Automated Update Management for XML Integrity Constraints

Michael Benedikt*, Glenn Bruns, Julie Gibson, Robin Kuss, Amy Ng
Bell Labs, Lucent Technologies

Abstract

A key problem in data management is the maintenance of integrity constraints as updates arrive. This is often done with handwritten code, but it is preferable to use code that is generated automatically from formally-defined constraints. A problem in doing so, especially for applications in which data is stored as an XML document, is the lack of a powerful integrity constraint language for which incremental maintenance can be performed efficiently. This paper presents ΔX, an XML-based language for integrity constraints and a compiler for producing incremental constraint-checking code. The ΔX language is based on XPath and XML Schema, but is specifically designed to support the automated generation of incremental constraint-checking code.

1 Introduction

Integrity constraints describe what it means for a database instance to be in a "good" state. A significant piece of database infrastructure is the code that checks whether a database instance satisfies its integrity constraints. One way to perform these checks is by a global analysis of the database instance. However, in most applications it makes sense to check the constraints incrementally. In this approach one assumes that no constraints are currently violated, and before a database entity is created, deleted, or modified, the relevant parts of the database are examined to see if performing the database operation would cause a constraint violation.

An application in which incremental constraint checking is used is the *provisioning* of communication networks. The operation of communication networks is governed by hardware and software configuration parameters, service and subscriber information, and parameters used by the protocols that construct and update routing tables. Much of this data is stored in databases within the network. The job of setting and updating this data is called *network provisioning*; the data itself is sometimes called *provisioned data*. Incremental checking is critical here because updates stream in over the network and must be applied to large datasets.

Network provisioning is a demanding database application, with requirements for high-speed data access, high reliability, and rapid data evolution. The need for high-speed access leads to significant redundancy in the data, which in turn leads to a complex data model. Reliability requirements imply that an enormous of amount of complex integrity checking must be done on update [GT94].

As XML is used to format data exchanged between machines over a network, it is not surprising that XML is increasingly used in network provisioning. In many ways XML is well-suited to the representation of provisioned data. For example, the hierarchical data model of

*Correspondence contact: benedikt@research.bell-labs.com, Room 9G-534, 2701 Lucent Lane Lisle, IL 60532. Telephone (630) 979-9954.

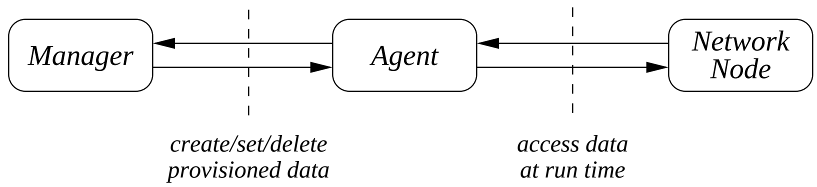

Figure 1: Data Provisioning with Network Management Processes

XML can easily represent network equipment hierarchies (e.g., shelves, slots, cards) and network topologies. The "self-describing" nature of XML data supports data evolution better than fixed data formats.

However, the use of XML also poses difficulties. First is the problem of expressing integrity constraints. XML Schema [Wor00] allows one to specify a restricted class of structural constraints along with the XML analogs of referential and key constraints. Nevertheless, the applications considered here require a larger class of constraints than can be accommodated in the XML Schema model. Powerful declarative languages such as XQuery and Schematron [Jel] could be used to express these constraints, but these languages are too expressive to permit the generation of efficient incremental checks. The problem of incremental evaluation of integrity constraints has received attention in the context of relational [GT94] and object-oriented databases [JQ92], but has yet to be investigated in the context of XML.

This paper describes ΔX, a declarative language for expressing integrity constraints on XML data and a compiler that generates incremental constraint-checking code from these constraints. The ΔX language is based on XML Schema, but allows a broader range of constraints to be expressed by allowing *value-based* constraints. The ΔX compiler translates constraints to incremental constraint-checking code. The translation is via predicate logic: the ΔX language maps to a logic that is expressive enough to handle a large class of constraints, but restrictive enough to enable efficient incremental checking. The use of logic as an intermediate language helps new classes of constraints and new target languages to be accommodated easily.

There are three main contributions of this work. First is the ΔX constraint language. The main novelty here is that the language was designed specifically to support incremental constraint checking. Second is the architecture of the ΔX compiler; particularly the use of predicate logic as an intermediate language. Third is the collection of algorithms used for generating incremental constraint checking code from logical formulas.

2 Network Provisioning

Network provisioning is one of the jobs of network management. Figure 1 shows how provisioning occurs through network management processes. The figure shows a *Network Node* that could be any kind of network element, such as a router or bridge. The *Manager* monitors and controls the Network Node with the help of the *Agent*. The Manager and Agent communicate via a network management protocol such as SNMP (Simple Network Management Protocol). The Manager includes a graphical user interface and event-handling code that translates GUI events to update or modification requests. The Agent performs actions based on these requests. These actions include the validation of an update to ensure that it will not cause data integrity constraints to be violated, the update to the persistent store, the updating of derived data, and the sending of notifications to other network elements.

We are interested in systems in which provisioning data is represented as an XML document.

According to our model of provisioning, this document is maintained in a persistent store on the Agent. Provisioning requests from the Manager take the form of XML data update commands.

3 Example Application

ΔX is used at Lucent Technologies for the provisioning of wireless telephone networks. We describe here an example application in which the data model and database APIs are simplified.

3.1 Data Schema

The provisioning data describes configuration information about regions within a network. We now describe the XML Schema for this application. Two conventions used in this application simplify both querying and update. First, every element occurs in the schema exactly once. Second, every element node has an identifying attribute, called `nodeid`, which codes the path from the root to that element. This convention allows direct addressing of nodes (as is done in LDAP), thus simplifying the update API described below. There are built-in functions Tag and Parent that allow the node's tag and the `nodeid` of its parent to be calculated from the `nodeid`, which allows several optimizations in query processing and integrity constraint simplification.

The main elements of the application are regions and configurations. In the schema a `Region` element contains within it a list of `AdjacentRegion` elements. An `AdjacentRegion` element has a `regionid` attribute that is a foreign key back to some `Region` element. A `Region` element also has a `configurationid` attribute, which is a foreign key into a `Configuration` element lying elsewhere in the same document.

Appendix A shows a fragment of a schema. We omit many additional attributes found in the actual application, as well as the XML Schema constraints that guarantee that `configurationid` points to the `nodeid` of a `Configuration` and that `regionid` points to the `nodeid` of a region. An instance of the schema would include elements such as the following:

```
<Region nodeid=... color=... configurationid=... accesscode=...>
        <AdjacentRegions nodeid=...>
                <AdjacentRegion nodeid=... regionid=.../>
                <AdjacentRegion nodeid=... regionid=.../>
        </AdjacentRegions>
</Region>
<Configuration nodeid= ... configurationtype=...>
```

3.2 Data Interfaces

For ΔX, two data interfaces are important. First is the interface used by applications to perform database updates. We call this the *application-level update interface*. A call by an application to this interface causes incremental integrity constraint checks to be performed. If a check fails, an error code is returned; otherwise the requested update is performed. The ΔX compiler assumes the following simple API for application-level updates:

- *Create*(*nodeid*, *attribute-value list*) Creates a new element with the given attribute values. The nodeid describes the exact position in the document that the insert should occur.
- *Delete*(*nodeid*) Deletes the node with the given nodeid, along with all of its children.
- *Set*(*nodeid*, *attribute-value list*) Replaces the attributes in the attribute-value list with the corresponding values.

The other data interface relevant to ΔX is the one used in the ΔX-generated code to access data in the document. For XML, two standard APIs are the DOM [Wor98], which provides navigational access, and XQuery, which provides a query-based interface. Because in our target application documents are stored in a proprietary database, providing only navigational access, ΔX uses a variant of the DOM interface for reads. The compiler is designed so that changes to this interface, including the adoption of a query-based interface, will require only localized changes to the compiler.

4 The ΔX Constraint Language

Code generation in ΔX is driven by a schema, which could be given as a DTD or XML Schema. The schema is supplemented with additional integrity constraints in the ΔX language. The design of the ΔX integrity constraint language was guided by several goals: to have a *declarative* language stating only invariants of the document (as in traditional relational integrity constraints), to allow automated analysis and code generation, to express a wide variety of constraints, and to base the language on standards. We chose to have the ΔX constraint language extend XPath [CD99] and XML Schema [Wor00].

4.1 XPath

XPath allows one to express queries on an XML document. The evaluation of an XPath expression on a document yields a collection of nodes, a set of values, or a boolean. For the ΔX constraint language we use a simplified form of XPath having the following abstract syntax, where a ranges over element labels and attribute names, and s ranges over string constants:

$$
\begin{array}{lcl}
p & ::= & \epsilon \mid \emptyset \mid a \mid * \mid // \mid p/p \mid p \cup p \mid p[q] \\
q & ::= & p \mid p = p \mid p = s \mid q \wedge q \mid q \vee q
\end{array}
$$

Expression ϵ denotes the empty path, a denotes the element or attribute it names, $\emptyset$ denotes the empty set, $\cup$ denotes set union, / denotes concatenation, $*$ denotes (downward) wildcard, and // denotes the combination of wildcard and Kleene closure. In expression $p[q]$, q is a *qualifier*. If p denotes a set of values, then $p[q]$ denotes the subset of those values satisfying q. For the semantics of XPath expressions, see [Wad00].

4.2 XML Schema

In XML Schema one can define key and referential constraints. For example, the key constraint stating that the `nodeid` attrbitute uniquely identifies a `region` tag would be expressed by attaching the following to the root:

```
<xs:unique name="regionkey">
   <xs:selector xpath="//region"/>
   <xs:field xpath="@nodeid"/>
</unique>
```

The referential constraint stating that the `regionid` attribute of every `AdjacentRegion` points to some `nodeid` of a `Region` would be expressed by attaching the following to the `AdjacentRegions` element in the schema:

```
<xs:keyref name="adjregionref" refer="regionkey">
   <xs:selector xpath="./AdjacentRegion"/>
   <xs:field xpath="@regionid"/>
</xs:keyref>
```

However, many constraints occurring in provisioning applications go beyond the key and referential constraints supported in XML Schema. For example, one sees restrictions on the combinations of attribute values that can occur either within a node or between nodes (e.g., the `mincapacity` attribute must be below the `maxcapacity` attribute). There are also often restrictions on the target of a referential constraint, based on values of an attribute in the target (e.g., the `adjacentregion` must point to some region with `maxcapacity` attribute below 3). Finally, one often sees bounds on the cardinality of children based on some attribute value (e.g., the number of adjacent regions of any region must not exceed the value of the `maxadjacent` attribute of the document's root element).

ΔX handles these *valued-based* constraints by loosening the XML Schema restrictions on XPath expressions in constraints (e.g. allowing qualifiers) and by adding *restriction* and *cardinality* constraints. Restriction constraints allow one to state that an XPath expression must return empty. For example, the requirement that the minimum capacity of a region must not be above the maximum capacity would be expressed by attaching the following to the `region` element:

```
<forbidden name="regionCapacity">
   <target xpath="@mincapacity &gt;= @maxcapacity"/>
</forbidden>
```

XPath 1.0 is limited by the inability to do full joins between different elements in the document (e.g. to navigate from an AdjacentRegion to an attribute inside the Region it points to). XPath 2.0 [BBC+] and XQuery [BCF+] allow simple iteration constructs that bind a variable to the results of an XPath expression. We are extending our language to include a simple fragment of XQuery , allowing loops with `forbidden` assertions embedded in them. For example, the following constraint attached to the `region` element of the schema expresses that the `maxcapacity` of a region must be the same as the capacity of related regions.

```
<loop var="c" xpath=".">
   <loop var="d" xpath="c/AdjacentRegions/AdjacentRegion">
      <loop var="e" xpath="//Region[@nodeid=$d/@regionid]">
         <forbidden xpath="($e/@maxcapacity=$c/@maxcapacity)"/>
      </loop>
   </loop>
</loop>
```

Appendix B shows the syntax of the ΔX language as a DTD. The referential, key, and restriction constraints all map into the universal existential (Π_2) fragment of first-order logic (see the next section), a fragment that is both expressive enough for many application constraints but limited enough to admit efficient analysis. The cardinality constraints map into a restricted fragment of first-order logic with counting. For brevity, we focus on restriction and referential constraints in the next section.

5 Code Generation Scheme

The generation of incremental-validation code in ΔX proceeds in several steps (see Fig. 2). The first step is to translate the surface syntax into an internal logical representation that

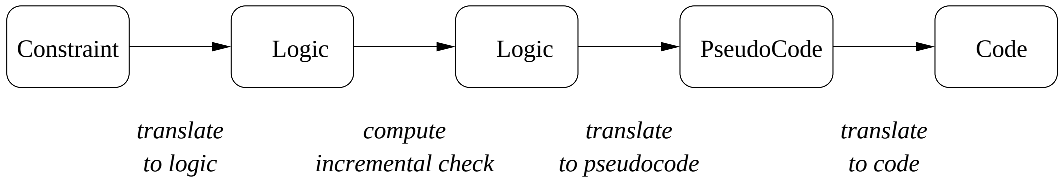

Figure 2: Code-generation Scheme

formula	ϕ	$::=$	$at \mid \phi_1 \vee \phi_2 \mid \phi_1 \wedge \phi_2 \mid \phi_1 \rightarrow \phi_2 \mid \phi_1 ? \phi_2, \phi_3 \mid \exists x \in C : \phi \mid \forall x \in C : \phi$
atomic	at	$::=$	$\mathit{true} \mid \mathit{false} \mid C(t) \mid \mathit{child}(t_1, t_2) \mid t_1\ \mathit{op}\ t_2$
term	t	$::=$	$ft \mid it \mid \phi ? t_1, t_2$
formula-free term	ft	$::=$	$k \mid \mathrm{Parent}(ft) \mid x \mid ft.@b$
integer term	it	$::=$	$\mathit{Count}(x \in C : \phi) \mid it + it \mid ft$
constant	k	$::=$	$i \mid b \mid s$

Figure 3: The Syntax of FOC Formulas

is more amenable to algorithmic manipulation. The logic can express both the constraints themselves and the parameterized constraints (formulas) that describe the incremental check for each update operation. The basic algorithms for generating these checks and simplifying them work completely at the logic level. In the next step, abstract pseudocode classes are generated that give a generic procedural implementation of an incremental check. In the final step, abstract pseudocode is translated into code in the desired target language. In this paper we shall focus on the first two phases.

5.1 Internal Representation

We require an internal representation of constraints that is sufficiently expressive, has good closure properties, and which admits tractable implementations (e.g. polynomial time in the size of the data). We therefore represent constraints using *first-order logic with counting over trees with data values* (FOC). Figure 3 shows the abstract syntax of FOC formulas, where x ranges over node variables, C ranges over element literals, and *op* ranges over the set $\{=, <, \leq, >, \geq, \neq\}$ of comparison operators. Also, i, b, and s range over integer, boolean, and string constants, respectively. The formula $\phi_1 ? \phi_2, \phi_3$ is logically equivalent to $(\phi_1 \rightarrow \phi_2) \wedge (\neg\phi_1 \rightarrow \phi_3)$. The term $\phi ? t_1, t_2$ evaluates to t_1 when ϕ holds, and otherwise t_2.

In defining the semantics of FOC we model an XML document D as a partial function that maps a `nodeid` to a tag and to a set of attribute/value pairs. D thus associates with each tag name C a finite collection of attribute/value pair sets in which the distinguished attribute `nodeid` is found in each set. We write $\overline{D}(C)$ for this collection.

The mapping $\overline{D}$ must be consistent with `nodeid` values in this sense: if $t \in \overline{D}(C)$ then 1) $\mathrm{Tag}(t.id) = C$, and 2) if $\mathrm{Parent}(t.id) = k$ for some id value k, and $\mathrm{Tag}(k) = C'$, then there exists a t' in $\overline{D}(C')$ such that $t'.id = k$. A D that is consistent is referred to as an *instance*.

The semantic function $\|\phi\|_{D,v}$ maps a formula or integer term to a boolean or integer, relative to a pair (D, v), where D is an instance and v is a valuation mapping variables to values. Figure 4 shows some clauses in the definition of the semantic function. The extension to the boolean operations and universal quantifiers is straightforward. We write $D, v \models \phi$ as an abbreviation for $\|\phi\|_{D,v} = \mathit{true}$.

$$
\begin{array}{lll}
\|\exists x \in C :\ \phi(x, \vec{y})\|_{D,v} & \stackrel{\text{def}}{=} & \|\phi(x,\vec{y})\|_{D,(v \cup (x,t))} = \text{true for some } t \text{ in } \overline{D}(C) \\
\|C(x)\|_{D,v} & \stackrel{\text{def}}{=} & \text{Tag}(x.id) = C \\
\|child(x_1, x_2)\|_{D,v} & \stackrel{\text{def}}{=} & \text{Parent}(v(x_2).id) = v(x_1).id \\
\|\tau(x_1)\ op\ \tau'(x_2)\|_{D,v} & \stackrel{\text{def}}{=} & \|\tau(x_1)\|_{D,v}\ op\ \|\tau'(x_2)\|_{D,v} \\
\|\tau(x)\|_{D,v} & \stackrel{\text{def}}{=} & \tau(v(x)) \quad (\tau \text{ a formula-free term}) \\
\|Count(x \in C : \phi(x,\vec{y}))\|_{D,v} & \stackrel{\text{def}}{=} & |\{t \in v(C) :\ \|\phi(x,\vec{y})\|_{D,v\cup(x,t)}\}|
\end{array}
$$

Figure 4: Some Clauses of the Semantics of *FOC* Formulas

Consider the simple constraint stating that the maximum capacity of every region is above its minimum capacity. This can be expressed in *FOC* by formula ϕ_1: $\forall r \in Region : r.maxcapacity \geq r.mincapacity$. As a more interesting example, consider the constraint stating that every two adjacent regions share the same maximum capacity. This can be expressed by *FOC* formula ϕ_2:

$$
\begin{array}{l}
\forall r \in Region : \forall s \in AdjacentRegions : (child(s, r) \rightarrow \\
\quad \forall t \in AdjacentRegion : child(t, s) \rightarrow \\
\quad\quad \forall w \in Region : (w.nodeid = t.regionid \rightarrow w.maxcapacity = r.maxcapacity)
\end{array}
$$

It is easy to see that the constraints described in the previous section can be translated into the logic above. In fact, all ΔX constraints other than cardinality constraints can be mapped into the *universal-existential fragment* (Π_2) of the logic. The Π_2 formulas have the form

$$\forall r_1 \in R_1 : \ldots \forall r_n \in R_n : \exists s_1 \in S_1 : \ldots \exists s_m \in S_m : \alpha$$

where α is quantifier-free.

5.2 Change analysis for *FOC*

An update operation, such as *Create*, has arguments $p_1, \ldots, p_n$, each of which becomes instantiated with either a nodeid or an attribute value. The goal of change analysis is to compute, given a formula $\phi(\vec{x})$ and an update operation T, the *incremental version* $\Delta(\phi, T)$ of ϕ, which is a formula $\phi'(p, \vec{x})$ such that for every t instantiating the parameters p of T, valuation v for $\vec{x}$, and instance D,

$$D, v \models \phi \ \Rightarrow\ (T(t)(D), v \models \phi \Leftrightarrow D,\, v \cup (p, t) \models \phi')$$

However, this condition does not uniquely define $\Delta(\phi, T)$. For example, consider formula ϕ_1 above and the operation $T(p) = Create(p)$, where p is a parameter list. Then the natural incremental version is clearly $region(p) \wedge p.maxcapacity \geq p.mincapacity$ but formula $(\forall r \in region : r.maxcapacity \geq r.mincapacity) \wedge (p.maxcapacity \geq p.mincapacity)$ satisfies the definition as well.

We are interested in defining a $\Delta(\phi, T)$ that will produce formulas that are efficient as possible. Two problems immediately arise. First, defined what "most efficient" means depends on a cost model for formulas, and would ideally also take into account a distribution over the values t that parameterize that update operation. The major heuristics incorporated into our algorithms can be justified by a simple cost model for query access plus some simple assumptions on the distributions of updates. Second, many rewritings that are clearly cost-reducing relative

$$
\begin{array}{lcl}
wp(\phi_1 * \phi_2, T) & \stackrel{\text{def}}{=} & wp(\phi_1, T) * wp(\phi_2, T) \quad (* \text{ a prop. connective}) \\
wp(\phi?\phi_1, \phi_2, T) & \stackrel{\text{def}}{=} & wp(\phi, T)?wp(\phi_1, T), wp(\phi_2, T) \\
wp(\exists x \in C : \phi, T) & \stackrel{\text{def}}{=} & C(p)?\exists x \in C : wp(\phi, T) \vee wp(\text{Sub}(x, p, \phi), T), \\
 & & \qquad \exists x \in C : wp(\phi, T) \\
wp(\forall x \in C : \phi, T) & \stackrel{\text{def}}{=} & C(p)?\forall x \in C : wp(\phi, T) \wedge wp(\text{Sub}(x, p, \phi), T), \\
 & & \qquad \forall x \in C : wp(\phi, T) \\
wp(child(t_1, t_2), T) & \stackrel{\text{def}}{=} & child(wp(t_1, T), wp(t_2, T)) \\
wp(t_1 \ op \ t_2, T)) & \stackrel{\text{def}}{=} & wp(t_1, T) \ op \ wp(t_2, T) \\
wp(\phi?t_1, t_2, T) & \stackrel{\text{def}}{=} & wp(\phi, T)?wp(t_1, T), wp(t_2, T) \\
wp(t, T) & \stackrel{\text{def}}{=} & t \quad (t \text{ a formula-free term or boolean constant}) \\
wp(Count(x \in C : \phi), T) & \stackrel{\text{def}}{=} & C(p)?Count(x \in C : wp(\phi, T) + (wp(\text{Sub}(x, p, \phi)?1, 0), \\
 & & \qquad Count(x \in C : wp(\phi, T)) \\
wp(t_1 + t_2, T) & \stackrel{\text{def}}{=} & wp(t_1, T) + wp(t_2, T)
\end{array}
$$

Figure 5: Weakest Precondition Calculation for Operation $T = Create(p)$

$$
\begin{array}{lcl}
wp(\exists x \in C : \phi, T) & \stackrel{\text{def}}{=} & C(p)?\exists x \in C : x.nodeid \neq p.nodeid \wedge wp(\phi, T), \\
 & & \qquad \exists x \in C : wp(\phi, T) \\
wp(\forall x \in C : \phi, T) & \stackrel{\text{def}}{=} & C(p)?\forall x \in C : x.nodeid \neq p.nodeid \rightarrow wp(\phi, T), \\
 & & \qquad \forall x \in C : wp(\phi, T) \\
wp(Count(x \in C : \phi), T) & \stackrel{\text{def}}{=} & C(p)?Count(x \in C : wp(\phi, T)) - (wp(\text{Sub}(x, p, \phi), T)?1, 0), \\
 & & \qquad Count(x \in C : wp(\phi, T))
\end{array}
$$

Figure 6: Weakest Precondition Calculation for Operation $T = Delete(p)$

to every reasonable cost model (e.g., the removal of a redundant clause) are not decidable in general. The problem of determining whether $\Delta(\phi, T)$ is equivalent to *true* is undecidable, even for the universal existential fragment. For the universal fragment, the problem is decidable, but intractable. Our algorithms thus rely on incomplete methods for removing redundant clauses even for universal formulas.

Although finding the *best* incremental version is undecidable, it is not difficult to find *some* incremental version. Given operation $T(p)$ and formula $\phi(\vec{x})$, a *weakest precondition of* $\phi(\vec{x})$ *with respect to* T, denoted $wp(\phi, T)$, is any formula $\phi'(p, \vec{x})$ such that, for any parameter value $t \in Tup$, document instance D, and valuation v on $\vec{x}$,

$$(T(t)(D), v \models \phi) \Leftrightarrow (D, v \cup (p, t) \models \phi')$$

Clearly every such ϕ' is an incremental version of ϕ'. Analogously for terms, $wp(\tau, T)$ is any term τ' such that

$$\|\tau\|_{T(t)(D),v} \ = \ \|\tau'\|_{D,v\cup(p,t)}$$

A simple inductive algorithm can be used to calculate $wp(\phi, T)$. The algorithms for computing $wp(\phi, T)$ for $T = Create(p)$ and $T = Delete(p)$ are shown in Figure 5 and Figure 6, respectively. In Figure 6, only the cases that differ from those in Figure 5 are shown. We write $\text{Sub}(x, p, \phi)$ for the formula obtained by substituting term p for free occurrences of variable x in ϕ.

Weakest preconditions can be thought of as a default case for incremental integrity checking. In fact, one can obtain an incremental check by generating $wp(\phi, T)$ and then simplifying with respect to ϕ. In practice, however, it is more useful to have a set of rules that generate incremental checks directly. This is done in ΔX by using recursive Δ-rules specific to the incremental-checking context, coupled with simplification algorithms for FOC.

Recursive Δ-rules capture basic subcases of incremental reasoning — the ones used by ΔX are analogous to the change propagation rules used in incremental view maintenance algorithms ([GM99]). The following rules (named $\exists_{Create}$, etc.) generate an incremental check that can be proved to be optimal under certain cost and update distribution assumptions, for any formula in the Π_2 fragment:

$$
\begin{array}{lll}
\exists_{Create}: & \Delta(\exists x \in C : \phi, \mathit{Create}(p)) & \stackrel{\text{def}}{=} \mathit{true} \\
\forall_{Create}: & \Delta(\forall x \in C : \phi, \mathit{Create}(p)) & \stackrel{\text{def}}{=} C(p)\,?\,\mathrm{Sub}(x, p, \phi), \\
 & & \qquad \forall x \in C : \Delta(\phi, \mathit{Create}(p)) \\
 & & \quad \text{(if } C \text{ does not appear in } \phi) \\
\exists_{Delete}: & \Delta(\exists x \in C : \phi(x, \vec{y}), \mathit{Delete}(p)) & \stackrel{\text{def}}{=} C(p)\,?\,\phi(p) \to \exists x \in C : x.\mathit{nodeid} \neq p.\mathit{nodeid} \wedge \phi, \\
 & & \qquad \exists x \in C : \Delta(\phi, \mathit{Delete}(p)) \\
 & & \quad \text{(if } C \text{ does not appear in } \phi) \\
\forall_{Delete}: & \Delta(\forall x \in C : \phi(x, \vec{y}), \mathit{Create}(p)) & \stackrel{\text{def}}{=} \mathit{true} \text{ (if } C \text{ does not appear in } \phi)
\end{array}
$$

The rule $(\exists, \mathit{Delete})$ can be justified using a cost model that assumes ϕ will often be false, rendering the need to check the conclusion of the implication unnecessary.

The following rules, used when C does appear in subformula ϕ, are heuristic, and no optimality result has been proved for them:

$$
\begin{array}{ll}
\Delta(\forall x \in C : \phi(x, \vec{y}), \mathit{Create}(p)) & \stackrel{\text{def}}{=} C(p)\,?\,\forall x \in C : \Delta(\phi, \mathit{Create}(p)) \wedge \mathit{wp}(\mathrm{Sub}(x, p, \phi), \mathit{Create}(p)), \\
 & \qquad \forall x \in C : \Delta(\phi, \mathit{Create}(p)) \\
\Delta(\bigvee_i A_i, \mathit{Create}(p)) & \stackrel{\text{def}}{=} \bigwedge_i \Delta(A_i, \mathit{Create}(p)) \vee \mathit{wp}(\bigvee_i A_i, \mathit{Create}(p)) \\
\Delta(\exists x \in C : \phi, \mathit{Delete}(p)) & \stackrel{\text{def}}{=} \mathrm{Sub}(x, p, \mathit{wp}(\phi, \mathit{Delete}(p))) \to \mathit{wp}(\phi, \mathit{Delete}(p))
\end{array}
$$

Simplification, both logical and cost-based, is done on the output of the algorithms. This step could at some point be deferred to an external optimizer (e.g., for XQuery). A more interesting optimization occurs in a preprocessing step, which attempts to transform formulas so that the major Δ-rules can be applied.

For example, consider the example formula ϕ_2 of Section 5.1. Computing $\Delta(\phi_2, \mathit{Create}(p))$ proceeds by first transforming ϕ_2 into the form:

$$
\begin{array}{l}
\forall w \in \mathit{region} : \forall s \in \mathit{AdjacentRegion} : (s.\mathit{regionid} = w.\mathit{nodeid} \to \\
\quad w.\mathit{maxcapacity} = \mathrm{Parent}(\mathrm{Parent}(s.\mathit{nodeid})).\mathit{maxcapacity}
\end{array}
$$

Then, applying rule $\forall_{Create}$, we obtain

$$
\begin{array}{l}
\Delta(\phi_2, \mathit{Create}(p)) = \\
\quad \mathit{AdjacentRegion}(p)? \\
\qquad \forall w \in \mathit{Region} : (p.\mathit{regionid} = w.\mathit{nodeid} \to \\
\qquad\qquad\qquad w.\mathit{maxcapacity} = \mathrm{Parent}(\mathrm{Parent}(p.\mathit{nodeid})).\mathit{maxcapacity}), \\
\qquad \mathit{region}(p)? \\
\qquad\quad \forall s \in \mathit{AdjacentRegion} : (s.\mathit{regionid} = p.\mathit{nodeid} \to \\
\qquad\qquad\qquad\qquad p.\mathit{maxcapacity} = \mathrm{Parent}(\mathrm{Parent}(s.\mathit{nodeid})).\mathit{maxcapacity}), \\
\qquad\quad \mathit{true}
\end{array}
$$

The simplification rules of ΔX make use of reasoning that is specific to trees. The simplification above, for example, takes into account both a general fact about trees (the functional nature of the Parent function) and information about the Schema (that the grandparents of AdjacentRegion elements are all region elements). ΔX has a general scheme for inferring functional relationships (e.g. $y = \text{Parent}(\text{Parent}(x))$) and using them to eliminate loops.

6 Compiler Design

The ΔX compiler has phases corresponding to the steps shown in Figure 2. The first phase includes not only parsing of the schema (including integrity constraints), but also type-checking the constraints against the data model, and translating each constraint to a formula of *FOC*.

The compiler is implemented in Java. The ΔX compiler has a top-level package plus one package for each of the languages it supports: the schema language, the constraint language, first-order logic, and an abstract programming language. Each language package contains one class for each language construct. For example, in the logic package, there is a class for each kind of term, atomic proposition, quantifier, propositional connective, etc. Using the logic package one can construct an object corresponding to a logical formula. Roughly, this object represents the formula's abstract syntax tree.

The compiler is designed to be easily adapted to various target languages and database APIs. Objects built using the code package represent programs in a generic imperative language with high-level operations for data access. For program structure, the code package contains classes such as `Program`, `Method`, `Assign`, and `Conditional`. For data access, the code package contains classes such as `GetAttr`, `GetParent`, and `IsChildOf`. From an object in the code package one obtains target code by defining appropriate `toString` methods. These methods encapsulate knowledge of the target language and database API.

The compiler contains much code that manipulates logical formulas. For example, there are methods that implement substitution and simplification. This code is tedious to write because for many formulas the method will act as a homomorphism. For example, in implementing the method `subst`, which substitutes a term for a variable within a formula, we might have

$$subst(t, x, *(\phi_1, \dots, \phi_n)) = subst(t, x, \phi_1) * \cdots * subst(t, x, \phi_n)$$

for many logical operators $*$. The compiler uses a technique that allows the implementation of these cases to be omitted. First, a class is defined that allows formulas and the terms within them to be treated as generic trees. This class is a superclass of the classes used to represent formulas and terms; every object of the class is required to support a method `buildTree` and a method `getChildren`. For example, the `buildTree` method of the `Conjunction` class will build a formula $\phi_1 \wedge \phi_2$ from ϕ_1 and ϕ_2, while the `getChildren` method will return ϕ_1 and ϕ_2 given $\phi_1 \wedge \phi_2$. Then, for every formula manipulation function to be implemented, a single method is defined that applies the function in the homomorphic manner shown above. Additional code must be written only for the cases in which the function does not work in the homomorphic manner.

7 Related Work

XML parsers and the XML editing tools built on top of them allow the validation of documents with respect to DTDs and XML Schemas. These editing tools do not support efficient incremental validation of XML Schema constraints on update, nor is this feature currently availability emerging XML database products. No ability to author and check constraints beyond those given by XML Schema is available in off-the-shelf parsers, although research systems such as Schematron [Jel] provide support for non-incremental validation of a much more powerful constraint language.

Within the database research community, integrity-constraint checking and the closely-related topic of view maintenance have received considerable attention. The book [GM99] discusses work on both of these topic in the relational setting, while [JQ92] studies the issue in the context of object-oriented databases. The approaches to incremental integrity checking in [GM99] tend to either focus on very restricted classes of constraints (e.g., classical referential), or on the whole of SQL, for which efficient incremental checking is impossible. The topic of XML integrity-constraint checking is much more recent, but there has still been a considerable amount written on the topic. [FL00b, FL00a] study static analysis problems associated with several classes of XML-Schema constraints. [DT01] defines a class of constraints (called "SXIC" constraints) that extend the XML Schema referential constraint model significantly. However, [DT01] does not discuss incremental-checking for SXIC constraints, only analysis and optimization questions. Incremental evaluation of the XML transformation language XSLT is discussed in [VL02].

Acknowledgments: We would like to thank Robert Sojka and Arun Sankisa for their help in the ΔX effort.

References

[BBC+] Anders Berglund, Scott Boag, Don Chamberlin, Mary Fernandez, Michael Kay, Jonathan Robie, and Jérôme Siméon. *XML Path Language (XPath) 2.0*. World Wide Web Consortium (W3C). http://www.w3.org/TR/xpath20/.

[BCF+] Scott Boag, Don Chamberlin, Mary Fernandez, Jonathan Robie, Jérôme Siméon, and Mogur Stefanescu. *XQuery 1.0 : An XML Query Language*. World Wide Web Consortium (W3C). http://www.w3.org/TR/2002/WD-xquery-20020430/.

[CD99] James Clark and Steve DeRose. *XML Path Language (XPath)*. W3C Recommendation, November 1999.
`http://www.w3.org/TR/xpath`.

[DT01] Alin Deutsch and Val Tannen. Containment for Classes of Xpath Expressions under Integrity Constraints. In *KRDB*, 2001.

[FL00a] Wenfei Fan and Leonid Libkin. Finite Implication of Key and Foreign Key constraints for XML Data. Technical Report TUCIS-TR-2000-003, Dept. of Computer and Information Sciences, Temple University, 2000.

[FL00b] Wenfei Fan and Leonid Libkin. Finite Satisfiability of Key and Foreign Key constraints for XML data. Technical Report TUCIS-TR-2000-002, Dept. of Computer and Information Sciences, Temple University, 2000.

[GM99] Ashish Gupta and Inderpal Mumick. *Materialized Views: Techniques, Implementations, and Applications*. MIT Press, 1999.

[GT94] Tim Griffin and Howard Trickey. Integrity Maintenance in a Telecommunications Switch. *IEEE Data Engineering Bulletin, Special Issue on Database Constraint Management*, 1994.

[Jel] Rick Jellife. *The Schematron : An XML Structure Validation Language using Patterns in Trees*. http://www.ascc.net/xml/resource/schematron/schematron.html.

[JQ92] H.V. Jagadish and Xiaolei Qian. Integrity Maintenance in an Object-oriented Database. In *VLDB 92*, 1992.

[VL02] Lionel Villard and Nabil Layaïda. An Incremental XSLT Transformation Processor for XML Document Manipulation. In *Proc. 12th Int'l World Wide Web Conf. (WWW'12)*, pages 474–485, 2002.

[Wad00] Philip Wadler. Two Semantics for XPath. Technical report, Computing Sciences Research Center, Bell Labs, Lucent Technologies, 2000.

[Wor98] World Wide Web Consortium (W3C). *Document Object Model (DOM) Level 1 Specification*, 1998. http://www.w3.org/TR/WD-DOM-971009.

[Wor00] World Wide Web Consortium (W3C). *XML Schema Part 0: Primer*, 2000. http://www.w3.org/TR/xmlschema-0/ .

A Example Schema Fragment

```
<element name="Region">
   <complexType>
      <element name="AdjacentRegions">
         <complexType>
            <sequence>
               <element name="AdjacentRegion" maxOccurs="unbounded"/>
                  <complexType>
                     <attribute name="regionid" type="xs:string"/>
                  </complexType>
               </element>
            </sequence>
         </complexType>
      </element>
      <attribute name="nodeid" type="xs:string"/>
      <attribute name="configurationid" type="xs:string"/>
      <attribute name="color" type="xs:string"/>
      <attribute name="accesscode" type="xs:string"/>
      <attribute name="maxcapacity" type="xs:integer"/>
      <attribute name="mincapacity" type="xs:integer"/>
   </complexType>
</element>

<element name="Configuration" >
   <complexType>
      <attribute name="nodeid" type="xs:string"/>
      <attribute name="configurationtype" type="xs:string"/>
   </complexType>
</element>
```

B Syntax of the ΔX constraint language

```
<!ELEMENT constraints (unique|field|keyref|forbidden|count|loop)*>
<!ELEMENT unique (field)*>
  <!ATTLIST unique name ID #REQUIRED>
<!ELEMENT field EMPTY>
  <!ATTLIST field xpath CDATA #REQUIRED>
<!ELEMENT keyref (target)*>
  <!ATTLIST keyref name ID #REQUIRED>
<!ELEMENT target (field)*>
  <!ATTLIST target name ID #REQUIRED>
<!ELEMENT forbidden (field)>
  <!ATTLIST forbidden name ID #REQUIRED>
<!ELEMENT count (field)>
  <!ATTLIST count name ID      #REQUIRED
             op     (equal | less | less-equal | greater | greater-equal) #REQUIRED
             value  NMTOKEN #REQUIRED>
<!ELEMENT loop (unique|field|keyref|forbidden|count|loop)*>
  <!ATTLIST loop var  NMTOKEN #REQUIRED
               xpath  CDATA   #REQUIRED>
```