Data Binding for Schemata with Integrity Constraints and Atomic Procedures

---

A Generative Approach for Object-Oriented Languages

Thomas Fischer

August 30, 2012

Supervisors:
Prof. Dr. Arnd Poetzsch-Heffter
Dipl.-Inf. Patrick Michel

Software Technology Group
Department of Computer Science
University of Kaiserslautern

Abstract

While XML has become the de facto standard to represent structured data, common XML schema languages are quite restricted in their definition capability. Structural constraints are widely used throughout all schema languages, some languages also support global reference constraints, yet integrity constraints based on local references or values and arithmetic comparisons are uncommon. Data binding techniques, connecting XML with programming languages in a more typesafe way than SAX or DOM, exist for most commonly used schema languages. These techniques are only capable of handling structural constraints. This thesis proposes and implements a binding between schemata supporting more data-centric constraints and class-based object oriented languages. The binding generates class files from a schema file, defines a translation between objects and documents, and checks adherence to integrity constraints of documents and manipulations performed on the objects. The binding puts focus on correctness of constraint checks and usability of the generated library, using several heuristics to assist the user rather than supporting unorthodox schema definitions.

Zusammenfassung

Obwohl XML de-facto Standard zur Darstellung strukturierter Datentypen ist, sind die Spezifikationsmöglichkeiten mit gebräuchlichen XML Schemasprachen beschränkt. Strukturelle Beschränkungen sind verbreitet und einige Sprachen unterstützen globale Referenzen, doch Integritätsbedingungen, welche auf Werten basieren, oder gar arithmetischen Vergleichen zwischen diesen Werten, sind ungebräuchlich. Datenbindungstechniken, welche für häufiger verwendete Schemasprachen verfügbar sind, stellen eine typsicherere und intuitivere Zugriffsmethode von Programmiersprachen auf XML Dokumente als SAX oder DOM dar. Diese Techniken überprüfen nur strukturelle Beschränkungen. Diese Arbeit präsentiert und implementiert eine Datenbindung zwischen Schemata, welche derartige, datenorientierte Beschränkungen unterstützen, und klassenbasierten, objektorientierten Sprachen. Zur Datenbindung zählt die Generierung von Klassen, einer definierten Übersetzung zwischen Objekten und XML Dokumenten, sowie Kontrolle der spezifizierten Beschränkungen von Dokumenten, aus einer Schemadatei. Dabei liegt der Fokus der Arbeit mehr auf Korrektheit der Integritätskontrollen und Benutzbarkeit der generierten Bibliotheken durch Verwendung verschiedener Heuristiken als auf Unterstützung ausgefallener Schemata.

Contents

1 Introduction **5**
- 1.1 XCend . . . 6
- 1.2 STAT System . . . 7
- 1.3 Katja . . . 8

2 Overview **9**
- 2.1 Process . . . 9
- 2.2 Abstract Syntax . . . 11
  - 2.2.1 Schema . . . 11
  - 2.2.2 Assertions . . . 12
  - 2.2.3 Procedures . . . 15
- 2.3 Aspect Implementation . . . 16

3 Structure **18**
- 3.1 Name Collisions . . . 19
- 3.2 Nodes . . . 20
- 3.3 Patterns . . . 22
- 3.4 Keys and Compatibility . . . 26
- 3.5 Isomorphic Attributes . . . 29
- 3.6 Collection Interface . . . 30
- 3.7 Marhsaling . . . 33
- 3.8 Translation Example . . . 34

4 Assertions **36**
- 4.1 Formulas . . . 37
  - 4.1.1 Ternary Logic . . . 37
  - 4.1.2 Conjunctive Normal Form . . . 37
  - 4.1.3 Free Variables . . . 38
  - 4.1.4 Literals . . . 40
- 4.2 Terms . . . 41
  - 4.2.1 Path Expressions . . . 41
  - 4.2.2 Value Expressions . . . 43
- 4.3 Translation Example . . . 45

5 Procedures **47**
5.1 Program Variables 47
5.2 Preconditions 49
5.3 Statements 49
5.4 Convenience Methods 51
5.5 Insert Procedures 53
5.6 Concurrency 54

6 Case Study **55**

7 Related Work **59**

8 Conclusion **62**
8.1 Future Work 64
8.1.1 Generation of a Web-Based Frontend 64
8.1.2 Extensions to the Source Language 64
8.1.3 Further Heuristic Improvements 64

Chapter 1

Introduction

XML has become a de facto standard to represent structured data and is often used as interchange format for data in distributed information systems. To restrict the format of the data, schema languages are used. XML documents are considered to be *valid* with respect to a given schema, if they adhere to the constraints defined within the schema. Commonly used schema languages like DTD [15], XML Schema [13, 14], or Relax NG [4] focus on structural constraints, like the existence of a specific child element.

Compared to data constraints possible in relational databases, these structural constraints are not sufficient for data restriction. Databases are based around primary keys used to address elements within a table and foreign keys which point to elements within another table. While there exists support for global uniqueness and references to such elements, local uniqueness is not supported by schema languages.

Simple constraints on values are possible in most schema languages (Relax NG for example offers definition of data types with restricting attributes and enums), but comparison of two values stored within the document is not possible using only these structural constraints let alone arithmetic operations on such values. A typical example for such constraint would be the sum of weights of all stored elements not exceeding an overall capacity which may be stored in the document as well or be a fixed value hard-coded into the constraint.

Specification of integrity constraints as the above within XML schemata is desirable and allows the definition of more restrictive data formats. Schematron [5] provides an extension to the traditional, structural schema languages. It is a rule-based technique which uses XPATH expressions for specification and thereby provides a huge extension for specification of XML schemata. However, since it is not intended as stand-alone language, but as addition to an existing schema, tool support lacks functionality.

To work with the data structures defined by these schemata in programming languages, so-called data binding techniques are applied. In an object-oriented environment, these techniques usually supply a set of classes in the target language which represent the schema definitions, or allow definition of a mapping between elements within the XML schema and user-defined classes. Code is subsequently generated to map between the two representations of data. When reading from XML documents matching the schema, objects are instantiated. This process is also called *unmarshaling*, *marshaling* being the inverse transfor-

mation from objects back to XML documents. Note, that the expression "data binding" is by no means restricted to XML, but rather describes the general mapping between data and programming languages. Still, XML data binding is the most common application of this term. Data binding in the XML domain offers a more intuitive and typesafe access method than low-level parsers like SAX and DOM.

This thesis focuses on a data binding between XML-like languages and Java. Between these two languages, a magnitude of different data binding tools is available. Open source binding tools like JAXB [11] or Castor [3] exist as well as commercial ones, for example the Liquid XML Data Binder [6]. For the most part, these techniques work on the widespread DTD and XSD schema languages, but binding techniques for more uncommon schema languages exist as well. For Relax NG, a data binding tool called Relaxer [2] is available. For Schematron assertions, no data binding tools seem to exist, only validators in every conceivable programming language.

The core contribution of this thesis is the design of a data binding between schemata that support mentioned constraints and Java. In more detail, this includes the generation of Java classes out of schemata at design time and a facility for translation between Java objects and XML documents. The binding should adhere to defined integrity constraints, i.e. inhibit manipulations violating these constraints, and be capable of checking validity of XML documents during unmarshaling. This concrete binding is based on the schema language provided by the XCend technology, which supports the desired constraints.

In the following parts of this chapter, used technologies are briefly introduced. Chapter 2 gives a more detailed overview of the envisioned translation process and concepts of the used schema language. Chapter 3 describes translation of structural constraints as well as the generated interface for read access, which already contains several constraint-based heuristics. The translation and evaluation of non-structural constraints themselves is treated in Chapter 4. Manipulating access is channeled through user-specified procedures, introduced in Chapter 5. A small case study in Chapter 6 describes some short examples how users can work with the binding. Chapter 7 evaluates selected binding tools and compares their handling of non-structural constraints with the proposed binding. Finally, Chapter 8 concludes and gives an outlook on possible extension of the presented binding.

1.1 XCend

The XCend technology describes an XML schema language, which does not only specify structural but also integrity constraints. This includes, for example, further restrictions on primitive data types (for example String enumerations), aggregate types (the sum of all `weight` elements), value comparisons (said sum is smaller than the `capacity`, and reference constraints (if X exist, Y also has to exist). While most of these restrictions are widespread in database systems, they are rather uncommon in XML schema languages.

The schema language of XCend combines a pattern-based approach similar to abbreviated Relax NG [4] with elements of rule-based approaches. While patterns are used to describe structural constraints, rules define additional integrity constraints. These rules are directly embedded into the patterns, which

allows implicit deduction of the context and a more abbreviated notation. A more complete introduction to the schema language is given in [9], though the presented syntax contains some outdated elements. Furthermore, a complex example specified with the XCend schema language is introduced in Section 1.2. XCend generates an invariant out of all specified constraints which has to hold in order for documents to be *valid* with respect to a XCend schema.

In contrast to Schematron and other schema languages, integrity constraints are checked incrementally for each change of an XML document, meaning that only those constraints that are relevant for the performed change are checked. This is achieved by generating *weakest preconditions* out of the schema invariant and a manipulating *procedure*. If such a precondition holds and the document is valid before, the document resulting from the manipulation is also guaranteed to be valid. Procedures can contain several atomic manipulations and the invariant might not hold in the intermediate states of the procedure execution. Still, procedures are intended to be used for primitive manipulations and can be composed to more complex methods.

Generation of the invariant and the weakest preconditions, as well as large parts of the simplification process, are formally verified using Isabelle/HOL [10]. Therefore, invariant and precondition generation, as well as several proven simplifications, will be referred to as the XCend *theory* in the following chapters.

For specification of integrity constraints, XCend utilizes the concept of paths. A document is represented by a finite set of path/value pairs. This path concept also implies, that the order of elements or attributes is not allowed to carry any information, since no such order exists on paths. Read access on such a document does not yield single values, but multisets of values. Paths not occurring in the document evaluate to the empty set. For all paths that occur in the document, but have no assigned value, the *unit* value is returned. The concepts of paths and multisets are explained in detail in [8] and [9].

The XCend technology, especially the theory for precondition generation and simplification, has been developed by Patrick Michel. The design of the schema language and abstract syntax used as input has been extended to fit the purpose of this thesis.

1.2 STAT System

The *SofTech Achievement Tracking System* (STATS) is a mid-sized web application using the XCend technology[1] augmented with a manually written Java data binding and web-based user interface. It shows, that XCend is powerful enough to describe practically useful schemata. Fragments of STATS are used as a continuous example throughout this work to highlight aspects of the translation.

It has been developed by the Software Technology Group at the University of Kaiserslautern and is used to manage students attending courses and their participation and results in exercises and exams.

[1] The detailed schema description of the STAT System can be reviewed at the XCend website https://xcend.de/stats/schema

1.3 Katja

The *Kaiserslautern attribution system for Java* (Katja) tool is also developed by the Software Technology Group Kaiserslautern. It is capable of "generating immutable term-sorted data types together which a rich library out of concise specifications." Katja is used to specify the abstract syntaxes relevant for this translation.

The code generator of the prototype implementation of the XCend data binding is generally inspired by Katja's translation process, described in [7], and uses a modified version of Katja's Java backend for code generation. More details about the implementation of the translation process and influences of Katja are described in Section 2.3.

Chapter 2

Overview

This chapter provides a more detailed overview of the binding in general as well as the used schema language. First, the envisioned translation process and generated artifacts are introduced. Afterwards, the abstract syntax of the XCend language is explained and further restrictions, not expressed by the syntax alone, are documented.

2.1 Process

The envisioned translation process is depicted by Figure 2.1. First of all, a Java library representing the abstract syntax of XCend is generated. This abstract syntax represent the input of the binding generator. A first abstract syntax tree is generated out of a XCend schema definition using the XCend frontend. Subsequently, a schema invariant is generated out of the constraints expressed by the schema. This invariant and procedure definitions are used to derive weakest preconditions. Schema constraints, the invariant and the preconditions are all subject to simplification. Precondition generation as well as simlpification are performed by code generated from the XCend theory itself, i.e., they are both formally verified. The schema invariant is not necessary for the binding generator and is only used for precondition generation and simplification.

The resulting AST is used as input for the XCend Binding Generator, which is the major contribution of this work. The output of this translation process is the desired data binding. This includes

- Java classes matching the XCend schema capable of holding data matching said schema
- methods mapping between the XML and Java representation of date
- a rich library for access and modification as defined within procedures

The binding can be used to easily access data from XML documents in form of Java objects, modify these objects with the specified procedures, and marshal them back to XML. During all these operations, the generated objects are guaranteed to adhere to all specified constraints. Creation of objects violating these constraints is not possible.

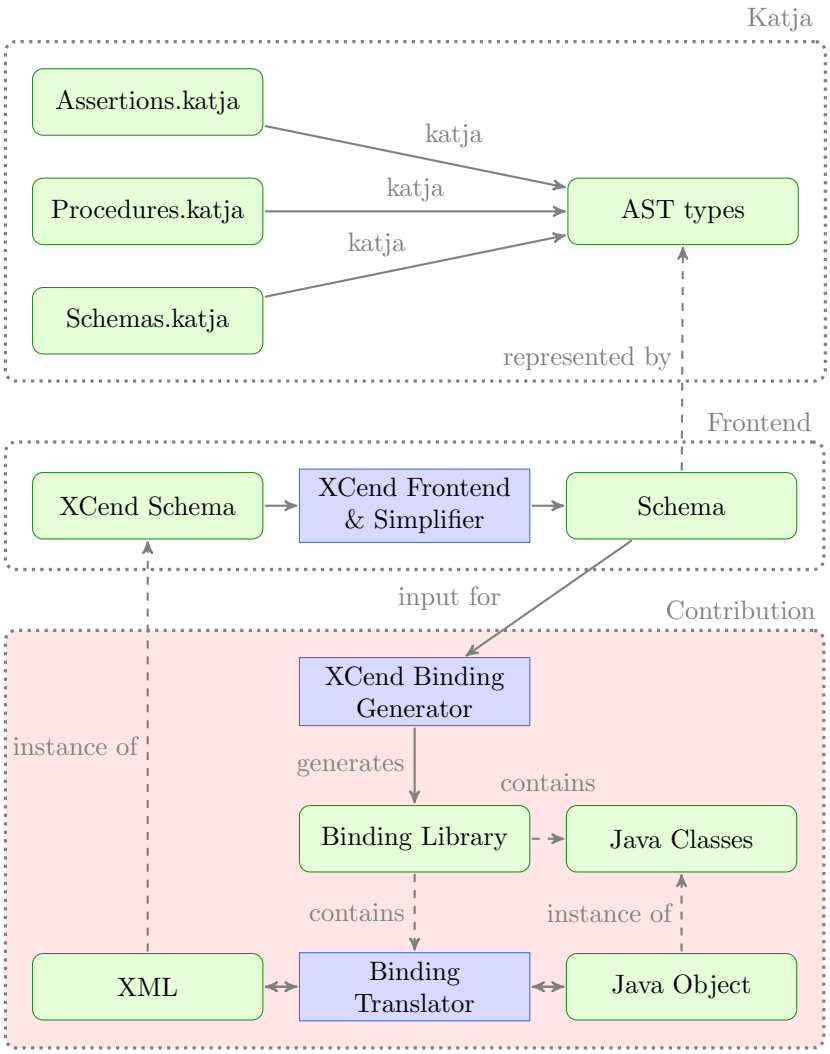

Figure 2.1: The overall translation process of the XCend data binding

2.2 Abstract Syntax

The input for the XCend binding generator is an abstract syntax tree defined in three Katja files described in the following.

This section is also used to introduce and define several terms concerning the semantics of XCend and non-structural requirements not expressed with the frontend syntax. These Katja terms are subject to simplification performed within the theory, which provides more strict guarantees on the input for the binding generator. These guarantees are explained here as well.

2.2.1 Schema

This part of the abstract syntax contains structural information about the XCend schema. This concerns nesting and repetition of elements and attributes. It therefore specifies all structural constraints of the schema in a similar way as commonly used XML schema languages. As already mentioned, these structural definitions are inspired by Relax NG compact syntax [4].

Furthermore, this part of the abstract syntax contains assertions for the defined nodes, which are used to specify integrity and value-based constraints, and a set of procedures. Assertions and procedures are refined in other Katja files described further below.

The translation of these structural constraints is explored in Chapter 3.

```
Schema ( Elem elem, Procs procedures)

Node = Elem ( Label name, Group content, Constraints cons )
     | Attr ( Label name, Simple content, Constraints cons )

Group       * Pattern
Constraints * Assertion
```

First of all, a schema has to consist of an element and a list of procedures. Schemata consisting only of a single attribute are not allowed. Elements and attributes are very similar and only vary in their content. Both have a name and can contain several embedded assertions. In the case of an element the content is a group, which is a list of patterns (described below). The attribute on the other hand contains only simple content. Elements and attributes are summarized as nodes.

```
Simple =
    Typed  ( Type type )
  | Enum   ( Type type, Values cases )
```

Simple content is defined as either a single type, or a typed enumeration of several values. Both `Type` and `Values` are defined in the imported assertion file, but are rather intuitive. So far, the only types supported are integers, strings, key values and the singleton value unit.

Keys are used as identifiers do differentiate repeated elements. The key sort of Katja contains a single string as value. Since keys are a complex and very important concept, both for XCend and the binding, they are explained in detail in Section 3.4. Note, that key typed enumerations are not allowed.

The unit value is used to model unassigned values. Considering this abstract syntax, this includes all read accesses using a path to an element, since elements cannot be assigned any value. Also, non-initialized attributes may yield unit

values. A unit typed enumeration is not allowed, since only one value exists for this type and an enumeration of several distinct values is not possible.

The content of elements is more complex and consists of several patterns.

```
Pattern =
    Fixed
  | Choice

Fixed =
    Single ( Node node )
  | Option ( Node node )
  | Kleene ( Element elem )

Choice      * Element
```

Four kinds of patterns exist. A `Single` pattern indicates exactly one node. So the child node always exists within the parent element and can be either another element or an attribute. The `Option` pattern allows the child node to be left out as well. `Kleene` describes a repeated element. Note, that attributes are not allowed to be repeated. Repeated elements are implicitly augmented with a special attribute `key`, which represents the locally unique key of the element used for selection. Last, the `Choice` pattern contains a list of possible child elements, one of which has to be contained within the parent element. Attributes are also not allowed here.

The design of this specification and therefore the XCend schema language requires explicit naming of all defined types. The names of elements nested into each other constitute the paths mentioned in the introduction. The keys of repeated elements are part of these paths as well. Since paths have to be unique, this also requires different elements or attributes nested in the same element to have different names. This greatly reduces naming problems during translation and also enhances understandability of the generated binding, since types can be named the same as their corresponding schema elements and few types without direct representation in the schema have to be generated.

2.2.2 Assertions

The assertion part of the abstract syntax is used, both by schemata and procedures. It defines assertions for nodes or procedures. Note, that not all combinations of this abstract syntax definition are semantically reasonable and further restrictions are made during translation. For the most part, these are explained in the following paragraphs and Section 3.1. Translation of assertions is described in Chapter 4.

```
Relation =
    Equal ( ValueExp left , ValueExp right )
  | Less  ( ValueExp left , ValueExp right )

Literal = Lit ( Bool positive , Relation rel )

Disjunction * Literal
Conjunction * Disjunction

Assertion ( Conjunction conj )
```

An assertion is declared in CNF (conjunctive normal form), meaning it is a conjunction of disjunctions of literals. Single literals compare two value expressions under a relation, which is either equality or less than. These top-level value expressions always have to yield single values and in case of the `Less` relation have to yield integer values.

Relations are evaluated using a ternary logic, which is explained in detail in Section 4.1.1.

```
ValueExp =
    Const ( Value val )
  | Variable

  | RKey  ( Label label )
  | Read  ( PathExp pth )

  | Scal  ( ValueExp left , ValueExp right )
  | VPlus ( ValueExp left , ValueExp right )

  | Aggregate

Variable =
    FVar  ( Name name )
  | PVar  ( Name name )

Aggregate =
    Sum   ( ValueExp arg )
  | Size  ( ValueExp arg )
  | Count ( ValueExp arg , ValueExp val )
  | Tally ( ValueExp arg , Type type )
```

Generally, value expressions represent multisets of values. While the theory works only using these multisets, the binding can exploit schema information to deduce additional details about typing and cardinality of the returned multiset.

The most simple value expression is the `constant`, which defines a single, typed value. As described in the schema syntax, the available types are strings, integers, keys and the unit value.

Variables can either be *free* or *program variables*. Free variables are implicitly universally quantified and exclusively used for key values. Their application is explained in Section 4.1.3. Program variables are only used in procedures. Their value and type are either provided by the environment, i.e., they are parameters of a procedure, or defined within the procedure itself (see Section 5.1).

`RKey` models access to the key of a repeated element. If the contained label is the empty string, the key of the current element is selected, otherwise the key of an ancestor with the given name. This element can only be interpreted for constraints embedded in a node, since a context is required for deduction of such a key. Furthermore, this expression is not part of the XCend theory, and is reduced for invariant and precondition generation.

`Read` expressions are the most complicated value expressions to translate, yet, at the same time, most interesting for the binding. A read expression yields the value stored in a document for a given path expression. Since the XCend theory allows values of different types to be stored for a single node, the type of such a read expression cannot be statically determined in the theory. However, the binding can exploit additional information provided by the schema part of the AST described earlier. Considering this schema information, the values of all nodes are homogeneously typed and the type of a value stored for a

specific path, and consequently the type of a read expression, can be statically determined. A read of a path without a mapping yields the empty set.

`Scal` calculates the scalar multiplication of two given value expressions left and right. Therefore, the left value has to be a single value, while the right side can be an arbitrary multiset. The XCend theory then filters this multiset down to only integer values and applies the scalar multiplication to only those. Consequently, a scalar multiplication with a multiset without any integer values will result in the empty set. If the left side is not a single integer value, the evaluation of the scalar multiplication will fail. Implications of this are described in detail in Section 4.1.1.

`VPlus` merges two multisets of values together. However, according to simplifications done on the XCend theory, it can only occur inside a `sum` aggregate. Therefore, it can be interpreted as the addition of all contained integer values.

The `vplus` is the only operation merging two multisets together. These multisets might have different types, but the wrapping `sum` aggregate guarantees the resulting value to be a single integer. Since the read expression, as described above, can also be statically typed, all value expressions are guaranteed to be homogeneously typed after simplification, even though the XCend theory is generally untyped.

Aggregates take a value expression as parameter and always yield a single Integer value. The `sum` aggregate calculates the sum of all integer values contained in its parameter. Non-integer values are simply ignored. The `sum` of the empty set or a set containing no integer values is 0. `Size` returns the number of values in the contained value expression. `Count` counts the occurrences of a single value val in the argument, `tally` counts how often values of a given type occur in the argument.

Path expressions point to nodes of a document. The abstract syntax for path expressions has been expanded compared to the theory, where only `Root` and `Step` are defined. Since the XCend binding also has the information from the schema syntax, further constructs can be used.

```
PathExp =
    Root   ()
  | Dot    ()
  | Parent( Label name )
  | Step

Step =
    Path   ( PathExp par, Label label, ValueExp keys )
  | Kind   ( PathExp par, Label label )
```

The `root` naturally references the root declared in the schema syntax. Assertions embedded within nodes can use additional constructs. The `dot` refers to the current node, the assertion is embedded in. The `parent` reference points to the first ancestor of the surrounding node with the given name. Translation will fail if these path expressions are used outside an embedded constraint or an ancestor with the specified name does not exist.

`Path` steps select all children of a node with the given label, if their key is contained in the given key set. For non-repeated elements, this key has to be the *null key*. *Kind* steps work in a similar way but do not require a key set. These steps basically represent a simplification on paths described in [9]. A kind step corresponds to a path step using all conceivable keys. Most of these accesses will

result in a unit value, since the path is not contained within the read document. Since these values will be filtered by surrounding expressions (i.e. read or an operation), this can be reduced to only those keys actually contained within the document under this path. For kind steps on non-repeated elements, this only leaves the null key.

Paths to nodes not occurring in the schema result in an exception beforehand. Usage of such paths is considered to be an error in the specification rather than intended read of an empty set. This is different from the theory, where paths cannot be evaluated statically due to the missing schema information and the empty set is returned.

2.2.3 Procedures

Procedures describe a way to manipulate a term. XCend generates a precondition for each procedure out of the schema invariant. If this precondition and the schema invariant hold before procedure execution, the invariant will also hold afterwards and the procedure can be safely executed. The translation of procedures is covered in Chapter 5.

Note that the preconditions, though naturally required as input for the binding generator, are generated by XCend and do not have to be written by the user. Still, the frontend does offer definition of additional constraints on procedures that cannot be generated automatically, for example constraints modeling access rights.

```
Procedure ( Name name, Parameters params, Statements body,
                Assertion pre )

Parameter ( Name name, Type type )

Procs      * Procedure
Parameters * Parameter
```

A procedure consists of a name, several parameters, a body and the mentioned precondition in form of an assertion. Parameters are typed and have a name. Since procedures describe modifying access, they do not need a return value. The procedure body consists of a list of statements.

```
Statement =
    Ite   ( Literal cond, Statements sthen, Statements selse )
  | Asgn  ( Name name, ValueExp exp )
  | Operation

Statements * Statement
```

`Ite` describes a conditional statement. If the given condition evaluates to true, the `then` statements are executed, otherwise the `else` statements. `Asgn` assigns a value to a program variable, which can be addressed in later statements of the procedure body. Both statements are basically for user convenience and could be achieved through other means. Furthermore, a statement can be one of the following operations.

```
Operation =
    Insert ( PathExp pth, Label label, ValueExp key, ValueExp val )
  | Update ( PathExp pth, ValueExp val )
  | Delete ( PathExp pth )
```

These three operations directly modify an XML document. Each is supplemented with a path expression, which specifies the context of the modification. `Delete` removes the node and all its descendants from the document. `Update` sets the value of the node to the given value expression. `Insert` adds a child node with the given label, key attribute and value. All value expressions in these operations are required to yield single values and path expressions are required to point to a single node.

2.3 Aspect Implementation

Inspired by the design of the Katja system, the implementation of the XCend data binding uses generation aspects for code generation (see Table 2.1). A single generation aspect contains every action that has to be taken to achieve a given aspect of the generated code. Hence, the generator is split by features rather than by generation phases, which allows easy addition of new or modification of existing functionality. Still, complete independence of these aspects is not possible, since the generated code is strongly interdependent. For example a change of the naming of classes in the basic aspect would also result in necessary changes for types of attributes and return types or parameter types of methods working with these classes in the component aspect.

The code generation backend uses a slightly modified version of the Java model and unparser of Katja's Java generation backend. Further details about the generation backend can be found in the description of Katja in [7].

basic aspects	Basic Aspect	- generates classes, interfaces, and enums for elements and nests them as in specification
	Name Aspect	- sets class names and modifiers
	Type Aspect	- handles subtype relations of choice subtypes
	Construction Aspect	- creates constructors for classes and enums
	Key Aspect	- creates classes and interfaces for used key types and sets compatibility between keys
	Component Aspect	- creates attributes and basic access methods
utility aspects	Collection Aspect	- creates collection classes with all their attributes and methods - generates methods for multiset selection of child elements
	Equality Aspect	- provides *equalsStructure* method for generated classes
constraint aspects	Parse Aspect	- creates methods for marshaling and unmarshaling concrete terms to and from XML
	Assertion Aspect	- creates methods for checking adherence to assertions defined within the schema
	Procedure Aspect	- generates static procedures from the schema in binding class - binds procedures in generated classes if applicable

Table 2.1: A list of generation aspects used by the generation backend

Chapter 3

Structural Constraints

After consideration of all input files and the translation process in general, the constructs defined within XCend now can be related to similar constructs in Java. First of all, the structural constraints of the XCend schema description have to be modeled.

Structural restrictions of XML documents are covered by the the first described Katja file. They are declared with a pattern-based approach using nodes, which represent XML elements or attributes, and patterns to describe their relations, cardinality, and nesting.

When comparing XML schemata and documents to the object oriented world, it seems intuitive to compare schemata with classes and XML documents matching said schemata with objects. Like in XML, objects do adhere to their specifying class and provide all the declared fields. Following these similarities, the general idea of the XCend binding is the generation of a class for every element occurring in the schema and a field for each attribute.

Mapping from a structural type system like XML to a nominal type system like Java, generally spawns naming collisions. While in XML, two elements with the same name may exist in the same element and be still uniquely identified by their order or structure, this is not possible in Java. Flattening of the XML structure to classes on the same hierarchy level would easily lead to conflicts between generated classes. In most schema languages, patterns don't even need a name at all. These problems usually require extensive (re)naming, making the resulting binding less intuitive. Section 3.1 describes, how the XCend binding treats such name conflicts between user-defined elements and generated types.

The translation of nodes and class generation is described in more detail in Section 3.2. The following Section 3.3 explains fields and methods for read access generated for each pattern contained within these nodes. Information about the structure of the schema is sufficient for these methods. Still, constraints are already analyzed to provide more advanced functionality, for example key compatibility introduced in Section 3.4 or selection of elements by isomorphic names described in Section 3.5. Section 3.6 covers how the binding handles the multiset semantic of XCend.

Generated classes are wrapped inside a *binding class*, which provides access to several utility functions working on the complete schema, for example marshaling and unmarshaling (see Section 3.7). Finally, Section 3.8 gives a more complete translation example covering all features introduced in this chapter.

3.1 Name Collisions

To provide unambiguous paths, the XCend schema language does not allow to nest nodes with the same name in the same parent element. Most patterns are implicitly named by their contained element, the only exception being the choice pattern, which contains several different elements (see Section 3.3 for details). Still, nodes defined within different elements may have the same name and flattening the structure would result in conflicts in Java. To prevent those conflicts, the generated Java classes are nested in the same way as their respective XML elements. Each element in XCend will define a new namespace in Java and nodes with the same name but different parents can no longer lead to conflicts. This nesting approach is also applied by JAXB, a common Java binding tool for XSD schemata [11]. If the schema is well-designed and no such duplicate names exist, static imports can be used to remove necessity of full qualification of each type. The class corresponding to the root element of schemata will further be addressed as *root class*. It should be mentioned, that the nesting depth of Java classes in Java 6 is only limited by the length of the class name, which is qualified using all outer classes. Since qualification with the names of parent elements would also be the most intuitive way to rename ambiguously named elements in the schema, using class nesting is not more restrictive than using a flat class hierarchy.

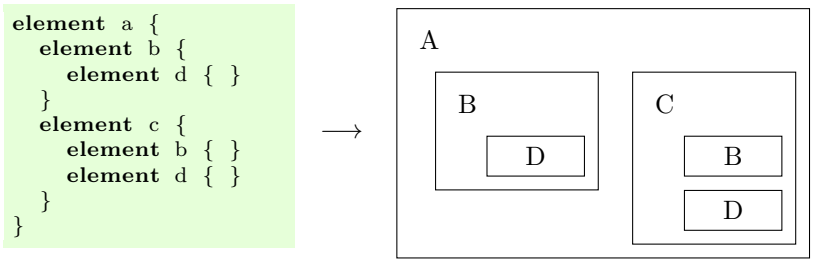

Figure 3.1: Different name collisions prevented by the nesting of classes

This nesting approach does avoid most conflicts resulting from user-defined classes. Figure 3.1 shows several element collisions that are avoided by the nesting of generated classes. Two elements with name b and two elements with name d are introduced by the XCend schema fragment. Generating those into a flat class hierarchy would require renaming of both classes using the parent element names for qualification. The proposed nesting solves these problems by nesting the classes of the same name in different parents, using the same hierarchy as XCend.

Not all conflicts can be solved by nesting. XCend allows nodes to be nested inside elements with the same name. The path to such an element is still unambiguous in XCend, for example the path `/a/b/a`, depicted in Figure 3.2. In Java, this nesting of such classes is not allowed, since a nested class is not allowed to hide an outer class.

Collisions with generated attributes, methods or classes not appearing in the schema are possible as well.

Furthermore, it is prudent to follow naming conventions of Java. These include type identifiers starting with an upper case letter, attribute, method

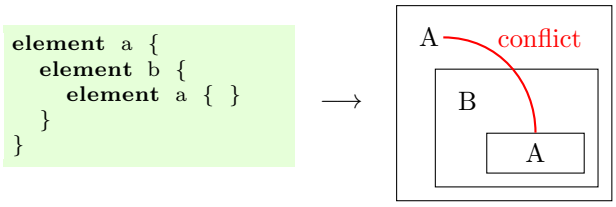

Figure 3.2: A remaining conflict between outer and nested class

and parameter identifiers to start with a lower case letter and generally usage of medial capitals. Following these conventions in a case-sensitive type system may result in additional collisions. Still, the benefits for usability are considered to outweigh these restrictions on the schema.

These remaining collisions of names, either directly from the schema, from naming conventions, or from names for generated methods, fields, or introduced helper classes, are not solved by the generator and simply lead to errors. Collisions with Java keywords also immediately stop the translation. Cryptic renaming to avoid name collisions would not be beneficial for usability and readability and therefore defeat the purpose of following Java naming conventions in the first place.

3.2 Nodes

The two general types of nodes to be translated are elements and attributes. While both do have names and may contain assertions, elements contain a list of patterns while attributes only contain `simple` content.

Elements Elements are translated as classes in Java. A class generated for a specific element is called *element class*. The nesting of elements is represented by nesting of generated element classes to avoid name collisions. Using the same nesting hierarchy makes it easier to relate the generated Java types with their respective schema elements. No deviation from this general rule is necessary.

For constraint translation, a reference to the parent element is required. Such a reference is added into each element class and set during object instantiation.

Attributes Attributes on the other hand do not contain other nodes, but only `simple` content. This either means a general typed value, or an enumeration, restricting the value space to a given set. XCend currently supports the four basic types integer, string, key and unit. Integer and string attributes are similar to their counterparts in other programming languages and are rather intuitive to handle. The XCend integer type covers the complete value space of integer and is therefore mapped to the Java type `BigInteger`. Key values are used to reference repeated elements. They are described in detail further below in Section 3.4. Unit values are not intended for attribute use at all. They rather describe a node, which has no value assigned. Furthermore, a unit attribute (or even enum) would make no sense, since all unit attributes always would have the same value. Therefore, they are not supported by this binding.

In theory access of an attribute would only yield the surrounding node and require an explicit read to retain the value. The binding removes this step and directly returns the content of the attribute. Non-existing attribute nodes are encoded using the value *null*. This results in the attribute node and the attribute value to be merged on the Java side. At first glance this might be a problem, since a state where the attribute node exists, but no value is assigned cannot be distinguished from a non-existing node. However, such a state violates the schema invariant and can only be reached as intermediate state of a procedure execution.

Typed Values For typed values of type integer or string, introduction of a new type is not necessary. These types are already provided by the Java API and can directly be used. This deviates from the general translation pattern of nodes, but simplifies user access to these attributes.

Key attributes provide a more complex problem. Since they reference repeated elements, a first idea would be translating them not explicitly as keys, but as direct reference to the repeated element. However, such key attributes can reference not only a single repeated element, but several different repeated elements, more specifically objects of different classes using the same key. Consequently, a simple field reference to an object is insufficient. Instead, a new key type is introduced for each key attribute, which stores the value of the key and can be used as parameter for access methods. The translation of keys is described in more detail in Section 3.4.

Enumerations They value space of an enumeration attribute is further restricted by listing a set of allowed values. This requires knowledge of possible values at specification of the structural constraints. Since keys are only intended as internal identifier, comparable to object identifiers, and should not be used to convey information, their values cannot be statically determined and key enumerations are not possible.

Before the translation of an enumeration, duplicate entries are eliminated and the user is informed of these cases. The same applies for enumerations consisting of only a single value (after duplicate elimination) or no value at all.

Enumerations can be translated in several ways. The most primitive translation would be similar to the translation of typed values, augmented with a check after each assignment to guarantee, that only the explicitly declared values are used. While this provides an easy translation, it seems rather unhandy, since erroneous assignments are only detected during runtime. Also, this construct allows no switches over return values of the enum type but requires again comparison to all declared values, which would require user knowledge about possible values.

Java supports enums itself since version 5.0. This construct can be used instead, providing the user with static guarantees and switches over all possible values of the enum. However, Java enums are not typed like enums in XCend. They can be seen as classes with an independent constructor for each possible value. This also means, that a simple Java enum does not contain any values, especially no `BigInteger` value, but encodes the value using the name. However, enums can be used like classes and therefore can specify further fields and methods. To realize the behavior of XCend enums with Java enums, it is

necessary to introduce an attribute to the enum which holds the actual value described. The instances of the enum are then identified using generic names, e.g., "value_", supplemented with running number. The actual value is annotated using JavaDoc, so the user can still distinguish these instances.

While being functional, preventing erroneous assignments statically, and supporting switches, this solution lacks usability as well, since users have to read the JavaDoc annotated value for each generic enum instance in order to find a specific instance required. Heuristics are possible to make this more user-friendly by changing the generated name of enum instances. For Integer enums, the running number in generation of the instance name can be replaced with the actual name. Since duplicate values are eliminated, it is guaranteed, that each number and therefore each generated name will only occur once and no conflicts arise. String enums are more restricted, but also leave room for improvements. If no conflicts with other instance identifiers exist and if it is a valid Java identifier, the upper case string of the actual value can be used. For reasonable examples, these heuristics should provide user-friendly enum implementations.

A different approach could be the usage of inheritance. This requires a translation of each possible value as a subtype of a common supertype which represents the type of the enum. While being functional, this solution generates a lot of additional classes and is less intuitive for the user of the binding. In addition, the problems with names of enum values as described above still has to be solved in a similar way.

Using Java enums augmented with naming heuristics for enum instances seems to be the most intuitive and practical approach and was chosen for this binding.

3.3 Patterns

Patterns are used to specify the cardinality of child elements, for example, that an element is optional or required. Child elements are generally translated as package local fields for the parent element, but the type of the field and accessing methods differ according to the type of pattern. Since the fields themselves are package local, they may be used for faster access by other generated classes, but not by the user himself, who is restricted to the generated access methods. The names of the generated field and all selection methods generally depend on the name of the node contained within the pattern. As a consequence, the selection chain for an element in Java is very similar to XCend path expressions, and relation between the two is intuitive. In the following, the characteristics of the patterns and the resulting translations are discussed.

Single and Option Patterns Single and optional patterns are the most simple ones to translate and are very similar. A single pattern denotes a child node that always exists. A node wrapped within an optional pattern may be left out, i.e., it may exist at most once. Consequently, it is sufficient to introduce a single field holding an object of the type of the contained node.

Non-existence of the contained element is denoted by assigning null to the field. A field resulting from a single pattern with a null value would violate the schema invariant, but may occur in intermediate states during procedure

execution. For all states exposed to the user, a field generated from a singleton pattern can never be null.

A simple *"get-method"* (*selection method*) is generated to retain the child node. For optional pattern the child node may not exist and the generated selection method may throw an exception. These patterns are augmented with an additional boolean method (*"has-method"*), which checks if the generated field is null and the child element does not exist.

Kleene Pattern Kleene patterns represent a repeated element, i.e., a set of elements of the same type. Repeated elements are identified and distinguished using a locally unique *key*, and no two elements with the same key can exist within the same parent. The element contained in the list is supplemented with a field for the key described in the and a method for retaining this key.

The usage of keys to identify elements in a set is identical to the Java map, which also models a set of indexed elements. Kleene patterns are translated as maps from the key type to the type of the contained element. As stated above, generation of key types is more complex and therefore handled an individual section below.

In contrast to `single` and `optional` patterns, a simple selection method is insufficient for kleene patterns in order to select a specific element. The pattern requires a key to distinguish individual elements. Thus, the basic selection method generated for kleene patterns takes a key value as parameter. Similar to optional patterns, this selection may fail with an exception, if no element with the given key exists. A method for existence checks is introduced as well to provide safer access, which also is supplemented with a key parameter.

```
element stats {
  element accounts {
    element account * { }
  }
}
```

The XCend schema above shows a small example of a kleene pattern definition. The star indicates, that the element `account` is repeated. This example will be translated into the code fragment below. The generation of key classes has not been covered so far, but will be explained in Section 3.4

```
public static class Stats {
  public static class Accounts {

    Map<Account.Key, Account> accountMap;

    public Account account(Account.Key key) throws
        NoSuchElementException {  ... }
    public boolean hasAccount(Account.Key key) { ... }

    public static class Account {
      Key key;
      public class Key { ... }
    }
  }
```

Note, that these methods are also provided by the Java `Map` interface, namely `get` and `containsKey`. The functionality of the methods generated for the binding can therefore easily be realized by delegating the calls to these methods.

Choice Pattern The choice pattern contains several elements one of which has to appear at this position in documents. The choice is *exclusive*, meaning that only one of these elements may appear.

Choice patterns also translate into a single field, typed using a *choice supertype*, which is extended by all possible choices. This translation models the XOR semantic of the choice pattern in Java. Translation as a single field guarantees, that only one of the choices can be stored at the same time. The supertype ensures that only elements from inside the choice pattern can be stored. Using a distinct field for each possible choice does not only require additional checks to ensure that only one element for the choice is stored at the same time, but does also contribute to namespace pollution, since each choice field needs to be named.

However, the choice supertype has disadvantages as well. Naming of this type is problematic, since choice patterns are not explicitly named. Consequently, the supertype also contributes to namespace pollution. There are three basic options for naming the choice supertype: using a generic name, using the name of the parent node or the names of child nodes. All three require some modification of the name to avoid collisions with parents, children or other choices within the same element.

For this binding, the second approach has been chosen. The name of the parent element is augmented with a choice suffix. This solution is less generic than the using a generic choice identifier and does not result in unreasonably long class names for larger choices. If several choices are contained within the same element, a number is added to the choice name. This solves name conflicts between choice supertypes with the same parent.

Using numbers to qualify choices spawns considerations about code stability[1]. First of all, adding a second choice to an existing single one changes the name of this choice, which may promote the wish to use numbers regardless of the choice count. Yet, how these numbers are used for qualification still has not been explained. There are two possibilities to be considered: either using the position within the parent element or only the choice order. Both solutions suffer from instability. While the first one is robust against changes of already declared patterns, it remains unstable when new patterns are inserted, even without the patterns being choices. The second solution on the other hand is unstable considering changes of existing patterns into choices or insertion of new choices. For the prototype the first solution has been chosen, yet both are considered to be of equal stability. In any way, the stability loss from omitting a number for single choices is considered negligible in comparison and doing so results in better readability and therefore usability.

A selection method is introduced for each possible subtype, which internally checks the type of the stored value, casts it if possible, and throws an exception otherwise. This translation allows to easily map paths from XCend into Java syntax, since these selection methods match the element names defined in the schema. While usage of a single selection method returning the choice supertype is possible, such a method exposes the supertype, requires explicit user casts of

[1] Code stability means that changes of the source file do not influence unchanged parts. For example, changing a node `A` should not influence the translation of any other node `B`. The generated code for the other node is "stable". This is important, when the generated libraries are embedded in other code fragments. If the schema file is modified, only the code using the modified parts should have to be changed.

the returned supertype, and does lead to different selection steps from XCend in order to select the same element. This translation hides the choice supertype from the user completely. It is not possible for the user to obtain an actual instance of the choice supertype. This allows the supertype to be realized as an abstract class and emphasizes, that intricate name generation for choices supertypes or explicit naming of choices in the schema is unnecessary for a user-friendly binding.

In Java, subtyping is realized with inheritance. The translation generates a *choice supertype*, i.e., an abstract class without any element-specific methods, which is extended by the classes generated from elements inside the choice pattern. Using abstract classes instead of interfaces for the choice supertype allows hiding of utility methods that are generated into the supertype and are then dynamically bound to each subtype, e.g., methods for (un)marshaling or checking assertions.

```
element entry {
  element book {
    attribute author      { string }
    attribute title       { string }
  }
| element cd {
    attribute artist      { string }
    attribute albumtitle  { string }
    element track * {
      attribute title     { string }
    }
  }
}
```

The XCend specification above shows a small usage example of choices. Below, the translation to Java is outlined. An abstract choice supertype `EntryChoice` is generated, which is extended by both subtypes. The attribute introduced for the choice is of this type. The generated methods however return one of the subtypes, but may fail if the object is of the wrong type for the selector method called.

```
public static class Entry {

  // attributes of the Entry class
  EntryChoice entryChoice;

  // methods of the Entry class
  public boolean hasBook() { return entryChoice instanceof Book; }
  public Book book() throws NoSuchElementException {
    if(hasBook) return (Book)entryChoice;
    else throw new NoSuchElementException("Book");
  }

  //analogue to the methods above
  public boolean hasCd() {... }
  public Cd cd() throws NoSuchElementException { ...}

  // nested classes of the Entry class
  public static abstract class EntryChoice { ... }
  public static class Book extends EntryChoice { ... }
  public static class Cd extends EntryChoice { ... }
}
```

3.4 Keys and Compatibility

One of the four basic types of XCend, and the most complex for the binding, is the key type. Keys are used to uniquely identify repeated elements, i.e., they are locally unique for elements with the same parent and name. In the XCend theory, keys are not refined any further leaving keys of repeated elements with different labels or parents interchangeable. Considering the STATS example, there are cases where this interchangeability of keys is desired. For example, the key of an examiner right in a user account should be usable as key for accessing the respective exam. However, there are more examples, where keys from different positions should not be used. For example, it would make no sense to use account keys to access exams.

So to prevent accidental misuse of a key at an unintended position, keys should generally not be exchangeable. The mentioned cases of desired interchangeability have to be annotated using assertions. To prevent general interchangeability of keys, an individual key class is created for each key occurring in the schema, i.e., repeated elements as well as key attributes. To realize interchangeability of selected keys, the concept of *key compatibility* is introduced, which describes which keys can be substituted for a given key.

Key compatibility is defined as an equivalence relation between key types, i.e., it is reflexive, symmetrical, and transitive.

A key type `A` is considered *compatible* to a key type `B` (and vice versa), if one of the following expressions occurs in any assertion within the document:

- `A` equals `B`
- `count` with `A` as parameter is called on a set of `B`
- inside a path expression, `A` is used as key where `B` was expected

This compatibility is realized using additional interfaces for each key class. The interface type is used whenever the key is used as parameter, the actual implementation type when the key is used as return value. The generated key class implements its own interface and the interfaces of all compatible keys. This allows key objects to be used whenever a compatible key is expected.

```
element stats {
  element account * {
    [ x != # ] // # marks the key of the current account element
    [ exists /stats/exercises/exercise/assistant[x] ]
    [ exists /stats/exams/exam/examiner[x] ]
  }
}
```

The fragment above shows an example using free variables. Since keys in the theory are not typed, free variables can take on values of different key types. These key types are all made compatible. If compatibility between a free variable and another key type is declared, this key type is compatible to all keys types, the free variable is used for. So looking at the example above, the key attribute of account and the keys of assistant and examiner are all compatible.

Key compatibility effectively allows usage of key attributes as parameters of selection methods of repeated elements on the one hand and easy assignment of the key values of such repeated elements to key attributes on the other hand. The following example depicts the effects on code working with the binding.

```
element stats {
  element account * {
    element examiner * { [exists /stats/exam[#] }
  }
  element exam * { [size ( /stats/accounts/examiner[#] ) > 0] }
}
```

Both assertions in the XCend fragment above state, that the examiner key is compatible with the exam key. Accordingly, an examiner key can be used, whenever an exam key is expected and vice versa. The XCend fragment is translated to the Java fragment below. Selection methods use the interface types, which are implemented by all compatible keys. Therefore, compatible keys can be used as input for these methods.

```
public static class Stats {
  ...
  public Exam exam(Exam.KeyIF key) throws NoSuchElementException {
      ... }

  public static class Account {
    ...
    public Examiner examiner(Examiner.KeyIF key) throws
        NoSuchElementException { ... }

    public static class Examiner {
      ...
      public static interface KeyIF { ... }
      public static class Key implements KeyIF, Stats.Exam.KeyIF {
          ... }

    }
  }
  public static class Exam {
    ...
    public static interface KeyIF { ... }
    public static class Key implements KeyIF, Stats.Account.
        Examiner.KeyIF { ... }

  }
}
```

The introduction of these interfaces on the same hierarchy level as their implementation has one significant drawback. Interfaces in Java are public, static, and have to be accessible at top-level. In order for the interfaces to be statically accessible at top-level, all outer classes of the interface have to be declared public and static as well. This basically requires all classes generated by the binding to be declared public and static. This implies, that elements are not translated as inner classes but as static nested classes. Instances of static nested classes do not implicitly refer an instance of the outer class, consequently, explicit storage of the parent reference is necessary. Alternatively, all key interfaces could be generated in the binding class. However, this would result in name collisions and require extensive renaming of all key interfaces. Namespace pollution due to the explicit parent reference is considered to be a negligible problem compared to renaming necessary due to key collision.

XCend keys are not supposed to carry information besides references to or from other repeated elements. To ensure this, key generation and management is handled by the binding itself and is not exposed. To this end, key constructors

are package local and each repeated element provides a static method, which generates a fresh, locally unique key. Note, that the prototype implementation generates globally unique keys. This internal key management also reveals why key enums being rejected by the binding. Since keys are created during runtime and their creation is not transparent to the user, it cannot be clear which keys may exist during specification of the source schema. Consequently, no meaningful key values can be provided for such an enum. If keys are managed internally and not exposed, the actual key values do not have to be strings as in the XML representation. Instead, integer values are used as key values. This increases efficiency of key comparison and generation of fresh keys.

Extension of the Key Concept A typical use case considering compatible keys is selecting an element with the compatible key of another element the user already has access to. The binding requires explicit selection of the key of such an element.

```
stats.exam(examiner.key())
```

In this small example, an examiner object has already been retrieved. To select the referenced exam, the key has to be selected explicitly. It would be easier to directly use the element itself as input for selection, i.e., removing the key step on the examiner object. This pattern occurs quite often in code working with the stats binding.

At first glance, this can easily be done by letting classes generated from repeated elements implement key interfaces or extend the contained key class. Since all classes are public and static, and therefore are all compiled into independent class files, this should not be a problem.

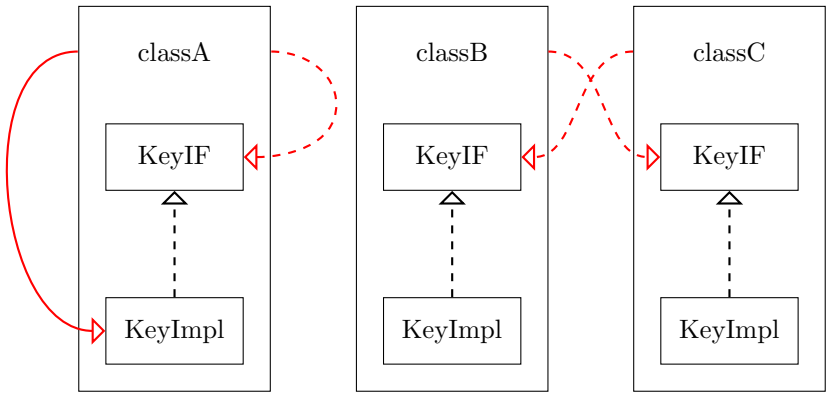

Figure 3.3: Class hierarchies forbidden in Java

However, Java does not allow class hierarchies such as depicted by Figure 3.3, not even for static classes. A class is neither allowed to implement nor extend classes or interfaces nested in itself. This also excludes cyclic constructs of classes implementing interfaces nested in the respective other class.

Flattening the type hierarchy and placing keys not in but beside their repeated element classes would resolve this problem, but lead to name collisions instead. For this binding, collision avoidance and shorter class names are deemed more important then the overhead resulting from this explicit key selection.

3.5 Isomorphic Name Selectors

With the binding described so far, users can select elements only with knowledge of their key. These keys are managed internally and not allowed to carry information other than references to other elements. This strict separation of keys from attributes, even if usable to identify an object, allows the attribute values to be changed without transitive effects on referencing objects.

Still, if a repeated element can be distinguished by a value other than the key, it should be possible to retrieve the element using this value. Considering STATS, a typical application would be selection of an account by its username or student id. For this purpose, the notion of *isomorphic* names is introduced. An child node is considered to be an isomorphic name iff

- it is a string attribute AND
- it is annotated with a *uniqueness constraint*,
  i.e., `count(., p) = 1`, where `p` is a path pointing to a multiset of nodes of the same type as the current node

Such an attribute is unique for one or several repeated elements. Consequently, a bijective mapping between these attributes and the key values of the repeated elements exists and the name can be used to select the key (or directly the element).

```
element accounts {
  element account * {
    attribute username {
      string [ count(., ..accounts/account/username) = 1 ]
    }
  }
}
```

In this simple, yet common, example the repeated account element contains an attribute `username`, which is unique over all `account` elements and therefore an isomorphic name to the account key. The binding should allow the user to select accounts with a username from the `accounts` element. This is a frequently occurring pattern and the XCend schema syntax provides an abbreviated notation (`element account * username`).

Yet there may be cases, where names are not just locally unique, i.e., stronger than the key of a single repeated element.

```
element library {
  element book * {
    element copy * {
      element uid ? {
        attribute id { string
          [ count(., ..library/book/copy/uid/id = 1 ]
        }
      }
    }
  }
}
```

Consider the above example of a library comprising several copies of possibly the same book. Still, each copy can be uniquely identified over all books and copies. The embedded constraint states this by restricting the count of a given

id occurring within the complete library to one. If the id is known, it is not necessary to know the book in order to select the referenced copy. In fact the opposite is the case: the copy with a specific id might not be selectable from the currently selected book object, since it is a copy of another book. Selection of elements using isomorphic names should be restricted to the outermost element, for which the name still is unique, in this case the library class.

Furthermore, it is unclear which element should be selected with this name. It seems natural to select an element which could not be addressed unambiguously without such a name selector. This would mean selection of repeated elements only, in this case, not the optional uid, but the wrapping copy. Since the id is unique for two repeated elements, copy and book, a selector for both can be provided. This also implies, that the unique name does not have to be located directly in a repeated element. In this case, the id is surrounded by the optional uid element, which has no influence on generated name selectors.

These name selectors are realized by additional methods in the class generated for the outermost repeated element, for which the name still is unique. Similar to selection using the key, execution of this method may fail if the provided name was not used.

```
public static class library {
  ...
  public Book bookById(String id) throws NoSuchElementException {
      ... }
  public boolean hasBookById(String id) { ... }

  public Copy copyById(String id) throws NoSuchElementException {
      ... }
  public boolean hasCopyById(String id) { ... }

   ...
}
```

The above code fragment shows which methods will be generated for the given specifications. In addition to the selection method, a "has-method" is generated as well, which checks if an element with the given name exists. This is identical to the methods generated for repeated patterns.

3.6 Collection Interface

The theory behind XCend is not based on single elements but instead considers multisets of elements. Read expressions may yield multiple elements, even the same element multiple times, or none at all. In the same way, `scal` expressions do yield multiset, if the right side is a collection. The `vplus`, even in its restricted form, still yields a multiset of integers.

XCend is based around the concept of paths, which can yield multisets of values. To support element access for the binding in a similar way, collections are introduced. The collections provided by Java are not sufficient for this purpose, since all selection steps performed on single elements should also be performable on a collection of said elements. Consequently an individual *collection class* is introduced for each element, which provides the same selection methods as the element class. This also implies collection classes being typed in the same way as their element classes, i.e., they only contain elements of a single type. Contrary

to the theory, this does not pose a problem, since simplification guarantees all multisets to be homogeneously typed. The collection interface does not only provide additional functionality and usability, but also simplifies assertion translation in Chapter 4.

When considering aggregates, multisets indeed make sense as parameters. Aggregates on single elements on the other hand can be statically simplified. Instead of realizing aggregates as static functions in a utility class, they are implemented in abstract collection superclasses. Four of these superclasses are introduced: a generic superclass and three more specific classes for collections of type integer, string and key. One of these abstract classes is extended by each collection class introduced.

While all aggregates would require filtering to a specific type in the theory of heterogeneous multisets, they can be easier calculated when considering typed collections. The `sum` aggregate on non-integer collections is always 0. The `count` aggregate always yields 0, if the types are incompatible, `tally` is either 0 or equal to the size of the collection. `VPlus` using a non-integer collection will add 0, since the surrounding `sum` would filter all non-integer values. A scalar multiplication using an integer as left parameter and a non-integer set as right parameter will return the empty set. These occurrences can be statically replaced. It suffices, to implement aggregates, `vplus` and `scal` only for the integer collection.

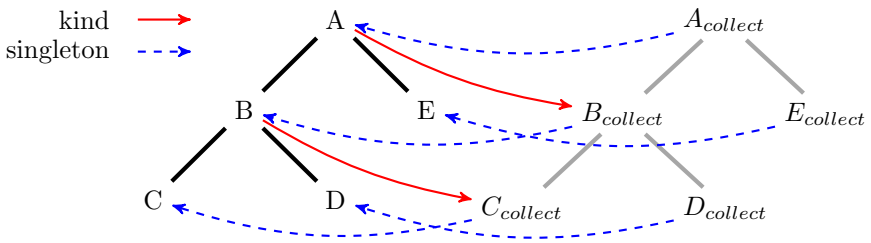

Figure 3.4: Collection classes generated for a set of singleton classes

The integration of the collection classes within the already generated classes and interaction between the two is depicted by Figure 3.4. Types B and C are Kleene types and therefore usually need a key for their respective selector methods. If the selector method is called without a key (a so-called *kind step*), a collection of B or C elements is returned instead. In return, if the collection contains only a single value, this can be retrieved with the special `singleton` method. The method fails if the collection contains no value or more than one.

Since now the generated code contains classes for collections of all classes, it especially contains one for each generated key class. Instead of using the collections provided by Java as return value for the `keys` method, these key collections can be used instead. One advantage over using classes of the Java API is access to aggregate methods. While most do not make a lot of sense on key collections, it makes translation of assertions easier (described in detail in Chapter 4) and maintains consistent usage of generated classes.

The advantage of using key collections over Java collections is more evident when considering the usage of key interfaces and key implementation types described in Section 3.4. The interface type is used for parameters, the imple-

mentation type for return values. When Java collections are used, the returned collections cannot be directly input into other methods, since Java collections are invariant. While this has good reasons, contravariance would be required to solve this without assigning each value to a new collection. For individual collection classes, the solution is analog to key classes themselves. An interface is introduced for each key collection, and all compatible collection interfaces are implemented. The interface type is used as parameter, the implementation type as return value. As a result, returned collection classes can directly be input into selection methods and the compatibility of keys is maintained throughout collections as well.

As described above, collection classes represent multisets of elements, meaning the same element can be contained multiple times. Due to this, the semantic of the parent reference is not clear for collection classes. Possible interpretations would be the multiset consisting of the parent of each element or a set containing each parent only once. Additionally, the user would expect round-tripping behavior, i.e., when calling parent first and then the selector method for the original collection, the user might expect to receive the original collection. Yet for both translations, simple examples can be constructed contradicting this behavior. Therefore, no parent reference is given for collection classes.

Filtering The selection methods described so far offer no refined selection of elements by their properties. The only possibility to select a subset of elements is to use the key or name for selection. To provide a more powerful method of selection, a *filter* method is introduced into the abstract superclass of all collections. This method returns the collection of only those elements, which fulfill a certain property. Properties for filtering can be defined by implementing a filter interface which is taken as parameter for the filter method.

```
AccountCollection accounts = stats.accounts().account();
AccountCollection students = accounts.filter(new Filter<Account>()
    {
  public boolean filter(Account e) {
    return e.hasStudent();
  }
});
```

The example above shows an application of this filtering mechanism. The collection of all accounts should be reduced to only student accounts, i.e., those with a student id. A new anonymous subtype of the filter interface is instantiated, which evaluates to true if the given account has a student element. This filter is applied to all account elements and a new account collection containing only those, for which the filter holds, is returned.

The overhead necessary for this is enormous in Java, since no high-order functions are supported. Closures, currently expected in Java 8, might be used to simplify the filtering mechanism. With closures, the filtering of the above fragment could be reduced to a line like the following.

```
AccountCollection students = accounts.filter(
  {e => e.hasStudent()});
```

Note, that this syntax is not necessarily final for Java, as Java 8.0 is still under development.

3.7 Marshaling and Unmarshaling

The Java classes generated from this process are capable of storing XML documents matching the XCend schema. To marshal concrete objects to XML and vice versa, two new methods are introduced into each element class. Implementation for both is rather straight-forward. Note, that abstract methods are also generated into choice supertypes, which allows dynamic binding of these methods to all subtypes. The visibility of these methods can generally be restricted to the package and only the binding class should provide public methods for (un)marshaling complete documents.

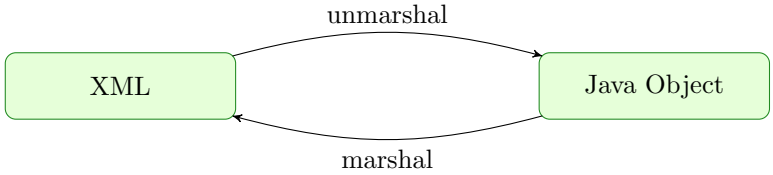

Figure 3.5: (Un)marshaling of XML documents to Java

Unmarshaling of an XML document not matching the schema naturally leads to an exception. Elements and attributes in the XCend specification have to be matched by XML elements and attributes using the same names, nesting hierarchy, and type restrictions. The value of a key attribute is represented by an integer. Required nodes have to occur in documents, optional nodes may be left out. A choice between elements requires exactly one of the specified elements to occur at the position of the pattern. Repeated elements may occur multiple times and have an additional attribute `key`. This attribute is required and stores the key value, which also is an integer. It can be referenced by the keys of other repeated elements or key attributes.

```
element stats {
  element account * name {
    element student {
      attribute id { int }
    }
  | element employee {
      attribute id { int }
    }
  }
}
```

⟶

```
<stats>
 <account name="john" key="1">
  <student studentId="123456" />
 </account>
 <account name="jane" key="2">
  <employee staffId="123456" />
 </account>
</stats>
```

Figure 3.6: XCend specification and a matching XML document

The code fragments in Figure 3.6 show an example specification using most functionality of the XCend schema language, which is matched by the XML document on the right. Note the additional key attribute in the repeated account element. A user-specified attribute with this name in a repeated element will lead to a name collision and is rejected by the generator.

After successfully unmarshaling an XML document, most structural constraints are automatically guaranteed by the type system. The last step of the parse method is checking adherence of the newly constructed object to all other

constraints, before returning it to the user. So even after successful unmarshaling, the binding might still encounter errors resulting from violated constraints. The constraint check is an expensive operation (depending on the specified constraints), and may not be necessary, if the XML document is already guaranteed to adhere to constraints, for example, if the document has been generated using the binding. Constraint check can be switched off using a flag in the unmarshaling method.

3.8 Translation Example

This section describes a translation example for a small XCend specification of accounts, which also can be found in the STAT system, using all functionalities described so far.

```
element accounts {
  element account * username {
    attribute lastName
    attribute firstName
    element student ? {
      attribute id {
        integer [ count(., ..accounts/account/student/id) = 1 ]
      }
    }
  }
}
```

The fragment above depicts a small XCend schema, which is a fragment of the STATS schema. The example describes a set of accounts, which all have a username, last and first name. They may also have a student id, which is stored in an optional element. The username is declared as unique name for account elements. A constraint embedded in the `id` attribute states, that this attribute is also unique for all accounts.

This is translated into an abstract syntax tree matching the syntax described in 2.2 and finally the following set of Java classes.

```
public static class Accounts {

  public boolean hasAccount( Account.KeyIF accountKey );
  public AccountCollection account( );
  public Account account( Account.KeyIF accountKey ) throws
      NoSuchElementException;
  public AccountCollection account( Account.KeyCollectionIF
      accountKeyCollection );
  public Account AccountbyUsername( String username ) throws
      NoSuchElementException;
  public Account AccountbyId( String id ) throws
      NoSuchElementException;

  static parse( XMLReader xmlr, Stats stats ) {
    ...
    Accounts accounts = new Accounts ( );
    Account.parse( xmlr, accounts);
    while(SchemaValidation.isOpeningTag(FxmlER.
        peekNextSignificantTag(), "account")) {
      accountSet.add(Account.parse(FxmlER, accounts));
      SchemaValidation.checkClosingTag(FxmlER.nextSignificantTag(),
          "account");
```

```
    }
    ...
  }
  unparse( XMLWriter xmlw ) {
    xmlw.writeStartElement("accounts");
    for( Account account : accountMap ) {
      account.unparse ( xmlw );
    }
    xmlw.writeEndElement();
  }
  check constraints ( ) {
    for( Account account : accountMap ) {
      account.checkConstraints ( );
    }
  }

  public static class Account {

    public Key key( );
    public String username( );
    public String lastName( );
    public String firstName( );
    public boolean hasStudent( );
    public Student student( ) throws NoSuchElementException;

    static parse( XMLReader xmlr, Accounts accounts ) { ... }
    unparse( XMLWriter wxmlw ) { ... }
    check constraints ( ) { ... }

    public static interface KeyIF extends KeyInterface { }
    public static class Key extends AbstractKey implements KeyIF {
        }

    public static class KeyCollectionIF {  }
    public static class KeyCollection extends AbstractKeyCollection
        <KeyIF, Key> implements KeyCollectionIF { ...  }
    public static class AdminCollection extends
        AbstractValueCollection<Admin>{ ... }
  }

  public static class AccountCollection extends
      AbstractValueCollection<Account>{
    public Account.KeyCollection key( );
    public Account.UsernameCollection username( );
    public Account.LastNameCollection lastName( );
    public Account.FirstNameCollection firstName( );
    public Account.StudentCollection student( );
  }
}
```

For readability, the fields and method bodies have been left out in this example translation. Code for the implementation of the methods is rather intuitive from the method signatures and the translation description above. For each child element, a field and access methods are created. Optional and repeated elements get additional methods. Collection classes generated contain the same selection methods, although `NoSuchElement` exceptions and methods checking existence of elements are removed. For such an access, an empty collection is returned. Additional library methods are introduced for (un)marshaling, checking non-structural constraints and selection using isomorphic names.

Chapter 4

Assertions

Assertions are used to express a non-structural property of an XML document. Typical examples are the number of repeated elements not exceeding a given limit or an element with the key defined in a key attribute existing at some other position in the schema. As described in Section 2.1 and Section 2.2, the definitions of the schema and its procedures both use assertions. In the schema, assertions appear embedded in nodes. Such embedded assertions describe non-structural constraints of the schema, which have to be matched by XML valid XML documents. In procedures, assertions are used to specify preconditions. The translation for both occurrences is very similar.

Embedded assertions implicitly define a context of ancestor elements. These ancestors can directly be accessed using dedicated value and path expressions. For precondition generation, all embedded assertions are merged to a single conjunction and this context is removed. Procedure preconditions do not define such a context. From procedures, only the root is directly reachable. Embedded constraints are implicitly guarded with an existence constraint for the surrounding node, i.e., they are only checked if the surrounding node occurs in a document.

Fulfillment of assertions cannot be ensured using only the type system of the programming language, but requires dynamic checks. These checks are expensive and should not be performed after each access. A valid document cannot become invalid just by reading data. When modifying the document, however, constraints have to be checked again. The translation described so far only covers read access, with the exception of unmarshaling in Section 3.7. There, an object representing the XML document is instantiated. Since in general it cannot be guaranteed, that the source document is valid with respect to the schema, constraints are checked before handing over the newly instantiated object to the user. If this can be guaranteed beforehand, for example if the XML document was generated using procedures provided within the binding, the check can be skipped as well. Structural constraints are automatically checked during parsing of the XML document. To check non-structural constraints, a special method `checkAssertions` is integrated into each generated class. This method also calls respective methods on child elements.

Manipulations of XML documents or the parsed object respectively do not require subsequent check of the schema invariant in XCend. This is possible, since manipulation is restricted through defined procedures, and prior check of

the precondition generated for each procedure guarantees, that the invariant will hold after procedure execution.

Embedded assertions can in turn affect the translation of the schema. Examples of this are the concept of key compatibility described in Section 3.4 or the notion of isomorphic names in Section 3.5.

4.1 Formulas

Assertions are logical formulas in conjunctive normal form. This section introduces the basic syntax and semantics for these formulas and explains how the semantic can be mapped to Java.

4.1.1 Ternary Logic

XCend formulas use a ternary logic, also known as *kleene logic*. The usual boolean values `TRUE` and `FALSE` are augmented with a third value called `BOTTOM` ($\bot$), which represents an unknown state. $\bot$ is used, whenever it is undefined, how the assertion should be evaluated. For example, comparing a single value with a set of values will result in $\bot$.

	$\wedge$			$\vee$		
	T	$\bot$	F	T	$\bot$	F
T	T	$\bot$	F	T	T	T
$\bot$	$\bot$	$\bot$	F	T	$\bot$	$\bot$
F	F	F	F	T	$\bot$	F

Table 4.1: Evaluation of conjunction and disjunction under ternary logic

The evaluation of $\bot$ in boolean formulas is equivalent to evaluation using an unknown value, which may be either `TRUE` or `FALSE`. When used within a conjunction, $\bot$ is dominant to `TRUE` but is suppressed by `FALSE`. For the disjunction, it is the other way around. The behavior of $\bot$ is shown in detail in Table 4.1.

In Java, this ternary logic is realized with an enum and with instances for each of the three values. Conjunction and disjunction are defined within the enum and work as described in Table 4.1.

4.1.2 Conjunctive Normal Form

Assertions are represented in CNF, i.e., each assertion is a conjunction of disjunctions of literals (cf. Section 2.2).

For the binding, explicit treatment of $\bot$ in disjunctions or conjunctions is not required. The interesting information is, if an assertion does or does not hold. If a literal evaluates to $\bot$, evaluation failed, which can only be result of an error raised by violation of an assumption, e.g., equality comparison with a non-singleton value or incompatible types. A false value on the other hand, marks a literal that could be evaluated, but yielded false. Both values imply, that the literal did not evaluate to true, and consequently, the invariant (or

precondition) does not hold. Bottom values in conjunctions and disjunctions can be interpreted in the same way as false values.

The evaluation of CNF formulas is done lazily. If one of the disjunctions in the conjunction does not hold, i.e., does not evaluate to `TRUE`, the complete assertion immediately fails. Error messages for the user can be augmented with the assertion or disjunction that failed, as well as the node it is embedded in. For the disjunction, it is the opposite. It suffices for a single literal to evaluate to `TRUE`.

The lazy evaluation of conjunctions is realized with the use of exceptions. If a single conjunct evaluates to `FALSE` or $\perp$, the complete assertion check fails immediately. Lazy evaluation of disjunctions is bit more intricate, since evaluation to `TRUE` does not lead to an exception. Instead, disjunctions are wrapped inside a `do/while` loop and occurrence of a true value breaks this loop. The loop is only used to implicitly create a break label and the condition is set to false.

Some literals can be statically evaluated by exploiting schema information which is not available for XCend. This primarily concerns the pattern `tally (parameter, type) = 1`, which is introduced by the simplifier to reduce other parts of the conjunction but cannot be removed afterwards. Since the type of a procedure parameter is stored in the schema, such literals can be statically evaluated and be used to evaluate surrounding disjunctions or even the complete conjunction. A disjunction statically evaluating to true can be left out completely, since it will not influence the overall result. At the same time, a disjunction statically evaluating to false or $\perp$ marks a specification error, since the overall conjunction can never be fulfilled. Single disjuncts evaluating to false can be dropped. The case of a complete disjunction statically evaluating to false is already caught during the attempt to statically evaluate the surrounding conjunction.

4.1.3 Free Variables

Free variables are implicitly universally-quantified over the domain of all keys. This means, assertions containing free variables are checked for each possible assignment of the variables, which results in an conjunction.

There are only finitely different assignments, for which the results can be distinguished. Remaining assignments can be tested symbolically with a single execution subsuming all equivalent assignments.

For free variables occurring in aggregates, a scalar multiplications or a `vplus`, the actual assignment of the variable does not matter. The `sum` of a free variable is always 0, the `size` of a free variable is always 1. Since free variables are always of type key, the `tally` aggregate can also be statically evaluated to 0 or 1 depending on the type parameter. A scalar expression with a key type parameter will evaluate to either $\perp$ or the empty set. Non-integer collections (or values) in `vplus` can be ignored due to the surrounding `sum` aggregate.

This leaves three positions where the assignment of a free variable actually matters: in a `path` step of a path expression, in a `count` aggregate, or directly in an `equal` relation.

To reduce the infinite set of possible keys for each variable, all *valid* and only some *invalid* keys are considered. A key `a` is considered valid for a free variable `x` iff one of the following statements holds with respect to the current assertion:

- ∃ path `p` with a step using `x` as key and `size(p[x→a]) > 0` OR
- ∃ `count(x,s)` and `count(a,s) > 0` (i.e., `a` ∈ `s`) OR
- ∃ `count(b,x)` and `count(b,a) > 0` (i.e., `b` = `a`)

While using different valid keys inside path expressions might yield different values, all invalid keys will yield the empty set. For count expressions, invalid key values will always yield 0. Consequently, only equality can be used to distinguish invalid keys, i.e., all invalid, equal key pairs and all invalid, but unequal key pairs are indistinguishable. For a single free variable, this reduces the values to be checked to all valid keys and a single invalid key, unequal to all valid ones. To reduce the number of checked assignments further, only the keys valid for the repeated elements, where free variables are used as key, are checked. So first all these repeated elements have to be determined. The valid key sets of these elements are then merged and some invalid keys are added for the symbolic check of all remaining invalid keys, for example the largest key occurring in the merged key plus one.

For more than a single free variable, the equality between invalid keys becomes important. A single invalid key is no longer sufficient, since it would always be the same key for all free variables, and the equivalence class of invalid, unequal keys would no longer be tested. To guarantee check of invalid but unequal keys for each free variable, a new invalid key, unequal to the invalid keys introduced for other free variables, has to be introduced. The invalid keys of all free variables handled before are also included, so equal invalid keys occur automatically. The new key can be easily constructed by using a different number than before. Consequently, the total number of invalid keys introduced matches the number of free variables.

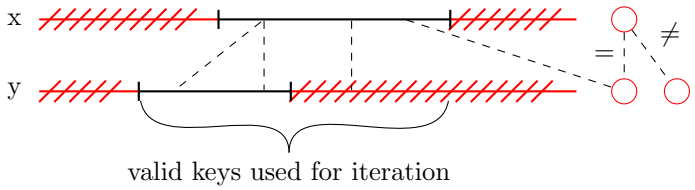

Figure 4.1: Assignments of free variables which are tested

Figure 4.1 shows the value space for an iteration with two free variables x and y. All distinguishable combinations of key assignments in a scenario of two variables are listed in Table 4.2, together with the steps required for key set generation. To test combinations of valid keys, it is sufficient to let each free variable iterate only over the keys valid for itself. If a combination between valid and invalid keys should be considered, this independent iteration is insufficient. To test each possible assignment of the same key to all free variables that is valid for one and invalid for another free variable, the valid key sets of all free variables have to be merged. Note that such a case does not necessarily occur, if the sets of valid keys are equal for all variables. For these cases a single invalid key has to be added in order to test the combination of a valid and an unequal invalid key. The same invalid key can be used to test the assignment of the same invalid key to all free variables. To test assignment of unequal invalid keys, `n` invalid keys are necessary, where `n` is the amount of free variables occurring

in the assertion. These assignments guarantee that all combinations with a distinguishable result are tested.

[x→a], [y→b]	a = b	a ≠ b
a and b valid	independent iteration	independent iteration
one valid, one invalid	merged key set	single invalid key
a and b invalid	single invalid key	n invalid keys

Table 4.2: Distinguishable combinations of keys for two free variables

In Java, this check is realized with loops. Each iteration represents another assignment of the free variables. Nesting of loops generated from multiple free variables guarantees, that every combination of values is tested. The value sets over which the iteration is performed are constructed as described above. All key sets where the free variable is used as a key are merged and invalid keys are added according to nesting depth of the free variable.

Since not all literals contain free variables, iteration over the complete conjunction would lead to unnecessary computation overhead. Instead, conjuncts without free variables are evaluated before free variable iteration, and the iteration is only performed for those conjuncts actually containing free variables. This could also be done in a more intricate fashion distinguishing not only between literals with and without free variables but between literals containing specific free variables. This could, for example, also move literals containing only the free variable x out of the assignment loop of variable y. The current prototype does not make this distinction.

4.1.4 Literals

Literals mark single logical values and compare two value expressions, which are evaluated with respect to an XML document. A Literal is composed of a relation, which is either the equality or less than relation, and a boolean value, stating if the relation should evaluate to true or false. Both together can be interpreted as $=$, $\neq$, $<$ or $\geq$. The left and right parameters of the relation have to represent a single value. In case of `Equal` all value types are allowed, in case of `Less` only integer values are permitted. All types can be determined statically, consequently the translation immediately fails if unpermitted types are encountered for the given relation, or the types used within the relation do not match.

Cardinality of the multisets cannot be determined statically, but only single values can be compared under the relations. The binding distinguishes statically between single values and collections modeling multisets. While single values simply can be input into literal comparisons, collections have to be dynamically cast to a single value. If such a cast fails due to the collection containing more than a single value or being empty, the literal evaluates to $\perp$. This can only be the case for `scal`, `vplus` or read expressions, since all other types are statically guaranteed to always yield single values.

4.2 Terms

Terms are the basic building blocks for formulas and procedures. They can be evaluated in the context to a specific XML document. Path expressions point to a multiset of nodes, possibly the empty set, if no elements exist at the path in the provided document. Value expressions can be evaluated to a (multiset of) value(s) of one of the four basic types of XCend.

4.2.1 Path Expressions

Path expressions point to a multiset of XCend nodes. Expressions pointing to a elements are called *element paths*, path expressions pointing to a attributes are *attribute paths*. The intuitive translation of path expressions is access of an object of a class generated for the node the expression points to. While this is indeed the way element paths are translated, such a class is not generated for attributes and the node and value are merged on the Java side, as stated in Section 3.2. This requires path expressions to be treated differently according to their context.

The only possible occurrences of path expressions are inside other path expressions, inside read expressions, or inside the body of a procedure. Path expressions nested inside other path expressions represent the parent fragment of the complete path. The current path expressions marks an additional *step* on the parent path. This parent path expression is not allowed to be an attribute path, since attributes mark leaves of the XML tree, and no further navigation is possible.

Other occurrences of path expressions need to be treated differently according to the type of nodes selected with the path expression. The distinction of path expressions inside read expressions are described in Section 4.2.2. Path expressions in procedures are covered in detail in Chapter 5.

Since the generator also has schema information, the existence of each path (except existence of specific key values) can be checked before introducing invalid access code in the generated binding. If a path does not coincide with a node in the schema, the path is considered to be *invalid* and the schema is rejected by the generator informing the user about the invalid path used. Path expressions are considered to always point to a multiset for the binding. This implies, that the path expression always returns a collection of elements or values. The components of the path expression can be translated as follows.

Dot, Parent, Root Path expressions are defined recursively and start with one out of three possible path expressions. Dot and parent mark paths relative to the context, the surrounding assertion is embedded in, and do not occur in the theory. Instead, those paths are replaced with absolute paths, that have quantified variables for each selection step of a repeated element. Since the binding generator has the context information necessary to evaluate relative paths, they are retained resulting in simpler and more efficient code.

The dot expression marks a reference to the current node, i.e. the node the assertion is embedded in. The translation of this expression is simply a reference to the current object, which is denoted by the keyword *this* in Java. The parent expression points to the first ancestor node with the name specified within the `parent`. Usage an empty string as name marks a reference to the actual

parent. This shows the requirement of the parent reference in element classes, which is set at object initialization. The necessary number of parent steps to be executed can be statically determined. If no ancestor with the specified name exists, the assertion and therefore the complete schema is rejected. The root expression points to an instance of the root class of the schema. It would be possible to store the root in a field within each class as well, but if considering the already introduced parent field this additional namespace pollution can be avoided. The number of parent steps required to reach the root node can again be statically calculated as well. This solution favors reduced memory consumption over runtime efficiency. In procedures, the class is an explicit parameter. Root expressions there are translated as reference to this parameter.

```
// context of the path expressions: /stats/exams/exam/tasks/task
// reference to current node (.)
this;
// reference to exam ancestor (..exam)
this.parent.parent;
// reference to root stats element (/)
this.parent.parent.parent.parent;
```

Note, that the root expression does not reference an instance of the root class, but the document root. These two elements are merged in the binding, since there exists an isomorphism between the two and the only step possible on the document root is selecting an instance of the root class, i.e., `/` has the same semantic as `/stats` in the STATS binding. All value expressions calculated on either an instance of the root class or the document root can be statically evaluated. The "root step" is still required for selection of child elements. Path expressions are checked for this step before translation.

Attributes are not translated to individual classes (cf. Chapter 3). Accordingly, the `this` reference does not point to the attribute, but an instance of the parent element class. Dot references embedded in attributes, require an explicit access of the field modeling the attribute in Java. For an ancestor reference, the number of necessary parent steps is reduced by one, i.e., the ancestor is referenced from the parent of the attribute.

```
// context of the path expressions: /stats/exams/exam/date
// reference to current node (.)
this.date;
// reference to exam ancestor (..exam)
this;
// reference to root stats element (/)
this.parent.parent;
```

Since a path expression is considered to always yield a collection, the node resulting from a root, dot or parent expression is explicitly cast to a collection. Subsequent steps of the path expression cannot reduce this collection back to a single value, resulting in the complete path expression to yield a collection.

Kind The kind step selects all child nodes with the identifier given as parameter. Due to introduction of the collection interface, this simply translates as a call of a selector method, which exists in both, the single class as well as the collection class generated for the parent of the kind step. This underlines the importance of the collection api for internal use, since without its existence the

iteration over all elements would have to be generated within code evaluating the path expression.

Path Path steps are more selective than kind steps and also use a set of key values for selection. For non-repeated elements only the empty key is valid, which is equivalent to the usage of a kind step for the binding. Such steps are translated in the same way as kind steps. Path steps over repeated elements may use other key values. Selector methods taking a single key or a collection of keys as input are generated for those nodes (cf. Section 3.2). The path expression is translated to a call to one of these methods (bound statically, depending on the type of the key expression), using the value expression given as key set as parameter. If the contained value expression does not yield a key type, the schema is rejected.

For paths occurring in embedded constraints, the collection returned by the *keys* value expression is guaranteed to be usable as input for the generated selector method, since key compatibility has been defined exactly that way (cf. Section 3.4 and 3.6). Paths occurring in procedures are checked beforehand for incompatible key values. These cases raise an exception (see Chapter 5).

4.2.2 Value Expressions

The type of a value collection can always be statically determined. Also, it can be statically determined if a value expression will yield a single value or a collection of values, even though the number of elements inside the collection is only known at runtime. Single values and collections are wrapped or unwrapped, if required by the context. These casts may dynamically fail, resulting in the complete literal evaluating to $\perp$. In case of unexpected types, the schema is rejected.

Constants and Variables Constants can simply be translated as the contained value. For variables, the surrounding code described in Section 4.1.3 guarantees, that a variable with this name has been declared and assigned beforehand. So the variable name can be used here.

Key read expressions These expressions do not occur in the theory but only as input for the binding generator and only in constraints embedded in nodes. They give access to a key used to identify a repeated ancestor element of the node, the expression is embedded in. As with parent and root references, the number of parent steps necessary to reach the specified ancestor node can be statically determined. Translation fails if either a parent with the given name does not exist or the addressed element is not repeated, i.e., always uses the empty key.

In the binding, such an expression is translated as access to the explicit key attribute of the respective ancestor object.

Read Expressions Read expressions access information stored within a given XML document using a path expression. XCend usually requires an explicit read expression to select the value stored for a given path expression.

In the binding, access to an attribute path automatically reads the contained value (cf. 4.2.1). Since it can be statically determined, if a path points to an attribute or an element, these cases can be distinguished here.

Access to an attribute path does not require additional action. The result from the path access already returns the stored values. A read expression on an element path would always return a collection of unit values. Such a collection is only meaningful as input for a size aggregate. Other combinations are rejected by the binding. Since the size can be directly calculated on the element collection, an explicit read is not required.

Consequently, read expressions are dropped completely and are only used to wrap path expressions in the AST.

Scal The scalar multiplication requires the left parameter to be a single integer value. In theory, the right side can be a single value or a multiset, but for the binding, it has to be a collection. Both sides are cast if necessary, resulting in `scal` dynamically evaluating to $\perp$ if such a cast fails. Since multisets are homogeneously typed in the binding, the scalar multiplication on non-integer values can be statically replaced with an empty integer multiset. Consequently, the right parameter is guaranteed to be an integer collection and it suffices to supplement the integer collection with a `scal` method, which handles multiplication with a single integer value and yields the multiplied collection. This method is called on the right side with the left side as parameter. Scalar multiplications are considered to always return an integer collection in this binding, though it might only hold a single or even no value, depending on the values contained in the right parameter.

A scalar multiplication using a non-integer value on the left side is statically evaluated to $\perp$. Note, that the XCend theory uses a scalar multiplication of the null key and 0 to represent $\perp$ itself.

VPlus According to simplification, `vplus` is restricted to only appear inside a `sum` aggregate (cf. Section 2.2). Non-integer values occurring in the parameters of `vplus` are filtered by this surrounding aggregate.

Since collections are homogeneously typed, a `vplus` operation using a non-integer collection as parameter is equivalent to adding 0 to the second parameter. Consequently, such a `vplus` can be seen as a specification error.

Similar to the `scal` expression, augmenting the integer collection with a method which adds either a single integer value or another integer collection, is sufficient for realizing `vplus`. This method is then called using the parameters of `vplus` as input. According to these considerations, `vplus` always returns an integer collection.

Aggregates Aggregates are defined in collection classes as well. However, methods for aggregates have to be added to the more general value collection, since aggregates can be used for all types of collections. Aggregates always return single integer values. The implementation of aggregates is done according to their definition in Section 2.2. The `tally` aggregate can be left out in the binding, since collections are always homogeneously typed and, `tally` can be statically replaced either with `size`, if the passed type parameter is equal to the type of the collection, or 0 otherwise.

4.3 Translation Example

In this section, the translation is explained with a small code example taken from STATS.

```
element grades {
  element grade * id {
    attribute value     { integer }
    attribute minPoints { integer }
  }  [ !exists ./grade[x]
    || !exists ./grade[y]
    || {x != y} -> ./grade[x]/value != ./grade[y]/value ]
}
```

This is the schema definition in XCend. The example consists of a `grades` element which contains several `grade` elements. These grades have a value and the minimal number of points necessary to attain the grade.

The only assertion for this small example is embedded in the `grades` element. It states, that for all keys x and y either there is no grade with that key, or, if x is unequal to y, the value of the grade with key x is also unequal to the value of the grade with key y. So basically, there exists an injective function from existing grade keys to grade values.

For the described abstract syntax and the XCend theory, existence constraints are translated into size constraints on the described path. Existence constraints are translated into a size greater than one, non-existence constraints in a size equal to one respectively.

Other parts of this example can be translated straight forward.

```
// setup key set for free variable iteration
AbstractKeyCollection<KeyInterface, AbstractKey> keySet = keySet(
    this.parent.parent.parent.singleton().exams().exam().grades().
    grade().key());

// iterate over free variables
for(KeyInterface y : keySet.keyUnion(new AbstractKeyCollection<
    KeyInterface, AbstractKey>(new HashMultiset<KeyInterface>(new
    AbstractKey(keySet.largestKey().add(BigInteger.valueOf(1)))){}))
    )) {
  for(KeyInterface x : keySet.keyUnion(new AbstractKeyCollection<
      KeyInterface, AbstractKey>(new HashMultiset<KeyInterface>(new
       AbstractKey(keySet.largestKey().add(BigInteger.valueOf(1)))
      {}))).keyUnion(new AbstractKeyCollection<KeyInterface,
      AbstractKey>(new HashMultiset<KeyInterface>(new AbstractKey(
      keySet.largestKey().add(BigInteger.valueOf(2))){}))){

    // check constraint: {ex ./grade[$x], ex ./grade[$y], ./grade[
       $x]/value[] = ./grade[$y]/value[]} $x = $y
    do {
      // evaluate single literal: 0 >= size(./grade[$x])
      if(de.xcend.util.Comparator.compare(GEQ, BigInteger.valueOf
          (0), BigInteger.valueOf(this.singleton().grade( new Stats
          .Exams.Exam.Grades.Grade.Key(x.keyValue()) ).size())) ==
          LogicalValue.TRUE) break;
      // evaluate single literal: 0 >= size(./grade[$y])
      if(de.xcend.util.Comparator.compare(GEQ, BigInteger.valueOf
          (0), BigInteger.valueOf(this.singleton().grade( new Stats
          .Exams.Exam.Grades.Grade.Key(y.keyValue()) ).size())) ==
          LogicalValue.TRUE) break;
      // evaluate single literal: $x = $y
```

```
      if(de.xcend.util.Comparator.compare(EQ, x, y) == LogicalValue
          .TRUE) break;
      // evaluate single literal: ./grade[$x]/value[] # ./grade[$y
          ]/value[]
      try {
        if(de.xcend.util.Comparator.compare(NEQ, this.singleton().
            grade( new Stats.Exams.Exam.Grades.Grade.Key(x.keyValue
            ()) ).value().singleton(), this.singleton().grade( new
            Stats.Exams.Exam.Grades.Grade.Key(y.keyValue()) ).value
            ().singleton()) == LogicalValue.TRUE) break;
      } catch ( NoSingletonException e) { }

      // if this point is reached, no disjunct evaluated to true
      throw new ConstraintViolationException("{ex ./grade[$x], ex
          ./grade[$y], ./grade[$x]/value[] = ./grade[$y]/value[]}
          $x = $y", "/stats/exams/exam[" + this.parent.key + "]/
          grades" );
    } while(false);
  }
}
```

Since the complete generated Java code is rather large, even for this small example, only the code checking the assertion is provided here. Note first, that for both free variables x and y a loop is created. Conjuncts containing free variables are evaluated inside this loop, which guarantees, that a value will be assigned to all free variables in the literal. These loops assign each valid key as well as some keys invalid for the free variables as stated in Section 4.1.3. The free variables are only used as grade keys in this example, so no additional key sets are merged. The `largestKey` method returns the largest key considering all keys occurring within the the given key collection. From this largest key, invalid keys for the given key set can be calculated. For the first free variable, only one invalid key is added, for the second free variable two invalid keys are added.

In the body of the loops, each Literal of the disjunction is evaluated inside another loop body. If a `TRUE` value occurs, this loop is broken. If no such value occurs, i.e., no disjunct evaluated to `TRUE`, and exception is thrown. Such a constraint violation represents a runtime error and therefore is unchecked in the binding.

Chapter 5

Procedures

The preceding chapters describe read access to the generated binding. Manipulation of documents is performed by user-specified procedures. This chapter covers, how these procedures can be realized in Java, so that unmarshaled objects can be manipulated.

The specification of a procedure (see Section 2.2) contains not only the modifying code and parameters, but also a precondition, that must hold before executing the procedure. If the precondition and the schema invariant hold before executing the procedure, it is guaranteed, that the schema invariant also holds after execution of the procedure. The precondition itself is an assertion as described in Chapter 4, but can also contain program variables, covered in Section 5.1. Section 5.2 points out other differences between embedded constraints and preconditions of procedures. The translation of the procedure body is explained in Section 5.3.

Procedures are not embedded in nodes like assertions, but in the document root and therefore work on an implicit document. The object-oriented approach would be generation of these procedures into the root class. Rather than following that approach, the root parameter as made explicit and a static procedure is generated in the binding class. In addition, procedures are bound in all classes where additional parameters can be derived from the context (see Section 5.4). A second heuristic, which derives more reduced signatures for procedures introducing new elements, and thereby removes the necessity to explicitly create new keys for these elements, is described in Section 5.5.

Allowing modification of documents spawns concerns about concurrent document access. These concerns are discussed in Section 5.6.

5.1 Program Variables

There are two very similar types of program variables: parameters and local variables. The only difference between the two is the way they are declared. Parameters are declared within the signature of the procedure, local variables are not explicitly declared. While parameters are augmented with type information in the abstract syntax, the type information of local variables is not explicitly stored in the AST, since this would require either annotation of the type at each assignment or a new statement for declaration of local variables.

Instead, type information for local variables has to be deduced from their respective applications. Type information of parameters could be derived from their applications as well instead of explicit type annotation at the parameter declaration. This is not done in the prototype for the sake of simplicity.

In the XCend theory variables are not typed at all. A variable identifier can be evaluated as a string in one application but as an integer in another. Similar to path expressions, program variables are designed as a mapping from variable identifiers to values. A variable not occurring in the map is interpreted as a unit value. If an assignment takes place in the procedure body, the mapping is updated accordingly.

Java can reflect this behavior using a map and a value supertype. However, these ambiguously typed variables can always be replaced with two uniquely typed variables, and allowing heterogeneously typed variables does not add functionality. So variables are considered to be uniquely typed. The types of parameters can be directly derived from their declaration. For local variables, the type is derived from the first occurrence of the variable, i.e., the first assignment. This allows translation of XCend variables as local variables of Java procedures. Using such a variable before it has been assigned marks a specification error.

Again, the typing of keys proves to be a more complex problem. First of all, the type of a key parameter cannot be determined from the parameter definition alone. This definition only states, that the parameter is indeed a key parameter, but not which specific key. Like with local variables, this information has to be deduced from applications of the variable. If no application exists, the parameter is not used and could also be omitted. For these cases a generic key interface is used, which basically allows all possible key types to be entered as parameter. Note that this doe not apply to local variables of type key, since the declaration of such a variable implicitly provides information about the specific key type with the value assigned to the variable.

On a further note, key compatibility as described in Section 3.4 does not consider procedures. Neither assignment of keys nor usage of keys in the precondition has influence on the type hierarchy of keys. The XCend theory, using untyped keys, naturally has no problems with any assignments or usage of keys within procedures, even if incompatible. If such an assignment occurs or a key is used for selection of another repeated element, compatibility is indeed suggested. Still, such a compatibility should be specified in the schema, not inside procedures. Key assignments and key usage in procedures in general is statically typechecked using the key compatibility as defined by embedded constraints, rather than being extended by procedure specifications.

If the key types are compatible, the value is cast internally. This would be unnecessary considering only compatible keys as defined in Section 3.4, but free variables are not typed in the prototype, i.e., generic key interfaces, and require such an explicit cast.

All parameters are checked to be not null at the beginning of each procedure. This is necessary, since Java types are nullable but XCend values are not. The assignment of a null value to, for example, an integer type parameter marks an error. To prevent access to undeclared local variables, all local variables are declared and instantiated with null after this parameter check. A sanity check performed before procedure generation guarantees, that local variables are only used after "proper" initialization with a value for the same reason.

5.2 Preconditions and Embedded Constraints

Though procedure preconditions are similar to contraints embedded in the structural definition, some differences exist. As mentioned above, preconditions may contain program variables, which cannot be used within schema assertions. On the other hand, free variables and path expressions depending on the context of a node are not allowed within procedures.

Free Variables In the procedure body and user-specified preconditions, usage of free variables is forbidden. However, since the actually checked precondition is generated from the schema invariant, which might contain free variables itself, free variables can still occur in preconditions. The translation of free variables in procedure preconditions is identical to the translation described in Section 4.1.3.

Path expressions In opposite to schema constraints, procedures are not embedded in a node. Therefore, references to the current element or an ancestor element are meaningless and considered to be invalid in path expressions within procedures. Only the root reference can be used to start a path expression. This applies to statements as well as the precondition. Note, that `RKey` also requires the context of a node and is not allowed to be used in procedures.

5.3 Statements

The body of a procedure consists of several statements which are successively executed. Each of these statements can be translated independently.

Insert Insert statements describe the introduction of a new child node. The node has a name and is added to a context given in form of a path expression. It also requires a key and a value. While for XCend this is a rather simple concept, the Java side of things is more complex.

Introduction of a new node generally equals creation of a new object, i.e., instantiation of a class. Only classes for those elements provided by the schema definition have been generated. Insertion of nodes not defined in the schema marks a specification error.

Even if the node is specified in the schema, a new object is not necessarily introduced. Not all nodes are represented by objects. Consequently, not all insert statements should introduce a new object and several cases need to be distinguished.

The abstract syntax distinguishes elements and attributes. Attributes can either be required or optional, but they can neither be repeated nor can there be a choice between several attributes. No classes are generated for attributes, instead a null value is used to encode a non-existing attribute node. Statements introducing an attribute do not introduce a new object in Java, but only assign the attribute value. A state where the attribute is added but not assigned, does not exist. At first glance this seems to be a problem, but this state can only be intermediate in XCend, since it would violate structural constraints of the invariant and a value has to be assigned afterwards within the same procedure. Such insert statements require the supplemented key to be the null key, since attributes cannot be repeated.

Considering insertion of elements, repeated elements have to be treated different from the rest. Here, the newly generated object is not simply assigned to a field of the parent object, but added to the contained map instead. The type of the used key has to be compatible to the key expected for such a repeated element. The default semantics of Java map is replacing an existing value using the same key. In XCend, such an insert is not allowed. Non-existence of an object using this key in the provided context is checked before insertion into the element map. Insertion operations on repeated elements do require a key, but no value.

Insertions of other elements require the supplemented key to be the null key. Since elements in the binding are not allowed to contain values, the provided value has to be a unit constant. Note, that constructors for all element classes but the root class require a parent reference, i.e., the parent reference is automatically set when creating a new object. The path expression of the insert statement points to the actual parent object.

Update The update statement in XCend assigns a new value to a node. On the Java side of the binding, this only makes sense for attribute paths. Other paths, especially those not defined in the structural definitions of the schema, are directly rejected by the generator. Also, the type to be assigned has to match the type of the attribute found under the given path. For key attributes, the newly assigned key also has to be compatible to the key type of the attribute.

Delete Delete is the opposing operation to insert, thus removing a child node. In Java, it has the same restrictions and considerations as the insert statement. Deletion of an attribute or a non-repeated element resets its value to null. Deletion of a repeated element however removes the element from the element map.

As with insert and update statements, the node type referenced in the provided path expression has to be defined in the schema.

Delete statements recursively delete all descendants of the removed node. This is trivially achieved in Java. If the removed object is no longer reachable from the parent object, descendants of the removed object also are not reachable.

The parent reference in the removed object is set to null as well. This prevents confusion from navigation to the parent of such an object and missing round-tripping behavior, if the object was locally stored by the user before deletion.

Conditional The conditional statement (also called `if/then/else` or branch) consists of a predicate, provided in form of a literal, and two statement blocks `then` and `else`. The predicate can be translated as stated in Section 4.1.4. The statements within the `then` and `else` branch are placed accordingly in a generated code evaluating the predicate.

If the predicate can be evaluated statically, only the code for the corresponding branch has to be generated. The user is informed of this with a warning message.

Assignment These statements assign a new value to a local variable. In XCend, all variable identifiers are valid and are initially assigned a unit value. Also bear in mind that XCend is not typed.

This behavior is less useful for the binding. Variables that have not been assigned before usage are marked as specification error. The same applies for variables, which are assigned values of different types, or values of incompatible key types, over the course of the program. Each used local variable is declared at the beginning of the procedure (cf. Section 5.1).

These properties guarantee that all program variables are unambiguously typed and initialized before assertion check. Since this also applies for procedure parameters, assign statements may also re-assign parameters without problems. The statement itself is translated as an assignment of the declared local variable in Java as well.

Note, that the described translation contradicts the usual usage of variables inside conditional statements, where a value is usually only declared for one branch. XCend also does not declare scopes for variables. If required, such behavior can be achieved using several different variable identifiers as well.

5.4 Convenience Methods

Considering the object oriented paradigm, methods should be contained within objects they are applied to. The introduced procedures are only generated into the binding class, similar to XCend, but can possibly be bound at many of the generated classes.

Looking at an example from the stats system, the user might for example have a specific account object, to which a student id should be assigned. It would be better to offer this method directly inside the account class, without necessity to explicitly pass the account key as parameter. This key can then be derived from the account object on which the method is called.

```
addStudentId(ident id, int studentId) {
  insert student at /stats/accounts/account[id]
  insert       id at /stats/accounts/account[id]/student
                  using studentId
```

The following simplified code fragment depicts how a convenience method for this procedure is integrated. The convenience method calls the static method generated in the binding class deriving parameters from the object context if possible. Such a call removes the necessity to generate the method body multiple times into the binding, which might be more bloated than in this example due to complex precondition.

```
public static class StatsBinding {
  public static addStudentId(Stats stats, Stats.Accounts.Account.
      KeyIF id, BigInteger studentId) {
    stats.accounts.account(id).student.id = studentId;
  }

  public static class Stats{
    public static class Accounts {
      public static class Account {
        Key key;
        Student student;
```

```
        public static interface KeyIF
        public static class Key implements KeyIF { ... }
        public static class Student { BigInteger studentId; }

        public addStudentId(BigInteger studentId) {
          StatsBinding.addStudentId(this.parent.parent, this.key,
              studentId);
        }
      }
    }
  }
}
```

To generate these convenience methods, it first has to be determined, which procedures can be bound to a specific node. This is done by analyzing the paths occurring within the body of each procedure. Binding is possible, if the currently analyzed node is described by a subpath within any statement of the procedure. The first such path of the procedure body is analyzed further. Binding of a procedure only makes sense, if **additional** parameters (other than the root) can be derived from the context of the node compared to the context of the parent node. This is only the case for repeated elements, where an additional key parameter, can be derived, i.e., the key of the repeated element itself. In the example above, no convenience methods are generated for the accounts or stats class, since no parameters other than the root can be derived from instances of these classes.

Considering the above account example, only a single parameter, the account key (and the root, which always can be determined), can be derived using a student id. It might be possible for several parameters to be derived. Therefore, the list of derivable parameters is calculated in a subsequent step. This list contains the root parameter as well as all parameters derivable from the path fragment pointing to the current node.

```
deleteGroup(ident uid, ident id, ident groupId) {
  delete /stats/exercises/exercise[id]/groups/group[groupId]
}
```

In case of the `deleteGroup` procedure, both `id` and `groupId` can be determined from the delete statement. The first step determines, that a convenience method will be added to the exercise and group classes. For the first convenience method, only the subpath `/stats/exercises/exercise[id]` is considered, and therefore only the `id` parameter will be derived. This makes sense, since the `groupId` is unknown for the exercise object. For the group object the `groupId` is known and is derived as well. No methods are generated for the exercises and groups classes, since those nodes do not allow derivation of **additional** key parameters compared to their parents.

When considering insertion methods, the introduction of convenience methods does not always make sense. It is not possible to invoke a method on an object which will be created with the call. Invariant generation from the XCend theory guarantees, that methods introducing a node contain a non-existence check for this node or a parent node in the precondition. This can be exploited by convenience method generation. Generation of procedures into nodes, for which such a non-existence constraint occurs in the precondition, is prevented.

5.5 Insert Procedures

Procedures introducing a new repeated element require prior generation of a fresh key, which does not violate the implicit local uniqueness constraint for keys. For such common procedures, it is desirable to hide this step from the user and automatically create this key within the procedure itself.

```
// full method call, requires explicit key
Task.Key taskKey = exam.tasks().task().nextKey();
exam.createTask(uid, taskKey, 5);

// reduced method call, implicitly creates new key
exam.createTask(uid, 5);
```

For this purpose, a heuristic is introduced which determines if a key parameter represents a new key. This information can be derived from the procedure's precondition.

Introduction of a new repeated element requires, that the key of the new element is not already used as key in the same context the element is introduced into. The precondition generator adds a non-existence constraint for such insert statements. Still, the key might be used in another context and mark a reference to another repeated element. A key parameter only marks a new key, if no additional constraints of the precondition, which imply usage of the key in another context, refer to it. This excludes usage of the key parameter in non-existence constraints as well as tally checks - those do not imply validity of the key.

The simplifier may introduce non-equality comparisons during case distinctions, which are irrelevant and also skipped for this analysis. They could be dropped in general, but this is rather a shortcoming of the simplifier and should be treated there to ensure formal correctness of this simplification step.

If later modification of a newly inserted element is required, or the new element is to be referenced from another element, the key has to be stored beforehand. This cannot be done with the new insert procedures, since neither key nor the new element are returned by the procedure. Since it is possible to introduce more than a single element, it is generally not possible to avoid this by returning the inserted element or its key.

```
Task.Key taskKey = exam.tasks().task().nextKey();
exam.createTask(uid, taskKey, 5);
exam.results().participant(accKey).addResult_task(uid, taskKey, 2);
```

The above code fragment shows such an example. There is no way to select the task key in order to add results for a participant if the reduced method call is used. Isomorphic names mitigate the problem, but the required selection of the element using the isomorphic name will have a negative influence on runtime efficiency compared with explicitly storing the new key. These new procedures do not generally replace all procedures inserting elements, but are provided in addition.

5.6 Concurrency

Considering reading access only, conccurrency is not a problem. Many users may access data at the same time without any conflicts. Concurrent access becomes a bigger issue considering manipulating procedures.

To prevent write conflicts, all procedures are synchronized over an instance of the root class. This also applies to convenience methods. The parent reference is explicitly set to `null` when a delete statement is executed. This avoids calls of convenience procedures on a removed object. The root access for synchronization fails for these cases throwing a `NullPointerException`. This exception is caught and replaced by a more verbose exception.

```
Group group = ex.groups().groupById("1");
group.deleteGroup(admin);

// The following call will fail
group.changeAttributes_group(...);
```

The above example shows such a call. The group is stored in a local variable and then deleted from the parent exercise. A subsequent call of a convenience method on the removed group will fail due to the parent reference of the group being null.

While synchronizing procedures prevents write conflicts, concurrent modifications still lead to problems between write and read access. An element may be removed during iteration over elements or keys of elements. This can be seen as less problematic, since values returned from the binding, including collections, only represent a snapshot of the object state and subsequent modifications have to be expected. Synchronization of reading access would not solve this problem. Instead, the user can manually synchronize his multiset selection and iteration, as outlined below.

```
synchronized(stats) {
  for(Group group : ex.groups().group()) {
    ...
  }
}
```

Chapter 6

Case Study

In a small case study, a slightly modified version of STATS was processed by the binding generator and a data set from the running system was copied and adjusted in order to match the modifications performed on the schema. The resulting binding file consists of about 17000 lines of Java code, the used data set contained about 300 accounts, exercise and exam entries, and participation and result information for all these accounts. Example calls on this system worked without unexpected exceptional behavior and performance was reasonable considering size of the binding and data set.

In the following, several example use-cases and their realization using the binding are described with the intention to demonstrate usability.

Calculating the points achieved in average A typical use case would be calculation of the points achieved on average in an exam.

```
Exam exam = stats.exams().examById("someExam");
ParticipantCollection participants = exam.results().participant();
double allPoints = participants.result().points().sum().intValue();
double avgPoints = allPoints / participants.size();
```

First, a specific exam is selected by its name. Afterwards, the set of all participants is selected. The points achieved by all participants in all tasks are summed up and divided by the number of participants, i.e., the expected calculation of the average points per participant. Note, how easy this calculation becomes with usage of the collection interface. The collection interface does most calculations, sparing the user the necessity of loops. The transformation into a Java integer value is possible, since the sum and size both will be far smaller than the maximal value of Java integer type variables. If this is not the case, the `BigInteger` class does also provide methods for arithmetic operations like the division, though the resulting code is a bit less readable.

Contact suitable Student Assistants For the selection of suitable student assistants, it seems appropriate to choose only those surpassing a given grade in their final exam. Therefore, we want to retain the email addresses of all students, which attained a "good" grade or better. Such an access can be easily realized using the Stats binding.

```
List<String> mails = new LinkedList<String>();
Exam exam = stats.exams().examById("someExam");
ParticipantCollection participants = exam.results().participant();
for(Participant participant : participants) {
  if(participant.result().sum().intValue() >= exam.grades().
      gradeByName("good").minPoints().intValue()) {
    mails.add(stats.accounts().account(participant.key()).email());
  }
}
```

The above code fragment does all the required work. First, a Java list for the emails is created. Next, the required exam object is accessed. Then, iteration over all participants allows access to the results stored for each participant. Since results are stored per task, the points for each task are aggregated using the `sum` aggregate. The resulting value, the overall points achieved by this participant in the exam, is compared to the points required for the grade "good". Since both values are far smaller than the maximal value of Java int type variables, they can be transformed to their int representation for easier comparison. Last, the corresponding account is selected and the email stored within the list.

This example shows the usefulness of the heuristics introduced in Chapter 3 and the collection interface. Isomorphic name selectors are used to identify exams and grades without usage of a key. Key compatibility allows usage of the participant key for selection of an account object. The collection interface allows selecting all participants, iteration over them and sums up their individual results. Only necessary key steps, transformation methods on `BigInteger` values, and the wrapping of mails into a Java list make the accessing code look a bit clunky. Usage of `BigInteger` values is unfortunately necessary to guarantee soundness of the XCend theory the precondition generation is based on and therefore should not be changed. Key steps could be replaced by allowing the element itself to be used as foreign key for selection of other repeated elements, but this proved to be more difficult than expected (cf. Section 3.4). The usage of Java lists in this example can be replaced with the introduced filtering mechanism.

```
final Exam exam = stats.exams().examById("someExam");
ParticipantCollection participants = exam.results().participant();
participants = participants.filter(new Filter<Participant>() {
  public boolean filter(Participant e) {
    return e.result().points().sum().intValue() >= exam.grades().
        gradeByName("good").minPoints().intValue();
  }
});
EmailCollection mails = stats.accounts().account(participants.key()
    ).email();
```

This fragment fulfills the same functionality, but uses the internal filtering mechanism. Instead of explicitly iterating over the participant collection and storing the mails of those participants who achieved enough points, the all participants are directly filtered by their points. The resulting collection contains only those participants with enough points. The collection of the keys of those participants can then be used to select a mail collection.

The code fragment can be reduced even more considering closures of Java 8, which remove necessity for an anonymous inner class.

```
final Exam exam = stats.exams().examById("someExam");
ParticipantCollection participants = exam.results().participant();
participants = participants.filter({ Participant e =>
  e.result().points().sum().intValue() >= exam.grades().gradeByName
      ("good").minPoints().intValue();
});
EmailCollection mails = stats.accounts().account(participants.key()
    ).email();
```

BinPackaging To show a more intricate example of procedure application and exception handling, the bin example is used.

```
element bins {
  element bin * name {
    attribute capacity { int [ . > 0 ] }
    element item * {
      attribute weight { int [ . > 0 ] }
    }
    [ ./capacity >= sum (./item/weight) ]
  }
}

addItem(key b, key i, int w) {
  insert item[i] at /bins/bin[b]
  insert weight  at /bins/bin[b]/item[i] using w
}

delItem(key b, key i) {
  delete /bins/bin[b]/item[i]
}
```

The bin example consists of several bins, which in turn contain several items. Each bin has a maximal capacity and each item has a weight. The sum of weights of contained items is not allowed to exceed the capacity. The specified procedures allow adding items to a bin or removing them.

The binpackaging algorithm described in Figure 6.1 is a backtracking algorithm, which allocates items from a source bin to a set of target bins such that no items remain in the source bin. Since no order is defined on items or bins, the algorithm is non-deterministic and may yield a different result for each run, but the result will always adhere to the specified constraints.

If the source bin contains no more elements, the algorithm is not required to do anything. Otherwise, the algorithm tries to allocate the first item, by executing the `addItem` procedure bound in each target bin. If the procedure throws an exception, the next bin is tested. If it the item was successfully allocated, it is deleted from the source folder and the next iteration takes place. If any subsequent allocation step fails, all performed allocations are rolled back and the item is assigned to the next bin with sufficient capacity instead.

The XML fragments in Figure 6.1 show the result of an example run of the binpackaging algorithm.

```
public static boolean pack(Bin source, BinCollection targets) {
  if(!source.item().iterator().hasNext()) return true;
  Item next = source.item().iterator().next();
  for(Bin target : targets) {
    try { target.addItem(next.key(), next.weight()); }
    catch(AddItemException e) { continue; }
    source.delItem(next.key());
    if (pack(source, targets)) return true;
    target.delItem(next.key());
    source.addItem(next.key(), next.weight());
  }
  return false;
}
```

```
<bins>
 <bin name="main" capacity="
     1000">
  <item weight="50"/>
  <item weight="20"/>
  <item weight="20"/>
  <item weight="30"/>
  <item weight="30"/>
  <item weight="50"/>
  <item weight="60"/>
  <item weight="40"/>
 </bin>
 <bin name="1" capacity="100"
     />
 <bin name="2" capacity="100"
     />
 <bin name="3" capacity="100"
     />
</bins>
```

$\longrightarrow$

```
<bins>
 <bin name="main" capacity="
     1000"/>
 <bin name="1" capacity="100"
     >
  <item weight="50"/>
  <item weight="20"/>
  <item weight="30"/>
 </bin>
 <bin name="2" capacity="100"
     >
  <item weight="20"/>
  <item weight="30"/>
  <item weight="50"/>
 </bin>
 <bin name="3" capacity="100"
     >
  <item weight="60"/>
  <item weight="40"/>
 </bin>
</bins>
```

Figure 6.1: Example run of the binpackaging algorithm

Chapter 7

Related Work

While most schema languages do not support arithmetic value constraints, all languages do support some kind of constraints that cannot be guaranteed purely by the Java type system and have to be checked explicitly.

Cardinality of occurring elements is supported by both DTD and XSD, though XSD allows more fine-grain definitions [15, 13, 14]. Both support the `ID` and `IDREF` types, which specify a **globally** unique identifier and a reference to said identifier. Though global uniqueness and existence of a referenced id have to hold, the id type itself is nillable, i.e., textttID type attributes may be left out. Only a single attribute of type `ID` is allowed to exist within an element. XSD additionally provides the patterns `unique`, `key`, and `keyref`. A `unique` element specifies the contained elements to be unique within a *scope*. The `key` element is similar to `unique`, but additionally, keys have to exist and may not be null, a `keyref` specifies correspondence to the referenced element. The scope of all three elements is defined using XPATH expressions [12]. These keys can be compared to the keys in XCend itself. Though scope definition using XPATH expressions is more flexible than XCend keys alone, such uniqueness can be emulated using isomorphic names. XSD 1.1 supports assertions using XPATH 2.0 expressions similar to Schematron and is therefore more powerful in terms of specification capability then XCend [17, 5, 16]. The question arises, how these constraints are translated within other data binding tools or if they are translated at all.

Due to the large number of these tools, only a subset is selected for this analysis. JAXB and Castor are evaluated for their recognition, and the Liquid XML Data Binder as representative for commercial tools. None of these tools supports XSD 1.1 schemata and therefore more complex XPATH assertions. Also, all these tools allow instantiation of non-schema conform Java objects and only check adherence to non-structural constraints during (un)marshaling or when explicitly requested by the user, which is considered to be the main advantage of the XCend language and binding over other XML schema languages and binding tools.

Castor Castor is an open source data binding tool for binding XSDs to Java. It can either generate the classes itself or bind the XSD to already existing Java classes with help of a *binding descriptor* defined in XML. The more interesting one for this thesis is the code generation approach. According to the docu-

mentation of Castors code generator, the types `ID` and `IDREF` are supported, while the patterns `unique`, `key`, and `keyref` are not [3]. Due to the age of the document, these statements were tested and verified using the latest stable build[1] provided at the homepage of the project. Castor-generated code indeed allows unmarshaling of XML documents, which violate constraints defined with the help of key and reference patterns. Still, Castor can handle cardinality and the global `ID` type and references, which are not supported by the type system. Validation is disabled per default, but `Marshaller` and `Unmarshaller` instances can be configured with several properties allowing validation and skipping of unknown attributes or elements. Validation of generated objects on the Java side can also be forced manually. However, it is possible to generate non-schema conform Java objects and encounter errors during marshaling or user-induced validation.

As an interesting fact, Castor ignores `IDREF`s to objects with no set `ID`. The marshaled XML is still valid, since such an `IDREF` attribute is left out completely, yet it seems questionable if this behavior is user-intended or if a missing `ID` value in the referenced object marks a specification error. References to instances of classes that specify no attribute of type `ID` at all are marked as errors.

Castor does not provide any functionality comparable with the collection interface of the XCend binding. It is possible to select a collection of direct children of an element, but no further navigation can be performed on the collection itself.

JAXB The *Java Architecture for XML Binding* (JAXB) is another open source framework for data binding between Java and XML [11]. It is primarily designed to handle XSD schemata, but also has experimental support for Relax NG and DTD schema files. For code generation, the XJC binding compiler is provided. The generated code does not provide any method of compliance check to the schema it was generated from. Rather, the schema has to be provided at runtime again in order to check compliance during (un)parsing of XML documents. While this validation process can handle reference and uniqueness constraints, it is purely based on XML validation using the javax.xml.validation package which is contained since Java 5.0 rather than adding only those checks not already guaranteed by the type system of Java. Older versions of JAXB provided a generated validator instead. Still, JAXB does nothing to prevent the creation of Java objects violating the schema except for the guarantees given by the type system.

In contrast to Castor, JAXB marks `IDREF`s to objects with no set `ID` attribute as error, even if the `IDREF` and `ID` attributes are set to be optional. This behavior is considered to be more intuitive.

JAXB also provides no functionality comparable to XCend collections. Elements with a `maxOccurrence` greater than one are wrapped in a Java list. Consequently, all functionality of Java lists is provided, but no navigation on the collection itself is possible.

While no statements are made about feature support concerning DTD and Relax NG schemata, and the XJC generator rejected test schemata for unknown reasons, generated code should suffer from the same problems. Validity is not guaranteed for runtime objects and validation is only performed during

[1] The tests were performed with version 1.3.3-rc1.

(un)parsing if the `(Un)Marshaller` was explicitly provided with a schema file. It also is questionable how well the validation API can handle DTD or Relax NG schema files.

Liquid XML Data Binder The Liquid XML Binder is a commercial XML binding tool. It supports XSD as well as DTD and is capable of not only binding these to Java but also to C++, C#, Silverlight, or Visual Basic [6].

The Java code generated also did not prevent creation of non-schema conform objects on the Java side and provided no guarantees beside the type system. In addition validation could not be deactivated, i.e. validation occurs inevitably during (un)marshaling. Validation is not performed by the generated library itself but by the used Xerces parser. Neither uniqueness nor key references were checked correctly. This includes `key` patterns from XSD as well as the globally unique types `ID` and `IDREF` of both XSD and DTD. Only errors concerning cardinality were detected. Consequently, Liquid XML allows generating and reading XML documents not matching the source schema the binding was generated from.

Considering collection functionality, the generated code only offers a `count` method, which calculates the number of element occurrences, i.e., can be compared to the XCend `size`. Other collection functionality is not provided, not even typesafe for each iteration over an element column, which wraps elements of the same type that occur more than once.

Overall, the functionality provided compared to non-commercial binding tools proved rather disappointing. While the GUI for binding generation was quite understandable and easier to handle then the command line interface of Castor and XJC, the generated binding did barely contain new features and the used Xerces parser performed worse in terms of correctness than the validator generated by Castor or the validation API used by JAXB.

Note, that only the trial version of this tool was tested, though the supported XML features are supposedly equal to the full version.

Schematron While Schematron in combination with one of the structural schema languages is superior in design capability, no data binding tools seem to exist for a combination of the two. Therefore, the language is of little consequence for this work. The same applies to XSD 1.1, which also supports assertions defined using XPATH 2.0 expressions, but cannot be used as source language by any binding tool so far.

Chapter 8

Conclusion

Currently, a wide variety of data binding tools is applicable for many different schema languages, binding XML documents to Java objects. Yet, Java libraries generated by these tools allow generation of objects not adhering to constraints of the schema that cannot be expressed by the type system alone. A mismatch is only detected when marshaling the object or validation is explicitly invoked by the user.

The proposed binding technique avoids this problem by separating read and write access, and restricting the latter through definition of procedures. Together with the schema invariant, which combines all constraints defined in a schema, weakest preconditions can be generated for such procedures. Adherence to the precondition can be checked before procedure execution and guarantees, that the manipulation will not violate the defined constraints [9]. The binding technique is implemented for the XCend technology and schema language. This allows definition of value and reference-based constraints, which are not supported by most schema languages [8]. The binding supports all features described by the XCend schema language.

The schema language is easy to bind by design. As a consequence, concepts of XCend can easily be related to their translation in the binding. Selection of elements and values in the binding works quite similar to selection using paths in XCend. Such paths can be easily mapped to selector chains in Java. The binding even provides a typesafe multiset selection using generated collection classes for each node in the schema. This collection interface allows not only selection of multiple elements but also navigation and even multiset references with keys. Such a multiset selection method is not provided by other binding tools like Castor [3], JAXB [11] or the Liquid XML Data Binder [6].

Most name collisions between generated classes are avoided by nesting these classes in in the same way as their respective elements. Other conflicts are not treated and rejected in order to preserve intuitive understandability of the connection between generated binding, the XCend specification, and matching XML documents.

Several heuristics have been introduced to increase typesafety and usability of the binding as well as performance of constraint checks.

Keys in the XCend binding are statically typed but retain intended interchangeability using the concept of key compatibility. Keys can be interchanged if such compatibility is expressed by the specification, but accidental misuse at

incompatible positions is prevented. Key management is handled by the system itself. Users are not required to generate unique keys and can obtain fresh keys from the system. Procedures inserting new elements do not even require an explicit key parameter, but introduce a new key automatically.

Isomorphic names are an extension of the key concept. They mark attributes, that are unique considering a scope specified with XCend paths. Such attributes are similar to key patterns in XSD [13] and are more flexible, often more powerful, than the built in XCend keys. Elements can not only be selected by their keys but also by isomorphic names, which is more intuitive than using keys. Multiple isomorphic names, even with different scopes, may exist within the same element.

Procedures are generated in the binding class. Binding of manipulating procedures into additional classes follows the object-oriented paradigm and provides this functionality directly in the objects that are manipulated. Parameters are derived from the context provided by the object, which simplifies the signature for these methods.

Static evaluation performed on procedure preconditions greatly increases runtime performance of their checks. Iteration over free variables has been reduced to symbolic checking for all invalid keys. Without such a reduction, check of free variables would not even be possible in an object oriented environment, since an infinite set of keys would have to be tested. Only assertions with free variables are evaluated inside such a loop. Incomplete evaluation further increases efficiency of constraint evaluation.

Channeling manipulation access through specified procedures together with precondition generation enables incremental maintenance of the schema invariant and guarantees, that objects on the Java side adhere to integrity constraints [9, 10]. No objects violating the schema invariant can be created with the binding.

A small case study indicates the results of this binding process to be practically applicable. While precondition generation takes quite some time, the duration of the actual binding generation is negligible. No perceivable delays were encountered during execution of procedures, which suggest that the steps taken to increase performance of assertion evaluation are sufficient for at least small to mid-sized applications.

8.1 Future Work

This section explores some directions for extension of the presented binding. This covers enhancement of the existing binding through further utility functions, generation of other useful artifacts as well as extension of the source language.

8.1.1 Generation of a Web-Based Frontend

The handwritten binding for the STAT System as described in Section 1.2 also contains a web-based frontend for access and manipulation of data, using the Wicket technology [1]. The generation of such a frontend can be automated as well using heuristics to chose appropriate presentation techniques based on the structure given by the schema and the reading and writing methods provided by the designed binding. Since adherence to constraints is already guaranteed by the binding, this is purely design effort.

8.1.2 Extensions to the Source Language

The current schema language in itself is already rather restrictive as described in Section 2.2.1. Even further restrictions are made on the schema (see Section 3.1). These constraints might limit usability and practicability of the schema language - and therefore the binding - too far. Examples are the explicit naming of elements, the necessity for keys, and missing choices over attributes. Also only four basic types are provided and additional ones like booleans or floating point values. Loosening of these constraints might make the language better suited for it's purpose, yet naturally complicate the binding and make it less intuitive.

Some of these extensions are purely syntactic sugar in the frontend and can be easily realized. Others require adjustments in the XCend theory for precondition generation and are not as trivial. Therefore, loosening of restrictions should only be done after careful consideration of implications for binding and theory and subsequent cost-benefit calculations concerning usability and understandability.

8.1.3 Further Heuristic Improvements

If the source language is not extended, several heuristic improvements could be made in order to generate a more convenient binding. The necessity of explicit naming leads to some unwanted elements in the generated binding. Examples can easily be found in the schema definition of STATS. The explicit wrapping of repeated elements can be removed (e.g. the map of `account` elements is named after the surrounding `accounts` element and could be put directly into `stats` instead of `accounts`). While this simplifies access in the binding, this pattern should not be translated into generated XML documents. There, the explicit wrapping is indeed desired, which allows better analysis in XML viewers. Optional elements without contents can be replaced by boolean values (this is for example the case with the `admin` element of accounts) and the methods generated for these elements can be changed accordingly (i.e., `isAdmin` instead of `hasAdmin`, no selection method). While this change is simpler, it is not clear

if the user wants this behavior or it was really intended to have an empty child element.

Such heuristics would improve usability of the generated binding, but it has to be determined beforehand which modifications are done via heuristics and which are integrated into the source language.

Some of the heuristics already implemented did prove to be insufficient for larger specifications such as STATS, mainly the new key heuristic, though it is debatable, if this is a problem of the heuristic being not liberal enough or the precondition generator not providing sufficient simplification. New keys generated by the prototype are not locally unique, but globally. While this does not influence correctness of the binding, reducing this to locally unique keys would increase performance not only in terms of key size but also free variable iteration, since several locally unique key values would coincide and the merged key set will be smaller.

Bibliography

[1] Apache Software Foundation: Apache wicket (2012), http://wicket.apache.org/

[2] Asami, T.: The Relaxer Project (2000), http://www.asamioffice.com/ja/2010/relaxer/index.html

[3] Blandin, A.: Castor XML Source Code Generator User Documentation. Tech. rep., Intalio (July 2001), http://www.castor.org/SourceGeneratorUser.pdf

[4] Clark, J.: RELAX NG Compact Syntax (December 2001), http://www.relaxng.org/compact-20021121.html

[5] ISO/IEC: ISO/IEC 19757-3 - Information technology - Document Schema Definition Languages (DSDL) - Part 3: Rule-based validation - Schematron (June 2006)

[6] Liquid Technologies: Liquid XML Data Binder 2012 (2012), http://www.liquid-technologies.com/xml-data-binding.aspx

[7] Michel, P.: Redesign and Enhancement of the Katja System. Tech. Rep. 354/06, University of Kaiserslautern (October 2006), https://softech.informatik.uni-kl.de/twiki/pub/Homepage/Publikationen/ir.katja.redesign.pdf

[8] Michel, P., Poetzsch-Heffter, A.: Assertion Support for Manipulating Constrained Data-Centric XML. In: International Workshop on Programming Language Techniques for XML (PLAN-X 2009) (January 2009), https://softech.cs.uni-kl.de/twiki/pub/Homepage/Publikationen/planx09.pdf

[9] Michel, P., Poetzsch-Heffter, A.: Maintaining XML Data Integrity in Programs - An Abstract Datatype Approach. In: Proceedings of the 36th Conference on Current Trends in Theory and Practice of Computer Science. pp. 600–611. SOFSEM '10, Springer-Verlag, Berlin, Heidelberg (2010), https://softech.cs.uni-kl.de/twiki/pub/Homepage/Publikationen/sofsem10.pdf

[10] Michel, P., Poetzsch-Heffter, A.: Verifying and Generating WP Transformers for Procedures on Complex Data. In: Interactive Theorem Proving (ITP) (2012)

[11] Ort, E., Mehta, B.: Java Architecture for XML Binding (JAXB) (March 2003), http://www.oracle.com/technetwork/articles/javase/index-140168.html

[12] W3C: XML Path Language (XPath) Version 1.0 (November 1999), http://www.w3.org/TR/1999/REC-xpath-19991116/

[13] W3C: XML Schema Part 1: Structures Second Edition (October 2004), http://www.w3.org/TR/2004/REC-xmlschema-1-20041028/

[14] W3C: XML Schema Part 2: Datatypes Second Edition (October 2004), http://www.w3.org/TR/2004/REC-xmlschema-2-20041028/

[15] W3C: Extensible Markup Language (XML) 1.1 (Second Edition) (September 2006), http://www.w3.org/TR/2006/REC-xml11-20060816/

[16] W3C: XML Path Language (XPath) 2.0 (Second Edition) (January 2011), http://www.w3.org/TR/2010/REC-xpath20-20101214/

[17] W3C: W3C XML Schema Definition Language (XSD) 1.1 Part 1: Structures (April 2012), http://www.w3.org/TR/2012/REC-xmlschema11-1-20120405/