114–118, May 1996.

[2] W. van de Aalst, P. Barthelmess, C. Ellis, and J. Wainer: Workflow Modeling using Proclets. In Proceedings of CoopIS'00, pp. 198–209, 2000.

[3] W. van der Aalst and K. van Hee: Workflow Management: Models, Methods and Systems. MIT Press, 2001.

[4] W. van der Aalst, A. Weijters, and L. Maruster: Workflow Mining: Discovering Process Models from Event Logs. IEEE Transactions on Knowledge and Data Engineering (TKDE), volume 16(9), pages 1128-1142, 2004.

[5] W. van der Aalst, A. ter Hofstede, B. Kiepuszewski, and A. Barros: Workflow Patterns. Technical Report Eindhoven University of Technology. `http://is.tm.tue.nl/research/patterns/documentation.htm`, 2002.

[6] R. Addie: FormsFree – a secure but accessible system development and use of on-line forms. University of Southern Queensland Technical Report, December 2004.

[7] Advantys: `http://www.workflowgen.com/en/`, 2004.

[8] R. Agrawal, D. Gunopulos, and F. Leymann: Mining process models from workflow logs. Lecture Notes in Computer Science, 1377:469498, 1998.

[9] N. Busi: Analysis issues in Petri nets with inhibitor arcs. *Theoretical Computer Science*, 275:1-2, pp. 127–177, 2002.

[10] D. Calvanese, G. De Giacomo, M. Lenzerini, R. Rosati: Logical Foundations of Peer-To-Peer Data Integration. In Proceedings of PODS'04: pp. 241–251, 2004.

[11] D. Calvanese, G. De Giacomo, M. Lenzerini, R. Rosati, G. Vetere: Hyper: A Framework for Peer-to-Peer Data Integration on Grids. ICSNW'04: pp. 144–157, 2004.

[12] S. Cosmadakis, and C. Papadimitriou: Updates on relational views. Journal of the ACM, 31(4):742–760, 1984.

[13] S. Dekeyser, J. Hidders, and J. Paredaens: A Transaction Model for XML Databases. World Wide Web Journal, 7(1): 29–57, Kluwer, 2004.

[14] M. Dubinko, L. Klotz, R. Merrick, and T.V. Raman: XForms 1.0. World Wide Web Consortium (W3C) Recommendation, October 2003.

[15] J. Glasgow, G. MacEwen, and P. Panangaden: A logic for reasoning about security. ACM Transactions on Computer Systems 10(3), pp 226–264, August 1992.

[16] J. Hidders, M. Dumas, W. van der Aalst, A. ter Hofstede, and J. Verelst: When are two Workflow Processes the same? Computing: the Australian Theory Symposium (CATS'05), 2005.

[17] J. Hidders, J. Paredaens, P. Thiran, G-J Houben, K. van Hee: Non-destructive Integration of Form-based Views. In Proceedings of ADBIS'05, 2005.

[18] D. Hollingsworth: Workflow Management Coalition: The Workflow Reference Model Ten Years On. Technical Report, February 2004.

[19] H. Kozankiewicz, J. Leszczylowski, and K. Subieta: Updatable XML Views. In Proceedings of ADBIS'03, LNCS 2798, 2003.

[20] M. Lenzerini: Data Integration Is Harder than You Thought. In Proceedings of CoopIS'01: pp. 22-26, 2001.

[21] M. Lenzerini: Data Integration: A Theoretical Perspective. In Proceedings of PODS'02: pp. 233–246, 2002.

[22] Microsoft: InfoPath 2003 Product Information: The Microsoft Office information gathering and management program. `http://www.microsoft.com/office/infopath/prodinfo/trial.mspx`, 2004.

[23] T. Murata: Petri Nets: Properties, Analysis and Applications. Proceedings of the IEEE, 77(4):541–580, April 1989.

[24] S. Nanchen and R. Stark: A Security Logic for Abstract State Machines. In Proceedings of the 11th International Workshop on Abstract State Machines '04. LNCS 3052, 2004.