The XForms Computation Engine: Rationale, Theory and Implementation Experience

John Boyer, Ph.D.
PureEdge Solutions, Inc.,
Victoria, BC, Canada
tel. +1 250 708 8047, fax +1 250 708 8010
email jboyer@PureEdge.com, jboyer@ACM.org

Mikko Honkala
Telecommunications Software and Multimedia Laboratory
Helsinki University of Technology, Finland
tel. +358-9-451 4794, fax +358-9-451 5253
email Mikko.Honkala@iki.fi

ABSTRACT

This paper reports the successful efforts to change the W3C's next-generation Web forms working draft specification, XForms, from a computation engine architecture based on form-author-specified recalculation order to an automated determination of recalculation order based on optimal graph algorithms. We trace the historical beginnings of these algorithms from Knuth and Tarjan to their first known applications in electronic spreadsheets and XFDL (the first XML-based electronic forms vocabulary). The algorithms are then presented in the context of a detailed example. Also included are the implementation details that were necessary to add the new XForms recalculation engine to the open-source X-Smiles web browser. When compared to an implementation of the approach in earlier XForms drafts, running times were reduced from seconds to instantaneity. Finally, the paper discusses some of the technical challenges encountered when rationalizing the graph algorithms with the properties of XPath and an implementation of XPath.

KEY WORDS: XForms, recalculation, topological sorting, depth-first search.

1. INTRODUCTION

The growth and success of the World Wide Web are predicated in no small part on the availability of interactive services such as search engines, online banking and e-commerce. A key technology used in interactive Web applications is HTML forms [HTML]. However, business requirements for these services have steadily increased in number and complexity since the advent of this technology, and many of today's high-end forms use complex client-side ECMAScript [ECMA] programming to achieve form field validation and simple computations (or bounce the form back and forth to the server). Heavy use of scripting inevitably leads to low maintainability and accessibility. Moreover, the current trend in server-side web application data processing is toward manipulation and even storage of information in XML format [XML], taking advantage of the flexibility of this more abstract document storage format. HTML forms do not integrate well with XML. Therefore, the World Wide Web Consortium (W3C) [W3C] is specifying XForms [XForms], the next generation Web forms.

XForms, a work in progress that tries to solve the problems mentioned above, has three logical layers:

Instance

An arbitrary XML document that is modified by client-side user interaction, then submitted to a server.

Model

Uses XML to define the *constraints* on items of the instance, which includes data types and ranges as well as computational relationships.

User Interface

Defines how the form is shown and expresses bindings to instance items. User input is governed by rules in the model for the instance item being modified (through a bound input control).

As an example, let's consider a purchase order form. A purchase order has multiple lines of items with units, price and total. The user can add and remove items as well as change the number of items to order. The form must calculate an item total for each line by taking the product of the number of units and the unit price. The form must also calculate a subtotal, which is the sum of the line totals, and an amount of tax on the subtotal. Finally, the form must calculate a grand total, which may include a discount factor if the total exceeds a certain value. Figure 1 depicts a typical screen display of such a form.

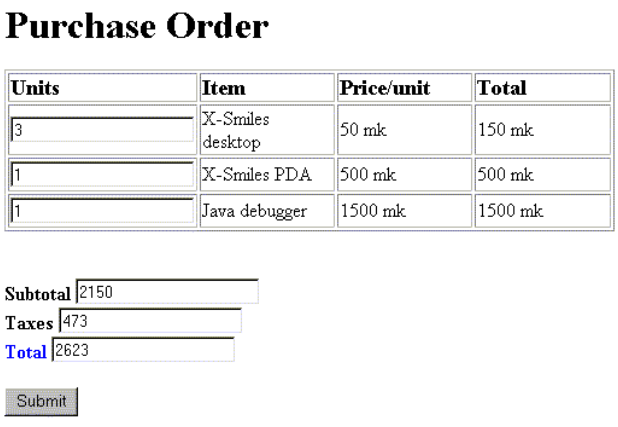

Fig. 1. A screenshot of a purchase order form

In order to show how XForms tackles this form, we start by designing an XML instance that describes a purchase order. The instance has exactly one element of the following: 'items', 'totals', and ' info'. The 'items' element consists of 0..n 'item' elements, which have 'name', 'units', 'price' and 'total' for each line of the purchase order. An example of such instance data is shown in Figure 2.

```
<purchaseOrder xmlns="">
    <items>
      <item><name>Item 1</name>
        <units>3</units><price>50</price>
        <total>0</total>
      </item>
      <item><name>Item 2</name>
        <units>1</units><price>500</price>
        <total>0</total>
      </item>
      <item><name>Item 3</name>
        <units>1</units><price>1500</price>
        <total>0</total>
      </item>
    </items>
    <totals>
      <subtotal>0</subtotal>
      <tax>0</tax>
      <total>0</total>
    </totals>
    <info><tax>0.22</tax></info>
  </purchaseOrder>
```

Fig. 2. The instance data for purchase order

The next step is to define a model for the form. Figure 3 shows a model definition for the purchase order form. The instance is fetched from an external URL. The datatypes and basic value constraints are defined by a separate XML Schema [Schema] document, which is also referenced in the model definition header. However, our focus in this paper is on the XForms computational constraints expressed by the 'bind' elements. These model item definitions are bound to the instance data using an XPath expression [XPath] in the `ref` attribute. Calculations of instance node values and other XForms constraints are also expressed using XPath. These calculation expressions are called constraints because they are not simply evaluated at the initialization time, but rather they are enforced throughout the life of the form by re-evaluation every time the instance changes. The XForms processor must decide which expressions to evaluate at which times. The part of the XForms processor that accomplished this is called the *XForms Calculation Engine*, the algorithmic details of which are discussed in the next section.

```
<html>
<title>XForms example</title>
<head>
<xfm:model>
 <xfm:instance xlink:href="data.xml"/>
 <xfm:schema xlink:href="purchase.xsd"/>
 <xfm:bind
  ref="purchaseOrder/items/item/total"
  calculate="../units * ../price"
  relevant="../units > 0" />
 <xfm:bind
  ref="purchaseOrder/totals/subtotal"
  calculate="sum(../../items/item/total)"/>
 <xfm:bind
  ref="purchaseOrder/totals/tax"
  calculate="../subtotal*../../info/tax"/>
 <xfm:bind
  ref="purchaseOrder/totals/total"
  calculate="if (../subtotal + ../tax>4000,
     ../subtotal + ../tax,
     (../subtotal + ../tax) *0.9)"/>

 </xfm:model>
</head>
```

Fig. 3. XForms model definition for the purchase order

The first bind element defines the calculation 'units * price' for every line item in the purchase order. It also defines that the 'total' field is relevant (shown) only when the 'units' field has a value greater than zero. Only a single declaration is required for all line items because the XPath expression in the attribute `ref` selects all `total` nodes from the items. The `calculate` attribute then defines the calculation for each `total` node using that node as the *context* node [XPath]. The other bind elements define the rest of the calculations, the tax and the total with the possible discount.

The final step is to design the UI for the purchase order. The design of the UI is out of the scope of this paper.

As we have now seen, it is possible to define complex interactive calculations in XForms using just a few lines of declarative markup. Similar calculations in HTML forms would have to be programmed with hard-to-maintain JavaScript code. Most of the work in XForms in done by the recalculation engine in the XForms processor.

2. THE RECALCULATION ALGORITHM

The XForms recalculation algorithm is based on a method called *topological sorting*, which creates a *natural order* or *linear order* to run computations such that all values reference in an expression have already been recalculated before the expression is selected for recalculation. According to [CLR90], Knuth was the first to provide a linear time algorithm for topological sorting for use in such diverse applications as interdependent task scheduling with 'PERT' charts, networking problems and even linguistics [K68]. In 1982, topological sorting also

became the basis of spreadsheet update algorithms. According to Dan Bricklin, inventor of the electronic spreadsheet, a simple row-by-row, column-by-column algorithm was used to update VisiCalc spreadsheets [Bricklin]. In general, the simplest possible algorithms were used to conserve memory. However, Mitchell Kapor, who founded Lotus Development Corporation, indicated that a natural ordering algorithm was used in Lotus 1-2-3 Release 1, the successor of VisiCalc (publicly released in January of 1983) [Kapor]. An interesting historical note made in [Bricklin] is that the Lotus 1-2-3 update algorithm was created after noting the similarities between the spreadsheet update problem and the LISP garbage collector, in which objects are destroyed when their number of referrents drops to zero. While the Lotus 1-2-3 update algorithm may not have been based on foreknowledge of Knuth's topological sort, in 1982 Frank Ruskey also communicated topological sorting and the insight of its applicability to spreadsheets to Peter Eichhorst and Jim Kearney, who were then working on the CalcStar spreadsheet from Micropro International Corporation [Ruskey].

Although electronic forms tend to permit computations on more than just values, the problem of updating computationally related properties and values in an electronic form is analogous to the spreadsheet update problem [BB99]. The first XML-based forms definition language, the Extensible Forms Description Language (XFDL) [XFDL-W3CNote], used a method of recalculation similar to the VisiCalc update algorithm. However, version 4.3 (and higher) of XFDL [XFDL-44] was the first XML-based form definition language to use (an elaborated variation of) the linear time algorithms described in this paper for resolving computations, and as such provided the conceptual basis and prior implementation experience for the current XForms effort.

The topological sort is a method for sequencing the members of a set *S* such that all constraints in a relation are satisfied. Given a set of integers and the less-than relation, a topological sort would produce the same output as a classical sort because less-than produces a result for every pair of integers. Topological sorting is used when the relation does not express a *total order* on the set, i.e. when the relation does not specify a result for every pair of elements in the set. A good example is a set of events in which certain events must precede certain other events, but some pairs of events are not dependent on one another and could therefore occur in any relative order. A topological sort would determine a schedule for the events by creating an event sequence in which each event *e* occurs after the events which must precede *e* and before any events that *e* must precede. While this line of thinking could be applied to many scheduling tasks, such as scheduling of inter-related software development tasks, it is also suitable for scheduling the order in which a sequence of computational expressions should be run when some of the expressions refer to other computed expressions.

2.1. XForms Recalculation as a Topological Sorting Problem

According to [XForms], the `bind` element expresses computations for the values and properties of the instance data nodes indicated by the `ref` attribute. In the example of the previous section, an expression in the `calculate` attribute of a `bind` element was applied to multiple nodes of instance data specified by `ref`. One can think of the calculate expression as being instantiated for each node indicated by `ref`, with the current values of the instantiated expressions being stored in the element content of the respective instance nodes. Using other attributes such as `readOnly`, `required` and `relevant`, the `bind` element can express properties of the instance nodes indicated by `ref`. Unlike calculated values, the current values of computed properties are not stored in the XML markup of the instance (i.e. they are stored by internal mechanisms of an implementation).

For the purpose of applying the topological sort, the computational dependencies in an XForms instance can be represented as a directed graph, or digraph, in which there is a vertex for each instance node and for each desired property of each instance node. Let *G* denote the computational dependency digraph consisting of a set *V* of vertices and a set of directed edges from referenced vertices to the vertices that reference them. A directed edge (*v*, *w*) exists in *v* if it is necessary to compute the value of *v* in order to compute the value of *w*. In this case, *w* refers to *v* such that a change of value of *v* implies the need to recalculate *w*.

For example, consider the purchase order example of Section 1. The dependency digraph for the bindings in Figure 3 is shown in Figure 4. Each of the three purchase order rows is represented by three vertices for the number of units, price per unit and item total. There are three more vertices to represent the subtotal of all item totals, the tax, and the grand total. The directed edges from each `Units` value vertex to the `Relevant` vertex for the same item represents the calculated `relevant` model item property on the item's `Total`. Likewise, the directed edges from the `Units` value vertex and `Price` value vertex of each item to the `Total` value vertex for each item represents the references made by the calculate attribute in `bind` element (1) in Figure 3. The second `bind` element in Figure 3 invokes the XForms sum() extension function on the node-set given by its parameter. The node-set finds all item `Total` elements, so the dependency digraph contains

directed edges from each item's `Total` value vertex to the value vertex for the `Subtotal`. Similarly, the directed edges from the `Subtotal` value vertex to the value vertices of `Tax` and `Total` are due to the references to the `Subtotal` element value in the calculate attributes of `bind` (3) and (4) in Figure 3. Finally, the calculate attribute in the fourth `bind` in Figure 3 also references `Tax`, resulting in a directed edge from the `Tax` value vertex to the `Total` value vertex.

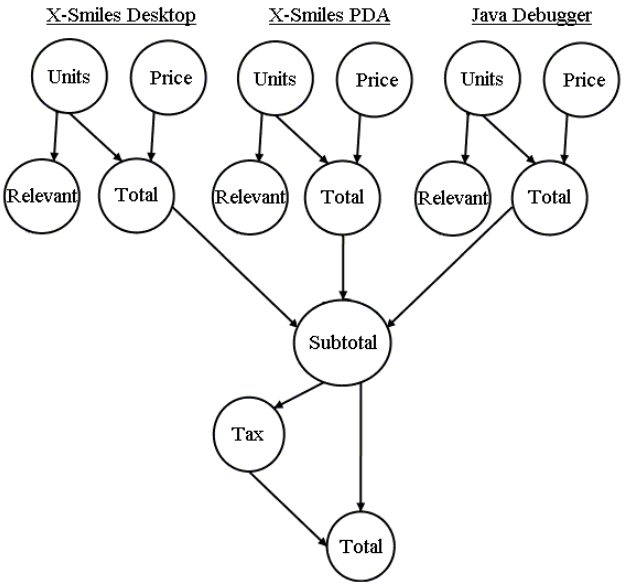

Fig. 4. Main Dependency Graph for the Purchase Order Example

Now suppose the user changes the number of units of X-Smiles Desktop licenses from 3 to a new value of 50. It is clearly necessary to first recalculate the `Total` value of the X-Smiles Desktop item before recalculating the purchase order `Subtotal`. Then, the `Subtotal` value vertex has directed edges leading to the `Tax` and `Total` value vertices, both of which must be recalculated. However, the `Tax` value vertex also has a directed edge to the `Total`, so `Tax` must be recalculated first. Finally, note that the `Units` vertex also has a directed edge to `Relevant`, but there are no interdependencies between `Relevant` and any of the other values needing recalculation, so the `Relevant` vertex could be recalculated at any time. Thus, this example demonstrates that XForms recalculation sequencing fits the profile of a topological sorting problem. The elements of the set to be sorted are vertices of the dependency digraph, and the *partial order* relation is expressed by paths of directed edges. The order of calculation of two vertices is interchangeable unless there is a path of directed edges that connects the two vertices.

2.2. Efficient Calculation of the Pertinent Dependency Subgraph

The topological sort systematically explores paths of directed edges emanating from a given vertex. In XForms, the given vertex is the one representing the value changed by user input. The topological sort explores all paths leading to a vertex before visiting the vertex (in XForms, visitation implies re-evaluation of the associated computational expression). However, before the topological sort can be used to determine a recalculation sequence, it is first necessary to create a reduced form of the dependency digraph called the *pertinent dependency subgraph* that contains only the paths in the dependency digraph that emanate from the vertex changed by user input. These paths are *pertinent* because they contain the vertices that may change in value upon re-evaluation of their expressions. As such, only the pertinent paths leading to a vertex need be explored before visiting the vertex. Computing the pertinent dependency subgraph eliminates the non-pertinent paths so that the topological sort can easily detect when all of the pertinent paths leading to a vertex have been explored.

In certain cases such as the initial load of a form, the pertinent subgraph is simply a copy of the entire master dependency graph. In other cases such as user input, the pertinent dependency subgraph is computed by duplicating the vertices and edges along pertinent paths emanating from the changed vertex. It is also possible to have multiple changes made to user input controls (e.g. with a JavaScript function), which implies multiple simultaneous changes to the underlying instance nodes to which the user input controls are bound. To account for this possibility, we use L_c to denote a set of vertices associated with changed form controls. In the simple case of a user input change, the change list L_c would contain only one vertex.

The pertinent subgraph S for a given change list L_c can be identified in linear time using depth-first search [T72], a suitable version of which appears in the Xforms specification. To continue the purchase order example, consider the depth-first search on the dependency digraph in Figure 4 under the aforementioned condition that the X-Smiles Desktop Units value is changed to 50. The results appear in Figure 5. Only those vertices reachable by a path of directed edges from the X-Smiles Desktop Units vertex are retained.

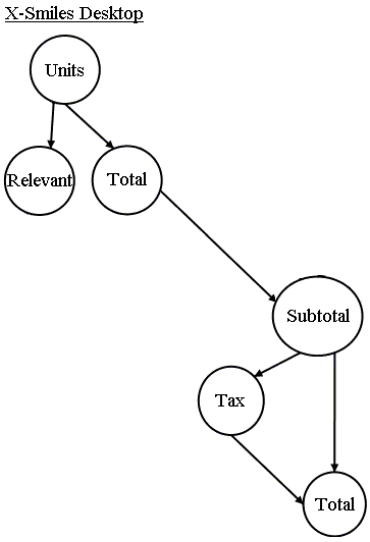

Fig. 5. Pertinent Subgraph for the Purchase Order Example

In order to ensure that construction and initialization costs for the pertinent subgraph are commensurate with the size of the pertinent subgraph and not the master dependency digraph, the initial size of the vertex array for *S* is one. To ensure that expansion costs remain at a constant factor as vertices are added to *S*, the array doubling technique can be used [DDJ98] (often, object oriented array implementations, such as the Java Vector class, use this technique). A side effect of having a smaller vertex array in the pertinent subgraph *S* is that the array index of a vertex *w* in the pertinent subgraph is likely different from the location of *w* in the master dependency digraph. Additional steps were taken in the XForms specification to account for this problem.

2.3. Topological Sorting of the Pertinent Dependency Subgraph

Except for the vertices corresponding to the initial elements from the change list L_c, the vertices of the pertinent dependency subgraph *S* form a set that must be sequenced by a topological sort, with the paths of directed edges in *S* as the partial order relation. The sequence of vertices produced by the topological sort dictates a valid order of XForms recalculation in which an expression is re-evaluated after all expressions on whose results it depends and before any expressions that depend on its result. The XForms specification contains pseudo-code for a version of the topological sort suitable for the XForms recalculation problem.

The topological sort begins by obtaining a list *Z* of all vertices in the pertinent subgraph that do not depend on any other vertices. Typically, these vertices in *S* correspond to vertices in the master dependency digraph *G* that are listed by the change list L_c. The exception is when L_c is not given, such as on form load when the full dependency graph is processed. The elements of *Z* are those that require no prior calculations, so they are not computed. The main loop of a topological sort removes a vertex *v* from *Z* for processing, The vertex *v* is processed by recalculating it as described below. Then, v and its outgoing edges are removed from the pertinent subgraph. If, as a result, any neighbors of *v* that drop to an in-degree of zero, then those neighbors are added to *Z* for processing (an in-degree of zero means that a vertex has no more unprocessed computational dependencies).

In XForms, the recalculation of a vertex is associated with a number of XForms-specific processing steps. If the vertex corresponds to the text content of an XML instance node, then the new value is stored in the instance node. Either using a simple dirty flag or using a queue, modified instance nodes are identified for subsequent update of the visual form controls bound to the instance nodes. If a vertex corresponds to some property of an instance node, then the property value is updated in the internal implementation and the effects of the property change are immediately propagated to the user interface.

Finally, it should be noted that topological sorting is an algorithm applied to directed acyclic graphs. However, it is not possible to prevent forms authors from writing circular references. The XForms specification defines additional steps that generate a circular reference exception one exists in the pertinent dependency subgraph. This is detected if the list *Z* of vertices with zero in-degree becomes empty without all vertices of *S* being recalculated. By contradiction, suppose *S* is a directed acyclic graph, but there exist one or more vertices that were not visited by the topological sort. Each such vertex must have an in-degree greater than zero, despite the fact that its in-degree has been decremented for each processed vertex on which it depends. Thus, each unprocessed vertex must be target of a directed edge that originates from another unprocessed vertex. Consider a subgraph *S′* of the original pertinent subgraph that contains only the (hypothetical) unprocessed vertices (and their directed edges). Consider traversing the directed edges of *S′* in reverse, i.e. from target to source. Starting with any vertex *v* of *S′*, traverse a directed edge (*u*, *v*) to obtain the source vertex *u*, which contains a dependency node indicating *v*. Perform this process repeatedly at each source vertex to find a successive source vertex. Every vertex in *S′* has in-degree greater than zero, so finding a successor is always possible. Moreover, if we find a successor that is also a vertex previously visited by this process, then a cycle has been found, contradicting the claim that *S′* and hence *S* are acyclic. So, we consider the case in which every vertex is visited without encountering a previously visited vertex. The last vertex *w* to be visited has an in-degree greater

than zero, so another previously visited vertex of *S′* has a directed edge leading to *w*, which again completes a cycle and proves that the XForms model contains a circular reference if the topological sort fails to process all vertices in the pertinent dependency subgraph.

2.4. Constraints on XPath Expressions and Their Rationale

The ability to compute a pertinent subgraph of the master dependency digraph is critically dependent on there being no way to create dynamic dependencies, i.e. dependencies that change as the result of recalculating values. If dependencies could change as the result of a recalculation then different parts of the master dependency digraph could become pertinent during the topological sort, so the pre-computed pertinent subgraph used by the topological sort would no longer be valid. Furthermore, the parts of the digraph that become pertinent could even be dependent on the order in which the recalculations are performed. Unfortunately, XPath has the expressive power to create dynamic computational dependencies. To solve this problem, the XForms specification characterizes the types of XPath expressions that can result in dynamic dependencies and creates constraints against those XPath expressions.

In XForms 1.0, the dependencies must be rebuilt if instance data elements are added or deleted, so actions such as adding a row to a purchase order still result in a form that operates as one would expect (i.e. due to the rebuild of dependencies, computations within the new row are operational, and the row's item total becomes part of the purchase order subtotal and related calculations). However, the addition and deletion of items is assumed to occur outside of a recalculation, so XForms forbids functions that have side effects on the structure of the instance data. Moreover, functions that have any side effects at all are also forbidden because these changes cannot be accounted for by the dependency digraph.

In general, XForms 1.0 also forbids the use of XPath language constructs that could result in a change of node-set references within an XPath expression based on any changes to the instance data. Since a function can receive node-sets as parameters and make calculations over the values in the nodes of the node-set, changes to the referenced nodes obviously affect the return result of the function. Therefore, it is forbidden to use a function that returns a node-set. As well, dynamic predicates are forbidden. For example, if a calculate attribute contains the subexpression `foo[@bar="xy"]`, then the subexpression is forbidden if the attribute `bar` is itself calculated or if it is bound to an input form control. Finally, XPath variables are forbidden, which is not actually problematic since XForms 1.0 provides no variables to the XPath evaluation context.

3. CALCULATION ENGINE IMPLEMENTATION IN X-SMILES

X-Smiles is an Open Source XML browser implemented in Java. It supports multiple XML languages and allows mixing them in a single document. Each XML language is handled by a Markup Language Functional Component (MLFC). An MLFC is basically a markup language-specific Document Object Model (DOM) implementation and renderer. Currently implemented host MLFCs include XSL Formatting Objects (XSL FO), Synchronized Multimedia Integration Language (SMIL) and Scalable Vector Graphics (SVG). XForms MLFC is a 'parasite' MLFC; it needs some other MLFC as the host. Currently, XForms can be embedded in SVG, SMIL and XSL FO documents. [X-Smiles]

The XForms implementation in X-Smiles is the first (and currently the only) [XForms1] browser implementation of the XForms Working Draft. Most of the XForms features are implemented in the browser. An important part of the implementation is the calculation engine, which is implemented with the algorithms described in Section 2.

3.1. Overview of the Implementation

Figure 9 presents an overview of the X-Smiles browser implementation of the XForms calculation engine. It includes three distinct types of data structures:

- *The presentation DOM* is the memory representation of the host XML document, e.g. the SMIL document that hosts the XForms elements. The implementation of this markup language is a specialized tree extended from a generic DOM implementation. It contains host DOM nodes implemented in the host MLFC and XForms DOM nodes which implement, for example, a specific form control.
- *The instance DOM* is an XForms internal DOM implementation, also derived from the generic DOM, that holds the current state of the instance, namely instance data item values and their constraint states, such as 'relevant'.
- *Dependency graphs* are used by the calculation engine. There are two dependency graphs per instance: *The Master Dependency Graph*, which is created at initialization time, and *The Pertinent Dependency Subgraph*, which is generated on the fly from the main graph whenever the calculation engine is run. The dependency graphs are composed of Vertices and Edges as described in Section 2.

An XForms presentation DOM node is linked two-ways to the corresponding instance DOM node. The only

relevance of the presentation DOM to the calculation engine, are the instance item value updates, which are reflected to the presentation DOM and vice versa. The dependency graph's vertices are linked one-way to the instance DOM nodes. There may be many vertices linked to the same instance node, as long as they are of different type (e.g. calculate and relevant vertices), as shown in the figure.

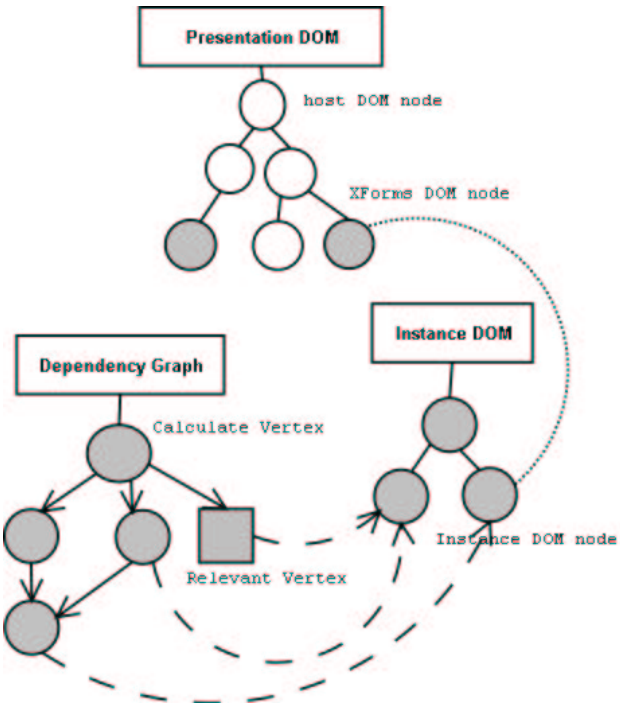

Fig. 6. Overview of the implementation

3.2. Obtaining the Referents of an XPath Expression

As shown in the Introduction, a computable constraint, such as `calculate` in a `bind` element, can express a computation for multiple instance data items. Each computation can refer to multiple data items. For example, the following bind expression attaches a calculation of `price * units` to every item `total`.

```
<bind ref="purchaseOrder/items/item/total"
          calculate="../price * ../units"/>
```

Fig. 7. Code for calculating the totals

This leads to an important prequisite for building the Main Dependency Graph; the ability to obtain the referents of each XPath expression, e.g. how to find out which instance data items the calculate expression `../price * ../units` in the above example references. This is not readily possible with an arbitrary XPath processor, so an implementation must usually build support for it. X-Smiles uses XPath processor in Xalan, which had to be extended to support this feature. We execute an XPath expression once in its correct context (namespace and XPath context nodes), and collect references to every node that the expression's location paths reference [XPath]. This is sufficient for XForms' purposes. Actually the expression need not be fully executed since the result is not interesting at initialisation time, but only the referents of the expression.

3.3. Class Structure

In X-Smiles, the following objects hold the state of the calculation engine: abstract superclass *DependencyGraph*, and derived classes *MainDependencyGraph* and *SubDependencyGraph*. These classes contain the methods for creating the graphs and for the recalculation.

The implementation of Vertices and Edges is done in the way described in Section 2. There are five types of vertices derived from a single class *Vertex*. The edges are represented by the vector *depList*. Vertex has the members described in Section 2 plus an additional member, the reference to a *bind* DOM element. The bind element is used to associate the Vertex with the XPath expression to be executed. The bind element is also the context node for resolving namespaces for the XPath expression. Each of the vertex types override single method: *compute()*. This is where the XPath expression is executed and the result is handled according to the Vertex type.

3.4. Run-time Processes

The **initialization** of the calculation engine consists of the creation of the Master Dependency Graph and performing a full calculation with it. The referents of each XPath expression are obtained as described above. For optimal performance, the X-Smiles implementation performs the first complete recalculation directly on the master graph. This has the side effect of invalidating the *inDegree* members of the vertices. However, it does not matter, because inDegree's are recalculated for each pertinent subgraph. The entry points for full initialization are:

- *Document load.* This is when the main initialization happens.
- *Insertion or deletion of instance data items.* This can happen only in the 'xform:repeat' construct. It could also be possible to optimize insertion or deletion by modifying only the corresponding part of the main graph, thus removing the need for full initialization.
- *Reset.* At Reset, the instance DOM is reconstructed completely and the whole form is reinitialised. This happens when the xforms:reset event is dispatched to a model.

Recalculation happens when an instance data item's value changes. First, a Pertinent Dependency Subgraph is

constructed, then the re-calculation algorithm is run on it. The possible entry points for recalculation are:

- *User input.* The user has changed a value in a form control.
- *setValue.* This XForms' declarative event is fired.
- *ECMAScript.* ECMAScripts can access the instance DOM and change values thereof.

Whenever an instance data item's value changes, the user interface has to be updated accordingly. In X-Smiles, the form controls register themselves to the instance data items, which keep an internal vector of referent form controls in memory. Every time an item's value or other property such as 'readOnly' changes, the item asks the form controls to change their display accordingly. Whether this has immediate effect depends on whether the component is currently on screen or not.

Since the instance DOM is a specialized implementation derived from a general DOM, the model item properties, such as relevant and readOnly, are kept within an extra object *InstanceItem*, created for each of the instance DOM node implementation. The instance DOM has extended implementations of *ElementNSImpl*, *AttrNSImpl*, and *AttrImpl*. There are accessor methods in the instance DOM for getting the InstanceItem and its properties.

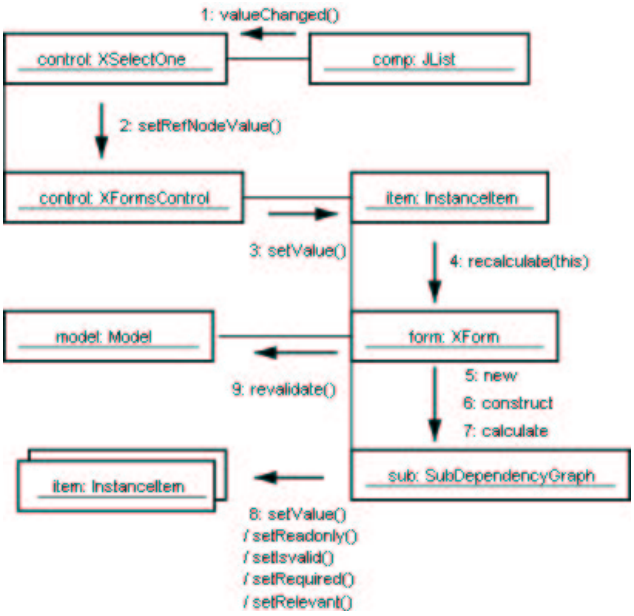

Fig. 8. Recalculate Collaboration Diagram

Figure 8 shows the collaboration diagram of user changing the value in a 'selectOne' form control. The sequence is initiated when the *JList* component used by the 'selectOne' element dispatches the *valueChanged* event, which the selectOne listens to. After recording the specific form control, the base class calls setValue() for its bound InstanceItem. The instance item then calls recalculate() with a change vector containing itself. The recalculate is applied to the XForm object, which creates a new *SubDependencyGraph* and calls calculate() on it. SubDependencyGraph then runs the recalculation algorithm on itself, as described in Section 2, and changes the values and properties on the recalculated InstanceItems, which in turn are reflected to the user interface. After the recalculation, the revalidate() function is called on the *Model* object to determine whether any schema constraints [Schema] have been violated or corrected.

4. CONCLUSION

In this paper, we presented the results of a successful application of optimal graph algorithms (topological sorting and depth-first search) to the W3C's next-generation Web forms specification, XForms. We tracked the history of these important and classic algorithms, especially their first known applications in electronic spreadsheets and XML-based electronic forms. Moreover, we presented the algorithms with pseudo-code and highly detailed explanations and examples. We also presented the implementation details necessary to run the recalculation engine within the X-Smiles web browser. Finally, we described some of the technical challenges that arose when applying these algorithms in the context of XForms (ongoing challenges are described below).

The application of these algorithms within the X-Smiles browser is especially significant because the initial approach to computations in early drafts of XForms was already implemented, providing a basis for empirically assessing the improvement resulting from this work. The earlier approach required the user to define the order of calculation, and all calculations would be run after any change of the instance data. This lead to a few problems. Firstly, the calculation order was difficult to define manually and harder to maintain. Secondly, running all calculations was quite inefficient, taking a few seconds on a large form. Using the depth-first search to calculate the pertinent subgraph in linear time, and using a topological sort to determine the recalculation order in linear time, the same forms were given to instantaneous update.

After the release of XForms 1.0, a number of areas of future work exist for XForms recalculation. The working group deliberately deferred the creation of XPath accessor functions for all model item properties. The value of an instance node can be referenced in a calculation, but properties such as `relevant` and `required` cannot be used in other computational expressions because they are not represented in the instance data. Adding accessors significantly complicates the creation of the dependency

digraph since we must not only know what references an expression makes but also which references appear as parameters to an accessor function. This, in turn, requires a node-by-node analysis of the XPath expression parse tree, which may not be available. A second area of exploration would be to specify how to most efficiently update the master dependency digraph based on the addition or deletion of instance data nodes. In XForms 1.0, we simply rebuild the entire dependency digraph, which could be a costly operation when adding a few elements to a large form. If the elements added or deleted are not heavily referenced, then a stream-lined approach would yield a great benefit. Finally, the use of advanced data structures to handle dynamic pertinence should be investigated. The performance would likely degrade to *O(n log n)*, which is slower than the *O(n)* algorithm reported in this paper but still fast enough in practice. The advantage would be the ability to overcome the constraints described in Section 2.4, thereby increasing the flexibility of the XPath expressions that could be used in XForms computations.

5. ACKNOWLEDGEMENTS

The work of Mikko Honkala has been funded by the HIIT XML Devices project. He would also like to thank the Nokia Oyj Foundation for scholarship and support during this research.

REFERENCES

[BB99] B. Blair & J. Boyer. XFDL: Creating Electronic Commerce Transaction Records Using XML. *Computer Networks: The International Journal of Computer and Telecommunications Networking*, vol. 31, pp. 1611-1622, 1999. Also presented at the Eighth International World Wide Web Conference and available at http://www8.org/w8-papers/4d-electronic/xfdl/xfdl.html.

[Bricklin] D. Bricklin. *Personal Communication.* Regarding update algorithm used in VisiCalc. October 23, 2001.

[CLR90] T. H. Cormen, C. E. Leiserson, & R. L. Rivest. *Introduction to Algorithms.* MIT Press, 1990.

[DDJ98] J. Boyer. Resizable Arrays, Heaps and Hash Tables. *Dr. Dobb's Journal*, January, 1998.

[ECMA] ECMA, *ECMAScript Language Specification 3rd Edition*. Standard ECMA-262, December 1999. Available at ftp://ftp.ecma.ch/ecma-st/Ecma-262.pdf

[HTML] Dave Raggett, Arnaud Le Hors, and Ian Jacobs (eds.). *HTML 4.01 Specification*. W3C Recommendation, December 24, 1999.

[K68] D. E. Knuth. *The Art of Computer Programming: Volume 1 Fundamental Algorithms.* Addison-Wesley, 1968. Third edition, 1997.

[Kapor] M. Kapor. *Personal Communication.* Regarding update algorithm used in Lotus 1-2-3 Release 1. October 24, 2001.

[Ruskey] F. Ruskey. *Personal Communication.* Regarding use of topological sort in CalcStar. October 19, 2001.

[T72] R. E. Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal of Computing*, Vol. 1, No. 2, pp. 146-160, 1972.

[Schema] H. Thompson et al. (eds.). *XML Schema part 1: Structures*. W3C Recommendation, May 2, 2001.

[XFDL-W3CNote] J. Boyer, T. Bray, & M. Gordon (eds.). *The Extensible Forms Description Language (XFDL) 4.0*. W3C Note, 1998. Available at http://www.w3.org/TR/NOTE-XFDL

[XFDL-44] J. Boyer, T. Bray, & M. Gordon (eds.). *XFDL Specification v4.4*. PureEdge Technical Manual, 2001. Available at http://docs.pureedge.com/xfdldocs/pdfdocs/XFDL44.pdf.

[XForms] M. Dubinko, et al. (eds.). *XForms 1.0.* W3C Working Draft, 18 January, 2002.

[XForms1] M. Honkala and P. Vuorimaa. *XForms in X-Smiles*. The 2nd Int. Conf. on Web Information Systems Engineering, Kyoto, Japan, December 3-6, 2001.

[XML] T. Bray et al. (eds.). *Extensible Markup Language (XML) 1.0 (Second Edition).* W3C Recommendation, October 6, 2000.

[XPath] J. Clark and S. DeRose (eds.). *XML Path Language (XPath) Version 1.0*. W3C Recommendation, November 16, 1999.

[XSmiles] P. Vuorimaa, T. Ropponen and N. von Knorring. *X-Smiles XML Browser*. The 2nd International Workshop on Networked Appliances, IWNA'2000, New Brunswick, NJ, USA, November 30 - December 1, 2000.

[W3C] World Wide Web Consortium, http://www.w3c.org/