Specification and Algorithms of Constrained XML Documents

Patrick Michel

July 23, 2007

Author: Patrick Michel
p_michel@informatik.uni-kl.de
Supervisor: Prof. Dr. Arnd Poetzsch-Heffter
Software Technology Group
Organization: University of Kaiserslautern
Contents

1 Introduction ...................................................... 9

2 XML Data Model .................................................. 11

3 Language Definitions ........................................... 13
   3.1 Structure: Relax CX and RCX ............................... 15
         3.1.1 Lexical Structure and Grammar ............. 15
         3.1.2 Differences to Relax NG .................. 17
         3.1.3 Abstract Syntax ............................ 19
         3.1.4 Limitations in RCX ....................... 20
         3.1.5 Semantics ................................ 22
   3.2 Base Language: XPath CX .................................... 23
         3.2.1 Lexical Structure and Grammar ............. 23
         3.2.2 XPath 2.0 Abstract Syntax ............... 23
         3.2.3 Restrictions ................................ 26
         3.2.4 XPath CX Abstract Syntax ............... 28
   3.3 Context Rules: CRL ......................................... 30
         3.3.1 Lexical Structure and Grammar ............. 30
         3.3.2 Abstract Syntax ............................ 31
         3.3.3 Semantics and Restrictions ............... 31
         3.3.4 XPath CX Specific Restrictions .......... 33
   3.4 Manipulations: AL ........................................... 33
         3.4.1 Lexical Structure and Grammar ............. 34
         3.4.2 Abstract Syntax ............................ 35
         3.4.3 Semantics and Restrictions ............... 36
         3.4.4 XPath CX Specific Restrictions .......... 40

4 Static Analysis .................................................. 43
   4.1 Comments on Basic Techniques ............................ 43
   4.2 Relax CX Analysis .......................................... 44
         4.2.1 References and Cyclic Patterns ............. 44
         4.2.2 XML Node Classes and Pattern Positions .... 45
         4.2.3 Axis Steps on Node Classes and Set Evolution 51
         4.2.4 Document References ........................ 56
   4.3 XPath CX Analysis .......................................... 63
Contents

4.3.1 Supported Features and Restrictions ........................................ 65
4.3.2 Treats and Casts ................................................................. 65
4.3.3 Path Expressions ................................................................. 67
4.4 CRL Analysis ............................................................................. 69
4.4.1 Types of Rules ....................................................................... 71
4.5 AL Analysis ............................................................................... 73
4.6 Preconditions for Context Rules .................................................... 76
4.7 Remarks to Structural Correctness ............................................... 88
4.8 Example Analysis ...................................................................... 89

5 Variations and Future Work ............................................................. 95
5.1 Handling Restrictions ................................................................... 95
5.2 Approximations ........................................................................ 96
5.3 Dynamic Decision Trees ............................................................. 98
5.4 Document Interactions ............................................................... 98

6 Conclusion ................................................................................... 101
List of Figures

3.1 Dependencies and Versions of Languages . . . . . . . . . . . . . 14
3.2 Types of ambiguous nodes: a) by repetition, b) by recursion . . 20

4.1 Example of Inlining References in Cyclic Patterns . . . . . . . . 45
4.2 A Congruous Pool of Meta Node Classes . . . . . . . . . . . . . 48
4.3 Ancestor Node Classes of a Meta Node Class . . . . . . . . . . . 54
4.4 Ambiguous Node Classes and Degrees of Ambiguity . . . . . . . 57
4.5 XPath CX Supported Feature Table . . . . . . . . . . . . . . . . 66
4.6 Types of Context Rules and References . . . . . . . . . . . . . . 72
List of Figures
1 Introduction

Whenever XML is a first-class object in large systems, used by various clients, it has to maintain validity and meaningfulness. Systems not only exchange raw data, but already agreed on the use of the XML format for standardization. The class of all possible XML documents, however, is of nearly no interest to most application domains, so there has to be additional information, which is distributed together with the data.

This information constitutes an application specific language definition, which both guarantees the data to obey to various rules, but also demands from all clients to maintain them.

This work supplies the means necessary to give such language definitions in the form of structural as well as integrity and domain-specific constraints. It defines languages, which have to be understood by clients working with constrained XML data, such that they understand the restrictions imposed on the raw XML data.

Furthermore this work will define an algorithm definition language, which is used to define basic manipulation algorithms on constrained XML. Such algorithms are defined for specific sets of constrained XML documents and can be invoked on them to get another valid document, with the intended changes made.

To guarantee that algorithms maintain validity of documents regarding their language specification, analysis techniques and algorithms will be developed. Ideally, such techniques would be used by the client to statically verify that each algorithm preserves all constraints defined in the language definition of the application.

This is most often impossible to do, as both the state of the input document as well as the algorithm parameters can make the execution of the algorithm impossible. This work will nevertheless show how it is possible to statically generate a minimal number of preconditions, which are checked at runtime before the method is executed and which then guarantee that the execution of the method succeeds and leaves the document in a valid state.

Opposed to many other approaches, which use runtime checks in various stages of the method execution, an input document cannot be left in an invalid state and it will also never be necessary to rollback changes already done or even maintain copies of documents. The results of statically analyzing both language definitions and algorithm definitions will prevent extensive runtime checks altogether and reduce execution time to a minimum.
1 Introduction

These ambitious goals are opposed by dynamic and powerful standard XML technologies on top of which languages will be defined. Many features of these languages, which are convenient for the user and make out their success, are hard to analyze statically. Defining the right subsets of these languages is one of the major challenges of this work, as interesting features have to prevail, while complete static analysis should still be possible.

Altogether the work defines the necessary technologies to use constrained data in the form of XML as first-class objects in distributed systems, while minimizing dynamic efforts through sophisticated static analysis methods and a restrictive selection of supported features.

I will start by giving an XML data model in Section 2 which formulates the view on XML used by this work and which is mandatory for all subsequent sections.

Section 3 will then define the various languages necessary to give constrained XML language specifications and algorithm specifications, based on the well known language standards Relax NG [4] and XPath 2.0 [16]. The defined languages will either be subsets of these standards or be host languages for such subsets.

Using these languages, Section 4 describes techniques and algorithms to retrieve static information from specifications made with them and ultimately achieve the goal to generate all necessary preconditions for algorithms. The section concludes with a small example specification, showing the results of static analysis.

Finally, in Section 5 I will comment on the choices made and future options for research in this field. Section 6 concludes the work and summarizes how different goals were achieved.

The attached electronic medium contains electronic versions of this document, but also a prototype system which implements most of what is shown here. Section 4 gives some more comments on the content and on the selection of code listings used in this document.
2 XML Data Model

The XML format is usable in a variety of applications for very different reasons. Especially the degree of dynamic variability is an important characteristic of each application and which features of XML are exploited to transport information. The absence or presence of a structural schema definition, for example, is a first indication on how XML is used in a context.

The focus of this work is the representation, storage, exchange and manipulation of semi-structured data in the form of trees. In this sense XML is used as standard to represent such data and is not utilized in its entireness. The reader should more think of the data we want to represent as class instances in an object oriented setting than a marked up text like present in HTML.

The abstraction of XML I use is inspired by James Clark\footnote{5th paragraph of the “Foreword by James Clark” in [11].}:

I would argue that the right abstraction is a very simple one. The abstraction is a labelled tree of elements. Each element has an ordered list of children in which each child is a Unicode string or an element. An element is labelled with a two-part name consisting of a URI and local part. Each element also has an unordered collection of attributes in which each attribute has a two-part name, distinct from the name of the other attributes in the collection, and a value, which is a Unicode string. That is the complete abstraction. The core ideas of XML are this abstraction, the syntax of XML, and how the abstraction and syntax correspond. If you understand this, then you understand XML.

When comparing this abstraction of XML with the model given in the XML Information Set of the W3C [13], the important points to note are the following:

- The abstraction I will be using ignores processing instructions, unexpanded entity references, character information, comments, document type declarations, unparsed entities and notations. This also means that the remaining information items do not include any of these elements.

- Like defined in the XML information set standard, an XML document is required to be well-formed and has to satisfy the namespaces rules defined there, to be a valid member of the defined XML Data Model.
• With most members missing, the document information item does not
  contain any more useful information to be of any interest, so it is also
  neglected in the abstraction.

• A document therefore consists of elements and attributes only and is rep-
  resented by the single root element of the document.

For the remainder of this work, the formal definition of the used XML ab-
straction is the one given by James Clark in the context of a Relax NG validator
[3]. The XML data model is given in the form of datatype definitions in the
language Haskell [5]:

```haskell
-- URIs are just strings.
type Uri = String

-- Local names are just strings.
type LocalName = String

-- A namespace prefix is just a string.
type Prefix = String

-- A Context represents the context of an XML element.
-- It consists of a base URI and a mapping from prefixes
to namespace URIs.
type Context = (Uri, [(Prefix, Uri)])

-- A QName is a URI/local name pair.
data QName = QName Uri LocalName

-- An XML document is represented as a ChildNode.
-- There are two kinds of child nodes:
-- (1) a text node containing a string;
-- (2) an element node containing a name (of type QName),
-- a Context, a set of attributes (represented as a
-- list of attribute nodes, each of which will be an
-- AttributeNode), and a list of children
-- (represented as a list of ChildNodes).
data ChildNode =
  TextNode String
  | ElementNode QName Context [AttributeNode] [ChildNode]

-- An attribute node consists of a QName and a String.
data AttributeNode = AttributeNode QName String
```

Listing 2.1: XML Data Model in Haskell

Even though an element node contains a list of attribute nodes, no ordering
on attributes is implied. The ordering on the list of child nodes, however, is
significant in XML. The XML Data Model defined here will honor this ordering,
but I will not utilize it in this work.
3 Language Definitions

In this and the following sections I will define the necessary languages to give a Constrained XML Language Definition (CXLD). A Constrained XML Language Definition will consist of two major parts, which are:

- A structural definition, given as Relax CX or RCX’ specification.
- A definition of context conditions, given as CRL specification.

A well-formed, generic XML document\footnote{Section 2 defines the XML data model used in this work and it is of course also required that the document adheres to this model.} will be called \textbf{valid} with regard to a CXLD, if it both conforms to the structural constraints, as well as to the context conditions. The different subsections of this section will define the corresponding facets of conformity.

The ultimate goal is to ensure that certain XML documents are always valid with regard to a CXLD whenever they are stored or not actively processed. While documents are processed I will allow certain restricted forms of \textbf{inconsistency}, which will also be defined in the corresponding sections.

Starting from both Relax NG and XPath 2.0, I will define both subset languages and host languages to use for the different tasks needed. While it is desirable to use complete standardized languages whenever possible, or at least convenient subsets, it is often necessary to restrict languages severely.

Especially in the case of Relax CX, which is a simplified and slightly restricted subset of Relax NG, more restrictions are necessary to get first results from static analysis in this work. Nevertheless I will depict Relax CX as desired subset, which will be aimed for in future work.

Figure 3.1 shows all languages, which are either used or defined in this work, as well as their relations. Relax CX is a subset of Relax NG, with most modularity and convenience features removed, but mostly intact otherwise, from a structural point of view. Relax CX patterns can be defined and combined nearly as free as in the original language.

Relax CX’ (or RCX’) has some more restrictions in place, which are necessary for the results of this work, but might be discarded in future revisions. Section 3.1 defines these two languages.

XPath 2.0 is a completely different matter, however. The language is very complex and most features are connected to each other. Allowing and disallowing certain features and combinations can have a huge impact on the usability,
expressiveness and user convenience. Unfortunately these properties often go along with expressions becoming ambiguous and losing the ability to analyze them statically.

It will not be possible to support XPath 2.0 in completion and still be able to always guarantee certain useful static properties. So there is no desired subset of XPath 2.0 that can be outlined right away, but this language has to be found by enhancing a prototype core towards full XPath 2.0. The starting core is called XPath CX and will be defined in Section 3.2 together with a brief model and description of full XPath 2.0.

---

Figure 3.1: Dependencies and Versions of Languages

The Context Rule Language and the Algorithm Language are both based on XPath CX. Each specification of rules or algorithms is analyzed with regard to one associated RCX' schema, which allows to do static analysis of rules and algorithms using XPath expressions at all. Section 3.3 and Section 3.4 define these two languages.

Altogether, there are several implications from each of the four major languages defined in this work, which restrict the features of others. The decisions for or against certain features are guided by the general goal to be able to get static analysis results for every algorithm specification on any CXLD.
3.1 Structure: Relax CX and RCX'

As structural schema definition language I have chosen a subset of Relax NG. Relax NG is an Oasis standard from James Clark and Murata Makoto [4] and later became part 2 of the ISO/IEC standard on Document Schema Definition Languages [2]. The latest document of this standard is [6], while the DSDL project website still only links to the draft.

The differences between those standards are marginal and do not concern the level of detail which is necessary for this work. I will, however, refer to the more recent ISO/IEC standard as “Relax NG standard”.

I will refer to the subset which will be defined here as Relax CX or RCX, which should be read as “Relax for Constrained XML” and points to the relation to Relax NG. Relax CX is a strict subset of Relax NG. Relax CX’ or RCX’ has even more restrictions in place and is a strict subset of Relax CX. Both languages differ in restrictions on patterns only and share syntax, grammar and abstract syntax.

3.1.1 Lexical Structure and Grammar

Relax CX is a strict subset of Relax NG and utilizes a subset of the compact syntax for Relax NG. The ISO/IEC standard describes the XML syntax of Relax NG, while the compact syntax is given by a separate ISO/IEC document in the context of DSDL [2].

The lexical structure is identical to the one of the Relax NG compact syntax. Whitespace and comments can be added arbitrarily between tokens and are often necessary to separate them. Both do not alter the semantics of a specification, so the grammar of Relax CX is given in EBNF on the token stream.

Tokens are found via the usual longest match and belong to the following classes:

- **KEYWORD** All keywords defined for the Relax NG compact syntax, even those of not supported features in Relax CX.

- **IDENT** All words starting with a letter or underscore and containing letters, digits, the underscore, the dot or the hyphen. Keywords do not belong to this class, but it is possible to escape identifiers with a single backslash at the beginning, so it is possible to use keywords as identifiers too.

- **ANYIDENT** Same as IDENT, but also contains all keywords directly, without the need to escape.

Please note that Relax NG uses the complete W3C XML Namespace [15] class NCName for (ANY)IDENT, while Relax CX leaves out the CombiningChar and Extender character classes, for the sake of simplicity.
3 Language Definitions

- **LITERAL** A string enclosed in single or double quotes, with special escaping rules and the ability to specify a literal using several parts. Literals are used to specify both URIs and datatype values.

- **CNAME** A colonized name, which is two ANYIDENT joined with the colon. This class is best described as the PrefixedName subclass of the W3C XML Namespace QName class, where again Relax CX only accepts the most commonly used subclass of words.

- **NAME** This is ANYIDENT joined with CNAME, i.e. a QName in W3C XML Namespace.

- **OPERATOR** All sequences of one or two symbols, which appear in the Relax NG compact grammar, like :, | or &.

Based on this lexical structure, the grammar of Relax CX is given in EBNF. Elements of **KEYWORD** and **OPERATOR** are used directly as terminal symbols, as rules always expect specific members of these classes, while the other classes can be thought of as defined elsewhere in EBNF. Normal rule definitions are given as lowercase names.

```plaintext
relaxcx := declaration* prefix? start definition*

declaration :=
    "namespace" ANYIDENT "=" LITERAL
| "default" "namespace" ANYIDENT? "=" LITERAL

prefix = "datatypes" ANYIDENT "="
    "http://www.w3.org/2001/XMLSchema-datatypes"

start := "start" "=" pattern
definition := IDENT "=" pattern

pattern :=
    "element" nameclass "{" pattern "}"
| "attribute" nameclass "{" pattern "}"
| IDENT
| "empty"
| datatype? LITERAL
| datatype
| pattern ("," pattern)+
| pattern ("&" pattern)+
| pattern ("|") pattern)+
| pattern "?"
```

2 Details to any token class can be found in the Relax NG compact syntax specification.

3 The URI given in the prefix rule does neither belong to **KEYWORD** nor **OPERATOR** and is in fact a **LITERAL**, so parsers will have to accept any literal and check for the specific URI.
3.1 Structure: Relax CX and RCX’

| pattern "*"   |
| pattern "+"   |
| "(" pattern ")" |

nameclass := NAME

datatype := CNAME | "token" | "string"

Listing 3.1: Relax CX Grammar in EBNF

The reader should note that Relax CX is still a valid subset of Relax NG in compact syntax. Relax CX should be seen as the most relevant core of Relax NG with regard to structural pattern definitions for data, whereas most of the features for modular specifications, grammars, imports, extendibility and marked up text have been removed.

3.1.2 Differences to Relax NG

The major difference to Relax NG is the supported number of language features and also the form of the simplified abstract syntax. No matter in what form (XML or compact) a Relax NG specification is given, it has to be analyzed by a processor and transformed into the simple syntax (described in Section 8 of the standard).

This syntax is also subject to several limitations, which go beyond the structure of the simplified abstract syntax. These limitations are either checked on transformation (described in Section 7) or checked after transformation is done (described in Section 10, called Restrictions).

All these checks and restrictions apply to Relax CX specifications in the exact same form. Additionally such specifications should conform to the following restrictions:

Start Pattern Specifications always start with an element definition. So in contrast to Relax NG the start pattern cannot be a reference or a choice between several elements. It is therefore also not possible for the top level element to be part of a recursive definition.

Text, Mixed and List Both the text and the mixed pattern from Relax NG cannot be used in Relax CX. This work follows a data-centric approach, rather than a mark-up one, so the latter is unnecessary and I don’t allow mixing text patterns with other content in an element altogether.

This would leave the text pattern for attribute and element content only, which interferes with using proper datatypes instead. The data-centric approach additionally welcomes the absence of raw, untyped data.
With the same reasoning I don’t allow the list pattern, which is another form of structuring raw attribute or element content, by repeating values of a datatype.

**Datatypes and Values** Relax NG knows two built-in datatypes and additionally allows to use external datatype libraries. In Relax CX I allow only the four XML Schema Datatypes [14] string, token, integer and boolean. As I also do not allow datatype parameters, string and token are equivalent to the two built-in datatypes from Relax NG.

To maintain compatibility with Relax NG, integer and boolean have to be specified with a datatype library prefix in Relax CX. Many Relax NG tools have the XML Schema Datatypes URI predefined to the prefix xs or xsd so I allow any prefix in front of any of the four supported datatypes.

If the URI is not predefined, a prefix for the XML Schema Datatypes URI can be bound by using the prefix rule in the grammar. If such a prefix is specified, however, all datatype prefix have to match the defined one.

Relax CX does also not allow to create union types using the choice pattern or restrictions of other datatypes by using an except syntax. Each attribute or element therefore has exactly one datatype throughout its lifetime.

It is, however, still possible to construct enumerations as restriction from one of the four datatypes by using the choice pattern together with values of that datatype.

**Grammars, Includes and Not Allowed** Relax CX is stripped of all language elements which allow modular specification, the extension of existing specifications or the design of specifications which can or have to be extended later.

**Name Classes** Each element or attribute definition has an associated name class, which defines a class of QNames in which the concrete name in an XML document has to be in. In Relax NG this can be as simple as a single QName, but can also be an infinite name class or a restriction of an infinite name class. The total wildcard represents the class of all QNames, whereas so called namespace name classes represent all QNames with a fixed URI prefix.

Attributes with infinite name classes have to be repeated in Relax NG, i.e. they have to appear in a one-or-more pattern. Altogether infinite name classes can be used to allow elements or attributes of unknown content and structure in a document, i.e. content the current schema is not aware of. This is not an intended use of the structural schema language defined here, so Relax CX forbids infinite name classes and a name class is always a fixed QName.
3.1 Structure: Relax CX and RCX’

3.1.3 Abstract Syntax

A Relax CX pattern in its simplified form has the following abstract syntax, given as Haskell datatype:

```haskell
data Pattern =
  Empty -- Absence of content.
  | Choice Pattern Pattern -- A choice between two pattern.
  | Interleave Pattern Pattern -- An unordered group of patterns.
  | Group Pattern Pattern -- An ordered group of patterns.
  | OneOrMore Pattern -- A repetition of a pattern, at least once.
  | Data Datatype -- Any value of a given datatype.
  | Value Datatype String -- A fixed value of a datatype.
  | Ref String Pattern -- A reference to a pattern.
  | Attribute NameClass Pattern -- An XML attribute or element,
  | Element NameClass Pattern -- whose content matches the given
   -- pattern and whose name is in
   -- the given name class.
```

Listing 3.2: Relax CX Abstract Syntax

This abstract syntax differs from the one given by James Clark [3] in the following important points:

- The constructors NotAllowed, List, DataExcept and Text are absent as these are unsupported features.

- The constructor Value does not get a context, as Relax CX does not support any datatype that can make use of it.

- The constructor After is absent, as it is not used to represent a pattern but to handle so called “derivatives” which James Clark uses to validate a document.

- The abstract syntax of Relax CX has a Ref constructor, which represents a reference to another pattern. For further analysis it is important to explicitly maintain references and to not inline them into the pattern.

As mentioned earlier, the abstract syntax of Relax CX patterns is subject to the same simplifications and restrictions as the abstract syntax of Relax NG. This especially includes the following conditions:

- The Empty pattern only occurs on the left side of Choice to form an optional pattern and in Attribute or Element patterns to define that they must not have any content.

- After simplification is done, all references contain Element patterns and each element has its own definition, so this applies to the Ref constructor of the abstract syntax, too.
3 Language Definitions

• Data in an element is exclusive and never mixed with other content.
• Grouped or interleaved attributes never have overlapping name classes, interleaved elements never have overlapping name classes.

Note that many Relax NG restrictions are trivial to check for Relax CX or concern features which are no longer supported.

3.1.4 Limitations in RCX’

Relax CX, as presented till now, is still a very powerful language which can cause enormous troubles to static analysis. The long term goal of constrained XML is to support the Relax CX subset of Relax NG or come even closer to it. For this work, however, to be able to get first static results, I have to restrict Relax CX even further. This section describes all further limitations which together make up the RCX’ subset of Relax CX.

ID Attributes In a data centric approach and to utilize static analysis, it is vital that most nodes of a document have a unique identity, such that they can be uniquely selected. Like within a database, each data row of a table needs a primary key, whenever this key is omitted, the database most often can fall back on an internal id which is always present.

In XML we don’t have such an internal id, but all nodes get their unique identity by document order. While the role of document order is vital in marked-up text, or complex documents like programming languages, I judge it to be a poor mechanism to identify nodes in a data centric approach.

XML has the advantage compared to databases that there is much more structure present and not all classes of content can be arbitrarily repeated. In fact only Relax CX patterns contained in a one-or-more pattern or a recursive definition can produce arbitrary ambiguous nodes, which do not have to differ in the slightest.
To get hold of single nodes of such ambiguous patterns, I need to give the nodes an identity which is present in the actual data values of the node, not only implicit in document order. A node needs a **primary key**, like in databases, which uniquely identifies the node. In XML documents such a primary key should consist of a set of non-optional, unambiguous descendant nodes of an ambiguous node.

This can, for example, be a pair of *firstname* and *lastname* attributes or a single *id* attribute. Element nodes which contain only data values can also be used in such primary keys. The essence of this solution is therefore the specification of a uniqueness constraint, which was counted to the integrity constraint class\[4\] of the context conditions in [10].

The desired way of handling ambiguous classes of nodes would therefore be to specify primary keys, either in the structural schema or the context conditions. For the sake of simplicity, however, uniqueness in RCX’ is achieved by a simple *id* attribute, which will be interpreted in a special way.

**Intersecting Name Classes** The task to identify nodes of a document uniquely goes even further than dealing with arbitrary repeated nodes. Even if a pattern definition is only repeated a second time and grouped with the first one, the result is two ambiguous nodes, which again may only differ in document order.

To avoid such situations and additionally simplify the creation of XPath path expressions to select specific nodes, I impose more restrictions on intersecting name classes. Name classes of grouped and interleaved attributes, as well as interleaved elements already have to differ. In addition to that RCX’ forbids intersecting name classes on grouped elements and even in elements combined by Choice, i.e. which are mutually exclusive.

Such severe restrictions may seem unusual in the context of XML, but are completely common in domains like databases or classes in object-oriented languages. In the latter it is also forbidden to have different attributes with the same name, as they cannot be separated or selected anymore.

I also think that defining choices of elements with the same name is generally a bad design in a data centric approach. If such an approach is chosen for convenience reasons, as both options can be navigated through using the same XPath expressions, the different contents of those elements should be mutually exclusive only and all common parts should not.

For the foundations of Relax NG, these restrictions are even more severe than it might look like. The mathematical basis was also chosen to be robust against several operations and to allow common operations like the composition and the union of specifications. Such properties are void for RCX’, but on the other hand this isn’t the goal which is tried to be achieved.

\[4\]Section 4.4 goes into more detail about the different classes of context conditions and how they depend on each other.
3 Language Definitions

**Recursive Patterns**  Recursive pattern definitions are the most complicated source of ambiguous nodes in Relax CX. Although Section 4.2 covers many important details for recursive patterns, I leave them out of the RCX’ subset and concentrate on one-or-more patterns as main source for arbitrary many ambiguous nodes.

From a data-centric point of view, this restriction isn’t as severe as it looks like, as repetition by recursive patterns can always be transformed to repetition by one-or-more, using additional reference attributes. XPath CX, however, does not yet support filtering nodes by references, so supporting recursive pattern definitions will be a good way to support more expressiveness of data integrity, without opening up XPath CX too much.

### 3.1.5 Semantics

The semantics of both Relax CX and RCX’ specifications are defined in the Relax NG standard and are not altered by this work. Both languages are still strict subsets of Relax NG, so normal tools and algorithms can be used here. So with regard to conformity in the context of a CXLD, a document is called valid if it is valid with regard to the semantics defined in the Relax NG specification.

When it comes to manipulations of documents in algorithms, I do not allow to break structural validity completely. The allowed forms of inconsistency include the absence of necessary nodes only, nothing else. So the statitical verification of an algorithm against a structural Relax CX schema reduces to many small schema validations on XML fragments, which can essentially be done using the validation algorithm given by James Clark[3].

The structure of algorithms will additionally facilitate this by allowing to remove nodes only at the beginning of a method, followed by appending new XML fragments and last but not least updating single node values. This essentially means that all removals are structurally valid and lead to allowed inconsistencies, which can then be fixed by appending new nodes.

Each append has to be valid in its context, i.e. the top level element has to be allowed there, but the rest of the validation is done as usual by checking an XML fragment against an identified schema fragment. As RCX’ also restricts intersecting name classes, it is trivial to identify the correct schema fragment for this check.

The only concern left is the datatypes, which have to be guaranteed by XPath CX analysis and runtime checks in the case of `xs:token`. Section 4.7 gives some more remarks to verifying structural correctness for algorithms, which is not the topic of this section.
3.2 Base Language: XPath CX

In this work I not only define subsets of commonly known languages, but also define new languages on top of them. XPath 2.0 [16] has the special role of being used in two host languages and with being a multi-purpose expression language, it will be used to express a major part of specifications.

Additionally, XPath 2.0 alone is the most complex language used in this work and that justifies to take a deeper look at the structure of XPath expressions and to define severe limitations for their use. In this section I will go through the abstract syntax of XPath 2.0, to visualize what XPath can do, and then define the **XPath CX** or **XCX** language, which should be read as “XPath for Constrained XML”.

3.2.1 Lexical Structure and Grammar

Both the lexical structure and the grammar of XPath CX remain completely unchanged and are as described in Appendix A of [16]. The grammar of the concrete syntax is highly optimized to utilize implementations, but is in no way suited as abstract syntax.

3.2.2 XPath 2.0 Abstract Syntax

XPath 2.0 is a highly flexible and dynamic language. The concrete syntax given in [16] can be difficult to read and understand, as it is optimized for parsing and encodes operator precedences and parenthesis rules by long chains of left recursive rules.

Essentially, XPath 2.0 allows to combine every possible subexpression with every other language construct. There are very few restrictions imposed by the grammatical structure. This goes along with the fact that XPath 2.0 is dynamically, strongly typed and does a lot of implicit casting.

This, of course, is a nightmare for static analysis, as it is in most situations completely unknown what type an expression really has. XPath 2.0 also knows static typing, but this applies mostly to function calls and with restrictions to operators. One major pitfall is the fact that XPath 2.0 knows sequences and virtually every expression can be used as a sequence.

A fact that is often exploited is that every XPath expression can be, often implicitly, cast to boolean. While this is very convenient for rapid development, it is again very difficult to analyze such expressions.

Listing 3.3 shows the core datatype of the XPath 2.0 abstract syntax I use, which models the complete XPath 2.0 language and furthermore guarantees that every element of the datatype represents a parsable XPath 2.0 expression. In contrast to the W3C grammar, the **Expr** datatype focuses on the logical
3 Language Definitions

structure of expressions and makes the recursive dependencies of most language constructs obvious.

| data Expr = |
| NumericLiteral Numeric <non-recursive cases:>
| StringLiteral String <literals (numeric or string)> |
| VarRef QName <variable references> |
| EmptySequence <the empty sequence> |
| ContextItem <the context of the expression> |
| FunctionCall String [Expr] <built-in function calls> |
| Seq [Expr] <a sequence of expressions> |
| For [VarDef] Expr <sequence transformer> |
| Quant QuantifierType [VarDef] Expr <quantified expressions> |
| If [Expr] Expr Expr <conditional expression> |
| Operator Binop Expr Expr <binary operators> |
| Instance Expr SequenceType <typing related expressions> |
| Treat Expr SequenceType |
| Castable Expr SingleType |
| Cast Expr SingleType |
| UPlus Expr <unary operators> |
| UMinus Expr |
| Path Expr Expr <sets the context item to the result of the first expression and evaluates the second> |
| Filter Expr [Expr] <applies a list of filters to a sequence expression> |
| ForwardStep ForwardAxis NodeTest <applies a forward/reverse step and special filter to all nodes of the context> |

| data Numeric = |
| Integer String |
| Decimal String |
| Double String |

data VarDef = VarDef (QName, Expr)

Listing 3.3: XPath 2.0 Abstract Syntax - Main Types

The long list of binary operators is condensed to a single case in the Expr datatype, whereas the W3C defines them, like all other constructs, as long cascade of left-recursive rules. Listing 3.4 defines all supported binary operators in XPath 2.0.
3.2 Base Language: XPath CX

Listing 3.4: XPath 2.0 Binary Operators and Comparators

One of the most important features of XPath 2.0 is the ability to query XML documents and navigate within its nodes. The constructors Path, Filter, ForwardStep and ReverseStep are the cornerstones of this ability and are backed up with corresponding axis and node test definitions.

Listing 3.5: XPath 2.0 Forward/Reverse Axis and Node Tests

Listing 3.5 shows all supported forward and reverse axis, as well as the node test framework. The KindTest datatype is also of great importance for the XPath 2.0 type system, as these constructors can all be used to specify sequence
3 Language Definitions

types.
Considering the restrictions defined in Section 2 it is clear that many of the kind tests supported in XPath 2.0 can be discarded. The PITest, for example, describes processing instructions, which are unsupported by the data model.

3.2.3 Restrictions

If we want to be able to analyze algorithms and context conditions, both formulated in a language based on XPath, we need to restrict the usage of XPath severely. The static information that can be extracted from arbitrary XPath expressions, even in the presence of a structural schema definition, is nearly non-existent.

Filters One of the major problems are filters. A filter takes a primary expression and a sequence of arbitrary other expressions, which are tested against each item of the primary expression (interpreted as sequence). In most cases this means that a filter expression is evaluated in the context of each item of a sequence and then cast to a boolean value, so it can be decided if the value remains in the sequence or not. Some filter expressions, however, have a special meaning, like integer values which select an element by its position in the primary expression sequence.

To select the 17th node of a document, for example, we can use the valid XPath 2.0 expression \[//\text{node()}[17]\]. There isn’t much that can be statically said about this expression, even in the presence of a structural schema any kind of node could be selected. Once an arbitrary node can be selected we loose any information about the result of an expression.

The position of a node in a sequence of nodes in document order is an information I do not allow to be used, so such filters have to be forbidden. But even then filters are a far too powerful construct to handle with static analysis. Any single property of a given document, ranging from presence and absence of certain nodes to the values of each attribute or element, can be used in a filter expression so in the worst case the resulting sequence of a filter can only be known after reading the complete content of a document.

In the context of this work, the only filters which are therefore allowed are filters which select specific nodes from a document. A filter expression should select descendant nodes of the filtered one and assert properties of these nodes, like their existence or their value. This allows to select specific data items or sequences of items to work with. But even such restricted forms of filters get arbitrary complex and can be hard to analyze statically.

It turns out that filters are not vital to define first meaningful context conditions and algorithms, except for selecting unique nodes in a document. Together with the decision to allow only special \text{id}\ attributes to assert uniqueness of nodes, filters can only be used to test \text{id}\ attributes against fixed values.
3.2 Base Language: XPath CX

Typing  A simple path expression, which selects some nodes from a document, can be used in various different ways. It can be used as a node in a count function, but also as value for an arithmetic binary operator. It can be cast to a boolean value and is always usable as sequence. If it is required to be exactly one node or value, this also imposes no problem, as long as the dynamic evaluation calculates exactly one node, which obviously depends on the document.

To facilitate analysis I define XPath CX to be a completely statically typed language. Each XPath CX expression will have a fixed type, which is statically guaranteed and the expression cannot dynamically be used in any other way or even be cast to other types. This approach is especially valid as the analysis of XPath CX expressions always has a structural schema definition as context, which gives a lot of vital information about path expressions.

The atomic datatypes allowed are those supported in Relax CX, i.e. boolean, integer, token and string. I will refer to these XML Schema datatypes as kinds, as the actual types of XPath CX contain more information. XPath CX also has a notion of sequences, which is necessary to allow functions like sum to work and also to support path expressions. The last part of an XPath CX type tells if the expression is guaranteed to also be a node or if this is not the case.

Integer literals, for example, are of the kind integer, are obviously not a sequence and are not nodes. An attribute node defined as integer in Relax CX, however, has the same kind and sequence information, but can also be used as node. An inner node of a document has the special kind complex, so it is incompatible with a context that assumes any atomic kind.

Some of the language constructs, which are used to dynamically deal with statically untyped expressions are of course unsupported and totally unnecessary. The castable operator, for example, is used to dynamically check if a value can be cast to a given datatype. As casts are in general not allowed and all expressions have a fixed type, the operator is unsupported. Casts might be necessary in some expressions though, to bridge the gap between Relax CX types and XPath.

Mixed Expressions and Sequences  In arbitrary XPath expressions, path steps and normal operators can be mixed extensively, which leads to very complex expressions and often to unused subexpressions. The expression //item/5, for example, returns a sequence of integer literals, even tough a path expression is used to create it. Such use of path expressions, which do not actually select any part of the document as a result, are undesired and therefore forbidden.

Every path in XPath CX is indeed a reference to the document being processed and can be mixed in completion with other XPath constructs. Sequences in XPath CX are therefore not as liberal as in XPath 2.0, where sequences can be arbitrarily created, transformed and filtered.

A path expression in XPath CX can only contain other path steps, axis steps
3 Language Definitions

and filters, so other expressions can only appear within filter expressions. These, however, are restricted to simple id attribute tests, but can be widened again in the future to allow more flexible expressions.

The empty sequence is then obsolete and is unsupported, as is the sequence operator for general cases. On node sequences this operator might be supported, but it is preferable to just use the union operator on those.

Functions and Compensations XPath has a lot of built-in functions that can be used in expressions, which are associated with the different datatypes of XPath. A lot of these functions are simply not supported because one or more of the necessary datatypes are unsupported in XPath CX.

But even then I have to disallow most other functions and pick the supported ones carefully. The major problem is that it will be necessary to compensate the effect of any given XML node on any given expression. This is the case since it is impossible to create nodes in XPath.

If a node is appended to an existing document and a context rule includes this new node, I have to create a test expression which checks the condition before the operation is even done. This means that any sequence including the new node would need to be adjusted, which is simply impossible, as no value in XPath can be used to compensate for the fact that the node is still not there.

So the only chance is to understand what properties of a node are used and compensate the effects of the node only. A sum, for example, is obviously only defined on nodes that contain a value and once this is certified, I can compensate for the node by adding its new value to the sum.

Such compensations are different for each function and can be done with varying difficulty. In the prototype of XPath CX presented in this work, I will support only very few functions, but this set can easily be widened in many cases.

Quantifier and For The two quantifiers some and exists, as well as the for construct all deal with sequences of values. I don’t want arbitrary sequences in XPath CX for now, but the three operators might still be valid on node sequences.

For this work I judge all three to be unnecessary complex and it is possible to define useful expressions without them. They are interesting to look at for further enhancements of Relax CX, but they also need more sophisticated analysis techniques.

3.2.4 XPath CX Abstract Syntax

Being a restriction of XPath 2.0, XPath CX uses the same lexical structure and in fact also uses the same parser and initial abstract syntax. An expression is
then analyzed, annotated with XPath CX types and restricted to a narrowed syntax. This syntax is subject to change in the future, as it is very dependant on the supported set of language features and especially the supported functions.

<table>
<thead>
<tr>
<th>data</th>
<th>Type = Type</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>{ atomic :: Bool — an atomic value, i.e. not a node</td>
</tr>
<tr>
<td></td>
<td>, simple :: Bool — one simple item, not a sequence</td>
</tr>
<tr>
<td></td>
<td>, kind :: Kind</td>
</tr>
<tr>
<td></td>
<td>}</td>
</tr>
<tr>
<td>data</td>
<td>Kind =</td>
</tr>
<tr>
<td></td>
<td>TInt</td>
</tr>
<tr>
<td></td>
<td>TComplex — special type for inner and empty nodes</td>
</tr>
<tr>
<td>data</td>
<td>TypedExpr = TE Expr Type</td>
</tr>
<tr>
<td>data</td>
<td>Expr =</td>
</tr>
<tr>
<td></td>
<td>Unknown String — e.g. parameters of functions</td>
</tr>
<tr>
<td></td>
<td>Literal String — the type of the literal is no longer embedded</td>
</tr>
<tr>
<td></td>
<td>FTrue — the boolean values true and false</td>
</tr>
<tr>
<td></td>
<td>FFalse — (these are function calls)</td>
</tr>
<tr>
<td></td>
<td>Union TypedExpr TypedExpr — defined on paths only</td>
</tr>
<tr>
<td></td>
<td>FNot TypedExpr — unary boolean function</td>
</tr>
<tr>
<td></td>
<td>BinOp BOp TypedExpr TypedExpr — boolean/integer operators</td>
</tr>
<tr>
<td></td>
<td>UnOp UOp TypedExpr — integer operators</td>
</tr>
<tr>
<td></td>
<td>Comp CompOp TypedExpr TypedExpr — only defined for atomic</td>
</tr>
<tr>
<td></td>
<td>— kinds, i.e. non-complex</td>
</tr>
<tr>
<td></td>
<td>If TypedExpr TypedExpr TypedExpr — the result of if cannot</td>
</tr>
<tr>
<td></td>
<td>— be used as node, i.e.</td>
</tr>
<tr>
<td></td>
<td>— as value only</td>
</tr>
<tr>
<td></td>
<td>Node DocRef — a value as reference into the document</td>
</tr>
<tr>
<td></td>
<td>Sum DocRef — a sum of referenced node values</td>
</tr>
<tr>
<td></td>
<td>Count DocRef — the node count of referenced nodes</td>
</tr>
<tr>
<td></td>
<td>Locked TypedExpr — internal use to prevent rewriting</td>
</tr>
</tbody>
</table>

Listing 3.6: XPath CX Abstract Syntax

As can be seen in the abstract syntax, the constructors Path, Filter, ForwardStep and ReverseStep are absent and replaced with document references. These contain all the information of the former path expressions, using the additional information of a Relax CX structural schema.

As motivated in the previous section, this also means that path expressions can only result in references into the document, not atomic values or arbitrary sequences. For XPath CX, these expressions collapse to a single value, no matter if node or node sequence, which also has a fixed type.
3 Language Definitions

The rest of the language is a very restricted subset of XPath 2.0, with many features missing. Quantifier, the for construct and also generic function calls are notably missing, only a few selected functions are present and treated in a special way. It will be possible, in some cases even trivially, to extend XPath CX and allow more and more language features, operators, functions and also more datatypes.

The presented subset of XPath 2.0, however, is sufficiently large to express various interesting context conditions and enough algorithms to work with XML documents. As this work is the prototype for constrained XML documents, algorithms and static analysis results, more complex language features can be added later, following the first results achieved with this subset.

Document references will be introduced in detail in Section 4.2.4. The conversion of XPath 2.0 to XCX and the type analysis, together with all necessary restrictions, will be covered in Section 4.3.

3.3 Context Rules: CRL

As rule-based constraint definition language I will define a new language, which is inspired by Schematron [6]. Schematron follows an XSLT [12] based approach to define rules which can match XML documents at their specified context and then checks conditions given as XPath expressions.

In Schematron each part of a document can only be matched by one most-specific rule, which is also inherited from XSLT. While this semantics make sense for transformers, it is undesired behavior for checking arbitrary many constraints on a document.

Therefore I will define a simple context rules definition language based on arbitrary XPath 2.0 [16] subsets, called Context Rules Language or CRL. The language will allow to define arbitrary many rules which can match multiple times at one given document and are allowed to check conditions potentially using the full strength of XPath 2.0 expressions.

The definition of the language does not depend on a concrete subset of XPath 2.0 used, so CRL can essentially be used with any. In this work, however, I will stick to the XPath CX subset and utilize the special restrictions for the analysis of CRL.

3.3.1 Lexical Structure and Grammar

Being a host language for XPath 2.0, which constitutes a major part of specifications, CRL uses the same lexical structure as XPath 2.0. As with Relax, this means that whitespace and comments are insignificant and often have to be used to separate other tokens. Appendix A.2 of the XPath 2.0 specification [16] gives all details regarding the lexical structure.
3.3 Context Rules: CRL

Additional keywords used in CRL, but not in XPath, belong to the \texttt{NCName} token class and are handled by requiring a specific \texttt{NCName} representing the keyword in the parser. The use of such keywords in the grammar is never ambiguous to occurrences of the \texttt{NCName} token class. Additional operators do not belong to any token class of XPath and therefore can’t appear ambiguously.

At present state CRL does not need any additional keywords and only uses the single exclamation mark as additional operator.

The CRL grammar is given in EBNF, using members of \texttt{KEYWORD} and \texttt{OPERATOR} as well as additional keywords and operators directly as terminals, all other token classes are used like external grammar rule definitions. The special reference \texttt{xpath} refers to the EBNF of XPath 2.0 and is not defined here.

\begin{verbatim}
contextrules := rule*
rule := NCNAME "(" xpath ")" vardef* check*
vardef := ":" ":" QNAME ":" xpath
check := "!" xpath
\end{verbatim}

Listing 3.7: Context Rules Language Grammar in EBNF

3.3.2 Abstract Syntax

The abstract syntax of CRL is as simple as the concrete one and quickly given as Haskell datatype definitions:

\begin{verbatim}
data CRules = ContextRules [Rule]
data Rule = Rule String XPath [Variable] [Check]
data Variable = Variable QName XPath
data Check = Check XPath
\end{verbatim}

Listing 3.8: Context Rules Language Abstract Syntax

3.3.3 Semantics and Restrictions

The semantics of a context rules specification are defined for the use of the language as part two of a Constrained XML Language Definition. A document is called valid with regard to a CXLD if it conforms to both the structural definitions and the context conditions. So the relevant part for CRL is the conformity of a document to a set of rules.

A concrete XML document conforms to a context rules specification if it conforms to all rules of the specification individually. To conform to a rule, the rule has to be interpreted in the following way, using standard XPath 2.0 technologies.

There is no concept of limited inconsistency for CRL. A document is either valid regarding a context rules specification or not. In the middle of an algorithm
execution several rules will be violated, which is both necessary and allowed to do meaningful operations. If an algorithm never violates any rules it is trivial to check statically.

**Context** The XPath expression given in parenthesis after the rule name is called *context* and is evaluated on the given XML document. It is intended to evaluate to a sequence of nodes, which are each tested one after another. Without the use of a structural pattern, it can statically not be asserted that a rule context only selects nodes. The same is true for expressions using full XPath 2.0, as it is impossible to say in general that an expression always results in a sequence of nodes.

The analysis section of CRL will be able to check context rules statically for selecting only nodes, but for the general language it is not important. Each item of the context sequence instantiates the body of the rule once. The current context item is available in the *dynamic context* of XPath 2.0 under the variable $context$ and the *context item*.

**Variables** A variable in CRL is defined using the colon operator, together with the equal sign. A variable name always starts with the dollar symbol and the content of the variable is the result of an XPath expression.

Using the dynamic context created on rule interpretation, all variables are evaluated in definition order and their result is bound in the dynamic context, using the variable name specified in the variable definition. Variable can therefore be used from the following variable definition onwards and especially in the check expression at the end of the rule body.

Multiple declarations of variables with the same name are not allowed and the variable name *current* is forbidden, as the context item is bound to it.

Note that many common XPath 2.0 interpreters do not support to transport internal representation of values to subsequent evaluation calls, so only atomic values which have a string representation can actually be evaluated and bound to variables. As XPath and CRL are completely read-only, variable definitions can just be inlined and evaluated in place. Variables are like macros and used for convenience only.

**Checks** Check expressions are defined using the exclamation mark and consist of an XPath expression which is evaluated as boolean value. For an item selected by the context to conform to the body of the rule, all check expressions, no matter in what order, have to evaluate to *true*, using the dynamic context defined after all variables are interpreted. If a check expression does not result in a boolean value it is cast to boolean, which is always possible in XPath.
**Static, Dynamic and Type Errors**  The evaluation of XPath expressions can yield various kinds of errors. If a check fails without result, for any type of error, it is counted as failed. It is of course desirable to define checks which include no static errors and static type errors, as no document can fulfill such a context rule.

### 3.3.4 XPath CX Specific Restrictions

Depending on the concrete subset of XPath used in CRL, more kinds of errors can be excluded statically, i.e. which can be checked on the specification level rather than at runtime when a concrete document is checked. If the goal is furthermore to statically guarantee that operations on valid concrete documents maintain validity for these documents, it is necessary to assert that none of the error classes of XPath occurs on context rule checks.

The usage of XPath CX guarantees the absence of type errors, whenever embedded XPath expressions are indeed valid XPath CX expressions. The static analysis of XPath CX is defined against a fixed RCX' specification, which means that CRL specifications are also defined against one specific RCX' specification. This allows to define that certain applications of XPath in CRL have to result in statically unique or narrowed document references.

The context expressions of CRL are required to result in either a node sequence or a single node, whose document reference includes only node classes which can be made unique by filtering. Rule checks need to result into simple boolean values. It is a specification error of CRL if any of these conditions is violated.

### 3.4 Manipulations: AL

Last but not least, I need a small language to perform basic manipulations on XML documents. As there are no suitable languages available and I also need a very specific language, which is simple enough to analyze, I define a small algorithm language or short AL.

The basics of the language are small operations, which change local nodes of an XML document in a unique context. There will be no batch operations or complex transformations available in the first prototypes. Starting from very small operations, it is possible to define greater operation blocks, which can be combined using conditional control structures like if.

A long term target is of course to support loops and operations on sets of nodes, but these are much harder to analyze and are not necessary to define first useful algorithms. With the algorithm language defined here it should already be possible to define useful applications on documents which are valid regarding a CXLD.
AL can be used with any subset of XPath, which is for the most part not important for the definition of AL itself. In this work I will use AL together with the XPath CX subset, as it allows to guarantee the necessary properties to assert that algorithms execute and terminate properly.

### 3.4.1 Lexical Structure and Grammar

AL uses the same lexical structure that CRL does. Both are host languages for an XPath subset and both need only very few additional keywords or operators. For a full description of the token classes and other details see Section 3.3.1.

Like usual, the AL grammar is defined using EBNF where members of **KEYWORD** and **OPERATOR**, as well as additional keywords and operators, appear directly as terminal symbols. All other token classes do not assert fixed members in rules and appear as external grammar rule definitions. The definition **xpath** refers to the EBNF of XPath 2.0, which is not defined here.

```plaintext
algorithms := method*

method :=
    NCNAME "(" ( param ("," param)+ )? ")" "{"
    commandblock
"}"

param := atomic NCNAME
atomic := "int" | "bool" | "string" | "token"

commandblock := variable* node* command

variable := ( "path" | atomic ) NCNAME "=" xpath ";"

node := "node" NCNAME "=" element ";"

command := if | ops

if :=
    "if" "(" xpath ")" "{"
    commandblock
"}"
(if "elseif" "(" xpath ")" "{"
    commandblock
"}" )*
(if "else" "{"
    commandblock
"}" )?

ops := remove* append* update*

remove := "remove" "[" xpath "]" ";"
append := "append" "[" xpath "]" element ";"
update := "update" "[" xpath "]" xpath ";"
```
3.4 Manipulations: AL

<table>
<thead>
<tr>
<th>element := newelement</th>
<th>newattribute</th>
<th>&quot;#&quot; NCNAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>newelement :=</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&quot;&lt;&quot; QNAME newattribute* &quot;/&quot; &quot;&quot;&gt;&quot;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&quot;&lt;&quot; QNAME newattribute* &quot;&quot;&gt;&quot;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>newelement*</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&quot;&lt;&quot; &quot;/&gt; QNAME &quot;&quot;&gt;&quot;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&quot;[&quot; xpath &quot;]&quot;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>newattribute := QNAME &quot;=&quot; &quot;[&quot; xpath &quot;]&quot;</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Listing 3.9: Algorithm Language Grammar in EBNF

The grammar has one further restriction which I count to syntax, but is not expressed in EBNF, which is that the closing tag of a newelement has to be the same QName as the corresponding starting tag. Depending on the used parser technology, such a restrictions can be expressed in the grammar itself.

3.4.2 Abstract Syntax

The abstract syntax of AL is very similar to the concrete one and no special transformations or abstractions need to be done.

| type Ident = NCName |
| data Algorithms = Algorithms [Method] |
| data Method = Method Ident [Parameter] CommandBlock |
| data Parameter = Parameter Ident AtomicType |
| data AtomicType = ATString | ATInteger | ATBoolean | ATToken |
| data CommandBlock = CBlock [Variable] [Node] Command |
| data Variable = Variable VType Ident XPath |
| data VType = |
| VPath |
| VAtom AtomicType |
| data Node = Node Ident Element |
| data Element = |
| ElementRef Ident |
| TrueElement NewChildNode |
| JustAttribute NewAttributeNode |
| data Command = |
| If XPath |
| CommandBlock |
| [(XPath, CommandBlock)] |
| (Maybe CommandBlock) |
| | Ops [Remove] [Append] [Update] |
3 Language Definitions

```haskell
data Remove = Remove XPath
data Append = Append XPath Element
data Update = Update XPath XPath

data NewChildNode =
  NewElementNode QName [NewAttributeNode] [NewChildNode]
| NewTextNode XPath

data NewAttributeNode =
  NewAttributeNode QName XPath
```

Listing 3.10: Algorithm Language Abstract Syntax

3.4.3 Semantics and Restrictions

The methods defined in algorithm specifications will be the point of interest for static analysis, as the goal is to show that any valid document, regarding a fixed CXLD, stays valid after executing a method, no matter what values the parameters are set to. This is of course impossible in general, so the smallest and shortest possible number of runtime checks is of interest, after which the method can be safely executed.

The language is therefore kept simple, with the three operation types being the cornerstones of manipulations. As seen in the grammar and abstract syntax, I keep a fixed order on the different types of operations, to keep analysis efforts at minimum for a start.

Algorithms and Methods  Each method is completely independent of all others, so there is no need to define them together. This will change in the future, when method calls will be a language feature, but for now we can assume each algorithm specification to consist of a single method.

The name of the method is an external identifier, used to identify the method, together with the number and type of parameters. For the semantics of the method it is irrelevant, however.

As with the context rules language, we also start with a fresh dynamic XPath environment when processing a method. The parameters of the method are then bound to variables of the same name, with empty prefix.

Parameters  All variables need to have different names or this is a specification error. Parameter values are given as plain strings, which are cast to the specified types in XPath.

The types `int`, `bool`, `string` and `token` correspond to the XML Schema types `xs:integer`, `xs:boolean`, `xs:string` and `xs:token`, which are also the supported types in Relax CX.
Command Blocks, Variables and Nodes  Each command block gets its initial XPath environment from the enclosing structure, which is either the method itself or a command in another command block. Starting with this environment, where at least the parameters of the method are bound in, the variable definitions are bound to the corresponding variable names.

This is done in definition order, from top to bottom, so variables can be used as soon as they are defined, even in other variable definitions. The scope of the variables is unlimited and variables cannot be redefined. For the same reason no variable can have the same name as a parameter of the method.

The types available for variable definitions are the four atomic types supported by Relax CX and which are available for method parameters. In addition to that it is possible to define paths, which select one or more nodes of the document.

Note that XPath is a read-only query language, but AL by design is not. Additionally, when using common XPath libraries, expressions that result in XML nodes cannot be assigned by value to variables for other XPath expressions, as they are not atomically typed. So it is usually impossible to evaluate variable definitions in place and save the resulting values.

Variables can therefore be seen as macro definitions or abbreviations and their definition is virtually inlined in their usages. They are there for convenience reasons, to shorten specifications by eliminating common subexpressions and to define additional type constraints, which lead to more static security for the programmer.

Treating variables like macros does not conflict with the nature of AL, where input documents are manipulated. This is due to the fact that variables of atomic types can only be used in expressions that are evaluated on the starting document and path variables can also be used in these contexts, which is no problem as well. If they are used in operations contexts, however, they need to specify exactly one existing node which always needs to exist, so they effects of manipulations are very limited and explicitly taken care of.

That being said, variables do in no way impact the possibilities of AL, which is also true for the so called node definitions. As XPath does not offer any support for the creation of XML document fragments and only supports nodes as temporary values in expression evaluation, I need another possibility to create nodes to append to an existing document.

Node definitions are labelled with names, which can be referred to later. All names for nodes have to be different, as it is with variables, but node names have their own scope and are actually not referred to from within XPath expressions. The parameter names of the method do not interfere with node names.

Nodes are defined in simplified XML syntax, which only allows element and attribute definitions. As it is also necessary to create single new attribute nodes, without an enclosing element, it is possible to do so by specifying the attribute as usual as an equality of attribute name and value.

All attributes and elements that have an atomic type, i.e. which are neither...
Language Definitions

inner elements nor empty, have their value assigned using XPath expressions. To separate the scope of the XPath expression from the node definition, it is enclosed in brackets. The dynamic context of these expressions is the current environment of the node definition.

Each command block then has a resulting XPath environment, as well as a dictionary of node definitions, which is passed on to the embedded command. The method itself starts with an empty dictionary of node definitions.

Commands At the moment there are only two types of commands, which are the essential operation blocks and the if construct to express options in the method execution, which can depend on document or parameter conditions. An additional command could, for example, be a for or map construct, which applies a command to a set of existing nodes. Such batch operations could be used to work with nodes that cannot be uniquely referenced.

The if construct is an arbitrary long list of command blocks which are all guarded by an XPath expression. Only the last command block, if contained in a simple else, does not need to have a test, which is interpreted as always true in this case.

The test expressions are evaluated in definition order in the current dynamic context and cast to boolean. As soon as the first result is true, the corresponding command block is chosen and all others are discarded. If no test is true and no else is specified, the method does not do anything and terminates. The selected command block inherits the environment and node dictionary from the if.

An operation block is a list of primary operations, which are evaluated in definition order. The environments used for each operation are those from the enclosing language element and they cannot be changed anymore. An operation block is always the last element in a method execution.

Operations The three basic operations are the only source of manipulations. Both their intention and their result is a new version of the document which was the input to the operation. When executing an operation block there are always two current documents of interest, the starting document which gets changed over and over again to the final result document and a copy of the starting document which stays fixed.

The reason is that we want to be able to apply operations to documents which were already changed by previous ones, while still being able to refer to removed information. For this reason the path expression which describes the context of an operation is interpreted on the current, mutable and steadily modified document, while expressions defining values to be placed in constructed elements or used in updates are evaluated on the starting document.

\footnote{As motivated earlier, such information cannot be saved in variables.}
The **remove** operation takes only one context expression, which simply specifies all nodes which should be removed. The context does not need to specify a single node only, but it is required that all elements in the sequence are indeed nodes of the document. The result document after executing a remove operation has all members of the context sequence removed.

The **append** operation takes a context expression and an XML fragment as parameters. The context expression needs to identify exactly one node of the document, which is the target of the append. It is an error if the expression returns an atomic value, a sequence or the empty sequence.

The XML fragment defined can be a single attribute node or an element node, which in turn can include more attribute and element nodes. A reference to a node definition is resolved using the dictionary from the environment. If the reference is not defined there it is also an error. All values in the constructed XML fragment are defined by XPath expressions, which are evaluated on the starting document.

The resulting document has the defined XML fragment added to the end of the list of children of the unique context node. As document order is insignificant in this work, it is not necessary to offer more sophisticated position specifications, but the reader should be aware that Relax CX schemata have to tolerate appending different optional nodes in any order.

The **update** operation also takes two parameters, but these are both XPath expressions. As with append operations, the first needs to specify a unique node of the current document. The difference is that the selected node additionally must not be an internal node of the document structure, so either an attribute node or an element node which contains only text.

The second expression is evaluated on the starting document and has to result into an atomic value or a node containing an atomic value. The resulting document has the content of the specified context attribute or element replaced with the resulting value of the second XPath expression.

**XPath Errors**  Static, dynamic and typing errors are of much more significance than for CRL. In CRL a rule is simply said to be violated if the check expressions do not terminate regularly. For AL, however, an abnormal termination can mean that a document is left in an invalid state. As this is exactly what we try to prevent with static analysis, it is crucial that the used subset of XPath for AL allows to give strong static guarantees.

For the language itself, an abnormal termination of an XPath expression evaluation leads to the termination of the current method. The same goes for each of the limitations defined in the previous paragraphs, for the result of specific expressions. Whenever the context of an append operation results in a sequence of elements, for example, the method execution is also terminated.
3 Language Definitions

3.4.4 XPath CX Specific Restrictions

Some restrictions of AL can be formulated with the vocabulary of the used XPath subset only. In this work I am using XPath CX as subset in AL, which both allows to formulate existing restrictions more precisely, but also means that some additional restrictions have to hold.

**ID attributes** XPath CX allows to filter node paths only by comparing id attributes with fixed unknown values. This restriction therefore also applies to the creation of new XML element nodes with such attributes. Whenever an ambiguous element with an id attribute is created, this id attribute has to be a fixed unknown value as well. For AL this means that only method parameters can be inserted into such id attributes.

This allows to statically compare filter values, without the need to dynamically compare them and generate conditional checks based on the outcome of filter comparisons. For the same reason, however, all parameter values of a method need to be different, to avoid having path expressions with different filter values reference the same node. This restriction is of course tedious and can most likely be dropped as soon as the analysis gets more complex.

**Operation Contexts** The context expression of append and update operations need to be unique nodes, in the former case of complex type, in the latter of any atomic type. By using XPath CX, which is in turn based on Relax CX, this restriction translates to the type of the context expressions being simple and node, whereas the kind has to be complex for appends and any kind but complex for updates.

This also means that both context expressions are XPath CX Nodes, which contain a document reference. This document reference is not only required to be unique, but it has to be statically unique\(^6\). Static analysis would be much more complicated, without this additional constraint, and need more runtime checks and conditional checks depending on their outcome.

Update operations are also restricted to not change any id attributes. They have the special role of primary keys in XPath CX and RCX\(^7\) and allowing to change them is an unnecessary feature at the moment, which only causes more dynamic checks to be created from static analysis. It would be possible to allow changing non-optional id attributes or id attributes of elements which are not ambiguous, but again this results in unnecessary complex checks in the context of this work.

**Removes** XPath CX introduces some severe restrictions to node selection by limiting filters, to be able to statically identify unique nodes when necessary.

\(^6\)See Section 4.2.4 for definitions of these terms.
For the semantics of AL in general, remove operations can remain as liberal as they are defined above and their context can include any number of nodes.

But on using XPath CX, is makes sense to restrict remove operations to two usual cases and preserve the good statical analysis conditions established by Relax CX. The context of a remove operation therefore needs to satisfy any one of the following two conditions:

- The context specifies only *optional node classes*, i.e. which are directly contained in a one-or-more pattern or in a choice pattern whose other child is the empty pattern. In this case the context might as well be a sequence of nodes.

- The context is a *statically unique reference*.

This restriction prevents, for example, that a sequence of nodes, which are each non-optionally embedded in other nodes are deleted. Such an operation would need to be compensated by adding nodes to all such existing nodes in the document, which is hard to handle by static analysis.

So the two options basically allow to delete optional nodes, which don’t need any compensation or nodes which can statically be identified. Another consequence of this restriction is that lists of ambiguous nodes, which do not have id attributes, can only be removed altogether, never individually.

**Types and Variables** Using XPath CX in AL allows to benefit from the static typing and to exclude XPath type errors altogether. Whenever an XPath 2.0 expression is a valid XPath CX expression, it is known to not contain any type errors. AL additionally profits by asserting types of variables, which can normally not be statically checked in all cases.

The declared types of variables have to match the kind of the resulting typed expressions. The special type *path*, however, asserts that the *node* flag is set and ignores the *kind*. Such paths do not need to have a complex kind, as leaf nodes most often are not complex and can still be used as node.

The type of expressions used in other contexts can now also be statically asserted. Expressions used as tests in *if* and *elseif* expressions need to be simple boolean values. Expressions used as values need to be *simple* and share the *kind* with the node they are associated with. Contexts of operations are always nodes, not *atomic* values.
3 Language Definitions
4 Static Analysis

In this section I will first go through all languages and define ways to analyze specifications in more detail and retrieve more information. This will often be done by using the concrete language subsets used and put their special restrictions to good use, e.g. to find more classes of specification errors. Each instance of XCX, RCX, CRL and AL can be statically analyzed to reveal such errors and to provide necessary information to further analysis steps.

The second part will then combine the result of an AL analysis together with the result of a CXLD analysis, to get to preconditions which have to hold for the algorithm to execute properly and keeping the CXLD valid on the result document.

The analysis of XCX and RCX are especially important, as both CRL and AL rely on the analysis results of both. Each CRL or AL specification is made against one RCX specification, which constrains the structure of all valid documents and therefore the form of XCX expressions used in these specifications.

4.1 Comments on Basic Techniques

Many of the techniques used in subsequent sections can be defined in common programming languages without too many troubles, but at some points it is necessary to use non-standard technologies to keep things on a manageable and concise level.

All languages, and most parts of the static analysis algorithms defined here, are implemented in Haskell and can be found in Haskell modules on the electronic medium. The sources are well documented using both Haddock \[8\] and normal comments. A HTML version of the Haddock comments, which also gives a good overview over the module structure, is included on the medium. The sources are split over nearly 40 modules and range over more than 7000 lines of code, including whitespace and comments.

As it is impossible to include all defined functions and datatypes in this work, I will stick to extracting the most vital or interesting functions, which often give an insight on the techniques used. Furthermore there are technical details, which can be conveniently solved by using advanced technologies, which are not described in this work.

These most notably include generic programming \[7\] and the concept of term positions \[9\][1]. As I will also use infinitely large datatype values, it is necessary
4 Static Analysis

to adapt both technologies to deal with it.

A special module for generic programming on infinite datatypes is included on
the medium, which contains adapted versions of the everywhere and everything
primitives to allow the definition of special termination conditions. I will also
define term positions on infinite data values, which results in infinitely large
position structures that are of course also infinitely large.

Even though these technologies are interesting by themselves, they are not
crucial to understand the techniques presented in this work and I therefore do
not present them in detail.

4.2 Relax CX Analysis

A Relax CX specification defines a pattern, which all valid XML documents
have to adhere to. The form of such patterns, together with all restrictions are
defined in Section 3.1.

The result of parsing a Relax CX specification is a forest of pattern definition
trees, which all have element patterns as root, together with an identifier used to
reference the pattern definition. The start pattern is therefore simply denoted by
the identifier of its definition, rather than by giving the whole pattern definition.

All pattern trees of the forest are known to be reachable from the start pattern
definition, by following pattern definition references. The Ref constructor of the
Pattern datatype also shows a Pattern component besides the identifier of the
reference, but this component is still unused and set to the Empty pattern after
simplifications are done.

4.2.1 References and Cyclic Patterns

The goal of the first step of pattern analysis is to fold the forest of pattern
definitions into a single cyclic pattern. The advantage of this representation of
the complete specification is the ease of use of references and the ability to get
rid of non-cyclic references completely.

Although functional datatypes are always trees, so they can never be cyclic,
the laziness of Haskell allows to produce cyclic values of recursive datatypes.
From a logical point of view, such values are infinitely large and lazily expanded
when they are traversed. In Haskell, however, the kind of definition of a cyclic
value allows the compiler to optimize and share reoccurring values. A reference
which would generate a reoccurring node is then in fact an upward reference in
the tree data structure.

Figure 4.1 shows a small example specification and how each of the four ele-
ment definitions is placed into its own pattern definition tree by the Relax NG
simplification steps. The right side of the figure shows the desired combined
cyclic pattern, which has all non-recursive references inlined and all recursive
ones left intact. The references which remain, however, still contain the complete pattern definition, which is bound to generate an infinitely large pattern, as it contains at least one other reference itself.

To identify the references whose pattern can be inlined completely, one can use a simple graph algorithm. Each element definition is a node, while an edge from node a to node b means a reference to b is contained in the pattern of a. Once the transitive closure is applied to the edge set of the graph, each node which has an edge to itself belongs to a recursive definition. Each node which has no reference to itself can be inlined safely.

The transformation from a forest of definitions to one pattern is done using the recursive let of Haskell.

### 4.2.2 XML Node Classes and Pattern Positions

An inlined pattern of a Relax CX specification can be seen as prototype of a valid XML document. Figure 4.1 gives an indication of the structure of each valid XML document, which can grow in depth arbitrarily, like the pattern itself. In fact each node of a valid XML document belongs to exactly one element or attribute node of a Relax CX pattern. The converse is not a unique mapping, as each node of the pattern defines a class of nodes of a valid XML document.

Given a Relax CX inlined pattern, the element and attribute nodes define XML Node Classes. The other pattern nodes transport vital information about elements and attributes, but do not define classes themselves.
Given an arbitrary XML document, the number of XML nodes possible in each class is statically known to be one of the categories:

- **Exactly one**, for non-optional, non-repeated pattern nodes.
- **Up to one**, for optional, non-repeated pattern nodes.
- **One or more**, for non-optional patterns contained in a non-optional one-or-more pattern.
- **Zero or more**, for optional patterns contained in a any one-or-more pattern or any pattern contained in an optional one-or-more pattern.

A Relax CX specification which contains cyclic references, i.e. which has a corresponding inlined pattern which still contains one or more references, has an infinite number of node classes. This is because XML nodes can be contained arbitrarily deep in the structure, each additional downward step unrolling one cycle of the recursive pattern. In contrast to patterns repeated by recursive references, patterns repeated in a one-or-more pattern do **not** create an infinite number of classes. These nodes are all summarized in one class.

This is because the upper and lower context of nodes is of special importance for XPath axis steps. All nodes in a one-or-more list have the same structural context, whereas nodes in a recursive structure have a different upper context. To be able to differentiate the node classes of a recursive pattern, such that the different upper contexts remain intact, we have to use pattern positions as node classes.

**Pattern Positions**  A pattern is only usable as node class, if the upper context of a pattern, i.e. the whole Relax CX specification, is still available and intact. Additionally, pattern positions allow to navigate arbitrarily between node classes, which is necessary to be able to interpret all XPath axes on them.

The necessary datatypes for pattern positions do not differ notably from the normal datatype. Each constructor has an additional context component, which contains the information about the parent node, as well as in which child position the node is in the parent.

Listing 4.1 shows the core datatypes of pattern positions, which is the type of the upper context of a pattern and the pattern position types themselves. Pattern positions are created from a normal pattern using the recursive let construct of Haskell. Whenever a position is constructed, it contains itself as component, as upper context of all its children.

This construction can again be seen in different ways. One can imagine a pattern position structure as infinitely deep term, which contains all paths in the original pattern which also include upwards steps. Another point of view is that each position contains a reference back to its parent term, i.e. the position structure remains a tree, but there are additional edges added.
4.2 Relax CX Analysis

Another important aspect in the creation of pattern positions is that a special
document node pattern has to be inserted. The position of this pattern will
represent the complete document and it will be possible to reach the top level
element of a document by doing a normal child axis step on the root pattern.
To keep the overhead of pattern position types to a minimum, the document
node pattern is a normal element pattern, with an empty local name and prefix.

| data PatternContext = Context [(PatternPos, Int)] |
| data PatternPos = EmptyPos PatternContext |
| ChoicePos PatternContext PatternPos PatternPos |
| InterleavePos PatternContext PatternPos PatternPos |
| GroupPos PatternContext PatternPos PatternPos |
| OneOrMorePos PatternContext PatternPos |
| DataPos PatternContext Datatype |
| ValuePos PatternContext Datatype String |
| AttributePos PatternContext NameClass PatternPos |
| ElementPos PatternContext NameClass PatternPos |
| RefPos PatternContext PatternPos String PatternPos |

Listing 4.1: Pattern Position Datatypes

There is one important difference between pattern positions and the original
patterns. While both can be considered to be infinite data trees, the pattern
position structure in fact consists of an infinite number of positions. This number
cannot be optimized by the compiler and as I use pattern positions as XML
node classes, there actually have to be infinitely many.

A pattern is a tree with some backward references, which make the pattern
infinite, in a way, but the representation is still a finite tree. A pattern position
structure for such a pattern, however, assigns one position to each possible upper
context in the original pattern, i.e. it unfolds backward references, which are
infinitely many.

So the pattern position structure is indeed an infinite tree plus additional
backward references, which can be optimized. As Haskell is a lazy language and
every XPath path expression can only contain finitely many steps, this does not
represent a fundamental problem for this analysis technique.

Meta Node Classes There is, however, still one problem for static analysis, as
it does not work on concrete XML documents but the class of all valid ones. So
whenever an XPath selects all descendant nodes of an unknown, but valid XML
document, we get an infinite number of possible node classes. As node classes
are represented by pattern positions, this results in the whole infinite pattern position structure to be expanded.

This again shows the major difference between repeated elements in one-or-more patterns and elements in a recursive definition. Still, the decision to give individual nodes in recursive definitions individual node classes, while putting all of a one-or-more pattern into one, remains valid. Especially as I disregard document order in this work, the latter are structurally all the same, while the former differ in their nesting depth, which is significant.

To be able to handle descendant steps with node classes, I introduce another infinite set of node classes, which can be considered to be meta classes. These meta node classes take back the nesting level information of positions in recursive patterns, whenever this information is statically not present and retains the most information possible about the represented XML nodes.

![Figure 4.2: A Congruous Pool of Meta Node Classes](image)

For this reason, the **RefPos** constructor has two child pattern positions. The left one contains the inlined pattern, like it is done in the normal pattern datatype, which contains the infinite number of **precise** node classes as described so far. The pattern position component on the right side, however, contains the same pattern, but each position represents a meta node class.

Figure 4.2 shows a pool of such meta node classes. The pool consists of all descendant nodes of the original reference (here Ref 2), whose selection path contains at most one reference for each id. Each pattern position, whose path would contain a reference twice could be collapsed by cutting out the path part between the two identical references.

Because of this, the children of the lower left Ref 2 position do not belong...
4.2 Relax CX Analysis

to the same pool of meta node classes. Each meta node class represents an
infinite number of precise node classes, which are all present in the same position
structure, following the left pattern component of the Ref 2 position. These node
classes could be reconstructed from the meta node class by following one of the
paths to the meta node class, but taking the left child instead of the right on
the root reference of the pool. A meta node class thereby stands for the paths
that can be constructed by:

1. Taking the path from the root position of the tree to the root reference of
   the pool.

2. Followed by an arbitrary number of downward paths crossing the pool to
   a reference of the same id, which is always at the bottom of the pool.

3. Finally adding the path part form the root reference of the pool to the
   meta node class.

In the example of Figure 4.2 only one path can be arbitrarily repeated in step
2, which is the path part starting at the upper Ref 2 positions, crossing the
Element c and Choice nodes, to the lower Ref 2. In mutually recursive patterns,
such repeatable pool paths can contain references with other ids and there are,
in general, several possible ways to cross a meta node pool.

In fact the whole concept of a meta node class pool is to collapse an arbitrary
number of reference-to-reference paths in the infinite set of precise node classes.
Through this collapse of paths, the root pattern position of a meta node class
pool gains additional parent pattern positions. The context of this root position
contains the obvious Ref 2 position outside of the pool, through which the pool
was accessed, but also all Ref 2 positions at the bottom of the pool.

In the example this is only one additional parent position in the context, but
in general this can be arbitrary, though finitely many positions. By the nature
of pattern position structures, there is always at least one additional parent
position for the root position of a pool.

Pattern Position Contexts To be able to express multiple upper contexts, the
Context constructor takes a list of pattern positions with associated position
indices. The use of a list also has the advantage that the root pattern position
of the whole structure has no special context and it can simply be left as an
empty list.

A downside of introducing meta node classes is that node classes are no longer
disjoint to each other. So whenever a static analysis algorithm has a reference to
a concrete node, which is known to belong to a given node class, it is not sufficient
to compare this node class with another one for equality only. Fortunately, all
pairs of node classes are either disjoint or one contains the other.
4 Static Analysis

The pattern positions which are descendants of the bottom positions of a meta node class pool describe subsets of the meta classes from within the pool. The reason for this is that they have the additional constraint of one fixed crossing through the pool at the end of the repetition step 2. Likewise it is again significant, if the pool is left using the left or the right edge of a Ref position. In the former case, only one step is added, whereas in the second case another pool is entered and there are again multiple parent position to step back to.

**Pattern Position Functions** For further analysis of pattern positions and especially those positions which also represent node classes, there are a lot of functions which need to be defined. Some of these functions will be relevant in the coming sections, so I will list their signatures here.

<table>
<thead>
<tr>
<th>Description</th>
<th>Signature</th>
</tr>
</thead>
<tbody>
<tr>
<td>Selects the context (list of parent/position pairs)</td>
<td><code>context :: PatternPos -&gt; PatternContext</code></td>
</tr>
<tr>
<td>Selects the root position</td>
<td><code>root :: PatternPos -&gt; PatternPos</code></td>
</tr>
</tbody>
</table>
| Selects the major parent pattern position or position index | `parent :: PatternPos -> Maybe PatternPos`
| so this is always an upward step in the pattern position tree | `position :: PatternPos -> Maybe Int` |
| Selects all parent pattern positions from the context | `parents :: PatternPos -> [PatternPos]` |
| Selects all pattern position components of a position constructor | `children :: PatternPos -> [PatternPos]` |
| Test functions for each constructor (omitting the suffix ‘Pos’) | `is<Name> :: PatternPos -> Bool` |
| Test function for references having a fixed id | `isRefR :: String -> PatternPos -> Bool` |
| Is true for positions which are the left/right child of a Ref | `isSingle :: PatternPos -> Bool`
| Is find | `isFind :: PatternPos -> Bool` |
| Combinations of test functions for nodes (element, attribute, data and value), info positions (choice, interleave, group) and positions for internal use (empty, ref) | `isNode :: PatternPos -> Bool`
| `isInfo :: PatternPos -> Bool`
| `isIntern :: PatternPos -> Bool` |

|Listing 4.2: Functions on Pattern Positions|
4.2.3 Axis Steps on Node Classes and Set Evolution

Node classes are used in static analysis to describe the abstract results of XPath path expressions. In this role, it is vital that XPath axis steps can be applied to node classes to get a new abstraction of the resulting nodes, which are again node classes.

As document order is disregarded in this work and XPath CX does not support such language constructs as well, not all forward and reverse axis are supported. Still most of the axis steps result in a set of node classes when applied to a single node class. The result is that most XPath path expressions will describe sets of nodes, at least most path expressions will do so in the intermediate steps.

Strong filters and node tests have to reduce the possible number of nodes, such that an XPath path can be known to refer to only one node or at least a single class of nodes. Filters and node tests therefore only reduce the current set of node classes, while axis steps often dramatically increase the number of classes.

To be able to define axis steps both conveniently and efficiently, without generating the same node classes over and over, I introduce the notion of set evolution.

Set Evolution  A set evolution is a repeated calculation on a set of items. In each step of the calculation, a function that creates a result set for any item can be applied to the whole set and the results are combined into a new current working set. Duplicates are automatically removed from the working set.

Sequences of one-to-many functions can therefore be conveniently applied in parallel to a number of items, without having to manage the result sets and evaluating the same intermediate result sets for different elements over and over. The initial set is efficiently evolving towards a desired result set.

Furthermore it is possible to mark items for future exclusion, so they are never selected or evolved again, which reduces redundant computations. Items can be transferred from the working set, which is evolved with each step, to a result set, which is not evolved. Items from the result set, which are again created in the working set are discarded, which again reduces redundant computations.

Set evolutions are expressed as values of a specialized state monad in Haskell. Only few primitive monadic functions are necessary to be defined, to specify arbitrary complex set evolutions. The cornerstone of this framework is the evolve function, which takes an initial set, as well as a set evolution, and returns the result set:

\[
\text{evolve} :: \text{Eq e} => [e] \to \text{SetEvolution e a} \to [e]
\]

As seen in the signature of evolve, the idea of evolving sets can be used on any datatype which supports equality, not only pattern positions. Listing 4.3 gives an overview of the signatures of all relevant definitions, which will be
introduced in this section and also used on several other occasions.

The state used in set evolutions consists of three sets, the set of marked items, the set of result items and the current working set. To be able to define basic set evolutions, only two commands are necessary. One that applies a function (Evolver) to the working set, which is called step, and one function to take items out of the working set to populate the result set, by selecting elements from the working set using a predicate (Pred), which is called lookFor.

| type SEState a = ([a], [a], [a]) |
| type SetEvolution e a = State (SEState e) a |
| type Evolver a = a ⇒ [a] |
| type Pred a = a ⇒ Bool |
| step :: Eq e ⇒ Evolver e ⇒ SetEvolution e () |
| acc :: Eq e ⇒ Evolver e ⇒ SetEvolution e () |
| lookFor :: Eq e ⇒ Pred e ⇒ SetEvolution e () |
| skip :: Eq e ⇒ Pred e ⇒ SetEvolution e () |
| fix :: Eq e ⇒ SetEvolution e a ⇒ SetEvolution e () |
| evolve :: Eq e ⇒ [e] ⇒ SetEvolution e a ⇒ [e] |

Listing 4.3: The Set Evolution Framework

Unfortunately such evolutions using only the two commands have always a fixed number of steps and there is no guarantee that the current working set is empty in the end or that some result has been found. The fix command takes a set evolver and iterates it as long as the result set is not empty, which allows to very easily search through domains of items and select the correct ones.

**Parent, Child and Attribute Axes** With this additional command we are able to define the parent axis of XPath on node classes.

parentAxis :: Evolver PatternPos
parentAxis pat = evolve [pat] $ fix $ do { step parents ; lookFor isNode }

Listing 4.4: The Parent Axis on Node Classes

Another basic command is used to discard items from the current working set, such that they are no longer evolved. The skip command takes a predicate to find items to skip and is therefore in a sense complementary to the lookFor command. With this command we can define the child and attribute axes.

Both axis steps result in precise node classes, so the right child of Ref positions has to be skipped, using the isFind pattern position test. The evolution of
4.2 Relax CX Analysis

positions additionally has to stop at every other node that is encountered during the children steps, only internal and information nodes should be walked over.

```plaintext
childAxis :: Evolver PatternPos
childAxis pat = evolve [pat] $ fix $
  do { step children
       ; skip isFind
       ; lookFor isElement
       ; skip isNode
     }

attAxis :: Evolver PatternPos
attAxis pat = evolve [pat] $ fix $
  do { step children
       ; skip isFind
       ; lookFor isAttribute
       ; skip isNode
     }
```

Listing 4.5: Child and Attribute Axes on Node Classes

**Ancestor Axis**  As the parent axis, like all other axis steps, is an evolver itself, the transitive ancestor axis can be defined easily using the accumulate (acc) command. This command applies an Evolver to the working set and automatically adds the results to the result set, without taking them out of the working set.

```plaintext
ancestorAxis :: Evolver PatternPos
ancestorAxis pat = evolve [pat] $ fix $ acc parentAxis
```

Listing 4.6: Ancestor Axis on Node Classes

The reader should note at this point, that the definition of the ancestor axis might seem trivial, but the actual function defined is far from it. Like all axis steps defined in this section, the ancestor axis step can of course be applied to meta node classes as well. So the ancestor axis does not only accumulate all element position nodes on the way to the root pattern, but also selects arbitrary many sibling or even descendant element positions of itself.

This is due to the fact, that some parent position of a meta node class has multiple contexts, which represent the collapsed loops through the pool. So stepping back up outside of the pool can result either in the parent position of the tree structure or in any of the Ref positions with the same identifier at the bottom of the pool.

This is where the special optimization features of set evolutions come into play. When going upwards from all Ref positions found in the multiple contexts, we are bound to encounter many similar parent nodes, which should not be all evaluated over and over again and more important we will come to the root
position of the pool again. At this point we mustn’t follow the parent axis again or the evolution would continue indefinitely.

Figure 4.3 shows an example position structure, in which the ancestors of A should be calculated. Element position B is obviously an ancestor of A, as are all the node classes on the path from the upper Ref 2 to the root of the pattern position structure. But the context of B also includes the three lower Ref 2 positions, as there exist paths through the pool to them. All meta node classes on these paths are also ancestors of A, which are B, C, D and also A itself!

Pattern position E does not describe an ancestor node class of A. It does contain an infinite pattern as well, but as it contains only Ref positions with other ids than 2, it cannot be an ancestor of A. There is no path up from A, to any bottom Ref 2, which then leads to E.

As references always have element positions as children and all element positions are node classes which are added to the result set, we rest assured that no node class is evolved twice and that the multiple contexts of the upper Ref 2 position are not followed indefinitely.

**Descendant Axis** In contrast to the ancestor axis, it is impossible for the descendant axis to just accumulate the child axis, as there are infinitely many pattern positions and therefore node classes involved. The general problem was already discussed earlier and solved by introducing meta node classes, so this has to be considered when implementing the descendant axis.

First of all we can not utilize the `childAxis` function, but have to define a second child axis, which skips the `single` steps and includes the `find` ones instead, i.e. which swaps following the left child of Ref nodes by following the
right one. Additionally, the special `findChildAxis` function collects not only element positions, but also reference positions. These are not node classes, but are needed to formulate the stopping criterion for the descendant axis.

The descendant axis can then be formulated as the accumulation of the `findChildAxis`, stopping at the second unfolding of a reference in the same pool. As we included the reference positions in the child axis, these positions have to be filtered out of the result set afterwards.

```plaintext
findChildAxis :: Evolver PatternPos
findChildAxis pat = evolve [pat] $ fix $
  do { step children
        ; skip isSingle
        ; lookFor isElement
        ; lookFor isRef
        ; skip isNode
  }

unfolded :: PatternPos -> Pred PatternPos
unfolded till pat = case pat of
  (RefPos ref) -> not $ null $ evolve [pat] $ fix $
    do { step (maybeToList . parent)
        ; lookFor (isRefR ref)
        ; skip (== till)
    }
  otherwise -> False

descAxis :: Evolver PatternPos
descAxis pat = filter isElement $ evolve [pat] $ fix $
  do { acc findChildAxis
        ; skip (unfolded pat)
  }
```

Listing 4.7: Descendant Axis on Node Classes

The `unfolded` predicate scans upwards in a pattern position tree, searching for an occurrence of a reference of the same id as the current reference, before crossing the original pattern position. Note that while scanning upwards only the major parents are selected, i.e. only real upward references are followed.

This stopping criterion, to calculate the pool of meta node classes, could also be formulated in another way. Whenever we step over a `Ref` position using the right `find` edge, the list of all bottom references of the same kind for this pool is stored in the context. This context information can be used to test references encountered while accumulating against and then skip them, so scanning upwards is unnecessary.

**Self Axes** The self axis is just the identity function on node classes and does not need any special attendance. The combined axes descendant-or-self and ancestor-or-self differ from the shown functions by adding the starting node.
class to the result as well, if it is not already contained.

4.2.4 Document References

The static analysis of XPath path expressions can be done using node classes only, but the result is an over approximation of the selected XML nodes. This might be enough in many cases, but is especially unsuited to deal with paths that are supposed to select exactly one node of a document. The reason is that most node classes for common Relax CX patterns describe an unbounded number of XML nodes.

Node classes which either contain one node or none, on the other hand, are not much of a problem to static analysis, as only the existence of a node has to be asserted. The difference between one node and an arbitrary number of nodes is that the former can be used as a value, whereas the latter has to be treated as a sequence. XPath allows to blend these two things together to a certain degree, but this is not an advantage for static analysis.

The two major sources of ambiguous node classes are one-or-more patterns and recursive patterns, as described in Section 3.1.4. Relax CX solves this problem by allowing the definition of primary key sets for repeated nodes, which have to consist of atomically typed, non-optional and of course non-repeated child nodes. RCX’ simplifies this further by saying that such primary key sets always consist of exactly one id attribute.

Using such primary key sets in filters allows to filter down infinite node classes to up-to-one XML node sets. A pair of a node class, represented by a pattern position, and a list of primary key filters is called a filtered class. Such a filtered class can represent a set of concrete nodes or up to one concrete node. A document reference is a static reference into a document, defined by a set of filtered classes.

It is thereby possible to specify both unique concrete nodes as well as classes of nodes using document references. The important restriction is that it is impossible to specify arbitrary subsets of node classes, each class is either selected completely or exactly one node from it is selected. Combinations of these selections are possible, but intersections or subtractions are prohibited.

Ambiguous Node Classes To precisely define document references, we need some more definitions. The first step is to define the source of ambiguities, so it becomes clear at which steps of a selection path filters have to be applied to get to a unique node.

The RCX’ subset of Relax CX disregards recursive pattern definitions completely, to reduce the amount of ambiguities for this work upfront. What remains is the one-or-more pattern, which already causes enormous troubles to static analysis.
A one-or-more pattern essentially states a 1:n relationship between the parent element and its children. So if the parent is not ambiguous by itself, the ambiguity is introduced by the child node class. All of their children node classes are also ambiguous, but they are again in 1:1 relation to their parent.

So if an ambiguous node class should be filtered down to a precise document reference, all elements directly contained in a one-or-more pattern have to be filtered. Figure 4.4 shows an example pattern with several sources of ambiguity, which are also contained in each other.

![Figure 4.4: Ambiguous Node Classes and Degrees of Ambiguity](image)

The point to note is that filtering an ambiguity of a node class is often not enough to reach a unique node, but all ambiguities of the node class and its
parents have to be resolved. Also, only ambiguous node classes which have a non-optional \texttt{id} attribute can be filtered at all, so if an ambiguous node class is missing this attribute, all ambiguities in contained descendant elements cannot be resolved as well.

Using set evolution it is easily defined if a node class is \textit{ambiguous} and if a unique node in this class is \textit{selectable} or not.

\begin{verbatim}
ambiguous :: PatternPos -> Bool
ambiguous p | isElement p =
  not $ null $ evolve [p] $ fix $ do { step (maybeToList . parent)
  ; lookFor isOneOrMore
  ; skip isElement
  }
ambiguous p | isAttribute p = False

ambiguities :: PatternPos -> [PatternPos]
ambiguities p = filter ambiguous (ancestorOSAxis p)

isID :: PatternPos -> Bool
isID (AttributePos . (Name (QName "" "id"")) _) = True
isID otherwise = False

idAtt :: PatternPos -> Maybe PatternPos
idAtt p | ambiguous p = listToMaybe $ filter notOptional $ filter isID $ attAxis p
  where

  notOptional :: PatternPos -> Bool
  notOptional a =
    not $ null $ evolve [a] $ fix $ do { step (maybeToList . parent)
    ; skip isChoice
    ; lookFor (== p)
  }

selectable :: PatternPos -> Bool
selectable = isJust . idAtt
\end{verbatim}

Listing 4.8: Ambiguities and Selectability of Node Classes

The function \texttt{ambiguous} tests if there is a one-or-more ancestor pattern before hitting the parent node class, which can only be an element pattern. Using that function it is easy to define the list of \texttt{ambiguities} for a node class, by just filtering the ancestor-or-self axis.

Using a test function for pattern positions, which identifies \texttt{id} attributes, we can define a function which returns the non-optional \texttt{id} attribute of an
ambiguous node, if it exists, and nothing otherwise. An ambiguous node can then be uniquely filtered if it has such an attribute. Note that being selectable only means that the last introduced ambiguity can be resolved, but if there are more unsolved ambiguities in ancestor node classes, a node class can still be ambiguous.

**Document References** XPath CX allows to filter ids only using fixed unknown values, which are the parameters in AL. An ambiguous node class can now be narrowed down step by step by adding such filter values for each ambiguity from the root of the pattern position structure down to the pattern of the node class itself. A filtered class is therefore simply a pair of a pattern position and a list of unknown values. For static analysis of context conditions, however, it will be necessary to specify that an id attribute is set to *any existing value*, too.

| data UFE = -- (U)nique (F)ilter (E)xpression |
| UnknownValue String --- contains a label for the value         |
| Existing 

| type FilteredClass = (PatternPos, [UFE]) |

| data DocRef = DocRef [FilteredClass] |

Listing 4.9: Filtered Classes and Document References

**Unique and Narrowed References** Document references will be the result of abstract path interpretation and the major items which are compared and analyzed. To be able to do so, a number of functions are available on document references.

| exclusive :: PatternPos -> PatternPos -> Bool |
| exclusive a b = isChoice $ relation a b |

where

| relation :: PatternPos -> PatternPos -> PatternPos |
| relation a b | (isAttribute a || isElement a) && (isAttribute b || isElement b) = |
| select (root a) $ map (flip get . fst) $ |
| takeWhile (uncurry (==)) $ zip (path a) (path b) |

| unique :: DocRef -> Bool |
| unique (DocRef l) = |
| and (map (uncurry exclusive) $ pairwise (map fst l)) && |
| and (map (\(pp, fi) -> length (ambiguities pp) == length fi) l) |

where

| pairwise :: [a] -> [(a, a)] |
4 Static Analysis

pairwise [] = []
pairwise (h:t) = [(h, o) | o <- t]++(pairwise t)

sunique :: DocRef -> Bool
sunique (DocRef [(pp, fi)]) = length (ambiguities pp) == length fi
sunique otherwise = False

narrowed :: DocRef -> Bool
narrowed dr@(DocRef [(pp, fi)]) =
  if sunique dr then True else
  let next = head $ drop (length fi) $ ambiguities pp in
  not (selectable next)
narrowed otherwise = False

Listing 4.10: Unique and Narrowed Document References

To define uniqueness on document references, it is necessary to define when two node classes are mutually exclusive. This is true whenever the last common parent position of two node classes is a choice pattern, so both node classes are defined in mutually exclusive branches of a choice.

A document reference is considered to be unique if it references either a single or no node of the document. This is true whenever all filtered classes are pairwise mutually exclusive and each filtered class has all ambiguities solved, i.e. there is a uniqueness filter for each ambiguous node.

To facilitate static analysis, there is one more category of uniqueness, which is the so called static uniqueness. To be statically unique, a node class needs to additionally consist of one filtered class only. The problem with unique document references with multiple exclusive filters is that it is unknown which of the options really exists.

This makes comparisons with other document references impossible in some cases and additional dynamic checks have to be introduced. To avoid this for now, the contexts of appends and updates, for example, need to be statically unique, not only dynamically unique.

The last function called narrowed tests if a document reference consists of one filtered class only, which has as many filters as possible. This function is used whenever statical uniqueness is impossible, as some ambiguities cannot be resolved. The idea is that a narrowed document reference is statically as precise as possible and there are no more precise references to XML nodes of this class.

Types of Document References  Document references are extensively used in XPath CX evaluation and they have to blend into its type system as well. It is therefore necessary to define the type of a document reference, which can easily be done as they are based on Relax CX patterns which have the same atomic types as XPath CX.

To define the type of a reference, we first need the kind of a node class, which only gets complicated for enumerations. They are defined as choices of values of
4.2 Relax CX Analysis

the same datatype, so it is necessary to descend a bit in the pattern to retrieve them. All other element and attributes are either empty, have a data pattern or are internal nodes. Empty attributes and elements are considered internal nodes, as they cannot be used as atomic values and are therefore considered to have the complex kind.

```
ppKind :: PatternPos -> Kind
ppKind p | isAttribute p || isElement p =
  let dt = nub $ map datatype $ evolve [p] $ fix $
    do  { step children
         ; lookFor isData
         ; lookFor isValue
         ; skip (not . isChoice)
    }
  in case dt of
    [DTString]    -> TString
    [DTToken]     -> TToken
    [DTInteger]   -> TInt
    [DTBoolean]   -> TBool
    []            -> TComplex

typed :: DocRef -> Type
typed dr@(DocRef l) =
  let kinds = nub $ map (ppKind . fst) l in
  Type False (unique dr && notOptional dr) $ case kinds of
    [k]    -> k
    [TString, TToken]    -> TString
    otherwise            -> TComplex
```

Listing 4.11: Typing on Document References

The document reference then collects all kinds of its filtered classes and either has the same type or is considered complex if there are two or more different types involved. Only tokens and strings can be unified to string.

A document reference is obviously not an atomic value for XPath CX, so this flag is set to false. The simple flag is basically equivalent to the uniqueness property of the document reference, but uniqueness does not guarantee that there is a node at all. Even though uniqueness guarantees that a node class has filtered all ambiguities, it can still be an optional node class, i.e. contain a choice pattern which is not associated with an ambiguity.

A document reference would basically be known to reference at least one node, if it contained a descendant node class from each child of such choices. Uniqueness contains a similar check, to guarantee that there is at most one node in the class. Unfortunately, such document references would make static analysis very complicated, as operations which remove one option of a choice and then later add the other are difficult to handle.

XPath CX therefore does not consider document references that contain unique but optional nodes as simple and treats them as sequences. So to use such values
one has to use an aggregate function like \texttt{sum}, which actually makes sense for
document references which do not include a member for each option of a choice,
as such a reference can be empty. The \texttt{sum} function handles such cases explicitly
by returning the value zero.

\textbf{Creation and Evolution} Document References are created from pattern posi-
tions and always start at the unique root reference, which is obviously unfiltered.
They can then be evolved using axis steps together with node tests and also spe-
cial filters.

How axis steps are applied to node classes is already covered and document
references simply apply the steps to all their filtered classes. Node tests are
trivial to check on node classes, as all supported node tests directly translate to
simple constructor checks of the pattern position datatype.

The only interesting thing considering axis steps is that starting from a filtered
class with one pattern position, the result of the step can be arbitrary many
positions or even none at all. So after applying step and filter to a filtered class,
we can end up with several new filtered classes, which initially all share the
filters.

Filtered classes which are the result of a reverse step may of course loose
some of the filters, whenever they stepped out of an ambiguous node, back to
the unique parent. Duplicate filtered classes are also removed so there are only
filtered classes with the same node class, but different filter to worry about.

If the filters are incomparable, the set of dynamic nodes described are also
different, with an empty intersection. If they only differ in length, however, and
one is a prefix of the other, one of the classes can be removed, as it is contained
in the other.

A document reference can only be filtered if all filtered classes are ambiguous,
selectable and all parent ambiguities are already filtered. The filter expression
needs to be an unknown value of the same type as the \texttt{id} attributes, which
obviously have to share the same type as well. If all prerequisites are met, the
filter value is inserted into the filter list of all filtered classes.

\textbf{Miscellaneous} Additional functions on document references, which will not be
explained in detail here, are, for example, the union operation, the root function
on document references and a special function used for context rules, which fills
up all filters of a document with the \texttt{Existing} constructor, so any of the existing
nodes in a document is selected for a node class. Whenever these functions are
relevant for other sections their mechanics are explained there.
4.3 XPath CX Analysis

The analysis of XPath CX expressions is connected to the conversion from XPath 2.0, as the restrictions defined for XPath CX cannot be enforced on the structural level only. One major reason for this is that XPath CX is statically typed, which enforces abstract interpretation of path expressions, for example, so the type of such expressions can be retrieved from the corresponding Relax CX pattern.

To get to an XPath CX expression from an XPath 2.0 expression, a mix of filters, conversions and abstract interpretation is applied. Most notably, all path expressions are interpreted and removed from expressions and replaced by document references.

The necessary abstract datatypes, for the result of the analysis step, have already been introduced in Section 3.2.4. One cornerstone for type analysis is the definition of a type, which includes the kind as well as two flags, one denoting an atomic opposed to a node value, the other denoting simple values opposed to sequences.

The conversion of an XPath 2.0 expression results in an XPath CX expression together with a type, which is a TypedExpr. Whenever subexpressions are converted, the parent expression will do a static type check and let the conversion fail if the check fails.

```haskell
data Env = Env
  { ci :: DocRef -- context item
  , vs :: Map QName TypedExpr -- variables
  }

expr :: Monad m => Env -> X.Expr -> m TypedExpr
expr env e = case e of
  X.NumericLiteral (X.Integer val) -> return $ TE (Literal val) (Type True False TInt)
  X.StringLiteral val -> return $ TE (Literal val) (Type True False TString)
  X.VarRef name ->
    guard (member name $ vs env) >> return (vs env ! name)
  X.If tests e1 e2 ->
    do { guard (length tests == 1)
      ; teTest@ (TE _ tyTest) <- expr env (head tests)
      ; teThen@(TE _ tyThen) <- expr env e1
      ; teElse@(TE _ tyElse) <- expr env e2
      ; guard (simple tyTest && kind tyTest == TBool)
      ; guard (simple tyThen == simple tyElse)
      ; guard (kind tyThen == kind tyElse)
      ; guard (kind tyThen /= TComplex)
      ; return $ TE (If teTest teThen teElse)
        tyThen { atomic = True }
    }
```

Listing 4.12: Fragment of XPath 2.0 to XCX Conversion
Listing 4.12 shows the first few cases of the `expr` function, which converts an XPath 2.0 expression, given in abstract syntax prefixed with an X, into a typed XPath CX expression. The function additionally takes an environment for abstract interpretation, which holds the current context item as well as all typed expressions which were bound to a variable.

In general, the context item would also be a typed expression. In XPath CX, however, atomic value expressions and path expressions mustn’t be mixed and there are severe limitations on constructing and using sequences. It is therefore not possible to bind any value except a path to the context item, which is therefore defined as document reference, as it would end up being a `Node` anyway.

The first two cases handle the supported literals, which are integers and strings only. The resulting expressions themselves are equal and the type information is added externally to the expression. Both, of course, result in simple atomic types of their kind. Variables have to be defined and are then taken from the environment.

The conversion of the `if` expression is one of the bigger and more interesting cases and I will explain it as an example of many similar cases, which are left out here. XPath 2.0 allows sequences as test expressions, which is not supported by XPath CX, so the first test is to assert that only one expression was given. This is but one example of the many kinds of small restrictions XPath CX has, that limit the often unnecessary complexity of XPath 2.0 to suite static analysis.

The next three lines convert the test expression, as well as the expressions given in the `then` and the `else` case of of the `if`. For the test to be type correct, it has to be a simple boolean value, no matter if it is atomic or not. The types of the other two expressions do not matter, but they have to be equal both in kind and in the simple flag.

I don’t want the result of `if` expressions to be used as nodes, as this would open up all kind of conditional expressions in XPath CX, which are really not essential at first. It would, for example, be possible to continue navigation on those nodes, which doesn’t fit with the idea of abstract interpretation of paths.

The `if` would become a path operator and it is unclear how this could be handled. It might be possible to just apply navigation steps from outside the `if` to both included expressions, but then this would be a path transformation, which is a concept I am not including in this work.

So by disallowing the use of `if` expressions as nodes, it doesn’t make any sense to allow the complex kind as a result, which is therefore forbidden. The resulting expression is transformed into XPath CX abstract syntax and the type of it is the type of one of the embedded expressions, but it is additionally considered to be atomic, so it cannot be used as a node.
4.3 XPath CX Analysis

4.3.1 Supported Features and Restrictions

As described in Section 3, XPath CX is a moving target and a most powerful and convenient, yet controllable subset of XPath 2.0 has yet to be found. XPath CX is a very restrictive first prototype, which yields first results for static analysis. In this and the following two sections, I summarize all supported XPath 2.0 features, together with the restrictions that apply to them.

The abstract syntax already gives a glimpse of what is allowed, but it does not give away all details which are involved. Like with the if expression described above, some constellations of types or features can cause troubles and have to be prohibited.

Figure 4.5 shows all supported features of XPath CX, except path expressions, casts and treats. Everything not shown in the table, like quantifiers, for or the sequence operator, are unsupported in this prototype.

4.3.2 Treats and Casts

The treat expression takes a sequence type as parameter and statically guarantees the contained expression to be of that type. As XPath 2.0 can’t guarantee this, it has to check it at runtime, which results in an exception if the check fails.

I have to support the treat expression, as the root operator is normalized into the root function, whose return type cannot express that the resulting node is a document node, which is then asserted with a treat. Treat expressions in general do not hurt XPath CX and they can be checked and discarded at conversion time. For this reason I support treat expressions for all atomic types, both simple and sequences, as well as all node tests which are supported for axis steps, which include the document node test. I Cast expression are in general unsupported, but it might be necessary to insert them in XPath 2.0 expressions to enforce correct evaluation. The values gathered from path expressions are statically known to be of a certain kind, as there is an associated Relax CX pattern which defines it. XPath 2.0 does not have this luxury and all attribute and element values are plain text.

If such values are used in an integer operator, XPath casts the text automatically, which we know will always be successful. If two nodes are compared, however, this can lead to problems, as comparing the textual representation of integer values does not coincide with comparing their values. So to be on the save side, the user can insert cast expressions, but it might as well make sense to insert such casts at any place a path expression is used as an atomic value.

Casts take so called single types as parameter, not sequence types, as only simple values can be casted. For XPath CX this simply means one of the four supported atomic types.
### 4 Static Analysis

<table>
<thead>
<tr>
<th>Feature</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Literals</strong></td>
<td>Only string and integer literals are supported. Tokens have to be casted from string and the boolean literals are function calls.</td>
</tr>
<tr>
<td><strong>If</strong></td>
<td>The if has to contain exactly one simple boolean test. The expressions have to match in kind and the simple flag and also mustn't be complex. The result is always atomic.</td>
</tr>
<tr>
<td><strong>Arithmetic Operators</strong></td>
<td>The operators <code>plus</code>, <code>minus</code>, <code>mult</code>, <code>div</code> and <code>mod</code> are defined for simple integer values and result in an <strong>atomic</strong> simple integer.</td>
</tr>
<tr>
<td><strong>Boolean Operators</strong></td>
<td>The operators <code>and</code> and <code>or</code> are defined on simple boolean expressions and result in an <strong>atomic</strong> simple boolean value.</td>
</tr>
<tr>
<td><strong>Union</strong></td>
<td>The union is defined on node sequences of any kind only. The corresponding document references are combined and the result type is the type of the resulting document reference.</td>
</tr>
<tr>
<td><strong>Comparators</strong></td>
<td>Comparisons are allowed on all atomic datatypes. The kind of both expressions has to match and they both need to be simple. The result is an <strong>atomic</strong> simple boolean value. General and value comparators do not differ and node comparators are unsupported.</td>
</tr>
<tr>
<td><strong>Unary Minus</strong></td>
<td>Both unary operators are supported on simple integer values and return an <strong>atomic</strong> simple integer.</td>
</tr>
<tr>
<td><strong>Unary Plus</strong></td>
<td>The context item always contains a document reference, which is returned as <code>Node</code>, using the type of the reference.</td>
</tr>
<tr>
<td><strong>True, False</strong></td>
<td>The functions <code>true</code> and <code>false</code> are the boolean literals. <code>Not</code> takes a simple boolean value and returns an <strong>atomic</strong> simple boolean.</td>
</tr>
<tr>
<td><strong>Count</strong></td>
<td>The function <code>sum</code> is defined on sequences of integers, which can only be created as node sequence. The constructor <code>sum</code> does not contain a typed expression, but the resulting document reference.</td>
</tr>
<tr>
<td><strong>Sum</strong></td>
<td>The function <code>count</code> is defined on sequences of any kind, which also can only be created as node sequence. Count therefore contains a document reference and no typed expression, too.</td>
</tr>
</tbody>
</table>

Figure 4.5: XPath CX Supported Feature Table
4.3.3 Path Expressions

The path expressions are one of the most important features of XPath 2.0 and also of XPath CX. They are the main sources of dynamic values in expressions and connect XPath to XML and by this also to Relax CX. The abstract interpretation of path expressions on Relax CX patterns is a cornerstone of this work and allows to transport information from structural schemas to both context rules and algorithms.

Just like the real XPath interpretation, abstract interpretation needs to maintain an environment of needed information. A lot of information contained in the static and dynamic context of XPath 2.0 expressions is either unnecessary to maintain for this analysis, belongs to unsupported features or is hard coded into the language. The list of function signatures, for example, is not explicitly maintained here, as all supported functions are handled individually.

So the environment used here, which is also described in the context of Listing 4.12, consists only of two dynamic context members, which are the context item and the list of variables. The context item is always a `Node`, which is saved as document reference for convenience reasons, whereas the variable map contains any typed expressions.

**The Path Operator** The single slash is the XPath path operator and the only binary path operator, which means it is the main combinator for path fragments as well. The left expression contained in a path operator is evaluated and assigned to the context item. The result of the path step is the right expression, which is evaluated in the same context as the left, except for the context item, which was changed.

The conversion of a path step therefore interprets the left expression and assumes that the resulting typed expression is a node. As motivated in Section 3.2.3, path expression should never be mixed with non-path content, so XPath CX forbids anything but nodes as the result of the left path expression.

The associated document reference of the resulting `Node` is placed in the environment as context item. Using this updated environment, the right expression is interpreted and returned as the result of the path step. It is a specification error, if this expression is not a node as well, as an atomic result would again mean that a path was mixed with another non-path expression.

**Axis Steps and Node Tests** Axis steps do not contain another expression, they are in fact leaf types of the abstract syntax. To combine axis steps or filters, the path operator has to be used. An axis step is always applied to the current context item, which is known to be a document reference. XPath CX supports all axis which are supported by Relax CX analysis, which are most axis that are independent of document order.
If the specified axis is supported, the corresponding function is applied to the
document reference, which either results in another document reference or in an
error, if the document reference would be empty. An empty document reference
means that there can never be any nodes selected, which is a specification error
by the user.

XPath 2.0 supports a lot of different node tests, of which many can be dis-
carded right away because of the XML model I use. The supported node tests
of XPath CX are the following:

- The name test, which selects all nodes, no matter if element or attribute,
  which match the specified name. This can easily be checked on node
classes, as Relax CX supports only singleton name classes.

- The any node test, which is true for all nodes and essentially means no
  node test is applied after the axis step.

- The document node test, which is true only for the special document
  node. This can also be easily checked on node classes, as the pattern
  position needs to be the root of the structure for the test to be true.

- The attribute and the element test, which are true for all attribute or
  element nodes, respectively. Again this is trivial to check on node classes.

The node test supplied with the axis is applied as filter to the document
reference, which looses all node classes that do not match the filter. If no class
is left, this is also a specification error, as the path in total can never select
anything. As many path expressions contain axis steps with concrete name
tests, this check is very valuable to the user and would be impossible to do
without the combination of XPath and Relax.

Filter  The last element of path construction is the filter. In XPath 2.0 it takes
a primary expression, which has to evaluate to a sequence, and a list of filter
expressions, which have to be true for an element to remain in the sequence.
XPath CX does neither support liberal sequences nor filters and has in general
to be very cautious with filters.

So the primary expression has to result in a sequence of complex nodes. The
 corresponding document reference has to contain one or more node classes which
are all ambiguous, selectable and whose filters leave exactly the current ambi-
guity open. XPath CX therefore does not support filtering of ambiguities at a
later time, though this might be included in a future prototype. As it is now, a
filter has to be applied at the exact node the filtered id attribute is situated in.

The filter expression list has to consist of exactly one item only, which is a
comparison using the general equality operator. The expression on the left side
of the operator has to result in a Node and the corresponding document reference
has to select the id attributes of the ambiguous node classes being filtered. The expression is interpreted using the result of the primary expression as context item, so the left side of the filter expression can be a single relative attribute axis step with name test.

Note that in XPath 2.0, the context item within the filter expressions would be bound to exactly one node of the primary expression. By using the document reference of the primary expression without filtering it down using Existing or another special filter value, the result of the id selection is in fact the whole class of id nodes.

Because of the severe restrictions of this expression, that fact isn’t really relevant for static analysis, but shows how complex offering more complex filters will be. Filters, comparable to the contexts of rules, work on single elements of sequences, which are only known at runtime. If one item of the sequence is changed, the effect of this change is difficult to compensate in a path expression.

In context rules, each node has its own instance of the rule and whenever a node is changed, this instance is separated from the rest, which is possible as they are independent. The remaining instances can be checked together by simply removing the changed node. This is in general impossible for filters, as the items of the resulting sequence of a filter need not be independent.

The expression on the right side of the equality needs to result in an Unknown value, which is per definition an atomic, non-complex simple value. So the only supported filters in XPath CX have the form [@id = $value], where the id selection can be formulated in other equivalent ways and the value can be something else than a variable, if the result is also Unknown. At the moment only AL parameters are available unknown values, so the normal case is indeed a variable selection.

If all conditions are met, the filter value is applied to all filtered classes of the current document reference, making each filtered class unique. The result of the filter is a Node containing the filtered document reference.

4.4 CRL Analysis

In this section I will make use of the benefits of XPath CX in CRL and transform the rules defined in specifications into a much simpler form, which facilitates static analysis even further. To achieve this, I am splitting a rule up into several independent rules, which have all variables inlined and basically consist of a name and a typed XPath CX expression to be checked only.

The first step is to split up the list of checks in a rule and create a separate rule for each check. The user is able to define as many checks as he wants in a single rule, but this is only for convenience reasons, as they are not dependent on each other. This increases the total number of rules on a specification but simplifies each rule into having only one check expression.
The next step is to analyze the context XCX expression of each rule. To be a valid context, the expression needs to be *non-atomic*, i.e. there are no values included in the context, only document nodes. This also means that the resulting XPath CX expression is indeed a `Node`, which contains a document reference. We also know that the document reference is not empty, as this is an XPath CX static error, so the reference contains at least one filtered class.

We can now break up each rule further, by creating a new rule for each filtered class of the context document reference. Again these rules are completely independent of each other and it can be seen as a convenience feature for the user, that he is able to specify rules with the same checks for multiple contexts at once. At this point, we are down to an even bigger number of rules, which have a single filtered class as context and a single check expression which has to hold there.

The semantics of a rule state that the check has to hold for each item in the context sequence, so we create a document reference to an arbitrary existing node by filling up the missing filters with the `Existing` constructor. This is of course only possible, if the context document reference can be filtered to uniqueness at all, i.e. all ambiguities of the filtered class have to be selectable. It is a specification error if the context of a rule includes a node class which cannot be made unique.

The resulting document reference is now known to be statically unique, which of course includes being unique, i.e. the reference selects exactly one existing document node. The document reference is then bound as XPath CX `Node` to the context item, as well as to the special variable `$context`, as described in the semantics. Using this environment, the variables are abstractly interpreted and also bound to variables of their names.

The environment now contains the reference to the context item in question as well as all variable expressions and is used to interpret the check expression. The resulting expression represents the complete rule and is only tested for being of the `simple` type boolean. Any static analysis error from the interpretation of the context, the variables or the check leads to the context rule specification being invalid, but also gives a hint to what is going wrong.

By using abstract interpretation of XPath CX, together with a RCX’ schema, we can condense a context rule into a list of typed XPath CX expressions, which represent all the different checks to be done for the rule. Both the context expression and the list of checks lead to the introduction of more of these checks, which are all independent of each other. Note that the resulting expression can still contain `Existing` filter, which therefore actually represent a set of checks to be done and can be seen as prototype for all checks.
4.4 CRL Analysis

4.4.1 Types of Rules

In [10] I defined several classes of context rules, like integrity constraints and domain specific constraints. The use of Relax CX additionally has some implications on the types of rules which can be specified with CRL, which makes analysis more comfortable, but limits expressiveness.

This work confirms that integrity constraints are indeed a special class of constraints, which need special attention. As we have seen in Section 3.1.4, the uniqueness of nodes is of vital interest to static analysis, which is based on statically knowing if path expressions select a single node only.

Primary keys and uniqueness therefore become a special category of context rules, as they are no longer independent from other rules. Algorithm specifications additionally depend on such constraints, as operation contexts can only work on ambiguous nodes, if they are filtered down to uniqueness. Batch operations can also be a solution for algorithms, yet it will always be necessary to manipulate specific nodes only.

Uniqueness constraints allow the definition of foreign keys, which is another class of integrity constraints, which will take up a special role. If foreign keys would be treated like normal constraints, algorithms and constraints would not be able to statically guarantee that the target node exists at all. If they get a special role like uniqueness, however, a whole new type of possibilities comes up. Filter expressions could be allowed to use foreign key values and the existence and uniqueness of a target node would be certified.

Foreign keys are not supported by this work, as XPath CX does not allow suitable filter expressions yet. To allow those we have to deal with generating checks for dynamically comparing filter expression values and dynamically switching between different sets of preconditions.

As far as domain specific constraints are concerned, there are also several interesting classes, which are all possible to be specified in XPath 2.0, but not so in XPath CX. This also has directly to do with the allowed kinds of filters and foreign keys in general.

Rules which have sequences of ambiguous nodes as context are instantiated for each single node. The number of these rule instances can therefore vary in their dimension, i.e. there can be exactly one instance, an unlimited number which differ in one parameter or even more, differing in several parameters.

Figure 4.6 also shows another property of rules, which is the kind of references to other nodes they contain. Type A is the most trivial one, as the reference goes to a node in the same instance of an ambiguous node. Whenever a node referenced by a type A reference is manipulated, exactly one instance needs to be checked, all other instances of the rule remain untouched.

Type B references already cause much more troubles, as many instances are affected at once, which all need to be checked separately. They are, however, still contained in the strict upper context, which shares the same filters as the
original context, i.e. the nodes are guaranteed to exist and can be precisely referenced.

Type C and D references are completely arbitrary and select specific nodes of ambiguous types classes, either in a sibling context or a descendant context. For these types of references, all instances in any dimension have to be checked.

An operation can therefore be in any relation to a rule. In the best possible case, a single operation touches exactly one instance of a rule, which is always true for rules which stay within their own subtree. Operations which change values which are less ambiguous as the rule context touch an arbitrary number of instances and cause all instances to be checked. Rules whose context is not ambiguous often need to be checked, but then there exists only one instance, which is also a convenient case.

The major concern for C and D type references is the identification of the node which is actually referenced, i.e. the source of the filter values used. In B references, the filter values are the same and nodes can be reached by using the reverse axis. For C and D references these values can either be constant or they are based on node values of the document.

The latter is the worst case scenario, as it makes statical analysis very hard to do. For each operation on a node which appears in a type C or D reference, it is completely unclear which instances of a multi-dimensional rule are touched, as this depends on the actual document values.

In XPath CX, however, nodes of ambiguous node classes can only be selected uniquely by using constant filter values taken from parameters. The other option is to select all nodes of the class and apply an aggregate function to them. Both scenarios can be handled by static analysis. As context rules don’t have any parameters they are unable to select specific nodes from ambiguous node classes,
which eliminates type C and D references altogether.

Rules can still be specified over arbitrary nodes, which are capable of referencing the less ambiguous upper context, but they cannot use dangerous, value-based references into arbitrary other ambiguous parts of the document. This can be possible in future revisions of CRL, when foreign keys are supported as special type of constraint.

Note that optional nodes can also not be uniquely referenced in XPath CX and therefore also in CRL. The consequence is that all unique node references in context rules are known to exist, whenever the rule instance itself exists. An algorithm operation can only alter the value of uniquely referenced nodes or add new ones, but if it removes such nodes, the instance is removed as well.

4.5 AL Analysis

For static analysis, a method can be condensed nearly as good as it is possible for context rules. The reason is that variable and node definitions can again be inlined or completely interpreted and the test expressions of \texttt{if} or \texttt{elseif} commands can be discarded altogether.

Each operation block is independent of all others, as only one will be executed each time the method is invoked. The test expressions can of course be used to optimize the runtime checks to be done for a block, but this is not within the scope of this work. So the static analysis result of an algorithm specification is only a list of operation blocks, which in turn are lists of elementary operations which are inlined as much as possible.

I will use the analysis results of XPath CX expressions to build up an environment which can be used to interpret the operations. Initially this environment contains the document as context item, to make the \texttt{root} function work, and all the parameters are bound as variables to \texttt{Unknown} values, whose label is the parameter name. In addition to this environment, a dictionary of node definitions, which is initially empty, is maintained, like it is described in the semantics of AL.

The variables of a command block are abstractly interpreted and the resulting typed XPath CX expression of each variable is checked against its declared type. A type missmatch is of course a specification error and the analysis can terminate. The typed expression is then bound to the variable name in the environment.

Node definitions have all their embedded XPath expressions evaluated in the current environment and the resulting typed expression is inserted there instead. The interpreted node definition is then bound to its identifier in the dictionary. Remember that all variable names have to be different, including the parameters, but node definitions have their own namespace, in which they also need to be unique.
Whenever an `if` expression is encountered, all cases of the `if` are split up and create a new operation block each and are analyzed independently from now on. The test expressions are disregarded in this work. After this step we are down to a list of operation blocks, with associated XPath CX environment and node definition dictionary.

**Removes** The remove operations are the first ones in line, so they operate on the starting document. As the removal of a node automatically removes all other descendant nodes, different remove operations can never depend one each other and ordering of removes becomes insignificant. At the current state of AL and XPath CX, it is also always known if two remove contexts intersect and two contexts can never include the same node without that being statically known.

The context XCX expression of each remove operation is analyzed using the current environment and needs to specify a single node or a sequence of nodes. In either way the resulting typed expression is known to be a document reference. If this reference is not statically unique, it has to contain only optional node classes. It is a specification error if none of the two conditions holds.

To get all possibly affected nodes, the descendant-or-self axis is applied to each document reference. The resulting sets of filtered node classes of each remove operation are required to be disjoint from all others. Note that two filtered classes which share their node class can still differ in their filter. If the filters of one are a prefix of those of another, however, one class includes the other and they are not disjoint.

The result of this analysis is then a list of document references of nodes which are removed. To facilitate later steps of analysis, the document references of each remove operation are split up, creating one separate remove operation for each filtered class. Remove operations are independent to each other, so the removes for each node class can be done and analyzed individually. Note that this does *not* make the context document reference of remove operations narrowed, as the number of filters can still be less than the maximum.

The original root node classes of remove operations are additionally returned as *primary* remove operations, whereas all remove operations for node classes calculated with the descendant axis are returned as *secondary* remove operations.

**Appends** The context XCX expression of an append is analyzed first and has to result in a *simple* complex node. The *Node* is then known to be a document reference, but it additionally has to be a statically unique reference. This guarantees that the context is described by a single, completely filtered node class of inner nodes.

Using this starting node class, the XML fragment to be appended is evaluated, so that finally all nodes appended are identified by their narrowed node
4.5 AL Analysis

class. As it is possible to append ambiguous nodes, which are not selectable, we cannot assume more static properties of these node classes than being narrowed down as much as possible. This is not a problem, as appended nodes of node classes, which are not statically unique cannot be referenced uniquely from within constraints anyway.

Note that appended XML fragments, which are defined inside of the append operation first need to be interpreted like described for node definitions above, using the current XPath environment. This replaces all XPath expressions inside the fragment with typed values. What happens now is another interpretation of the fragment, together with the context of the operation, to see which node classes are appended and if they exist at all.

Interpreting an XML fragment with a context reference is a recursive procedure, starting with the top level element, attribute or node reference. Node references are looked up in the dictionary, which leads to a specification error, if the label is not defined there. If the label is found, interpretation continues with the saved fragment.

If an attribute is encountered, an attribute axis step is performed on the current context node class, which is followed by a name test using the attribute name. This can select at most one attribute node class and it is a specification error if no such attribute exists.

The type of the value to be placed in the attribute is compared with the type of the narrowed document reference and their kind has to match. The value also needs to be a simple value, not a sequence. It is then returned together with the narrowed node class, as a resulting append operation.

If a text node is encountered, the current active context element has to be non-complex and the typed expression given as value has to share the kind with the type of the element. Essentially the same checks have to be done as for an attribute definition, as the element is used like one. The element does not contain any attributes or elements, so its value and node class are returned as a resulting append operation.

The most interesting and also the only recursive case, is it that of an element definition. Similar to the attribute case, the narrowed class of this new element has to be retrieved. This is done by applying the child axis to the context reference, followed by a name test using the elements name. As RCX' has additional restrictions concerning intersecting name classes, the result is also either one class or none.

The latter case is a specification error, as the appended element cannot appear in the context element. In the former case, we got a new context node class, which is not known to be narrowed yet. To achieve this we have to test if the parent node class is statically unique and if the current one is ambiguous and selectable. If that is the case, it can and has to be filtered by one more step, using the value of the id attribute, which has to exist.

The existence is asserted as ambiguous nodes are only selectable if they have a
4 Static Analysis

A non-optional id attribute. This id attribute is now searched for in the attribute definition list of the XML fragment. If it is absent or defined more than once, it is a specification error. The typed expression of the value of the id attribute needs to be an unknown value, i.e. a parameter value in AL. This value is used to filter the document reference, which yields a new narrowed context reference.

Now all recursive interpretations are done on the list of attributes and elements embedded in the element and the results are returned, together with the elements own filtered class, but no value. If a document reference is returned in an append with no value, the node was either appended as inner node or as empty node.

Finally, the result of the static analysis of an append operation is a list of all appended nodes, represented by their document reference which contains only one narrowed node class, together with a typed expression as value, if the node has indeed a value and is no inner node or empty node. The root nodes of XML fragments appended are additionally returned as primary append operations, whereas all other nodes subsequently added are returned as secondary append operations.

A last consistency check, which is done on the result set, checks for duplicate statically unique node classes in the append list. This would mean that a uniqueness constraint is violated, which is a specification error. It is no problem, however, if a narrowed class appears more than once, as there can be arbitrary many nodes added of such classes.

Updates The context XCX expression of an update operation is analyzed and has to result in a simple non-complex node, i.e. a node with an atomic type. The corresponding document reference also has to be statically unique, so the context of the update is a single, completely filtered class.

The value XCX expression is analyzed and the resulting typed expression has to match the kind of the context reference and must not be a sequence. The result of the update operations analysis is the pair of a statically unique document reference and a value expression.

4.6 Preconditions for Context Rules

In this section, I will combine the results of the analysis steps of the different languages and retrieve more static information. The major goal is the generation of preconditions for methods, which guarantee the successful execution of the method, whenever they are true for an input document.

The analysis starts with the results of the algorithm analysis, which is one operation block, with the initially removed document references, a list of narrowed append operations with optional typed value expressions and a list of statically unique updates with values.
An initial document, which is a given to an algorithm, has to be valid according to the same CXLD the algorithm was defined for. Only for this case viable information and guarantees can be retrieved. If this is the case, the document is structurally correct according to the Relax CX specification, and all matching rule instances hold.

After the execution of the operation block, the structure should be valid again, the remaining rule instances should still hold and any newly introduced instances have to hold as well. So the analysis will split into two major parts, one concerned about the structure, the other about the context rules. For this section, I will assume that the algorithm maintains structural correctness, so the focus is on context rules and their instances.

The different operations have different effects on rule instances and their embedded XPath CX expressions. There are three major ways to influence an expression:

- An item is removed from a sequence, its effect on the sequence has to be compensated.
- An item is added to a sequence, its effect on the sequence has to be added.
- An item is changed, its effect on sequences has to be both compensated and added. Direct references to the item are replaced by the value of the item.

The essential advantage of context rules with XPath CX is that direct references to unique nodes can never be removed or added, without removing or adding the whole instance of a rule. This allows to handle appends of nodes which are directly referenced like updates, i.e. the reference is simply replaced by the appended value. Optional nodes, which are not necessarily appended, cannot be referenced directly, so this procedure is safe.

Note that uniqueness constraints of selectable, ambiguous nodes are not explicitly formulated, but still have to be maintained by methods. As remove operations can only remove such nodes and updates cannot modify id attributes, append operations are the only ones that can cause such constraints to fail. It can therefore be necessary to generate additional preconditions for uniqueness constraints, which express that some nodes mustn’t exist yet.

In addition to instances of rules and uniqueness constraints, we also have to make sure that the XPath expressions used in the method terminate correctly and do not yield any errors. The analysis of XPath CX expressions takes care of any static errors and type errors, but there can still be some dynamic errors.

The problem are statically unique references, as the set of nodes is only known to consist of either one node or no node at all. The list of filter values used need not exist in a concrete document. For this reason both update and append operation contexts can generate additional preconditions, which express that some nodes have to exist.
4 Static Analysis

**Updates** The generation of preconditions starts at the end of the operation block, where all instances of all rules should hold. Each rule is represented by the result of its static analysis, which is a typed expression containing references which are filtered with the special *Existing* filter. The typed expression therefore contains all *existing* instances of the rule at the end of the method at once.

Each update operation has a statically unique context, which references exactly one node. The existence of that node has to be shown as well, but for the moment we assume it to exist and take care of it later. The single node class of the reference is maximally filtered, i.e. all ambiguous nodes have a fixed filter value. Using this filtered class, we now calculate the instances of each rule, which are manipulated by the update.

This is done by going through all document references contained in the rules typed expression and comparing the filtered classes with the one of the update. Unique document references can contain several filtered classes, which are all mutually exclusive. As the context of the update operation is known to exist though, all other cases are known not to exist.

Whenever a document reference in the check contains a filtered class that has the same node class as the update context, the filters have to be compared. *Existing* filter values act as a wildcard for concrete filter values and whenever the filters of an embedded filtered class are a prefix of the filters of the update, the rule references the updated node.

The matching filtered class can thereby specify some of the *Existing* filters of the rule, which determines the instances the update affects. The idea is that “update x touches rule y, whenever the context node of y has the same filters as the context of x”. The updated node can be an ancestor node of the rules context, in which case not all *Existing* filters are determined and a whole class of instances is affected.

Instances of rules are identified by lists of concrete filter values which have to be inserted for *Existing* filters. The empty list therefore specifies all instances, not maximized lists specify subclasses of instances and complete lists define exactly one instance.

```markdown
<table>
<thead>
<tr>
<th>type</th>
<th>InstanceMap = Map [String] TypedExpr</th>
</tr>
</thead>
<tbody>
<tr>
<td>data</td>
<td>Precondition = Pre Rule InstanceMap</td>
</tr>
</tbody>
</table>
```

Listing 4.13: Preconditions for Rules

To handle the different classes of instances of a rule, I use the types defined in Listing 4.13. An *instance map* contains all relevant classes of instances which need to be checked, together with their current typed expression. At the start of analysis, the instance map of a rule contains the original typed expression using the empty list as key. A *precondition* is a rule together with its modified instances.
4.6 Preconditions for Context Rules

The classes of instances, which were determined to be touched by an update, are now separated from the their parent classes, if this wasn’t yet the case. If the instance map already contains typed expressions for the touched classes, nothing has to be done. If they don’t contain a touched class, the class is added to the instance map by inserting the parent class using the key of the instance.

The key of the parent instance class has one filter value removed from the list of fixed filters. If the parent class does not exist as well, it is created recursively. As the root class, where no filters are fixed, always exists, it is always possible to create a subclass.

The idea behind this procedure is to identify all classes of instances which are modified and create a special check for them. On creation of a subclass, the typed expression of the parent class is no longer checked for the instances of the subclass, both now have their own typed expression to check.

After the instance map of each rule is enriched with the touched subclasses of the rule, the effect of the update is applied to all typed expressions of instances in all instance maps. This is done by replacing direct references to the update context, which do not contain Existing filters, with the typed expression of the update. References in sequences are compensated for each different use of the sequence individually.

Occurrences in a sum are compensated by creating a subtraction of the old value and an addition of the new one. The former is inserted using the context of the update itself as value, the latter is created by inserting the Locked value of the update. The inserted Node value, using the update context document reference, is not locked, as it might have been modified by another update or an append earlier.

Occurrences in a count can be disregarded, as changing the value of a node does not change the fact that the node is still there and can be counted.

These changes create new rule checks, which can be executed before the update is actually done, as the effects of the update are compensated. To prevent further manipulation of the inserted value, which can contain other document references, it is inserted into the typed expression using the Locked constructor.

Whenever effects of operations are applied to typed expressions, all subexpressions in Locked constructors remain unaffected. Note that update values are evaluated on the original document, so they must not be adapted by operations in any way.

This procedure of compensating update operations for touched instances, is done for all initial preconditions, starting with the last update and moving backwards in definition order. If a rule is touched by any update at all, its precondition is likely to contain more singled out instance classes and not the prototype instance only.

Besides the preconditions, it is necessary to collect all statically unique context references of the updates, as these have to exist for the method to work. Statically unique classes only restrict the number of concrete nodes to at most
one, but the existence can of course not be asserted. Append operations can statically guarantee some nodes to exist, whereas the remaining nodes mustn’t be removed at the beginning of the block and their existence has to be asserted by a runtime check.

**Appends** The first thing we have to do for append operations is to single out newly created instances from their instance classes. This is because new specific instances cannot always be checked with an instance class check, as the instance class might be empty in the original document. The new instance will be checked on its own, only if the check contains references which are restricted to the newly appended subtree, which might not be the case.

For this reason we have to insert newly created instances into the instance maps of matching rules, which is done by comparing each append operation context with the rule context. Both contain only one filtered class, whose pattern position has to match for an instance to be created. If this is the case, the filters of the append operation context define the precise instance.

The difference between instance classes and precise instances is that the former have to be checked using the `every` quantifier of XPath 2.0, which only instantiates the check for as many nodes as the original document had, while the latter is a lone check expression, which is checked no matter how the original document looks like.

```hs
 updTouched :: DocRef -> DocRef -> [[String]]
 updTouched drCon drRef | narrowed drCon = maybeToList $ do
   let (DocRef ref) = drRef
   let (DocRef [(cl, fi)]) = drCon
      -- either get the filtered class that matches
      -- or return an empty list of touched instances
   f <- lookup cl ref
      -- get the "Existing" filters of the rule and the rest
   ; let (rex, rfi) = span isExisting f
      -- take same number of filters to determine the instance(s)
   ; let (cex, cfci) = splitAt (length rex) fi
      -- the rest of the filters have to be equal
      -- or the reference is not touched by the operation
   ; guard (rfi 'isPrefixOf' cfci)
      -- return the filters which were fixed by the operation
   ; return (map fromUnknown cex)

Listing 4.14: Determining Touched Instances of a Rule
```

The compensation of append operations does not differ much from the one
for updates. Again all touched instances of each appended node are calculated, using the same algorithm as before. Listing 4.14 shows the algorithm, which takes the document reference of an embedded reference in a rule, as well as the narrowed instance of an update or append operation.

Update contexts are always statically unique, whereas appends can either be statically unique or narrowed. Note that statically uniqueness implies being narrowed.

Being narrowed, the context reference is known to contain only one filtered class. The node class it then looked up in the list of filtered classes of the rules reference, where it is either found or the reference cannot be touched by the operation.

To determine which instances are affected, the longest prefix of Existing filter values in the reference are extracted from the operation context. The remaining filters of the reference have to match those of the operation context or the rule does not reference the manipulated node. In fact, references in context rules can have no other filter values at the moment, but the check will be necessary in the future.

As for update operations, all instance classes touched by an append have to exist in the instance map and are created if the map does not contain their key. The value of the appended node is then compensated in all typed expressions of all instance maps.

The first difference to updates is that appended nodes need not have values associated, in which case they are either inner nodes or empty nodes. Such nodes are never directly references and can therefore only have an affected on sequences which are used in functions like count.

The compensation for this function is the next difference, as appending a node to a sequence which is counted means that the result has to be adjusted by one. The function is therefore wrapped in an addition of the integer literal 1.

The last difference is the sum, where no compensation for the old value is needed, as the node did not exist. Note that append is also valid after an existing node was removed, but in this case the compensation for the old value will be done when the remove operation is crossed.

Another thing to note is that the compensation of node manipulations in sequences, which are referenced by a unique reference, can be optimized by removing the reference from the sequence altogether, in which case the compensation is unnecessary and previous manipulations of the node need not compensate the compensation just done. This is a rare case, however, as sequences most often will contain complete node classes and no rewriting within the limits of XPath CX can be done to remove single node values.

Listing 4.15 shows the function used to compensate the affect of an operation, given as narrowed context, on a document reference contained in the check of a rule. The function is applied in post order to each node of the typed expression and is the identity for each constructor which is not manipulated.
4 Static Analysis

 updModify :: DocRef -> Maybe TypedExpr -> Bool -> TypedExpr -> TypedExpr

 updModify drCon value' new te | narrowed drCon = case ex te of
 Node drRef -> maybe te id $ do { let (DocRef ref) = drRef
 let (DocRef [(cl, fi)]) = drCon

  -- test if the reference is touched
  ; f <- lookup cl ref
  ; guard (f == fi)

  -- get the value and return it instead
  ; let value = fromJust value
  ; return $ TE (Locked value) $ ty value
 }

 Count drRef -> maybe te id $ do { let (DocRef ref) = drRef
 let (DocRef [(cl, fi)]) = drCon

  -- test if the reference is included
  ; f <- lookup cl ref
  ; guard (f 'isPrefixOf' fi)

  -- no action needed if the update is not new
  ; guard new

  -- return the same expression, plus an additional one
  ; let one = TE (Literal "1") proto { kind = TInt }
  ; return $ TE (BinOp BPlus te one) $ ty te
 }

 Sum drRef -> maybe te id $ do { let (DocRef ref) = drRef
 let (DocRef [(cl, fi)]) = drCon

  -- test if the reference is included
  ; f <- lookup cl ref
  ; guard (f 'isPrefixOf' fi)

  -- get the value to insert
  ; let value = fromJust value'

  ; let oldValue = TE (Node drCon) $ typed drCon
  ; let compensated = if new
    then te -- no compensation needed if new
    else TE (BinOp BMinus te oldValue) $ ty te
  ; let locked = TE (Locked value) $ ty value
  ; return locked $ ty te
 }
 otherwise -> te

Listing 4.15: Rewriting References in Rule Checks
Every possible effect of a node has to be handled individually, as different actions have to be taken to compensate it. Each relevant case of constructor, in which a document reference can appear, returns the original expression, if any guard of the case fails.

As the operation context is known to be narrowed, it consists of one filtered class only, which is compared to the document reference embedded in the constructor. There are many more implicit assertions, which could be explicitly checked, but are not shown in the listing.

A Node, for example, is never a sequence, i.e. always simple, as the expression which contains the node would not be type correct otherwise. For the same reason, the value parameter is known to actually contain a value, as the append cannot have an inner node class as context if it matches the Node which is type correct.

As with the update operations, the precondition for each rule is modified by each append in reverse definition order. The resulting preconditions contain specialized check expressions for different classes of instances, which can be checked on the document without actually doing any append or update operation.

The set of statically unique references, which have yet to be asserted to exist, which was initialized with the contexts of all update operations, is now enlarged by inserting the parent elements of all primary append operations. A primary append operation is the append of the root node of an XML fragment, so the parent is the statically unique context of an original append.

This set is now reduced by discarding all document references for which an append operation exists, i.e. a node was created by the method. This takes care both of appended nodes which are later updated, but also of sequenced append operations, where the context of a later one is appended by an earlier one. What is left over are all statically unique references whose existence is not statically known.

Append operations introduce another class of preconditions, which assert that no uniqueness constraint is violated. Each append operation, which appends a node of an ambiguous node class that is filtered by a parameter value, generates a check that no such node already exists.

This kind of constraint is exactly the opposite of the existence constraints handled above in that it explicitly forbids certain statically unique nodes to exist. The set of all such constraints mustn’t contain duplicate entries or a specification error is raised, as two nodes are added which violates the uniqueness of a node class.

Removes The last actions which need compensations are the remove operations. We cannot always use the same functions as for the other operations, as the contexts of removes are not necessarily narrowed down, i.e. filters are not always maximized.
But the first thing we need to check is if any nodes which are required to exist, and which were not appended by the method itself, are possible to exist at all. This check can be done by using the `updTouched` operation shown in Listing 4.14, as the context of a remove is either a sequence or a direct reference, so a necessary node is removed if it is touched be the remove context.

It is a specification error if the check fails, as the operation block can never be executed, no matter what the input document looks like. If the check holds, an existence check for each document reference needs to be created.

Next, we have to check that certain nodes do not exist, as it is required by certain append operations. The list of all statically unique references is again filtered using the `updTouched` function, but this time an item which is touched is simply removed form the list and does not yield a specification error. For the remaining items it is again necessary to create runtime checks.

The more important consequence of remove operations is that rule instances can both be removed or altered by removing nodes. Lets first look at the instances which are removed. The problematic thing at this point is to figure out which instances are really removed and which have to be checked anyway, as an append brought them back up.

So the first step is to go through all remove operations and compare their document references with the context node classes of each rule. Instances of a rule are removed, whenever their is a filtered class in the remove operation, which shares its pattern position with the rule context and whose filters are a prefix of those of the context, where the `Existing` filter values of the rule act as wildcards.

The list of fixed `Existing` values specifies the class of instances removed. By going through all remove operations we get a list of all instance classes which are removed at the start of the operation block. As nodes can only be added to the document one by one, append operations can each create single instances only. By going through all append operations with statically unique contexts, and comparing their context with the rule contexts, we get another list of most specific instances which are known to exist.

As rule context references and append context references both contain one filtered class only, they can only match if their classes are equal. Both filter lists are then known to be equally long, as the only appends with statically unique contexts were considered. This means that the filter list of the append specifies the instance precisely.

The list of appended, precise instances is now removed from the list of instance classes gathered from the remove operations. For each rule, the remaining removed instance classes are either created or replaced with the constant typed expression `fn:true()` in the instance map of the corresponding rule. Additionally, all more precise instance classes, which are not included in the append instance class list, are removed from the instance map, as they are not appended and covered by their parent instance class check.
4.6 Preconditions for Context Rules

What is left to do now is to compensate for the remove operations in the remaining checks in all instance maps. The procedure to do so is essentially the same as for append and update operations, but the main differences are that removes can work on sequences of nodes, not only single ones, and that they don’t have associated values for these nodes, as they just remove the old ones.

These differences originate in the context references of removes not being narrowed. This means that the list of filters can actually be shorter than the list of filters of a matching reference in a rule check. In this case a whole part of a sequence referenced by the check is removed altogether.

So to determine the touched instances of a remove operation, the embedded references in the prototype check of a rule are again compared to the context reference of the remove. This time, however, the context reference touches the embedded reference, if it either completely contains the referenced sequence of nodes or is completely contained in it.

In both cases the node class of the context reference has to be found in the list of filtered classes of the embedded document reference, but the cases differ in which filter list is a prefix of the other. Overlapping parts of filters still mustn’t differ for a remove to touch a rule check, but it is also possible that the filters of the operation context are actually shorter.

At this point, it is vital to note that there can never be direct references which are removed, as the existence of directly referenced nodes in rule instance checks is bound to the existence of the instance itself. So the remove operation either already removed the instance and placed the value \texttt{fn:true()} in the map or the removed value was added again and the reference is replaced in all checks with the new value.

What is left are references in sequences and these can always be removed safely which leaves a shorter sequence, but never an invalid expression. All touched instance classes are inserted into the instance map, if not already present, by copying the check of a parent class. Then all touched references in all check expressions in all instance maps are compensated.

The compensation of references which are completely contained in the remove context is simply done by removing that filtered class from the document reference. Compensations for sequences that contain the remove context cannot be done that simple, as rewriting the expression on XPath CX does not allow to formulate the subtraction of two node classes which represent sequences.

So for \texttt{sum} functions the sum of all removed nodes has to be subtracted from the main sum, i.e. for removes it is not enough to subtract single values, but we need another sum to do it. The same goes for the \texttt{count} function, where we need to count the removed nodes and subtract that number from the main count.
Static Analysis

Translation  The main results of the precondition generation split into three different groups:

- A list of statically unique document references which have to exist.
- A list of statically unique document references which must not exist.
- A list of instance maps for each rule.

To actually convert this data into executable expression we have to convert them back to XPath 2.0.

Existence checks in XPath 2.0 can be trivially done by giving a path that selects the node in question and wrapping it with the count function and a test if the result is greater than zero. The conversion of a document reference to a selection path is a subproblem of converting general XPath CX expressions back to XPath 2.0.

Conversely, non-existence of nodes can be asserted by counting the result of a reference and comparing the value to zero. This concludes the check that appended nodes never violate uniqueness constraints.

The translation of instance classes needs some additional logic, as the check has to be done for all existing context nodes of the class. Whenever subclasses exist in the instance map, which means that they have their own check, they also need to be excluded. The subclasses have additional filter values for an ambiguity of the context reference, so all contexts with this specific value mustn’t be selected.

To check an instance class, the every operator of XPath 2.0 can be used. It has to contain a sequence of items, which are each in turn bound to a variable. The expression given as body of the quantifier is then evaluated for each node of the sequence using that variable and the result of the evaluation has to be true.

every $id in /context/path/@id satisfies <check>

The idea is now to bind every existing id attribute value for each unfiltered ambiguity of the context reference of the rule to a variable, which is then used in the filters of document references in the check. To exclude a subclass, which has an id attribute fixed, the check can be made conditional by testing for this specific id and returning true if it matches.

every $idA in /context/path/prefix/@id satisfies
every $idB in /context/path/prefix[@id = $idA]/full/@id satisfies
  if ($idA = $id) then fn:true() else
    /context/path/prefix[@id = $idA]/full[@id = $idB]/@counter <
    /context/path/prefix[@id = $idA]/@maxcounter

The example expression tests a context rule which specifies that the counter attribute of the context node is always small than a maximal value set in the
4.6 Preconditions for Context Rules

parent element. The instance class used here is the complete class of all instances, as both ambiguities are not filtered with a fixed node, so we need to introduce two quantifiers to cover all existing ids.

As there exists a subclass which has one more filter value fixed, it is excluded by making the check optional in this case. This class was most probably created because the maxcounter attribute was changed by an operation, maybe it was set to the value of another parameter of the method. The check for this instance class would then be look like this:

\[
\text{every } \$idB \text{ in } /context/path/prefix[@id = \$id]/full/@id \text{ satisfies } /context/path/prefix[@id = \$id]/full[@id = \$idB]/@counter < \$param
\]

In this case only one ambiguity was unfiltered and the first one was fixed. Additionally, the check was rewritten by an operation, which placed the new value for the maxcounter into the check. The example also shows that the check for the complete class of instances is in fact the original, unmodified rule check.

It will happen quite often that the prototype check is inserted into the instance map, copied to more precise instances later, but is then never modified itself. Note that as long as all references of a check include Existing filter values, they can never be changed. This makes sense as such references still abstract from a concrete value and need to be fixed in a more precise instance class.

Only if a check contains references to a less ambiguous context it can happen that an instance class has a modified check. The consequence is that instance classes which still have the prototype check are not generated as precondition, as the initial document conforms to all rules and all such checks are known to hold. In the example above this would mean that the first of the two check expressions would not be created at all.

**XPath CX to XPath 2.0** What’s left to do is to convert the check expressions back to executable XPath 2.0 expressions. Most conversions are trivial, as XPath CX is a strict subset of XPath 2.0. The most interesting part is how document references can be converted back to path expressions, which select exactly the members of the reference, as the old selection path is no longer available.

Fortunately, the RCX’ subset of Relax CX has additional constraints, which make the selection of nodes by simple axis steps statically secure. There can never be two elements on the child axis of a pattern position, which have intersecting name classes. So each path which starts at the root pattern and applies only child axis steps with name tests, as well as filters, can select at most one node class.

So to convert a node class back to a selection path, it is sufficient to create an XPath path expression which is a cascade of child axis steps with name tests for all ancestor element positions of the associated pattern position of the node
class.

To convert a filtered class one needs to insert filters at each ambiguous, selectable node crossed, if a filter is actually present for it. If the filter value is Existing, the value inserted in the filter test needs to be the corresponding variable bound in an enclosing every operator of the instance class.

Finally, to convert a complete document reference to a selection path, all filtered classes are converted to selection paths, which are then combined using the union operator.

### 4.7 Remarks to Structural Correctness

The second part which needs to be checked and was left out in the previous section is the structural correctness of an operation block. An input document which conforms to a CXLD was assumed to be structurally correct after the operation block is done and it was then shown how to calculate all additional checks which need to hold for it to conform to the context rules after the block and to generally not produce any errors.

I will not give any algorithms in this work which statically assert structural correctness and eventually produce additional runtime checks, as structural correctness is a well understood domain in the area of XML. I will, however, give the general approach of the analysis, point out some of the more difficult parts and show some implications of the used languages in this work.

Remove operations are restricted to be able to remove only optional nodes or statically unique nodes. The only thing to watch out for when removing optional nodes is that sometimes one node has to remain in the end, if a one-or-more pattern is not contained in a choice pattern combined with the empty pattern.

If all such optional nodes are removed there has to be at least one matching append operation. If only single nodes are removed and there are also not as many added by append patterns, it will be necessary to generate runtime checks. If a statically unique node is removed, it can of course also be optional, in which case the same rules apply.

Append operations now face several different problems. The first one is that nodes which cannot be repeated mustn’t be added more than once, which coincides a bit with the previous checks of removed nodes. Whenever non-repeated nodes are added there has to be a corresponding remove operation for it.

Handling choices is the next problem and will be the most difficult problem for static analysis. Choices are not very restricted at the moment, so it is, for example, possible to combine a single element as the first option, with a repeated element as the second option. But even if there are only two mutually exclusive elements, the checks for removed nodes have to be made more sophisticated.

It is possible to remove a statically unique node, which is not optional, yet contained in a choice with mutually exclusive other elements. Subsequent ap-
4.8 Example Analysis

pend operations can solve this problem by adding the mutually exclusive nodes instead of the removed one. Similarly it has to be possible to remove all nodes of a repeated node class, which is mutually exclusive to another node class and than add nodes necessary in this node class.

The final problem of append operations is in fact the easiest, as it is necessary to check if an element or attribute can be added at all to the context. This is not hard to check, as starting from the node class of the context, the node class of the appended element or attribute is either non-existant or uniquely defined.

In the former case this is a specification error, in the latter the pattern for the appended XML fragment is known, so it can easily be checked that the rest of the fragment is structurally valid by using the validation algorithm provided by James Clark [3]. To check that the root element is no duplicate or exclusive to other nodes, is only a problem because of choice patterns, which was already covered.

As RCX is even a subset of Relax NG, with several severe restrictions, the validation algorithm by James Clark can most probably even be simplified, as several sources of ambiguities are forbidden and datatypes as well as name classes are also severely restricted.

Datatypes are the last source of problems for structural correctness. That appended and updated values have the correct datatypes is already taken care of by the static typing of XPath CX, but Relax CX additionally allows to define enumerations of existing datatypes.

So whenever a value is assigned to a node which has an enumeration type, the assigned value has to be checked for being within the enumeration. Such checks can only be omitted if the value originates in a node value of the same enumeration, a subset of the enumeration or is in fact a literal of the correct type which is defined in the enumeration.

4.8 Example Analysis

To conclude the static analysis section, I want to supply a very small example CXLD and some methods working with it. The example shows some of the things that can be done using the defined languages and also gives a feeling for the generated preconditions.

Listing 4.16 show a very small CXLD of a system which manages an inventory containing items of some kind. The inventory stores the date of the information stored in the document and has a capacity. There can then be arbitrary many items contained in the inventory, which have an identifier, the date they were added, a name, the number of items of that kind and the total size.

The rules defined for such inventories are that all dates, counters and sizes are always greater than zero, that the contained items actually fit into an inventory of that size and that the date an item was added is always smaller or equal to
the current date.

\begin{verbatim}
start =
element inventory {
    attribute date { xs:integer },
    attribute capacity { xs:integer },
element item {
    attribute id { xs:integer },
    attribute since { xs:integer },
    attribute name { xs:string },
    attribute count { xs:integer },
    attribute size { xs:integer }
}
}

R1 (/ inventory)
! @date >= 0
! @capacity > 0
! sum(./item/@size) <= @capacity

R2 (// item)
! @since >= 0
! @since <= ancestor::inventory/@date
! @count > 0
! @size > 0
\end{verbatim}

Listing 4.16: Example CXLD

The example does in no way utilize the possibilities of Relax CX and CRL, but it is sufficiently complex to show the static analysis results of some algorithms.

\begin{verbatim}
addItem(int id, string name, int count, int size) {
    path inv = /inventory;
    append [ $inv ]
    <item id = [$id]
    since = [$inv/@date]
    name = [$name]
    count = [$count]
    size = [$size] />
}
\end{verbatim}

Listing 4.17: Example Method: addItem

The first method we look at simply adds an item to the inventory. Most of the properties of the item are given by parameters, but the insertion date is set to the current date. The preconditions generated are the following:

\begin{verbatim}
count(/inventory/item[@id = $id]) = 0
fn:sum(/inventory/item/@size) + $size <= /inventory/@capacity
/inventory/@date >= 0
/inventory/@date <= /inventory/@date
$count > 0
$size > 0
\end{verbatim}

90
The first precondition generated asserts that no item with the supplied id exists, as the append operation would violate the uniqueness otherwise.

The first and second check of the first rule did not result in any precondition, which makes sense as they concern the inventory date and capacity only, which are unchanged. The check containing the sum, however, generated a precondition which checks that after adding the item, the sum of item sizes is still within the limits of the capacity.

The next precondition seems redundant, as it looks exactly like the second check of the first context rule. This check was actually generated by the first check of the second rule, which states that the insertion date has to be greater than zero. This date was set to the value of the current date of the inventory, so this check is perfectly valid. Of course this check can be removed, the condition is known to hold for the input document, but this is a property which cannot be discovered that easily in general.

The next precondition is another example for potential optimization. The insertion date has to be smaller or equal than the current date, which results in a check which is a tautology. Again this is not recognized by the static analysis and has to be discarded by other technologies. Last but not least the remaining two checks of the second rule generate preconditions which check the associated parameter values.

To demonstrate dependent operations and preconditions for instance classes, I will now append more and more update operations at the end of the method. At first the current date of the inventory will additionally be incremented by one:

```
update [$inv/@date] $inv/@date + 1;
```

This change leads both to changes of old preconditions and some new ones:

1. /inventory/@date + 1 >= 0
2. /inventory/@date <= /inventory/@date + 1
3. /inventory/item[$other]/@since <= /inventory/@date + 1

The first precondition is a new one, which originates from the first rules, whereas the second is adjusted, but they both are of course also valid now. The second check, however, also shows that the reference to the date on the left side was not the same reference as the one on the right side.

The third preconditions, which is also new, checks a complete class of instances. For convenience reason I have left out the complex construction which selects the correct set of id attributes, so the variable $other should be read as “for each id but $id”. This precondition was anticipated, as we changed a global value, which was referenced from all instances of the second check of the second rule.

The new item, however, needs a special check to establish this rule and is excluded from the instance class check. In this case it would not be necessary
to exclude the new item, as it did not exist in the old document anyway. If the
item had been removed first, it would be necessary to exclude the id, as that
instance of the rule is removed.

The instance class check is generated correctly, yet it can be completely dis-
carded again. All instances checked did hold for the old value of the inventory
date and as the new value is even greater, the comparison cannot fail. This
shows once more, that from the few checks which are generated by static anal-
ysis, even more can be optimized or even discarded altogether. This shows the
great benefit of static analysis, opposed to dynamically checking all instances of
all rules, whenever a method is done.

As a last change to the method, we add another update operation, which
changes the size value of another item. Remember that the two ids supplied as
parameter have to be different, which will result in an additional check.

```
update [ $inv/item[@id = $adj]/@size ] 50;
```

The method also got a new parameter adj, which describes the id of the
item to adjust. The added update operation does nothing really useful, but
shows how constant values can lead to more preconditions that can be statically
discarded by an optimizer. The new or adjusted preconditions are the following:

```
not( $id = $adj )
count(/ inventory/item[@id = $adj] ) > 0
fn:sum(/ inventory/item/@size ) + $size
   - / inventory/item[@id = $adj]/@size + 50 <= / inventory/@capacity
50 > 0
```

The first condition checks the parameters to be different, whereas the second
condition needs to assert that the item to adjust exists at all, as it is not ap-

dended in the method itself. The third condition was adjusted from an old one
and got more complex. The old size of the adjusted item has to be compensated
and the new one is added to the sum. Finally, the new size also has to be greater
than zero, which can even be statically checked and discarded.

This method also shows the problem of different parameter values having
the same value, which are then used in filters. If $id is equal to $adj, the
same items are specified by the append and the last update operation, which
invalidates several preconditions in the example.

The check for the initial value of the sum being greater than zero mustn’t
be checked in this case and the constraint of the sum of sizes would need to
be adjusted to the version that is generated if the update contained the $id
variable instead, which is:

```
fn:sum(/ inventory/item/@size ) + $size - $size + 50
   <= / inventory/@capacity
```

The update operation generates a compensation, as seen in the former precon-
dition, which is then again adjusted by the append operation, which changes the
4.8 Example Analysis

expression to the initial value. As the append operation also added the initial value to the sum, the changes even out and only the final value remains.

If parameter values are allowed to have the same values, both variants of preconditions have to be generated and selected with a dynamic check. This is but one example why current prototype languages are as restrictive as they are, as many additional features need more dynamic checks, which then have a huge impact on preconditions.
4 Static Analysis
5 Variations and Future Work

Section 4 showed that by severely restricting the input languages, analyzing the input specifications and most importantly combining the information gained, it is possible to achieve very powerful results. Yet one has to wonder how much more is possible and what limitations might be able to get lifted later on.

So in this section I want to comment on the analysis techniques used in this work and also on future possibilities.

5.1 Handling Restrictions

The goal of this work was to define language subsets which can always be analyzed to give desired static results, like preconditions for methods. This lead to various restrictions that limit the expressiveness of the base languages Relax NG or XPath 2.0.

There are, however, many possibilities to mix structural schemata and XPath expressions in rules and algorithms, which use the complete powerful standards, with those languages defined in this work.

A Relax pattern, for example, can in principal contain recursive patterns, ambiguous name classes or infinite name classes, as long as they are not constrained with context rules. An even then, if they are used in context rules only, but never manipulated, it would also be possible to work with such schemata.

The same goes for context rules which use way too powerful check expressions, where static analysis cannot create any preconditions from. It would be possible to allow such checks anyway and mix the two approaches of checking preconditions and checking postconditions or even do intermediate checks.

This has the severe downside of giving up on only manipulating documents when its safe, so backups have to be kept to allow rollbacks. This essentially nullifies some of the main advantages of this work, but in turn allows to check much more powerful conditions, while getting guarantees for the common ones.

Another example of balancing features with analysis possibilities are ambiguous, non-selectable nodes. The current language subsets allow to add such nodes, but they do not allow to manipulate them later. The only possible way to deal with them is to delete all nodes of the node class at once.

I decided to allow such nodes, as they can be used in various ways, for example to store intermediate data which is later transformed into another node. It is therefore possible to assemble pieces of information, constrain them and then
combine them to the final node and removing the pieces. These nodes can also never be referenced directly from a context rule instance, so it is only possible to define rules which combine them all together using aggregate functions.

The feature is very restricted, yet useful and does not break static analysis. These are the interesting classes of features and they show much potential for enhancement. One idea is to allow batch operations in algorithms, which would allow to manipulate non-selectable ambiguous nodes by using expressions which change in the current context. Static analysis would know that all nodes are manipulated and it might be possible to compensate the changes to generate preconditions.

Being able to compensate addition, removal and update of nodes is one of the key features of XPath CX for static analysis, yet there are easy ways to increase the supported subset towards XPath 2.0. The reason only very few functions are supported is that their effect on nodes has to be compensated. If their was an easy way to remove nodes from sequences in XPath CX, functions like \texttt{min} or \texttt{max} could very easily be supported.

But there is simply no way to remove single nodes of a node class from references to the complete node class. So to compensate for the removal of one value in a \texttt{max} function, it would be necessary to somehow calculate the maximum of the rest of the values using an XPath 2.0 expression and insert a dynamic check including it. This would mean the compensation is no simple rewriting on XPath CX expressions anymore.

But what if the maximum does not include any node sequences, if it includes only statically unique references or atomic values? In this case we can either compensate for a change or there is no compensation needed at all. The consequence is that most XPath 2.0 functions and operators can easily be supported for atomic values and \texttt{Node}.

The major point here is that once a document reference is embedded in a \texttt{Node}, \texttt{Sum} or \texttt{Count}, the effects of operations can be compensated and it doesn’t matter at all where these constructors are embedded themselves. So there is no reason to forbid the \texttt{min} or \texttt{max} function on two or more sums of nodes, for example.

Note that while XPath CX is not capable of removing nodes from sequences, which can be done in XPath 2.0, it is impossible to add nodes to sequences, which do not yet exist in the document. This means that even if XPath CX is enhanced to support removal, the concept of compensations is still necessary to deal with new nodes.

5.2 Approximations

Another interesting field are static analysis algorithms which only approximate information. Such algorithms can often work on huge subsets of the standardized
languages, but only return inaccurate information, which might not be precise enough.

One interesting member of this class of algorithms is based on node classes. Using these classes only, it is possible to statically guarantee that certain operation blocks do not affect certain rules. It is very easily possible to over approximate the node classes which are referenced by a rule, which is even defined on virtually the full XPath 2.0 language.

This is done by walking over the XPath expression and abstractly interpreting it, using an environment as shown in this work. All path expressions result in a set of node classes, where the approximation just assumes that all node of this class are referenced. Filter expressions are interpreted in the context of the outer node class and result in more node classes the expression touches.

Axis steps are always followed by node tests and these often dramatically reduce the number of node classes which are left as result of a path. In such an over approximation it is even possible to support axis like preceding or following, as information about ordering is still present in pattern positions.

The same approximation can now be done for operation blocks, by calculating all possibly altered node classes. This is done by over approximating the context expressions using a similar algorithm as for context rules. This time, however, filter expressions and also test expressions in if statements, can be ignored altogether, as the nodes referenced there will not be altered, they are just used to define the precise context of the operations.

It is now statically known, that a specific operation block can never violate a specific rule, if the two sets of touched and possibly manipulated nodes are disjoint. There simply cannot be a single node which is altered and appears in the rule. The analysis works amazingly well and often excludes huge classes of rules, which need not be checked anymore.

The obvious advantage of such an approximation is that it works for a huge XPath 2.0 subset and also supports recursive patterns and intersecting name classes. As normal for an approximation, it can fail terribly for certain classes of context rules, whose analysis result is just the complete set of node classes, so they are necessary to be checked for any operation block.

The static analysis done in this work also covers the same analysis results, as it also identifies rules which cannot be violated at all, by creating a precondition which only includes the prototype check, which is still unmodified and therefore discarded. The big difference is that this analysis is 100% precise and also yields exactly those instances that need to be checked before executing the block, together with a modified check that compensates all coming actions.

Approximation algorithms can therefore be a viable support for developers, yet they are not capable of always returning useful information. Still, this is an interesting class of algorithms, which can lead to the development of “best practice” methods, for which they are known to work very well.
5 Variations and Future Work

5.3 Dynamic Decision Trees

A lot of restrictions in this work were introduced to keep the generated preconditions statically definite. This includes, for example, that parameter values need to be different and in general that filter values have to be unknown labeled values. To support foreign keys, such a restriction has to be lifted, but this will mean that it is statically unknown if two filtered path expressions reference the same dynamic node.

This can only be covered by runtime checks, which leads to two different sets of preconditions depending on their outcome. The result can be arbitrarily deep decision trees, whose leaves contain the correct preconditions for a concrete input document and parameters. First and foremost, such trees are much more difficult to handle for static analysis, but they do not automatically lead to a longer or more complex runtime of the precondition phase.

So by going this way and supporting much more ambiguities which cannot be statically solved, the prototype languages defined in this work can be widened up to support many more interesting classes of context rules and algorithms, while still being able to work completely on preconditions, without the need to check anything after operations are done.

5.4 Document Interactions

When XML documents are used to store and transmit data, they not always stay monolithic or get manipulated in total only. Even the small example shown in Section 4.8 might suggest that items can be created elsewhere and are later inserted in the inventory or that many inventories are in use, which are all synced in a meta inventory.

This can either mean that parts of documents can be pulled out of their context or that some, potentially totally different, documents can be combined into one. In either case there needs to be a way to split or combine CXLDs and a mechanism to check constraints on both splitting documents into smaller parts and recombining them later.

Context rules, which have their context in a split off fragment might be able to get checked without the context, but as soon as they have references in the upper context, they need to be deferred and checked later. The same goes for rules in the main document, which have references into the split off fragments. These constraints either always have to hold, so they need to be checked on removal and recombination, or they can be violated as long the fragment is not present.

Relax NG supports various forms of modular or dependant schemata, so it will consequently be possible to define such split off points in Relax CX schemas with very few additions. Context Rules themselves are independent of each
other anyway and their context trivially gives away the parts of the document they belong to. As it is furthermore possible to statically analyze all references to other node classes, it can also be decided if a check has to be deferred or can stay with a fragment or the main document.

Another class of document interaction which will be inevitable is that algorithms manipulating one document want to access information in another. This can be done by allowing to explicitly name several input documents and give them to the method via special parameters. There might even be no implicit current document anymore, so path expressions need to be specified using the parameters or the root function on the parameters, to make their context explicit.

If their is no implicit document associated to methods, it can be possible to either specify which parameters are mutable or to allow manipulations on all input documents. This work supplies enough infrastructure to handle such situations, as it is no problem to maintain multiple pattern position structures, node classes of different specifications and consequently document references.

Different position structures can be differentiated quite easily by storing some kind of name of the corresponding CXLD in the root pattern position, which represents the document node. The same then has to be done in some way for multiple input documents of the same CXLD. These documents need to be disambiguated pretty much in the same way as it is necessary for ordinary nodes, which can be, for example, primary keys in the top level element.

If this primary key is also stored in the document node pattern position, all references into documents are unambiguous and the normal analysis methods can be used. Preconditions will then contain references to various input documents and in general will be necessary to be generated for all rules of each input document.

Another form of interaction is the transfer of complete fragments of a document to another. This cannot be done using the current restrictive algorithm language, but will be desirable to do so. One possibility to handle such situations is to allow only the transfer of fragments which are constrained with their own CXLD, which is included in both parent CXLDs. This would mean that local correctness is guaranteed and only those rules need to be checked which connect the data.

Another approach can be to analyze both CXLDs and allow only transfers which are statically known to maintain the structure in the target document and again generate preconditions for the transfer. In this case it would also be very beneficial to include other technologies which optimize the preconditions, using the context rules which held on the transferred fragment.
5 Variations and Future Work
The first goal of this work was to precisely define languages that can be used to give Constrained XML Language Specifications, which define advanced classes of valid XML documents. As XML is a very broad standard that can be used for various reasons in many possible ways, I defined a data centric data model inspired by James Clark. On top of this and besides commonly accepted structural restrictions, it should be possible to constrain the data itself which is stored in such documents and guarantee its integrity with additional rules.

To achieve this goal I first defined the Relax CX and Relax CX’ subsets of Relax NG, where the former is the desired language target and the latter a necessary restriction to achieve first static analysis results. Both languages cover the structural conformity of documents and are accompanied by the Context Rules Language, which allows to specify integrity and domain-specific constraints using a rule based approach and in principal the full strength of XPath 2.0.

Again this language was further restricted to the use of XPath CX, a subset of XPath 2.0 I also defined in this work, to allow static analysis to work on rule checks. CRL on XCX, together with RCX’ now make up the prototype languages to give CXLD language definitions. Though being severely restricted, it is still possible to express a variety of classes of data, using interesting context rules.

Finally, I also defined a very small local update language for CXLD constrained documents. The language allows basic remove, append and update operation with some rudimentary control structures and convenient macro definitions. The language is also defined on full XPath 2.0 but needs to be restricted to using XPath CX to allow static analysis to gather the necessary information.

The second and final goal of this work was to give analysis techniques and algorithms to work on the newly defined languages, to achieve powerful static results. It should be possible to reduce dynamic checking to the absolutely necessary level and furthermore get rid of all runtime checks necessary after manipulation of the input document started.

It should therefore be possible to keep initially valid XML documents valid after doing operations on them, without checking any or even all rules on the result document. A concise set of necessary preconditions should guarantee the flawless execution of a method whenever possible.

To achieve this goal I presented sophisticated analysis techniques and algorithms on Relax CX pattern definitions. Using these results, I then defined how abstract interpretation can be used on XPath CX expressions, to determine
all reference into an input document which are used to both manipulate the
document and retrieve data from it.

To facilitate analysis, I defined how each of the four used languages is stat-
ically analyzed to retrieve the relevant information and how to express that
information in the terms also defined in this work. Using this information I
finally showed how preconditions for each context rule can be generated and
what instances of rules need to be checked at all.

The presented static analysis works for all CXLDs and methods given and
discovers many classes of specification errors which are impossible to detect with
common XML technologies. Furthermore it is also always possible to generate
preconditions for error-free specifications, so no fallback to expensive runtime
checks is necessary.

Finally, after giving an example CXLD and method and showing first analysis
results, I showed that many more interesting analysis techniques and problems
can be engaged in this area, but that there are always trade offs to be made
between supported features and quality of analysis results.

This work approached the problem by trying to achieve strong static results,
while still keeping the involved languages interesting and usable. This modelling
process of the right subsets of common ISO and W3C standards is even possible
to continue and allow more features, while still maintaining the completeness
for precondition generation.

Last but not least all languages defined in this work and most of the analysis
methods shown are implemented in a set of Haskell modules which are sup-
plied on the electronic medium. The crucial results shown for a small example
specification can be reproduced using the example system, which also makes it
possible to define own CXLD and method specifications and get analysis results
for them.
Bibliography


[10] Patrick Michel. Static guarantees for algorithms on constrained XML documents, December 2006. [http://softech.informatik.uni-kl.de/twiki/bin/view/Homepage/PublikationsDetail?id=100](http://softech.informatik.uni-kl.de/twiki/bin/view/Homepage/PublikationsDetail?id=100).


Bibliography


