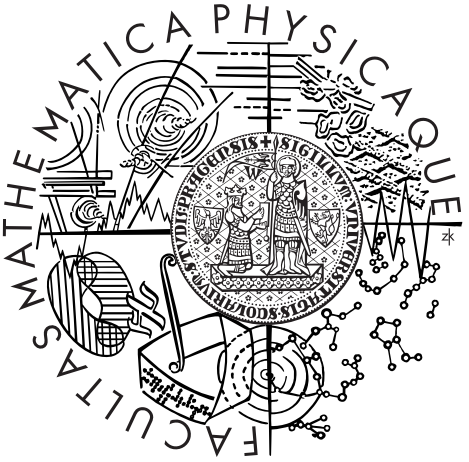
Univerzita Karlova v Praze
Matematicko-fyzikální fakulta

Diplomová práce



Jan Červák

Statická analýza XSLT programů

Katedra softwarového inženýrství

Vedoucí diplomové práce: RNDr. David Bednárek
Studijní program: Informatika
Studijní obor: Softwarové systémy

Děkuji vedoucímu své diplomové práce RNDr. Davidu Bednárkovi za podporu a cenné rady v průběhu psaní práce. Dále děkuji své rodině a přátelům, bez jejich podpory by tato práce nikdy nevznikla.

Prohlašuji, že jsem svou diplomovou práci napsal samostatně a výhradně s použitím citovaných pramenů. Souhlasím se zapůjčováním práce.

V Praze dne 11. srpna 2006 Jan Červák

Název práce: Statická analýza XSLT programů

Autor: Jan Červák

Katedra: Katedra softwarového inženýrství

Vedoucí diplomové práce: RNDr. David Bednárek

E-mail vedoucího: David.Bednarek@mff.cuni.cz

Abstrakt: Zabýváme se statickou analýzou XSLT programů se znalostí schématu vstupních dokumentů. Analýza je zaměřena na odhalování běhových chyb v programech. Konkrétně hledáme nedosažitelná pravidla, slepá volání a cykly ve voláních pravidel, jako zdroj potenciálně nekonečného běhu programu. XSLT je turingovsky úplný jazyk, kompletní statická analýza je z principu neproveditelná, řešení tohoto problému jsou jen přibližná. Prezentované řešení je kvazi-simulace běhu programu nad modelem XML dokumentu. Výsledný graf toku řízení programu dává informace o zkoumaných problémech. Algoritmus simulace je zaveden nezávisle na použitém modelu, což přináší možnost analýzy na různé úrovni detailu.

Klíčová slova: analýza toku řízení, XSLT, XML Schema, běhové chyby, zastavení

Title: Static analysis of XSLT programs

Author: Jan Červák

Department: Department of Software Engineering

Supervisor: RNDr. David Bednárek

Supervisor's e-mail address: David.Bednarek@mff.cuni.cz

Abstract: The aim of this work is static analysis of XSLT programs with the knowledge of an input document's schema. Detection of runtime errors in program is the goal of this analysis. Especially finding out unreachable template rules, blind calls and call-cycles is crucial. Because XSLT is a turing-complete language, full static analysis is in principle impossible. The solution is therefore only approximate. Result of our research is a quasi-simulation of program execution over a model of XML document. The control-flow graph contains the information about problems of interest. The simulation algorithm is model-independent, which gives us the chance to perform analyse on different detail-levels.

Keywords: control-flow analysis, XSLT, XML Schema, run-errors, termination

Obsah

1 Úvod **1**

2 Principy a cíle analýzy **3**
2.1 XML technologie a výzkum transformací 3
2.2 Terminologie 6
2.2.1 XML 6
2.2.2 XML Schema 8
2.2.3 XSLT 9
2.2.4 XPath 10
2.3 Cíle práce 12
2.3.1 Slepé cesty 12
2.3.2 Nedosažitelná pravidla 13
2.3.3 Cykly ve vzájemném volání pravidel 14

3 Modelování **15**
3.1 Staticky analyzovatelné problémy 15
3.2 Zavedená zjednodušení 17
3.2.1 Zjednodušení jazyka XML Schema 18
3.2.2 Zjednodušení jazyka XSLT 20
3.3 Modelování dokumentu a vyhodnocování XPath výrazů 26
3.3.1 Model XML dokumentu 26
3.3.2 Zjednodušení XPath a vyhodnocování funkcí Eval . . . 27
3.4 Uvažované modely 31
3.4.1 Společné principy vytvoření modelu 31
3.4.2 Model plain 32
3.4.3 Model tree 34
3.4.4 Rozšíření plain$^{\mathsf{m}}$ a tree$^{\mathsf{m}}$ 34

4 Algoritmy analýzy **37**
4.1 Konstrukce grafu řízení 37
4.1.1 Graf řízení 38

4.1.2 Konstrukce toku kontextů . . . 39
4.1.3 Zohlednění priorit při výběru pravidel . . . 41
4.1.4 Propagace parametrů . . . 45
4.2 Konečnost výpočtu a odhady složitosti . . . 46
4.3 Graf řízení a zkoumané problémy . . . 48
4.3.1 Slepé cesty . . . 48
4.3.2 Nedosažitelná pravidla . . . 49
4.3.3 Cykly a nekonečný běh programu . . . 49
4.4 Postprocessing kontextových cyklů . . . 51
4.4.1 Eliminace dopředných cyklů . . . 51
4.4.2 Eliminace zavlečených cyklů . . . 54
4.5 Uvažovaná řešení . . . 55

5 Výsledky **57**
5.1 Úspěšnost analýzy . . . 57
5.1.1 Reálná testovací data . . . 57
5.1.2 Procedurální cykly . . . 58
5.2 Srovnání modelů . . . 58
5.2.1 Model plain vs. model plain$^{\mathsf{m}}$ . . . 58
5.2.2 Model plain vs. model tree . . . 59
5.2.3 Velikost modelu tree . . . 61
5.2.4 Vnitřní struktura typů schématu . . . 62
5.2.5 Idea hybridního modelu . . . 63
5.3 Související práce . . . 65

6 Závěr **67**

Literatura **68**

A Příklady **71**
A.1 Schémata . . . 71
A.2 Programy . . . 74

B Algoritmy **76**
B.1 Algoritmus konstrukce grafu řízení . . . 76
B.1.1 Inicializace . . . 76
B.1.2 Konstrukce toku . . . 77
B.1.3 Propagace parametrů . . . 77

C Obsah přiloženého CD **78**

D Aplikace WXSA **79**
D.1 Překlad a instalace aplikace 79
D.2 Použití aplikace 81

Kapitola 1

Úvod

V současnosti jsou XML technologie jedním z hlavních prostředků pro reprezentaci a výměnu strukturovaných dat. Velké oblibě se těší zejména díky širokému použití na webu. Ve spoustě případů je XML použito jako univerzální meziformát dat, která jsou dále transformována do nejrůznějších podob – v případě webových stránek například do formátu HTML.

Transformacím XML dat se věnuje i tato práce. Je zaměřena zejména na statickou analýzu transformací v jazyce XSLT, jejímž cílem je zkoumání transformačních programů bez znalosti konkrétních vstupních dat, pouze se znalostí jejich schématického popisu. Obecně lze „staticky“ analyzovat spoustu aspektů transformačních programů, my se zaměříme na odhalování chyb v programech. Ke zkoumání jsme vybrali klasické běhové chyby jako nedosažitelnost pravidel a slepé cesty, pokusíme se také detekovat některé případy nekonečného běhu progamu.

Jelikož jazyk XSLT nemá implicitní typovou kontrolu, v programu nejsou nijak vyjádřeny požadavky na tvar vstupních dat. Právě nedostatečná znalost struktury může vést ke značně záludným chybám, které se typicky projevují pouze na některých vstupních datech. Analýza se znalostí všech vstupů by chyby jistě odhalila, ale právě zkoumání chování programu na všech vstupech je těžko proveditelné, neboť množina vstupních dokumentů je typicky nekonečná. Řešením tedy může být statická analýza, kdy jsou použity údaje ze schématického popisu a s jistou mírou přesnosti zachyceny všechny vstupní dokumenty.

Teorie ukázala, že XSLT patří mezi turingovsky úplné jazyky. Spousta problémů, které bychom chtěli umět řešit, je tedy z principu nerozhodnutelná. V takovém případě přichází řada na heuristická řešení.

Zvolené řesení se zakládá na odhadu toku řízení transformačního programu nad modelem potenciálního vstupního dokumentu. Podle schématu

vstupu je zkonstruován model dokumentu – struktura, která svým obsahem zachycuje s určitou přesností vztahy ve skutečných dokumentech. Nad modelem je pak provedena kvazi-simulace běhu transformace. Informace o přítomnosti zkoumaných chyb následně získáme z grafu toku řízení simulovaného běhu.

Kapitola 2

Principy a cíle analýzy

2.1 XML technologie a výzkum transformací

Extensible Markup Language [3] je jazyk pro zápis dat stromové struktury, standardizovaný konsorciem W3C. Data ve formátu XML, označovaná jako dokumenty, jsou vyjádřena jako text obohacený o značky – tagy. Značky dokument logicky rozdělují na objekty, které mohou být v sobě vzájemně vnořeny. Přirozenou reprezentací dokumentu je tedy zmíněná stromová struktura. Značka je pevně daná sekvence znaků, obsahující jméno, které charakterizuje objekt ohraničený touto značkou. Krátký příklad formátu XML je uveden na obr. 2.1. Rozšiřitelnost (eXtensible) ve zkratce XML odráží skutečnost, že jazyk neobsahuje žádnou předdefinovanou množinu jmen a významů značek použitelných pro zápis dokumentu. Uživatel může ve svém XML dokumentu použít zcela libovolné značky k vyjádření požadované struktury dat.

XML se tedy nabízí jako univerzální formát pro zápis strukturovaných dat. Konkrétní aplikace však zpracovávají data o pevně dané struktuře, XML se svou volností obsahu dokumentů se tedy nemusí na první pohled jevit jako vhodné řešení. Strukturu dokumentu lze logicky omezit pomocí tzv. *schémat*. Zjednodušeně je schéma sada značek spolu se vztahy mezi nimi, což definuje strukturu nějaké třídy dokumentů. Říkáme, že dokument vyhovuje schématu, jestliže splňuje všechna omezení obsažená ve schématu. Schéma definuje typ dokumentů, vyhovující dokument je *instancí schematu*. K zápisu schématu je k dispozici několik jazyků, uveďme například DTD, RELAX NG [22], DSD [17], XML Schema [8]. Kromě syntaktických rozdílů se jednotlivé jazyky liší také vyjadřovací silou [15, 25]; ne ve všech jazycích je například možné omezit obsah objektu pouze na slova nějakého regulárního jazyka. Hlavním přínosem schémat jsou *validace* dokumentu, což je algoritmické ověření, že dokument

```
<file-system>
  <dir>
    <name>/</name>
    <content>
      <file ref="1">
        <name>autoexec.bat</name>
      </file>
    </content>
  </dir>
  <files>
    <file id="1">
      <content></content>
    </file>
  </files>
</file-system>
```

Obrázek 2.1: Příklad formátu XML. Jednoduchá reprezentace obsahu souborového systému.

vyhovuje schématu. Součástí schéma–jazyka je také specifikace procesu validace.

Tato koncepce dělá z XML univerzální nástroj pro výměnu dat mezi aplikacemi. Aplikace pracují s podmnožinou jazyka XML definovanou schématem a mohou využívat obecné nástroje a knihovny pro manipulaci s XML, což může významně ulehčit a zrychlit vývoj.

Mezi takové nástroje se řadí například jazyk XSLT [10]. Extensible Stylesheet Language Transformations je jazyk, vyvinutý speciálně pro transformace XML dokumentů. Transformace mají velké využití při přenosu dat mezi aplikacemi. Ačkoli obsah předávané informace je jeden, její reprezentace se může na obou stranách zásadně lišit. Program v jazyce XSLT popisuje převod jednoho XML dokumentu na jiný. Tělo programu je tvořeno sadou transformačních pravidel, běh programu spočívá ve vyhledávání a následném použití pravidel podle struktury stromu vstupního dokumentu. Použitím pravidla vzniká část stromu výstupního dokumentu. K pohybu po struktuře dokumentu se v XSLT programech používá prostředků jazyka XPath.

XML Path Language (XPath) [7] je jazyk vyvinutý primárně pro adresaci částí XML dokumentu. Navržen byl pro použití v jazycích XSLT a XPointer [30]. XPath pracuje se stromovou reprezentací XML dokumentu, adresují se uzly stromu. Klíčovým výrazem XPath je *cesta* charakterizující průchod uzly stromu dokumentu, tvoří ji posloupnost *kroků*. Krok vyjadřuje přechod od jednoho uzlu stromu k jinému. XPath také obsahuje prostředky pro operace s řetězci, čísly a logickými hodnotami.

Transformace v jazyce XSLT

Vykonávání XSLT programu se zásadně liší od vykonávání programu zapsaného v nějakém z imperativních jazyků, jako je např. C nebo Pascal. V případě imperativních jazyků běh programu odráží posloupnost příkazů ve zdrojovém kódu. Volání procedur jsou identifikována jménem a jsou známa již v době překladu[1]. Naproti tomu v XSLT programu nejsou použití pravidel nijak explicitně zapsána, k použití dochází na základě tvaru vstupních dat.

Ačkoli XSLT program bývá navržen k převodu dokumentů nějaké konkrétní třídy na dokumenty nějaké jiné třídy, není program a priori svázán s žádnými schématy, která by tyto třídy popisovala. Nic nebrání program použít i na dokument, který do dané třídy nepatří. Korektní program bude korektně vykonán, vstupní dokument bude transformován, ať již je výsledek jakýkoli (například může být prázdný).

Pro zaručení správnosti transformace vzhledem ke schématům vstupu a výstupu by byl nezbytný formální důkaz, jehož provedení by však bylo pracné a s ohledem na rozsáhlost některých programů prakticky nemožné. Částečným řešením může být použití validátoru XML. Aplikace před provedením XSLT programu zvaliduje vstupní dokument ke vstupnímu schématu a následně transformovaný dokument k výstupnímu schématu. Tento přístup detekuje chyby transformačního programu a zaručuje korektní výstup aplikace, nicméně neustálé provádění validace výrazně snižuje výkon.

Jako vhodnější se jeví metoda označovaná jako statická kontrola typů (static type checking, nebo static typing). Transformační program je analyzován, zda pro libovolný vstup vyhovující schématu vstupního dokumentu generuje výstup vyhovující schématu výstupního dokumentu. Typovou analýzu programu tedy stačí provést pouze jednou, v době překladu programu. V současnosti existuje několik systémů XML transformací se zaměřením na statickou kontrolu typů. Jedná se například o Relaxer [23], HaXml [21], XDuce [29]. Z uvedených příkladů XDuce definuje zcela nový programovací jazyk, zatímco zbylé dva jsou postaveny nad tradičními jazyky jako je Java, resp. Haskell. Využití existujících programovacích jazyků spočívá v mapování schémat na typy těchto jazyků, typová kontrola transformace je pak zajištěna překladem programu. Kromě typové bezpečnosti jsou pak k dispozici i všechny vlastnosti tohoto cílového jazyka. Cena tohoto řešení spočívá v komplikované reprezentaci XML typů a hodnot, která většinou zahrnuje dodatečné vrstvy značek popisující data [29].

Přestože tyto speciálně vyvinuté transformační systémy zajišťují typovou bezpečnost, není snadné je prosadit k širšímu užívání před XSLT. Proto existují i práce věnující se statickému typování přímo pro jazyk XSLT, jako

[1] neuvažujeme-li výjimky jako ukazatele na procedury, virtuální metody

například Towards static type checking for XSLT [5]. Tozawa v ní ukázal pro podmnožinu jazyka XSLT a schéma v jazyce DTD algoritmus, který rozhoduje úlohu statické kontroly typů. Uvažovaný jazyk je proti plnému XSLT, co se týká vyjadřovací síly, omezený, není turingovsky úplný. Jak dokázal Kepser, jazyk XSLT turingovsky úplný je [12].

Kepserův důkaz se opírá sílu XSLT v manipulacích s řetězci, ukázal, že tak lze simulovat turingův stroj. Takový přístup je však velmi těžkopádný, pro „skutečné programování“ je nepoužitelný. Obecně však tento teoretický závěr ukázal, že statická kontrola typů nad neomezeným XSLT je z principu algoritmicky neřešitelná.

Analýza XSLT se však může ubírat i směrem zkoumání konkrétních programátorských chyb v programech, což má praktický přínos při vývoji ladících nástrojů. Například Dong a Bailey v práci Static Analysis of XSLT programs [1] navrhli detekci problémů jako jsou nedosažitelná pravidla a potenciální nekonečnost běhu programu. Jejich analýza je založena na aproximaci toku řízení programu ze znalosti schématu vstupu. Uvažované algoritmy postihují opět podmnožinu jazyka XSLT.

V naší práci se vydáme jako Dong a Bailey cestou detekce chyb v XSLT programech. Pokusíme se navázat na jejich závěry, zaměříme se hlavně na rozšíření uvažovaných podmnožin jazyků XSLT a XPath a zpřesnění analýzy toku řízení.

2.2 Terminologie

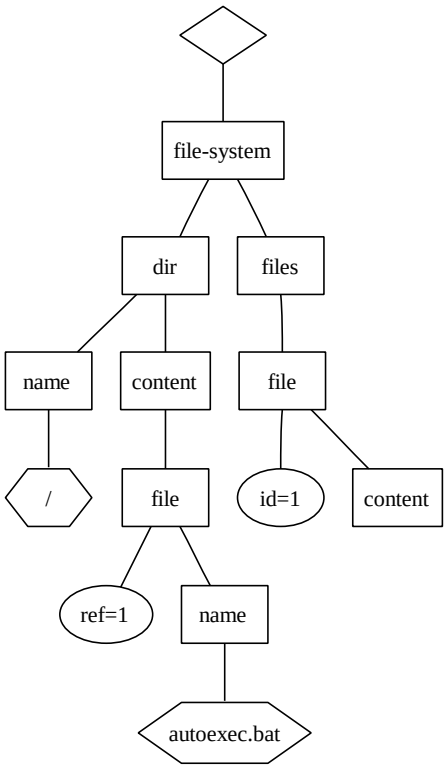
XSLT a XPath jsou, stejně jako XML, standardy konsorcia W3C. Pro popis XML dokumentů budeme v této práci uvažovat jazyk XML Schema, který je také standardem z dílny W3C.

Předpokládáme, že čtenář má alespoň základní znalosti uvedených XML technologií, přesto v této sekci jednotlivé technologie stručně popíšeme, hlavně pro ujasnění terminologie, kterou budeme v této práci používat. V textu budeme používat také některé počeštěné anglické termíny, jako například běžně používaný termín *matchovat*.

2.2.1 XML

Dokument v jazyce XML chápeme jako strom objektů. Rozlišujeme objekty *element*, *atribut*, *text*, *komentář*, *instrukce* a *kořen*. Například XML dokument z obr. 2.1 můžeme zachytit stromem na obr. 2.2.

Objekt je charakterizován svým druhem, jménem, hodnotou (obsahem) a polohou v dokumentu. Jméno je dvojice textových řetězců, *namespace* a



Obrázek 2.2: Reprezentace XML dokumentu, obr. 2.1 stromem. Použité symboly objektů: *obdélník* – element, *elipsa* – atribut, *kosočtverec* – kořen, *šestiúhelník* – text. U elementů uvádíme jména, u atributů i přiřazenou hodnotu, u textových objektů pouze hodnotu.

vlastní jméno. Jméno objektu je prázdné u objektů text, komentář a kořen. Množinu všech přípustných jmen XML objektů budeme označovat Σ^{Q}. Symbol $\lambda \in \Sigma^{\mathsf{Q}}$ zastupuje prázdné jméno objektu. Bude-li to nezbytné, ke zdůraznění složek jména použijeme symbol „:“ jako oddělovač.

Hodnota objektu záleží na jeho druhu. Hodnota objektu komentář, text, atribut i instrukce je textový řetězec. Element může obsahovat objekty text, komentář, instrukce, může mít přiřazenu množinu atributů. Navíc objekt element může obsahovat další elementy, což vytváří strukturu dokumentu. Z gramatiky jazyka XML má tato struktura charakter stromu, nejsou tedy přípustné cykly ve vnoření elementů. Jediný objekt kořen vyjadřuje počátek dokumentu, obsahuje právě jeden element, ozn. jako *kořenový element*.

Protože XML dokument je běžně modelován jako strom, o objektech se také mluví jako o uzlech stromu dokumentu, běžně se pro popis vztahu mezi objekty používají termíny jako otec, syn, apod. Umístění uzlu ve stromě od-

povídá poloze objektu v dokumentu. V textu budeme se stejným významem používat pojmy objekt a uzel.

Pokud budeme v textu zmiňovat nějaký element dokumentu, budeme jeho jméno zapisovat neproporcionálním písmem, na přiřazený atribut elementu se budeme odkazovat pomocí tečkové notace. Např. element „a“ budeme zapisovat jako `a`, přístup k atributu name zapíšeme `a.name`. Místo termínu XML dokument budeme v místech, kde nehrozí vznik nejednoznačnosti, používat pouze dokument.

2.2.2 XML Schema

XML Schema je jeden z jazyků pro zápis omezujících podmínek obsahu XML dokumentů. Konkrétní schéma zapsané v tomto jazyce je také XML dokument. Ne všechny druhy objektů jsou stejně významné, například objekty typu komentář slouží pouze čtenáři dokumentu, nemají žádný vliv na obsah, v XML Schema nejsou nijak postihnuty. Objekt text popisuje XML Schema současně s elementem, ve kterém se text má vyskytovat. Pro náhled následujících pojmů může posloužit např. schéma A.1.2 popisující strukturu dokumentu na obr. 2.1.

V XML Schema k zápisu obsahu dokumentu slouží *deklarace elementů a atributů* a *definice typů*. Typ vyjadřuje souhrn omezení obsahu objektu, deklarace spojuje druh objektu se jménem, s typem a místem výskytu v dokumentu[2]. XML Schema zavádí dva druhy typů, *complex* a *simple*. Objekt typu simple může obsahovat pouze text, objekt typu complex může obsahovat text, vnořené elementy a může mít přiřazeny atributy. Objekt atribut může být pouze typu simple, objekt element může být typu simple i complex. Budeme používat *complex element*, resp. *simple element*, k označení elementu typu complex, resp. simple. Ve schématu jsou definice vyjádřeny elementy `simpleType` a `complexType`, deklarace elementy `element` a `attribute`. K vyjádření struktury complex-elementů slouží tzv.*deklarační skupiny* `choice`, `sequence` a `all`. Skupiny v kombinaci s atributy opakování deklarace – `minOccurs` a `maxOccurs` – podobně jako regulární výraz vyjadřují všechna povolená uspořádání vnořených objektů. Definici omezení obsahu elementu skupinami, původním anglickým termínem *content–model*, budeme označovat jako *vnitřní struktura* typu.

XML Schema umožňuje při tvorbě nových typů využít dědičnost. Odvozené typy přebírají vlastnosti svých předků, které mohou buď rozšířit, nebo naopak omezit. Rozlišujeme tedy odvození *extenzí* a *restrikcí*. Zatímco u typů simple lze provést pouze restrikci a omezit tak množinu textových řetězců,

[2] v jazyce DTD jméno objektu identifikuje zároveň i typ objektu

které může objekt toho typu obsahovat, u typů complex lze provést extenzi i restrikci. Extenze rozšiřuje complex typ o nové deklarace, odpovídá klasickému dědění v objektově orientovaných jazycích. Restrikce se týká změny počtu výskytů deklarovaných elementů a atributů, což v některých případech může vést k „odebrání“ objektu z obsahu typu.

Dědičnost lze, mimo usnadnění tvorby typů, využít také v instanci schématu. Obsah elementu `a` typu t_0, lze nahradit obsahem splňujícím omezení libovolného typu t, který je odvozen z typu t_0. Informace o takové náhradě musí být v dokumentu explicitně zmíněna pomocí speciálního atributu `xsi:type`, který je přiřazen elementu `a`.

Pro pohodlnější náhrady elementů slouží *substituce*. Informace o substituovatelnosti elementů musí být uvedeny ve schématu již v deklaraci elementu, atributem `substitutionGroup`. Například je-li element `b` substituovatelný za element `a`, v instanci schématu se pak na libovolném místě, kde je očekáván element `a`, může vyskytnout element `b`. Tuto náhradu není potřeba explicitně oznamovat. Uvedená substituce vyžaduje, aby element `b` byl deklarován jako element typu t, stejného nebo odvozeného typu od t_0, kde t_0 je typ elementu `a`.

Zavedeme ještě označení *rekurzivní element* pro element e, který může v instanci schématu obsahovat element e jako svého potomka.

2.2.3 XSLT

XSLT je deklarativní jazyk, jehož programy jsou zapisovány také formou XML dokumentu, viz ukázka na obr. 2.3. Program sestává ze sady *pravidel*, která jsou vyjádřena elementy `template`; často se proto o pravidlech mluví jako o šablonách. Pravidlo je dvojice *vzor–šablona*. Vzor je výraz z podmnožiny XPath a popisuje objekty vstupního dokumentu, k jejichž transformaci může být pravidlo použito. Šablona je část stromu výstupního dokumentu obohacená o XSLT instrukce, jejichž vykonání dynamicky vytváří části výstupu a řídí běh programu, což zahrnuje použití jiných pravidel programu. Proces vytvoření podstromu podle pravidla se označuje jako *instanciace pravidla/šablony*.

Běh programu je založen na matchování vzorů pravidel a uzlů stromu vstupního XML dokumentu. Instrukce použití pravidla – `apply-templates` – vybírá množinu uzlů tohoto stromu, pro každý vybraný uzel pak hledá transformační pravidlo. Použité pravidlo je pak instanciováno s vybraným uzlem jako *kontextem*. Vzniklé části stromu dokumentu, pro každý z vybraných uzlů, jsou pak spojeny do jednoho stromu a ten je vložen na místo instrukce `apply-templates`. Uzel může matchovat více než jedno pravidlo, zpracován může být však jen jedním. K řešení konfliktu více matchujících

pravidel zavádí XSLT priority pravidel. Jedná se o číselnou hodnotu přiřazenou pravidlu, při konfliktu matchujích pravidel se vybere pravidlo s nejvyšší prioritou. Instanciace šablon probíhá rekurzivně až do okamžiku, kdy jsou všechny instrukce zpracovány. Běh programu začíná implicitním vykonáním instrukce `apply-templates` na kořen vstupního dokumentu.

Mimo pravidla se vzory je možno v XSLT programech používat i tzv. *pojmenovaná pravidla*. Pojmenované pravidlo nemá vzor, k instanciaci jeho šablony dojde při vyhodnocení instrukce volání `call-template`. Pojmenovaná pravidla a jejich použití jsou zcela totožné s procedurami tradičních programovacích jazyků. Volání pojmenovaného pravidla nemění zpracovávaný kontext, zpracovává stejný uzel jako volající. Pojmenovaná pravidla spolu s předáváním parametrů jsou ekvivalentní pravidlům se vzory a někteří programátoři volí tento procedurální přístup, toto však není zamýšlené použití jazyka. Použití pravidla se vzorem budeme přeneseně také nazývat voláním, instrukci `apply-templates` pak instrukcí volání.

Identifikace pravidla se vzorem sestává kromě vzoru ještě z tzv. *módu*, který však není povinnou součástí pravidla. Pokud je mód explicitně uveden v instrukci `apply-templates`, budou pro transformaci uzlu použita pouze pravidla s daným módem, v opačném případě se použijí pouze pravidla, která mód nemají uveden.

2.2.4 XPath

XPath je jazyk pro zápis adresace obsahu stromu XML dokumentu. Klíčovým XPath výrazem je *cesta* – PathExpr v gramatice jazyka [7] – používaná k výběru uzlů dokumentu. Cesta se skládá z posloupnosti *kroků*, které popisují přechod mezi uzly stromu.

Každý krok vybírá podmnožinu uzlů stromu, vychází přitom od uzlů, které vybral předchozí krok. Výsledek vyhodnocení cesty je pak množina vybraná posledním krokem. Uzel, vzhledem ke kterému se vyhodnocování cesty, resp. kroku, provádí, se označuje jako *kontext*. Kontext vyhodnocování je určen vnějším činitelem, který XPath využívá, pro nás je to kontext instanciace pravidla, ve kterém se výraz nachází. Speciální případ, *absolutní cesta*, jako kontext vyhodnocování používá kořen dokumentu.

Krok cesty sestává ze tří částí: *osy*, *testu* a libovolného počtu *predikátů*, kde

Osa (axis)
: je pojmenování vztahů mezi uzly stromu dokumentu. Celkem jazyk definuje 13 os. V kroku cesty osa reprezentuje primární výběr uzlů

podle daného vztahu ke kontextovému uzlu. Například osa *child* vybírá všechny syny kontextového uzlu.

Test (nodetest)
zjemňuje výběr uzlů provedený osou. Test provádí kontrolu typu, případně i jména objektu. Uzly nevyhovující testu jsou z výběru odstraněny. Například test *text()* odstraní všechny uzly, které nereprezentují objekt typu text.

Predikát (predicate)
je booleovský výraz, uzly, pro které není výraz pravdivý jsou z výběru odstraněny. Posloupnost predikátů tak funguje jako vícekrokový filtr. Predikát může obsahovat složitější testy uzlu, jako jsou porovnání textového obsahu uzlu s konstantami, hodnotami potomků, apod.

Syntaxe kroku je tvaru $axis :: nodetest[predicate_1] \ldots [predicate_n]$, jednotlivé kroky jsou odděleny symbolem lomítka. Například cesta

```
/a/descendant::b[@id]
```

vybírá elementy jména `b`, které jsou potomky kořenového elementu jména `a` a zároveň mají přiřazen atribut `id`. Příklad zároveň ukazuje zkrácený zápis XPath výrazů, osu child vyjadřující nejčastější přechod lze vynechat, symbol „@“ zastupuje výběr osou attribute.

Jazyk XPath definuje pořadí uzlů dokumentu, označované jako *document order*. Jedná se o očíslování uzlů preorder-průchodem stromu. Vzhledem k tomuto pořadí jsou osy rozděleny na *dopředné* a *zpětné* podle toho, které uzly vzhledem ke kontextu vybírají.

V naší práci budeme používat tuto klasifikaci v pozměněném významu. *Dopředná osa* vybírá uzly, které leží v podstromu kontextového uzlu. *Zpětná osa* vybírá uzly, které leží na cestě od kontextového uzlu ke kořeni stromu. Osu self označíme jako *nehybnou* a osu, která vybírá uzly neležící ani v podstromu kontextu ani na cestě z kontextu do kořene, budeme označovat přívlastkem *příčná*.

Datový model jazyka definuje čtyři datové typy – množina uzlů, logická hodnota, číslo, textový řetězec – a konverze mezi nimi. Výrazy tak mohou být i aritmetické a logické operace, manipulace s řetězci, využívající cesty pouze k získání hodnot objektů dokumentu. V XSLT se proto také XPath výrazů užívá k výpočtu hodnot objektů cílového dokumentu.

Místo termínu *XPath výraz* budeme také zkráceně používat jen *výraz*.

2.3 Cíle práce

Jak jsme již nastínili, v naší práci budeme analyzovat XSLT programy za účelem odhalování chyb. Za zajímavé z hlediska statického přístupu považujeme problémy *slepé cesty*, *nedosažitelná pravidla* a *cyklická volání pravidel*, které nyní uvedeme. Pro názornost nám poslouží jednoduchý příklad, program na obr. 2.3, který zpracovává vstupy odpovídající schématu A.1.2. Ukázkové schéma popisuje strukturu jednoduchého souborového systému, bude použito i v některých dalších příkladech.

2.3.1 Slepé cesty

Jako *slepou cestu*, budeme označovat výraz, který je vyhodnocen na prázdnou množinu objektů. Ze sémantiky XPath výrazů plyne, že se skutečně jedná o cestu, případně sjednocení cest.

Přítomnost cesty, která je při běhu programu vždy, anebo v nějakém kontextu slepá, je velmi pravděpodobně programátorská chyba. K této chybě dochází zejména

- nesprávnou adresací uzlů z neznalosti struktury vstupních dokumentů, nebo
- použitím pravidla, ve kterém se výraz nachází, při běhu programu i na jiné než zamýšlené uzly.

V našem příkladu z obr. 2.3 je slepou cestou volání

```
(13)  <xsl:apply-templates select="/files/file[@id = current()/@ref]"/>
```

Záměrně jsme v absolutní cestě vynechali krok přes kořenový element `file-system`.

Poznamenejme, že neprůchodnost cesty v některých případech chyba být nemusí, v instrukci větvení může být přímo očekávána. Obecněji, cesta, vyskytující se jako část složitějšího výrazu, může mít smysl i v případě, že je neprůchozí – specifikace XPath definuje konverze prázdné množiny uzlů na hodnotu jiných typů. Pro ilustraci uveďme trochu umělý fragment kódu (nesouvisející s příkladem)

```
<xsl:if test="concat(@a, 'b') = 'b'">...</xsl:if>
```

a uvažujme vyhodnocení v kontextu, který neobsahuje atribut `a`. Je vidět, že tento výraz má dobře definovanou hodnotu i pro neprůchozí cestu.

My budeme neprůchodnost cesty primárně považovat za chybu z neznalosti struktury. Domníváme se, že případy jako výše uvedený nepatří k čistým

```
(1)   <xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
(2)     <xsl:template match="/">
(3)       <xsl:apply-templates select="file-system/dir"/>
(4)     </xsl:template>
(5)     <xsl:template match="dir">
(6)       <xsl:apply-templates select="content/file"/>
(7)       <xsl:apply-templates select="//dir"/>
(8)     </xsl:template>
(9)     <xsl:template match="files/file">
(10)      <xsl:value-of select="content"/>
(11)    </xsl:template>
(12)    <xsl:template match="content/file">
(13)      <xsl:apply-templates select="/files/file[@id = current()/@ref]"/>
(14)    </xsl:template>
(15)  </xsl:stylesheet>
```

Obrázek 2.3: Ukázka zkoumaných problémů – XSLT program obsahující zkoumané chyby: slepou cestu, nedosažitelné pravidlo, cyklus ve vzájemném volání pravidel.

programátorským praktikám, a proto je na ně vhodné upozornit. Hledáme tedy cesty, které mohou být při běhu programu vyhodnocovány v kontextu, pro který budou neprůchodné.

2.3.2 Nedosažitelná pravidla

V XSLT programu mohou existovat pravidla, která nebudou nikdy použita, nezávisle na vstupním XML dokumentu. Přítomnost takového pravidla může opět vyjadřovat chybu v programu, případně možnost optimalizace programu pro použití na konkrétní třídě vstupních dokumentů.

Říkáme, že pravidlo XSLT programu je *nedosažitelné*, jestliže

- vzor pravidla nematchuje žádný uzel vstupního dokumentu, nebo
- program při svém běhu nezpracovává uzly vstupního dokumentu, které matchují pravidlo, nebo
- pravidlo je ve všech možných případech použití zastíněno pravidlem s vyšší prioritou.

Uvažujeme-li příklad na obr. 2.3 nedosažitelným pravidlem bude

```
(9)  <xsl:template match="files/file">,
```

zde v důsledku slepého volání, které jsme ukázali v minulém oddílu.

2.3.3 Cykly ve vzájemném volání pravidel

Jak jsme zmínili, XSLT program zpracovává vstupní dokument postupným použitím pravidel na uzly vstupního dokumentu. Klíčová instrukce běhu, `apply-templates`, vybírá uzly ke zpracování. Výběrový výraz neidentifikuje jednoznačně pravidla, která budou použita na vybrané uzly vstupního stromu, výběr závisí na tvaru stromu. V XSLT programu je poměrně snadné nechtěně napsat kód který na některých vstupních dokumentech povede na nekonečný běh programu.

Ukázku jsme si připravili i v tomto případě. Program z obr. 2.3 přejde do nekonečného výpočtu jakmile zpracuje volání

```
(7)  <xsl:apply-templates select="//dir"/>
```

Zkratka `//` za `/descendant-or-self::node()/` vychází z kořene, výraz pak vybere i právě zpracovávaný uzel, na který se opět použije aktuálně vykonávané pravidlo

```
(5)  <xsl:template match="dir">
```

Pokud by program obsahoval výběr `*/dir`, budou zpracovány pouze elementy `dir` z podstromu a k nekonečnému výpočtu nedojde.

Bex s kolegy ukázal, že zastavení XSLT programu je stejně jako zastavení Turingova stroje nerozhodnutelné [19]. Vydáme se tedy cestou Donga a Baileyho [1], budeme zkoumat cykly ve vzájemném volání pravidel a jejich vztah k potenciálně nekonečnému běhu programu.

Kapitola 3

Modelování

U statického přístupu odhalování problémů v programech je úspěšnost detekce zcela závislá na odhadu běhu programu na vstupních dokumentech. Jako základní krok provedeme kvazi-simulaci běhu a následně její výstup použijeme k hledání vybraných typů chyb. Předpona *kvazi* odráží skutečnost, že půjde o iterativní vyhodnocování řídících konstrukcí programu nad strukturou, která pouze modeluje vstupní dokumenty. Dále v textu si dovolíme používat přímo výraz simulace v uvedeném „kvazi-významu".

V této kapitole ukážeme obecně vztah simulace ke zkoumaným problémům, popíšeme zjednodušení XSLT a XML Schema, která přijmeme. Dále zavedeme model pro popis vstupních dokumentů spolu se semantikou vyhodnocování XPath výrazů nad tímto modelem.

3.1 Staticky analyzovatelné problémy

Nahlédněme, co můžeme o uvažovaných chybách programu rozhodnout pouze ze znalosti programu a schématu vstupu. Uvažujme nějakou reprezentaci schématu, nad kterou umíme vyhodnocovat XPath výrazy.

Nedosažitelná pravidla

Protože nemáme k dispozici posloupnost volání pravidel, pro rozhodnutí nedosažitelnosti vyjdeme z množin matchovaných objektů pravidla. Podle jejich obsahu vyhodnotíme instrukce volání a získáme tak potenciálně volaná pravidla. To zachytíme grafem a jako nedosažitelná označíme pravidla, která neleží v komponentě souvislosti, která obsahuje pravidlo zpracování kořene.

Při tomto jednoduchém přístupu přináší velkou ztrátu přesnosti implicitní pravidlo – matchuje všechny objekty a obsahuje volání pro všechny děti matchovaných objektů.

Slepé cesty

Ke každému objektu match-množiny pravidla, resp. jejímu zúžení, vyhodnotíme všechny cesty a testujeme neprázdnost výsledné množiny. Protože v případě XML Schema nejsou objekty jednoznačně typově identifikovány jménem, pravidlo může být určeno pouze pro podmnožinu ze všech objektů, které matchuje. V takovém případě toto řešení může přinést spoustu falešných slepých cest.

Cykly ve volání pravidel

Uvažujeme-li vztah potenciálního volání, zavedený u nedosažitelných pravidel, nabízí se hledání cyklů v grafu tohoto vztahu. I když provedeme restrikce podle objektů vybraných voláním, původ cyklu zůstává v match-množinách, spojitost s (nekonečným) během programu je velmi hrubá.

Provedení simulace je tedy motivováno získáním údajů o tom,

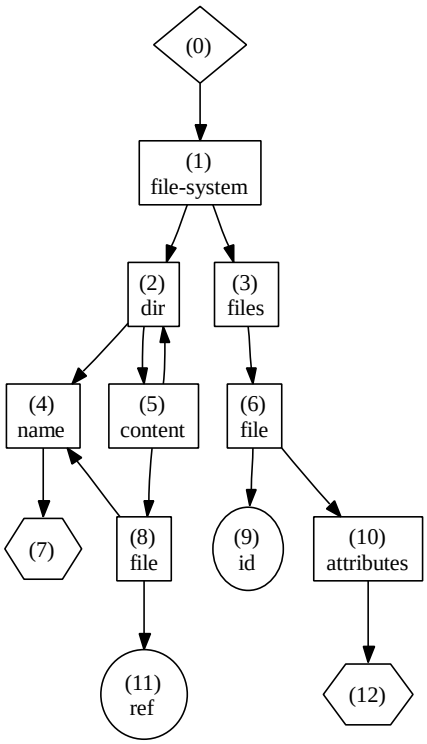
- která pravidla a pro jaké objekty mohou být skutečným programem použita
- která pravidla mohou být volána konkrétní instrukcí volání
- jakých hodnot nabývají parametry pravidel.

Abychom postihli všechny varianty vstupního dokumentu potřebujeme vhodným způsobem reprezentovat údaje ze schématu vstupů. Formalizaci modelu dokumentu uvedeme v oddílu 3.3.1. Pro představu uvažujme model jako graf na obr. 3.1 – jedná se o reprezentaci schématu A.1.2, modelem plain.

Simulace běhu programu probíhá analogicky zpracování programu XSLT procesorem. Začíná výběrem pravidla zpracovávající kořenový objekt a přiřazení tohoto objektu pravidlu jako kontext vyhodnocování. Ve vybraném pravidle vzhledem ke kontextu vyhodnotíme instrukce volání pravidel, čímž získáme přenos kontextu. K novým kontextům vybereme pravidla ke zpracování. Simulace probíhá do stavu, kdy jsou všechny kontexty zpracovány. Během simulace konstruujeme graf toku řízení, který zachytí

- všechna použitá pravidla v běhu jako vrcholy,
- kontexty, pro které byla jednotlivá pravidla použita,
- volání pravidel jako hrany mezi vrcholy,
- přenos kontextů jako rozšiřující údaj hran.

Ukázka grafu toku řízení je uvedena na obr. 3.2. Graf je výsledkem simulace běhu programu z obr. 2.3 nad modelem z obr. 3.1.

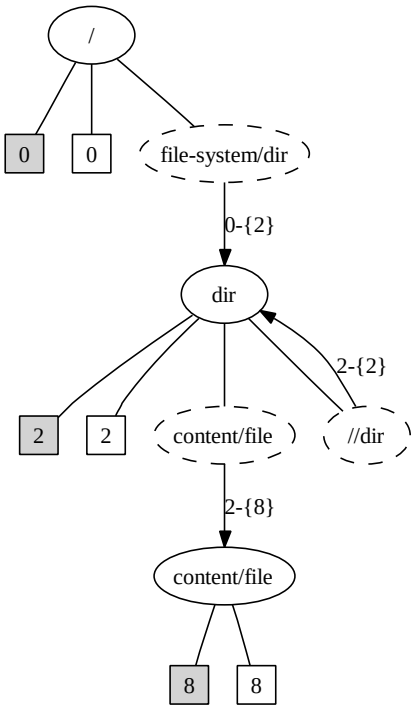


Obrázek 3.1: Graf možného modelu dokumentů (model plain) pro schéma A.1.2. Použité symboly: *obdélník* – element, *elipsa* – atribut, *kosočtverec* – kořen, *šestiúhelník* – text. Textový popisek u atributu a elementu je jméno objektu. Číselný údaj je identifikátor objektu.

3.2 Zavedená zjednodušení

Vybrané chyby XSLT programů úzce souvisejí s vyhodnocováním XPath výrazů. Užití výrazů v XSLT se týká převážně adresace objektů vstupního XML dokumentu, které jsou programem zpracovány. Jazyk XPath poskytuje prostředky pro přesný výběr objektů, který zohledňuje i detaily dokumentu. Ve výrazech lze přistupovat přímo k textové hodnotě objektu, např. porovnat hodnotu atributu s konstantou. Bez znalosti konkrétního vstupu takové výrazy z principu nelze vyhodnotit. Ačkoli XML Schema umožňuje omezit textový obsah objektů, např. regulárním výrazem, zpracování vstupů na této úrovni detailu by odpovídalo množinovým operacím nad jazyky a tedy by bylo značně náročné. Na tak nízké úrovni detailu naše analýza pracovat nebude, a proto se zaměříme na vztah programu a struktury vstupního dokumentu.

Provedeme nyní zjednodušení jazyků XML Schema a XSLT, která od-



Obrázek 3.2: Zkonstruovaný tok řízení programu z obr. 2.3 na modelu z obr. 3.1. Použité symboly: *elipsa* – pravidlo programu, *čárkovaná elipsa* – instrukce volání pravidla, *bílý obdélník* – kontexty pravidla jako seznam identifikátorů objektů modelu, *šedý obdélník* – match množina pravidla, *popisek hrany* – kontextový přenos voláním.

straní části postihující právě tyto nízkoúrovňové detaily. Redundantní konstrukce, které bývají zavedeny pro pohodlí uživatelů, vyjádříme základními, což později povede k jednoduššímu návrhu algoritmů.

3.2.1 Zjednodušení jazyka XML Schema

Následujícími zjednodušeními ukážeme, co z jazyka XML Schema je pro naši analýzu významné a z čeho budeme vycházet při modelování vstupních dokumentů. Pro názornost shrneme efekt zjednodušení v podobě gramatiky pomocného jazyka ZSchema. Gramatika je uvedena na obrázku 3.3, atributy jsou pouze pro úsporu místa vyjádřeny jako terminály.

```
Schema ::= <schéma> ElementDeclaration TypeDefinition* </schéma>

TypeDefinition ::= <type name ( restriction | extension ) text >
                   TypeContentMain? AttributeDeclaration*
                   </type>

TypeContentMain ::= <sequence minOccurs maxOccurs> TypeContent* </sequence>
                  | <choice minOccurs maxOccurs> TypeContent* </choice>
                  | <all minOccurs maxOccurs> TypeContent* </all>

TypeContent ::= <sequence minOccurs maxOccurs> TypeContent* </sequence>
              | <choice minOccurs maxOccurs> TypeContent* </choice>
              | ElementDeclaration

ElementDeclaration ::= <element name type minOccurs maxOccurs />

AttributeDeclaration ::= <attribute name minOccurs maxOccurs />
```

Obrázek 3.3: Gramatika jazyka ZSchema. Hodnoty atributů `minOccurs`, `maxOccurs` v deklaraci atributu jsou triviálním přepisem hodnoty atributu `use` z XML Schema.

Schema v jednom dokumentu

Schema dokumentu může být v jazyce XML Schema zapsáno ve více dokumentech. Pro kombinaci více schémat obsahuje jazyk mechanismy vkládání obsahu jiných dokumentů. V případě, že chceme popsat třídu dokumentů, které mají obsahovat objekty různých namespace, nelze se bez nich obejít. V ZSchema uvažujeme již vyřešené prvky include, import a redefine. V jednom schématu tedy mohou figurovat deklarace objektů různých namespace.

Jeden druh typu

Typ simple považujeme za speciální případ typu complex. Omezení textového obsahu objektu, které vyjadřuje typ simple není pro naši analýzu podstatné. Typ simple je tedy ekvivalentní typu complex, který má ve svém obsahu povolen výskyt textu a nemá povolen výskyt vnořených elementů ani přiřazeny atributy. Přítomnost textu je v ZSchema zachycena atributem `text` přiřazeným typu.

Konstrukce group, attributeGroup

Často opakované deklarace skupin elementů, resp. atributů umožňuje XML Schema nahradit odkazem na pojmenovanou skupinu `group`, resp. `attributeGroup`

a deklaraci tak zapsat pouze jednou. V ZSchema tyto zkratky neuvažujeme.

Globální deklarace, reference a substituční skupiny

V XML Schema je možné deklarovat element uvnitř typu odkazem na nějakou globální deklaraci elementu. Deklarace elementu tvaru

```
<element ref="e" minOccurs="i" maxOccurs="a"/>
```

bude nahrazena konstrukcí

```
<choice minOccurs="i" maxOccurs="a">
  <element name="e" type="t" minOccurs="1" maxOccurs="1"/>
  <element name="e1" type="t1" minOccurs="1" maxOccurs="1"/>
  ...
  <element name="en" type="tn" minOccurs="1" maxOccurs="1"/>
</choice>
```

kde element e je typu t a může být substituován elementy e_1, e_2, ..., e_n. Nahrazení mechanismu substituce pomocí konstrukce vnitřní struktury choice, s vyjmenovanými deklaracemi všech substitucí, vede na stejnou třídu dokumentů.

Jelikož nejsou definovány substituce atributů, odkaz na globální deklaraci atributu bychom převedli na obyčejnou deklaraci zcela přímočaře. Omezíme se však pouze na lokální definice atributů.

Každý z globálně deklarovaných elementů může v instanci schématu figurovat jako kořenový element, předpokládejme po přijatých zjednodušeních, že globálně deklarovaný element je právě jeden.

Shrneme-li uvedená zjednodušení, pro naši analýzu je zajímavá struktura vstupního dokumentu, ZSchema postihuje deklarace elementů, atributů, definice typů včetně jejich vnitřní struktury a eviduje informaci o typové hierarchii. Přechod od XML Schema k ZSchema je bezztrátový co do strukturálních vztahů mezi objekty popisovaného dokumentu.

3.2.2 Zjednodušení jazyka XSLT

Pro analýzu vybraných problémů nepotřebujeme zachytit zcela detailně vstupní program. Provedeme obdobně jako Olesen [2] následující zjednodušení, která zpřehlední algoritmy analýzy. Zjednodušený jazyk budeme nazývat *ZXSLT*, jeho gramatika je shrnuta na obrázku 3.4.

```
Program ::= <stylesheet> (XPath | TemplateRule)* </stylesheet>

TemplateRule ::= <template match mode priority precedence>
                 <param name select/>*
                 <variable name select/>*
                 Instruction*
                 </template>

Instruction ::= <apply-templates select mode minPrecedence maxPrecedence>
                <with-param name select/>*
                </apply-templates>
              | XPath

XPath ::= <xpath select/>
```

Obrázek 3.4: Gramatika jazyka ZXSLT

Program v jednom dokumentu

Jazyk XSLT podobně jako XML Schema poskytuje mechanismy pro kombinaci programu z více dokumentů. Opět předpokládáme, že všechna vložení jsou již vyřešena. Pro zachycení importní priority zavedeme pomocný atribut `precedence`, který přiřadíme každému pravidlu. Hodnota tohoto atributu charakterizuje výskyt pravidla v importním stromě programu [10] a hraje roli při výběru pravidla pro transformaci uzlu dokumentu. Připomeňme, že importní priorita je dána postorder průchodem importního stromu.

Pro transformaci uzlu iniciované instrukcí `apply-templates` se uvažují pravidla s maximální importní prioritou z množiny matchujících. V případě instrukce `apply-imports` jsou to pravidla s maximální importní prioritou mezi pravidly, které se nacházejí v importním stromě v podstromu pod uzlem, jemuž náleží zpracovávaná instrukce `apply-imports`. V ZXSLT je instrukce `apply-imports` nahrazena instrukcí `apply-templates`. Pro docílení požadovaného chování je potřeba instrukci `apply-templates` předat informaci o použitelných pravidlech podle importního stromu.

Z povahy postorder očíslování stromu přidáme instrukci `apply-templates` atributy `minPrecedence` a `maxPrecedence`, jejichž hodnoty vymezují interval importních priorit pravidel, které se nacházejí v importním stromu pod uzlem s nahrazovanou instrukcí `apply-imports`. Tento rozsah pak zůží množinu matchujících pravidel. Normální použití instrukce `apply-templates` bude v těchto atributech obsahovat minimální a maximální importní prioritu z celého importního stromu.

Předpokládáme také, že do programu byla vložena built-in pravidla s nastavenou minimální hodnotou rozšiřujícího atributu precedence.

Jeden druh pravidel

Mimo pravidla se vzorem se v XSLT programu používají i pojmenovaná pravidla. Pro zjednodušení uvažujeme pojmenované pravidlo jako speciální případ pravidla se vzorem, kde vzor je prázdný. Zároveň nechť platí, že prázdný vzor matchuje libovolný uzel vstupního dokumentu. Instrukci `call-template` potom nahradíme instrukcí `apply-templates` na aktuálním uzlu. Jednoznačný výběr volaného pravidla zajistíme použitím jedinečného módu.

Například pravidlo

`<template name="`*templatename*`">...</template>`

a všechna volání

`<call-template name="`*templatename*`"/>`

nahradíme pravidlem

`<template match="" mode="`m`">...</template>`

a voláním

`<apply-templates select="self::node()" mode="`m`"/>`

kde m je jedinečná hodnota módu.

Hybridní XSLT pravidla (pojmenovaná pravidla se vzorem) uvažujeme rozložená na dvě, pojmenovanou verzi pravidla pak převedenou výše uvedeným způsobem na pravidlo se vzorem.

Nechť má také každé pravidlo přiřazenu prioritu – pokud není zadána explicitně, je spočtena defaultní hodnota. Potom můžeme předpokládat, že vzory pravidel jsou absolutní cesty. Vzor *pattern*, který není absolutní cesta je nahrazen výrazem

`/descendant-or-self::node()/`*pattern*,

což je ekvivalentní tomu, jak se v XSLT pracuje se vzory.

Rozklad pravidel se sjednocením ve vzoru

Pravidlo, které obsahuje ve vzoru operaci sjednocení se při řešení výběru pravidla k transformaci zpracovávaného uzlu chová jako několik samostatných pravidel. To má význam například v situaci, kdy pravidlo nemá explicitně uvedenu prioritu a dílčí výrazy vzoru mají různou prioritu.

Předpokládejme tedy, že se v ZXSLT vyskytují pravidla bez operace sjednocení. Pravidla XSLT, která tuto podmínku nesplňují jsou tedy rozložena na odpovídající počet „obyčejných“ pravidel.

Tento rozklad však není v XSLT korektní, neboť dílčí pravidla se mohou ve výběru uzlů překrývat, což přináší konflikt ve volbě pravidla znamenající chybu programu. V naší analýze je adresace uzlů zatížena přijatými zjednodušeními, konflikty nepovažujeme za chybu. Tento rozklad nám pak může dát informaci, že některá dílčí část vzoru je redundantní.

Osamostatnění vložených pravidel

Jak zmiňují například Dong a Bailey v [1], instrukci iterace `for-each` můžeme chápat jako pravidlo vložené na místo jeho jediného použití, přímo do jiného pravidla. Obdobným způsobem, jako jsme převedli pojmenované pravidlo na pravidlo se vzorem, nahradíme `for-each` instrukcí `apply-templates` a nově vytvořeným pravidlem, které obsahuje tělo instrukce `for-each`.

Výskyt iterace

```
<for-each select="s">...</for-each>
```

je nahrazen voláním

```
<apply-templates select="s" mode="m"/>
```

spolu s pravidlem

```
<template match="" mode="m">...</template>
```

kde m je jedinečná hodnota módu.

Obdobně uvažujme náhradu instrukcí větvení `if` a `when`. Narozdíl od `for-each`, nedochází při zpracování těla větvení ke změně aktuálního uzlu, výraz podmínky tedy použijeme pouze jako filtr aktuálního uzlu formou predikátu. Instrukce volání náhrady větve podmínky bude tvaru

```
<apply-templates select="self::node()[expr]" mode="m"/>
```

kde *expr* je výraz podmínky a m je jedinečná hodnota módu.

V případě větve `otherwise` struktury větvení `choose`, bude predikát obsahovat konjunkci negací podmínek větví `when`.

Při osamostatnění vložených pravidel musíme brát ohled na výskyty proměnných a parametrů uvnitř těla. Toto bude popsáno níže v sekci o proměnných a parametrech.

Instrukce xpath

Mimo instrukce řízení běhu programu obsahuje tělo pravidla instrukce pro tvorbu výstupního dokumentu. Naše analýza se tvarem výstupního dokumentu nezabývá, instrukce tvorby dokumentu, včetně literálů, jsou pro nás

zajímavé jen z hlediska obsahu XPath výrazů v jejich atributech. Tyto instrukce a literály nahradíme nově zavedenou instrukcí `xpath`, která bude obsahovat zkoumaný XPath výraz ve svém atributu `select`.

Například instrukce

```
<xsl:value-of select="@nr"/>, nebo
<myliteral myattr="item{@nr}"/>
```

budou nahrazeny instrukcí

```
<xpath select="@nr"/>
```

Proměnné a parametry

XSLT v souvislosti s proměnnými rozšiřuje datový model jazyka XPath o datový typ *result tree fragment*. Hodnota takové proměnné je strom XML objektů, vycházející přímo z podstromu elementu proměnné. Takový strom je možno použít pouze jako celek pro vložení do výstupního dokumentu, nelze se po něm pohybovat a vybírat jeho objekty. Proměnnými tohoto typu se nebudeme zabývat a jejich výskyt ve výrazech budeme považovat za *nevyhodnotitelný*. Uvažujeme pouze proměnné a parametry, jejichž hodnota je zadána výrazem, tedy prostřednictvím atributu `select`. Konkrétně jsou pro nás zajímavé výrazy vztahující se ke struktuře vstupnímu dokumentu, tedy jejich hodnotou je množina objektů dokumentu. Výskyt proměnné jiného typu budeme ve výrazech také považovat za nevyhodnotitelný.

Výrazným rysem XSLT proměnných je neměnnost jejich hodnoty[1], hodnota tedy může být přiřazena jen inicializací. Hierarchická podoba XML také definuje rozsah viditelnosti proměnných. Stínění proměnných je možné pouze mezi lokálně definovanou proměnnou pravidla a globální proměnnou.

Díky neměnnosti můžeme proměnné následujícím způsobem eliminovat:

(i) Výskyt globální proměnné ve výrazu nahradíme její definicí.

(ii) Výskyt lokální proměnné s definičním výrazem *expr* nahradíme výrazem

- (a) *expr*, je-li *expr* absolutní cesta
- (b) current()/*expr*, jinak

Ukažme, že výše uvedená zjednodušení jsou korektní.

Hodnota globální proměnné je získána vyhodnocením definičního výrazu vzhledem ke kořeni dokumentu, definiční výraz proto můžeme považovat za

[1] pojem proměnná tedy v XSLT není úplně přesný

absolutní. Hodnota absolutního výrazu je stejná pro libovolný kontext vyhodnocování, zjednodušení je tedy v případě (*i*) korektní.

Lokální proměnná definovaná absolutní cestou, je nahrazena jako globální proměnná. V případě relativní cesty nemůžeme provést nahrazení stejně přímočaře, je nutno brát v úvahu kontext, ve kterém se proměnná vyhodnocuje. Ke změně kontextu dochází, pokud se proměnná vyskytuje v predikátu kroku cesty[2], nebo pokud se výraz vyskytuje ve vloženém pravidle `for-each`. Nechť při osamostatnění pravidla `for-each` jsou lokální proměnné vyskytující se v jeho těle a definované vně tělo předány pravidlu jako parametry. Ostatní vložená pravidla nemění aktuální uzel vyhodnocování, tedy změna kontextu vzhledem k náhradě proměnných nastává jen v predikátech. V takovém případě uvedené navázání cesty na funkci current() zajistí vyhodnocení ke kontextu, pro který je pravidlo instanciováno. Korektní je tedy i zjednodušení v případě (*ii*).

Klíče

Jazyk XPath zavádí mechanismus klíčů k jednoznačnému výběru objektů v XPath výrazech. Klíč definovaný elementem `key` popisuje v match–výrazu množinu objektů zájmu. Use–výraz specifikuje objekt, který svou textovou hodnotou identifikuje objekt zájmu, typicky se jedná o nějaký id–atribut. Ve výrazech se pak používá funkce key(), která z množiny objektů zájmu vybere podle předané textové hodnoty.

Konstrukce `key` nebudeme uvažovat, volání funkce key() pak nahradíme match–výrazem z definice klíče. Konkrétní objekt, který je odkazován klíčem zjevně bude v množině, kterou vybírá match–výraz, dojde pouze ke ztrátě přesnosti.

Uvažujme následující klíč

```
<xsl:key name="MyKey" match="pattern" use="@id"/>
```

Potom výraz volání

```
<xsl:apply-templates select="key('MyKey', .)/a/b"/>
```

bude nahrazen výrazem

```
<xsl:apply-templates select="pattern/a/b"/>
```

[2]kromě kroku s osou *self*

3.3 Modelování dokumentu a vyhodnocování XPath výrazů

Se zjednodušením schématu úzce souvisí také zjednodušení procesu vyhodnocování XPath výrazů. Textová omezení objektů byla zanedbána, nyní je tedy můžeme zanedbat i u výrazů. Protože se vyhodnocování vztahuje k dokumentu, přichází vhodný okamžik k zavedení modelu dokumentu. Zjednodušení jazyka XPath pak vyplynou z funkce Eval, kterou definujeme nad modelem a množinou výrazů.

3.3.1 Model XML dokumentu

Pro korektní výstup analýzy je nezbytné, aby v sobě model dokumentu zachycoval všechny instance schématu, tj. žádná nesmí chybět. Tedy

model je uzávěr dokumentů vyhovujících schématu.

Je-li například ve schématu element `a` deklarován jako instance typu, který *může* obsahovat element `b`, pak model zaručuje, že v něm *existuje* element `a`, který obsahuje podelement `b`.[3]

S ohledem na přijatá zjednodušení, která vedla k ZSchema, odpovídá „kvalita“ modelu tomu, jak přesně je v něm zachycena vnitřní struktura typů. V jednoduchém případě, jako na obr. 3.1, jsou v modelu zcela zanedbány vztahy vyplývající ze struktury, není také nijak zohledněn fakt, že element `dir` je rekurzivní. V modelu bude mít vždy dva otce, při zpracování parent–kroku přes tento element pak nutně dochází ke ztrátě přesnosti.

Detaily použitých modelů shrneme v následujícím oddílu, nyní model popíšeme pouze z hlediska „rozhraní“, které poskytuje vyhodnocovací funkci Eval. Definujme model XML dokumentu jako pětici $(O, \mathit{objtype}, \mathit{objname}, \mathit{axis}, \varrho)$, kde

O
: je konečná množina objektů modelovaného dokumentu.

$\mathit{objtype} : O \to \{\mathit{element}, \mathit{attribute}, \mathit{text}, \mathit{root}\}$
: identifikuje druh XML objektu.

[3] Důraz na existenci uvádíme záměrně, v žádném případě nevyžadujeme aby každý element `a` odpovídající dané deklaraci obsahoval podelement `b`. To by naopak okamžitě zavinilo ztrátu přesnosti vyhodnocení příčných os v případě konstrukce `choice` vnitřní struktury typu.

$objname : O \to \Sigma^{\mathsf{Q}}$
: vrací jméno pojmenovaného objektu. Pro objekty typů *text* a *root* vrací λ.

$axis : axisname \mapsto O \times O$
: přiřazuje ke jménu XPath–osy binární relaci nad O, která zachycuje vztahy mezi objekty modelu významově odpovídající výběru osy.

$\varrho \in O$
: je *kořen* modelu.

Poznamenejme, že uvedené rozhraní omezuje skutečný model pouze konečností množiny O. Podmínku konečnosti budeme potřebovat pro zajištění konečnosti kvazi-simulace běhu analyzovaného programu.

Podmínka konečnosti množiny O v souvislosti s rekurzivními elementy přináší ztrátu přesnosti bez ohledu na konkrétní model. Mějme rekurzivní element e a objekty o_1, o_2, …, o_n reprezentující e v modelu. Protože jich je konečný počet, existují i, j, že

$$\langle o_i, o_i \rangle \in \underbrace{axis(\text{child}) \circ \ldots \circ axis(\text{child})}_{j}$$

Objekt modelu o_i tedy obsahuje sám sebe jako svého potomka.

3.3.2 Zjednodušení XPath a vyhodnocování funkcí Eval

Na základě zjednodušení schématu přijatých v oddílu 3.2.1 se při vyhodnocování výrazů omezíme pouze na cesty. Výraz jiného typu než cesta budeme automaticky považovat za nevyhodnotitelný.

Nejprve ale rozšíříme množinu jmen objektů

$$\overline{\Sigma^{\mathsf{Q}}} = \Sigma^{\mathsf{Q}} \cup \{*:*\} \cup \{n:* \mid n \text{ je identifikátor namespace}\}$$

a zavedeme pomocnou booleovskou operaci matchování jména $\approx$.

$$\approx \; : \; \overline{\Sigma^{\mathsf{Q}}} \times \overline{\Sigma^{\mathsf{Q}}} \to \{1, 0\}$$

$$a{:}b \approx c{:}d = \begin{cases} 1, & a = c \wedge b = d \\ 1, & a = c \wedge (b = * \vee d = *) \\ 1, & (a = * \wedge b = *) \vee (c = * \wedge d = *) \\ 0, & \text{jinak} \end{cases}$$

kde $a{:}b, c{:}d \in \overline{\Sigma^{\mathsf{Q}}}$.

Vyhodnotitelnou podmnožinu XPath ukážeme definicí funkce

$$\text{Eval} : \Phi \times 2^O \times \mathcal{M} \rightarrow 2^O \cup \{\mathbb{U}\}$$

kde Φ označuje množinu všech výrazů, mapování $\mathcal{M} : \Sigma^{\mathsf{Q}} \rightarrow 2^O$ sdružuje jména parametrů pravidel, s jejich hodnotou. Hodnota $\mathbb{U}$ je vyhrazena pro výrazy, které nejsou cesty.

O vyhodnocovaném výrazu předpokládáme, že

- všechny zkratky syntaxe jsou rozvinuty do plného tvaru.
- všechny testy jména objektu ve výrazu jsou v plně kvalifikovaném tvaru, tj. vyjádřeny prvkem množiny $\overline{\Sigma^{\mathsf{Q}}}$.

Vyhodnocení výrazu definujme jako posloupnost vzájemného volání dílčích funkcí – E^{path}, E^{axis}, E^{test}, E^{pred}a E^{013}. Definice je uvedena na obr. 3.5.

Zanedbání funkcí

Funkce v jazyce XPath slouží převážně k operacím nad detaily dokumentů, jako jsou např. řetězcové manipulace s textovou hodnotou uzlů, nebo dotazy na pozici objektů ve stromu dokumentu. V důsledku přijatých omezení nejsou tyto údaje v modelu zachyceny, funkce až na výjimky budeme považovat za nevyhodnotitelné.

Výjimkami budou funkce current() a booleovské funkce true(), false() a not().

Poznámka:
Některé řetězcové funkce jako local-name(), či namespace-uri() zůstávají vyhodnotitelné i s přijatými omezeními. Jejich zpracováním bychom určitě dosáhli větší přesnosti, přínost v tomto směru však nepovažujme za zásadní, a tudíž se jimi nebudeme zabývat.

Funkce current()

V programu se ve výrazech často odkazujeme na aktuální kontext pravidla, pro tento případ rozšiřuje XSLT jazyk XPath o funkci current(). Při vyhodnocování výrazu se přechodem os kontext mění, informaci o aktuálním kontextu pravidla tedy uchováme v posledním parametru všech dílčích funkcí.

$$\mathrm{Eval}(e, \mathcal{C}, \mathcal{M}) = \bigcup_{c \in \mathcal{C}} \mathrm{Eval}(e, c, \mathcal{M}, c)$$

$$\mathrm{Eval}(e, \mathcal{C}, \mathcal{M}, c) = \begin{cases} \mathrm{E}_e^{path}(\mathcal{C}, \mathcal{M}, c), & e \text{ je cesta} \\ \mathbb{U}, & \text{jinak} \end{cases}$$

$$\mathrm{E}_p^{path}(\mathcal{C}, \mathcal{M}, c) = \begin{cases} \mathrm{Eval}(e_1, \mathcal{C}, \mathcal{M}, c) \cup \mathrm{Eval}(e_2, \mathcal{C}, \mathcal{M}, c), & p = e_1 | e_2 \\ \mathrm{E}^{path}_{\chi::\tau[\pi_1]\ldots[\pi_n]}(\mathrm{Eval}(e, \mathcal{C}, \mathcal{M}, c), \mathcal{M}, c), & p = e/\chi :: \tau[\pi_1]\ldots[\pi_n] \\ \mathrm{E}^{pred}_{\pi_n}(\ldots \mathrm{E}^{pred}_{\pi_1}(\mathrm{E}^{test}_{\tau}(\mathrm{E}^{axis}_{\chi}(\mathcal{C}, \mathcal{M}, c)))\ldots), & p = \chi :: \tau[\pi_1]\ldots[\pi_n] \\ \{\varrho\}, & p = \lambda \\ \mathcal{M}(x), & p = \$x \\ c, & p = \mathrm{current}() \\ O, & \text{jinak} \end{cases}$$

$$\mathrm{E}_{\chi}^{axis}(\mathcal{C}, \mathcal{M}, c) = \{n \in O \mid \exists m \in \mathcal{C} : \langle m, n \rangle \in \mathit{axis}(\chi)\}$$

$$\mathrm{E}_t^{test}(\mathcal{C}, \mathcal{M}, c) = \begin{cases} \mathcal{C}, & t = \mathrm{node}() \\ \{n \in \mathcal{C} \mid \mathit{objtype}(n) = \mathit{text}\}, & t = \mathrm{text}() \\ \{n \in \mathcal{C} \mid \mathit{objname}(n) \approx t\}, & t \in \overline{\Sigma^{\mathsf{Q}}} \\ \emptyset, & \text{jinak} \end{cases}$$

$$\mathrm{E}_e^{pred}(\mathcal{C}, \mathcal{M}, c) = \{n \in \mathcal{C} \mid \mathrm{E}_e^{013}(n, \mathcal{M}, c) \in \{1, ?\}\}$$

$$\mathrm{E}_e^{013}(n, \mathcal{M}, c) = \begin{cases} \mathit{nonempty}(\mathrm{E}_e^{path}(\{n\}, \mathcal{M}, c)), & e \text{ je cesta} \\ \mathrm{E}^{013}_{e_1}(n, \mathcal{M}, c)\, \mathrm{op}_3\, \mathrm{E}^{013}_{e_2}(n, \mathcal{M}, c), & e = e_1 \,\mathrm{op}\, e_2 \wedge \mathrm{op} \in \{\texttt{and}, \texttt{or}\} \\ \mathrm{E}^{013}_{e_1}(n, \mathcal{M}, c)\, \mathrm{op}_3\, \mathrm{E}^{013}_{e_2}(n, \mathcal{M}, c), & e = e_1 \,\mathrm{op}\, e_2 \wedge \mathrm{op} \in \{\texttt{=}, \texttt{!=}\} \wedge \\ & \quad \text{nejvýše jeden z výrazů } e_1, e_2 \text{ je cesta} \\ \neg_3 \mathrm{E}^{013}_{e_1}(n, \mathcal{M}, c), & e = \mathrm{not}(e_1) \\ 0, & e = e_1 \,\mathrm{op}\, e_2 \wedge \mathrm{op} \in \{\texttt{=}, \texttt{!=}\} \wedge e_1, e_2 \text{ je cesta} \wedge \\ & \quad (\mathrm{E}^{path}_{e_1}(n, \mathcal{M}, c) = \emptyset \vee \mathrm{E}^{path}_{e_2}(n, \mathcal{M}, c) = \emptyset) \\ 0, & e = \mathrm{false}() \\ 1, & e = \mathrm{true}() \\ ?, & \text{jinak} \end{cases}$$

Obrázek 3.5: Definice funkce Eval

Vyhodnocení predikátů

Obsah predikátu filtrující kroky cesty je libovolný výraz, derivace neterminálu Expr [7]. Filtraci rozhoduje hodnota predikátu konvertovaná na booleovskou hodnotu.

Musíme však počítat s podvýrazy nevyhodnotitelnými v důsledku přijatých omezení. Nevyhodnotitelný operand logické operace převedeme na hodnotu *nevím* – „?“ – tříhodnotové logiky, ve které budeme logické operace vyhodnocovat. Filtrovaný objekt pak bude ve výběrové množině ponechán, pokud je hodnota predikátu *pravda* nebo *nevím*. Tabulky hodnot operací tříhodnotové logiky uvádíme pro úplnost na obr. 3.6.

$\texttt{and}_3$	0	1	?
0	0	0	0
1	0	1	?
?	0	?	?

$\texttt{or}_3$	0	1	?
0	0	1	?
1	1	1	1
?	?	1	?

$\neg_3$	0	1	?
	1	0	?

$=_3$	0	1	?
0	1	0	?
1	0	1	?
?	?	?	?

$\texttt{!=}_3$	0	1	?
0	0	1	?
1	1	0	?
?	?	?	?

Obrázek 3.6: Tabulky logických operací a relací v tříhodnotové logice.

Konverze množiny uzlů na logickou hodnotu je v XPath přímočará, jedná se o test neprázdnosti. Predikáty často obsahují testy struktury okolí kontextového uzlu, vyjádřené jako logická operace nad cestami, například výraz

```
a[@z and not(b)]
```

vybírá elementy jména `a`, které mají atribut `z` zároveň neobsahují podelement `b`. V případě predikátů tedy má smysl vyhodnocovat kromě cest i logické operace.

Jelikož schéma v definici vnitřní struktury typů povoluje deklarovat nepovinné objekty (hodnotou `minOccurs` = 0), musíme tuto nepovinnost zohlednit při konverzi na logickou hodnotu. To je nezbytné pro korektní vyhodnocení funkce not(). Vyjděme z předchozího příkladu. Uvažujme, že element `b` je nepovinný v `a`. Postupným vyhodnocením podvýrazů nám dotaz na přechod k elementu `b` dává objekt modelu reprezentující tento element, tedy neprázdnou množinu. Přímočará konverze pak dává 1, a tedy hodnota vrácená funkcí not() je 0. To ale není korektní, neboť element `b` je nepovinný, tudíž v nějaké instanci schématu může existovat element `a`, který nemá podelement `b`.

Dodefinujme proto rozhraní modelu o mapování

$$\mathit{nonempty} : 2^O \to \{1, 0, ?\}$$

které rozhoduje, zda je množina uzlů neprázdná. Vyhodnocovací funkce tedy zůstává oproštěna od detailů použitého dokumentu.

Vyhodnocení cesty nad modelem svým významem odpovídá dotazu, zda je průchod stromem uskutečnitelný aspoň v jednom z dokumentů. Zjednodušení byla provedena tak, abychom v případě že nedokážeme rozhodnout, žádnou cestu neopustili, funkce Eval tedy na tento dotaz dává odpovědi „možná ano“ a „ne“.

3.4 Uvažované modely

Abychom demonstrovali nezávislost algoritmu kvazi-simulace na modelování vstupů, zavedli jsme dva modely, *plain* a *tree*, a jejich rozšíření $plain^{\mathsf{m}}$, resp. $tree^{\mathsf{m}}$. Připomeňme, že při konstrukci modelu vycházíme ze schématu zjednodušeného do jazyka ZSchema.

3.4.1 Společné principy vytvoření modelu

Při konstrukci modelů začneme vytvořením objektu kořene, pod který zavěsíme kořenový element. V ZSchema je to jediný globálně deklarovaný element. Rekurzivním způsobem vytvoříme instanci typu. Provedení je závislé na konkrétním modelu, obecně se jedná o vytvoření synovských objektů a následné vytvoření vazeb na tyto objekty.

Instanciace v sobě implicitně zahrnuje kombinaci deklaračních skupin vnitřní struktury complex-typu podle dědičnosti. Abychom měli možnost zkoumat chování programu na vstupu, ve kterém se používá změna typu elementu formou atributu `xsi:type`, zavedeme pojem *uzávěr* typu.

Mějme následující typovou hierarchii.

$$B_1 - B_2 - \ldots - B_n - X - \left\langle \begin{array}{l} E_1 - \ldots \\ E_2 - \ldots \\ \vdots \\ E_m - \ldots \end{array} \right.$$

Nechť D_T označuje hlavní deklarační skupinu typu T. Uzávěr typu X, ozn. $\overline{X}$, vyjádříme ve smyslu deklarační skupiny $D_{\overline{X}}$ jako

```
<sequence minOccurs="1" maxOccurs="1">
  D_X
  <choice minOccurs="0" maxOccurs="1">
    D_E1 D_E2 ... D_Em
  </choice>
</sequence>
```

Použitá deklarační skupina `choice` s uvedeným rozsahem opakování přesně vyjadřuje možnost náhrady nejvýše jedním odvozeným typem E_i.

Vnitřní struktura typu X má pak v okamžiku instanciace tvar

```
<sequence minOccurs="1" maxOccurs="1">
  D_B1 D_B2 ... D_Bn D_x
</sequence>
```

kde x zastupuje X, resp. $\overline{X}$, podle toho zda modelujeme i možnost změn typu.

Pro popis modelů zavedeme následující pomocné pojmy.

Nechť e_d a e'_d jsou elementy `element`*, a t element* `type` *ve schématu. Říkáme, že e'_d* deklaruje syna *elementu e_d, jestliže se e'_d vyskytuje ve schématu jako XML-potomek t, kde e_d.*`type` $= t$.`name`*.*
Analogicky uvažujeme pojem a_d deklaruje atribut *elementu e_d, pro a_d element* `attribute`*.*

3.4.2 Model plain

Jednoduchý model plain jsme bez podrobnějšího vysvětlení už ukázali v předchozím oddílu, viz obr. 3.1. Model zachycuje pouze vztah otec–syn mezi elementem a jeho obsahem. Nejsou uvažovány složitější vztahy definované deklaračními skupinami typu elementu spolu s atributy opakování `minOccurs` a `maxOccurs`.

Konstrukce je provedena průchodem deklarací a definic schématu do hloubky s preorder tvorbou objektů. Při průchodu se využívá množiny zkonstruovaných element-objektů jako řídící struktury. Vytváříme-li instanci e deklarace elementu e_d a existuje objekt e' pro deklaraci e'_d, kde e'_d deklaruje syna e_d, vytvoříme synovskou vazbu z e na e'. Pokud neexistuje, rekurzivně vytvoříme instaci e' a následně synovskou vazbu.

Prvky rozhraní modelu popíšeme ve vztahu k výchozímu schématu následovně.

Množina objektů

Množinu objektů O vyjádříme jako sjednocení různorodých množin objektů stejného druhu, $O = \{\varrho\} \cup E \cup A \cup T$.

- Kořenový objekt ϱ není ničím zvláštní.
- Elementy z množiny E budou dvojice $\langle e_d.\texttt{name}, e_d.\texttt{type}\rangle$, pro deklarace elementu e_d ve schématu. $E \subseteq \Sigma^{\mathsf{Q}} \times \Sigma^{\mathsf{Q}}$.
- Objekty atributů, A, zachytíme ve vztahu k elementu, kterému náleží, jako trojice $\langle e_d.\texttt{name}, e_d.\texttt{type}, a_d.\texttt{name}\rangle$, kde a_d deklaruje atribut elementu e_d.
- Obdobně jako množinu atributů definujeme množinu textových objektů, T. Jelikož textové objekty nemají jméno, poslední člen trojice položíme λ.

Jméno a typová identifikace

Přiřazení jména a typu objektům, *objname* a *objtype*, vyplývají přímo z definic objektů.

Relace mezi objekty

Relace *axis* vyjádříme, podobně jako Gottlob s kolegy formalizoval XPath osy, relacemi nad množinou objektů dokumentu [9]. Zadefinujeme vztahy attribute a child, z child potom vyjádříme zbylé vztahy.

- Vztah mezi objekty, odpovídající XPath ose attribute, vyjádříme snadno jako $\mathit{axis}(\text{attribute}) = \{\langle e, a\rangle \mid e = \langle n, t\rangle \in E \wedge a = \langle n, t, \cdot\rangle \in A\}$
- Vztah child zachycuje elementy a jejich textové objekty obdobně jako vztah attribute elementy a atributy. Dále objekt kořen a kořenový element. Zejména pak vztah mezi elementy vzájemně:
  $\mathit{axis}(\text{child}) \supseteq \{\langle e_1, e_2\rangle \mid$ konstrukce vytvořila synovskou vazbu z e_1 na $e_2\}$

Nechť zápis R^n označuje mocninu relace ve významu zřetězení. R^+, resp. R^*, vyjadřuje tranzitivní, resp. reflexivní a tranzitivní, uzávěr relace R. Potom definujeme

$$\begin{aligned}
&\text{self} = \text{child}^0\\
&\text{parent} = \text{child}^{-1}\\
&\text{descendant} = \text{child}^+\\
&\text{descendant-or-self} = \text{child}^*\\
&\text{ancestor} = \text{parent}^+\\
&\text{ancestor-or-self} = \text{parent}^*\\
&\text{preceding-sibling} = \text{following-sibling} = \text{parent} \circ \text{child}\\
&\text{following} = \text{preceding} = \{\langle \varrho, \varrho\rangle\} \circ \text{descendant}
\end{aligned}$$

V případě dopředných a zpětných os jsou výše uvedené relace zadefinovány zcela přirozeně. Neuchováváme informace o možných pořadí dětí elementu, zkreslení je v případě os following a preceding velmi výrazné. Nutno však podotknout, že příčné osy nebývají v programech tak časté.

Omezení počtu výskytů elementů neuchováváme, předpokládáme, že deklarace povoluje libovolný počet výskytů. Mapování *nonempty* proto zadefinujeme jako

$$\mathit{nonempty} = \{\langle \emptyset, 0\rangle\} \cup \{\langle M, ?\rangle \mid M \subseteq O \wedge M \neq \emptyset\}$$

což říká, že prázdná množina je určitě prázdná, jinak nevíme.

3.4.3 Model tree

Předchozí model obsahuje pouze jeden objekt pro zachycení elementu deklarovaného se jménem *name* a typem *type*. Tento objekt elementu, e, je potom ve vztahu child ke všem elementům e', jestliže e_d deklaruje syna elementu e'_d, kde e_d a e'_d jsou odpovídající deklarace ve schématu. Jelikož jsme vztah parent zadefinovali pomocí child, okamžitě dostáváme ztrátu přesnosti tohoto vztahu. To je patrné na obr. 3.1, kde objekt $\texttt{name}_4$ má dva otce, $\texttt{dir}_2$ a $\texttt{file}_8$.

Model tree je navržen právě pro odstranění tohoto nedostatku. Ani v tomto modelu neuvažujeme vztahy vyplývající z deklaračních skupin vnitřní struktury typu.

Konstrukce modelu je stejná jako v případě modelu plain až na řídící strukturu. V případě modelu tree použijeme pro vytvořené element-objekty zásobník, ze kterého při návratu z rekurze odebíráme právě instanciovaný element. Vzniká tak kvazi-strom, kde stromová struktura může být porušena pouze zpětnými hranami v případě rekurzivních elementů. Obrázek 3.7 zachycuje schéma A.1.2 v tomto modelu.

Popis rozhraní modelu se od modelu plain liší pouze v množině objektů:

- Abychom rozlišili objekty elementů stejného jména a typu v nezávislých podstromech, rozšíříme objekt elementu o číselný identifikátor, jehož hodnotu v průběhu konstrukce získáme z nějakého inkrementálního generátoru[4]. Máme tedy

  $$E \subseteq \Sigma^{\mathsf{Q}} \times \Sigma^{\mathsf{Q}} \times \mathbb{N}$$

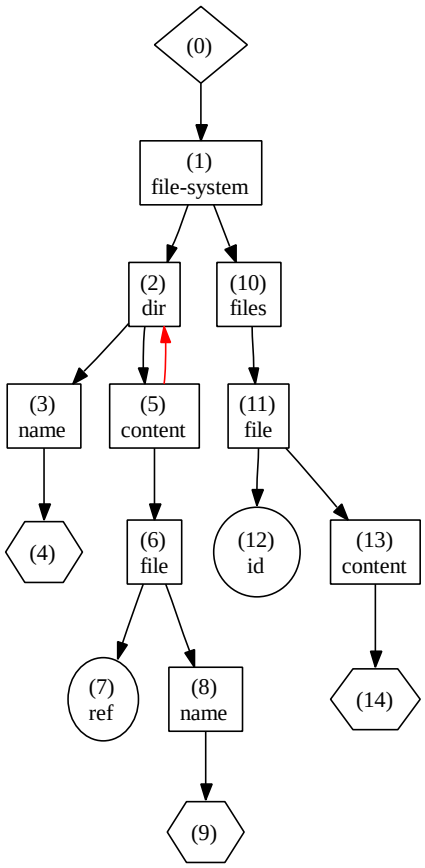
- Protože objekty atributů, A, a textové objekty, T, se v modelu vyskytují v závislosti na objektech elementů, jejich reprezentaci také rozšíříme o tuto složku.

Vztahy mezi objekty v tree vychází jako v plain také ze vztahů child a attribute, které jsou dány konstrukcí. Vyšší přesnosti vztahu parent je docíleno tím, že elementy, až na rekurzivní případy, „nesdílí své syny“.

3.4.4 Rozšíření plain$^{\mathsf{m}}$ a tree$^{\mathsf{m}}$

V rozšířených modelech plain$^{\mathsf{m}}$ a tree$^{\mathsf{m}}$ jsme se pokusili odlišit povinné objekty od nepovinných. Povinný objekt modelu, o, vyjadřuje omezení otcovského objektu o': vyskytne-li se ve skutečném dokumentu objekt odpovídající o', musí se v dokumentu jako jeho syn vyskytovat objekt odpovídající o.

[4] Aby metoda konstrukce fungovala správně, prohledávání zásobníku po tomto rozšíření uvažujeme pouze podle složek jména a typu



Obrázek 3.7: Graf modelu tree pro schéma A.1.2. Červená označuje zpětnou hranu přechodu do rekurzivního elementu.

Objekty množin elementů a atributů, E, resp. A, rozšíříme o příznak *mandatory*, který bude vyjadřovat tuto povinnost výskytu. Uvažujme e_d deklaraci elementu a k ní objekt $e \in E$ z konstrukce modelu plain, resp. tree$^{\mathsf{m}}$.

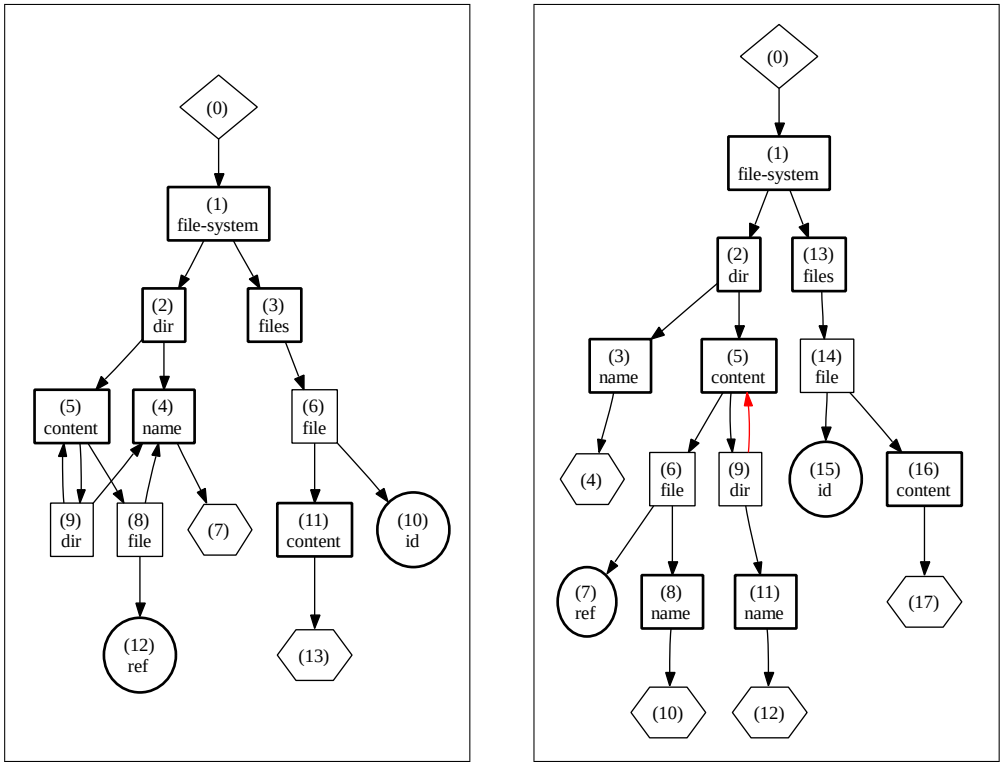
$$e.mandatory = \begin{cases} 1, & e_d.\texttt{minOccurs} > 0 \text{ a na cestě } p \text{ v XML stromě od } e_d \\ & \text{ke kořenu neexistuje element } \texttt{choice} \text{ a všechny} \\ & \text{elementy } \texttt{sequence} \text{ a } \texttt{all} \text{ na } p \text{ mají } \texttt{minOccurs} > 0. \\ 0, & jinak \end{cases}$$

Poznamenejme, že uvedené rozdělení dělá automaticky kořenový element povinným.

Potom mapování *nonempty*, rozhodující neprázdnost množiny objektů definujeme následovně

$$nonempty(M) = \begin{cases} 0 & M = \emptyset \\ ? & M \neq \emptyset \wedge \forall_{m \in M} : m.mandatory = 0 \\ 1 & M \neq \emptyset \wedge \exists_{m \in M} : m.mandatory = 1 \end{cases}$$

Na obrázku 3.8 je modelováno schéma A.1.2. Uvedené rozšíření demonstruje element `dir`, který se zde vyskytuje ve dvou instancích, jako povinný $\texttt{dir}_2$ zastupující kořen souborového systému a jako nepovinný $\texttt{dir}_9$.



Obrázek 3.8: Graf modelu plain^{m} *vlevo* a tree^{m} *vpravo*, pro schéma A.1.2. Povinné objekty jsou zvýrazněny silným rámečkem.

Kapitola 4

Algoritmy analýzy

Nyní, když máme k dispozici formalizovaný popis vstupních dat, popíšeme princip analýzy zkoumaných chyb. Jako základ uvedeme algoritmus kvazi-simulace konstruující graf toku řízení programu a následně postup detekce chyb.

4.1 Konstrukce grafu řízení

Model dokumentu jsme zadefinovali jako uzávěr instancí schématu. Provedeme-li nad modelem simulaci běhu programu, dostaneme graf toku řízení, který je odhadem uzávěru všech běhů programu nad instancemi schématu. To znamená, že k pravidlům dostáváme seznam objektů, pro jejichž zpracování může být pravidlo použito.

Idea simulace představená v předchozí kapitole pro jednoduchost neobsahovala zpracování parametrů.

Při běhu XSLT programu je hodnota parametru pravidla spjata s objektem, k jehož zpracování bylo pravidlo použito. Předávání hodnot parametrů pravidlům také není povinné, pokud není hodnota ve volání předána, použije se implicitní. Může také dojít k situaci, kdy je pravidlo použito pro daný objekt několikrát, s různými parametry. Skutečný kontext použití pravidla tedy není pouze objekt, pro který je instanciováno, kontext tvoří objekt spolu s vektorem hodnot parametrů.

Protože zanedbáváme detaily dokumentů, hodnota parametru je pro nás podmnožina objektů dokumentu. Pokud budeme vyžadovat co nejpřesnější zachycení propagace parametrů, hrozí exponenciální nárust počtu zpracovaných kontextů. Přímý důsledek je potom až exponenciální nárust doby běhu simulace. Pro rozhodnutí vybraných problémů programu potřebujeme znát podobu toku řízení, exponenciální nárust potom hrozí i v případě prostorové

složitosti.

Provedeme tedy následující zjednodušení. Hodnoty parametrů pravidel nebudeme považovat za součást kontextu, budeme je uvažovat bez přímé vazby na objekty instanciace pravidel. Půjde tedy o další formu uzávěru, získáme informaci o tom, s jakými parametry mohou kdy být objekty pravidlem zpracovávány.

4.1.1 Graf řízení

Graf řízení programu formalizujeme pětic (N, E, C, P, F), kde

N
: množina vrcholů odpovídá pravidlům programu. V rámci grafu řízení budeme uvažovat pojmy vrchol a pravidlo ve stejném významu.

$E \subseteq N \times \mathcal{A} \times N$
: množina hran zachycuje volání pravidel přes konkrétní instrukce, kde $\mathcal{A}$ je množina všech instrukcí volání v programu.

$C : N \to 2^O$
: kontext použití pravidla. Obsahuje objekty modelu dokumentu, pro jejichž zpracování bylo pravidlo vybráno při simulaci běhu programu.

$P : N \times \Sigma^{\mathsf{Q}} \to 2^O$
: přiřazení hodnot parametrům pravidla. Množina objektů modelu dokumentu.

$F : E \to O \times 2^O$
: přenos kontextu přes hranu.

Vrcholy grafu jsou pevně dány vstupním programem, simulace běhu konstruuje hrany spolu s informacemi o přenosu kontextů a parametrů. Vstupem algoritmu je ZXSLT program a model dokumentu. Konstrukce probíhá iterativně, posloupností střídajících se fází *konstrukce toku kontextů* a *propagace parametrů*.

Celý průběh lze shrnout do následujících bodů

- V iniciální fázi spočteme implicitní hodnoty parametrů pravidel. Dále vybereme pravidlo matchující objekt kořene a přiřadíme mu kořen jako nezpracovaný kontext.
- Ve fázi konstrukce iterativně zpracováváme pravidla obsahující nezpracované kontexty, které šíříme podle instrukcí volání do dalších pravidel.

- Pokud došlo ke změně kontextových množin pravidel, ve fázi propagace spočteme podle jejich obsahu hodnoty propagovaných parametrů a ty přiřadíme volaným pravidlům, jinak končíme.
- Pokud fáze propagace přinesla nové hodnoty parametrů pravidel, pokračujeme fází konstrukce, jinak končíme.

Návaznost fází je zajištěna aktualizací řídící struktury doplňující fáze. Fáze konstrukce využívá jako řídící strukturu příznaky nezpracovaných kontextů pravidel, fáze propagace používá příznaky pro zpracování pravidel. Formálně budeme tyto řídící struktury označovat jako

$$\mathit{pending} : N \to 2^O$$

$$\mathit{queue} \subseteq N$$

kde *pending* eviduje každému pravidlu jeho nezpracované kontexty a kde příznaky zpracování pravidel jsou reprezentovány jako fronta *queue*.

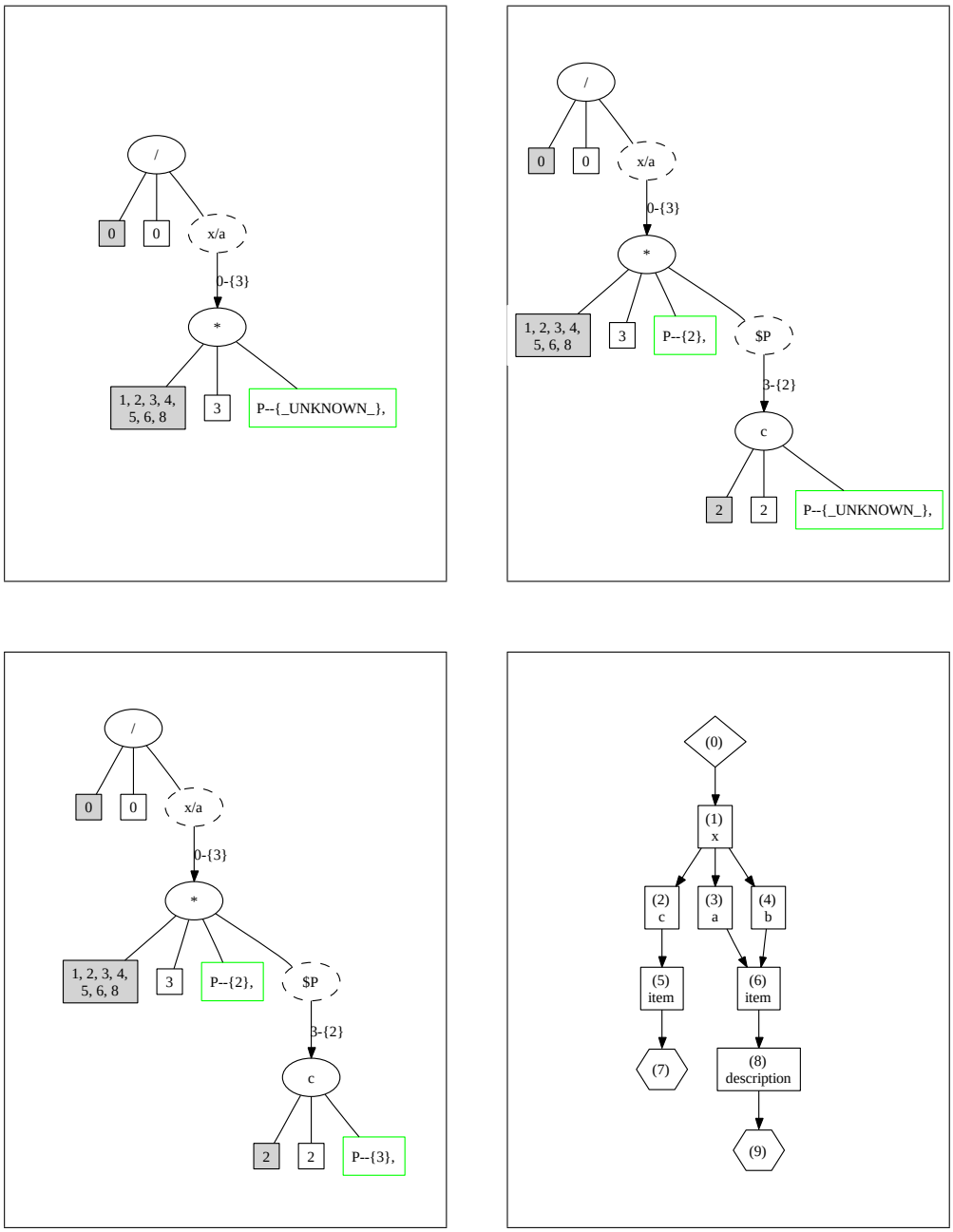
Pro ilustraci uveďme malý příklad. Analyzujeme běh programu A.2.1 nad vstupy podle schématu A.1.1. Ponecháme-li stranou samoúčelnost příkladu, vizualizace na obr. 4.1 zachycuje postupné šíření toku přes volání vyzházející z hodnoty předaného parametru.

4.1.2 Konstrukce toku kontextů

V průběhu této fáze jsou vytvářeny kontextové množiny pravidel a do grafu jsou přidávány hrany. Algoritmus lze shrnout následovně.

Dokud existuje pravidlo $n \in N$ s nezpracovaným kontextem $c \in \mathit{pending}(n)$

1. Vybereme n a odstraníme c z množiny $\mathit{pending}(n)$
2. Vyhodnotíme každou instrukci volání v pravidle vzhledem ke kontextu c. Bez újmy na obecnosti dále uvažujeme jednu instrukci volání a s výrazem x_s. Získáme množinu kontextového přenosu $S = \mathrm{Eval}(x_s, \{c\}, P(n))$.
3. Nechť pro každý přenesený kontext s je $M \subseteq N$ množina matchujících pravidel, tj. $s \in \mathrm{Eval}(x_m, O, \emptyset)$, kde x_m je vzor pravidla $m \in M$. Vybereme $M' \subseteq M$ pravidla, která nejsou *prioritně stíněna* žádným pravidlem z M.



Obrázek 4.1: Ukázka postupné konstrukce grafu toku řízení. *vlevo nahoře:* stav po iniciálním kroku konstrukce. *vpravo nahoře:* první propagace parametrů spolu s následujícím druhým krokem konstrukce. *vlevo dole:* druhá propagace parametrů a třetí (poslední) krok kontrukce. *vpravo dole:* graf modelu plain, na kterém byl program simulován.

4. Pokud již neexistovala, přidáme do E hranu $e = \langle n, a, m \rangle$ pro $m \in M'$. Pokud s nebyl kontextem pravidla m, vložíme jej do $C(m)$ a *pending*(m), pravidlo m označíme pro zpracování ve fázi propagace parametrů přidáním do *queue*. Přenos kontextu přes hranu zaznamenáme přidáním s do množiny $F(e, c)$.

Formální zápis algoritmu je uveden v příloze B.1.2.

Přesnost, s jakou volací vztahy zkonstruované uvedeným algoritmem, odpovídají reálnému běhu programu, je zcela závislá na použitém modelu. Vyplývá to z výpočtu přeneseného kontextu a volby matchujících pravidel, které jsou založeny pouze na vyhodnocování výrazů funkcí Eval (definice na obr. 3.5). Konkrétně je rozhodující, jak dobře objekt modelu reprezentuje uzel vstupního dokumentu. Ukážeme to na následujícím fragmentu programu

```
(1)  <template match="content/file">
(2)    <apply-templates select="name"/>
(3)  </template>
(4)  <template match="name">...</template>
(5)  <template match="dir/name">...</template>
```

Uvažujme simulaci běhu programu nad vstupem popsaným schématem A.1.2 v modelu plain, obr. 3.1, a algoritmus právě zpracovává pravidlo (1) pro kontext $\texttt{file}_8$. Vyhodnocení instrukce volání přenáší kontext na element $\texttt{name}_4$. Je vidět, že tento úsporný model smývá strukturální rozdíly mezi elementy stejného jména a typu. Objekt modelu $\texttt{name}_4$ matchují z uvedených pravidel pravidla (4) a (5), při skutečném běhu pravidlo (5) v této situaci nikdy použito být nemůže.

Tento typ ztráty přesnosti může přinést falešné volání pravidla, nestane se však, že by algoritmus nezpracoval volání, která může program při skutečném vykonávání provést.

4.1.3 Zohlednění priorit při výběru pravidel

Při skutečném běhu programu, v případě, že více pravidel matchuje přenesený kontext, volí XSLT procesor pravidlo s nejvyšší prioritou. Zavedení prioritní filtrace do algoritmu analýzy je nezbytné pro větší přiblížení skutečnému běhu – pokud bychom neřešili priority, v konečném důsledku by celým dokumentem prošlo vždypřítomné pravidlo se vzorem `*`.

Protože model je aproximací skutečného vstupního dokumentu, nemůžeme jen přímočaře zvolit matchující pravidlo p s nejvyšší prioritou – jak jsme ukázali na příkladu v oddílu 4.1.2 objekt zpracovávaného modelu může ve skutečnosti odpovídat více uzlům v některém ze vstupních dokumentů

a pravidlo p nemusí některý z uzlů vůbec matchovat. Uvažujme následující modifikaci zmiňovaného příkladu

```
(4)  <template match="name" priority="0">...</template>
(5)  <template match="dir/name" priority="1">...</template>
```

Kdybychom použili naivní prioritní výběr, bylo by ke zpracování objektu $\texttt{name}_4$ v kontextu $\texttt{file}_9$ použito pouze pravidlo (5). Tím bychom dostali chybný odhad běhu programu, což je nežádoucí pro následné zkoumání vytyčených problémů. Korektní zpracování je v tomto případě šíření toku řízení oběma pravidly.

Dodefinujme pojem *prioritní stínění*, který jsme použili v algoritmu konstrukce toku.

> *Pravidlo* p_1 prioritně stíní *pravidlo* p_2 *na kontextu* s*, ozn.* $p_1 \rhd_s p_2$*, jestliže* p_1 *má vyšší prioritu než* p_2 *a v libovolném vstupním dokumentu pro libovolný uzel* n*, odpovídající kontextu* s*, platí* p_2 *matchuje* $n \Rightarrow$ p_1 *matchuje* n*.*

Protože je prioritní stínění zadefinováno s ohledem na reálné vstupní dokumenty, můžeme jej použít při výběru z matchujících pravidel. Hledáme tedy algoritmus, který stínění dokáže rozhodnout.

Informace z modelu pro jejich možnou nepřesnost nemůžeme použít, o vstupním dokumentu v souvislosti se stíněním však dostatečné informace obsahují vzory pravidel. Stínění by bylo rozhodnuto, pokud bychom pro dva vzory dokázali říct, zda jeden z nich s ohledem na schéma vstupu obsahuje druhý. Tuto teoretickou úlohu zkoumali například Neven a Schwentick [11], či Wood [16]. Jejich závěry dávají kategorizaci této úlohy do tříd složitosti podle uvažovaných podmnožin XPath. Například pro podmnožinu obsahující *child* a *descendant* kroky s možností použití * je tento problém v třídě P. Je-li však tato podmnožina obohacena o predikáty nebo sjednocení, octneme se až ve třídě EXPTIME-úplných problémů. Pro určité podmnožiny se jedná dokonce o nerozhodnutelný problém.

Vzory pravidel obsahují pouze dopředné kroky a po zjednodušeních přijatých v oddílu 3.2.2 neobsahují proměnné ani sjednocení. Komplikací tedy zůstávají predikáty. Mějme pravidla p_1 a p_2, kde první z nich má vyšší prioritu. Mohou nastat dvě situace.

(a) Vyskytuje-li se predikát v nějakém kroku vzoru pravidla p_2, uvažujme pravidlo p_2' jako p_2 bez predikátů ve vzoru. Triviálně platí $p_1 \rhd_s p_2' \Rightarrow p_1 \rhd_s p_2$. Samozřejmě může dojít ke ztrátě přesnosti v případě, že pravidlo p_1 nestíní pravidlo p_2'.

(*b*) Vyskytuje-li se predikát ve vzoru pravidla p_1, je situace obtížnější. Obsah predikátu nepodléhá žádným striktním pravidlům, mohou se v něm vyskytovat libovolné výrazy. V důsledku přijatých zjednodušení schématu v oddílu 3.2.1 a závisle jazyka XPath v oddílu 3.3.2 nedokážeme predikát dostatečně přesně vyhodnotit. To by mohlo přinést falešné prioritní zastínění pravidla, proto v tomto případě ponecháme prioritní stínění nerozhodnuto. Budeme tedy uvažovat přenos kontextu do obou cílových pravidel, což vede k méně přesné, ne však chybné aproximaci toku řízení.

Algoritmus

Nyní, když jsme vzory pravidel dostatečně zjednodušili, použijeme jako Olesen [2] algoritmus rozhodující prioritní stínění, který je založen na množinových operacích nad konečnými automaty. Ukážeme, že zjednodušený vzor je regulární výraz popisující cestu ve stromě dokumentu, a tedy jej lze reprezentovat konečným automatem. Nad konečnými automaty pak máme k dispozici množinové operace, pomocí kterých můžeme otázku zastínění pravidla p_2 pravidlem p_1 zapsat jako výraz

$$\mathrm{FA}(p_2.\texttt{match}) \cap \mathrm{FA}(s, M) \subseteq \mathrm{FA}(p_1.\texttt{match}) \qquad \text{(OA)}$$

kde funkční zápis FA() zastupuje automat pro vzor pravidla a automat pro šířený kontext.

Jelikož pravidlo p_2 může být určeno i pro zpracování jiných objektů vstupního dokumentu, než reprezentuje šířený kontext s, potřebujeme kvůli ověření inkluze provést zúžení pouze na kontext s – pokud bychom neuvažovali kontext, nikdy bychom např. nezastínili pravidlo *. Pro zúžení zkonstruujeme automat $\mathrm{FA}(s, M)$ odrážející strukturu modelu s koncovým stavem v šířeném kontextu.

Přistupme k formálnímu vyjádření automatů. Uvažujme pravidla p_1, p_2 a kontextový objekt s modelu $M = (O, \mathit{objtype}, \mathit{objname}, \mathit{axis}, \varrho)$.

Objekty vstupního dokumentu (a tedy i modelu) jsou ve vzorech pravidel charakterizovány typem a jménem, abecedu konečných automatů budou tvořit právě dvojice typ–jméno objektů modelu.

$$\Sigma^{\mathsf{FA}} \subseteq \{\mathit{element}, \mathit{attribute}, \mathit{text}, \mathit{root}\} \times \Sigma^{\mathsf{Q}}$$

$$\Sigma^{\mathsf{FA}} = \{\langle \mathit{objtype}(o), \mathit{objname}(o)\rangle \mid o \in O\}$$

Automat $\mathrm{FA}(s, M) = (\Sigma, Q, q_0, F, \delta)$ vyjádříme pomocí struktur modelu

$\Sigma = \Sigma^{\mathsf{FA}}$

$$\mathrm{R}(/) = \mathrm{R}(\lambda) = \langle \mathit{root}, \lambda \rangle^1$$

$$\mathrm{R}(P/\chi :: \tau) = \mathrm{R}(P) \cdot \mathrm{R}(\chi :: \tau)$$

$$\mathrm{R}(\text{child} :: \tau) = \begin{cases} (\langle \mathit{element}, *:* \rangle \mid \langle \mathit{text}, \lambda \rangle), & \tau = \text{node}() \\ \langle \mathit{text}, \lambda \rangle, & \tau = \text{text}() \\ \langle \mathit{element}, \tau \rangle, & \text{jinak} \end{cases}$$

$$\mathrm{R}(\text{attribute} :: \tau) = \begin{cases} \langle \mathit{attribute}, *:* \rangle, & \tau = \text{node}() \\ \langle \mathit{element}, \tau \rangle, & \text{jinak} \end{cases}$$

$$\mathrm{R}(\text{descendant-or-self} :: \text{node}()) = \langle \mathit{element}, *:* \rangle^*$$

Obrázek 4.2: Formalizace převodu vzoru pravidel na regulární výraz. Symboly $\cdot$, resp. $\mid$, resp. * zastupují operátor zřetězení, resp. disjunkce, resp. iterace.

$$Q = O \ \cup \{start\}$$

$$q_0 = start$$

$$F = \{s\ \}$$

$$\delta = \{\langle start, \langle root, \lambda \rangle, \varrho \rangle\} \cup \{\langle p, \alpha, q \rangle \mid \langle p, q \rangle \in \mathit{axis}(\chi) \wedge \\ \alpha \in \langle \mathit{objtype}(q), \mathit{objname}(q) \rangle \wedge \chi \in \{\text{child}, \text{attribute}\}\}$$

Poznamenejme, že obecně se jedná o nedeterministický automat. Nedeterminismus je závislý na reprezentaci vnitřní struktury typů schématu v modelu, např: více synů-elementů stejného jména.

Abeceda regulárních výrazů Σ^{RE} musí postihovat i výskyt symbolu * ve vzoru, proto ji definujme jako

$$\Sigma^{\mathsf{RE}} = \{\mathit{element}, \mathit{attribute}, \mathit{text}, \mathit{root}\} \times \overline{\Sigma^{\mathsf{Q}}}$$

Výraz vzoru převedeme na regulární výraz pomocí funkce R(), jejíž definice je uvedena na obr. 4.2.

Znaky abecedy, které obsahují ve jmenné složce zástupné symboly z $\overline{\Sigma^{\mathsf{Q}}} \setminus \Sigma^{\mathsf{Q}}$ jsou zkratky za disjunkci odpovídajících znaků abecedy konečných automatů.

$$\langle t, x \rangle = (\langle t, n_1 \rangle \mid \langle t, n_2 \rangle \mid \ \ldots \ \mid \langle t, n_k \rangle)$$

kde $\langle t, x \rangle \in \Sigma^{\mathsf{RE}} \setminus \Sigma^{\mathsf{FA}}$, $\forall i = 1 \ldots k : \langle t, n_i \rangle \in \Sigma^{\mathsf{FA}} \wedge n_i \approx x$.

Převod regulárního výrazu na konečný automat je pak proveden standardně, výsledkem je nedeterministický konečný automat. Výchozí vztah (OA) upravme jednoduchými množinovými operacemi do tvaru

$$(\mathrm{FA}(p_2.\texttt{match}) \cap -\mathrm{FA}(p_1.\texttt{match})) \cap \mathrm{FA}(s, M) = \emptyset \qquad (\mathrm{OA}')$$

Pro rozhodnutí neprázdnosti průniku se nemusíme explicitně zbavovat nedeterminismu automatů. Pouze v případě konstrukce doplňkového automatu dojde k odstranění nedeterminismu. Cesty ve vzorech pravidel jsou typicky složeny pouze z malého počtu kroků, čemu odpovídá počet stavů. Případný exponenciální nárust vyplývající z převodu na determinizmus v tomto případě tedy nebude kritický.

4.1.4 Propagace parametrů

Mějme částečně zkonstruovaný graf řízení a k pravidlům množinu příznaků zpracování. Každému pravidlu ke zpracování přepočteme implicitní hodnoty parametů podle aktuálního kontextu.

Dokud existuje pravidlo $n \in \mathit{queue}$

1. Vybereme pravidlo n z množiny queue.
2. Pro každou hranu $\langle n, a, m\rangle \in E$ a parametr p, který je předáván voláním a, spočteme předávanou hodnotu $X = \mathrm{Eval}(x_p, C(n), P(n))$, kde x_p je výraz parametru p.
3. Spočtenou hodnotu X sjednotíme s hodnotou parametru v cílovém pravidle, $P(m, \mathit{name}_p)$, kde name_p je jméno parametru p. Pokud došlo k aktualizaci hodnoty, přidáme m do queue a $C(m)$ do $\mathit{pending}(m)$.

Formální zápis je uveden v příloze B.1.3.

Při konstrukci toku je průběh dán postupným odebíráním nezpracovaných kontextů pravidel z tzv. *pending–množiny*. Pro vybraný kontext jsou pak vyhodnocována volání v pravidle.

V případě propagace parametrů je situace poněkud jiná a řešení pending–množinou nemůžeme použít. V XSLT jsou uvažované hodnoty parametrů do pravidel předávány jako celky, tedy jako množiny objektů. Při instanciaci pravidla pak také figurují v předané podobě.

Uvažujme opět poněkud umělý případ.[2]

```
<template ...>
  <param name="A"/>
  <apply-templates ...>
```

[2] řekněme, že programátor chtěl ušetřit řádky za instrukci větvení, v parametru předává k dalšímu zpracování celý obsah parametru A, pokud některý z jeho objektů obsahuje synovský element `z`, jinak prázdnou množinu.

```
    <with-param name="B" select="$A[$A/z]"/>
  ...
```

Protože konstruujeme uzávěry hodnot parametrů postupným přidáváním, nemůžeme zaručit, že při přenosu parametru ve volání bude jeho hodnota obsahovat některou ze skutečně předávaných množin objektů. Na příkladu je vidět, že pokud bychom nově příchozí objekty v hodnotě parametru zpracovávali po jednom, nezpracovali bychom korektně přenos do parametru B cílového pravidla.

S naším přístupem může být naopak přes parametr přeneseno více objektů než při reálném běhu, máme však zaručeno, že se nic neztratí.

4.2 Konečnost výpočtu a odhady složitosti

Konečnost výpočtu obou dílčích kroků je podmíněna konečností modelu, nad kterým běh zkoumaného programu simulujeme.

Vyhodnocení XPath výrazu, což je základ simulace, je na takovém modelu zřejmě konečná operace.

V případě konstrukce toku je v každé iteraci zpracován jeden objekt modelu jako kontext nějakého pravidla. Jednou zpracovaný objekt již s daným pravidlem zpracováván nebude. V těle iterace pak vyhodnocujeme konečný počet instrukcí volání.

Při propagaci parametru zpracováváme až do vyprázdnění frontu pravidel. Každá iterace odstraní z fronty jedno pravidlo. Zpět do fronty se pravidlo dostane tehdy, když je hodnota některého jeho parametru zvětšena o aspoň jeden objekt modelu. Uvažovaná hodnota každého z parametrů pravidla může nabývat maximálně všech objektů modelu, což je z předpokladu konečná množina.

Ukažme, že ani posloupnost kroků konstrukce a propagace není nekonečná. Konstrukce toku končí, když byly vyčerpány všechny nezpracované kontexty. Krok propagace parametrů dostává ke zpracování neprázdnou frontu pravidel tehdy, pokud došlo k rozšíření toku kontextu přes nějakou hranu. Počet takových rozšíření toku na hraně je shora omezen počtem objektů modelu. Neproběhne-li propagace, konstrukce toku nemá co zpřesňovat, běh potom končí. Žádný z kroků neodebírá informace z grafu, nemůže se tedy stát, že provedeme vícekrát stejné rozšíření. Neformálně také shrneme časovou a prostorovou složitost algoritmů.

Složitost

Nechť $|N|$ je počet pravidel programu, $|O|$ počet objektů modelu. Dále k bude označovat maximální počet instrukcí volání v pravidle, p maximální počet parametrů pravidla a q bude maximální míra složitosti XPath výrazu v programu[3].

U časové složitosti fází využijeme provedených úvah o konečnosti, které nám dávají k dispozici horní odhady počtu iterací hlavních cyklů. V obou algoritmech je stěžejní operací vyhodnocení výrazu, časovou složitost vyjádříme relativně k časové složitosti funkce Eval, ozn. T_{Eval}.

Konstrukce toku je založena na vyhodnocování instrukcí volání. Z úvahy o konečnosti je počet zpracovaných volání $\leq k\ |O||N|$.

- Zpracování volání vyžaduje vyhodnocení cesty, $T_{Eval}(|O|, q)$.
- Množinové operace pro nalezení matchujících pravidel a pozdější šíření kontextu do cílových pravidel jsou $\mathcal{O}(|N||O|)$.
- Výběr z nejvýše $|N|$ kandidátů cílových pravidel $\leq \binom{n}{2} T_{Override}(|O|)$.

Časová složitost rozhodnutí prioritního stínění, $T_{Override}(|O|)$, je složitost průniku automatů, kde $|O|$ figuruje jako horní uzávěr velikosti abecedy a počtu stavů automatů, což je třídy $\mathcal{O}(|O|^m)$, kde m je konstanta. Celkem je tedy časová složitost konstrukce toku prvkem

$$\mathcal{O}(k|O||N| \cdot (T_{Eval}(|O|, q) + |N|^2|O|^2))$$

Analogicky uvažujeme propagaci parametrů. V nejhorším uvažovaném případě budou hodnoty parametrů do pravidel šířeny po jednom objektu, tedy k aktualizaci hodnot přenášených z pravidla by došlo nejvýše $|N||O|^p$ krát.

- Aktualizován je přenos do nejvýše $k|N|$ pravidel.
- Přepočtení a šíření hodnot parametrů do jednoho cílového pravidla vyžaduje čas $\leq p \cdot (T_{Eval}(|O|, q) + |O|)$.

Což je ve třídě

$$\mathcal{O}(kp|N|^2|O|^p \cdot (T_{Eval}(|O|, q) + |O|))$$

[3]tento pomocný pojem zachycuje délky dílčích cest, hloubku zanoření predikátů, ...

Uvedené složitosti jsme vyjádřili relativně k T_{Eval} záměrně. Naše definice vyhodnocování, obr. 3.5, která je přímočarým přizpůsobením sémantiky XPath pro naše modelované dokumenty, má exponenciální časovou složitost. Gottlob s kolegy [18] ukázal alternativní metodu vyhodnocování XPath výrazů, která má polynomiální časovou složitost. Ačkoli se jedná o vyhodnocování výrazů nad skutečným dokumentem, domníváme se, že princip lze aplikovat i na naši situaci modelovaného dokumentu.

Prostorová složitost obou fází odpovídá velikosti konstruovaného grafu toku řízení.

- Vrcholy odpovídají pravidlům, $|N|$.
- Počet hran grafu $\leq k|N|^2$.
- Pro každé pravidlo evidujeme objekty kontextových množin, množin hodnot parametrů a objekty pomocné match-množiny. Počet objektů uchovávaných pro pravidlo $< |O| \cdot (p + 2)$.
- Kontextový tok přes hranu je zachycen jako zobrazení vstupního objektu na množinu výstupů, tedy $\leq |O|^2$ záznamů.

V porovnání s počtem pravidel a počtem objektů modelu můžeme počet parametrů pravidla uvažovat jako konstantu, vynecháním aditivního členu pak dostáváme odhad, že velikost uchovávaných informací je

$$\mathcal{O}(k|N|^2|O|^2)$$

Uvedené odhady jsou značně hrubé, ukazují však, že se jedná o v principu polynomiální aloritmy.

4.3 Graf řízení a zkoumané problémy

Nyní ukážeme, jak zkonstruovaný graf řízení využít k odhalování zkoumaných problémů.

4.3.1 Slepé cesty

Rozhodnutí problému slepých cest provedeme přímo z definice. Ke každému pravidlu dává graf řízení seznam kontextů a uzávěry hodnot parametrů. Zkoumaný výraz cesty tedy stačí vyhodnotit pro každý z kontextů pravidla, ve kterém se nachází.

Ohlášení slepé cesty označuje *potenciální chybu* v programu. Nejistotu přináší vyhodnocení vzoru pravidla. Protože vyhodnocení výrazu není přesné, zejména v případě predikátů, které se často ve vzorech pravidel používají, pravidlu mohl být přiřazen jako kontext objekt modelu, pro jemuž odpovídající skutečné objekty pravidlo nikdy být použito nemusí. Vyhodnocení výrazu cesty vzhledem k objektu, ke kterému nikdy vyhodnocováno být nemělo pak snadno vede na prázdnou množinu.

Jako speciální případ slepých cest uveďme *slepá volání*. Jedná se o případ, kdy výraz výběru je v instrukci volání vyhodnocován na prázdnou množinu objektů pro každý kontext pravidla. Instrukce volání a je slepá, jestliže v grafu řízení neexistuje hrana odpovídající tomuto volání, nebo-li

$$\neg\exists e \in E : e = \langle n, a, m\rangle$$

Na rozdíl od slepé cesty je výskyt slepého volání *skutečnou chybou programu*, neboť výraz volání je slepou cestou ve všech kontextech pravidla.

4.3.2 Nedosažitelná pravidla

Nedosažitelná pravidla získáme z grafu toku řízení zcela přímočaře. Jedná se o pravidla těch uzlů, která mají prázdnou kontextovou množinu. Výjimkami jsou implicitní pravidla programu, která proto i v případě že mají prázdnou kontextovou množinu, nepovažujeme za nedosažitelná.

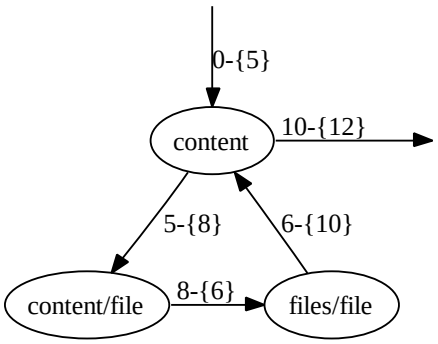
Jako speciální případ jsou mezi nedosažitelnými pravidly automaticky zahrnuta i pravidla, která svým vzorem nematchují žádný z objektů modelu.

Protože je model dokumentu uzávěr instancí vstupního schématu a kvůli omezením ve vyhodnocování výrazů, matchují vzory pravidel určitě aspoň objekty odpovídající reálným objektům, ohlášení nedosažitelného pravidla udává *skutečnou chybu* v původním XSLT programu.

4.3.3 Cykly a nekonečný běh programu

Cyklus v grafu toku řízení vzniká, pokud v simulovaném běhu při zpracování instrukcí nějakého pravidla, ozn. A, bude pro zpracování kontextu přímo, či skrz nějaké z podstromu volaných pravidel, vybráno opět pravidlo A. Samotný cyklus tvořený hranami grafu však zdaleka neříká nic o možné nekonečnosti běhu programu – vzor pravidla může matchovat více objektů modelu, pravidlo mohlo být použito vždy pro jiný objekt. Tuto situaci zachycuje obrázek 4.3 k příkladu A.2.2.

Uvažujeme tedy *kontextový graf* $G^{\mathsf{C}} = (N^{\mathsf{C}}, E^{\mathsf{C}})$ vyjádřený z grafu toku řízení jako



Obrázek 4.3: Cyklus v grafu toku řízení bez ohledu na kontexty. Pravidlo se vzorem `content` figuruje v řetězci volání pokaždé s jiným kontextovým objektem. Pro jednoduchost byly v obrázku vynechány symboly instrukcí volání, kontextových a match množin.

$$N^{\mathsf{C}} \subseteq N \times O,\ E^{\mathsf{C}} \subseteq N^{\mathsf{C}} \times \mathcal{A} \times N^{\mathsf{C}}$$

$$N^{\mathsf{C}} = \{\langle n, c\rangle \mid n \in N \wedge c \in C(n)\}$$

$$E^{\mathsf{C}} = \{\langle\langle n_1, c_1\rangle, a, \langle n_2, c_2\rangle\rangle \mid e = \langle n_1, a, n_2\rangle \in E \wedge \langle c_1, c_2\rangle \in F(e)\}$$

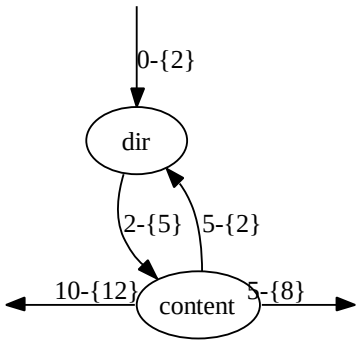
Z definice je patrné, že cyklus v kontextovém grafu již zohledňuje přenos kontextu mezi pravidly. Vztah cyklu k nekonečnému běhu je zatížen jak přesností modelu, tak principem naší kvazi-simulace běhu programu. Nepřesnost je způsobena zejména problematickou reprezentací rekurzivních elementů v modelu a oddělením propagace parametrů do samostatné fáze mimo šíření kontextů. Ukážeme, že i cyklus v kontextovém grafu může být falešný, tedy podobně jako v případě cyklu grafu řízení nedochází k opakovanému přenosu kontextu. Nalezený cyklus proto odráží nejvýše *možnost* nekonečného běhu na některém ze vstupů.

Pro zvýšení přesnosti rozdělíme analýzu nekonečného běhu do fází

(i) hledání cyklů v kontextovém grafu

(ii) postprocessing nalezených cyklů

Nespokojíme se tedy pouze s detekovanými cykly, dodatečně budeme v cyklech hledat situace, za kterých můžeme rozhodnout, že cyklus nesouvisí s nekonečností běhu. První z fází je čistě technická, nebudeme se jí proto v textu věnovat, druhou rozebereme v následujícím oddílu.



Obrázek 4.4: Dopředný cyklus v grafu toku řízení. Pro jednoduchost byly v obrázku vynechány symboly instrukcí volání, kontextových a match množin.

4.4 Postprocessing kontextových cyklů

V následujících úvahách uvažujme libovolné modulo-očíslování vrcholů cyklu podle uspořádání definované hranami.

4.4.1 Eliminace dopředných cyklů

> *Dopředným nazýváme cyklus, jehož proběhnutí nad reálným dokumentem odpovídá přenosu kontextu hlouběji ve stromě dokumentu.*

Proběhnutí cyklu v kontextovém grafu vede k výchozímu kontextovému objektu. Pokud je cyklus dopředný, museli jsme přejít přes rekurzivní element. Dopředný cyklus v reálném prostředí tedy není cyklem, z konečnosti vstupního dokumentu plyne, že nemůže způsobit nekonečný běh programu.

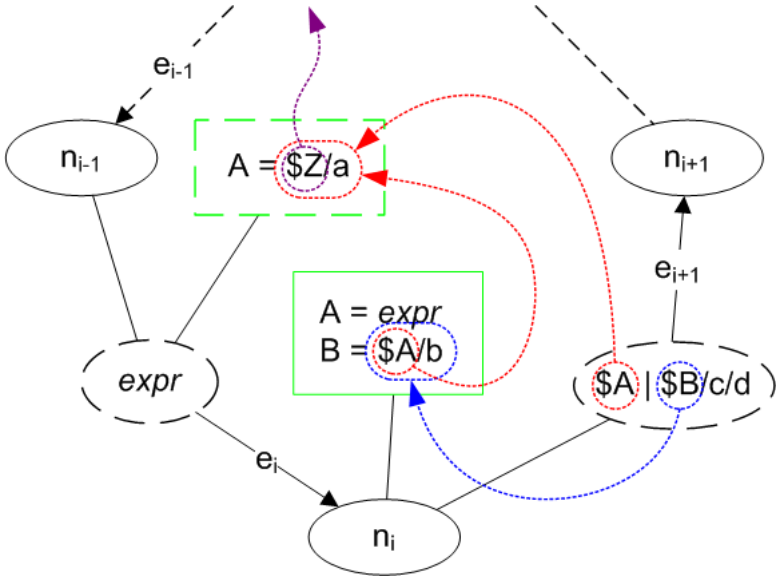
Z nalezených cyklů tedy chceme dopředné cykly odfiltrovat. Následující podmínka charakterizuje podmnožinu dopředných cyklů.

(i) Jsou-li v cyklu C všechny instrukce volání tvořeny relativními cestami složenými pouze z dopředných a nehybných os a navíc aspoň jedna z uvažovaných os je dopředná, potom je cyklus C dopředný.

Ověření uvedené podmínky je přímočaré, tento triviální případ dopředného cyklu je zachycen na obrázku 4.4, který se vztahuje také k příkladu A.2.2.

Uvažujme obecnější případ, kdy některý z výrazů volání obsahuje parametr jako výchozí krok cesty, např.

```
<apply-templates select="$A|$B/c/d"/>
```



Obrázek 4.5: Závislosti parametrů. Tečkované šipky ukazují závislost hodnot parametrů na výrazech. Do pravidla n_i je přenášena hodnota parametru A, výraz přenosu má přednost před definičním výrazem.

Původ hodnot parametrů může sahat několik kroků zpět proti směru cyklu, voláním přes parametr tak můžeme přejít do již opuštěného kontextu. Definičním výrazem parametru rozumíme `param.select` v pravidle, výrazem přenosu parametru pak `with-param.select` v instukci volání. Původ parametru je tvořen definičním výrazem a výrazy přenosu až k místu použití. Díky operátoru sjednocení může mít hodnota parametru více zdrojů, definice parametru může také ležet mimo cyklus a v cyklu je jen dokola předávána hodnota.

Mějme v cyklu pravidla n_i a n_{i+1}, kde volání od i k $i+1$ přenáší parametr A. Výraz e přenosu parametru z pravidla n_i zastiňuje definiční výraz A v pravidle n_{i+1}. Jinými slovy všechny výrazy $\$A$ v pravidle n_{i+1} jsou závislé na výrazu e. Nechť pro naše účely zdrojový výraz hodnoty parametru v cyklu určuje funkce

$$\mathrm{Src} : \Sigma^{\mathsf{Q}} \times N \to \Phi \times N$$

Příklad závislostí parametrů v cyklu ukazuje obr. 4.5.

Na základě směru os, vyjádříme směr výrazu jako prvek množiny $\Delta = \{\bullet, \to, ?\}$, kde uvedené symboly zastupují nehybný, dopředný a nerozhodnutelný směr. Směr výrazu potom definujeme rekurzivní funkcí

$$\mathrm{Dir} : \Phi \times N \times \mathcal{H} \to \Delta$$

kde $H \in \mathcal{H}$ je pracovní argument pro zastavení rekurze, množina zpracovaných dvojic parametr – pravidlo.

$$\mathrm{Dir}_{n,H}(P/step) = \mathrm{Dir}_{n,H}(P)/\mathrm{Dir}_{n,H}(step)$$

$$\mathrm{Dir}_{n,H}(step) = \begin{cases} \rightarrow & \text{pro osy child, attribute, descendant, descendant-or-self} \\ \bullet & \text{pro osu self} \\ ? & \text{jinak} \end{cases}$$

$$\mathrm{Dir}_{n,H}(P_1 \mid P_2) = \mathrm{Dir}_{n,H}(P_1) \mid \mathrm{Dir}_{n,H}(P_2)$$

$$\mathrm{Dir}_{n,H}(\$X) = \begin{cases} \bullet & \text{pokud } \langle X, n\rangle \in H \\ \mathrm{Dir}_{n',H'}(expr) & \text{jinak, kde} \\ & \langle expr, n'\rangle \in \mathrm{Src}(X,n), H' = H \cup \{\langle X, n\rangle\} \end{cases}$$

$$\mathrm{Dir}_{n,H}(\lambda) = ?$$

kde operátory / a | nad množinou Δ jsou definovány jako

/	?	•	→
?	?	?	?
•	?	•	→
→	?	→	→

|	?	•	→
?	?	?	?
•	?	•	•
→	?	•	→

Definice směru vychází z rozkladu cesty na kroky, kterým jsme přiřadili směr podle os. Skládání je zadefinováno tak, aby pokrývalo podmínku dopřednosti cyklu (*i*). Nerozhodnutelný krok vede vždy na nerozhodnutelnost cesty. Směr hodnoty parametru vždy vychází ze zdrojového výrazu.

V případě cyklické závislosti parametru uvažujeme při druhém zpracování parametru nehybnost. Pokud zbylé výrazy původu parametru dokola cyklu obsahují ve svých krocích jen nehybné osy a hodnoty parametrů, zakončení nehybností nic nezmění. Pokud se v aspoň jednom z uvažovaných výrazů vyskytuje dopředná osa, neutrální nehybnost dopřednost nepokazí.

Nyní zformulujeme druhou podmínku charakterizující podmnožinu dopředných cyklů.

(*ii*) Nechť pro cyklus C platí předpoklady z (*i*) a nechť pro libovolný parametr P, který figuruje v instrukci volání a_i pravidla n_i jako výchozí krok platí

(*a*) $\mathrm{Dir}(P, n_i, \emptyset) = \rightarrow$, nebo

(*b*) P má acyklický původ, $\mathrm{Dir}(P, n_i, \emptyset) = \bullet$ a $c_{i+1} \neq c_{j_k}$, pro $\langle n_{i+1}, c_{i+1}\rangle, \langle n_{j_k}, c_{j_k}\rangle \in N^{\mathsf{C}}$, kde n_{j_k} jsou pravidla, ve kterých má P definiční výraz.

Potom je cyklus C dopředný.

V případě (*a*) přechod přes parametr P vede do kontextu, který je hlouběji ve stromě než kontext, ve kterém byl počátek P definován. V případě (*b*) je kontextový objekt c_o, ve kterém byl definován počátek P jiný než objekt c_{i+1}, který vybírá volání a_i. Z předpokladu (*i*) pak c_{i+1} musí být hlouběji než c_o. Uvedená podmínka je tedy korektní.

Obecně nic nebrání tomu, aby dopředný cyklus obsahoval cesty i se zpětnými osami. Jak ukážeme v následující kapitole o uvažovaných modelech, zpětný směr ke kořeni dokumentu je v modelu zachycen s menší přesností, tento případ neřešíme.

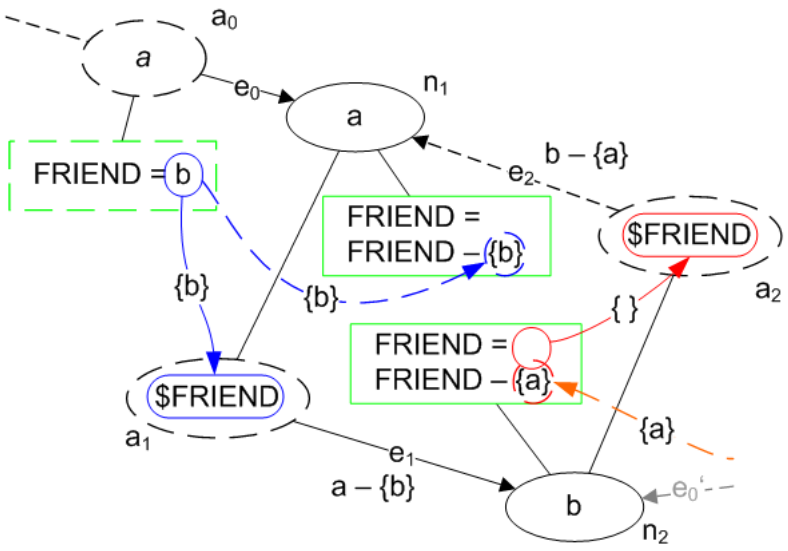
4.4.2 Eliminace zavlečených cyklů

Metoda iterativní konstrukce toku, ve které jsme oddělili fáze šíření kontextu a přenosu parametrů, má neblahý vliv na vznik falešných cyklů. Abychom se vyhnuli možnému nárustu grafu řízení, nejsou v něm zachyceny přenosy parametrů s vazbou na kontexty, se kterými k přenosu došlo. Pravidlo programu figuruje ve výsledném grafu řízení nejvýše jednou, se všemi možnými kontexty a hodnotami parametrů, ne pro každou instanciaci zvlášť. V kontextovém grafu nejsou parametry také nijak zohledněny.

Uvažujme program A.2.3 a model schématu A.1.1 jako jeho vstup. Obrázek 4.6 ukazuje cyklus, který vznikne v grafu toku řízení při simulaci. Pravidlo n_1, resp. n_2, získalo v kroku propagace parametrů element `b`, resp `a`, jako hodnotu parametru FRIEND. Následný krok konstrukce toku přes instrukce a_1, resp. a_2, podle hodnoty parametru FRIEND vytvořil hranu volání e_1 s kontextovým přenosem $\mathtt{a} \to \{\mathtt{b}\}$, resp. e_2 s $\mathtt{b} \to \{\mathtt{a}\}$, což dohromady tvoří kontextový cyklus. Volání a_1 ale nepředává hodnotu parametru FRIEND do n_1. Uvažujeme-li možný vstup do cyklu hranou e_0, při vyhodnocování volání instrukce a_2 bude použita implicitní hodnota FRIEND, v tomto případě tedy prázdná množina objektů, k přechodu přes hranu e_2 a uzavření cyklu nedojde.

Cyklus C označíme jako zavlečený, jestliže pro libovolné k_0 existuje $l < 2|C|$, že $c_{i+1} \notin Eval(a_i.\mathtt{select}, c_i, M_i)$, kde

- $i = k_0 + l$,



Obrázek 4.6: Zavlečený cyklus

- $M_{k_0} = P(n_{k_0})$ *jsou výchozí hodnoty parametrů v pravidle* n_{k_0} *z grafu toku řízení,*
- $M_{k_{j+1}}$ *jsou přenesené hodnoty parametrů přes volání* a_j *spolu s vyhodnocenými default hodnotami v pravidle* n_{j+1}.

Tato definice říká, že zavlečený cyklus nebude proběhnut dvakrát při vyhodnocování volání s ohledem na přenášené parametry.

Platí také, že výraz diskutované instrukce volání a_i je závislý na nějakém předávaném parametru. Pokud by nebyla, byl by to ihned spor s definicí hrany kontextového cyklu, která vzniká právě přenosem kontextu a jestliže výraz volání nevyužívá žádný parametr, musí cesta projít od zdrojového k cílovému kontextu.

Detekce zavlečeného cyklu spočívá v ověření přenosu kontextů přes dvojitý průběh cyklu z libovolného vrcholu.

4.5 Uvažovaná řešení

Uvažovali jsme také alternativní postup konstrukce grafu řízení. Princip spočíval v opačném směru konstrukce, tedy od maximálního toku kontextů, postupným prořezáváním k pevnému bodu. Iniciální konstrukce toku kontextů byla provedena s horním odhadem hodnoty každého parametru celou množinou objektů dokumentu. Následně se pak střídaly kroky propagace hodnot

```
(1)   <xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
(2)     <xsl:template match="/">
(3)       <xsl:call-template name="process">
(4)         <xsl:with-param name="NODES" select="//file[@id]"/>
(5)       </xsl:call-template>
(6)     </xsl:template>
(7)     <xsl:template name="process">
(8)       <xsl:param name="NODES"/>
(9)       <xsl:apply-templates select="$NODES"/>
(10)    </xsl:template>
(11)    <xsl:template match="dir">
(12)      <xsl:apply-templates />
(13)    </xsl:template>
(14)  </xsl:stylesheet>
```

Obrázek 4.7: XSLT program pro rozbor uvažovaného řešení „propagace parametrů – prořezání toku“.

parametru a prořezání toku. Ve fázi prořezání toku byl přepočítán kontextový přenos přes volání závislá na parametrech s aktualizací kontextových množin pravidel. Předpokládaný přínos této metody byla rychlejší konvergence.

Tato idea se však ukázala jako ne úplně vhodná. Vyskytuje-li se v programu pravidlo s voláním, které vybírá cílové uzly hodnotou parametru, budou v iniciálním toku zahrnuty všechny objekty modelu a s nimi odpovídající pravidla.

Uvažujme program na obr. 4.7 při běhu nad modelem z obr. 3.1. Hodnota parametru NODES v pravidle (7) bude odhadnuta celou množinou O modelu. Po iniciálním kroce bude existovat přenos z pravidla (7) do (11). Díky rekurzivnímu elementu $\texttt{dir}_2$ vzniká cyklus mezi (11) a implicitním pravidlem *. Po propagaci hodnot parametrů a prořezání toku, kdy bude přenos (7)–(11) odstraněn, zůstane tento cyklus bez vstupního bodu viset v grafu.

Protože se jedná o dopředný cyklus, dodatečné zpracování cyklů by v tomto případě cyklus odstranilo z kandidátů nekonečného běhu, nicméně pravidlo (11), které by mělo být označeno za nedosažitelné, obsahuje tok. Možné řešení by mohlo být zavedení detekce komponent souvislosti jako součást fáze prořezávání toku, podrobně jsme se tím však nezabývali.

Kapitola 5

Výsledky

Tato kapitola je věnována výsledkům analýzy. Implementovaný algoritmus, viz aplikace WXSA v dodatku D, byl kromě interních příkladů testován také na vzorku reálných dat.

5.1 Úspěšnost analýzy

V průběhu vývoje demonstrační aplikace vznikla řada mini-příkladů za účelem ladění a ověření funkčnosti. Ačkoli testy na datech tohoto charakteru nemají velkou vypovídací hodnotu jsme na ně bohužel také odkázáni.

Dostupnost testovacích dat se ukázala jako závažný problém. Pro testování implementovaného algoritmu analýzy je nezbytná dvojice program – schéma. Ačkoli jsou XML technologie velmi rozšířené, na internetu lze většinou dohledat pouze polovinu zmíněné dvojice. Schémata jsou většinou použita pro popis veřejných datových rozhraní, zpracování dat se většinou už nezmiňuje. Podobně XSLT programy jsou k nalezení samostatně, případně se schématem v DTD. Ani benchmarkové balíky XSLT procesorů [33, 36] v tomto směru nejsou přínosem, vedle programů obsahují pouze referenční XML data, ne schémata.

5.1.1 Reálná testovací data

Celkem jsme měli k dispozici jen 11 testovacích příkladů, viz dodatek C. Jejich analýza odhalila po různu všechny druhy chyb. V sadě příkladů kml2xxx ve výsledcích figurovala i nematchující pravidla – to mohlo být způsobeno odlišností verze programu a schematu, neboť v tomto případě jsme dvojici dat získali odděleně.

Sesbírání samostatných souborů schémat bylo o poznání snazší. Z nalezených 68 schémat jsme konstruovali modely k získání statistikých údajů o jejich velikosti, viz tabulka 5.1.

5.1.2 Procedurální cykly

Ze vzorku reálných dat se ukázalo, že programátoři používají procedurální přístup k průchodu stromu tvořeném rekurzivními elementy, pojmenovaná pravidla s parametry. To ilustruje např. kostra programu na obr. 5.1 pro zpracování dokumentů schématu A.1.2. Simulací běhu dostáváme v grafu řízení cyklus mezi pravidlem (7) a osamostatněným pravidlem (9). Z definice osamostatnění v oddílu 3.2.2, pravidlo (9) matchuje všechny objekty, hodnota parametru DIR je použita jako filtr ve vzniklém volání

```
<apply-templates select="self::node()[$DIR]" mode="m"/>
```

Parametr DIR nabývá hodnoty množiny objektů dokumentu. Mezi iteracemi cyklu je přenášená hodnota získána sestupem od objektů aktuální hodnoty hlouběji do stromu dokumentu. To v reálném dokumentu nelze provádět donekonečna, logicky se tedy jedná o dopředný cyklus. Tuto variantu však naše detekce z 4.4.1 nepostihuje. Uvedený příklad je značně specifický tím, že v podmínce figuruje pouze hodnota parametru jako dotaz na neprázdnost množiny objektů, které obsahuje. Obecně je podmínka predikátu složitější výraz nad parametrem, z omezení, která jsme přijali v definici vyhodnocování, nedokážeme rozhodnout, zda pravdivost závisí na neprázdnosti.

Případ cyklů tohoto typu tedy považujeme za případ potenciálně nekonečného běhu programu.

5.2 Srovnání modelů

Reprezentace vstupních dat je rozhodující pro přesnost simulace. Ukážeme rozdíly v přesnosti zpracování vybraných rysů programů při simulaci.

5.2.1 Model plain vs. model plain$^{\mathsf{m}}$

V modelu plain$^{\mathsf{m}}$ jsou označeny objekty, které reprezentují objekty dokumentu, jejichž výskyt je povinný. Tato drobnost se projeví v případě, kdy ve výrazu figuruje predikát s negativní adresací uzlu – funkcí not(). Tedy např. ve výrazu

```
a/b[not(@id)]
```

```
(1)   <xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
(2)     <xsl:template match="/">
(3)       <xsl:call-template name="PrintDir">
(4)         <xsl:with-param name="DIR" select="file-system/dir"/>
(5)       </xsl:call-template>
(6)     </xsl:template>
(7)     <xsl:template name="PrintDir">
(8)       <xsl:param name="DIR"/>
(9)       <xsl:if test="$DIR">
(10)        <xsl:call-template name="PrintDir">
(11)          <xsl:with-param name="DIR" select="$DIR/content/dir"/>
(12)        </xsl:call-template>
(13)      </xsl:if>
(14)    </xsl:template>
(15)  </xsl:stylesheet>
```

Obrázek 5.1: Program s cyklem přes pojmenovaná volání s parametry, který naše analýza neodhalí.

kde je požadován výběr elementu `b`, který je synem `a` a neobsahuje atribut `id`. Právě pomocí funkce not() v predikátu jsme zadefinovali osamostatnění vloženého pravidla `otherwise` z řídící struktury větvení `choose`.

Povinný výskyt objektu ve spojení s negací dává jistotu nepravdivosti podmínky, což omezuje omezuje šíření kontextů z nerozhodnutelnosti. Tomu odpovídá menší počet hran grafu toku řízení, zvyšuje se tak pravděpodobnost odhalení nedosažitelných pravidel, slepých volání a sníží se množství falešných cyklů.

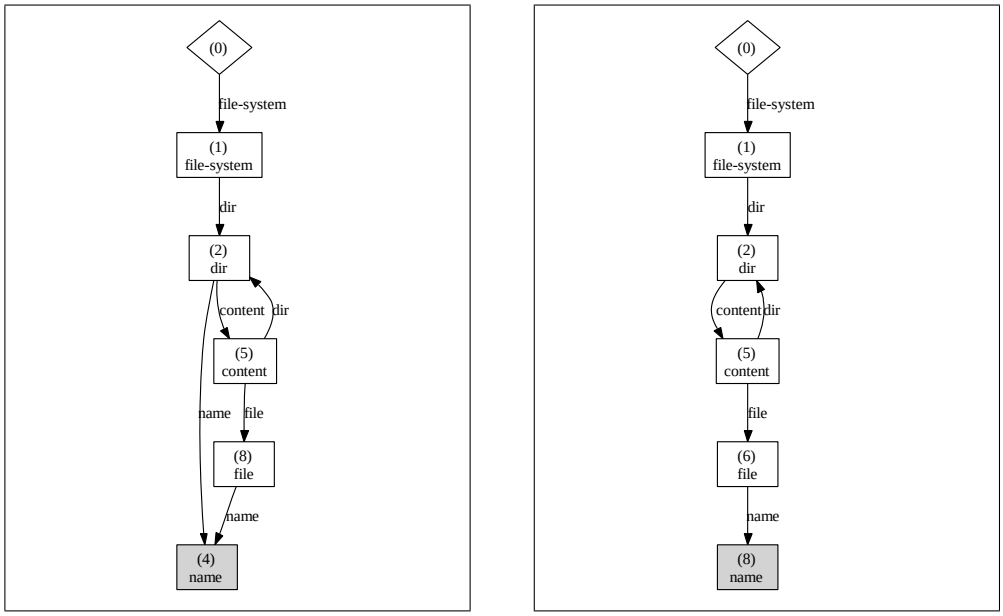
Analogicky, stejný závěr platí pro vztah modelů tree a tree^{m}.

5.2.2 Model plain vs. model tree

V oddílu 4.1.2 jsme na fragmentu programu

```
(1)  <template match="content/file">
(2)    <apply-templates select="name"/>
(3)  </template>
(4)  <template match="name">...</template>
(5)  <template match="dir/name">...</template>
```

ukázali nepřesnost toku vzniklou použitím modelu plain. Uvažujme stejnou situaci nad modelem tree, obr. 3.7. Pravidlo (1) nechť zpracovává kontextový objekt $\texttt{file}_6$. Instrukce volání (2) vybírá objekt $\texttt{name}_8$. V případě tohoto modelu má element $\texttt{name}_8$ pouze jednoho otce, kterým není $\texttt{dir}_2$, proto pravidlo



(a) (b)

Obrázek 5.2: Automaty modelů plain a tree pro vyhodnocení prioritního stínění. Šedý obdélník značí koncový stav, počáteční stav je dán kořenem modelu. Hrany grafů zobrazují přechodovou funkci. Popisek je jméno objektu ze znaku automatové abecedy. (a) automat z modelu na obr. 3.1 pro koncový stav daný objektem $\texttt{name}_4$. (b) automat z modelu na obr. 3.7 pro koncový stav daný objektem $\texttt{name}_8$.

(5) není uvažováno jako možný cíl volání. Zvoleno je tedy pouze pravidlo (4).

Podobná je situace v případě přesnosti vyhodnocení prioritního stínění. Uvažujme modifikaci diskutovaného fragmentu programu, nechť se v programu vyskytuje ještě pravidlo

```
(6)  <template match="file/name"/>...</template>
```

Uvažujeme opět volání (2) a chceme rozhodnout prioritní stínění pravidla (4) pravidlem (6) – z tvaru vzorů plyne jeho vyšší priorita.

Při vyhodnocování nad modelem plain, obr. 3.1, pravidlo (2) vybírá objekt $\texttt{name}_4$, obě uvažovaná pravidla objekt matchují. Rozebereme vztah (OA′)

Automat FA($\texttt{name}_4$, plain) je zachycen na obr. 5.2(a). Přijímací automat FA('name') pro vzor pravidla (2) je tvořen stavy $Q = \{q_0, q_F\}$, kde q_0 je počáteční a q_F koncový stav. Přechodová funkce δ je definována jako $\delta = \{\langle q_0, a, q_0\rangle \mid a \in \Sigma^{\mathsf{FA}}\} \cup \{\langle q_0, \text{'name'}, q_F\rangle\}$

Lze nahlédnout, že

$$\mathrm{FA}(\texttt{name}_4, \mathrm{plain}) \cap \mathrm{FA}(\text{'name'}) = \mathrm{FA}(\texttt{name}_4, \mathrm{plain})$$

Ze struktury doplňkového automatu $-\mathrm{FA}(\text{'file/name'})$ je pro nás nyní zajímavý jeden z koncových stavů q_F, kde $q_F \in \delta(q_{F-1}, \text{'name'})$ a neexistuje $q_{F-2} : q_{F-1} \in \delta(q_{F-2}, \text{'file'})$.[1]

Tedy průnik automatu, rozhodující prioritní stínění bude neprázdný, přenos kontextu půjde do obou pravidel (4) i (6).

Při vyhodnocování nad modelem tree, obr. 3.7, pravidlo (2) vybírá objekt $\texttt{name}_8$, který opět obě uvažovaná pravidla matchují. Automat $\mathrm{FA}(\texttt{name}_8, \mathrm{tree})$ je zachycen na obr. 5.2(b), automaty $\mathrm{FA}(\text{'name'})$ a $-\mathrm{FA}(\text{'file/name'})$ jsou zcela identické. Rozdíl je právě v automatu modelu, kde se již vyskytuje pouze jedna cesta do koncového stavu, přechody 'file' a 'name', která se z definice v doplňkovém automatu nevyskytuje.

Průnik bude v tomto případě prázdný, pravidlo (6) stíní pravidlo (4).

Vyhodnocení nad tree tedy zachycuje jemněji volací vztahy, což odpovídá menšímu počtu hran v grafu řízení. Opět tedy dochází ke zvýšení pravděpodobnosti odhalení nedosažitelných pravidel a slepých volání a naopak snížení detekce falešných cyklů.

5.2.3 Velikost modelu tree

Velikost množiny objektů modelu tree se však ukázala jako problém limitující použití tohoto modelu. V případě velkých schémat se složitou strukturou dochází kvůli stromovému charakteru modelu ke značnému nárustu počtu objektů.

Označme D jako množinu všech *deklarovatelných elementů* vstupního schématu. Tím rozumíme dvojice $\langle e_d.\texttt{name}, e_d.\texttt{type}\rangle$ z deklarací elementů e_d. Pokud bychom uvažovali patologický případ schématu, kde každý z typů obsahuje deklarace všech deklarovatelných elementů, viz schéma A.1.3, potom by model tree obsahoval $|D|!$ elementů. Podle testů, není nárust ve většině případů tak drastický, některá schémata však nebylo možné kvůli extrémnímu nárustu vymodelovat. Tabulka 5.1 zachycuje velikosti modelů části testovaných dat. Celkem 3 případy z 68 vedly k vyčerpání paměti při konstrukci modelu. Průměrný nárust počtu objektů mezi modely plain a tree byl 178 násobný se směrodatnou odchylkou 1288, resp. 205 násobný s odchylkou 1188 mezi $\mathrm{plain}^{\mathsf{m}}$ a $\mathrm{tree}^{\mathsf{m}}$. Vysoká hodnota směrodatné odchylky ukazuje vychýlení malým vzorkem „extrémních“ dat.

[1] tuto situaci v doplňkovém automatu předkládáme jako fakt

Už v případě velikosti modelů v řádu tisíců objektů vede použitá metoda vyhodnocování, vinou exponenciální složitosti vyhodnocování, k nepoužitelné době běhu analýzy.

5.2.4 Vnitřní struktura typů schématu

Ačkoli ve schématu ZSchema i v obecných principech modelování bereme v úvahu vnitřní strukturu typů vyjádřenou deklaračními skupinami, žádný z modelů tuto detailní informaci nezachycuje. Jak jsme naznačili, příčné osy, kde přesnost vyhodnocení je velmi závislá na znalostosti vnitřní struktury, nebývají v programech časté. Tato znalost však může zvýšit přesnost simulace uvažujeme-li například fragment programu

```
(1)  <template match="a">
(2)    <apply-templates select="b"/>
(3)  </template>
(4)  <template match="b">
(5)    <apply-templates select="..[c]"/>
(6)  </template>
```

a schématu

```
<element name="a" type="A"/>
<complexType name="A">
  <choice>
    <element name="b"/>
    <element name="c"/>
  </choice>
</complexType>
```

Volání na řádku (5) bude slepé, neboť schéma omezuje obsah elementu `a` na právě jednoho syna `b` nebo `c`.

Domníváme se ale, že takové rozšíření nemá u konkrétně uvažovaných modelů smysl. Model plain je ve zpětných osách ztrátový, rozšíření by v tomto směru nic nepřineslo. U modelu tree by reprezentace vnitřní struktury vedla k dalšímu nárustu množiny objektů – všechny deklarace elementu e_d, stejného jména a typu, z různých úrovní deklaračních skupin jsou nyní reprezentovány jedním objektem.

Uvedená vyjádření s minimální ztrátou informace o struktuře jsme prezentovali z důvodu obecnosti metod a jako východisko pro případné pokročilejší modely.

5.2.5 Idea hybridního modelu

Uvedené nevýhody modelů by mohl řešit hybridní přístup. Model by obsahoval pouze základní statickou strukturu, jako model plain (s případným vyjádřením vnitřní struktury), jeho objekty by pak byly generované podle potřeby simulace a obsahovaly by i informaci o svém výskytu ve formě jakési historie průchodu stromem. Realizace takového řešení by vyžadovala důkladný rozbor situací rekurzivních elementů, aby byla zajištěna konečnost množiny objektů modelu.

	ZSchema	plain	plain^{m}	tree	tree^{m}	plain_c	$\text{plain}_c^{\text{m}}$	tree_c	tree_c^{m}
StyledLayerDescriptor.xsd	145	287	340	3340938	3457474	415	357	–	–
gml.xsd	598	2263	2499	–	–	5792	2578	–	–
wfs.xsd	771	2360	2599	–	–	5366	2695	–	–
XMLSchema.xsd	57	288	296	305746	306330	292	301	529184	530473
lyirmeta.xsd	54	123	128	282	282	123	128	282	282
flat.xsd	25	57	60	9267	158362	57	60	9267	158362
Fictionbook2.xsd	61	164	175	52363	79160	168	179	53883	81460
DGL.xsd	321	456	462	35657	35657	463	463	35664	35664
Bibliography.xsd	49	107	107	298	298	107	107	298	298
DictionaryUnit.xsd	66	143	143	7858	7858	143	143	7858	7858
CatML-strict-inherit.xsd	83	154	166	15562	15562	161	166	22992	22992
stmml.xsd	28	74	74	134	134	74	74	134	134
dcmes-xml-xsd.xsd	20	57	57	102	102	57	57	102	102
gpx.xsd	35	74	74	201	201	74	74	201	201
kml_.xsd	98	205	217	2841	2841	205	217	2841	2841
rim.xsd	38	166	190	2653	2791	227	226	3719	3881
cmlCore.xsd	36	347	359	955	955	347	359	955	955
Variable.xsd	18	53	53	131	131	53	53	131	131
xliff-core-1.1.xsd	40	285	305	44203	44203	285	305	44203	44203
uddi_v3policy.xsd	216	220	236	2908	2908	220	236	2908	2908

Tabulka 5.1: Srovnání velikosti modelů schématu. Hodnoty ve sloupci ZSchema jsou počty deklarovatelných elementů, v ostatních sloupcích jsou to počty objektů. Pomlčka označuje případy, ve kterých došlo při konstrukci k vyčerpání paměti.

5.3 Související práce

Více prací se snaží řešit komplikovanější úlohu, typovou kontrolu XSLT [4, 5, 27]. Práce v tomto směru většinou staví na formálních metodách odvozování typů, mají jen málo společné s našimi vytyčenými cíly detekce chyb.

Dong a Bailey

Jak jsme zmínili v úvodu, vycházíme zejména z práce Static Analysis of XSLT programs [1]. Dong a Bailey analyzují chyby v XSLT programech na základě znalosti DTD vstupních dokumentů. Za důležité považují detekovat *nedosažitelná pravidla*, *chybějící pravidla*,*chybné volací vztahy* a *nekonečnost běhu*. Nedosažitelná pravidla a nekonečnost běhu jsou předmětem analýzy i v našem případě. Pod chybnými volacími vztahy rozumí situace, kdy je v pravidle požadováno zpracování neexistujících uzlů. V naší analýze je tento případ zahrnut mezi slepými cestami. Případ chybějícího pravidla, podle Donga a Baileyho, nastává, pokud je v programu požadováno zpracování uzlu, který nematchuje žádné pravidlo. Vzhledem k tomu, že XSLT obsahuje built-in pravidla, která se aplikují v případě, že není definováno uživatelské pravidlo, je přínos této detekce sporný.

Analýzu provádějí s využitím grafu nazvaným TAG (template association graph). Uzly odpovídají pravidlům programu, hrany aproximují volací vztahy mezi pravidly. Hrany TAGu jsou konstruovány ve dvou krocích. Nejprve jsou vytvořeny hrany mezi uzly bez ohledu na schéma vstupu, pouze z tvaru programu na základě blíže neuvedeného vyhodnocení průniku XPath výrazů. V druhém kroce je provedeno zjemnění podle informací ze schématu. Zjemněný TAG spolu s rozdíly od hrubého TAGu slouží k rozpoznání vybraných problémů.

Zjemněný TAG je však stále hrubá aproximace volacích vztahů. Vyjadřuje pouze možnost volání pravidla. Hrany grafu nijak neodrážejí běh programu. Použití pravidla je v grafu zachyceno pouze na základě kompatibility kroků výběrového výrazu instrukce `apply-templates` a vzoru pravidla.

Například pro následující fragment XSLT programu nesprávně detekují možnost volání pravidla (1) instrukcí (3).

```
(1)  <template match="c/a">...</template>
(2)  <template match="b">
(3)    <apply-templates select="a"/>
(4)  </template>
```

Nutno přiznat, že při vyhodnocování nad modelem plain, se naše simulace zachová stejně a v grafu řízení bude figurovat toto volání.

Analýza Donga a Baileyho si neklade za cíl postihnout všechny aspekty XSLT/XPath. Uvažují například jen některé osy XPath, obsah predikátu omezují pouze na jednu cestu.

Olesen

Olesen se ve své práci Static Validation of XSLT Transformations [2] zabývá pokročilejší úlohou, statickou typovou kontrolou XSLT se znalostí DTD schémat. Jeho práce se opírá o strukturu nazvanou Summary graph. Jedná se o fragment stromu výstupního dokumentu, který obsahuje tzv. *díry*, na které se napojují jiné summary grafy. Počáteční grafy jsou zkonstruovány pro každé z pravidel programu, díry figurují v místech instrukcí volání. Analýza toku řízení programu nad schématem vstupu potom dává informaci o napojení grafů, typová kontrola je dána porovnáním výsledného grafu se schématem výstupu.

Právě z hlediska analýzy řízení pro nás byla Olesenova práce inspirující. Olesen provádí simulaci běhu programu propagací kontextu. Protože pracuje s DTD, které jednoznačne identifikuje typ objektu jménem, kontext charakterizuje textově tímto jménem.

Vrcholy grafu toku řízení tvoří pravidla programu, hrany vyjadřují použití pravidla pro určitý kontext. Volací vztahy jsou zachyceny přesněji než v TAGu Donga a Baileyho. Jsou výsledkem testu kompatibility vzoru potenciálního cílového pravidla a spojení vzoru zdrojového pravidla s uvažovaným kontextem vyhodnocování a výběrového výrazu právě uvažované instrukce volání.

Test kompatibility je dvoukrokový. První krok, *path simulation test* simuluje průchod obou cest podle schématu. Kompatibilita je pak dána neprázdností průniku výsledných množin obou cest. Nepřesnost tohoto testu plyne z faktického porovnání pouze posledního kroku cest. Například cesty `a/c` a `b/c` jsou tímto testem prohlášeny za ekvivalentní.

Ke zpřesnění slouží druhý krok, *path automata test*. Pro obě uvažované cesty a schéma jsou zkonstruovány konečné automaty, neprázdnost jejich průniku pak rozhoduje kompatibilitu. V tomto případě je tedy zohledňeno i několik posledních kroků cest. Tento test však nedokáže zpracovat jiné než dopředné osy.

Protože kontext je charakterizován pouze jménem objektu, není uvažována poloha objektu v potenciálním dokumentu. Jinými slovy, kontext si nepamatuje své předchůdce. Průchodem zpětných os dochází ke ztrátě přesnosti, jako u našeho modelu plain.

Olesen také zjednodušuje jazyk XPath, je zcela vynecháno vyhodnocování predikátů. Vyhodnocování a předávání parametrů také není postihnuto.

Kapitola 6

Závěr

Cílem této práce bylo staticky analyzovat běhové chyby v XSLT programech. Na základě existujících prací jsme se vydali cestou zkoumání toku řízení programu na schématicky popsaných vstupech.

Zavedli jsme abstrakci modelování vstupů, odděleně od zbytku analýzy. Model slouží jako vstup kvazi-simulace běhu, která zpracovává řídící instrukce programu. Naší snahou bylo postihnutí co největšího počtu aspektů vstupních jazyků. Za úspěch považujeme dostatečně obecné vyhodnocování XPath výrazů a zahrnutí odhadu předávání hodnot parametrů pravidel jako součást analýzy toku řízení programu. Podle dostupných informací se zpracováním parametrů žádná ze souvisejících prací nezabývá.

Prezentovaný algoritmus iterativní konstrukce toku, který je výchozí pro následnou detekci problémů, je exponenciální časové složitosti. Nicméně exponeciální člen, kterým je vyhodnocování výrazů, by podle existujících studií měl být nahraditelný polynomiálním. Ukázali jsme že za takové podmínky by byl diskutovaný algoritmus polynomiální.

Prezentované algoritmy byly naprogramovány jako samostatná aplikace provádějící detekci vybraných chyb. Jako značný problém se ukázala špatná dostupnost testovacích dat. Pro analýzu je nezbytné mít k dispozici dvojici program a schéma vstupu, spolu se však ve veřejně dostupných zdrojích vyskytují jen zřídka. Algoritmy a modely byly, kromě umělých dat vytvořených pro účely vývoje, otestovány na vzorku reálných dat. Jeho velikost však není zdaleka ideální pro formulování jednoznačných závěrů.

Samozřejmě i v této práci existuje prostor pro možná zlepšení. Domníváme se, že mimo již uvedený polynomiální algoritmus vyhodnocování je pole pro experimentování otevřeno odděleným modelováním vstupů.

Všechny idee zlepšování a zobecňování však musí mít své hranice, zadání práce proto považujeme za splněné.

Literatura

[1] C. Dong, J. Bailey (2004): *Static Analysis of XSLT Programs*, Conferences in Research and Practice in Information Technology, Vol. 27, 151–160

[2] M. K. Olesen (2004): *Static Validation of XSLT Transformations*, Master's thesis

[3] W3C (2004): *Extensible Markup Language (XML) 1.0 (Third Edition)*, http://www.w3.org/TR/2004/REC-xml-20040204

[4] P. Audebaud, K. Rose (2000): *Stylesheet validation*, Technical Report RR2000-37, ENS-Lyon

[5] A. Tozawa (2001): *Towards static type checking for XSLT*, Proceedings of the 2001 ACM Symposium on Document engineering, 18–27

[6] M. Murata, A. Tozawa, M. Kudo, S. Hada (2003): *XML Access Control Using Static Analysis*, Proceedings of the 10th ACM conference on Computer and communications security, 78–84

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

[8] W3C (2004): *XML Schema Part 0, 1, 2*, http://www.w3.org/TR/2004/REC-xmlschema-0-20041028/

[9] G. Gottlob, C. Koch, R. Pichler (2003): *XPath processing in nutshell*, ACM SIGMOD Record, Vol. 32, Issue 2, 21–27

[10] W3C (1999): *XSL Transformations (XSLT) Version 1.0*, http://www.w3.org/TR/1999/REC-xslt-19991116

[11] F. Neven, T. Schwentick (2003): *XPath containment in the presence of disjunction, DTDs, and variables* , Lecture Notes in Computer Science, Vol. 2572, 312–326

[12] S. Kepser (2002): *A Proof of the Turing-completeness of XSLT and XQuery*

[13] P. Wadler (2000): *A formal semantics of patterns in XSLT*

[14] E. Bae, J. Bailey (2003): *CodeX: An approach for debugging XSLT transformations*, Proceedings of the Fourth International Conference on Web Information Systems Engineering

[15] D. Lee, W. W. Chu (2000): *Comparative Analysis of Six XML Schema Languages*, ACM SIGMOD Record, Vol. 29, Issue 3, 76–87

[16] P. T. Wood (2003): *Containment for XPath fragments under DTD constraints*, Lecture Notes in Computer Science, Vol. 2572, 297–311

[17] A. Moller (2002): *Document Structure Description 2.0*, http://www.brics.dk/DSD/dsd2.html

[18] G. Gottlob, C. Koch, R. Pichler (2002): *Efficient Algorithms for Processing XPath Queries*, Proceedings of the 28th VLDB Conference

[19] G. J. Bex, S. Maneth, F. Neven (2000): *Expressive Power of XSLT*

[20] M. Murata (2001): *Extended path expressions of XML*, Proceedings of the twentieth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems, 126–137

[21] *HaXml*, http://www.cs.york.ac.uk/fp/HaXml/

[22] OASIS (2001): *RELAX NG Specification*, http://relaxng.org/spec-20011203.html

[23] M. Fitzgerald (2003): *Relaxer Tutorial* , http://www.relaxer.org/doc/tutorial/tutorial.html

[24] C. Kirkegaard, A. Moler, M. I. Schwarzbach (2004): *Static analysis of XML transformations in Java*, IEEE Transactions on Software Engineering, Vol. 30., No. 3, 181–192

[25] M. Murata, D. Lee, M. Mani, K. Kawaguchi (2004): *Taxonomy of XML schema languages using formal language theory*, ACM Journal Name, Vol. V, No. N, 1–44.

[26] J. Simeon, P. Wadler (2003): *The essence of XML*, Proceedings of the 30th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, 1–13

[27] T. Milo, D. Suciu, V. Vianu (2000): *Typechecking for XML transformers*, Journal of Computer and System Sciences 66 (2003), 66–97

[28] *Uniform Resource Identifiers (URI): Generic Syntax*, http://www.ietf.org/rfc/rfc2396.txt

[29] H. Hosoya, B. C. Pierce: *XDuce: A typed XML Processing Language (Preliminary report)*, Selected papers from the Third International Workshop WebDB 2000 on The World Wide Web and Databases, 226–244

[30] W3C (2001): *XML Pointer Language*, http://www.w3.org/TR/WD-xptr

[31] *Bison*, http://www.gnu.org/software/bison/

[32] *Boost C++ Libraries*, http://www.boost.org

[33] *Datapower XSLTMark*, www.datapower.com/xmldev/xsltmark.html

[34] *Flex*, http://www.gnu.org/software/flex/

[35] Petr Doležal (2005): *Modul Smart, část EBM projektu*

[36] *Sarvega XSLT Benchmark*, http://www.sarvega.com/xslt-benchmark.html

[37] *Xalan XSLT processor*, http://xalan.apache.org/

[38] *Xerces C++ parser*, http://xml.apache.org/xerces-c/

Příloha A

Příklady

A.1 Schémata

Schéma A.1.1

```
<xsd:schema version="1.0" xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="x">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="a" type="Type"/>
        <xsd:element name="b" type="Type"/>
        <xsd:element name="c" type="AnotherType"/>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
  <xsd:complexType name="Type">
    <xsd:sequence>
      <xsd:element name="item" type="ComplexItem"/>
    </xsd:sequence>
  </xsd:complexType>
  <xsd:complexType name="ComplexItem">
    <xsd:sequence>
      <xsd:element name="description" type="xsd:string"/>
    </xsd:sequence>
  </xsd:complexType>
  <xsd:complexType name="AnotherType">
    <xsd:sequence>
      <xsd:element name="item" type="xsd:string">
    </xsd:sequence>
  </xsd:complexType>
</xsd:schema>
```

Schéma A.1.2 „file-system“

Schéma reprezentace jednoduchého souborového systému v XML. S hard-linky souborů a acyklickou adresářovou strukturou.

```
<xsd:schema version="1.0" xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="file-system">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="dir" type="Directory"/>
        <xsd:element name="files">
          <xsd:complexType>
            <xsd:sequence minOccurs="0" maxOccurs="unbounded">
              <xsd:element name="file" type="File"/>
            </xsd:sequence>
          </xsd:complexType>
        </xsd:element>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
  <xsd:complexType name="Directory">
    <xsd:sequence>
      <xsd:element name="name" type="xsd:string"/>
      <xsd:element name="content">
        <xsd:complexType>
          <xsd:choice minOccurs="0" maxOccurs="unbounded">
            <xsd:element name="dir" type="Directory"/>
            <xsd:element name="file" type="FileRef"/>
          </xsd:choice>
        </xsd:complexType>
      </xsd:element>
    </xsd:sequence>
  </xsd:complexType>
  <xsd:complexType name="File">
    <xsd:sequence>
      <xsd:element name="content" type="xsd:string"/>
    </xsd:sequence>
    <xsd:attribute name="id" type="xsd:string"/>
  </xsd:complexType>
  <xsd:complexType name="FileRef">
    <xsd:sequence>
      <xsd:element name="name" type="xsd:string"/>
    </xsd:sequence>
    <xsd:attribute name="ref" type="xsd:string"/>
  </xsd:complexType>
</xsd:schema>
```

Schéma A.1.3 „Noční můra modelu tree“

Konstrukce modelu tree z typu schémat, jako následující příklad, vede k faktoriálovému počtu objektů.

```
<xsd:schema version="1.0" xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="a" type="A"/>
  <xsd:complexType name="A">
    <xsd:sequence>
      <xsd:element name="a" type="A" minOccurs="0"/>
      <xsd:element name="b" type="B" minOccurs="0"/>
      <xsd:element name="c" type="C" minOccurs="0"/>
    </xsd:sequence>
  </xsd:complexType>
  <xsd:complexType name="B">
    <xsd:sequence>
      <xsd:element name="a" type="A" minOccurs="0"/>
      <xsd:element name="b" type="B" minOccurs="0"/>
      <xsd:element name="c" type="C" minOccurs="0"/>
    </xsd:sequence>
  </xsd:complexType>
  <xsd:complexType name="C">
    <xsd:sequence>
      <xsd:element name="a" type="A" minOccurs="0"/>
      <xsd:element name="b" type="B" minOccurs="0"/>
      <xsd:element name="c" type="C" minOccurs="0"/>
    </xsd:sequence>
  </xsd:complexType>
</xsd:schema>
```

A.2 Programy

Program A.2.1 „Postupná konstrukce“

Program pro ukázku postupné konstrukce grafu toku řízení. Jeho běh nad schematem A.1.1 je uveden na obr. 4.1. Jedná se vskutku o samoúčelný program, do kterého je záměrně nahuštěno použití parametrů.

```
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:template match="/">
    <xsl:apply-templates select="x/a">
      <xsl:with-param name="P" select="x/c"/>
    </xsl:apply-templates>
  </xsl:template>
  <xsl:template match="*">
    <xsl:param name="P"/>
    <xsl:apply-templates select="$P">
      <xsl:with-param name="P" select="."/>
    </xsl:apply-templates>
  </xsl:template>
  <xsl:template match="c">
    <xsl:param name="P"/>
    <xsl:value-of select="$P"/>
  </xsl:template>
</xsl:stylesheet>
```

Program A.2.2 „Cykly v grafu toku řízení“

Graf řízení běhu uvedeného programu nad modelem z obr. 3.1, schematu A.1.2 obsahuje dopředný cyklus zachycený na obr. 4.4 a také zdánlivý cyklus, obr. 4.3.

```
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:template match="/">
    <xsl:apply-templates select="/file-system/dir"/>
  </xsl:template>
  <xsl:template match="content">
    <xsl:apply-templates/>
  </xsl:template>
  <xsl:template match="dir">
    <xsl:apply-templates select="content"/>
  </xsl:template>
  <xsl:template match="content/file">
    <xsl:apply-templates select="//file[@id = current()/@ref]"/>
  </xsl:template>
  <xsl:template match="files/file">
    <xsl:apply-templates/>
  </xsl:template>
</xsl:stylesheet>
```

Program A.2.3 „Zavlečený cyklus“

Demonstrační program pro odstraňování zavlečených cyklů popsané v oddílu 4.4.2.

```
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:template match="x">
    <xsl:apply-templates select="a">
      <xsl:with-param name="FRIEND" select="b"/>
    </xsl:apply-templates>
    <xsl:apply-templates select="b">
      <xsl:with-param name="FRIEND" select="a"/>
    </xsl:apply-templates>
  </xsl:template>
  <xsl:template match="a | b">
    <xsl:param name="FRIEND"/>
    <xsl:apply-templates select="$FRIEND"/>
    ...
  </xsl:template>
</xsl:stylesheet
```

Příloha B

Algoritmy

B.1 Algoritmus konstrukce grafu řízení

Ke struktuře grafu řízení, viz sekce 4.1.1, dodefinujme následující mapování pro zpřehlednění zápisu algoritmu, jejich význam je intuitivní.

$rule : n \in N \mapsto <$ `template` $>$

$target : E \to N;\ target(\langle n, a, m\rangle) = m$

$source : E \to N;\ source(\langle n, a, m\rangle) = n$

$instr : e \in E \mapsto <$ `apply` $-$ `templates` $>;\ instr(\langle n, a, m\rangle) = a$

Pro konstrukci toku uvažujeme pomocnou strukturu

$matched : N \to 2^O$

která obsahuje objekty modelu dokumentu, které pravidlo matchuje.

B.1.1 Inicializace

[1] **foreach** $n \in N, p \in rule(n).<$ `param` $>$ **do**
[2] $\quad P(n, p.\texttt{name}) := \text{Eval}(p.\texttt{select}, \emptyset, P(n))$
[3] **done**
[4] $\forall n \in N : matched(n) := \text{Eval}(rule(n).\texttt{match}, \{\rho\})$
[5] vyber pravidlo n_0 pro zpracování kořene ϱ[1]
[6] $pending(n_0) := \{\rho\}$
[7] $\forall n \in N, n \neq n_0 : pending(n) := \emptyset$
[8] $queue := \emptyset$

[1] Pro úplnost: $n_0 \in N : \rho \in matched(n_0) \wedge \forall n \in N : (rule(n_0).\texttt{precedence} \geq rule(n).\texttt{precedence} \wedge rule(n_0).\texttt{priority} > rule(n).\texttt{priority})$

B.1.2 Konstrukce toku

[1] **while** $\exists n \in N : \mathit{pending}(n) \neq \emptyset$ **do**
[2] vyber $c \in \mathit{pending}(n)$
[3] **foreach** $a \in \mathit{rule}(n).<\texttt{apply}-\texttt{templates}>$ **do**
[4] $S := \mathrm{Eval}(a.\texttt{select}, \{c\}, P(n))$
[5] $T := \{m \in N \mid \mathit{matched}(m) \cap S \neq \emptyset \wedge \mathit{rule}(m).\texttt{mode} = \mathit{rule}(n).\texttt{mode} \wedge$
$\mathit{rule}(m).\texttt{precedence} \in \langle a.\texttt{minPrecedence}, a.\texttt{maxPrecedence}\rangle\}$
[6] $E_{new} := \{\langle n, a, m\rangle \mid m \in T\}$
[7] $F_{new} := \{\langle e, c, S \cap \mathit{matched}(\mathit{target}(e))\rangle \mid e \in E_{new}\}$
[8] $\mathrm{Override}(E_{new}, F_{new}, c)$
[9] $E_{new} := \{e \in E_{new} \mid F_{new}(e, c) \neq \emptyset\}$
[10] $E \cup= E_{new}$
[11] $F \cup= F_{new} \restriction E_{new}$
[12] **foreach** $e \in E_{new}$ **do**
[13] $\mathit{pending}(\mathit{target}(e)) \cup= F(e, c) \setminus C(\mathit{target}(e))$
[14] $\mathit{queue} \cup= \mathit{target}(e)$
[15] **done**
[16] **done**
[17] $\mathit{pending}(n) \setminus= \{c\}$
[18] $C(n) \cup= \{c\}$
[19] **done**

B.1.3 Propagace parametrů

[1] **foreach** $n \in \mathit{queue}, p \in \mathit{rule}(n).<\texttt{param}>$ **do**
[2] $P(n, p.\texttt{name}) := \mathrm{Eval}(p.\texttt{select}, C(n), P(n))$
[3] **done**
[4] **while** $\exists n \in \mathit{queue}$ **do**
[5] $\mathit{queue} \setminus= \{n\}$
[6] **foreach** $e \in E : \mathit{source}(e) = n, w \in \mathit{instr}(e).<\texttt{with}-\texttt{param}>$ **do**
[7] $X := \mathrm{Eval}(w.\texttt{select}, \{o \in C(n) \mid F(e, o) \neq \emptyset\}, P(n))$
[8] **if** $X \not\subseteq P(\mathit{target}(e), w.\texttt{name})$ **then**
[9] $P(\mathit{target}(e), w.\texttt{name}) \cup= X$
[10] $\mathit{pending}(\mathit{target}(e)) := C(\mathit{target}(e))$
[11] $\mathit{queue} \cup= \{\mathit{target}(e)\}$
[12] **endif**
[13] **done**
[14] **done**

Příloha C

Obsah přiloženého CD

Součástí práce je disk CD obsahující digitální verzi textu, aplikaci WXSA implementující popisované algoritmy a příklady. Instalaci a použití aplikace se věnuje následující dodatek.

Adresářová struktura

\3rd	Binární soubory použitých knihoven a aplikací třetích stran nezbytné pro běh WXSA.
\bin	Spustitelný soubor aplikace (wxsa.exe), pomocné skripty.
\doc	Text diplomové práce v digitální podobě. Zahrnuje dokument ve formátu PDF a zdrojové texty pro systém TEX.
\examples	Sada XSLT programů a schemat pro ukázku funkčnosti aplikace.
\sources	Zdrojové texty aplikace včetně nástrojů a knihoven třetích stran, nezbytných k překladu aplikace.

Příklady

V adresáři \examples\internal jsou přiloženy některé z uměle vytvořených příkladů demonstrující, jaké problémy v XSLT programech analýza postihuje a jaké ne. Adresář \examples\real obsahuje reálná data, která jsme měli k dispozici.

V každém adresáři příkladu jsou připraveny dávkové skripty _run-*.bat pro pohodlné spuštění aplikace WXSA (viz následující dodatek), bez nutnosti zadávání argumentů příkazové řádky. Hvězdička zastupuje zkratku názvu modelu (pdm – plain, pmdm – plain^{m}, ...), se kterým bude analýza pracovat.

Adresář \examples\xsd obsahuje nasbíraná samostatná schémata. Data v něm nejsou nijak uspořádána.

Příloha D

Aplikace WXSA

Součástí této práce je také aplikace WXSA, ve které je implementována analýza zkoumaných problémů. WXSA je konzolová aplikace, uživatelské rozhraní je příkazová řádka. Vstup tvoří zkoumaný XSLT program spolu se schematem XML dokumentů, ke kterým se program váže. Výstupem jsou primárně textová hlášení o detekovaných problémech.

Tento dodatek slouží jako návod k instalaci aplikace a také jako stručný popis použití. Účel aplikace je zejména demonstrace popisovaných algoritmů, obsahuje tedy řadu parametrů pro generování dodatečných informací – ať už pro dodatečné informace o simulaci běhu zkoumaného programu, či pro účely ladění. Omezíme se však pouze na popis základního použití, kompletní reference by byla nad rámec této práce.

D.1 Překlad a instalace aplikace

Cílová platforma

Aplikace byla vyvíjena na platformě Win32 ve vývojovém prostředí Microsoft Visual Studio 7.1, použitým jazykem je C++. Ačkoli program nevyužívá žádné specifické rysy překladače ani platformy[1], cílem práce nebylo vytvořit přenositelnou aplikaci, na jiných platformách a překladačích jsme překlad nezkoumali.

[1] pomineme-li rozšířené zobrazení výstupů v aplikaci Internet Explorer, viz oddíl Použití aplikace

Použité knihovny a nástroje

Aplikace WXSA využívá knihovnu *Xerces* [38] k parsování XML vstupů. Část „filesystem“ knihovny *Boost* [32] pro navigaci na souborovém systému. Modul *Smart* [35] implementující smart pointery v jazyce C++ a nástroje *Bison* [31] a *Flex*[2] [34] pro implementaci parseru XPath výrazů.

Pro vytvoření HTML verze výstupu je využit XSLT procesor *Xalan* [37].

Všechny použité součásti třetích stran jsou volně šiřitelné a jsou obsaženy na přiloženém disku.

Překlad

Pro překlad aplikace ze zdrojových textů je nutno mít zmiňované Microsoft Visual Studio 7.1[3] s překladačem jazyka C++.

Postup překladu aplikace

1. Vytvořte na zapisovatelném disku adresář a zkopírujte do něj obsah \sources. Dále budeme uvažovat, že vytvořený adresář je c:\wxsa.
2. V souboru c:\wxsa\tools\bin\bison.bat změňte proměnnou *PATH* na adresář, ve kterém se tento soubor nachází.
3. Otevřte c:\wxsa\wxsa.sln ve Visual Studiu.
4. Nastavte mód překladu na „Release“ a proveďte překlad. Vznikne spustitelný soubor aplikace, c:\wxsa\src\wxsa\Release\wxsa.exe.

Instalace

1. Zvolte některý z adresářů, které odkazuje systémová proměnná *PATH*, nebo vytvořte nový adresář a upravte proměnnou *PATH*. Uvažujme adresář c:\bin.
2. Do adresáře c:\bin zkopírujte obsah adresářů \bin a \3rd z CD. Pokud jste provedli překlad, zkopírujte výsledný wxsa.exe soubor také do adresáře c:\bin.
3. Pokud si přejete zobrazovat výsledky ve formě HTML, proveďte následující kroky.

[2] jemně modifikovanou verzi wflex++

[3] překlad ve vyšších verzích jsme nezkoušeli

4. Nastavte hodnotu proměnné *RESULT_XSLT* v souboru c:\bin\wxsa_result2html.bat, aby odkazovala plnou cestou soubor result.xsl.

5. Nastavte hodnotu parametrů *RESULT_CSS* a *RESULT_JS* jako URL souborů result.css a result.js.

D.2 Použití aplikace

Pokud jste provedli kroky ze sekce instalace, spustíte aplikaci příkazem tvaru:

```
wxsa.exe [PARAMETRY] -p program.xsl -s schema.xsd
```

Informace o detekovaných problémech aplikace implicitně vypisuje na standardní výstup. Výstup je možno také uložit jako mezivýsledný XML dokument a ten transformovat přiloženým XSLT programem do formátu HTML. Výsledný dokument obsahuje program v JavaScriptu, který využívá rysů prohlížeče Internet Explorer.

Aplikace vyžaduje, aby vstupy byly korektní dokumenty jazyků XSLT a XML Schmema. Ačkoli během parsování vstupů jsou prováděny kontroly správnosti, cílem nebylo vytvořit úplný parser těchto jazyků, spuštění na nekorektních vstupech proto může vést ke zcela neočekávaným výsledkům.

Parametry příkazové řádky

-p SOUBOR	Soubor se vstupním XSLT programem. Případné includované a importované součásti jsou automaticky otevřeny[4].
-s SOUBOR	Soubor se schematem vstupů analyzovaného programu. Případné includované a importované součásti jsou automaticky otevřeny[4].
-M	Analýza pravidel programu, která nebudou nikdy použita, neboť nematchují žádný objekt vstupních dokumentů.
-U	Analýza nedosažitelných pravidel programu.
-C	Analýza cyklického použití pravidel v simulovaném běhu programu.
-B	Analýza slepých volání.
-G	Analýza slepých cest v globálních výrazech.
-P	Analýza slepých cest v neřídících výrazech pravidel.
-A	Kompletní analýza všech uvedených problémů.

-m MODEL	Volba modelu dokumentu, nad kterým bude analýza probíhat. Možnosti jsou *pdm*, *pmdm* a *tdm*, což odpovídá modelům plain, resp. plain^{m}, resp. tree.
-r	Konstrukce modelu včetně objektů uzávěru dědičné hierarchie.
--de-name JMÉNO	Konstrukce modelu dokumentu od kořenového elementu zadaného jména. Pokud tento parametr není zadán konstruuje se model od prvního deklarovaného elementu ve schematu.
-x	Výstup výsledků do souboru result.xml (ze kterého lze vygenerovat HTML dokument).
--no-stdout	Vypnutí výstupu výsledků na standardní výstup.
-v	Zvýšení podrobnosti textového výstupu. Lze kumulovat, maximální úroveň je 3. Slouží zejména jako indikace činnosti aplikace.
-h --help	Klasický help–výpis všech parametrů aplikace se stručným popisem.

Textový výstup běhu aplikace

Textový výstup aplikace přiblížíme nejlépe příkladem, viz Obr. D.1. Výstup je rozčleněn podle analyzovaných problémů do sekcí, ve kterých figurují odkazy do vstupních souborů. Protože parser Xerces neuchovává poziční informace výskytu XML objektů v souboru, jsou odkazy zapisovány formou posloupnosti XML elementů od odkazovaného objektu ke kořenovému elementu.

Interní reprezentace XSLT programu a schematu není identická se vstupy, což se ve výstupu projevuje následovně:

Jména objektů a namespace

Jednotlivé vstupní soubory mohou používat různé asociace prefixů a namespace, ve výstupech jsou proto u objektů s neprázdným namespace použity číselné prefixy, např. `3:doc(1)`. Výstup je pak zakončen tabulkou mapování těchto prefixů na namespace.

Údaj v závorce je interní identifikátor modelovaného objektu, není součástí jména.

[4] Vkládané součásti do schemat i programů jsou běžně odkazovány pomocí URL a míří na nějaký webový server. Použitý parser XML dokumentů, Xerces, umožňuje dokumenty otevřít i na základě URL. Pro případ zobrazení výsledků v HTML formě doporučujeme upravit vstupní soubory, aby odkazovaly jen součásti uložené na disku.

```
...

ANALYZING UNMATCHABLE RULES
{4:doc}<xsl:template match="x:doc|y:doc">
    @ file:///c:/examples/ns/trans.xsl

...

ANALYZING BLIND PATHS
in rule : {3:doc}<xsl:template match="x:doc|y:doc">
    @ file:///c:/examples/ns/trans.xsl
    path: {3:b}<xsl:value-of select="x:b"><p><div>
        with context: 3:doc(1)

Finalizing
URI mappings
1       http://worm.matfyz.cz/xmlns/ZSchema
2       *
3       x-namespace
4       y-namespace
```

Obrázek D.1: Příklad výstupu aplikace WXSA

Upravené XPath výrazy

Výrazy vstupního programu jsou v aplikaci vnitřně uchovávány v pozměněné formě. Ve výrazech je např. provedeno nahrazení nevyhodnotitelných podvýrazů, substituce globálních proměnných, ... Ve výstupu je uživateli zobrazován právě tento interní tvar, upravené výrazy jsou uváděny ve složených závorkách, např. `{4:doc}`.

Stejně jako u jmen objektů, namespace jsou ve výrazech také zastoupeny číselnými prefixy.

Zobrazení výsledků prohlížečem Internet Explorer

Textový výpis analýzy může být v případě velkých programů[5] poměrně rozsáhlý. Pokud navíc uživatel nemá zkoumaný program dobře nastudovaný, může se výstup jevit jako velmi nepřehledný. Proto aplikace obsahuje možnost výstupu do XML (viz parametry příkazové řádky), který lze transformovat do HTML podoby.

Transformace není provedena v rámci běhu aplikace, je nutno ji provést ručně. Výstupní soubor result.xml je uložen do pomocného podadresáře, jehož

[5]které mohou obsahovat spoustu potenciálních chyb :-)

název odráží provedenou analýzu. Název je tvaru

model -- schema -- kořenový element -- program,

tedy např. pmdm--FictionBook2--FictionBook--html. V tomto podadresáři spusťte skript wxsa_result2html.bat, který vygeneruje soubor result.html.

Tento HTML dokument obsahuje výsledky analýzy provázané s textovou podobu vstupního programu a schematu. JavaScript v pozadí zajišťuje navigaci z výsledků na odkazovaná pravidla a instrukce vstupního programu, či na deklarace ve schematu, které odpovídají odkazovaným objektům modelu dokumentu. Náhled na výsledky tvoří celkem tři pohledy. Prohlížeč, ve kterém byl soubor výsledků zobrazen se chová jako hlavní okno. Podle potřeby je pak otevřeno okno náhledu na programy a okno náhledu na schéma. V oknech náhledů není navigace na pravidla programu a deklarace schématu podporována.

Doplňky

V případě, že máte v systému příkazový interpret bash, můžete ke spuštění aplikace používat skript c:\bin\wxsa.sh, který zavádí nové parametry příkazové řádky. Před prvním použitím skriptu v něm nastavte hodnoty proměnných *RELEASE_BUILD* a *DEBUG_BUILD*. Podrobnější popis najdete v komentáři skriptu.