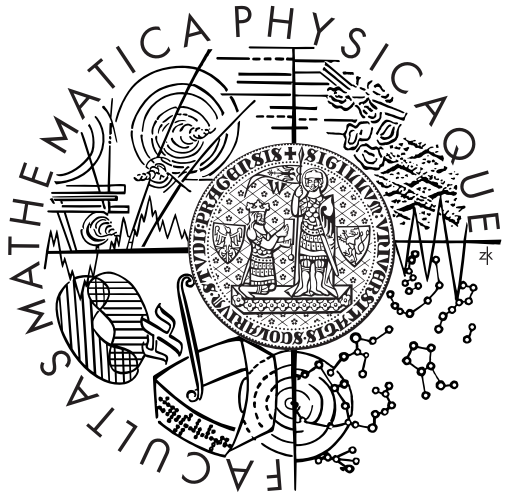
Univerzita Karlova v Praze
Matematicko-fyzikální fakulta

BAKALÁŘSKÁ PRÁCE



Kateřina Opočenská

Integritní omezení v XML

Katedra softwarového inženýrství

Vedoucí bakalářské práce: RNDr. Michal Kopecký, Ph.D.

Studijní program: Obecná informatika

2007

Děkuji panu RNDr. Michalu Kopeckému, Ph.D. za odborné vedení mé práce, za rady a za čas, který mi během vypracovávání této práce věnoval.

Prohlašuji, že jsem svou bakalářskou práci napsala samostatně a výhradně s použitím citovaných pramenů. Souhlasím se zapůjčováním práce a jejím zveřejňováním.

V Praze dne 6.8. 2007 Kateřina Opočenská

Obsah

1 Úvod **6**
1.1 Cíle práce 7

2 DTD, XML Schema a Relax NG **8**
2.1 Regulární stromové gramatiky 8
2.2 DTD 10
2.3 XML Schema 10
2.4 Relax NG 12

3 Jazyky Schematron a CliX **14**
3.1 Schematron 14
3.2 Využití síly XPath v Schematronu 15
3.3 Schematron v XML Schema nebo Relax NG 16
3.4 Kde Schematron nestačí 16
3.5 CliX - Constraint Language in XML 17
3.6 Ideje pro jazyk popisující integritní omezení 18

4 Specifikace jazyka Incox **19**
4.1 Formát souboru podmínek 19
4.2 Deklarační sekce 20
4.2.1 Deklarace globálních konstant CONST 20
4.2.2 Deklarace globálních výčtů ENUM 21
4.2.3 Deklarace celočíselných intervalů INTERVAL 22
4.2.4 Deklarace prefixů pro namespacy NAMESPACE 22
4.3 Constraint sekce 23
4.3.1 Výběr uzlů 25
4.3.2 Přesnější specifikace množiny ve výběru uzlů 27
4.3.3 Predikát 28
4.4 Komentáře 29
4.5 XML formát podmínek 30
4.6 Fuzzy pravdivost 31
4.6.1 Počítání fuzzy pravdivosti 32
4.6.2 Konkrétní počet vyhovujících 32
4.6.3 Volitelnost fuzzy pravdivosti a počtů 32
4.7 Konverzní chyby u proměnných 33
4.8 Pomocné funkce pro predikát a konstanty 34

5 Incox v kontextu ostatních schema jazyků **35**
5.1 CliX a Incox 35
5.2 Schematron a Incox 37

6 Uživatelská dokumentace **40**
6.1 Program Icval 40
6.2 Příklady výstupů v závislosti na parametrech 42
6.3 Další příklady použití programu 45

7 Programátorská dokumentace **48**
7.1 Zdrojové kódy a pomocné soubory 48
7.2 Detekce formátu vstupních podmínek 49
7.3 Parsování souboru s podmínkami 49
7.4 Vyhodnocování podmínek 52
7.5 XPath výběry 54
7.6 Konverze 54
7.7 Funkce 55
7.7.1 Přidání nové funkce 55
7.8 Výstup 57

8 Testy aplikace **58**
8.1 Test 1 - FOR ALL 59
8.2 Test 2 - FOR ALL, FOR ALL | EXISTS 60
8.3 Test 3 - FOR ALL + FOR ALL 61
8.4 Test 4 - XPath výběr 63

9 Závěr **65**

Literatura **66**

A Gramatika jazyka Incox **68**

B IncoxInput.xsd **70**

C IncoxOutput.xsd **74**

D Obsah přiloženého CD-ROM **76**

Název práce: Integritní omezení v XML
Autor: Kateřina Opočenská
Katedra: Katedra softwarového inženýrství
Vedoucí bakalářské práce: RNDr. Michal Kopecký, Ph.D.
e-mail vedoucího: Michal.Kopecky@mff.cuni.cz

Abstrakt: Tato práce se zabývá možnostmi validace omezujících podmínek v XML dokumentech. Z tohoto hlediska jsou zhodnoceny zavedené schema jazyky DTD, XML Schema a Relax NG. Dále jsou představeny na problematiku přímo specializované jazyky Schematron a CliX. Smyslem práce je návrh a následná implementace vlastního jazyka Incox (Integrity Constraints in XML). Ten je založen na využití logiky prvního řádu a jazyka XPath. Incox vychází myšlenkově z CliXu, ale přináší také řadu nových prvků jako je například volitelná syntaxe (přirozený jazyk i XML), konstanty, nebo upravené kvantifikátory, s jejichž pomocí lze přesněji specifikovat míru požadovaného splnění podmínky.

Klíčová slova: integritní omezení, XML schema jazyky, XML sémantika, Incox, Icval

Title: Integrity Constraints in XML
Author: Kateřina Opočenská
Department: Department of Software Engineering
Supervisor: RNDr. Michal Kopecký, Ph.D.
Supervisor's e-mail address: Michal.Kopecky@mff.cuni.cz

Abstract: This work deals with possibilities of integrity constraints validation in XML documents. From this point of view we review well-kown schema languages DTD, XML Schema and Relax NG. Next we present more specialized languages Schematron and CliX. The purpose of this work is to design and implement new language Incox (Integrity Constraints in XML). Incox is based on using of first-order logic and XPath language. It draws inspiration from Clix, but it also brings new features such as optional syntax (natural language as well as XML), constants or extended quantifiers to help specify required degree of truth more precisely.

Keywords: integrity constraints, XML schema languages, XML semantics, Incox, Icval

Kapitola 1

Úvod

Jazyk XML [1] je v současné době nejen velmi populárním formátem pro výměnu dat mezi aplikacemi a pro publikování na Internetu, ale prosazuje se i ve světě databází. Díky jeho textové povaze, snadnosti zpracování a faktu, že je třeba dodržovat pouze pár základních pravidel, si může prakticky každý nadefinovat jeho konkrétní podobu, odpovídající přesně potřebám své aplikace.

Tak vznikne de facto nový značkovací jazyk, odpovídající určité gramatice a pouze autor sám může rozhodovat o tom, jaké instance takového XML budou považovány za validní. O obecném XML dokumentu nelze tedy bez dalších znalostí rozhodnout, zda je či není validní. Globálně lze pouze určit, zda je či není dobře zformátovaný (well-formed), což ale může být i považováno za nutnou podmínku pro to, aby se vůbec dalo o XML hovořit.

Na druhou stranu je velmi žádoucí, abychom uměli zcela automaticky, strojově rozhodnout o tom, zda příchozí XML data jsou „ta, která chceme“. Toto testování můžeme rozdělit do dvou hlavních kategorií – na ověřování správné struktury dokumentu, neboli validaci syntaxe, a na testování korektního obsahu a závislostí mezi jednotlivými elementy, tedy na testování sémantiky dokumentu.

V současné době existuje řada jazyků a nástrojů na validaci XML dokumentů. Většina z nich se ale zaměřuje především na validaci struktury a validaci obsahu řeší pouze okrajově. Obvykle se jedná maximálně o datové typy a základní referenční integritu. Mnohdy si mohou být také požadavky na strukturu a obsah velmi blízké. Například povinnou přítomnost atributu u nějakého elementu můžeme brát jako syntaktickou záležitost, povolené hodnoty, kterých pak může nabývat, jsou už věcí obsahu.

Otázka definice struktury XML dokumentu je v současné době díky DTD a silnějším jazykům jako XML Schema [2] či Relax NG [3] dobře vyřešena. Rovněž omezení typů a povolených hodnot u elementů a atributů již nemá smysl více rozvíjet. Co naopak stojí za pozornost, jsou vzájemné vztahy mezi obsahy elementů či atributů, které mohou být mnohem komplikovanější, než je například pouze vzájemná unikátnost.

Pro definici a ověřování integritních omezení v XML neexistují v současné době žádné obecně uznávané standardy, například od W3C. Za jediný standard (ISO), který se funkčností něčemu takovému blíží, lze považovat jazyk Schematron [4], který bude v práci rovněž blíže představen. I přes jeho nespornou sílu ve vyjádření sémantických závislostí si lze ale představit celou kategorii podmí-

nek, které v něm validovat nelze. Typicky půjde o testování netriviálních vazeb mezi jednotlivými částmi dokumentu/dokumentů, což jistě může mít v rozsáhlých informačních systémech své opodstatnění.

1.1 Cíle práce

V práci budou zhodnoceny možnosti validace obsahu XML dokumentů pomocí současných schema jazyků DTD, XML Schema a Relax NG. Dále bude provedeno srovnání s jazykem Schematron, který narozdíl od výše jmenovaných uplatňuje zcela jiný přístup a pro popis a testování integritních omezení se tak stává mnohem vhodnější.

Hlavním cílem práce bude navržení a následná implementace snadno použitelného, leč dostatečně silného jazyka, určeného primárně pro popis složitějších integritních omezení (někdy nazývaných též „business rules“) v XML. Od zpracovávaných dokumentů budeme požadovat, aby již prošly určitou předchozí „syntaktickou“ validací. Máme tedy rozhodně jistotu, že předložený dokument je dobře zformátovaný a také že strukturálně jde o typ dokumentu, který chceme. Z tohoto důvodu není navrhovaný jazyk pokusem o konkurenci zavedeným schema jazykům, ale představuje další krok v celkovém procesu validace XML dokumentu.

Kapitola 2

DTD, XML Schema a Relax NG

Síla současných XML schema jazyků se obvykle formálně vyjadřuje pomocí příslušných regulárních stromových gramatik. Analogie jazyků DTD, XML Schema a Relax NG v lokálních, jednotypových a plnohodnotných regulárních stromových gramatikách nám sice dává možnost srovnání jejich vyjadřovacích možností, ty jsou ale při takovém pohledu omezeny pouze na popis struktury.

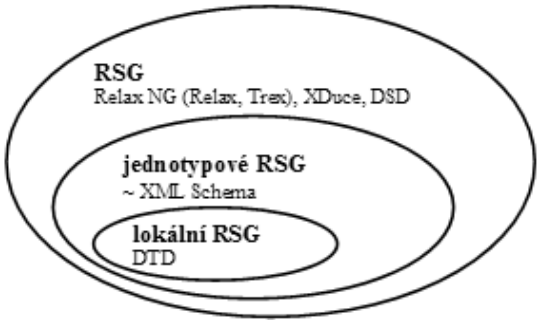
Schema jazyky ovšem svými dodatečnými interními prostředky často podporují i určité formy integritních omezení, které tak vlastně přesahují vyjadřovací možnosti dané jejich gramatickými třídami. Jako tyto „sémantické přesahy“ můžeme vnímat v podstatě jakoukoliv vyjádřitelnou informaci, která se netýká samotné struktury dokumentu a nelze ji tedy zapsat pomocí gramatik. Prakticky se jedná především o datové typy a mechanismy pro popis referenční integrity. Případně můžeme zkoumat též speciální možnosti pro popis specifičtějších kardinalit elementů, ačkoliv vyjádření počtů je gramatikami popsatelný jev.

Možnosti jazyků DTD, XML Schema a Relax NG ve zmíněných oblastech budou hlavní tématickou náplní této kapitoly.

2.1 Regulární stromové gramatiky

Cílem této práce není zabývat se teorií regulárních stromových gramatik, která je podrobně představena v [5]. Jako základ pro srovnání schema jazyků bude ovšem užitečné zopakovat základní definice a některá tvrzení, v tomto případě převzatá ze skript [6].

Definice 1: Regulární stromová gramatika (RSG)
RSG je čtveřice (N,T,S,P), kde:
N – konečná množina neterminálů
T – konečná množina terminálních symbolů
S – množina počátečních neterminálů ($S \subseteq N$)
P – konečná množina přepisovacích pravidel tvaru $X \rightarrow a\ r$,
kde $X \in N$, $a \in T$ a model obsahu r je regulární výraz nad N



Obrázek 2.1: Vztah lokálních, jednotypových a plnohodnotných RSG

Příklad 1: Regulární stromová gramatika
Následující gramatika G = (N,T,S,P) je dle [5] RSG.

```
N = {Doc, Para1, Para2, Pcdata}
T = {doc, para, pcdata}
S = {Doc}
P = {Doc -> doc (Para1, Para2*),
     Para1 -> para (Pcdata),
     Para2 -> para (Pcdata),
     Pcdata -> pcdata }
```

Definice 2: Soutěžící neterminály
Dva neterminály A, B ∈ N (A≠B) spolu soutěží, jestliže v P existují dvě pravidla A → c r a B → c r‘, kde c ∈ T a r,r‘ jsou regulární výrazy nad N.

Definice 3: Lokální RSG
Regulární stromovou gramatiku nazveme lokální, pokud neobsahuje žádné soutěžící neterminály.

Poznámka 1:
Gramatika `G` v příkladu 1 není lokální, protože obsahuje soutěžící neterminály `Para1` a `Para2`.

Definice 4: Jednotypová RSG
Regulární stromovou gramatiku nazveme jednotypovou, jestliže:
- pro každé pravidlo z P neterminály v jeho modelu obsahu nesoutěží
- neterminály v množině počátečních neterminálů S nesoutěží

Poznámka 2:
Gramatika `G` v příkladu 1 není jednotypová, protože se v modelu obsahu pravidla `Doc -> doc (Para1, Para2*)` vyskytují soutěžící neterminály `Para1` a `Para2`.

Tvrzení 1
Třída jednotypových RSG je vlastní podtřídou RSG.
Třída lokálních RSG je vlastní podtřídou jednotypových RSG.

2.2 DTD

Jazyk DTD, který je již součástí specifikace XML, je formálně nejslabší, odpovídá lokálním RSG. Pro praktické použití však v jednodušších případech dostačuje. Jazyk DTD nemá datové typy tak, jak je známe například z vyšších programovacích jazyků. Nelze tedy specifikovat, že daný atribut musí být například přirozené číslo nebo jediný znak. U elementů lze specifikovat pouze přípustný obsah, u atributů však i některé mírně složitější vlastnosti. Mezi ty, které mohou pomoci s integritními omezeními, patří především `výčty`, `ID` a `IDREF/IDREFS`.

U výčtového typu atributu lze předem zadat seznam povolených hodnot, kterých může daný atribut nabývat.

```
<!ATTLIST vyrok pravdivost (true|false) >
```

Typ atributu `ID` je v podstatě obdobou databázového klíče a zaručuje tedy jednoznačnou identifikaci v rámci dokumentu. Lze jej však použít pouze pro jeden atribut daného elementu a jeho unikátnost se vztahuje na celý dokument, což nemusí být vždy žádoucí.

Podobně je obdobou cizích klíčů typ atributu `IDREF`, resp. `IDREFS`. Tímto způsobem lze vyjádřit odkaz na atributy typu `ID`, vše v rámci jednoho dokumentu.

```
<!ELEMENT person (#PCDATA)>
<!ATTLIST person id  ID #REQUIRED
                 partner IDREF #REQUIRED>
```

Požadavky na konkrétní počty, resp. početní rozmezí elementů lze v DTD specifikovat pouze tím, že se daný element v příslušném počtu zopakuje. Následující příklad zavádí omezení, kdy element `skupina` musí mít 1 až 4 podelementy `člen`. Je zřejmé, že v případě vyšších čísel by zápis mohl být již velmi dlouhý a nepřehledný.

```
<!ELEMENT skupina  (člen, člen?, člen?, člen?)>
```

Přesahy DTD: výčty, ID/IDREF/IDREFS pouze u atributů

2.3 XML Schema

Jazyk XML Schema, doporučený konsorciem W3C, odpovídá zhruba jednotypovým RSG. Při jeho vytváření nevycházeli autoři z žádné formální teorie, proto jej lze ztotožnit s jednotypovými RSG jen s vynecháním některých jeho konstruktů. Oproti DTD má XML Schema propracovaný typový systém, který kromě řady vestavěných jednoduchých typů nabízí také možnost odvozování jednoduchých uživatelských datových typů. Odvozovat lze přitom restrikcí (`restriction`), seznamem (`list`) nebo sjednocením (`union`). Vestavěné i nově vytvořené typy lze aplikovat jak na obsah atributů, tak i na datový obsah elementů.

Pro popis referenční integrity může jazyk XML Schema využívat také mechanismus `ID/IDREF/IDREFS` známý z DTD. K těmto typům ale přidává ještě vlastní prvky `key`, `keyref` a `unique`. Díky nim lze narozdíl od DTD specifikovat požadavek na hodnoty, které musí být unikátní, ale nemusí být klíčové, tedy mohou být i prázdné. Dále je například možno definovat i klíčové elementy, nejen atributy a také pomocí jazyka XPath [7] určit konkrétní část dokumentu, na kterou se bude klíč/unikátnost vztahovat. V tomto určení lze navíc kombinovat elementy a atributy.

Pro popis kardinalit lze v XML Schema využít atributy `minOccurs` a `maxOccurs`, kterými lze omezit minimální a maximální výskyt daného elementu v rámci daného kontextu.

```
<xs:element name="skupina">
  <xs:element name="člen" minOccurs="1" maxOccurs="4"/>
</xs:element>
```

Připravovaná verze XML Schema 1.1 přináší z hlediska možnosti popisu integritních omezení zajímavé možnosti. Pomocí nových elementů `assert` a `report`, resp. jejich atributu `test` lze specifikovat podmínky na určité hodnoty ve vztazích k podelementům či atributům. Celý systém je zjevně inspirován jazykem Schematron, z něhož si XML Schema 1.1 vypůjčil obě klíčová slova i jejich logiku. Podmínka elementu `assert` je vyhodnocena jako `true`, pokud platí, `report` naopak jako `true`, pokud neplatí.

Následující příklad, převzatý z návrhu konstruktu `assert/report` [8] zavádí omezení, dle kterého musí být hodnota atributu `min` u daného elementu vždy menší nebo rovna hodnotě atributu `max`. Pro zápis samotné podmínky je využívána podmnožina jazyka XPath 2.0.

```
<xs:complexType name="intRange">
  <xs:attribute name="min" type="xs:int"/>
  <xs:attribute name="max" type="xs:int"/>
  <xs:assert test="@min le @max"/>
</xs:complexType>
```

S pomocí elementu report by stejná podmínka šla vyjádřit jako:

```
<xs:report test="@min gt @max"/>
```

Druhý příklad z návrhu vyjadřuje podmínku, podle které musí být hodnota atributu `length` stejná jako je počet podelementů `entry`. S využitím XPath funkce `count()` ji lze zapsat velmi elegantně:

```
<xs:complexType name="arrayType">
  <xs:sequence>
    <xs:element name="entry" minOccurs="0" maxOccurs="unbounded"/>
  </xs:sequence>
  <xs:attribute name="length" type="xs:int"/>
  <xs:assert test="@length eq fn:count(./entry)"/>
</xs:complexType>
```

Jazyk XML Schema není nejsilnějším ani nejčitelnějším schema jazykem, který je k dispozici. Díky silné podpoře velkých firem jako Microsoft nebo Oracle je ale v praxi spolu s DTD nejpoužívanější.

Přesahy XML Schema: datové typy jak pro elementy, tak i atributy, key, keyref, unique, minOccurs, maxOccurs, v chystané verzi 1.1: assert, report

2.4 Relax NG

Relax NG patří do formálně nejsilnější skupiny jazyků, odpovídají regulárním stromovým gramatikám. Na rozdíl od XML Schema vychází přímo ze základů postavených na teorii RSG a v tomto smyslu je také „nejčistší“ - nedisponuje navíc mechanismy, které by možnosti jeho gramatické třídy přesahovaly.

Specifikace jazyka jako taková tedy ani nedefinuje žádný závazný datový systém, konkrétní podporované datové typy jsou závislé na dané implementaci. Ta tak může například podporovat stejné typy jako XML Schema a uživatel takové implementace Relaxu k nim pak má pomocí externích referencí jednoduchý přístup, jak ukazuje následující příklad.

```
<element name="number">
  <datatype="integer"
   datatypeLibrary="http://www.w3.org/2001/XMLSchema-datatypes"/>
</element>
```

Fakt, že datové typy nejsou s jazykem pevně spojeny, může přinášet i výhody v podobě možnosti validovat velmi speciální podmínky na základě specifikace vlastních datových typů a následného referencování příslušných knihoven. V článku [9] autor nabízí ukázku vytvoření knihovny datových typů v jazyce Java, použitelnou pro validátory jako Jing nebo Multischema Validator, které umožňují přidávání dalších knihoven. Tato uživatelsky definovaná knihovna obsahuje datový typ prvočíslo (funkce na test prvočíselnosti). Po jejím přidání mezi další knihovny validátoru už stačí pouze v příslušném Relax NG schématu uvést referenci na její namespace a používat datový typ prvočíslo jako jakýkoliv jiný.

```
<?xml version="1.0"?>
<element name="numbers"
         xmlns="http://relaxng.org/ns/structure/1.0">
 <oneOrMore>
   <element name="number">
    <data type="prime"
     datatypeLibrary="http://ns.cafeconleche.org/relaxng/primes"/>
   </element>
 </oneOrMore>
</element>
```

V teoretických srovnáních vychází Relax NG proti XML Schema jako vítěz, a to nejen kvůli formálně silnější vyjadřovací síle, ale také kvůli své jednoduchosti, snadné čitelnosti a možnosti zápisu v tzv. kompaktní formě, podobnější EBNF.

Praktickou výtkou, kterou k němu někteří autoři schémat mají, je nečastěji poněkud těžkopádná možnost vyjádření kardinalit elementů, která je i přes možnost využití pojmenovaných vzorů v podstatě podobná jako u DTD.

```
<element name="skupina" >
  <ref name="člen"/>
  <optional><ref name="člen"/></optional>
  <optional><ref name="člen"/></optional>
  <optional><ref name="člen"/></optional>
</element>
```

V tomto případě se zdá být způsob s využitím `minOccurs` a `maxOccurs` z XML Schema mnohem elegantnější. Tento způsob také inspiroval autora článku [10] k myšlence, jak by se daly kardinality do hypotetické další verze Relaxu NG doplnit. K elementům určení četnosti jako `zeroOrMore`, `oneOrMore` nebo `optional` by přibyl ještě element `cardinality` s určením přesného rozmezí (který by vlastně v důsledku mohl vyjádřit i předchozí požadavky). Podle tohoto návrhu by šla naše podmínka zapsat takto:

```
<element name="skupina">
  <cardinality min="1" max="4">
    <ref name="člen"/>
  </cardinality>
</element>
```

Přesahy Relax NG: přímo žádné, uživatelsky definované knihovny datových typů

Kapitola 3

Jazyky Schematron a CliX

3.1 Schematron

Jazyk Schematron [4] je mezi ostatními schema jazyky unikátní tím, že není založen na gramatikách, ale na vyhledávání vzorů daných XPath výrazy. Výsledkem validace je potom sada uživatelsky nadefinovaných textů, informujících o splnění nebo naopak nesplnění daného pravidla. Díky tomuto odlišnému přístupu mohou být zpracovány i podmínky, které jinak nejsou vyjádřitelné pomocí regulárních gramatik.

První verze jazyka pochází od Ricka Jelliffa z roku 1999, v roce 2006 se pak Schematron stal součástí veřejného DSDL standardu ISO [11].

Syntakticky i implementačně je Schematron v porovnání s ostatními velmi jednoduchý. Dokonce jej lze ve verzi 1.5 implementovat formou XSL šablony [12], která po zpracování XSLT procesorem vyprodukuje ze zdrojového schematu jiný XSL soubor a ten už plní přímo úlohu validátoru pro (v předchozím kroku zadané) podmínky. Tedy po předání tohoto XSL a XML dat libovolnému XSLT procesoru je výstupem nadefinovaná sada textových výrazů, pokud podmínka platí, resp. neplatí.

Samotná validace jedné podmínky v Schematronu se skládá ze tří kroků:

- vybrání požadované množiny uzlů, zadané výrazem v jazyce XPath (atribut `context` elementu `rule`)
- kontrola, zda další XPath výrazy (atribut `test` elementu `report`/`assert`) jsou v kontextu vybraného uzlu pravdivé
- vypsání daného textu, pokud podmínka `test` splněna je (element `report`) nebo naopak pokud splněna není (element `assert`)

Následující Schematron schema popisuje samo sebe, respektive základní strukturu validního Schematron schematu.

```
<schema xmlns="http://www.ascc.net/xml/schematron">
  <title>A Schematron Mini-Schema for Schematron</title>
  <ns prefix="sch" uri="http://purl.oclc.org/dsdl/schematron">
```

```
  <pattern>
    <rule context="sch:schema">
     <assert test="sch:pattern">
         A schema contains patterns.
     </assert>
     <assert test="sch:pattern/sch:rule[@context]">
         A pattern is composed of rules.
         These rules should have context attributes.
     </assert>
     <assert test="sch:pattern/sch:rule/sch:assert[@test]
                or sch:pattern/sch:rule/sch:report[@test]">
         A rule is composed of assert and report statements.
         These rules should have a test attribute.
     </assert>
    </rule>
  </pattern>
</schema>
```

3.2 Využití síly XPath v Schematronu

Díky jazyku XPath, především jeho predikátům a vestavěným funkcím, lze pomocí Schematronu lehce ověřovat i jinak ostatními schema jazyky nepopsatelné podmínky. Jedná se přirozeně o podmínky, které nelze vyjádřit pomocí gramatik, případně ani s využitím datových typů.

Už v samotném výběru testovaných uzlů lze použít mnohem jemnější specifikace než je pouhé jméno a zařazení ve stromové struktuře. Užitečné je především určení pořadí elementu (`last()`, `first()` a aritmetika). Rovněž lze validovat přítomnost určitého počtu netriviálně zadaných podelementů či celkového součtu jejich hodnot, a to opět díky XPath funkcím `count()`, resp. `sum()`.

Některé rozumné podmínky, obtížněji nebo vůbec zvládnutelné klasickými schema jazyky, demonstruje tutorial na [13], například:

Součet hodnot všech vybraných elementů je určité číslo
Součet hodnot všech podelementů Percent elementu Total je dohromady 100:

```
<schema xmlns="http://www.ascc.net/xml/schematron" >
  <pattern name="Suma je 100%">
    <rule context="Total">
      <assert test="sum(//Percent) = 100">
            Součet hodnot není 100%.
      </assert>
    </rule>
  </pattern>
</schema>
```

Dokument obsahuje stejný počet zvolených elementů
Počet elementů AAA a BBB je v celém dokumentu stejný.

```
<schema xmlns="http://www.ascc.net/xml/schematron" >
  <pattern name="Test počtu elementů">
    <rule context="/*">
     <report test="count(//BBB) = count(//AAA)">
           O.K.
     </report>
     <assert test="count(//BBB) &lt;= count(//AAA)">
           Nějaké elementy AAA chybí
     </assert>
     <report test="count(//BBB) &lt; count(//AAA)">
           Nějaké elementy BBB chybí
     </report>
    </rule>
  </pattern>
</schema>
```

3.3 Schematron v XML Schema nebo Relax NG

Schematron je jazyk vhodný především k souběžnému použití s XML Schema nebo Relax NG. Těmto jazykům se přenechá úloha popsání celkové struktury, v Schematronu se pak dospecifikují dílčí sémantická omezení.

Zařazení prvků Schematronu do schemat zmíněných jazyků je v obou případech jednoduché, názorné příklady obsahuje například tutorial [14]. Validace celého schematu pak může proběhnout buď speciálním validátorem nebo odděleně, kdy se Schematron část vyextrahuje pomocí XSLT a následně zvaliduje přímo validátorem Schematronu.

U XML Schema se celá Schematron sekce `pattern` s popisy podmínek vloží do elementu `appinfo`. Syntaxe Relax NG je volnější a `pattern` sekce může být prakticky kdekoliv. K dispozici je rovněž speciální Schematron syntaxe pro kompaktní zápis Relaxu [15], tedy zápis, který není v XML.

3.4 Kde Schematron nestačí

Možnosti Schematronu stojí a padají s XPathem. Ten nabízí poměrně velkou vyjadřovací sílu a lze v něm zapsat i celou řadu na první pohled obtížných podmínek jako je třeba struktura haldy či stromu součtů[1] nebo konzistence rodných čísel a jmen (záznamy se mohou opakovat, ale mají-li dva stejné RČ, musí mít i stejné jméno).

I přesto ovšem existuje řada komplexních omezení, které ve Schematronu popsat nelze. Typicky jsou to složitější sémantická omezení v business aplikacích,

[1] Součet hodnot všech uzlů v každém podstromu je roven hodnotě v kořeni tohoto podstromu; hodnota v listech může být libovolná

pro jejichž popis sám autor Schematronu doporučuje využít spíše CliX [16] nebo OASIS CAM [17].

Nepopsatelné podmínky si můžeme zjednodušeně představit jako požadavky typu „každý element A má (alespoň) jeden podelement B takový, že pro všechny jeho podelementy C platí podmínka P“. Problém u Schematronu není v samotném testování, ale spíše ve způsobu výstupu.

Například následující schema sice správně vypíše `OK` pro každý podelement `b` elementu `a` takový, že všechny jeho podelementy `c` jsou rovny 1. Z takového výstupu ovšem nemáme šanci poznat, zda takový podelement `b` existuje skutečně pro každé `a`, a to dokonce ani při známém počtu elementů `a` (v některém může být takových `b` více, v jiném žádné).

```
<pattern name="abc">
  <rule context="//a">
    <report test="b[c=1]">OK</report>
  </rule>
</pattern>
```

3.5 CliX - Constraint Language in XML

Většina podmínek, které v Schematronu zpracovat nelze, připomíná svoji strukturou (podobně jako ta motivační) logickou formuli, tedy výraz s kvantifikátory $\exists$ a $\forall$. Právě na logice prvního řádu je také založen jazyk Constraint Language in XML, zkráceně CliX [16], jehož první a zatím též poslední specifikace pochází z roku 2004. Schema v CliXu má zhruba následující strukturu:

```
<clix:rules
  xmlns:clix="http://www.clixml.org/clix/1.0" version="1.0">
  <clix:header>
    <clix:author>Michael Marconi</clix:author>
    <clix:description>A set of rule examples</clix:description>
    <clix:comment>
           This should help to demonstrate the rules element
    </clix:comment>
  </clix:header>

  <clix:rule id="rule1">
    <clix:forall var="x" in="//elements">
      <clix:equal op1="$x/value" op2="'15'"/>
    </clix:forall>
  </clix:rule>

  <clix:rule id="rule2"> ...  </clix:rule>
</clix:rules>
```

Nutno podotknout, že ačkoliv je CliX postaven na formálně silnějších základech než Schematron, není ani zdaleka tak rozšířený nebo podporovaný - částečně i

díky tomu, že se dnes již nijak nevyvíjí. Přitom teoreticky nabízí poměrně silné prostředky i čitelnou syntaxi. Například popsání podmínky z 3.4 je v něm velmi intuitivní:

```
<forall var="A" in="//a">
  <exists var="B" in="$A/b">
    <forall var="C" in="$B/C">
      <equal op1="$C" op2="'1'" />
    </forall>
  </exists>
</forall>
```

3.6 Ideje pro jazyk popisující integritní omezení

Využití predikátové logiky pro popis vzájemných závislostí a zároveň využití XPathu pro jednotlivé výběry se jeví jako velice přirozená myšlenka, korespondující se způsobem běžného lidského vyjádření požadovaných podmínek („aby pro každý uzel platilo, že pro všechny jeho děti … “ atd.)

Rozhodně by ale bylo pěkné mít v popisu samotné podmínky větší volnost a širší nabídku funkcí, než nabízí Clix. Je ovšem pravda, že i autoři specifikace CliXu počítali s možností uživatelského rozšíření ve formě plugin operátorů a (zatím blíže nespecifikovaných) maker.

Rovněž by celý jazyk mohl mít volitelně i kompaktní syntaxi podobně jako Relax NG.

V neposlední řadě by se mohl hodit výstup, který dokáže pro danou podmínku říci více, než jen to, zda je či není splněna. Například počet nebo procento vyhovujících uzlů a případně i mechanismus, dovolující tuto část vyjádřit přímo v požadavku na dokument („aby pro 60-70% uzlů platilo; aby existovaly aspoň tři uzly, pro které platí“).

Všechny tyto ideje se pokusí realizovat jazyk, popsaný v následující kapitole.

Kapitola 4

Specifikace jazyka Incox

Jazyk Incox (Integrity Constraints in XML) vychází z výrazových prostředků logiky prvního řádu, a je tedy založen na používání kvantifikátorů. Na podmínkovou formuli, vyjadřující příslušné integritní omezení, jsou z hlediska syntaxe kladeny určité požadavky, díky kterým v podstatě jedna podmínka odpovídá jedné logické formuli, zapsané v prenexním tvaru.

Narozdíl od podobně zamýšleného, v předchozí kapitole představeného, projektu CliX [16], připomíná primární syntaxe Incoxu spíše XQuery nebo SQL. Záměr je takový, aby psaní podmínek kopírovalo přirozený jazyk a podmínky se tedy daly i formálně zapisovat velmi podobně tomu, jak bychom je vyjádřili slovně. Pro případy, kdy více vyhovuje vstup přímo v XML, lze použít XML ekvivalent Incoxu, který je představen v kapitole 4.5. Pro navigaci v rámci dokumentu a pro výběr množin testovaných prvků využívá Incox běžně používaný XPath 1.0.

Celý jazyk je zamýšlen jako case insensitive. Pro přehlednost budou ovšem v dalším textu psána klíčová slova velkými písmeny.

Příklady, uvedené dále v této kapitole, lze spolu s příslušnými testovacími daty nalézt na přiloženém CD v adresáři `examples/spec`.

4.1 Formát souboru podmínek

Soubor s podmínkami, podle kterých chceme obsah XML dokumentů validovat, se skládá z úvodní nepovinné deklarační sekce a dále z jednotlivých podmínkových bloků. Tyto bloky jsou vyhodnocovány nezávisle na sobě, pro každý je tedy samostatně určeno, zda podmínka platí či ne a případně jak moc je porušena (viz kapitola 4.6).
Základní struktura souboru s omezujícími podmínkami vypadá následovně:

```
declarations
constraint1
constraint2
...
constraintN
```

4.2 Deklarační sekce

V deklarační sekci se mohou v libovolném počtu i proložení nacházet čtyři typy deklarací:

- deklarace globálních konstant `CONST`
- deklarace globálních výčtů `ENUM`
- deklarace globálních celočíselných intervalů `INTERVAL`
- deklarace globálně použitelných prefixů pro namespacy `NAMESPACE`

První tři typy deklarací jsou vždy konstantní a v rámci validace už nemohou být nijak dodatečně pozměňovány. Jejich využití spočívá v zpřehlednění zápisu podmínek a v odstranění redundancí, pokud bychom nějakou hodnotu museli z dokumentu pomocí XPathu získávat opakovaně. Globalní deklarace `NAMESPACE` mohou být pro stejný prefix překrývány, platná je vždy ta naposledy použitá.

4.2.1 Deklarace globálních konstant CONST

CONST[**:**] constname = expr

`Constname` je identifikátor, reprezentující v celém následujícím dokumentu hodnotu výrazu `expr`. Musí splňovat podmínky pro identifikátor (viz Příloha A).

V momentě překladu musí být hodnota výrazu `expr` jednoznačně určená a následně se již nikdy nebude měnit. Do konstanty lze typově přiřadit číslo (celé, reálné), řetězec a nebo XPath výraz. Pokud XPath výraz vrací množinu uzlů, pak je konstanta rovněž typu množina a její identifikátor může být uveden v podmínkovém výběru na místě množiny.

Ve výrazu `expr` lze dále používat přetížené konverzní funkce `str()`, `int()` a `real()`, popsané v kapitole 4.8. Platí, že pokud XPath výraz vrací jednoprvkovou množinu uzlů a použijeme-li na tuto množinu některou z konverzních funkcí, pak je výsledkem zkonvertovaná hodnota daného uzlu. V případě pokusu zkonvertovat víceprvkovou nebo prázdnou množinu je zahlášena běhová chyba.

Objeví-li se v `expr` (resp. kdekoliv jinde v zápisu) XPath výraz, může se před ním pomocí odkazu `#n` určit číslo souboru, ke kterému se bude tento výběr vztahovat. Hodnota `n` je celé číslo, odpovídající pořadí zdrojového XML souboru při volání validátoru. Pokud je sekce `#n` vynechána, automaticky se XPath výběr vztahuje k prvnímu zdrojovému souboru.

Příklady

```
CONST deset = (2+3)*2
CONST jmenoJan = "Jan"
CONST vsechnyPobocky = #2 '/firma/pobocka/'
CONST posledniPobocka = str('/firma/pobocka[last()]/jmeno')
CONST oPosledniPobocce
      = "Posledni pobocka je: " + posledniPobocka
CONST maxCislo = int('//num[not(../num > .)]')
```

V prvním případě je vypočtena hodnota konstanty `deset` rovna 10. V druhém případě jde o řetězcovou konstantu.

Třetí příklad demonstruje konstantu, která reprezentuje množinu uzlů všech poboček firmy, vybraných z dokumentu, předaného jako druhý zdrojový XML soubor. V případě, kdy by tato množina obsahovala dva či více prvků (předpokládáme více poboček), by jakákoliv případná konverze vyvolala chybu. Naopak, u dalšího příkladu, kdy XPathem vybíráme ze všech poboček jméno té poslední (poslední v rámci řazení v dokumentu), již můžeme uplatnit konverzní funkci `str()`. Problém by mohl nastat pouze v případě, kdy by neexistovala žádná pobočka.

Konstanta `oPosledniPobocce` ukazuje zřetězení dvou výrazů typu string, konstantního řetězce a již dříve definované konstanty `posledniPobocka`. Poslední příklad pak předvádí konverzi vybrané hodnoty, nejvyšší ze všech hodnot elementů `num`, na celé číslo. Konverze opět může proběhnout pouze v případě, že existuje ostré maximum. V opačném případě (více nejvyšších hodnot) by totiž byly XPathem vybrány všechny vyhovující uzly s nejvyšší hodnotou a následná konverze víceprvkové množiny na číslo by nebyla možná.

4.2.2 Deklarace globálních výčtů ENUM

ENUM[**:**] constname = (expr1, ... , exprN)

Tato deklarace sestrojí posloupnost prvků, která se bude v podmínkách moci vyskytnout všude tam, kde očekáváme množinu, tedy typicky XPath výraz, vracející uzly. `Constname` je identifikátor. Pro `expr` platí stejná pravidla jako v klauzuli `CONST`. Ve výčtu se může rovněž objevovat odkaz na již definovanou konstantu.

Příklady

```
ENUM hledanaJmena = ("Josef", "Jakub", jmenoJan)
ENUM kontrolovanePobocky = (deset, 4, 6, 28)
ENUM nehomogenni = (1, 1.1, "Hello World")
```

První příklad ukazuje výčet řetězcových konstant, druhý celočíselných konstant. Třetí deklarace demonstruje, že množina nemusí být nutně homogenní. Jejími prvky ovšem mohou být pouze jednoduché typy, ne jiné množiny.

4.2.3 Deklarace celočíselných intervalů INTERVAL

INTERVAL[**:**] constname = (start, end [, step])

Výrazy `start`, `end` a nepovinný `step` (implicitně 1) musí být přirozená čísla. Zadána mohou být přímo, pomocí celočíselných konstant nebo konverzní funkcí `int()`. Příkaz v podstatě zajistí zkonstruování posloupnosti přirozených čísel daného rozsahu. Typ intervalové konstanty je tedy stejně jako u `ENUM` vždy množina.

Příklady

```
INTERVAL suda = (0, deset, 2)
INTERVAL cisla = (1, int('/nums/num[last()]'), 1)
```

První příkaz sestrojí posloupnost (množinu) čísel 0, 2, 4, 6, 8 a 10. Druhá konstanta `cisla` reprezentuje posloupnost přirozených čísel od 1 do hodnoty posledního elementu `num`.

4.2.4 Deklarace prefixů pro namespacy NAMESPACE

NAMESPACE[**:**] px = "ns"

Příkaz zavede vazbu mezi prefixem `px` a namespacem `ns`. Tento prefix bude dále použitelný ve všech XPath výběrech ve všech podmínkách. Jeho použití zajistí vybrání jen takových uzlů, vyhovujících příslušnému XPath dotazu, které ve zdrojovém XML patří do asociovaného namespace `ns`.

Příklad

Hodnota každého elementu `num` z namespace `incox` je 1:

```
NAMESPACE: ix="incox"

CONSTRAINT "XPath s prefixem" {
FORMULA:
 FOR ALL x IN '//ix:num'
 ( int(x) = 1 )
}
```

Dokument:

```
<?xml version="1.0" ?>
<ic:root xmlns:ic="incox">
  <ic:num>1</ic:num>
  <ic:num>1</ic:num>
</ic:root>
```

V případě, že bychom nezavedli vazbu mezi prefixem `ix` a namespacem `incox`, skončila by validace podmínky na neplaném XPath dotazu (neznámý prefix). Pokud bychom se naopak pokoušeli uplatnit dotaz bez prefixu, tedy pouze `'//num'`, byla by množina vybraných prvků prázdná, protože ve zdrojovém souboru patří všechny elementy **num** do namespace `incox` a ten je třeba explicitně uvést. Asociovaný prefix nemusí být přitom stejný jako ve zdrojovém souboru (zde `ic`), pouze se musí odkazovat na stejný namespace (zde `incox`).

Pokud je atributem `xmlns` zaveden defaultní namespace, pak pro výběr prvků z tohoto namespace musíme v podmínkovém souboru zvolit prefix a svázat jej s tímto namespacem. V XPath výběrech se pak do defaultního namespace odkazujeme přes tento prefix.

Příklad

Dotazem `'//x'` nad následujícím dokumentem vybereme oba elementy `x`.

```
<?xml version="1.0" ?>
<root>
  <x>1</x>
  <x>2</x>
</root>
```

V případě stejného dotazu nad tímto druhým dokumentem bude ale množina vybraných uzlů prázdná.

```
<?xml version="1.0" ?>
<root xmlns="default">
  <x>1</x>
  <x>2</x>
</root>
```

Řešením je zavedení speciálního prefixu pro namespace `default`:

```
NAMESPACE pf="default"
```

Všechny prvky `x` z druhého dokumentu pak získáme dotazem `'//pf:x'`.

Určení, do kterého namespace jaký element patří, se v případě překrývání řídí standardními pravidly pro namespacy v XML.

4.3 Constraint sekce

Constraint sekce je uvozena klíčovým slovem `CONSTRAINT`, za kterým je uvedeno jméno podmínky v uvozovkách. Pojmenování podmínek je zde z důvodu uživatelské přívětivosti výstupu, na kterém tak mohou být jednotlivé podmínky dle jména snáze identifikovány.

Uvnitř složených závorek, ohraničujících a zabalujících podmínku, může být v libovolném proložení uvedeno několik parametrů s případnými povolenými hodnotami, které ukazuje následující schéma.

CONSTRAINT ”jmenoPodminky” **{**
[**STATE**[:] (**ON** | **OFF**)]
[**ONERROR**[:] (**TRUE** | **FALSE**)]
[**FUZZY**[:] (**ON** | **OFF**) [precision]]
[**COUNTS**[:] (**ON** | **OFF**)]
{ **NAMESPACE**[:] px = ”ns” }
FORMULA[:]
logical_formula
}

STATE
Parametr je nepovinný a určuje, zda se podmínka vyhodnocovat bude (`ON`) nebo ne (`OFF`). Není-li parametr uveden, podmínka se vyhodnocuje.

ONERROR
Parametr je nepovinný a definuje defaultní vyhodnocení podmínky, ve které nastala chyba z důvodu typové konverze proměnné (viz kapitola 4.7). Pokud není chování explicitně definováno, vyvolá se při chybné konverzi běhová chyba.

FUZZY
Parametr je nepovinný a jeho nastavení na hodnotu `ON` zajistí vypsání poměru vyhovujících prvků ku všem pro množinu prvního kvantifikátoru (viz kapitola 4.6). Volitelný atribut `precision` je celé číslo v rozmezí 0-15, určující nejvyšší počet desetinných míst, na které se fuzzy pravdivost vypíše.

Pokud je parametr `OFF`, fuzzy pravdivost se nepočítá. Pokud není uveden vůbec, řídí se výstup parametrem příkazové řádky a není-li ani tento uveden, fuzzy pravdivost se nepočítá.

COUNTS
Parametr je nepovinný a jeho nastavení na hodnotu `ON` zajistí vypsání konkrétních počtů vyhovujících prvků a všech prvků v množině prvního kvantifikátoru.

Pokud je parametr `OFF`, počty se nevypisují. Pokud není uveden vůbec, řídí se výstup parametrem příkazové řádky a není-li ani tento uveden, počty se nevypisují.

NAMESPACE
Parametr je nepovinný a určuje svázání daného prefixu `px` se zadaným namespacem `ns`, který je uveden v uvozovkách. Tento prefix lze pak používat v XPath výběrech, ale pouze v dané podmínce. Pro jeho používání platí stejná pravidla jako v případě globální deklarace. Pokud je prefix shodný s jiným prefixem, dříve deklarovaným v globální deklaraci `NAMESPACE`, pak má tento lokálně deklarovaný přednost (tedy překrývá na úrovni dané podmínky globální).

Paramter `NAMESPACE` může být použit násobně pro zavedení různých prefixů a k nim asociovaných namespaců. V případě kolize prefixů má vždy přednost ten později uvedený.

FORMULA
Parametr je povinný, nesmí být uveden násobně a jeho obsahem je logická formule podmínek, která vypadá následovně:

```
select1
select2
...
selectN
( predicate )
```

Logická formule se skládá z libovolně mnoha libovolně proložených výběrů pomocí `FOR { ALL | AT LEAST | AT MOST }` nebo `EXISTS [!]` a z uzávorkovaného predikátu. Výběry a omezení odpovídají logické formuli v prenexním tvaru.

V sekcích `select` definujeme jména proměnných a určujeme domény (množiny), které budeme s těmito proměnnými procházet. V sekci `predicate` potom dáváme takto definované proměnné do relací, které chceme, aby příslušné prvky (všechny, jeden, právě jeden, určité procento či určitý počet) z dané domény splňovaly.

4.3.1 Výběr uzlů

Klauzule pro výběr kopírují použití buď existenčního, nebo všeobecného kvantifikátoru. Ten je navíc k dispozici i v rozšířené verzi.

FOR ALL var **IN** set

V případě výběru `FOR ALL` bude proměnná `var` nabývat postupně hodnoty všech prvků z množiny `set` (viz 4.3.2). Je-li tato množina prázdná, bude predikát vždy splněn, tedy `TRUE` (pro prvky prázdné množiny platí cokoliv).

FOR AT LEAST n[%]**, AT MOST** n[%] var **IN** set

V případě rozšířeného `FOR { AT LEAST | AT MOST }` výběru můžeme přirozenými čísly specifikovat, pro kolik nejméně a nejvíce prvků či % prvků v dané množině má podmínka platit. Nemusí být nutně zadány obě hranice, výběr lze uvést též pouze s `AT LEAST` nebo pouze s `AT MOST`. Požadovaný počet platných hodnot může být určen absolutně nebo % z celkové velikosti množiny, v případě % musí být přirozené číslo n $\in$ [0,100]. Přípustná je rovněž možnost určení jedné hranice absolutně a druhé %.

Je-li rozmezí zadané oběma hranicemi prázdé, hlasí validátor chybu při překladu. Tato chyba je ovšem detekovatelná pouze v případě, že jsou obě hranice zadány buď absolutně (`FOR AT LEAST 3, AT MOST 2`), nebo % (`FOR AT LEAST 25%, AT MOST 5%`). Pokud je jedna mez zadána absolutně a druhá % a je-li určené rozmezí pro zadaná data prázdné, není tento problém na úrovni překladu zjistitelný. Taková podmínka ovšem nemůže být nikdy splněna, protože na to, aby splněna byla, by musely souběžně platit oba požadavky, což není možné.

Pokud specifikujeme při použití pouze horní mez, tedy `FOR AT MOST m[%]` a neurčíme-li dolní mez, znamená to, že požadujeme, aby podmínka byla splněna,

platí-li nejvýš pro m[%] prvků množiny. Tomu vyhovuje i případ, kdy neplatí pro žádné prvky, tedy defaultní dolní mezí je vždy `FOR AT LEAST 0`.

Naopak, určíme-li pouze dolní mez, tedy `FOR AT LEAST n[%]`, je defaultní horní mezí `FOR AT MOST 100%`. Jedině tak mohou být přijaty opravdu všechny stavy, kdy podmínka platí pro n[%] prvků a více.

Je-li testovaná množina `set` prázdná, vyhodnotí se podmínka v závislosti na tom, jak jsou zadány hranice. Jsou-li zadány některým z následujících způsobů, bude podmínka vyhodnocena jako `TRUE`:

- `FOR AT LEAST n%`
- `FOR AT MOST m%`
- `FOR AT LEAST n%, AT MOST m%` (n<m)
- `FOR AT LEAST 0, [AT MOST m[%]]`
- `FOR [AT LEAST n%,] AT MOST 0`

Jsou-li zadány obě meze %, nebo je-li zadána pouze jedna mez, a ta je určena %, je podmínka pro prázdnou množinu vždy splněna (0 prvků prázdné množiny tvoří 0% i 100% a tedy také libovolných n%).

U absolutně zadaných mezí je splněna pouze tehdy, pokud je jedna z hranic určena jako `FOR AT LEAST 0` nebo `FOR AT MOST 0`. Druhá hranice může být v prvním případě určena libovolně (i vůbec), v druhém případě může být určena buď % nebo vůbec. Nejméně i nejvýše 0 prvků prázdné množiny vyhovuje jakékoliv podmínce. Odpovídá to vlastně situaci, kdy chceme, aby pro všechny prvky (`FOR ALL`) prázdné množiny (kterých je 0) platila daná podmínka, a to bude vždy. Ve všech ostatních případech, než jsou ty uvedené výše, bude podmínka v případě prázdné množiny vyhodnocena jako `FALSE`.

Příklady podmínek s kvantifikátorem `FOR { AT LEAST | AT MOST }` obsahuje soubor `for.txt`, nacházející se na přiloženém CD v adresáři `examples/spec`.

EXISTS [**!**] var **IN** set

V případě výběrů `EXISTS [!]` můžeme požadovat buď pouhou existenci proměnné `var` v rámci množiny `set` a nebo (verze s vykřičníkem) existenci právě jednoho takového prvku. Chceme-li se pokusit tento výběr provést z prázdné množiny, budou příslušné predikáty `FALSE` (neexistuje vyhovující prvek).

Příklady

Plat každého zaměstnance musí být kladná hodnota:

```
CONSTRAINT "kladne mzdy" {
FORMULA:
 FOR ALL plat IN '//zamestnanec/vyplata'
 ( int(plat) > 0 )
}
```

V seznamu existuje alespoň jedno sudé číslo:

```
CONSTRAINT "sude" {
FORMULA:
 EXISTS y IN '//num'
 ( int(y) MOD 2 = 0 )
}
```

Ve firmě pracuje zhruba stejné množství můžů a žen:

```
CONSTRAINT "rovnomerne rozdeleni" {
FORMULA:
 FOR AT LEAST 45%, AT MOST 55% x in '//zamestnanec/pohlavi'
 ( str(x) = "M" )
}
```

Na každé pobočce existuje právě jeden zaměstnanec s platem nad 50 000

```
CONSTRAINT "dobre placeni" {
FORMULA:
 FOR ALL pob IN '//pobocka'
  EXISTS ! plat IN '$pob/zamestnanec/vyplata'
  ( int(plat) > 50000 )
}
```

Poslední příklad demonstruje využití již získané proměnné `pob` (prvku výběru) v dalším výběru. Pro druhý výběr (`EXISTS!`) již v danou chvíli reperezentuje proměnná `pob` konkrétní jeden uzel. Odkaz v rámci XPath výběru se provádí pomocí znaku `$`, aby se rozlišilo mezi případně stejně pojmenovanými elementy XML.

4.3.2 Přesnější specifikace množiny ve výběru uzlů

Množina prvků `set` může být v případě všech kvantifikátorů zadána buď přímým XPath výrazem a nebo konstantou typu množina.

Pokud je množina uzlů `set` určena XPath výrazem, platí pro něj určitá omezení. Předně, jde o podmnožinu jazyka XPath 1.0 [7], což vylučuje použití některých funkcí z verze 2.0 jako je třeba `fn:doc()` pro výběr dokumentu. Stejně jako u konstant, i pro výběry s kvantifikátory platí, že dokument, ke kterému se daný XPath výběr vztahuje, může být určen předcházející direktivou `#n`, kde n je celé číslo, odpovídající pořadí zdrojového XML souboru při volání validátoru.

Pro použitý XPath výraz dále platí:

- vrací množinu uzlů, ne hodnotu
  (nelze tedy například psát `FOR ALL x IN 'count(//num)'`)
- jako implicitní kontext se vždy bere kořen dokumentu, pokud není referencovanou proměnnou určeno jinak

- referencovaná proměnná `var` může pocházet pouze z předchozího XPath výběru v téže podmínce a objevuje se ve tvaru `$var` (viz demonstrační příklad)
- obsahuje nejvýš jednu referencovanou proměnnou pro určení kontextu
- obsahuje-li referencovanou proměnnou, je touto proměnnou určen pochopitelně i dokument, proto nemá smysl používat direktivu `#n` před XPath výrazy s referencovanými proměnnými

V případě XPath výrazů, které jsou využity jako argumenty funkcí v sekci predikátu, platí stejná omezení kromě prvního. XPath výrazy v sekci predikátu mohou libovolně využívat XPath funkce (verze 1.0) a vracet tedy i čísla, boolovské hodnoty a řetězce. Celková korektnost výrazu v predikátu záleží pochopitelně na použitých funkcích Incoxu, viz 4.7.

Příklad

V každé pobočce z druhého souboru (#2) pracuje více pokladních než prodavačů:

```
CONSTRAINT "#pokladnich > #prodavacu" {
FORMULA:
  FOR ALL pob IN #2 '//pobocka'
  ( int('count($pob/zamestnanec[pozice="prodavac"])')
    <
    int('count($pob/zamestnanec[pozice="pokladni"])') )
}
```

4.3.3 Predikát

(boolval1 LOGOP boolval2 LOGOP ... LOGOP boolvalN)

Logický operátor `LOGOP` může být buď `OR`, `AND` nebo `->`, představující implikaci. Podmínka `boolval` produkuje boolovskou hodnotu, a to buď jako výsledek porovnání porovnatelných typů nebo jako vrácenou hodnotu funkce. Základní funkcí pro testování (ne)prázdnosti vybrané množiny uzlů (XPath výraz) je funkce `empty()`.

Pokud není na proměnnou z výběru uplatněna žádná konverze, pohlíží se na ní skutečně jako na proměnnou (místo v dokumentu). Pokud je uplatněna některá z konverzních funkcí `str()`, `int()` nebo `real()`, pokusí se validátor interpretovat hodnotu aktuálního uzlu jako příslušný typ.

Hodnota uzlu je pro použití v predikátu definována jako zřetězení (ve směru sekvenčního čtení) veškerého jeho datového (textového) obsahu. Například:

- hodnota uzlu `<num>1</num>` je `1`
- hodnota uzlu `<a><num>1</num><num>2</num></a>` je `12`
- hodnota uzlu se smíšeným obsahem `<a>text<num>1</num></a>` je `text1`

Proměnná nabývá v každé chvíli hodnoty nějakého prvku ze zadané množiny a nedochází tedy k chybám z důvodu pokusu „konverze množiny na hodnotu“. Přesto může chyba nastat, a to například v případě, že se textovou hodnotu uzlu snažíme interpretovat jako číslo.

Příklady

Unikátnost hodnot elementů num:

```
CONSTRAINT "unikatnost" {
FORMULA:
 FOR ALL x IN '//num'
 FOR ALL y IN '//num'
 ( int(x) = int(y) -> x = y )
}
```

Interpretace: pro všechny dvojice uzlů `num` musí platit, že jsou-li jejich hodnoty stejné, pak se musí jednat o tytéž uzly (stejné místo v dokumentu).
Uvažme nyní následující data:

```
<cisla>
  <num>1</num>
  <num>1</num>
</cisla>
```

V případě, že proměnná `x` momentálně reprezentuje první značku `num` a proměnná `y` druhou značku `num`, platí že `int(x) = int(y)`, ale neplatí `x = y` (`x` i `y` představují různé uzly v dokumentu). Z tohoto důvodu je třeba rozlišovat porovnávání uzlů (proměnných bez konverze) a porovnávání jejich hodnot (číselných, stringových).

4.4 Komentáře

V souboru s podmínkami lze používat jednořádkové komentáře, uvozené znakem `##`. Veškeré další znaky až do konce řádky jsou ignorovány. Víceřádkový komentář začíná `/*` a končí `*/`. Veškeré znaky uvnitř jsou ignorovány. Víceřádkové komentáře mohou být rovněž vnořené.

```
/* podminka pro HTML dokument:
   /* odkazy jsou neprazdne */ */
CONSTRAINT "odkazy" {
FORMULA:          ## zapis formule v prenexnim tvaru
 FOR ALL a in '//a/@href'
 ( length(str(a)) != 0 ) ## predikat
}
```

4.5 XML formát podmínek

Představený jazyk Incox se hodí pro ruční psaní podmínek, protože kopíruje přirozený jazyk. V některých případech by ale mohla být užitečná možnost zapsat podmínky strukturovaně v XML. Z tohoto důvodu bude validátor podporovat vstupy nejen v textové formě v jazyce Incox, ale také v jeho XML ekvivalentu.

Kořenový element `incox` zavádí namespace `incox`, a to buď implicitně atributem `xmlns` nebo explicitně s tím, že všechny příslušné elementy jsou dále uvozeny určeným prefixem.

Element `incox` může obsahovat podelement `globals` a musí obsahovat podelement `constraints`. Element `globals` obaluje možné nepovinné deklarace konstant, v `constraints` se nacházejí samotné podmínky.

```
<?xml version="1.0" encoding="utf-8"?>
<incox xmlns="incox">
  <globals>  ...  </globals>
  <constraints> ...  </constraints>
</incox>
```

Podelementy elementu `globals` mohou být v různém počtu i pořadí elementy `const`, `enum`, `interval` a `namespace`. První tři určují konstanty, použitelné v celém programu. Stejně jako v textové verzi platí, že je-li místo hodnoty použit identifikátor konstanty, musí být v době překladu již hodnota této konstanty známa, tedy její deklarace musí i v XML předcházet použití. Taktéž, je-li kdekoliv v XPath výrazu používán prefix, musí být již tento prefix dříve zaveden.

```
<globals>
  <const>
    <name>sto</name>
    <value>50+50</value>
  </const>
  <interval>
    <name>od1do100</name>
    <begin>1</begin><end>sto</end><step>1</step>
  </interval>
  <enum>
    <name>jmena</name>
    <values>
      <value>"Petr"</value><value>"Pavel"</value>
    </values>
  </enum>
  <namespace>
    <prefix>ic</prefix>
    <uri>incox</uri>
  </namespace>
</globals>
```

Element `constraints` obsahuje jeden či více podelementů `constraint`, z nichž každý reprezentuje jednu podmínku. Přímými podelemety `constraint` jsou povinný element `name`, který určuje jméno podmínky a dále nepovinný `parameters` a povinný `formula`.

V rámci elementu `formula` musí být povinně uvedeny `selects` a `predicate`. Uvnitř `selects` se pak jednotlivými elementy `select` určují kvantifikátory, proměnné a elementem `in` příslušné množiny. Kvantifikátory reprezentují elementy `forall`, `exists`, `exists1` a `for`. Poslední jmenovaný musí mít alespoň jeden z atributů `atleast` nebo `atmost`. Obsahem těchto elementů je název proměnné, v elementu `in` je pak uvedena množina (jméno konstanty typu množina nebo XPath výraz, vracející uzly), která se bude touto proměnnou procházet.

```
<constraints>
  <constraint>
    <name>jmeno podminky</name>
    <parameters>
      <fuzzy accuracy="4" >on</fuzzy>
      <counts>on</counts>
    </parameters>
    <formula>
      <selects>
        <select>
          <for atleast="9%">c</for><in>'//c'</in>
        </select>
        <select>
          <exists>b</exists><in>'$c/b'</in>
        </select>
      </selects>
      <predicate>
            int(b) = 1
      </predicate>
    </formula>
  </constraint>
</constraints>
```

Formální popis vstupního XML s podmínkami určuje schema `IncoxInput.xsd`, uvedené v příloze B.

4.6 Fuzzy pravdivost

V případě více podmínek v souboru je každá vyhodnocena zvlášť a toto hodnocení vráceno jako výstup aplikace. Každou podmínku jde v základní verzi vyhodnotit na `TRUE` (platí) nebo `FALSE` (neplatí). V případě, že je predikát významově chybný, může být výsledkem ještě hodnota `INVALID` (nevyhodnotitelné).

Někdy by se nám ovšem mohly hodit ještě další dodatečné informace o tom, „jak moc je podmínka splněna resp. porušena“. Například pokud požadujeme po každém elementu daného jména určité vlastnosti a v obrovském dokumentu jej

pouze jeden element porušuje, je podmínka vyhodnocena jako `FALSE`. Prakticky by nám ale taková chyba nemusela při dalším zpracování vždy vadit (například jde-li o data určená k statistické analýze).

4.6.1 Počítání fuzzy pravdivosti

V případě více výběrů, tedy více různě proložených klauzulí `FOR { ALL | AT LEAST | AT MOST }` a `EXISTS[!]`, může být celková podmínka a provázanost proměnných značně komplikovaná, a aby měl celkový výstup smysl (byl pochopitelný a případně dále programově zpracovatelný), je potřeba doplňkové informace exportovat v jednoduchém a kompaktním formátu.

Proto například nemá smysl vypisovat tabulky pravdivostních hodnot (kolik jakých proměnných jak odpovídalo určitým podmínkám), ale spíše se nabízí dodatečný výstup ve formě jediného čísla. To může nabývat hodnot v rozmezí `[0,1]` a alternativně tak poskytuje doplňkové informace o míře pravdivosti k základnímu vyhodnocení, které je naopak pouze buď `1` nebo `0`. Toto číslo z `[0,1]` je samozřejmě vázáno na první použitý kvantifikátor v podmínce.

Fuzzy pravdivost je v případě všech kvantifikátorů určena procentuálním podílem prvků, u kterých je podmínka splněna. Pro motivační příklad s jedním porušujícím záznamem a použitým kvantifikátorem `FOR ALL` by tak výstup mohl vypadat například `(0; 0,99)`. To odráží stav, kdy podmínka sice splněna není (`0`), ale to jenom kvůli velmi malé části testovaných elementů `(1%)`.

Získání této informace lze u podmínky vynutit nastavením parametru `FUZZY` na hodnotu `ON`.

4.6.2 Konkrétní počet vyhovujících

Další užitečnou částí výstupu by jistě mohlo být i vypsání konkrétního počtu vyhovujících prvků a počtu všech. V předchozím případě tedy například `99/100`. Taková forma doplňující infromace je zřejmě mnohem užitečnější pro kvantifikátory `EXISTS` a `EXISTS!`, kdy nás může zajímat nejen to, zda určený prvek vůbec existuje, ale také kolik jich opravdu je.

Vypsání počtů vyhovujících a všech prvků pro množinu prvního kvantifikátoru lze u konkrétní podmíky vynutit nastavením parametru `COUNTS` na hodnotu `ON`.

4.6.3 Volitelnost fuzzy pravdivosti a počtů

Je zřejmé, že v případě, kdy bychom chtěli fuzzy pravdivost nebo počty jako součást výstupu, naroste obecně počet kroků, které musí validátor vykonat. Například v případě podmínky `FOR ALL`, v momentě, kdy narazíme na první nevyhovující element, je již jasné, že celkový výstup bude `FALSE`. Validátor by tedy mohl ihned skončit, stejně jako v případě detekce prvního vyhovujícího elementu u kvantifikátoru `EXISTS`. Máme-li ovšem znát procento, potažmo počet vyhovujících, musí se i přes zřejmý celkový výsledek projít skutečně všechny vybrané elementy, což bude obecně celý proces validace zpomalovat.

Z tohoto důvodu se obě doplňkové informace implicitně nepočítají a lze je vynutit pouze explicitně zadáním příslušných parametrů.

4.7 Konverzní chyby u proměnných

Uvažme dokument, pro jehož některý element není přesně určen typ dat, nebo je určen tak, že mohou být jak numerické, tak řetězcové povahy. V případě číselné hodnoty je chceme jako čísla zpracovávat, v případě znakové buď ignorovat nebo považovat za chybná, případně zpracovávat jinak. Za stávajících podmínek by takový přístup vyvolal běhovou chybu, jako v následujícím příkladě.
Příklad uvažovaného dokumentu:

```
<cisla>
 <num>1</num>
 <num>chyba</num>
 <num>16</num>
 <num></num>
 <num>-8</num>
</cisla>
```

Podmínka, která vyvolá běhovou chybu:

```
CONSTRAINT "vsechna kladna?" {
FORMULA:
 FOR ALL x IN '//num'
 ( int(x) > 0 )
}
```

Hodnota „`chyba`" a prázdná hodnota (odpovídající prázdnému elementu) nejdou přetypovat na celé číslo. Takové chování by nám ale nemuselo vždy vyhovovat, protože by bylo znemožněno jakékoliv pokročilejší testování a validování elementů s nehomogenními datovými typy.

Chování v těchto případech navíc nelze nastavit pevně. Někdy budeme chtít nepřetypovatelné hodnoty ignorovat (například prázdný element bez hodnoty jakoby nebyl), jindy budeme takové elementy považovat za narušení požadovaného integritního omezení (tam, kde má být číslo, musí být číslo) a nezřídka je budeme chtít testovat na jiné podmínky (je-li to řetězec a ne číslo, musí to být „null").

Z těchto důvodů lze pro každou podmínku definovat parametr `ONERROR`, který může nabývat buď hodnot `TRUE` nebo `FALSE`. Dle toho je v případě konverzní chyby při použití proměnné automaticky za daný běh vygenerována hodnota buď `TRUE` nebo `FALSE`.

Tento výsledek se však týká pouze konverzí, ve kterých figuruje proměnná. Například v případě chybné konverze `int("slovo")` bude chyba vyvolána vždy, i při použití parametru `ONERROR`.

```
CONSTRAINT "vsechna kladna?" {
ONERROR: TRUE
FORMULA:
 FOR ALL x IN '//num'
 ( int(x) > 0 )
}
```

V našem příkladě bude tedy výstup při použití `ONERROR: TRUE` roven `FALSE`. Nezkonvertovatelné hodnoty elementu `num`, tedy prázdný element a řetězec `„chyba"` se nuceně vyhodnotí na `TRUE`, celkově ale podmínka platit stejně nebude, protože ji porušuje typově správná hodnota `-8`.

4.8 Pomocné funkce pro predikát a konstanty

Pro plnohodnotné testování podmínek obsahuje jazyk následující funkce, jejichž argumenty a návratové hodnoty jsou pro srozumitelnost uvedeny ve stylu jazyka C.

- `bool not(bool b)`
  Neguje libovolnou podmínku, u které lze získat její boolovskou hodnotu.
- `bool empty(xpath xp)`
  Vrací `TRUE` v případě, že zadaný XPath výběr je prázdný, `FALSE` naopak. Použitý XPath výraz musí vracet množinu prvků, ne hodnotu.
- `string str(expr e)`
  `int int(expr e)`
  `real real(expr e)`
  Funkce zkonvertují výraz `e` na řetězcovou/celočíselnou/reálnou hodnotu. Výraz `e` může být identifikátor konstanty nebo proměnné, řetězec, číslo nebo XPath výraz.

  Pokud XPath výraz vrací množinu uzlů, musí být tato množina jednoprvková (výsledkem je pak zkonvertovaná hodnota tohoto jednoho uzlu). V opačném případě je vyvolána běhová chyba.
- `int length(string s)`
  Vrací celé číslo, určující délku řetezce (počet znaků).
- `string tolower(string s)`
  Převede všechna velká písmena v řetězci na malá.
- `string toupper(string s)`
  Převede všechna malá písmena v řetězci na velká.
- `string trim(string s)`
  Odstraní bílé znaky ze začátku a konce řetězce.
- `string trimall(string s)`
  Odstraní bílé znaky z řetězce.
- `bool match(string s, string regexp)`
  Vrátí, zda řetězec `s` obsahuje podřetězec, odpovídající zadanému regulárnímu výrazu `regexp`.

Kapitola 5

Incox v kontextu ostatních schema jazyků

Jazyk Incox je podobně jako Schematron či CliX určen především pro popis sémantických závislostí v rámci jednoho či více XML dokumentů. Jeho cílem tedy není popis struktury těchto dokumentů, ačkoliv i to by bylo – byť těžkopádně a zdlouhavě - s využitím XPathu a jeho funkcí teoreticky možné.

Podobně jako u Schematronu si využití Incoxu můžeme představit hlavně ve spojení s nějakým klasickým schema jazykem, typicky XML Schema či Relax NG. V schema jazyce bude popsána struktura a případně ověřeny datové typy. Druhý krok validace pak bude spočívat v ověření sady sémantických podmínek, zapsaných v Incoxu. Namísto srovnávání Incoxu a XML Schema nebo Relax NG se tedy nabízí spíše srovnání Incoxu se Schematronem, resp. CliXem. Tyto jazyky hrají totiž v procesu validace XML dokumentu stejnou roli.

5.1 CliX a Incox

Jazyk Incox vychází ze stejných myšlenek jako CliX, tedy z využití predikátové logiky ve spojení s XPathem. Proto je jeho výchozí teoretická síla stejná. Jazyk Incox ale nabízí ještě další konstrukty, které jeho možnosti značně rozšiřují, především:

- **konstanty[2] a konstantní množiny, možnost jejich využití v podmínce**

Uvažme například situaci, kdy nás zajímá, zda v XML dokumentu s knihou nechybí (resp. ani není duplikována) žádná její kapitola. V logice prvního řádu je tato podmínka vyjádřitelná způsobem „pro všechna přirozená čísla, menší než číslo nejvyšší kapitoly, existuje (právě jedna) kapitola“.

Množinu, pro jejíž prvky bychom chtěli podmínku testovat (přirozená čísla 1 až max), nemáme ovšem v CliXu jak vyjádřit. Množiny lze totiž určovat jen XPath výběry. S využitím konstant a konstantních množin, určených jinak než XPathem, lze ovšem v Incoxu podmínku zapsat následovně:

[2] V CliXu existuje jako určitý slabší ekvivalent konstrukt variable, umožňující pojmenovat XPath výraz pro globální použití

```
CONST  maxChap =
int('/kniha/kapitola[not(../kapitola/@cislo > @cislo)]/@cislo')
INTERVAL chapNums = (1, maxChap, 1)

CONSTRAINT "chybejici kapitoly" {
FORMULA:
 FOR ALL chap IN chapNums
  EXISTS ! rec IN '/kniha/kapitola'
  ( int('$rec./@cislo') = chap )
```

Nejdříve pomocí XPathu vybereme číslo nejvyšší kapitoly, zkonvertujeme na int a uložíme jej do konstanty `maxChap`. Předpokládáme, že každé číslo kapitoly se může objevit jen jednou. V případě, že bychom si jisti nebyli, lze výběr upravit takto:

```
CONST maxChap =
int('/kniha/kapitola[not(../kapitola/@cislo > @cislo)][1]/@cislo')
```

Je-li nejvyšších kapitol více, vezme se jen jedno číslo a konverze bude moci proběhnout. Podmínka nakonec skončí `FALSE` kvůli klauzuli `EXISTS!`, kdy pro číslo nejvyšší kapitoly bude existovat více záznamů.

Dále deklarujeme interval `chapNums`, který obsahuje všechny celočíselné hodnoty od 1 do čísla nejvyšší kapitoly. V samotné podmínce pak tento interval procházíme proměnnou `chap` a hledáme v XML vždy právě jednu takovou kapitolu, jejíž číslo bude rovno aktuální hodnotě.

Tento příklad lze spolu s testovacími daty nalézt na přiloženém CD v adresáři `examples/kapitoly`.

Konstanty se tedy hodí nejen pro zpřehlednění a zefektnění testování, ale spolu s konstrukty `ENUM` a `INTERVAL` i pro testování podmínek typu „pro všechny zadané hodnoty existuje záznam / určitý počet záznamů“. Užitečné je to především v situacích, kdy ony testované hodnoty nemáme vyjádřené jinde v XML nebo by takové vyjádření bylo nepraktické (dlouhé posloupnosti čísel).

- **nové kvantifikátory FOR { AT LEAST | AT MOST } a EXISTS!**

Již v předchozí podmínce byl využit kvantifikátor `EXISTS!` (existuje právě jeden), Incox vedle něj ale nabízí i kvantifikátor `FOR { AT LEAST | AT MOST }`, blíže specifikovaný v kapitole 4.3.1. Díky jeho vyjadřovací síle dokáže zastoupit i ostatní kvantifikátory, jejich použití je ovšem zase přehlednější.

```
FOR ALL x   =  FOR AT LEAST 100% x
EXISTS y    =  FOR AT LEAST 1 y
EXISTS ! z  =  FOR AT LEAST 1, AT MOST 1 z
```

Důvody pro zavedení možnosti specifikace konkrétního počtu uzlů nebo % uzlů, pro které má podmínka platit, byly diskutovány v kapitole 4. Využití kvantifikátoru `FOR { AT LEAST | AT MOST }` si lze představit především při testování dat, u kterých typicky tolerujeme určitou „chybu“ (určité části vybraných uzlů dovolujeme podmínku nesplňovat), chceme však mít její velikost pod kontrolou.

- **vlastní funkce, například testování oproti regulárnímu výrazu**

Funkce pro porovnání hodnoty uzlu s regulárním výrazem může mnohdy posloužit jako náhrada místo příslušného datového typu. Například následující podmínka testuje, zda je hodnota elementu korektní římské číslo. Příklad lze spolu s testovacími daty opět nalézt i na přiloženém CD, a to v adresáři `examples/strfnc`.

```
CONSTRAINT "test rimska cisla" {
FORMULA:
 FOR ALL r IN '//romnum'
 ( match(trim(str(r)),
   "^m*(d?c{0,3}|c[dm])(l?x{0,3}|x[lc])(v?i{0,3}|i[vx])$")
 )
}
```

Kompletní seznam funkcí Incoxu je k dispozici v kapitole 4.8, samozřejmě je v části predikátu možno využívat i funkce XPathu 1.0.

5.2 Schematron a Incox

Incox se vůči Schematronu vymezuje minimálně stejně tak jako CliX, který oproti němu umožňuje popis složitějších strukturálně-logických vazeb, jak bylo naznačeno v kapitole 3.5. Protože je Schematron ve své kategorii poměrně uznávaný, bude jistě zajímavé zkusit na abstraktní úrovni naznačit, jak vyjádřit podmínku v Schematronu ekvivalentně také v Incoxu.

Síla Schematronu spočívá ve využívání XPathu, což je také jeden ze základních stavebních kamenů Incoxu. Oba jazyky mají tedy stejné možnosti, co se týče vyjadřovací síly pro dílčí výběry testovaných uzlů.

Pokud se podíváme znovu na proces zpracování podmínky v Schematronu, uvidíme v jazyce logiky prvního řádu určitou analogii s logickou formulí „pro všechny uzly platí podmínka: XPath výraz je neprázdný / vrací true“. Přesně to nakonec i podmínka v Schematronu říká. Pro všechny uzly výběru daného atributem `context` elementu `rule` se testuje, zda platí podmínka daná atributem `test` u elementu `report/assert`. Obecná podmínka v Schematronu má tedy základní tvar (pro `assert` obdobně):

```
<pattern name="name">
  <rule context="context">
    <report test="test">
        Splněno.
    </report>
  </rule>
</pattern>
```

V Incoxu lze stejnou podmínku vyjádřit následovně, pokud je `test` XPath výraz, vracející boolovskou hodnotu (používá boolovské XPath funkce nebo operátory):

```
CONTRAINT "name" {
FORMULA:
 FOR ALL x in 'context'
 ( str('$x~test') = "True" )
}
```

a nebo takto, pokud je `test` XPath výraz, vracející množinu uzlů:

```
CONTRAINT "name" {
FORMULA:
 FOR ALL x in 'context'
 ( not(empty('$x~test')) )
}
```

Vztažení testovací podmínky `test` k aktuálnímu uzlu výběru `context`, které je schematicky naznačeno jako `$x~test`, musí být učiněno s ohledem na syntaxi XPathu.

Například, je-li `context` procházen proměnnou `x` a hodnota `test` je rovna `sum(//Percent)`, pak výraz `$x~test` odpovídá použití `sum($x//Percent)` namísto mechanického `$x/sum(//Percent)`.

Pokud je v původním XPath výrazu `test` použita funkce `current()`, odkazující se na aktuálně testovaný prvek z množiny dané `context@rule`, musí být tato funkce nahrazena referencí na příslušnou proměnnou výběru. Sémanticky je tato náhrada přirozená. Například z `current()/@id` se stane `$x/@id`.

Příklady, uvedené v sekci 3.2, demonstrující možnosti Schematronu, by tak šly v Incoxu podle předchozího návodu na přepis vyjádřit následujícím způsobem. Na CD se nacházejí v adresáři `examples/sumcount`.

Součet hodnot všech vybraných elementů je určité číslo
Součet hodnot všech podelementů Percent elementu Total je dohromady 100:

```
CONSTRAINT "Suma je 100%" {
FORMULA:
 FOR ALL x in 'Total'
 ( str('sum($x//Percent) = 100') = "True" )
}
```

Ačkoliv s ohledem na možnosti Incoxu by byl mnohem přirozenější zápis:

```
CONSTRAINT "Suma je 100% #2" {
FORMULA:
 EXISTS x in 'Total'
 ( int('sum($x/Percent)') = 100 )
}
```

Dokument obsahuje stejný počet zvolených elementů
Počet elementů AAA a BBB je v celém dokumentu stejný.

```
CONSTRAINT "Test počtu elementů" {
FORMULA:
 FOR ALL x in '/*'
 ( str('count(//BBB) = count(//AAA)') = "True" )
}
```

V tomto příkladě jsme zohlednění kontextu (`x` ve výběru `FOR ALL`) mohli v sekci predikátu dokonce úplně vynechat (`x` vybírá kořen, tudíž `'$x//BBB'` vybírá stejné uzly jako `'//BBB'`). Podobně by šel samozřejmě upravit i první příklad. V tomto případě navíc také existuje o něco přirozenější ekvivalent zápisu:

```
CONSTRAINT "Test počtu elementů #2" {
FORMULA:
 EXISTS x in '/*'
 ( int('count(//BBB)') = int('count(//AAA)') )
}
```

Příklad: Mini-Schema for Schematron
Část příkladu mini schematu pro Schematron z úvodu kapitoly o Schematronu:

```
<rule context="sch:schema">
  <assert test="sch:pattern">
        A schema contains patterns.
  </assert>
</rule>
```

by se pak dala do Incoxu přepsat podle druhého konstruktu (`test` v tomto případě testuje přítomnost určitého uzlu):

```
CONSTRAINT "Schematron Mini Schema" {
NAMESPACE sch="http://purl.oclc.org/dsdl/schematron"
FORMULA:
 FOR ALL x in 'sch:schema'
 ( not(empty('$x/sch:pattern')) )
}
```

Zatímco Schematron při detekci splněné či nesplněné podmínky vypisuje uživatelský text, výstupem u Incoxu jsou hodnoty `TRUE` a `FALSE`. Informační hodnota je ale v obou případech stejná. Je-li v rámci jednoho kontextu testováno více podmínek (více `assert`/`report` elementů v rámci jednoho elementu `rule`), lze tento konstrukt přepsat ekvivalentně do Incoxu rozepsáním do více dílčích podmínek.

Například v druhém uvedeném příkladě jsme v Schematronu s výstupem získali informaci nejen o tom, zda počet `AAA` je roven počtu `BBB`, ale pokud ne, tak jsme se také dozvěděli, kterých elementů je víc. V Incoxu bychom to ekvivalentně vyjádřili rozepsáním do tří podmínek `#AAA = #BBB`, `#AAA < #BBB` a `#AAA > #BBB`, z nichž by vždy právě jedna byla `TRUE`.

Kapitola 6

Uživatelská dokumentace

6.1 Program Icval

Program Icval (Integrity Constraints Validator) je implementace jazyka Incox pro Windows. Icval je konzolová aplikace napsaná v jazyce C#. Pro svůj běh potřebuje .NET Framework 2.0.

Icval není potřeba instalovat, stačí do stejného adresáře jako `icval.exe` umístit také soubory `IncoxInput.xsd` a `XmlToIncox.xsl`. Všechny uvedené soubory se nachází přímo v kořenovém adresáři na přiloženém CD.

Program se spustí příkazem `icval` na příkazovém řádku. Syntaxe je následující:

```
icval CONSTRAINTS FILE.xml [FILE.xml ... ] [options]
```

`CONSTRAINTS` – Soubor s podmínkami, zapsanými buď přímo v jazyce Incox, nebo v jeho XML ekvivalentu. Detekci typu vstupu provádí validátor automaticky. Pokud jsou vstupní podmínky zapsány v XML, musí toto XML odpovídat schematu `IncoxInput.xsd` (příloha B). Doporučeno je kódování UTF-8.

`FILE.xml` – Soubor(y) s daty, která chceme testovat. Odkazujeme-li se v podmínkách číslem na různé soubory, je jejich pořadí dáno pořadím při volání, přičemž číslujeme od 1. Povinný je alespoň jeden soubor.

```
Options:
  -v        verbose output
  -f [prec] fuzzy truth [precision]
  -c        counts
  -x        XML output
  -t        temporary Incox TXT file not deleted
  -s        silent mode for results
  -e [enc]  encoding for output text/XML

 (Error reporting modes):
  -ef   full mode (default)
  -es   silent mode
  -ew   no warnings mode
  -ec   compact mode
```

Nastavení podrobněji

Parametry z řádku se zohledňují, není-li u podmínek nastavení explicitně jiné. Nastavení pomocí parametrů přímo u podmínky tedy překrývá nastavení z příkazové řádky.

- `-v`: Delší textový výstup, určený pro čtení člověkem.
- `-f [prec]`: Fuzzy pravdivost ve formě reálného čísla z intervalu [0,1], která určuje poměr vyhovujících prvků ku všem pro množinu prvního kvantifikátoru. Nepovinné celé číslo `prec` (0 až 15) určuje maximální počet desetinných míst, na která bude fuzzy pravdivost vypsána (defaultně 3). Je-li parametr uveden víckrát, platí poslední nastavení.
- `-c`: Vypsání počtu vyhovujících a počtu všech prvků množiny prvního kvantifikátoru.
- `-x`: XML výstup. Formát XML odpovídá schematu `IncoxOutput.xsd` (Příloha C).
- `-t`: Pomocný textový soubor s ekvivalentem XML vstupu v Incoxu (výsledek XSL transformace) nebude smazán. Má smysl pouze jsou-li podmínky zapsány v XML verzi Incoxu.
- `-s`: Vypíší se pouze ty podmínky, které nebyly splněny (`FALSE`) nebo nemohly být vyhodnoceny z důvodu chyb (`INVALID`).
- `-e [enc]`: Nastaví kódování `enc` pro výstupní text nebo XML. Není-li zadáno, nastaví se defaultní systémové kódování.

Režimy výpisu chyb

Error reporting modes se týkají vypisování sémantických chyb a varování. Syntaktické chyby jsou vypsány vždy a mají za následek ukončení procesu zpracovávání podmínek. Přepínače pro error reporting lze používat pouze výlučně, nelze je kombinovat. V případě uvedení více těchto přepínačů je brán jako platný poslední z nich.

- `-ef`: Všechny chyby i varování jsou vypisovány (defaultní nastavení).
- `-es`: Žádné chyby ani varování nejsou vypisovány.
- `-ew`: Varování nejsou vypisována, chyby ano.
- `-ec`: Chyby i varování jsou vypisovány v kompaktní formě. Pokud je kvůli substituci hodnot za proměnné na určitém místě vygenerováno více chyb téhož druhu, nevypisuje se tato chyba tolikrát, kolikrát opravdu nastala. Místo toho se vypíše jen jednou, s tím, že je u ní pouze číselně poznamenáno, kolikrát by měla být skutečně vypsána.

Poznámka k režimu vyhodnocování

V této implementaci jazyka není zavedeno zkrácené vyhodnocování výrazů. Výraz v predikátu je tedy i v případě `OR`, `AND` a `->` vyhodnocován celý.

6.2 Příklady výstupů v závislosti na parametrech

V následujícím příkladu budou používany soubory `centers.txt` a `centers.xml`, dostupné na přiloženém CD v adresáři `examples/centers`.

`Centers.xml` (192kB) je soubor s daty, která chceme testovat. V tomto případě se jedná o seznam firem, smluvních partnerů pro poskytování určité služby. Položky seznamu vypadají následovně:

```
<?xml version="1.0" encoding="utf-8" ?>
<CENTERS>
  <CENTER id="437415" city="Zlín" address="Nám. Práce 2512"
          name="XDIGITAL Zlín" zip="762 70" club="FALSE" />
  <CENTER id="420129" city="Praha 2" address="Plavecká 8"
          name="Tabák" zip="120 00" club="TRUE" />
     ...
</CENTERS>
```

Obsah souboru s podmínkami `centers.txt`:

```
ENUM vybrana_mesta= ("BRNO", "HRADEC KRÁLOVÉ", "LIBEREC", "PLZEŇ")

## alespoň 10% všech partnerů poskytuje též výhody členům klubu
CONSTRAINT "alespoň 10% klubů" {
ONERROR: FALSE
FORMULA:
  FOR AT LEAST 10% pobocka in '//CENTER'
    ( str('$pobocka/@club') = "TRUE")
}

## v Praze se nachází nejvýše 200 smluvních partnerů
CONSTRAINT "max 200 v Praze" {
FORMULA:
  FOR AT MOST 200 pobocka in '//CENTER'
   ( match( str('$pobocka/@city'), "Praha \d+$") )
}

/* v každém městě z výčtu vybrana_mesta
   je alespoň jeden smluvní partner */

CONSTRAINT "pokrytí vybraných měst" {
FORMULA:
  FOR ALL mesto IN vybrana_mesta
    EXISTS pobocka IN '//CENTER'
      ( trim(toupper(str('$pobocka/@city'))) = mesto )
}
```

Nyní ukážeme několik výstupů v závislosti na použitých parametrech Icvalu. Předpokládáme, že soubory `centers.txt` i `centers.xml` jsou umístěny v aktuálním adresáři.

- **:\> icval centers.txt centers.xml**

```
Constraint "alespoň 10% klubů" : (1)
Constraint "max 200 v Praze": (0)
Constraint "pokrytí vybraných měst" : (1)
```

Číslo v závorce vyjadřuje celkové vyhodnocení (`1=TRUE`, `0=FALSE`, `X=INVALID`).

- **:\> icval centers.txt centers.xml -c**

```
Constraint "alespoň 10% klubů" : (1; 401/1679)
Constraint "max 200 v Praze": (0; 252/1679)
Constraint "pokrytí vybraných měst" : (1; 4/4)
```

Za celkovým vyhodnocením následuje počet vyhovujících elementů / počet všech elementů množiny. Vidíme tedy nejen to, že požadavek na maximum 200 poboček v Praze není splněn, ale navíc zjistíme, že v Praze je 252 poboček (z celkového počtu 1679).

- **:\> icval centers.txt centers.xml -f**

```
Constraint "alespoň 10% klubů" : (1; 0,239)
Constraint "max 200 v Praze": (0; 0,15)
Constraint "pokrytí vybraných měst" : (1; 1)
```

Za celkovým vyhodnocením následuje číslo, vyjadřující podíl vyhovujících prvků ku všem prvkům v množině prvního kvantifikátoru. Vidíme tedy nejen to, že požadavek na alespoň 10% klubů je splněn, ale rovněž se dozvíme, že kluby tvoří 23,9% všech poboček.

- **:\> icval centers.txt centers.xml -c -f -s**

```
Constraint "max 200 v Praze": (0; 252/1679; 0,15)
```

Jsou-li parametry `-c` a `-f` použity souběžně, vypisují se po celkovém výsledku nejdříve počty, poté podíl. V tomto případě jsme navíc parametrem `-s` zadali, že chceme zobrazovat pouze ty podmínky, které celkově splněny nebyly, což je v tomto případě pouze jediná.

- **:\> icval centers.txt centers.xml -v -c -f**

```
CONSTRAINT: "alespoň 10% klubů"
-------------------------------------------------
OVERALL RESULT : TRUE
Conversion errors resolved as FALSE
True/All for quantifier FOR AT LEAST  : 401/1679
Fuzzy truth: 0,239
-------------------------------------------------
CONSTRAINT: "max 200 v Praze"
-------------------------------------------------
OVERALL RESULT : FALSE
Conversion errors resolved as INVALID
True/All for quantifier FOR AT MOST  : 252/1679
Fuzzy truth: 0,15
-------------------------------------------------
CONSTRAINT: "pokrytí vybraných měst"
-------------------------------------------------
OVERALL RESULT : TRUE
Conversion errors resolved as INVALID
True/All for quantifier FOR ALL : 4/4
Fuzzy truth: 1
-------------------------------------------------
```

Z delšího textového výstupu (`-v`) se dozvíme také informace o tom, jak jsou pro danou podmínku vyhodnocovány konverzní chyby proměnných (odpovídá nastavení parametru `ONERROR` přímo u dané podmínky). Počty a podíly byly vypsány díky parametrům `-c` a `-f`.

- **:\> icval centers.txt centers.xml -x -e utf-8**

```
<?xml version="1.0" encoding="utf-8" ?>
<icval xmlns="incox">
  <global_errors />
  <constraints>
    <constraint>
      <local_errors /><name>alespoň 10% klubů</name>
      <overall_result>1</overall_result>
    </constraint>
    <constraint>
      <local_errors /><name>max 200 v Praze</name>
      <overall_result>0</overall_result>
    </constraint>
    <constraint>
      <local_errors /><name>pokrytí vybraných měst</name>
      <overall_result>1</overall_result>
    </constraint>
  </constraints>
</icval>
```

XML výstup (`-x`) je vypsán v požadovaném kódování (`-e`) UTF-8 a odpovídá schématu `IncoxOutput.xsd` (Příloha C). Parametr `-x` lze rovněž kombinovat s parametry `-c`, `-f`, `-s` a speciálně také s parametrem `-t`.

6.3 Další příklady použití programu

V adresáři `examples` na přiloženém CD se nachází několik dalších podadresářů s ukázkami podmínek i XML dat, na kterých lze tyto podmínky testovat. Jeden podadresář odpovídá jednomu příkladu.

V každém podadresáři se kromě uvedených souborů nachází též samostatně spustitelný soubor `*.bat`, který aplikaci Icval na dané podmínky a data zavolá a výsledek zobrazí na konzoli.

Obsah adresáře examples je následující:

- **/centers**

Podmínky a data z předchozí kapitoly.

Podmínky	**XML data**
centers.txt	centers.xml

- **/countries**

Ukázky podmínek na skutečných XML datech.

Podmínky	**XML data**
countries.txt	countries.xml

- **/dvojice**

Test, zda jsou všechna čísla uvedená alespoň nebo právě po dvou. Různé varianty demonstrují použití deklarace `namespace` v závislosti na tom, zda a jak XML soubor s daty využívá namespacy (bez namespace, defaultní, s prefixem, více namespaců a více prefixů).

Podmínky	**XML data**
dvojice1.txt	dvojice1.xml
dvojice2.txt	dvojice2.xml
dvojice3.txt	dvojice3.xml
dvojice4.txt	dvojice4.xml

- **/empty**

Chování na různých kvantifikátorech, nevrátil-li zadaný XPath dotaz pro určení výběru žádné uzly.

Podmínky	**XML data**
empty.txt	empty.xml

- **/errors**

Demonstrace výstupů v případě nejrůznějších sémantických chyb.

Podmínky	**XML data**
errors.txt	errors.xml

- **/kapitoly**

Test, zda nechybí ani nepřebývá žádná kapitoly knihy (viz 5.1). Podmínka je zapsána také v XML verzi Incoxu. Jedotlivé XML soubory s podmínkami se liší zacházením s namespace `incox` (není uvedeno, je defaultní, je uvedeno s prefixem a ten je dále používán).

Podmínky	**XML data**
kapitoly.txt	kapitoly_ok.xml
kapitoly_constraint1.xml	kapitoly_navic.xml
kapitoly_constraint2.xml	kapitoly_chybi.xml
kapitoly_constraint3.xml	

- **/quantifiers**

Podmínky kombinující více kvantifikátorů.

Podmínky	**XML data**
quantifiersT.txt	quantifiers.xml
quantifiersF.txt	

- **/spec**

Příklady ze specifikace jazyka Incox (Kapitola 4) a podmínky pro demonstraci kvantifikátoru `FOR { AT LEAST | AT MOST }`.

Podmínky	**XML data**
spec.txt	spec.xml
for.txt	for.xml

- **/strfnc**

Podmínky, využívající funkce pro práci s řetězci (`toupper`, `tolower`, `trim`, `trimall`, `match`).

Podmínky	**XML data**
strfnc.txt	strfnc.xml

- **/strom**

Test na strom součtů. Součet hodnot všech uzlů v každém podstromu je roven hodnotě v kořeni tohoto podstromu; hodnota v listech může být libovolná.

Podmínky	**XML data**
strom.txt	strom_ok.xml
	strom_bad.xml

- **/sumcount**

Podmínky uvedené jako příklady v Kapitole 5.2.

Podmínky	**XML data**
sum.txt	sum_ok.xml
	sum_bad.xml
count.txt	count_ok.xml
	count_bad.xml

- **/typerr**

Chování při typové chybě z důvodu chybné konverze proměnné. Demonstrace parametru `ONERROR` (viz Kapitola 4.7).

Podmínky	**XML data**
typerr.txt	typerr.xml

Kapitola 7

Programátorská dokumentace

7.1 Zdrojové kódy a pomocné soubory

Projekt `icval.sln`, uložený na CD v adresáři `sources`, je určen pro MS Visual Studio 2005 a obsahuje soubory dle tabulky 7.2. Jde o kompletní zdrojové kódy v jazyce C#, jež jsou potřeba pro kompilaci programu Icval.

Podrobnou programátorskou dokumentaci zdrojových kódů lze najít v souboru `icval.chm`, nacházejícím se na přiloženém CD v adresáři `doc`. Tento soubor byl vygenerován ze speciálních komentářů ve zdrojových kódech pomocí volně dostupných aplikací Sandcastle [18], Sandcastle Help File Builder [19] a HTML Help Workshop [20].

Součástí projektu je dále i gramatika jazyka Incox, zapsaná ve formátu pro compiler generátor Coco/R [21]. Použitím programu Coco/R na soubor `incox.atg` (příloha A) se vygenerují soubory `Scanner.cs` a `Parser.cs`, které jsou nezbytné pro kompilaci programu Icval.

Schemata `IncoxInput.xsd` (Příloha B) a `IncoxOutput.xsd` (Příloha C), zapsaná v jazyce XML Schema, popisují namespace `incox`, do kterého patří všechny elementy ze vstupních souborů s podmínkami i výstupních souborů s vyhodnocením.

Soubor	**Stručný popis**
incox.atg	gramatika jazyka Incox pro compiler generátor Coco/R
Parser.frame	upravená část Coco/R pro generování Parser.cs
Scanner.frame	upravená část Coco/R pro generování Scanner.cs
IncoxInput.xsd	XML Schema pro vstupní XML s podmínkami
IncoxOutput.xsd	XML Schema pro výstupní XML s vyhodnocením
XmlToIncox.xsl	XSL pro transformaci z XML do Incoxu

Tabulka 7.1: Pomocné soubory

7.2 Detekce formátu vstupních podmínek

Vstupním místem programu je funkce `static void Main(string[] arg)` třídy `Compile` (`Program.cs`).

Je-li vstupem soubor s podmínkami, zapsanými v jazyce Incox, předá se tento soubor rovnou parseru. Jsou-li podmínky zapsány v XML, provede se nejdříve validace tohoto XML proti schematu `IncoxInput.xsd`. Pokud je XML validní, transformuje se podle `XMLToIncox.xsl` na obsahově ekvivalentní dočasný textový soubor v jazyce Incox, který se dále předá parseru. Uvedené akce zajišťují metody statické třídy `XmlIncoxHandler`, volané přímo z `Main`. Schematicky je přístup znázorněn na obrázku 7.1.

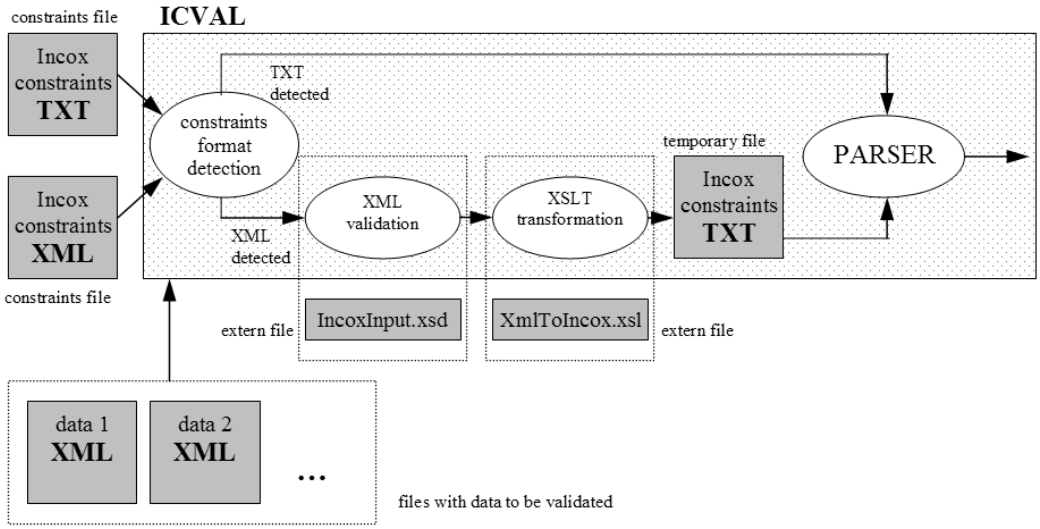
Soubor	**Významné třídy**	**Stručný popis**
Program.cs	Compile (Main), XmlIncoxHandler, Usage	detekce typu vstupu, případná validace a transformace XML, volání parseru
Scanner.cs	Buffer, Scanner	vygenerovaný lexikální analyzátor
Parser.cs	Parser	vygenerovaný syntaktický analyzátor
ParserSupport.cs	ParserSupport	zpracovávání a vyhodnocování podmínek
Value.cs	Token, Value, Oper, QVar	uchovávání hodnot
Quantifiers.cs	forall, forext, exists, existsone, qinvalid	kvantifikátory
ConstTable.cs	ConstTable	tabulka pro uchovávání globálních konstant (const, enum, interval)
Functions.cs	Functions	pomocné funkce pro predikát a konstanty
ErrorHandler.cs	Error, ErrorHandler	jednotné zpracování a vypisovní chyb

Tabulka 7.2: Zdrojové kódy projektu

7.3 Parsování souboru s podmínkami

V rámci sémantických akcí za běhu parseru (metoda `Parse()`) se pracuje s jeho atributy `ps` (objekt typu `ParserSupport`), `ct` (objekt typu `ConstTable`) a `eh` (objekt typu `ErrorHandler`)

- `ps`: informace o aktuálně vyhodnocované podmínce, vlastní vyhodnocování, XPath výběry a konverze, udržování informací o namespaces



Obrázek 7.1: Detekce a případná transformace vstupního souboru

- `ct`: informace o konstantách (`const`, `enum`, `interval`)
- `eh`: vypisování chyb a varování

Při parsování souboru s podmínkami se jinak zpracovávají konstantní deklarace (`const`, `enum`, `interval`) a jinak vlastní podmínky.

Konstantní deklarace

Konstanty, do kterých není přiřazen XPath výraz, jsou pomocí `ps.EvalExpr(..)` vyhodnocovány přímo během parsování a výsledky se ukládájí do `ConstTable ct`. Tato tabulka pro převod jména konstanty na hodnotu (instance třídy `Value`) je pak dále k dispozici při parsování všech podmínek.

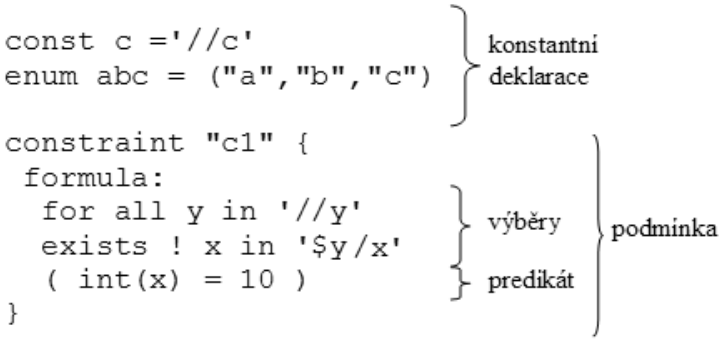
Konstanty typu `const c = 'xpath'` jsou uloženy jen textově (jaký XPath řetězec patří k jaké konstatně), ale reálně (skutečný dotaz do dokumentu) jsou vyhodnoceny až v momentě použití dané konstanty v podmínce. Je-li ovšem již na úrovni deklarace uplatněna konverze na tento XPath výraz, například `const c = str('xpath')`, pak je hodnota konstanty vyhodnocena okamžitě (a v tomto případě uložena jako typ řetězec).

Vlastní podmínky

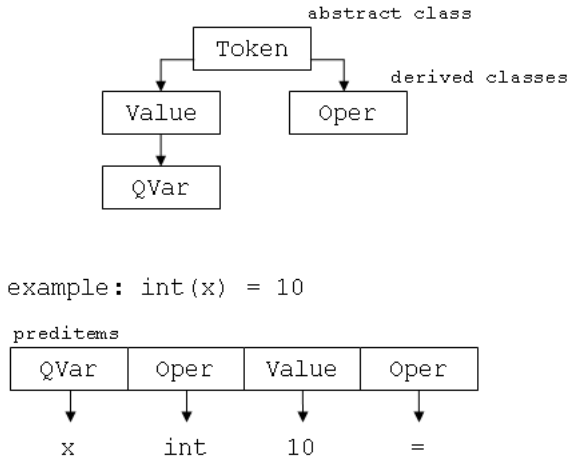
Během parsování jedné konkrétní podmínky jsou pouze naplňovány vnitřní struktury objektu `ps`. Po detekci konce podmínky se volá metoda `ps.Validate(..)`, která teprve zajistí skutečné vyhodnocení podmínky a vypsání výsledků.

- **Parsování a zpracovávání výběrů**

  Ze sekcí `FOR { ALL | AT LEAST | AT MOST }` nebo `EXISTS [!]` se do tabulky `ps.QVars` uloží proměnné a k nim příslušné množiny podle zadání. Identifikátory pro proměnné mají oblast platnosti pouze v rámci dané podmínky, tudíž je lze opakovaně definovat v různých podmínkách. Identifikátory proměnných se ovšem nesmí shodovat se jmény konstant, která mají platnost v rámci celého souboru.



Obrázek 7.2: Konstantní deklarace a podmínka



Obrázek 7.3: Typy tokenů v predikátu, příklad postfixové tokenizace

- **Parsování a zpracovávání predikátu**

  Zápis predikátu je tokenizován na hodnoty (`Value`), proměnné (`QVar`), konstanty (`Value`) a operátory (`Oper`), které jsou dále v postfixové notaci uloženy do `ps.preditems` (pole typu `Token`). V případě výskytu identifikátoru konstanty, je do `ps.preditems` uložena přímo její příslušná hodnota. V případě výskytu proměnné není zatím prováděna žádná substituce.

Je-li podmínka syntakticky v pořádku, volá se `ps.Validate()`. V případě sémantické chyby je podmínka „vyhodnocena“ jako `INVALID` a analyzátor přejde na podmínku následující. V případě syntaktické chyby vyhodnocování končí.

Pro každou další podmínku se tento proces opakuje. `ParserSuppor ps` má tedy vždy k dispozci data jen o jedné, aktuálně zpracovávané podmínce. Po jejím vyhodnocení tato data odstraní (`ps.Clear()`) a nahradí je těmi, která se vztahují k další podmínce.

7.4 Vyhodnocování podmínek

Za proměnné se postupně substituují konkrétní hodnoty z příslušných množin. Prozkoušení všech možných kombinací hodnot zajišťuje rekurzivně volaná funkce `ps.Substitute()`. V každém jejím volání se pro jednu proměnnou vybere konkrétní hodnota z odpovídající množiny.

V momentě, kdy je každé proměnné přiřazena hodnota (dno rekurze), se zavolá vyhodnocení postfixového výrazu z `ps.preditems`, přičemž proměnné jsou svými aktuálními hodnotami nahrazeny.

Celý algoritmus vyhodnocování je schematicky popsán následujícím pseudokódem. Tabulka `h` slouží k ukládání hodnot, kterými budou při následujícím vyhodnocování nahrazeny proměnné v predikátu. Parametr `pos` určuje číslo proměnné (v pořadí, jak se vyskytují ve formuli). Rekurze tedy proběhne tolikrát + 1, kolik obsahuje formule kvantifikovaných proměnných.

```
bool Substitute(Table h, int pos) {
  if (pos < variables.Count ){

      foreach value x in set[pos] {
          // choose value from set as a current value for variable
        h.Add(x);
          // recursion for next variable and set
        bool b = FillQvars(h, pos + 1);
          // do we know final result already?
        if (EvalCurState(ref b, ...))
           return b;
          // if not, try another value from set in next iteration
        h.Remove(x);
      }
      return EvalEndState(ref b, ..);
   }
   else { // all variables have been substituted
          // their current values are stored in h

          // evaluate predicate for current values
          return PostfixEval(h);
   }
}
```

Po vyhodnocení výrazu klasickým zůsobem, který zajišťují `ps.PostfixEval(..)` a `ps.EvalStack(..)` s opakovaným využitím `ps.EvalExpr(..)`, je pro danou kombinaci substituovaných hodnot proměnných známo, zda je podmínka splněna či ne. Výsledek vyhodnocení (`TRUE`, `FALSE`, případně `INVALID`) je vrácen nahoru, tedy do rozpracované `ps.Substitute(..)`, která byla volána pro poslední (nejpravější) kvantifikátor z formule.

Dle výsledku a typu kvantifikátoru může být zkoušení všech možností v dané větvi předčasně ukončeno. Za aktuální proměnnou se pak v tomto případě již

nedosazují další možné hodnoty (neprobíhají další iterace cyklu), ale rovnou se o úroveň výš posílá příslušný logický výsledek. Předčasné ukončení cyklu zajišťuje v pseudokódu funkce `EvalCurState(..)`, která implementuje chování popsané v tabulce 7.3.

kvantifikátor	b=**TRUE**	b=**FALSE**
EXISTS	konec cyklu; b = TRUE	zkusit další
FOR ALL	zkusit další	konec cyklu; b= FALSE
EXISTS!	poprvé: zapamatovat a zkusit další; (podruhé: konec cyklu, b = FALSE)	zkusit další
FOR AT LEAST/ AT MOST	Průběžně se počítají proběhlé iterace a kolik z nich bylo TRUE. Tyto hodnoty se porovnávají s velikostí množiny a parametry AT LEAST/AT MOST.	

Tabulka 7.3: Testování, zda je celkový výsledek znám ještě před vyzkoušením všech hodnot z množiny

Pokud nebylo možno v průběhu cyklu testování ukončit a opravdu se prošly všechny možné hodnoty z dané množiny, volá se funkce `EvalEndState(..)`, která na základě kvantifikátoru, počtu iterací a počtu `TRUE` / `FALSE` výsledků určí celkový výsledek, který se pošle o úroveň výš.

Fuzzy pravdivost

Pokud je požadována fuzzy pravdivost, musí se z množiny přislušné prvnímu kvantifikátoru projít vždy všechny hodnoty. V programu se pak namísto metody `ps.Substitute(..)` volá partnerská `ps.Substitute1L(..)` a z ní už zase dále (místo rekurze) `ps.Substitute(..)`. Vyhodnocování tedy probíhá prakticky stejně jako v předchozím případě, pouze na nejvyšší úrovni odpovídající prvnímu kvantifikátoru se prochází všechny možné hodnoty (cyklus nelze přerušit). Nebere se tedy ohled na to, zda už je celkový `TRUE` / `FALSE` výsledek znám, ale zkouší se další hodnoty, aby bylo možné určit, jak moc je podmínka splněna či porušena.

Vyhodnocení INVALID

Je-li je v některé fázi vrácen jako výsledek `INVALID` (nevyhodnotitelné, typicky odpovídá typové chybě), propaguje se tento výsledek až na nejvyšší úroveň a jako `INVALID` je pak v důsledku vyhodnocena i celá podmínka.

Pokud je v podmínce použit parametr `ONERROR` (viz kapitola 4.7), zohledňuje se tato informace již při samotném vyhodnocování postfixu, tedy uvnitř `ps.PostfixEval(..)`. Do běhu funkce `ps.Substitute(..)` je pak již vráceno (dle nastavení `ONERROR`) buď přímo `TRUE` nebo `FALSE`, případně i znovu `INVALID`, netýká-li se typová chyba konverze proměnných.

7.5 XPath výběry

Množiny uzlů jsou určeny výrazy v jazyce XPath 1.0. V programu provádění těchto výběrů odpovídá volání metody `ps.XPathSelect(..)`. Samotnou realizaci vyběru zajišťuje metoda `Evaluate(XPathExpression xp)` třídy `XPathNavigator` z namespace `System.Xml.XPath`. Ta dostává za parametr zkompilovaný řetězcový XPath výraz, ke kterému byly jako kontext přidány informace o dostupných namespacech, které uživatel dříve nadeklaroval.

Pro minimalizaci skutečných dotazů do dokumentu je v programu používána tabulka `ps.solvedXPathQueries`, která udržuje informaci o již provedených výběrech. Kvůli stejnému výběru se tedy do dokumentu nemusíme dívat zbytečně vícekrát. Z výsledků testů v Kapitole 8 je evidentní, jak velkou úsporu tento přístup přináší.

7.6 Konverze

Libovolnou hodnotu (číslo, řetězec, XPath výběr atd.) reprezentuje instance třídy `Value`. V případě potřeby zkonvertovat `Value` na typ, který opravdu reprezentuje (tedy na skutečné číslo, řetězec, XPath výběr) lze použít následující metody třídy `ParserSupport`:

- `ValToInt(..)`
- `ValToReal(..)`
- `ValToStr(..)`

Konverze uzlu (místa v XML dokumentu) na hodnotu:

- `XPathToInt(..)`
- `XPathToReal(..)`
- `XPathToStr(..)`

Konverze výsledku XPath funkce (například `count()` nebo `sum()`) na hodnotu:

- `XPathNavToInt(..)`
- `XPathNavToReal(..)`
- `XPathNavToStr(..)`

7.7 Funkce

Funkce, použitelné v predikátu i v konstatních deklaracích (viz 4.8), jsou implementovány jako statické metody třídy `Functions`. Výjimku tvoří pouze konverzní funkce `int()`, `real()` a `str()` (viz sekce 7.6) a funkce `empty()`, které jsou implementovány přímo v třídě `ParserSupport` (kvůli vyhodnocování XPath výrazů). V následujícím textu budou uvažovány pouze nekonverzní funkce.

Parametry funkcí, odpovídající jednotlivým operandům, jsou vždy typu `Value` a rovněž výsledek je typu `Value`. Skutečný typ hodnoty pak specifikuje atribut `t` třídy `Value`, který může nabývat hodnoty z výčtu `type` (`invalid, integer, real, str, boolean, xpathstr, xpath, enumdecl, interval`). Dle toho je také platný jiný atribut objektu, který již přímo nese požadovanou hodnotu.

Například pro `t = type.integer` je to atribut `i`. Podrobnou dokumentaci k třídě `Value` lze nalézt v `doc/icval.chm`.

7.7.1 Přidání nové funkce

Pro implementaci nové funkce do Icvalu jsou nezbytné následující kroky:

- **1] přidání identifikátoru funkce do výčtu** `Operators`

Nechť se přidávaná funkce jmenuje `myfnc` a nechť je `myfnc` i její ientifikátor v rámci výčtu `Operators`.

```
public enum Operators { myfnc, ...  }
```

- **2] implementace těla funkce jako statické metody v tříde** `Functions`

Předpokládejme, že `myfnc` je binární funkce. Do třídy `Functions` přidáme metodu s následující hlavičkou:

```
public static class Functions
{
  public static bool myfnc (Value v1, Value v2, ref Value ov)
  { /* Provede příslušnou operaci s operandy v1 a v2
       Výsledek vrátí jako Value ov.
       Jsou-li operandy typově chybné, vrací false, jinak true.
    */ }
  ...
}
```

- **3] přidání case větvě (podle zvoleného identifikátoru) do metody `EvalExpr(..)` třídy `ParserSupport`**

```
switch (o){
  case Operators.myfnc:
     if (!Functions.myfnc(v1, v2, ref ov))
         errlist.Add(new Error(err.typemis, line,
                       "Typová chyba myfnc", xmloutput));
     break;
 ...
 }
```

- **4] přidání pravidla do gramatiky `incox.atg`**

Nejprve přidáme identifikátor `myfnc` do sekce Tokens:

```
TOKENS
  myfnc = "myfnc".
```

Do podstromů neterminálů `Expr` a `ExpP` je potřeba přidat na vhodné místo (tj. dle typu výsledku buď k neterminálu StrVal, NulVal či BoolVal resp. StrValP, NumValP či BoolValP) pravidlo:

```
myfnc "(" Expr "," Expr ")"
```

resp.

```
myfnc "(" ExprP "," ExprP ")"
```

V případě použití v konstantách (`Expr`) se na dané úrovni vyhodnotí příslušný výraz a výsledek je poslán výš. Sémantická akce tedy může vypadat například takto (předpokládáme, že `myfnc` vrací řětezec):

```
StrVal<out Value v>    (.  v = new Value(); Value e; Value f; .)
=
myfnc "(" Expr<out e> "," Expr<out f> ")"
(. v.Copy( ps.EvalExpr(e, f, Operators.myfnc, t.line, null)); .)
```

V případě použití v predikátu (`ExprP`) se výsledná hodnota nepočítá hned, ale identifikátor funkce je uložen do postfixového zápisu predikátu `ps.preditems` (operandy již byly uloženy na jiné úrovni). Příslušná sémantická akce tedy vypadá například takto:

```
StrValP
= myfnc "(" ExprP "," ExprP ")"
(. op.o = Operators.myfnc; op.n=2; ps.preditems.Add(op.Clone());.)
```

Po modifikaci souboru s gramatikou `incox.atg` musí být pomocí Coco/R znovu vygenerovány soubory `Scanner.cs` a `Parser.cs`.

7.8 Výstup

Po vyhodnocení podmínky je ještě v rámci `ps.Validate(..)` volána dle požadovaného typu výstupu jedna z následujících metod:

- `ps.WriteOutput(..)` – textový výstup
- `ps.WriteVerboseOutput(..)` – delší textový výstup (parametr -v)
- `ps.WriteXmlOutput(..)` – XML výstup

V rámci vypisování výsledků jsou vypsány rovněž chyby a varování, detekované během zpracovávání celé podmínky. Tyto chyby se tedy nevypisují ihned, jakmile jsou zjištěny, ale ukládají se do `ps.errlist`. Tento buffer chyb se pak formou volání metody `Trace.WriteLine(errlist)` vypíše vždy až na konci pro celou podmínku najednou.

V případě, kdy je požadován výstup v XML, je toto XML během zpracovávání podmínek generováno do `XmlTextWriteru writer` (datová položka `Parseru`), založeném na třídě `System.IO.MemoryStream`. Na výstup je tedy XML najednou vypsáno až po dosažení konce souboru s podmínkami, a to pouze v případě, že nebyla během jeho parsování detekována syntaktická chyba. Naopak, v případě textového výstupu je tento text postupně generován na výstup tak, jak je zpracováván soubor s podmínkami.

Kapitola 8

Testy aplikace

Cílem této kapitoly je demonstrovat rychlost aplikace Icval při různě složitých omezujících podmínkách a především na velkých XML datech (soubor >100 MB).

Všechny testy byly prováděny na PC následující konfigurace a při testování nebyly spuštěny žádné jiné náročnější aplikace:

- CPU AMD Athlon 64 3200+ 2.00GHz
- 1GB RAM
- Windows XP Professional, Service Pack 2

Pro demonstraci toho, jak velkou část celkové doby zabere pouhé načtení daného XML souboru do paměti (vytvoření DOM), je zde uveden krátký přehled. Jde vlastně o dobu, potřebnou pro provedení metody `Load` třídy `XmlDocument` z namespace `System.Xml`.

Typicky platí, že načítání většího dokumentu trvá déle, záleží ovšem také na vnitřní struktuře (košatosti XML), proto například načtení největšího souboru `standard.xml` zabíralo méně času než u `order.xml`.

Soubor	**Velikost (MB)**	**Doba pro načtení**
blog.xml	1,27	<1 sec
cd.xml	25,5	<3 sec
order.xml	72,9	<10 sec
standard.xml	111	<8 sec

Tabulka 8.1: Doba potřebná pro načtení XML souboru do paměti

Uvedené soubory se na přiloženém CD nachází v adresáři `tests/xml`.

Všechny dále uvedené soubory s podmínkami i soubory s výstupy a časovými údaji jsou umístěny v adresáři `tests/constraints`.

8.1 Test 1 - FOR ALL

V následujícím testu byly procházeny všechny uzly určitého druhu ze souboru `standard.xml` (111 MB) a na nich testována daná podmínka. Kolik uzlů musel validátor skutečně projít, je patrné z výstupu.

test1.txt:

```
CONSTRAINT "existujici idy" {
FORMULA:
  FOR ALL x IN '//item'
  ( not (empty('$x/@id')) )
}

CONSTRAINT "neprazdne idy" {
FORMULA:
  FOR ALL x IN '//item'
  ( length(str('$x/@id')) > 0 )
}

CONSTRAINT "spravny nazev reference kategorie" {
FORMULA:
  FOR ALL cat IN '//incategory/@category'
  ( match(str(cat), "^category\d+$") )
}
```

:\> icval test1.txt standard.xml –c

Výstup:

test1.out.txt:

```
-----------------------------------------------------------------
Loading XML: 00:00:07.5625000

Constraint "existujici idy" : (1; 21750/21750)
time: 00:00:01.3906250

Constraint "neprazdne idy" : (1; 21750/21750)
time: 00:00:00.8593750

Constraint "spravny nazev reference kategorie" : (1; 82151/82151)
time: 00:00:01.7187500

TOTAL time: 00:00:11.5625000
-----------------------------------------------------------------
```

Z výstupu je vidět, že nejvíce času zabralo načítání XML (přes 7 sec), každá z podmínek pak byla vyhodnocena do 2 sec. Přitom bylo otestováno v prvním a druhém případě vždy 21 750 uzlů, v třetím 82 151 uzlů. Celkový čas pro validaci těchto tří podmínek tedy činil necelých 12 sec.

Důvodem, proč byl čas zpracování druhé podmínky téměř poloviční, ačkoliv bylo procházeno a testováno stejné množství uzlů jako u první, je využití výsledku již jednou provedeného XPath výběru `'//item'`. V případě druhé podmínky se již aplikace nemusí znovu dotazovat do dokumentu, pouze probíhá test. Tento rys aplikaci zrychluje tím více, čím by byl daný XPath dotaz složitější. Ukládání se týká pouze XPath výběrů bez proměnných (neobsahují referenci ve formě $var).

Pokud bychom druhou podmínku nechali testovat samostatně, byl by čas jejího provádění delší a zhruba by odpovídal času první podmínky:

```
-----------------------------------------------------------------
Constraint "neprazdne idy" : (1; 21750/21750)
time: 00:00:01.4531250
-----------------------------------------------------------------
```

8.2 Test 2 - FOR ALL, FOR ALL | EXISTS

V tomto testu byly kvantifikátorem `FOR ALL` vybírány ze souboru `standard.xml` podstromy, na kterých byly dále testovány podmínky za využití kvantifikátorů `FOR ALL` a `EXISTS`. Opět byly procházeny všechny vyhovující uzly (parametr -c).

test2.txt:

```
## vzajemna unikatnost vsech incategory/@category
## v ramci kazde item

CONSTRAINT "unikatnost @category" {
FORMULA:
  FOR ALL it IN '//item'
    FOR ALL cat1 IN '$it/incategory/@category'
    FOR ALL cat2 IN '$it/incategory/@category'
    ( str(cat1)=str(cat2) -> cat1=cat2 )
}

## v kazdem neprazdnem mailboxu existuje mail,
## v jehoz textu je nejaka fraze oznacena jako keyword

CONSTRAINT "mail s kw" {
FORMULA:
  FOR ALL mb IN '//mailbox[mail]'
    EXISTS ml IN '$mb/mail'
    ( not( empty('$ml/text/keyword') ) )
}
```

```
## v kazdem neprazdem mailboxu musi mit
## vsechny maily ve svem textu keyword

CONSTRAINT "vsechny maily s kw" {
FORMULA:
  FOR ALL mb IN '//mailbox[mail]'
    FOR ALL ml IN '$mb/mail'
    ( not( empty('$ml/text/keyword') ) )
}
```

:\> icval test2.txt standard.xml –c

Výstup:

test2.out.txt:

```
-----------------------------------------------------------------
Loading XML: 00:00:07.9375000

Constraint "unikatnost @category": (0; 21599/21750)
time: 00:00:08.2812500

Constraint "mail s kw": (0; 6686/13174)
time: 00:00:01.4843750

Constraint "vsechny maily s kw": (0; 3855/13174)
time: 00:00:00.8906250

TOTAL time: 00:00:18.6250000
-----------------------------------------------------------------
```

Je patrné, že nejnáročnější byla první podmínka, jejíž testování zabralo téměř 9 sec. Pro každý z 21750 uzlů `item` musely být otestovány všechny dvojice vnořených `incategory/@category`. Tedy pro každou `item` bylo provedeno n^2 testů, kde n je počet jejích vnořených `incategory` elementů.

Ve třetí podmínce mohl být opět použit již jednou provedený XPath výběr z druhé podmínky `'//mailbox[mail]'`.

8.3 Test 3 - FOR ALL + FOR ALL

Jak bylo vidět z předchozího testu, podmínky na vzájemnou unikátnost mají kvadratickou složitost co do počtu jednotlivých testů. Následující podmínky demonstrují, jak se tato složitost odrazí do skutečného času, potřebného pro provedení.

test3.txt:

```
CONSTRAINT "africa" {
FORMULA:
  FOR ALL i1 IN '/site/regions/africa/item'
  FOR ALL i2 IN '/site/regions/africa/item'
  ( (str(i1)=str(i2)) -> i1=i2 )
}

CONSTRAINT "australia" {
FORMULA:
  FOR ALL i1 IN '/site/regions/australia/item'
  FOR ALL i2 IN '/site/regions/australia/item'
  ( (str(i1)=str(i2)) -> i1=i2 )
}

CONSTRAINT "europe" {
FORMULA:
  FOR ALL i1 IN '/site/regions/europe/item'
  FOR ALL i2 IN '/site/regions/europe/item'
  ( (str(i1)=str(i2)) -> i1=i2 )
}
```

:\> icval test3.txt standard.xml –c

Výstup:

test3.out.txt:

```
-----------------------------------------------------------------
Loading XML: 00:00:07.3906250

Constraint "africa" : (1; 550/550)
time: 00:00:18.0468750

Constraint "australia" : (1; 2200/2200)
time: 00:05:00.2343750

Constraint "europe": (0; 5970/6000)
time: 00:34:54.6250000

TOTAL time: 00:40:20.3437500
-----------------------------------------------------------------
```

Z výsledků testů je evidentní, že pro ověřování vzájemné unikátnosti hodnot větší skupiny uzlů už přestává být způsob „testování po dvou“ (`FOR ALL`, `FOR ALL`) vhodný. Přesto se ale konstrukce může hodit pro „lokální unikátnost“ jako v testu 2. Unikátnost ID elementů, jejichž počet jde do tisíců, jako v případě těchto testů, je lépe popsat nějakým klasickým schema jazykem. Vyhodnocování tak může být sofistikovanější.

8.4 Test 4 - XPath výběr

Způsob, jakým jsou zapsány XPath dotazy, rovněž ovlivňuje rychlost vyhodnocování. Následující podmínky jsou „stejné“ ve smyslu, že stejným způsobem testují stejnou skupinu uzlů. V jednom případě je tato skupina získána pomocí absolutně zadaného XPath výběru, v dalším případě je sice vybráno ve finále to samé, ale kvůli použití XPath operátoru // (začátek kdekoliv) musí být prohledán celý dokument, což samozřejmě proces zpomaluje.

test4.txt:

```
## relativni vyber

CONSTRAINT "neprazdny text #1"
{
FORMULA:
  FOR ALL x IN '//item/description[parlist]'
    FOR ALL y IN '$x/parlist[listitem]'
      FOR ALL z IN '$y/listitem'
      ( not (empty ('$z/text')) )
}

## absolutni vyber

CONSTRAINT "neprazdny text #2" {
FORMULA:
  FOR ALL x IN '/site/regions/*/item/description[parlist]'
    FOR ALL y IN '$x/parlist[listitem]'
      FOR ALL z IN '$y/listitem'
     ( not (empty ('$z/text')) )
}
```

:\> icval test4.txt standard.xml –c

Výstup:

test4.out.txt:

```
-----------------------------------------------------------------
Loading XML: 00:00:07.5000000

Constraint "neprazdny text #1": (0; 3397/6298)
time: 00:00:01.7187500

Constraint "neprazdny text #2": (0; 3397/6298)
time: 00:00:00.4687500

TOTAL time: 00:00:09.7187500
-----------------------------------------------------------------
```

Díky absolutní cestě se doba zpracování jedné podmínky zkrátila více než trojnásobně. Z toho je rovněž patrné, jak velkou část celkového času může zabrat pouhé vyhodnocení XPath dotazu na úkor vlastního testování.

Kapitola 9

Závěr

Smyslem této práce bylo ukázat, že ačkoliv jsou současné schema jazyky dostatečně silné pro popis strukturálních omezení, pro vyjádření složitějších sémantických omezení se příliš nehodí. Specializovaný jazyk Schematron sice dokáže postihnout řadu rozumných omezení, stále však jde jen o část toho, co lze popsat pomocí spojení výrazových prostředků logiky prvního řádu a jazyka XPath. Právě na těchto základech stojí zatím spíše experimentální jazyk CLiX, který teoreticky nabízí široké možnosti, ale díky stagnaci ve vývoji a malé podpoře, se zatím výrazněji neprosadil.

Cesta využití kvantifikátorů pro popis vzájemných vztahů mezi prvky dokumentu, které blíže určíme XPathem, tedy není originální. Na MFF byl stejný princip využit například i pro popis vazeb a omezení pro XML s neurochirurgickými daty, a to v diplomové práci [22].

Cílem návrhu jazyka Incox bylo vytvořit již od začátku jednoduchý kompaktní jazyk, určený speciálně pro popis omezujících podmínek v libovolných XML dokumentech. Jako praktické se ukázaly nápady s využitím dvojí syntaxe, hodící se pro různé kontexty. Za užitečný prvek lze považovat rovněž i možnost ukládání hodnot do konstant nebo konstantních výčtů, což bylo demonstrováno na praktickém příkladě s chybějícími kapitolami.

Další zcela nový rys je zavedení určitého rozšíření kvantifikátoru `FOR ALL`, a to ve formě konstruktu `FOR { AT LEAST | AT MOST }`. S jeho pomocí lze v podmínce přesně specifikovat počet nebo % požadovaných vyhovujících prvků. S tím souvisí i rozšířené možnosti výstupu, ze kterého lze získat též další dodatečné informace o tom, jak je podmínka splněna nebo porušena. Výstup se tedy neomezuje na pouhé určení `TRUE/FALSE`.

Na přiloženém CD je v adresáři `examples` k dispozici sada XML dokumentů a vzorových podmínek, demonstrujících praktickou využitelnost implementace jazyka. V hojné míře jsou v predikátech využity také vlastní vestavěné funkce, především pro práci s řetězci. Skutečné využití navrženého jazyka, zde pro jednoduchost implementovaného jako konzolová aplikace, si lze představit především ve formě modulu v nějakém větším informačním systému.

Literatura

[1] Extensible Markup Language (XML), http://www.w3.org/XML

[2] W3C XML Schema, http://www.w3.org/XML/Schema

[3] Relax NG, http://relaxng.org

[4] Schematron,
An XML Structure Validation Language using Patterns in Trees,
http://www.schematron.com/

[5] Taxonomy of XML Schema Languages using Formal Language Theory,
Makoto Murata, Dongwon Lee, Murali Mani, 2001,
http://www.cobase.cs.ucla.edu/tech-docs/dongwon/mura0619.pdf

[6] Technologie XML,
Mlýnková I., Pokorný J., Richta K., Toman K., Toman V.,
Nakladatelství Karolinum, 2006

[7] XML Path Language (XPath), Version 1.0, http://www.w3.org/TR/xpath

[8] XML Schema 1.1 Part 1: Structures - 3.12 Assertions,
W3C Working Draft 31 August 2006,
http://www.w3.org/TR/xmlschema11-1/#cAssertions

[9] RELAX NG with custom datatype libraries,
Elliotte Rusty Harold, 2004,
http://www-128.ibm.com/developerworks/xml/library/x-custyp/

[10] XML Matters: Kicking back with RELAX NG, Part 2,
David Mertz, 2003,
http://www-128.ibm.com/developerworks/xml/library/
x-matters26.html

[11] ISO/IEC 19757 - Document Schema Definition Languages (DSDL),
Part 3: Rule-based validation – Schematron,
http://standards.iso.org/ittf/PubliclyAvailableStandards/
c040833_ISO_IEC_19757-_2006(E).zip

[12] Schematron Implementation,
skeleton1-5.xsl - An Implementation of Schematron 1.5 in XSLT,
http://xml.ascc.net/schematron/1.5/

[13] Schematron Tutorial,
Miloslav Nic, Zvon.org,
`http://www.zvon.org/xxl/SchematronTutorial/`

[14] XML Schémata, Kapitola 5. Schematron,
Jiří Kosek,
`http://www.kosek.cz/xml/schema/sch.html`

[15] Relax NG Compact syntax,
Non-XML syntax for Relax NG,
`http://www.oasis-open.org/committees/relax-ng/compact-20021121.html`

[16] Constraint Language in XML (CLiX), `http://www.clixml.org`

[17] The OASIS Content Assembly Mechanism (CAM),
`http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=cam`

[18] Sandcastle - June 2007 Community Technology Preview (CTP),
`http://www.microsoft.com/downloads/details.aspx?FamilyID=e82ea71d-da89-42ee-a715-696e3a4873b2`

[19] Sandcastle Help File Builder, `http://www.codeplex.com/SHFB`

[20] HTML Help Workshop,
`http://www.microsoft.com/downloads/details.aspx?FamilyID=00535334-c8a6-452f-9aa0-d597d16580cc`

[21] The Compiler Generator Coco/R, `http://www.ssw.uni-linz.ac.at/coco`

[22] Konceptuální modelování neurochirurgických dat,
Jakub Bystroň,
Diplomová práce, MFF UK, 2006

Příloha A

Gramatika jazyka Incox

Gramatika jazyka Incox je zapsána v EBNF (Extended Backus–Naur Form) tak, jak ji přijíma parser generátor Coco/R [21]. V sekci Tokens jsou uvedeny pouze složené tokeny, za tokeny jsou dále také považována všechna slova složená z malých písmen, vyskytující se v sekci Productions (například `constraint`, `str`, `div`) a také všechny znaky, uzavřené mezi uvozovkami (`"+"`). Jazyk Incox není case-sensitive, tedy u použitých klíčových slov nezáleží na velikosti písmen.

```
IGNORECASE
CHARACTERS
  digit    = '0' .. '9'.
  letter   = 'A' .. 'Z'.
  stringCh = ANY - '"' - '\\' - '\r' - '\n'.
  xpathChr = ANY - "'" - '\\' - '\r' - '\n'.
TOKENS
  string
  = '"' { ( stringCh | '\\' (stringCh| '"' | '\\') ) } '"'.
  xpath
  = "'" { ( xpathChr | '\\' (stringCh| '\''| '\\') ) } "'".
  intNum   = digit {digit}.
  idf      = letter {letter | digit | '_'}.
  realNum  = digit {digit} '.' digit {digit}.

COMMENTS FROM "/*" TO "*/" NESTED
COMMENTS FROM "#" TO '\n'
IGNORE '\t' + '\r' + '\n'

PRODUCTIONS
Incox = { Decl } Constr { Constr }.
Constr = constraint string "{" { Parameters } Formula "}".

/***** CONSTANTS *****/
Decl = ConstDecl | EnumDecl | IntervalDecl | NsParam.

ConstDecl    = const[":"] idf "=" Expr.
EnumDecl     = enum[":"] idf "=" "(" Expr { ","  Expr } ")".
IntervalDecl
  = interval[":"] idf "=" "(" Expr "," Expr ["," Expr ] ")" .

/***** EXPRESSIONS *****/
Expr  = ValOp { OpPM ValOp }.
ValOp = ValUm { OpMD ValUm }.
```

```
ValUm = ["+"] Val | "-" Val.

Val
= NumVal | StrVal | BoolVal
| "(" Predicate ")" | XPath | idf.

StrVal
= string | str "(" Expr ")"
| tolower "(" Expr ")" | toupper "(" Expr ")"
| trim "(" Expr ")" | trimall "(" Expr ")".

NumVal
= intNum | realNum
| int "(" Expr ")" | real "(" Expr ")" | length "(" Expr ")".

BoolVal
= match "(" Expr "," Expr ")"
| empty "(" ( XPath | idf ) ")".

/***** PARAMETERS *****/
Parameters = StateParam | OnErrorParam | FuzzyParam | CountsParam | NsParam.

NsParam      = namespace [":"] idf "=" string.
StateParam   = state     [":"] ( on | off ).
OnErrorParam = onerror   [":"] ( true | false ).
FuzzyParam   = fuzzy     [":"] ( on | off ) [intNum].
CountsParam  = counts    [":"] ( on | off ).

/***** SELECTS *****/
Formula = formula [":"] Select {Select} "(" Predicate ")".

Select
= for all idf in Set
| for ForSpec idf in Set
| exists ["!"] idf in Set.

ForSpec
= at ( (least intNum ) | ( most intNum ) )  ["%"]
[ "," at ( (least intNum )| ( most intNum ) )  ["%"] ].

Set = XPath | idf.

/***** PREDICATE *****/
Predicate
= ( Cond | not "(" Cond ")")
        { LogOp ( Cond | not "(" Cond ")" ) }.

Cond = Expr { RelOp Expr }.

XPath = ["#" intNum ] xpath.

/***** OPERATORS *****/
OpPM  = "+" | "-".
OpMD  = "*" | "/" | div | mod.
RelOp = "=" | "!=" | "<" | ">" | "<=" | ">=".
LogOp = and | or | "->".
```

Příloha B

IncoxInput.xsd

```
<?xml version="1.0" encoding="windows-1250"?>
<!-- ..............................-->
<!-- Incox Constraints XML Schema -->
<!-- ..............................-->

<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
xmlns="incox" targetNamespace="incox" elementFormDefault="qualified">

<xs:element name="incox">
<xs:complexType>
  <xs:sequence>

<!--   globals   -->
<xs:element name="globals" minOccurs="0" maxOccurs="1">
<xs:complexType>
<xs:sequence minOccurs="0" maxOccurs="unbounded">
<xs:choice>

<!--   const  -->
<xs:element name="const">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="name" type="idf"/>
      <xs:element name="value" type="notempty_string" />
    </xs:sequence>
  </xs:complexType>
</xs:element>

<!--   enum  -->
<xs:element name="enum">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="name" type="idf"/>
      <xs:element name="values">
        <xs:complexType>
          <xs:sequence minOccurs="1" maxOccurs="unbounded">
            <xs:element name="value" type="notempty_string" />
          </xs:sequence>
        </xs:complexType>
      </xs:element>
    </xs:sequence>
  </xs:complexType>
```

```
</xs:element>

<!--   interval  -->
<xs:element name="interval">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="name" type="idf"/>
      <xs:element name="begin" type="notempty_string" />
      <xs:element name="end" type="notempty_string" />
      <xs:element name="step" type="notempty_string" minOccurs="0" />
    </xs:sequence>
  </xs:complexType>
</xs:element>

<!--   namespace  -->
<xs:element name="namespace" type="ns_type" />

</xs:choice>
</xs:sequence>
</xs:complexType>
</xs:element>

<!--   constraints   -->
<xs:element name="constraints" minOccurs="1" maxOccurs="1">
 <xs:complexType>
 <xs:sequence minOccurs="1" maxOccurs="unbounded">

<!--   constraint   -->
<xs:element name="constraint">
  <xs:complexType>
  <xs:sequence>
  <xs:element name="name" type="notempty_string" />

  <!--   parameters   -->
  <xs:element name="parameters" minOccurs="0" maxOccurs="1">
    <xs:complexType>
      <xs:sequence minOccurs="0" maxOccurs="unbounded">
        <xs:element name="state" type="OnOff" minOccurs="0" />
        <xs:element name="onerror" type="TrueFalse" minOccurs="0" />
        <xs:element name="fuzzy" type="fuzzy_type" minOccurs="0" />
        <xs:element name="counts" type="OnOff" minOccurs="0" />
        <xs:element name="results" type="OnOff"  minOccurs="0" />
        <xs:element name="namespace" type="ns_type" minOccurs="0" />
      </xs:sequence>
    </xs:complexType>
  </xs:element>

  <!--   formula  -->
  <xs:element name="formula">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="selects" minOccurs="1" maxOccurs="1">
          <xs:complexType>
            <xs:sequence minOccurs="1" maxOccurs="unbounded">
              <xs:element name="select">

                <!--   quantifiers  -->
```

```
                <xs:complexType>
                  <xs:sequence>
                    <xs:choice minOccurs="1" maxOccurs="1">
                      <xs:element name="for" type="for_type" />
                      <xs:element name="forall" type="idf" />
                      <xs:element name="exists" type="idf" />
                      <xs:element name="exists1" type="idf" />
                    </xs:choice>
                    <xs:element name="in" type="notempty_string" />
                  </xs:sequence>
                </xs:complexType>
              </xs:element>
            </xs:sequence>
          </xs:complexType>
        </xs:element>

        <!--   predicate  -->
        <xs:element name="predicate" type="notempty_string"
                    minOccurs="1" maxOccurs="1" />
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  </xs:sequence>
  </xs:complexType>
</xs:element>
</xs:sequence>
</xs:complexType>
</xs:element>

<!-- - - - - - - - - - -->
</xs:sequence>
</xs:complexType>
</xs:element>

<!-- COMPLEX TYPE definitions -->
<!-- namespace type -->
<xs:complexType name="ns_type">
  <xs:sequence>
    <xs:element name="prefix" type="idf" />
    <xs:element name="uri" type="notempty_string" />
  </xs:sequence>
</xs:complexType>

<!-- FOR type -->
<xs:complexType name ="for_type">
  <xs:simpleContent>
    <xs:extension base="idf">
      <xs:attribute name="atmost" type="ValuePerc" use="optional" />
      <xs:attribute name="atleast" type="ValuePerc" use="optional" />
    </xs:extension>
  </xs:simpleContent>
</xs:complexType>

<!-- fuzzy parameter type -->
<xs:complexType name="fuzzy_type">
  <xs:simpleContent>
    <xs:extension base="OnOff">
```

```
      <xs:attribute name="accuracy" type="xs:positiveInteger"
                                    use="optional" />
    </xs:extension>
  </xs:simpleContent>
</xs:complexType>

<!-- SIMPLE TYPE definitions -->
<!-- non empty string -->
<xs:simpleType name="notempty_string">
  <xs:restriction base="xs:string">
    <xs:minLength value="1"/>
  </xs:restriction>
</xs:simpleType>

<!-- identifier -->
<xs:simpleType name="idf">
<xs:restriction base="notempty_string">
  <xs:pattern value="^[a-zA-Z][a-zA-Z0-9_]*$" />
</xs:restriction>
</xs:simpleType>

<!-- 'true' OR 'false' values -->
<xs:simpleType name="TrueFalse">
  <xs:restriction base="notempty_string">
    <xs:enumeration value="true" />
    <xs:enumeration value="TRUE" />
    <xs:enumeration value="false" />
    <xs:enumeration value="FALSE" />
  </xs:restriction>
</xs:simpleType>

<!-- 'on' OR 'off' values -->
<xs:simpleType name="OnOff">
  <xs:restriction base="notempty_string">
    <xs:enumeration value="on" />
    <xs:enumeration value="ON" />
    <xs:enumeration value="off" />
    <xs:enumeration value="OFF" />
  </xs:restriction>
</xs:simpleType>

<!-- numeric value or percentage value -->
<xs:simpleType name="ValuePerc">
  <xs:restriction base="notempty_string">
    <xs:pattern value="^[0-9]+[%]?$" />
  </xs:restriction>
</xs:simpleType>

</xs:schema>
```

Příloha C

IncoxOutput.xsd

```
<?xml version="1.0" encoding="utf-8" ?>
<!-- ..............................-->
<!--   Icval Output XML Schema    -->
<!-- ..............................-->
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
xmlns="incox" targetNamespace="incox" elementFormDefault="qualified">

<xs:element name="icval">
<xs:complexType>
  <xs:sequence>
  <!--   global errors   -->
  <xs:element name="global_errors" type="errlist"
                                   minOccurs="0" maxOccurs="1" />
  <!--   constraints   -->
  <xs:element name="constraints" minOccurs="1" maxOccurs="1">
  <xs:complexType>
  <xs:sequence>
  <!--   constraint   -->
  <xs:element name="constraint" minOccurs="0" maxOccurs="unbounded">
   <xs:complexType>
    <xs:sequence>
      <!--   local errors   -->
      <xs:element name="local_errors" type="errlist"
                                      minOccurs="0" maxOccurs="1" />
      <!--   name   -->
      <xs:element name="name" type="xs:string" />
      <!--   overall result   -->
      <xs:element name="overall_result" type="t_result" />
      <!--   additional info   -->
      <xs:element name="additional_info" minOccurs="0" maxOccurs="1">
        <xs:complexType>
          <xs:sequence>
            <xs:element name="first_quantifier" type="quant" minOccurs="0" />
            <xs:element name="true_count" type="xs:integer" minOccurs="0" />
            <xs:element name="all_count" type="xs:integer" minOccurs="0" />
            <xs:element name="fuzzy_truth" type="myDouble" minOccurs="0" />
          </xs:sequence>
        </xs:complexType>
      </xs:element>
    </xs:sequence>
  </xs:complexType>
  </xs:element>
```

```
  </xs:sequence>
  </xs:complexType>
  </xs:element>
  </xs:sequence>
</xs:complexType>
</xs:element>

<!-- COMPLEX TYPE definitions -->
<!-- error list -->
<xs:complexType name="errlist">
  <xs:sequence minOccurs="0" maxOccurs="unbounded">
    <xs:choice>
      <xs:element name="error" type="t_error" />
      <xs:element name="warning" type="t_error" />
    </xs:choice>
  </xs:sequence>
</xs:complexType>

<!-- error type -->
<xs:complexType name="t_error">
  <xs:sequence>
    <xs:element name="line" type="xs:integer" />
    <xs:element name="message" type="xs:string" />
    <xs:element name="count" type="xs:integer" minOccurs="0" />
  </xs:sequence>
</xs:complexType>

<!-- SIMPLE TYPE definitions -->
<!-- '0', '1' OR 'X' -->
<xs:simpleType name="t_result">
  <xs:restriction base="xs:string">
    <xs:enumeration value="0" />
    <xs:enumeration value="1" />
    <xs:enumeration value="X" />
  </xs:restriction>
</xs:simpleType>

<!-- quantifiers -->
<xs:simpleType name="quant">
  <xs:restriction base="xs:string">
    <xs:enumeration value="FOR AT LEAST" />
    <xs:enumeration value="FOR AT MOST" />
    <xs:enumeration value="FOR AT LEAST AT MOST" />
    <xs:enumeration value="FOR ALL" />
    <xs:enumeration value="EXISTS" />
    <xs:enumeration value="EXISTS!" />
  </xs:restriction>
</xs:simpleType>

<!-- double (delimiters: '.' , ',') -->
<xs:simpleType name="myDouble">
  <xs:restriction base="xs:string">
    <xs:pattern value="^[0-9]*(.|,)[0-9]*$" />
  </xs:restriction>
</xs:simpleType>

</xs:schema>
```

Příloha D

Obsah přiloženého CD-ROM

```
bc.pdf                          text bakalářské práce
icval.exe                       validátor Icval
IncoxInput.xsd                  schéma pro vstupní podmínky v XML
XmlToIncox.xsl                  XSL transformace XML Incox => TXT Incox

/sources                        zdrojové kódy a pomocné soubory

          icval.sln             projekt pro MSVS 2005
          *.cs                  zdrojové kódy C# podle tabulky 7.2
          incox.atg             gramatika Incoxu
          IncoxInput.xsd
          IncoxOutput.xsd
          XmlToIncox.xsl
          Parser.frame
          Scanner.frame

/examples                       příklady, podrobněji viz Kapitola 6.3
        /centers
        /countries
        /dvojice
        /empty
        /errors
        /kapitoly
        /quantifiers
        /spec
        /strfnc
        /strom
        /sumcount
        /typerr

/tests                          podklady pro "zátěžové" testy (viz Kapitola 8)
        /constraints
        /xml

/doc                            dokumentace
       userdoc.pdf              Kapitola 6 samostatně
       pgdoc.pdf                Kapitola 7 samostatně
       icval.chm                vygenerovaná dokumentace zdrojových kódů
```