Maintaining XML Data Integrity in Programs
An Abstract Datatype Approach[*]

Patrick Michel and Arnd Poetzsch-Heffter

University of Kaiserslautern, Germany
{p_michel, poetzsch}@cs.uni-kl.de

Abstract. In service-oriented loosely coupled distributed information systems, the format and semantics of the exchanged data become more and more important. We envisage that there will be an increasing number of general and domain-specific XML-based data formats for service-oriented computing. A typical example is a tax declaration form. If the schemas defining the formats specify structural and additional integrity constraints, we speak of constrained XML.
The paper describes a technique for an integration of constrained XML data into programming, which is able to handle integrity constraints. Our technique allows to automatically check the correctness of programs manipulating constrained XML data. In our approach, constrained XML data is treated like an abstract data type with an interface of schema-specific procedures. Programs use these procedures to manipulate the XML data. The preconditions of the procedures guarantee that procedures maintain the constraints. Our approach allows to automatically generate the preconditions and to simplify them to a minimal form. The technique is based on a path representation and logical embedding of XML data. The weakest precondition generation is implemented and exploits an SMT-solver for simplification.

1 Introduction

In service-oriented loosely coupled distributed information systems, the format and semantics of the exchanged data become more and more important. Having standardized data formats will simplify communication between different applications as well as among enterprises and clients. We envisage that there will be an increasing number of general and domain-specific complex data formats for service-oriented computing. Typical examples are formats for tax declaration or for negotiating insurance contracts. As XML has become a de facto standard to represent structured data, we assume that the formats are defined by XML schema languages.

We present a technique to improve programming support for constrained XML data where *constrained* means that the XML data satisfies a structural schema

[*] In "SOFSEM 2010: Theory and Practice of Computer Science" (LNCS 5901).
The original publication is available at www.springerlink.com:
http://dx.doi.org/10.1007/978-3-642-11266-9_50.

and *additional integrity constraints*. Our technique allows to automatically check the correctness of programs manipulating constrained XML data. In particular, we guarantee that the XML data resulting from the manipulation satisfies the constraints. This is a first for integrity constraints, which are far more complex than structural constraints.

In our approach, constrained XML data is treated like an abstract data type with an interface of schema-specific procedures. These *interface procedures* can be used in a host programming language to manipulate the XML data. They hide the special aspects of structured XML data handling behind a host language API. The implementation of the interface procedures is kept separate from the host language and is tailored to the specific aspects of XML data handling. Domain experts develop the interface procedures together with the schema.

Implementing such an approach splits into a number of key challenges, which have to be tackled together. Both the schema language and the language for interface procedures have to be accessible by domain experts and feature explicit data specific primitives. To verify the correctness of interface procedures, their semantics have to be formally defined and they have to relate to the semantics of schemata. In realistic scenarios, however, correctness proofs cannot be created by domain experts, as this would require them to have extensive knowledge in formal methods. So to offer comprehensive support for the definition and *correct* usage of XML data with integrity constraints, it is necessary to maintain schemata as invariants automatically.

The contributions of this paper are the following:

– An abstract datatype approach for the integration of XML data with integrity constraints, which automatically guarantees that updates are indeed valid.
– A logical framework based on a path representation of XML data, which allows us to use the necessary automated analysis techniques.
– A pattern-based schema language with embedded integrity constraints, whose semantics can be completely expressed within the logical framework.
– An update language used to formulate the interface procedures of the datatypes, whose semantics can also be expressed within the framework.
– A direct application of the automated techniques first described in [7], which allow us to generate minimal preconditions that guarantee that the integrity constraints defined by the schema are maintained by procedures.

Related work. Support for constrained XML data has always been weak in programming languages. Low level APIs like DOM [10] and SAX [9] are cumbersome to use and offer no support to check or maintain schema constraints. Data binding approaches like JAXB [8] and language extensions like XJ [5] have eased the problem at least for basic structural constraints, yet they are unable to cope with integrity constraints. New languages like XDuce [6] have been developed to offer comprehensive support for sophisticated structural type systems. Such languages, however, do not integrate well into the existing languages and tools and suffer from being too specific. In the case of XDuce, the language excels at handling severe structural changes and transformations, which is not needed when working with a fixed schema. It also does not cover integrity constraints at all.

Gardner, et al. applied context logic [1] to an XML subset and a corresponding fragment of the DOM API [4]. Using their weakest precondition generation, they are able to analyze straight-line Java code. As an example they show how to prove structural schema invariants. Schema constraints are directly expressed in context logic, which makes them hard to read for common programmers. They do not define a higher level schema language and do not show how integrity constraints could be supported. Furthermore, there is currently no technique available to simplify preconditions. Their approach relies on manual proofs, rather than automated methods, which renders the technique inaccessible to domain experts.

In [7] we have presented an assertion language together with a technique to automatically generate preconditions for a core update language. These preconditions guarantee that integrity constraints are maintained. As the constraints can be complex, we developed a technique that allows to simplify the preconditions to a minimal form. In particular, we do not want to check the full set of integrity constraints of the manipulated XML data at every call site. Many constraints are unaffected by an update and remain true, so they need not be checked over and over. Other affected constraints can be checked by a much smaller incremental check. By using this technique, we are able to present minimal preconditions to the programmer in a readable form, so he can prevent individual constraints from failing and knows how to react to the failure of others.

Overview. Section 2 describes our approach in more detail and gives an illustrating example. Section 3 introduces the formalization on which the rest of the paper is based. Section 4 and 5 define the schema and the update language, respectively. Section 6 concludes the paper.

2 XML Data as Abstract Datatype

We propose to view XML data as an abstract datatype and interface it with a target language by a set of *interface procedures*. To perform more complex tasks, programs can be written that use the interface procedures as primitive operations. In this way, the intricacies of the constraint handling are hidden from the programmer. The interface procedures are developed together with the schema by domain experts in a light-weight XML manipulation language which incorporates concepts for constrained data. We explain the manipulation language that we developed with a tiny usage scenario: Clients exchanging packets. Each packet has a packet header represented as constrained data. Here is the schema that a domain expert might develop for packet headers:

```
packetheader {
  capacity { INT [ sum(//kind/count) ≤ . ] } &
  kind ∗ {
    count { INT [ . > 0 ] }
} }
```

A header consists of an integer expressing the `capacity` of the packet and a set of `kind` elements. A `kind` element records for each kind of item in the packet the count of items of that kind. Implicitly, the schema defines for each element an *identifier attribute*. If elements can occur with an arbitrary multiplicity, like the `kind` elements, all their identifiers have to be pairwise distinct. This implicit *uniqueness* constraint is enforced to make sure that elements are always structurally distinguishable. In addition, the above schema defines two integrity constraints: A `count` has always to be positive and the `capacity` of the packet has to be larger than the sum of the `counts`. These constraints are expressed by the *embedded* context rules enclosed in brackets. The dot refers to the element to which the context rule is attached.

For the manipulation of packet headers, the domain expert writes interface procedures. The following interface procedure adds `amount` items of kind `k` to an implicit packet header argument. If the amount is negative or zero, procedure `add` fails. Otherwise it creates a new element for items of kind `k` if necessary and increases the count:

```
add(ident k, int amount) {
  assume amount > 0;
  if not //kind[k] then
    new //kind[k];
    new //kind[k]/count;
    //kind[k]/count := 0
  fi
  //kind[k]/count := //kind[k]/count + amount
}
```

We are able to automatically generate weakest preconditions for such procedures and translate them to languages like Java. The example procedure `add` would result in a member method of a type `Packetheader` with the following signature:

```
// Precondition:
//   amount > 0                                ( AssumptionException )
//   sum (//kind/count) + amount <= //capacity  ( CapacityException )
Packetheader add(Ident k, Integer amount) { ... }
```

In Java programs, the interface procedures are used to manipulate data of the schema type and realize more complex tasks. The `Ident` type is provided by the generated API and can be assumed to subsume at least all `Strings`. The exceptions raised by failing preconditions are generated as unchecked exceptions, as programmers should have the liberty to ignore them in contexts where they are sure they cannot arise. A Java method for packing a list of items and sending them to other clients could look like this:

```
void pack(List<Ident> items) {
  Packetheader cur = new Packetheader(42); // 42 is packet capacity
  for(Ident item : items) {
    try { cur.add(item, 1); }
    catch(CapacityException e) {
      sendPacket(cur);
      cur = new Packetheader(42).add(item, 1);
    }
  }
  sendPacket(cur);
}
```

The method creates new packets and fills them up with items. The programmer has to make sure that whenever he calls the `add` method, he can either guarantee the precondition or handle its failure. As the `amount` is set to the constant 1, the only thing that can possibly go wrong is that the packet is already full. In this case, the add method will abort *without changing anything* in the packet. The `pack` method then sends the full packet and creates a new one. Adding the item to the new packet will now never result in an exception, so the call is safe.

3 Paths and Documents

As usual in the domain of XML, we will refer to constrained XML data as *documents*. The basis of our approach is a path representation and logical embedding of such documents. Paths are an intuitive concept known to domain experts. At the same time, however, paths are the key to using automated methods to ensure schema correctness of manipulating procedures. We use paths to identify and give an identity to elements in a document. Documents can then easily be defined as sets of paths with attached values.

`Example path: /packetheader/kind[k]/count`

We present an enhanced version of the original formalization described in [7], which is in particular prepared to handle arbitrary values. The weakest precondition generation and simplification techniques described there easily extend to these changes.

Our formalization supports identifiers, strings, integers and the special complex value clx, which marks elements without value and inner elements of a document. There can be arbitrary many constants c and variables v of each value type and it is possible to cast values to the base types.

$$\begin{aligned}
\text{values } V &::= I \mid S \mid Z \mid \text{clx} \mid D(P) \\
\text{identifier } I &::= c_I \mid v_I \mid \text{null} \mid \text{cast}_I(V) \\
\text{strings } S &::= c_S \mid v_S \mid \text{cast}_S(V) \\
\text{integer } Z &::= 0 \mid 1 \mid v_Z \mid Z + Z \mid Z * Z \mid -Z \mid \text{cast}_Z(V) \\
&\quad \mid \text{sum}(V^*) \mid \text{count}(V^*) \mid \text{rcount}(V, V^*)
\end{aligned}$$

Identifiers I and strings S are words over the alphabets Σ_I and Σ_S and we assume those words can be distinguished. Z represents the integer numbers with common arithmetic connectives. The constant null refers to the empty identifier. $D(P)$ denotes reading a value from document D at path P.

Paths represent the different elements in a document. The root path is denoted by root. Longer paths can be constructed by extending another path with a label, to select an element with that name, and an identifier, which separates this element from others with that name. Labels L are non-empty words on the alphabet Σ_L, like `packetheader` or `capacity` in the example.

$$\begin{aligned} \text{labels } L &::= c_L \\ \text{paths } P &::= \text{root} \;\mid\; P/L[I] \\ \text{documents } D &::= \text{blank} \;\mid\; D[P \to V] \;\mid\; D[P \to] \;\mid\; \$ \end{aligned}$$

The example path from above has the following formal representation:

`Formal path:` $\text{root}/\texttt{packetheader}_L[\text{null}]/\texttt{kind}_L[\texttt{k}_I]/\texttt{count}_L[\text{null}]$

A document consequently is a finite map from paths to values, where the blank document represents an initial document, which only contains the root path mapping to clx. We support local modifications in the form of adding (or replacing) a single path-value-pair and removing all pairs for a path and all its extensions. The dollar symbol is the only variable we use for documents. For example, the initial document with count of kind k set to zero is expressed with the following expression, in slightly abbreviated path syntax:

`Example document:` $\$\big[/\texttt{packetheader}/\texttt{kind}[\texttt{k}]/\texttt{count} \to 0\big]$

The sorts V^* and P^* represent value and path multisets and offer the usual singleton and union operations. We allow variables v_m on value multisets, have a constant named 'all', representing complete I, and can convert path multisets to value multisets by reading from the document using the syntax $D(P^*)$. Path multisets can also be created by extending all paths in another multiset with a label and an identifier from a value multiset, using the same syntax $./.[.]$ as for single paths.

$$\begin{aligned} \text{value multisets } V^* &::= \{V\} \;\mid\; V^* \cup V^* \;\mid\; D(P^*) \;\mid\; \text{all} \;\mid\; v_m \\ \text{path multisets } P^* &::= \{P\} \;\mid\; P^* \cup P^* \;\mid\; P^*/L[V^*] \end{aligned}$$

To sum up all the count elements in the example, we first create a path multiset representing all these elements and then read their values from the document. Within our schema, the expression sum(//kind/count) expands to:

`Example:` $\text{sum}\big(\$\big(\{\text{root}\}/\texttt{packetheader}_L[\{\text{null}\}]/\texttt{kind}_L[\text{all}]/\texttt{count}_L[\{\text{null}\}]\big)\big)$

4 Schema Language

In this section, we show by example how a schema language accessible by domain experts can look like. Any paradigm and feature, which can be translated into propositions on the terms of Section 3, can be supported.

We choose to combine the simplicity of a pattern-based approach in the spirit of abbreviated Relax NG [2, 3] with a rule-based approach. The patterns define the structure of documents, while the rules define arbitrary integrity constraints. By embedding paths directly into the logic, integrity constraints can be understood and written by domain experts and read by programmers.

4.1 Syntax

We support a usual base of patterns, namely the empty pattern, groups, choice, element definitions and repetition of elements. The content of elements can be complex, a type or an enumeration (cf. pattern example in Section 2).

$$\begin{aligned} \text{pattern } Q &::= \epsilon \mid Q\&Q \mid Q|Q \mid L\{C\} \mid L*\{C\} \\ \text{content } C &::= Q \mid T \mid E \\ \text{enums } E &::= V \mid E|E \end{aligned}$$

Integrity constraints are defined in a rule-based approach, as propositions on paths and values. We support the normal boolean connectives, equality, integer comparisons and top level quantification over identifier variables. To formulate structural constraints and guards, we can also express that a path is contained in a document and that a value has a specific type.

$$\begin{aligned} \text{formulas } G &::= \forall v_I.G \mid F \\ F &::= \text{false} \mid F \wedge F \mid F \vee F \mid \neg F \\ &\mid \alpha = \alpha \mid Z < Z \mid P \in D \\ \text{types } T &::= \mathit{INT} \mid \mathit{ID} \mid \mathit{STR} \mid \mathit{CLX} \mid \text{typeOf}(V) \end{aligned}$$

4.2 Semantics

Patterns The semantics of patterns are given in terms of the assertion language, i.e. patterns are completely translated into formulas. Each element definition in a pattern defines a set of paths using the same labels, but different identifiers. The existence of each of these paths is tied to that of other paths, like the parent path or non-optional children.

To formulate these dependencies, we define the *characterizing path* of an element definition. To compute the characterizing path of an element, the characterizing path of the next enclosing element definition is extended with the label of the element and the null identifier. For repeated elements, we choose a fresh variable as identifier instead, which is quantified in the context. If there is no next enclosing element definition, we extend the root path instead.

The characterizing paths for all element definitions of the example schema are the following:

`/packetheader`	`/packetheader/kind`$[x]$
`/packetheader/capacity`	`/packetheader/kind`$[x]$`/count`

Using these paths, the relations defined by patterns can be expressed as propositions. For each element, for example, a proposition is created, which ensures the existence of its parent:

$$\forall x.\ \texttt{/packetheader/kind}[x]\texttt{/count} \in \$ \rightarrow \texttt{/packetheader/kind}[x] \in \$$$

Groups of elements exist if both patterns exist. For patterns combined in a choice, at least one of them has to exist. Both the empty pattern and a repeated element always exist, as the latter can also be an empty sequence.

Elements containing non-complex content, i.e. a datatype or enumeration of values, are translated into similar propositions. The typeOf function is used to guarantee that an assigned value has the appropriate type. Enumerations translate into a disjunction of equalities testing the different values.

$$\texttt{Example:}\ \forall x.\ \texttt{/packetheader/kind}[x]\texttt{/count} \in \$ \rightarrow \\ \text{typeOf}\big(\$(\texttt{/packetheader/kind}[x]\texttt{/count})\big) = \mathit{INT}$$

Propositions The semantics of propositions are given in terms of the respective models of values, paths, multisets and documents, with respect to an interpretation of all variables, called *evaluation environment* E. We denote an evaluation in environment E with $[\![\cdot]\!]_E$. In order for the techniques of [7] to work and to facilitate static analysis, the semantics have to be carefully designed. Nevertheless, we are able to support the intuition behind the connectives.

Due to space limitations, we do not define them formally here and point out the important aspects instead. For simplicity of the logic, both the read function and all cast functions are defined as total, i.e. they return default values for invalid parameters. Section 4.4 explains how to deal with this behavior. The multiset operations sum and ./.[.], as well as .(.), are *selective* in the sense that they ignore values of the wrong type or paths which are not present in the document, respectively. This leads to much simpler specifications and allows to use infinite path multisets to create finite value multisets by reading from documents.

4.3 Embedding Rules

We now extend the classical pattern-based structural schema language with the possibility to embed rules. A *rule* is a single proposition attached to an element definition. This approach has the following benefits:

- By tying rules to a structural schema, the paths in rules can be checked for validity, resulting in the detection of more errors at compile time.
- Paths in rules can use the usual convenience axes, like *parent* or *descendants*.
- The context of a rule is implicitly given by the location of the rule in the pattern, allowing to use much shorter relative paths starting with the dot.

Rules can be embedded in any element definition, regardless if it is repeated or not, by enclosing it in brackets. Defining a rule at the top level is also possible, which means it has the root path as context. To support more navigational axes and relative paths, we extend the sort P^* as follows and allow explicit conversions of singleton sets to paths using cast_P.

$$\begin{array}{rl} \text{paths } P & ::= \text{root} \ \mid \ P/L[I] \ \mid \ \text{cast}_P(P^*) \\ \text{path multisets } P^* & ::= \{P\} \ \mid \ P^* \cup P^* \ \mid \ P^*/L[V^*] \\ & \mid \ . \ \mid \ P^*/.. \ \mid \ P^*/.L \ \mid \ P^*//L[V^*] \end{array}$$

The dot refers to the characterizing path of the enclosing element definition. The double dot can be used to navigate to a direct parent, whereas the single dot followed by a label jumps to the next parent of that name. A double slash refers to all possible descendants matching the supplied label.

To be able to use the new axes to select meaningful multisets and even single paths, the resulting path multisets have to be filtered with the paths defined in the structural schema. The set of all these paths is derived analogously to characterizing paths. The only difference is that instead of inserting a variable as identifier for repeated element definitions, we add an infinite number of paths, consisting of one version for each identifier.

With this definition, the semantics of the new constructs can be defined to work within the limits of the schema. Again, the formal definition of the new connectives leaves the scope of the paper, yet it is easily possible to capture the intuition behind them. Note that the reverse axes parent and ancestor both need to filter out duplicates, as they could otherwise produce an infinite number of duplicates of paths.

In order to further improve readability and conciseness of schema specifications, a concrete syntax for embedded rules will offer the following features:

– *Implicit casts* both on values and paths. In almost every case, the needed sort of an expression can be inferred from the context and a cast introduced accordingly.
– *Implicit document* both for reading values and domain checks. Schema constraints always refer to the document \$, without any manipulations, which makes the \$ obsolete. We can then neglect the parenthesis used for reading and the $\in$ symbol, as the context makes clear if a value or proposition is needed.
– *Implicit all and null.* Leaving out the brackets in a path step has the implicit meaning of 'all' for path multisets and 'null' for single paths.
– *Implicit root.* All paths either start with root or {root}, so in concrete syntax absolute paths just start with a slash.

4.4 Specification Errors

Although all operators are defined as total, there are several situations which should be considered specification errors and can in fact support the development of schemata immensely. Resulting empty sets, for instance, should be considered specification errors, as they most likely do not reflect the intention of the programmer, but reveal an inconsistency with the schema or just a simple typo. In particular it is undesirable that one of the following ever happens:

– A $\text{cast}_{I/S/Z}$ fails and returns the default value of that sort instead.
– A cast_P fails and returns the root path instead.
– A non-existent path is read from the document, resulting in the value clx.

All of these can easily be prevented by construction or treated as error by modifying the specification. In embedded rules, for example, it makes sense to automatically guard each document access using a unique path with a domain check for this path. The implied semantics is that a rule need only hold, if the paths used actually exist in the document in question. Especially the specification of rules in repeated elements benefits from this implicit assumption.

In procedures, it may be desirable to assert that no cast or document access fails, by extending the precondition of statements containing such terms with a guard for each. Value casts can be guarded using typeOf, unique document access with $\in$. We have defined the structural schema language in such a way, that all typeOf comparisons can be statically decided. Unique path casts should be rejected if they are not statically decidable, which only happens in the context of choices or repeated elements and hints at bad programming practice.

Finally, it is possible to transform a formula such that failing casts and reads are treated as errors. This is done by case analysis and simply removing literals containing a failed cast or read access from their disjunction in conjunctive normal form. The underlying idea is that a literal containing a term raising an exception can no longer contribute to fulfilling a disjunction.

5 Procedures

We now define am exemplary core procedural update language, which can be used by domain experts to define the atomic manipulations necessary to use a schema. Analogous to the schema language defined in Section 4, any paradigm and feature can be supported, as long as the semantics of the language can be described in the logical framework of Section 3. In particular, all modifications have to translate to set $.[. \to .]$ or delete $.[. \to]$ operations.

The guiding principal for the core language is to provide the means to encapsulate alien aspects like paths from the host language and allow to build a toolset of atomic procedures to work with. At the same time, the language allows us to use automated methods to show the programmer the minimal conditions he has to ensure for an interface procedure to maintain the integrity constraints.

5.1 Syntax

Procedures have a name, a list of variables marked as parameters and a body containing a sequence of statements. A statement can create a new element, free an element or set the value of an existing one. It is possible to annotate assumptions and use conditionals for alternative control flows. Finally, variables for single values and value multisets can be declared and assigned.

$$\begin{array}{rcl}
\text{procedures } R & ::= & L\ (\ M\ )\ \{\ U\ \} \\
\text{parameters } M & ::= & \mathbf{int}\ v_Z \ \mid\ \mathbf{str}\ v_S \ \mid\ \mathbf{ident}\ v_I \ \mid\ M, M \\
\text{sequences } U & ::= & O \ \mid\ U; U \\
\text{statements } O & ::= & \mathbf{new}\ P \ \mid\ \mathbf{free}\ P \ \mid\ P := V \\
 & \mid & \mathbf{assume}\ G \ \mid\ \mathbf{if}\ F\ \mathbf{then}\ U\ \mathbf{else}\ U\ \mathbf{fi} \\
 & \mid & v_Z = Z \ \mid\ v_S = S \ \mid\ v_I = I \ \mid\ v_m = V^*
\end{array}$$

Note that formulas in conditionals have to be quantifier-free. We do not support loops, as these would prevent *automated* weakest precondition generation.

5.2 Semantics

The operational semantics of procedures define how sequences of statements modify evaluation environments: $U, E \rightsquigarrow E'$. As there is only one document variable, we use the abbreviated notation $E[\![D]\!]_E$ instead of $E\big[\$ \mapsto [\![D]\!]_E\big]$.

$$\frac{U_a, E \rightsquigarrow E' \qquad U_b, E' \rightsquigarrow E''}{U_a; U_b, E \rightsquigarrow E''} \qquad \frac{[\![P/L[I] \notin \$]\!]_E \qquad [\![P \in \$]\!]_E}{\textbf{new } P/L[I], E \rightsquigarrow E[\![\$[P/L[I] \to \text{clx}]]\!]_E}$$

$$\frac{[\![P \in \$]\!]_E \qquad P \neq \text{root}}{\textbf{free } P, E \rightsquigarrow E[\![\$[P \to]]\!]_E} \qquad \frac{[\![P \in \$]\!]_E \qquad P \neq \text{root}}{P := V, E \rightsquigarrow E[\![\$[P \to V]]\!]_E}$$

$$\frac{[\![F]\!]_E \qquad U_t, E \rightsquigarrow E'}{\textbf{if } F \textbf{ then } U_t \textbf{ else } U_e \textbf{ fi}, E \rightsquigarrow E'} \qquad \frac{[\![\neg F]\!]_E \qquad U_e, E \rightsquigarrow E'}{\textbf{if } F \textbf{ then } U_t \textbf{ else } U_e \textbf{ fi}, E \rightsquigarrow E'}$$

$$\frac{[\![G]\!]_E}{\textbf{assume } G, E \rightsquigarrow E} \qquad \frac{}{v_m = V^*, E \rightsquigarrow E\big[v_m \mapsto [\![V^*]\!]_E\big]}$$

$$\frac{}{v_{I/S/Z} = I/S/Z, E \rightsquigarrow E\big[v_{I/S/Z} \mapsto [\![I/S/Z]\!]_E\big]}$$

The initial environment is constructed by binding the input document to \$, the parameter values to the declared parameter variables and default values to all other variables used in statements.

Note that although the formalization of documents allows arbitrary finite maps from paths to values, it is in practice desirable to only deal with documents which represent a tree. A document is a tree, if for every path it contains, it also contains all prefixes of that path. The presented update language is designed in such a way, that this property is maintained by all statements.

Weakest precondition generation for sequences and statements is defined as follows. The weakest preconditions for sequences and conditionals are as usual. We use the syntax $G_{[b/a]}$ to express that a is substituted by b in the formula G.

$$\begin{aligned}
&\text{wp } (\textbf{new } P/L[I], && G) = G_{\left[\$[P/L[I]\to\text{clx}]/\$\right]} && \wedge P \in \$ \wedge P/L[I] \notin \$ \\
&\text{wp } (\textbf{free } P, && G) = G_{\left[\$[P\to]/\$\right]} && \wedge P \in \$ \wedge P \neq \text{root} \\
&\text{wp } (P := V, && G) = G_{\left[\$[P\to V]/\$\right]} && \wedge P \in \$ \wedge P \neq \text{root} \\
&\text{wp } (\textbf{assume } G_a, && G) = G \wedge G_a && \\
&\text{wp } (v_{I/S/Z} = I/S/Z, && G) = G_{\left[I/S/Z/v_{I/S/Z}\right]} && \\
&\text{wp } (v_m = V^*, && G) = G_{\left[V^*/v_m\right]} &&
\end{aligned}$$

The document manipulations of statements in procedures are directly expressed in the logical embedding. In [7] we have introduced the necessary rewriting rules to remove the two document manipulations from formulas. In most cases, these rules even reduce the size of formulas and allow subsequent simplification. In this way, the weakest preconditions can indeed be checked on the prestate of procedures.

6 Conclusion

In this paper we presented a first approach to deal with XML data with integrity constraints from within common programming languages. Such XML data is seen as abstract datatype, exposing an interface of basic procedures for manipulation. Domain experts write structural schemata in a normal pattern-based approach, but can also embed sophisticated integrity constraints using paths. Using these paths, they also define the basic atomic manipulations associated with the schema as procedures. We are able to automatically translate and analyze procedures, forming a comprehensive approach to correctness.

The weakest preconditions of procedures are derived automatically and can also automatically be reduced to a minimal form. These preconditions guarantee that a procedure maintains the integrity constraints. The programmer using procedures is able to read the preconditions, guarantee them on calls or react to their failure. Procedures interface with the host language by using primitive types, especially identifiers.

All this is made possible by the underlying path-based formalization first presented in [7]. It allows specifications both to be easily accessible by domain experts and programmers and to be processed by automated tools. We have implemented a prototype system integrating an SMT-solver for simplification. In all example specifications, the system was able to derive the minimal precondition automatically.

References

1. C. Calcagno, P. Gardner, and U. Zarfaty. Context logic and tree update. *SIGPLAN Not.*, 40(1):271–282, 2005.
2. J. Clark. RELAX NG compact syntax, Nov. 2002. http://www.oasis-open.org/committees/relax-ng/compact-20021121.html.
3. J. Clark and M. Makoto. RELAX NG specification, Dec. 2001. http://www.oasis-open.org/committees/relax-ng/spec-20011203.html.
4. P. A. Gardner, G. D. Smith, M. J. Wheelhouse, and U. D. Zarfaty. Local hoare reasoning about DOM. In *PODS '08: Proceedings of the twenty-seventh ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 261–270, New York, NY, USA, 2008. ACM.
5. M. Harren, M. Raghavachari, O. Shmueli, M. G. Burke, R. Bordawekar, I. Pechtchanski, and V. Sarkar. XJ: facilitating XML processing in Java. In *WWW '05: Proceedings of the 14th international conference on World Wide Web*, pages 278–287, New York, NY, USA, 2005. ACM.
6. H. Hosoya and B. C. Pierce. XDuce: A statically typed XML processing language. *ACM Trans. Internet Techn*, 3(2):117–148, 2003.
7. P. Michel and A. Poetzsch-Heffter. Assertion support for manipulating constrained data-centric XML. In *International Workshop on Programming Language Techniques for XML (PLAN-X 2009)*, January 2009.
8. S. Microsystems. Java architecture for XML binding (JAXB). http://java.sun.com/developer/technicalArticles/WebServices/jaxb//.
9. Sourceforge. Simple API for XML (SAX). http://www.saxproject.org/.
10. W3C. Document object model (DOM). http://www.w3.org/DOM/.