On Guard: Producing Run-Time Checks from Integrity Constraints

Michael Benedikt and Glenn Bruns

Bell Labs, Lucent Technologies

Abstract. Software applications are inevitably concerned with data integrity, whether the data is stored in a database, files, or program memory. An *integrity guard* is code executed before a data update is performed. The guard returns "true" just if the update will preserve data integrity. The problem considered here is how integrity guards can be produced automatically from data integrity constraints. We seek a solution that can be applied in general programming contexts, and that leads to efficient integrity guards. In this paper we present a new integrity constraint language and guard generation algorithms that are based on a rich object data model.

1 Introduction

Every programmer understands the issue of data integrity. In database applications, updates must be checked before they are applied to prevent data corruption. In object-oriented programs, method parameters must be checked to ensure that object attributes are not corrupted. In networking, updates to configuration data and routing tables must be checked.

Ideally, data integrity would be ensured in advance through program verification using static analysis, model checking, or theorem proving. However, large programs and unbounded data values are problematic for model checkers, while theorem provers generally require human guidance. More importantly, many programs accept data from the environment, and this data cannot be checked before run time, even in principle. An alternative to preventing bad updates is to monitor at run-time for the presence of corrupted data. An issue with this approach is how data that has been found to be corrupted can be "repaired" (see e.g., [1]).

In this paper we focus on another run-time approach: the *integrity guard*. An integrity guard is a program that is executed just before a data update is performed. If the guard returns "true" then the update is guaranteed to cause no data corruption. It is desirable that a guard be *exact* – it should return true if and only if the update will not corrupt the data. A guard should also be efficient – it should not interfere with application performance requirements. Guards are often used with constraints that are invariants. When this is the case it is desirable that a guard be *incremental*: it can assume that prior to the update the data is valid. The *guard generation problem* is to automatically generate an exact, incremental, and efficient guard from a data schema, a data update language, and set of data integrity constraints.

Much work on the guard generation problem comes from the database community, where it is known as the "integrity constraint maintenance problem". A related and more general problem is "view maintenance" – recomputing the result of a query as the input data is updated.

Our goal is to make guard generation useful for general-purpose programming. The database community has concentrated on the guard generation problem primarily for the relational data model and for constraint languages whose expressiveness subsumes first-order logic (see Section 5 for details). However, common programming data structures can be hard to model relationally. And in a general programming context, we wish to generate imperative guards, not relational queries. Furthermore, guards generated from arbitrary first-order constraints may have prohibitive execution cost.

Our approach is to use an object data model in which program data structures can be naturally expressed. We use an XPath-based constraint language that can express classical functional and inclusion dependencies and data restriction constraints, but with expressiveness limited enough to ensure that generated guards are inexpensive to evaluate (in particular, much less expensive than for traditional relational query languages). By using XPath we also gain in readability and user-familiarity. Finally, our architecture for guard generation allows guards to be produced for a wide range of data infrastructure.

Thus the main contributions of our work are a) a constraint language that is expressive while having advantages for generation and evaluation of guards b) algorithms for generating low-cost guards and c) a modular implementation framework that allows guard generation in multiple data storage paradigms. The language and algorithms we describe have been implemented as part of the Delta-X system at Lucent. This system is used to generate production code in several complex network management applications.

The paper is organized as follows. Section 2 presents our data model, update model, and constraint language, and gives properties of the constraints that will be useful in guard generation. Section 3 defines formally the incremental precondition problem in general, and presents the high-level structure of our procedures. Section 4 gives a detailed look at guard-generation. Section 5 overviews related work and discusses the implementation and applications of the framework, including experience and open issues.

2 Specifying Data Integrity

In this section we present a simple object data model, a set of data update operations, and our language for expressing constraints.

2.1 Data Model

A *data signature* is a set $A = \{a_1 \ldots a_n\}$ of unary function symbols, where each symbol a_i in A is either of *integer type* or *node type*, and a set *Classes* of class names. An *tree* t of this signature has the form

$$t = (nodes, class, I, child)$$

where *nodes* is a finite set, *class* is a mapping from *nodes* to *Classes*, I is a function interpreting members of A of integer type as functions from *nodes* to the integers, and members of A of node type as functions from *nodes* to *nodes*. Function *child* maps each node n to a sequence $m_1 \dots m_k$ of distinct nodes, such that the binary relation defined by $m \in \mathit{child}(n)$ forms a tree. For a_i a symbol of A we define $\mathit{type}(a_i)$ to be the integers if a_i has integer type, and *nodes* if a_i has node type.

This data model is a simplified object model – nodes represent objects, function symbols in A represent attributes, and members of *Classes* represent class names. The model is simplified in that our model captures only attributes of integer and pointer type. Also, we capture only a weak notion of class in our model, as all objects share the same set of attributes. The absence of an attribute in a class can be modeled through the use of distinguished attribute values. Finally, we do not model class inheritance.

Running Example. Figure 1 shows an example tree that describes a network configuration for routing software in a telecommunication system. The tree represents regions in the network, with attributes storing the minimum and maximum call capacity of each region and the children of a region representing the regions' neighbors. In the figure each box represents an object, with the object's class at the top and its attributes listed below. The system's call-processing component reads this data in order to route calls, while the provisioning component updates the data in response to commands from a technician.

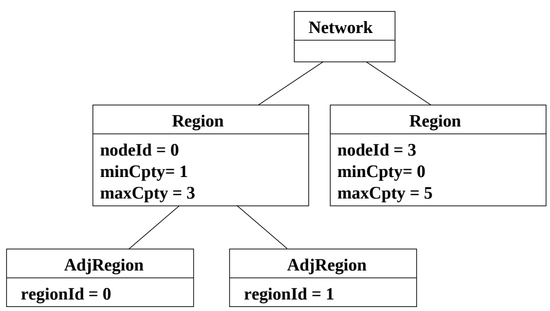

Fig. 1. Data for Running Example

We provide two basic update operations on trees, each of which is parameterized. A create operation $\mathit{Create}(n, C, U)$ for tree t has as parameters a node n of t, a class name C of *Classes*, and elements $U =< u_1, \dots, u_n >$, with each u_i in $\mathit{type}(a_i)$. The effect of applying $\mathit{Create}(n, C, U)$ to t is that a fresh node n' is added to $\mathit{nodes}(t)$, function *class* is updated so that $\mathit{class}(n') = C$, each function a_i is updated so that $a_i(n') = u_i$, and function *child* is extended so that $\mathit{child}(n)$ has additional, final element n'. A delete operation $\mathit{Delete}(n, C)$ for t has as parameters a leaf node n of t and a class name C of *Classes*. The effect of applying $\mathit{Delete}(n, C)$ to t is that node n of class C is removed from $\mathit{nodes}(t)$.

2.2 Constraint Language

We present a two-tiered language, consisting of an *expression language* and an *assertion language*. The expression language allows one to define sets of nodes in a data tree, while the assertion language consists of statements about sets of nodes.

Our expression language is based on XPath [5], an XML standard that allows one to express queries on an XML document. Evaluating an XPath expression on a tree node yields a set of nodes, or a set of values, or a boolean. In Delta-X we use a simplified form of XPath having the following abstract syntax, where $\propto$ ranges over $\{=, <, \leq, >, \geq\}$, *axis* ranges over $\{\leftarrow, \leftarrow^*, \rightarrow, \rightarrow^*, \downarrow, \downarrow^*, \uparrow, \uparrow^*\}$, a ranges over attribute names, C ranges over class names, and k ranges over integers:

$$E ::= \epsilon \;\Big|\; / \;\Big|\; @a \;\Big|\; \; axis \;\Big|\; C \;\Big|\; E_1 \cup E_2 \;\Big|\; [q] \;\Big|\; E_1/E_2$$
$$q ::= E \;\Big|\; E_1 \propto \; E_2 \;\Big|\; E \propto \; k \;\Big|\; class = C \;\Big|\; q_1 \wedge q_2 \;\Big|\; q_1 \vee q_2$$

We now briefly describe, in the style of [20], the meaning of expressions by defining the set $E(n)$ of nodes "returned" by expression E relative to a node n of tree t. Expression ϵ returns $\{n\}$. Expression / returns the root node of t. Expression $@a$ returns the singleton set containing the value of attribute a of n. Expression *axis* returns the nodes related to n by the axis (for example, $\downarrow$ returns the children of a node, $\downarrow^*$ the descendants, and $\rightarrow$ the right siblings). Expression C returns the children in class C of a node. Expression $E_1 \cup E_2$ returns the union of $E_1(n)$ and $E_2(n)$. Expression $[q]$ returns $\{n\}$ if *qualifier* q returns false, and $\emptyset$ otherwise. Expression E_1/E_2 returns the functional composition of E_1 and E_2 (in other words, the union of what E_2 returns relative to each of the nodes in the set E_1 denotes).

A qualifier denotes a node predicate. Qualifier E holds of a node n in tree t iff $E(n)$ is non-empty. Qualifier $E_1 \propto E_2$ holds of n iff $E_1(n)$ contains an element n_1 and $E_2(n)$ contains an element n_2 such that $n_1 \propto n_2$. Similarly, $E_1 \propto k$ holds of n iff $E_1(n)$ contains an element n_1 such that $n_1 \propto k$. Qualifier $class = C$ holds of n iff $class(n) = C$. Qualifiers $q_1 \wedge q_2$ and $q_1 \vee q_2$ provide logical conjunction and disjunction.

Assertions are built up from expressions. Our assertion language supports three types of assertions (or "constraints"). A *restriction constraint* $Never(E)$ asserts that for every node in a data tree, E returns empty on that node.

A *referential constraint* $Reference(E)$: $Source(E_1 \dots E_n), Target(F_1 \dots F_n)$ asserts that for every node n, such that E evaluated at the root satisfies n, there is another node n' in the tree such that for each i, $E_i(n)$ has nonempty intersection with $F_i(n')$.

A *key constraint* $Key(E)$: $Fields(F_1 \dots F_n)$ asserts that for any two distinct nodes n_1, n_2 returned by E at the root of a data tree, there is some i such that $F_i(n_1)$ is distinct from $F_i(n_2)$: that is, a node can be uniquely identified by the values of $F_1, \dots, F_n$.

We write L_{XP} for the language obtained from this assertion language by taking expressions from XPath. L_{XP} allows one to express a wide range of program invariants.

Example. Returning to the network configuration example, one can express that the minimum capacity of a region must be above the maximum capacity of adjacent regions by

$$\begin{aligned}&\mathit{Never}([\mathit{class} = \mathit{Region} \wedge \\ &\quad @\mathit{minCpty} \leq \mathit{AdjRegion}/@\mathit{regionId}/@\mathit{maxCpty}])\end{aligned}$$

One can express that a region ID uniquely identifies a region node by

$$\mathit{Key}(\downarrow^* /\mathit{Region}) : \mathit{Fields}(@\mathit{regionId})$$

One can express that the region ID of every adjacent region points to some node ID of a region by

$$\mathit{Reference}(\downarrow^* /\mathit{AdjRegion}) : \mathit{Source}(@\mathit{regionId}), \mathit{Target}(@\mathit{nodeId})$$

L_{XP} is strictly more expressive then the standard key and referential constraints of relational and XML data. On the other hand, evaluation of the language is tractable:

Theorem 1. *The problem of evaluating a constraint $\phi \in L_{XP}$ on a tree t can be solved in polynomial time in $|\phi|, |t|$, on a machine that can iterate through t with unit cost for each child, sibling, or parent navigation step.*

The proof follows from the fact that evaluating a XPath expression E can be done in time $|E||t|^2$. This quadratic bound requires a refinement of an argument in [8]. For now we sketch a simpler argument that gives a bound of $|E||t|^3$, and which will be relevant to our later algorithms. An XPath expression can be translated in linear time to a first-order formula in which every subformula has at most three free variables. The evaluation of such formulae on data trees is well known to be in cubic time ([11]) using a bottom-up algorithm. The polynomial bounds now follow easily. A more detailed look at this translation and its target logic is presented in Section 4.

A L_{XP} constraint is called *local* if the initial expression is of the form $\downarrow^*$ $/E$, and all other expressions do not involve navigation of axes or following node-valued (i.e. pointer) attributes. The second and third example constraints above are local. For local constraints, we can get extremely efficient bounds on verification: linear for *Never*, quadratic for all others.

3 Computing Guards from Integrity Constraints

Imagine a system in which condition ϕ currently holds and must continue to hold. To ensure that a state update will not take the system to a state not satisfying ϕ, we want to perform a check such that 1) the check will succeed iff applying the update would leave the system in a state satisfying ϕ, and 2) the check will be inexpensive to perform. We now formalize the problem of finding such a check.

For simplicity we will for a moment treat preconditions in a purely semantic way. Assume a set S of states. Let $op : S \to S$ be a state update operation and let $\phi : S \to Bool$ be a state predicate. The *weakest precondition* [7] of ϕ and op, written $wp(op, \phi)$, is the predicate that holds of s exactly when $\phi(op(s))$ holds, for all s in S. A predicate ψ is an *incremental precondition* of ϕ and op if $\phi(s) \Rightarrow (\psi(s) \Leftrightarrow wp(op, \phi))$ holds for all s in S. In other words, ψ is a predicate that acts like $wp(op, \phi)$ for all states in which ϕ holds. Trivially, $wp(op, \phi)$ is an incremental precondition for op and ϕ, and so is the predicate $\neg\phi \vee wp(op, \phi)$. We are interested in incremental preconditions that can be evaluated cheaply, and generally we can expect no relationship between the logical strength of a predicate and its cost. For example, $wp(op, \phi)$ is not necessarily more costly than $\neg\phi \vee wp(op, \phi)$.

In putting these ideas into practice we need languages to describe updates and properties of states. In our case we actually need two property languages: a specification language in which to describe constraints, and an executable target language in which to describe incremental preconditions. Our computational problem is then: given a constraint in the specification language and an operator in the update language, compute a minimal cost incremental precondition in the target language. An issue is whether an incremental precondition exists in the target language for each constraint in the specification language (see [2]).

In Delta-X the specification language is L_{XP} and the update language is the one defined in Section 2, consisting of create and delete operations. These updates have parameters that are instantiated at runtime; we want to generate incremental preconditions with the same parameters.

In producing guards (i.e. incremental preconditions in an executable target language) from L_{XP} constraints, we work in multiple steps using intermediate languages. Delta-X supports several target language and environments (e.g. Java code storing objects within main memory, or C++ code storing objects within a relational database), so we translate constraints to guards via an intermediate language named DIL. This language has the basic control structures of an imperative language plus data types based on our data model.

Another intermediate language is called for because DIL is unsuited to the simplification of guards. We first produce the guards in a first-order logic on trees. This logic, called FOT, is powerful enough to express both the constraints of L_{XP} and the generated guards. The basic precondition generation algorithms are easy to express as transformations on FOT, and as a declarative language it provides a good framework for simplification.

Hence we compute guards in four steps, as shown in Figure 2. We first translate a L_{XP} constraint ϕ into a formula ψ_0 of FOT (see Section 4.1). Next we generate an incremental precondition of ψ_0 within FOT, and then simplify to obtain formula ψ_1. In the next step we translate ψ_1 into a DIL program, which is finally translated into a program in the target programming language. Before going into these steps in detail, we state some results about the algorithm as a whole.

Theorem 2. *For any L_{XP} constraint and any update operation, let ψ be the*

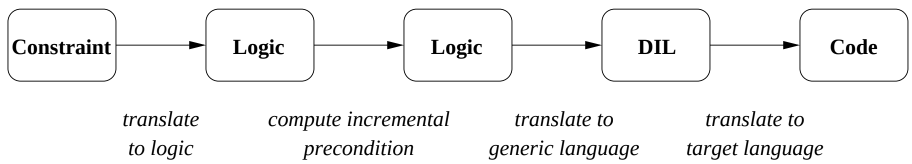

Fig. 2. Code-generation Scheme

result of the algorithm shown in Figure 2.

- *ψ runs in time polynomial in the data (where atomic operations of the target language are assumed to have unit cost). In fact, we can give the same bounds for ψ as we can for evaluation of constraints in Theorem 1 – e.g. quadratic in $|t|$ for Never constraints.*
- *If ϕ is a local constraint, then ψ can be evaluated in: constant time for a Never constraint, linear time for a Key constraint, and linear time for a Reference constraint for Create operations.*

On the other hand, there are limits to what one can hope to accomplish for any incremental precondition algorithm:

Theorem 3. *The problem of determining, given a ϕ in L_{XP} and an update operation, whether or not there is an incremental precondition of ϕ with a given quantifier rank, is undecidable. If ϕ consists of only Never constraints, then the problem is decidable but NP-Hard.*

The proof uses the undecidability of the implication problem for XML keys and foreign keys, the decidability of existential first-order logic over graphs, and the intractability of conjunctive query satisfaction.

4 Algorithms

4.1 Translating Constraints to Logic

The first step of our translation is from L_{XP} constraints to FOT formulas. The syntax of FOT formulas and terms is as follows, where C ranges over $Classes$, k over integers, $axis$ over $\{\downarrow, \downarrow^*, \uparrow^*, \rightarrow, \rightarrow^*\}$, and $\propto$ over $\{=, <, \leq, >, \geq\}$:

$$\begin{aligned}
\phi ::= {} & \forall x \in C : \phi \;\Big|\; \exists x \in C : \phi \;\Big|\; \phi_1 \wedge \phi_2 \;\Big|\; \phi_1 \vee \phi_2 \;\Big|\; \phi_1 \Rightarrow \phi_2 \;\Big|\; \neg\phi \;\Big| \\
& class(t) = C \;\Big|\; t_1 \, axis \, t_2 \;\Big|\; t_1 \propto t_2 \\
t ::= {} & x \;\Big|\; t.a \;\Big|\; k \;\Big|\; parent(t)
\end{aligned}$$

A formula is interpreted relative to an instance t as follows. The logical symbols have their expected meaning. Formula $class(t) = C$ holds if t represents a node of class C. Formula $t_1 \downarrow t_2$ holds if the node represented by t_1 is a child of the node represented by t_2. Other formulae of the form $t_1 \, axis \, t_2$ are interpreted similarly. For example, for axis $\downarrow^*$, t_1 must be a descendent of t_2, and for axis $\rightarrow$, t_1 must be the right sibling of t_2. Formula $t_1 = t_2$ holds if the terms represent the same integer values or the same node values. Other formulae of the form $t_1 \propto t_2$ are interpreted similarly.

The term x is a node variable, the term $t.a$ represents the a attribute of the node represented by t, the term k represents integer value k, and the term $parent(t)$ represents the parent of the node represented by t.

We translate a constraint in L_{XP} to a closed formula of FOT in two stages. First, the XPath expressions within the constraint are each translated into open formulas. Second, the constraint itself is translated, with the XPath-related formulas used as parameters.

Example. The *Never* constraint of our running example translates to:

$$\forall x \in Region: \ \forall y \in AdjRegion: \ y \downarrow x \Rightarrow$$
$$\forall z \in Region \ y : regionId = z.nodeId \Rightarrow z.maxCpty < x.minCpty$$

In translating an XPath expression to FOT, we obtain a formula containing free variables x and y. The formula represents a function from an input node x to an output node y. Because of space limitations we cannot present details of the translation. One issue is that an XPath expression can return nodes of different classes. Since our logic provides only for quantification over nodes of a single class, translation requires us to bound the set of classes in which a variable can occur. We do a static analysis to conservatively estimate which classes a particular variable may range over. The analysis can be made more precise in the presence of schema information, such as a DTD.

This translation satisfies two properties. First, ϕ has no subformula containing more than three free variables. As explained in the proof of Theorem 1, this guarantees low evaluation complexity. Second, ϕ has *limited-alternation*. This means that no universal quantifier is found within the scope of a existential quantifier, no negation symbols are present, and in an implication $\psi_1 \rightarrow \psi_1$ the formula ψ_1 contains no universal quantifications and no implications. Precondition algorithms are much simpler for this class.

The translation of assertions is a straightforward implementation of the semantics. This translation produces formulas that are limited-alternation, while our translation of expressions produces formulas that are both limited-alternation and within the three-variable fragment.

4.2 Computing Incremental Preconditions

We now present our incremental precondition algorithm, which accepts as input an update operation and a specification in FOT, and produces an incremental precondition as a formula of FOT.

As a first step towards incremental precondition generation we have a simple inductive algorithm $f_{wp}(op, \phi)$ that calculates a weakest precondition for ϕ under *op*. Weakest preconditions can be thought of as a default case for incremental integrity checking. In fact, one can obtain an incremental precondition by computing $f_{wp}(op, \phi)$ and then simplifying, using ϕ as an axiom. Instead, we begin with a set of rules that generate incremental checks directly, thus ensuring that the most important simplifications are performed. The algorithms for computing $f_{wp}(op, \phi)$ for Create and Delete operations are shown in Figures 3 and 4. Only the cases that differ from those of Figure 3 are shown in Figure 4. We write $\phi[x/p]$ for the formula obtained by substituting term p for free occurrences of variable x in ϕ. In the figures, p is a parameter for the created object, C the class of p and D an arbitrary class other than C.

$$
\begin{array}{lll}
f_{wp}(op, \phi_1 * \phi_2) & \stackrel{\text{def}}{=} & f_{wp}(op, \phi_1) * f_{wp}(op, \phi_2) \qquad (* \text{ a prop. connective}) \\
f_{wp}(op, \exists x \in C : \phi) & \stackrel{\text{def}}{=} & \exists x \in C : f_{wp}(op, \phi) \vee f_{wp}(op, \phi[x/p]) \\
f_{wp}(op, \exists x \in D : \phi) & \stackrel{\text{def}}{=} & \exists x \in D : f_{wp}(op, \phi[x/p]) \\
f_{wp}(op, \forall x \in C : \phi) & \stackrel{\text{def}}{=} & \forall x \in C : f_{wp}(op, \phi) \wedge f_{wp}(op, \phi[x/p]) \\
f_{wp}(op, \forall x \in D : \phi) & \stackrel{\text{def}}{=} & \forall x \in D : f_{wp}(op, \phi[x/p]) \\
f_{wp}(op, p \downarrow t) & \stackrel{\text{def}}{=} & t = n \\
f_{wp}(op, t \downarrow p) & \stackrel{\text{def}}{=} & false \\
f_{wp}(op, p \downarrow^* t) & \stackrel{\text{def}}{=} & n \downarrow^* t \\
f_{wp}(op, t \downarrow^* p) & \stackrel{\text{def}}{=} & p = t \\
f_{wp}(op, t \rightarrow p) & \stackrel{\text{def}}{=} & t \downarrow n \wedge \bigwedge_d \forall x \in D : (x \neq t \wedge x \downarrow n) \Rightarrow x \rightarrow t \\
f_{wp}(op, p \rightarrow t) & \stackrel{\text{def}}{=} & false \\
f_{wp}(op, t \rightarrow^* p) & \stackrel{\text{def}}{=} & t \downarrow n \\
f_{wp}(op, p \rightarrow^* t) & \stackrel{\text{def}}{=} & p = t \\
f_{wp}(op, axis(t_1, t_2)) & \stackrel{\text{def}}{=} & axis(t_1, t_2) \qquad (t_i \neq p) \\
f_{wp}(op, t_1 \; op \; t_2) & \stackrel{\text{def}}{=} & t_1 \; op \; t_2 \\
f_{wp}(op, t) & \stackrel{\text{def}}{=} & t \qquad (t \text{ a term or boolean constant})
\end{array}
$$

Fig. 3. Weakest Precondition Calculation for Operation $op = \mathit{Create}(n, C, U)$

$$
\begin{array}{lll}
f_{wp}(op, \exists x \in C : \phi) & \stackrel{\text{def}}{=} & \exists x \in C : x \neq n \wedge f_{wp}(op, \phi) \\
f_{wp}(op, \exists x \in D : \phi) & \stackrel{\text{def}}{=} & \exists x \in D : f_{wp}(op, \phi) \\
f_{wp}(op, \forall x \in C : \phi) & \stackrel{\text{def}}{=} & \forall x \in C : x \neq n \Rightarrow f_{wp}(op, \phi) \\
f_{wp}(op, \forall x \in D : \phi) & \stackrel{\text{def}}{=} & \forall x \in D : f_{wp}(op, \phi)
\end{array}
$$

Fig. 4. Weakest Precondition Calculation for Operation $op = \mathit{Delete}(n, C)$

Figures 5 and 6 show the incremental precondition algorithms for Create and Delete operations. Again, the Delete case includes only rules differing from the

Create case. These rules are valid for the limited-alternation fragment of FOT only. For example, in both the Create and Delete case we use the fact that since limited-alternation formulas are Π_2, there will be no universal quantifiers nested inside existential quantifiers. In the case for implication in Figure 5, we use that if $\phi_1 \Rightarrow \phi_2$, then ϕ_1 must contain only universal quantifiers.

$$
\begin{array}{lll}
\Delta(op, \phi) & \stackrel{\text{def}}{=} & true \quad (\phi \text{ does not contain } c) \\
\Delta(op, \exists x \in D : \phi) & \stackrel{\text{def}}{=} & true \quad (d \text{ any class, including } c) \\
\Delta(op, \forall x \in C : \phi) & \stackrel{\text{def}}{=} & \forall x \in C : \Delta(op, \phi) \wedge f_{wp}(op, \phi[x/p]) \\
\Delta(op, \forall x \in D : \phi) & \stackrel{\text{def}}{=} & \forall x \in D : \Delta(op, \phi) \\
\Delta(op, \phi_1 \wedge \phi_2) & \stackrel{\text{def}}{=} & \Delta(op, \phi_1) \wedge \Delta(op, \phi_2) \\
\Delta(op, \phi_1 \vee \phi_2) & \stackrel{\text{def}}{=} & (\Delta(op, \phi_1) \wedge \Delta(op, \phi_2)) \vee f_{wp}(op, \phi_1 \vee \phi_2) \\
\Delta(op, \phi_1 \Rightarrow \phi_2) & \stackrel{\text{def}}{=} & f_{wp}(op, \phi_1) \Rightarrow \Delta(op, \phi_2) \\
\Delta(op, \phi) & \stackrel{\text{def}}{=} & \phi \quad (\phi \text{ atomic})
\end{array}
$$

Fig. 5. Incremental Precondition Calculation for Operation $op = Create(n, C, U)$

$$
\begin{array}{lll}
\Delta(op, \exists x \in C : \phi) & \stackrel{\text{def}}{=} & CBF(op, \exists x \in C : \phi) \Rightarrow f_{wp}(op, \phi) \\
\Delta(op, \exists x \in D : \phi) & \stackrel{\text{def}}{=} & true \quad (d \neq c) \\
\Delta(op, \forall x \in C : \phi) & \stackrel{\text{def}}{=} & \forall x \in C : x \neq n \Rightarrow \Delta(op, \phi) \\
\Delta(op, \forall x \in D : \phi) & \stackrel{\text{def}}{=} & \forall x \in D : \Delta(op, \phi) \\
\Delta(op, \phi_1 \Rightarrow \phi_2) & \stackrel{\text{def}}{=} & f_{wp}(op, \phi_1) \Rightarrow \Delta(op, \phi_2) \\
CBF(op, \exists x \in C : \phi) & \stackrel{\text{def}}{=} & \phi[x/n] \quad (c \text{ does not appear free in } \phi) \\
CBF(op, \exists x \in C : \phi) & \stackrel{\text{def}}{=} & (\exists x \in C : x \neq n \wedge CBF(op, \phi)) \vee CBF(op, \phi[x/n]) \\
 & & \quad (c \text{ appears free in } \phi) \\
CBF(op, \exists x \in D : \phi) & \stackrel{\text{def}}{=} & \exists x \in D : CBF(op, \phi) \\
CBF(op, \phi_1 \vee \phi_2) & \stackrel{\text{def}}{=} & CBF(op, \phi_1) \vee CBF(op, \phi_2) \\
CBF(op, \phi_1 \wedge \phi_2) & \stackrel{\text{def}}{=} & CBF(op, \phi_1) \vee CBF(op, \phi_2) \\
CBF(op, \phi) & \stackrel{\text{def}}{=} & \phi \quad (\phi \text{ atomic})
\end{array}
$$

Fig. 6. Incremental Precondition Calculation for Operation $op = Delete(n, C)$

The algorithm for deletion uses an auxiliary function $CBF(op, \phi)$ ("can become false"), defined for every formula without universal quantifiers, which returns true whenever operation op can cause ϕ to change from true to false. We present a basic algorithm for computing *CanBecomeFalse*, but algorithms based on more precise static analyses are possible. Indeed, much more precise, albeit ad-hoc, algorithms are used in our implementation.

Example. In our example constraint, for the operation *op* of creating a new AdjRegion, the algorithm will produce the incremental precondition:

$$\forall x \in Region.\ p \downarrow x \Rightarrow$$
$$\forall z \in Region.\ p.regionId = z.nodeId \Rightarrow z.maxCpty < x.minCpty$$

Theorem 4. *Algorithm $f_{wp}(op, \phi)$ computes the weakest precondition of ϕ and op, while algorithm $\Delta(op, \phi)$ computes an incremental precondition of ϕ and op.*

One can also verify that for local constraints the output of these algorithms meets the complexity bound of Theorem 2. For general constraints the claims we can make are weaker. First of all, there are cases where the logical complexity of an incremental precondition is higher than that of the original constraint. For example, in a constraint of the form $\exists x \in C : \phi$, a delete operation $Delete(n, C)$ may yield the precondition $\phi(n) \Rightarrow \exists x \in C : x \neq n \wedge \phi(x)$, which is logically more complex but should yield better performance in the "average case". However, one can show that the worst-case running time for evaluation of the precondition can never be more than a constant factor above that for evaluation of the original constraint. This follows because f_{wp} preserves the structure of formulas, hence preserves the running-time bounds, while the running time of Δ is at worst linear in that of f_{wp}.

4.3 Logical Simplification

The Delta-X simplifier is a rewrite system that takes a formula of *FOT* as input and produces a logically equivalent formula of *FOT* as output. The quantifier depth and maximum number of free variables of subformulas does not increase through simplification. Indeed, because maximum quantifier depth is a factor that relates strongly to performance of the generated code, a main goal of simplification is to reduce quantifier depth.

The rewrite rules of the simplifier are based on laws of first-order logic and the domain of trees. The following are a few sample rules:

$$\exists x \in A : \phi_1 \vee \phi_2 \rightsquigarrow \phi_1 \vee \exists x \in A : \phi_2 \ (x \text{ not free in } \phi_1) \tag{1}$$
$$x \neq t \vee \phi \rightsquigarrow x \neq t \vee \phi[x/t] \ (x \text{ not free in } t) \tag{2}$$
$$t \downarrow x \rightsquigarrow x = parent(t) \tag{3}$$

Rule 1 captures a validity of first-order logic. Rule 2 is a demodulation rule, allowing one term to be substituted for an equal term. Rules 3 is domain-specific; it says that t is a child of x just when x is the parent of t. Additional rules would be present if schema information were available. For example, rules would capture the relationships given by a class hierarchy.

Example. Figure 7 shows how our system simplifies the example precondition of Section 4.2. The subformula $class(parent(p)) \neq Region$ could be simplified to *false* if schema information showed that the parent of p must be of class Region.

$$
\begin{array}{l}
\quad \forall x \in Region: \; p \downarrow x \Rightarrow \\
\qquad \forall z \in Region: \; p.regionId = z.nodeId \Rightarrow z.maxCpty < x.minCpty \\
\rightsquigarrow \forall x \in Region: \; \neg p \downarrow x \vee \\
\qquad \forall z \in Region: \; p.regionId \neq z.nodeId \vee z.maxCpty < x.minCpty \\
\rightsquigarrow \forall x \in Region: \; parent(p) \neq x \vee \\
\qquad \forall z \in Region: \; p.regionId \neq z.nodeId \vee z.maxCpty < x.minCpty \\
\rightsquigarrow \forall x \in Region: \; parent(p) \neq x \vee \\
\qquad \forall z \in Region: \; p.regionId \neq z.nodeId \vee z.maxCpty < parent(p).minCpty \\
\rightsquigarrow \forall z \in Region: \; p.regionId \neq z.nodeId \vee z.maxCpty < parent(p).minCpty \vee \\
\qquad \forall x \in Region: \; parent(p) \neq x \\
\rightsquigarrow \forall z \in Region: \; p.regionId \neq z.nodeId \vee z.maxCpty < parent(p).minCpty \vee \\
\qquad class(parent(p)) \neq Region
\end{array}
$$

Fig. 7. Simplification of the Example Precondition

To get a feeling for the cost savings achieved by the simplifier, we generated preconditions from the 83 constraints in the Delta-X regression test suite, which are modelled after constraints in Delta-X applications. The cost of a precondition was computed as n^d, where n is the number of nodes per class, and d is the maximum loop nesting depth in the code generated for the precondition. Using this approach, the cost before simplification was 9.6×10^6, and the cost afterward was 3.5×10^6 – a savings of about 63%. Although the cost savings are good for this constraint set, the simplifier lacks robustness. We have had to make extensions to the rule set as new constraint examples appear.

Our rewriting system currently uses a bottom-up rewriting strategy. To increase the power of our system, we are experimenting with alternative rewriting strategies. We are also considering the use of third-party systems (e.g., Simplify [14], PVS [16]), but have yet to find a system that meets our needs. Most systems are geared towards theorem proving rather than simplification (e.g., they use skolemization, which does not preserve logical equivalence). There are licensing issues with most academic systems, and many commercial systems are targeted to special domains, such as hardware design. Finally, we require that simplification be directed by our cost function.

4.4 Translating Logic to Code

We translate formulas of our logic into code via the Delta-X Imperative Language (DIL). The translation of DIL to popular imperative languages is straightforward, so here we describe only the translation from logic to DIL.

DIL looks roughly like C++ and Java. It has classes, methods, statements, and expressions. The types, expressions, and basic statements of the language relate directly to the data model we use. For example, there are expressions to get the attribute of a node and to get the parent of a node. Sequencing, looping, and conditional statements are provided as control structures. However, only special-purpose looping statements are provided: for iterating over all objects of a class, or over all children of a node.

The translation to DIL takes a formula of our logic, plus a boolean variable, and produces a statement such that the value assigned by the statement to the variable is exactly the value our semantics dictates for the formula. Terms are translated similarly, except that the computed value need not be boolean.

Example. The following shows the DIL code produced from a formula similar to the simplified precondition of Figure 7.

```
// forall z in Region:
//    p.region <> z.nodeId or z.maxCpty < parent(p).minCpty
okay := true
for all z in Region while okay {
   // p.region <> z.nodeId or a.maxCpty < parent(p).minCpty
   okay := p.region <> z.nodeId
   if (!(okay)) {
      okay := a.maxCpty < parent(p).minCpty
   }
}
```

The translation of logic to DIL works in a bottom-up fashion on the structure of the formula. The details are mostly straightforward, because most terms of the logic correspond closely to expressions of DIL. However, in the translation of quantifiers one can tradeoff space for time, so we define two translation methods. In the *iterative method*, a quantifier is translated directly to a loop construct of DIL, as in the example above. In the *tabular method*, a quantifier is translated to a mapping represented as a table. The mapping obtained from the universally quantified formula in the example above takes as input a value for free variable p, and produces a boolean value as output.

The iterative translation method produces code that in the worst case requires dk space and n^k time, where d is the maximum size of data values and $O(n^k)$ time, n is the number of nodes in the tree, and k is the maximum quantifier depth of the formula. The tabular method produces code with worst case time and space $O(n^v)$, where v is the maximum number of free variables found in any subformula of the formula.

5 Related Work and Discussion

The relational data model is the basis of much work on integrity constraint maintenance. [15, 3] deal with a rich class of constraints on relational databases, with an emphasis on static verification of transformational programs. Runtime approaches using relational calculus (equivalent in expressiveness to first-order logic) as the specification language include [10, 18, 9]. [12] surveys the problem, dealing with questions such as which additional data structures are to be maintained. [10] gives algorithms that can be used to provide a solution to the integrity constraint maintenance problem for relational calculus. The maintenance of arbitrary first-order constraints is problematic because the evaluation of first-order formulas has PSPACE-complete combined complexity. For this reason the incremental evaluation of such powerful languages has not become a standard part of commercial database management systems.

Richer than the relational model is the object data model, in which program data structures can be captured in a natural way. [13] studies constraint maintenance within an object-oriented database, again using a language that can capture all first-order constraints. Due to the richness of the language [13] cannot provide efficient exact guards, and hence looks for ways to provide weaker guarantees.

There is much recent work on the hierarchical XML data model, which lies in expressiveness between the relational and object models. XML constraint languages have evolved from DTDs, which express purely structural properties, to XML Schema [19], which extends DTDs with further structural features as well as the key and foreign key constraints of [4]. Yet more expressive languages [6] include both structural and data-oriented features. [17] presents integrity constraint algorithms for a subset of XML Schema dealing only with the tree structure, not with data values. To our knowledge no work on incremental constraint checking for more expressive XML constraint languages exists.

The specification language of the Lucent version of Delta-X differs in several ways from the one presented here. The most significant difference is that node expressions allow just the child axis, and that only key constraints must be of the form $Key(\downarrow^* /C)\ldots$ for C a classname. On the other hand, Delta-X allows a generalization of *Never* constraints that restricts the cardinality of a node expression — *Never* asserts that this cardinality is 0; the general version allows any integer upper or lower bound. The production version can make use of schemas specifying for each class C the possible child classes. The Delta-X simplifier allows these additional constraints to be exploited in guard generation – the schema is read in along with the constraints, and a set of simplification rules are dynamically generated. The schema for these applications requires the class containment hierarchy to be acyclic, and hence implies a fixed bound on the depth of the object hierarchy. With this restriction the use of the descendant relation in constraints becomes unnecessary. The absence of sibling axes is due to the fact that in these data-oriented applications the sibling ordering is arbitrary. In addition to *Create* and *Delete*, Delta-X supports a *Modify* update operation on trees, which allows the modification of selected attribute values.

Acknowledgements

Lucent engineers Robin Kuss, Amy Ng, and Arun Sankisa were involved in the inception of the Delta-X project and influenced many aspects of Delta-X, especially the design of the constraint language. Lucent engineers Julie Gibson and James Stuhlmacher contributed to the design of the constraint language and the DIL language. We thank Kedar Namjoshi and Nils Klarlund for helpful comments and discussion.

References

1. Marcelo Arenas, Leopoldo Bertossi, and Jan Chomicki. Consistent query answers in inconsistent databases. In *PODS*, pages 68–79, 1999.
2. Michael Benedikt, Timothy Griffin, and Leonid Libkin. Verifiable properties of database transactions. *Infomation and Computation*, 147:57–88, 1998.
3. Véronique Benzaken and Xavier Schaefer. Static integrity constraint management in object-oriented database programming languages via predicate transformers. In *ECOOP '97*, 1997.
4. Peter Buneman, Susan Davidson, Wenfei Fan, Carmem Hara, and WangChiew Tan. Keys for XML. In *WWW 10*, 2001.
5. James Clark and Steve DeRose. *XML Path Language (XPath)*. W3C Recommendation, November 1999. `http://www.w3.org/TR/xpath`.
6. Alin Deutsch and Val Tannen. Containment for classes of XPath expressions under integrity constraints. In *KRDB*, 2001.
7. E.W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976.
8. Görg Gottlob, Chistroph Koch, and Reinhard Pichler. Efficient algorithms for processing XPath queries. In *VLDB*, 2002.
9. Tim Griffin and Howard Trickey. Integrity maintenance in a telecommunications switch. *IEEE Data Engineering Bulletin, Special Issue on Database Constraint Management*, 1994.
10. Timothy Griffin, Leonid Libkin, and Howard Trickey. An improved algorithm for the incremental recomputation of active relational expressions. *TKDE*, 9:508–511, 1997.
11. Martin Grohe. Finite variable logics in descriptive complexity theory. *Bulletin of Symbolic Logic*, 4, 1998.
12. Ashish Gupta and Inderpal Mumick. *Materialized Views: Techniques, Implementations, and Applications*. MIT Press, 1999.
13. H.V. Jagadish and Xiaolei Qian. Integrity maintenance in an object-oriented database. In *VLDB*, 1992.
14. K. Rustan M. Leino and Greg Nelson. An extended static checker for Modula-3. In *Compiler Construction: 7th International Conference*, 1998.
15. William McCune and Lawrence Henschen. Maintaining state constraints in relational databases: A proof theoretic basis. *Journal of the ACM*, 36(1):46–68, 1989.
16. S. Owre, S. Rajan, J.M. Rushby, N. Shankar, and M.K. Srivas. PVS: Combining specification, proof checking, and model checking. In *CAV*, pages 411–414, 1996.
17. Yannis Papakonstantinou and Victor Vianu. Incremental validation of XML documents. In *ICDT*, 2003.
18. Xiaolei Qian. An effective method for integrity constraint simplification. In *ICDE*, pages 338–345, 1988.
19. Henry S. Thompson, David Beech, Murray Maloney, and Noah Mendelsohn. *XML Schema Part 1: Structures*. W3C Working Draft, April 2000. `http://www.w3.org/TR/xmlschema-1/`.
20. Philip Wadler. Two Semantics for XPath. Technical report, Computing Sciences Research Center, Bell Labs, Lucent Technologies, 2000.