MATEMATIČKI FAKULTET

UNIVERZITET U BEOGRADU

Magistarski rad

Automatizacija procesa generisanja i validacije formi u poslovnim aplikacijama

Andrija Jandrlić

Mentor: dr Dušan Tošić

Beograd, 2012.

Predgovor

Ova magistarska teza nastala je kao rezultat rada na ispitivanju mogućnosti automatskog generisanje veb formi i validacije podataka koji se u okviru njih nalaze. Ideja za automatizaciju nastala je kao proizvod višegodišenjeg rada na razvijanju različitih poslovnih aplikacija za potrebe državnih uprava.

Rad je organizovan u šest glava. U uvodnoj glavi dat je opis klasičnih veb formi, metodologija i najčešći problemi njihovog razvoja. Opisan je takođe proces validacije podataka. Istaknuti su osnovni ciljevi i plan istraživanja. U prvoj glavi dat je detaljniji prikaz XML-a i XML-zasnovanih jezika. U okviru glave 2, detaljno je opisan XML-zasnovani XForms jezik za razvijanje veb formi. Jasno razdvajanje modela od kontrola korisničkog interfejsa u definiciji XForms forme olakšava automatizaciju njenog generisanja. Glave 4 i 5 predstavljaju centralne delove rada. U glavi 4 opisani su koncepti validacije i implementacija validacione biblioteke koja se sastoji od semantičkog modela validacije, semantičkog modela sistema za proizvodnju validacionih pravila, kao i unutrašnjeg domenski-specifičnog jezika sistema za proizvodnju validacionih pravila. Takodje su razmatrani XML šema, jezik za definiciju podataka relacionih baza i OCL jezik kao kandidati za opis ograničenja modela. U glavi 5 detaljno je predstavljena realizacija generatora formi, zasnovanog na XForms jeziku. Na kraju, u okviru zaključka, sumirani su dobijeni rezultati i prikazani mogući pravci u daljem radu.

Ovom prilikom želim da se zahvalim svom mentoru profesoru Dušanu Tošiću, na izuzetnoj pomoći prilikom izrade ove teze. Prof. Tošić je više puta detaljno pročitao različite verzije magistarske teze i svojim primedbama i sugestijama značajno doprineo njenom kvalitetu. Zahvaljujem se i članovima komisije za pregled i ocenu rada – prof. Nenadu Mitiću, prof. Ivanu Aranđeloviću i prof. Vladimiru Filipoviću.

Veliku zahvalnost dugujem svom kolegi Ivanu Sladoju, na ohrabrivanju, idejama, kritikama i nemerljivoj pomoći oko izrade generatora i validacione biblioteke. Zahvaljujem se Aareu Laponenu i Draganu Maleševu na vremenu i idejama kojima su doprineli da ovaj rad bude bolji.

Svom ocu Željku i majci Zorici zahvaljujem se na tome što su me od mojih prvih godina učili o tome koliko je nauka lepa i koliko je lepo baviti se njome i u njoj uživati. Majci i sestri Ani se zahvaljujem i na bezrezervnoj podršci koju su mi pružili tokom rada na tezi.

Najveću zahvalnost dugujem ćerki Ivi i suprugi Davorki na bezgraničnoj podršci, osloncu i vremenu koje sam provodio radeći a ne sa njima. Njihovo podsticanje, vedrina i pažnja mi je uvek davala motiv za rad u trenucima kada bih klonuo. Supruzi Davorki sam zahvalan i na vremenu koje je provela slušajući moje ideje i dajući korisne savete.

Svim svojim prijateljima sam zahvalan na velikom razumevanju i savetima.

Andrija Jandrlić

Beograd, decembar 2012.

Sadržaj

1 Uvod

Poslovna aplikacija predstavlja softver koji organizacije koriste za rešavanje svojih poslovnih problema i zadataka. Centralne stavke poslovnih aplikacija čine: prikaz, upravljanje i čuvanje velike količine često veoma složenih podataka i podrška ili automatizacija poslovnih procesa nad tim podacima [1]. Razvoj poslovnih aplikacija predstavlja izazov jer su, po pravilu, veoma komplikovane i treba da ispune niz unapred definisanih zahteva. Složenost se ogleda u samoj implementaciji poslovnih pravila, validacije, izračunavanja i sl., kao i u ciljevima koje poslovne aplikacije treba da ispune: pouzdanost, distribuiranost, konkurentnost, sigurnost, korisničku prilagođenost (eng. user-friendly), itd. Prema [2], arhitekturu poslovnih aplikacija čine tri glavna sloja (tzv. troslojna arhitektura):

- *prezentacioni sloj* - opisuje korisnički interfejs i interakciju između korisnika i aplikacije. Može biti u obliku komandne linije, bogatog grafičkog korisničkog interfejsa (eng. rich-client) ili (X)HTML-a u slučaju veb aplikacija. Glavni zadatak korisničkog interfejsa je da prikaže informacije korisniku i da interpretira komande korisnika kao akcije nad domenom i izvorom podataka [1],
- *domenski (poslovni) sloj* - najvažniji deo poslovne aplikacije u kojem se opisuje poslovna logika, vrše se izračunavanja na osnovu unosa korisnika i sačuvanih podataka, obavlja se validacija, itd,
- *izvor podataka* - sadrži logiku za komunikaciju sa sistemom (sistemima), kao i podatke nad kojima se operiše. Za većinu poslovnih aplikacija glavni izvor podataka je relaciona baza podataka.

Osnovna ideja slojevite arhitekture je jasno logičko i fizičko razdvajanje uloga različitih modula aplikacije. Na taj način se dobija bolja iskorišćenost slojeva, minimalna međusobna zavisnost, laka zamena ili dopuna odgovarajućeg sloja, itd. Na primer, za jednu poslovnu veb aplikaciju je moguće definisati različite korisničke kanale: XHTML kroz veb pretraživač, grafički korisnički interfejs implementiran u Swing-u, MIDP (Mobile Information Device Profile) [3], SMS, itd. Iz tog razloga je veoma važno da implementacija poslovne logike, validacije i izračunavanja, pripadaju zasebnom sloju (domenskom), na koji se prezentacioni sloj oslanja, bez obzira na svoju implementaciju.

Razvojem Veb-a i Interneta, poslovne veb aplikacije postale su najzastupljeniji tip poslovnih aplikacija i sa njima veb pretraživači dominatni tip klijenta. U takvim aplikacijama, korisnici svoje podatke šalju kroz različite vrste formi. Sadržaj takvih formi je dinamičan: u slučaju inicijalizacije, podaci na formi se prikupljaju iz definisanog izvora podataka, dok se u slučaju slanja nevalidnih podataka isti podaci, zajedno sa porukama o greškama, vraćaju i ponovo prikazuju korisniku. Menjanje prirode formi iz statičke u dinamičku, u zavisnosti od njene složenosti, predstavlja zamoran zadatak podložan greškama, zbog njegove prirode.

1.1. Klasične veb forme

Forma definiše struktuirane podatke za razmenu između klijenta, sa jedne strane, i servera sa druge. U XHTML formama [4], tip podataka predstavlja skup parova u obliku ime-vrednost. Parovi oblika (ime, vrednost) su u potpunosti određeni od strane kontrola korisničkog interfejsa forme. Dodavanje nove kontrole u okviru forme takođe dodaje novi ime-vrednost par u skup podataka forme.

Pored standardnih kontrola korisničkog interfejsa, forma gotovo uvek sadrži neku vrstu klijentske validacije podataka. Takva validacija izvršava se tokom unosa podataka, u cilju obaveštavanja korisnika o nevalidnim podacima što je pre moguće, eliminišući nepotreban zahtev ka serveru [5]. Takođe, forme kao što su one za povraćaj poreza, izračunavanja iznosa kamata i plana otplata, računovodstva, itd. definišu različite vrste izračunavanja.

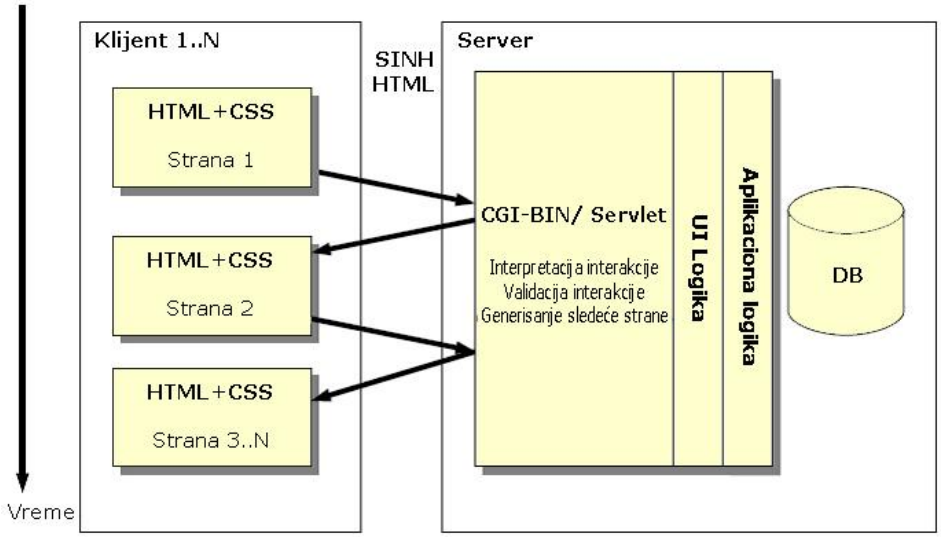
Osnovne poteškoće u ravoju XHTML formi predstavlja:

- oslanjanje na JavaScript jezik za izvršavanje validacije i izračunavanja, prikaz poruka o grešci, označavanje kontrole kao obavezne i dinamičko raspoređivanje komponenti na formi. Upravo ova zavisnost dovodi do toga da je rezultujuća forma veoma kompleksna, a kôd jako težak za pisanje i održavanje.
- Inicijalizacija podataka forme, bilo pri prvom dolasku na formu bilo nakon neuspešnog slanja podataka sa forme usled nevalidnih podataka.

Tipičan tok razvoja XHTML forme u poslovnim veb aplikacijama može se opisati na sledeći način: tim analitičara zadužen je za analizu specifičnog problema i odgovoran je za prikupljanje informacija o tome šta je potrebno da forma sadrži i kako bi forma okvirno trebalo da izgleda. Tim veb dizajnera zatim definiše **statičku** XHTML formu, zajedno sa CSS stilovima i usklađuje je sa zahtevima analitičara, odnosno klijenta čiji su zahtevi prikupljeni. U narednom koraku programeri treba da dobijenu formu pretvore u **dinamičku**, obično dodavanjem izraza za evaluaciju, serverskih etiketa (eng. tag) ili naredbi "razumljivih" od strane servera (npr. JSP tagova ili Velocity / Freemarker naredbi u slučaju J2EE [6] aplikacija), zajedno sa klijentskim JavaScript-om. Prilikom obrade ovakvih naredbi, server generiše odgovarajuće delove forme i rezultujući XHTML isporučuje korisniku. Ciljna forma bi trebalo da uključi sekvence za obradu i prikaz grešaka serverske validacije, koja nije mogla adekvatno da se opiše i izvrši na klijentskoj strani.

Model interakcije između klijenta i servera u veb aplikacijama kroz XHTML forme prikazan je na *Slika 1*. Veći deo obrade formi odvija se na serveru: inicijalizacija podataka formi, validacija, itd. dok je obrada na klijentskoj strani obično jednostavna i omogućava prostu interakciju kroz popunjavanje formi, aktivaciju linkova, slanje podataka sa forme i sl. Na svaki

zahtev server generiše novu XHTML stranu i nikakvo međustanje između strana (zahteva) se ne pamti na klijentu.



Slika 1. Model interakcije između klijenta i servera kroz XHTML forme

1.2. Validacija

Validacija podataka u poslovnim aplikacijama predstavlja proces kojim se proverava ispravnost podataka nad kojima se operiše i predstavlja važan deo poslovne logike. Validacija može biti *sintaksna* ili *semantička*. Sintaksna validacija obuhvata proste provere tipa da li je određen podatak zadat, prihvatljive dužine, određenog formata itd, dok semantička validacija obuhvata složenije provere koje često zahtevaju pristup izvoru podataka: jedinstvenost podataka, postojanje/nepostojanje određenih podataka, itd. U poslovnim aplikacijama sa troslojnom arhitekturom validacija može biti sastavni deo svakog sloja.

Relacione baze podataka, kao primarni izvor podataka u poslovnim aplikacijama, predstavljaju odlično mesto za implementaciju poslovne logike i validacije. Osnovne provere korektnosti podataka se postižu kroz definicije ograničenja poput primarnih i stranih ključeva, jedinstvenosti i dr, dok se složenije provere izvršavaju kroz tzv. uskladištene procedure (eng. stored procedure) i okidače (eng. trigger). Njihov kôd se najčešće većim delom generiše iz deklarativnih opisa pravila pomoću takozvanih CASE (Computer Aided Software Engineering) alata, poput Oracle Designer-a [7], uz mogućnost finog prilagođavanja koda. U slučaju narušene validnosti podataka, kao naznaka greške se može izbaciti izuzetak iz skupa predefinisanih izuzetaka od strane proizvođača relacione baze podataka kada su narušena osnovna ograničenja, ili izuzetak iz skupa prilagođenh izuzetaka specijalno dizajniranih u cilju naznačavanja narušavanja određenog poslovnog pravila. U oba slučaja moguće je razlučiti o kakvoj vrsti

greške je reč (npr. na osnovu koda greške) i korisniku prikazati lokalizovanu poruku o grešci. Lokalizovane poruke se, po pravilu, čuvaju u nekoj od tabela.

Osnovne prednosti ovakvog pristupa, osim metodološke korektnosti i zasnovanosti na konceptima sistema za upravljanje bazom podataka, su:

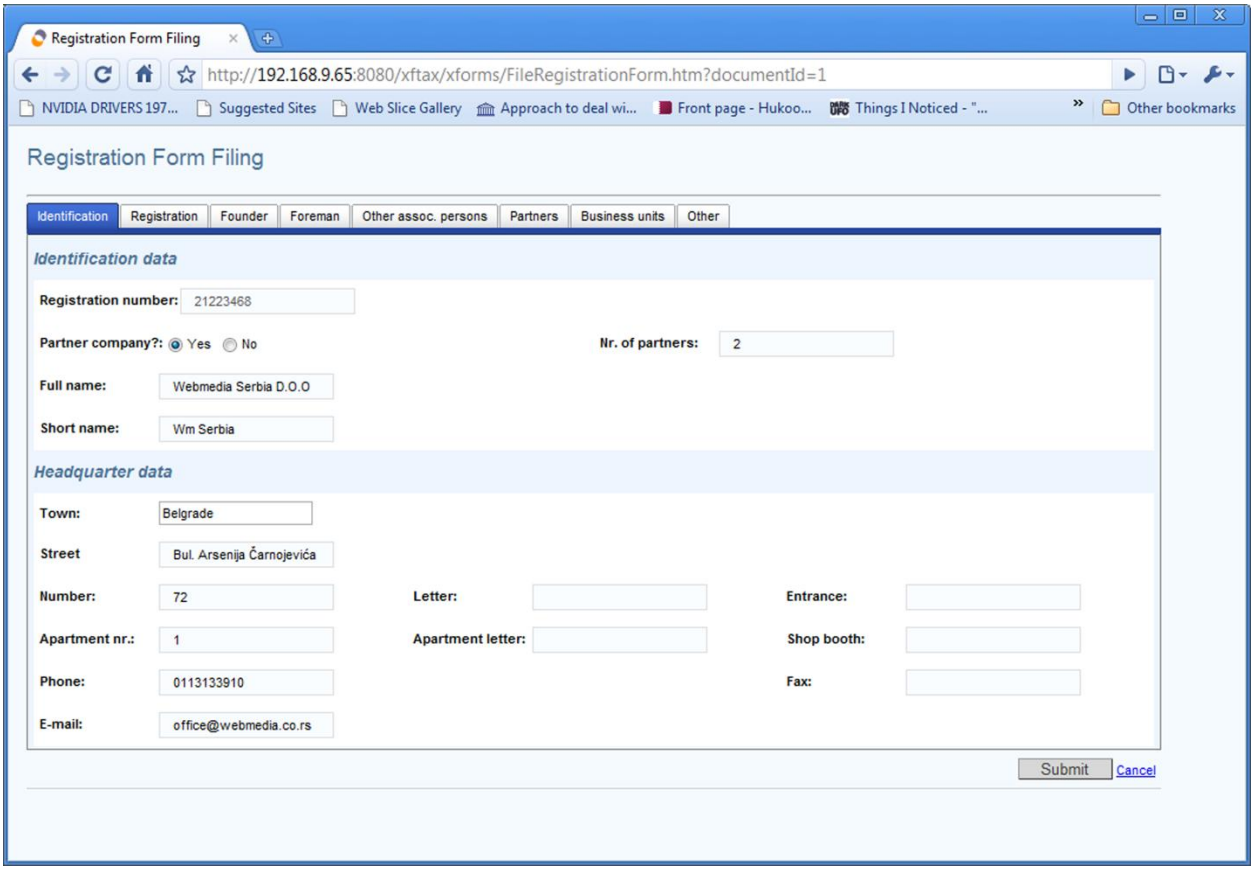
- plan za pristup podacima (eng. data access plan) koji se koristi od strane uskladištenih procedura se optimizuje i kešira za naredna izvršavanja,
- generisani kôd okidača i uskladištenih procedura nema grešaka i lak je za održavanje,
- lakše održavanje same aplikacije u globalu, jer je lakše promeniti kôd uskladištenih procedura i okidača nego SQL kôd smešten u poslovnom sloju aplikacije.

Glavni nedostatak predstavljaju prenosivost uskladištenih procedura među različitim bazama podataka (mada se u praksi promena relacione ili baze druge vrste jako retko događa) i činjenica da se kao argumenti uskladištenih procedura, u nekim implementacijama RDMSa, ne mogu proslediti objekti, već samo argumenti prostog tipa pa se često dešava da procedura ima previše argumenata.

U praksi se u poslovnim aplikacijama poslovna pravila i validaciona logika najčešće implementiraju u poslovnom sloju. Kada se identifikuju slučajevi u kojima su performanse od kritičnog značaja, izmeštaju se u uskladištene procedure i okidače. Iz tog razloga izloženi koncepti u ovoj tezi, kao i dizajn i implementacija validacione biblioteke, se odnose isključivo na srednji sloj poslovnih aplikacija.

1.3. Ciljevi i plan istraživanja

Poslovni sistemi koji su u velikoj meri zasnovani na formama, kao na primer sistem Poreske Uprave za registraciju poreskih obveznika, imaju često sklonost ka menjanju struktura formi zajedno sa poslovnom logikom, validacijom i izračunavanjima. U takvim slučajevima, automatizacija generisanja formi, zajedno sa generisanjem koda domenskog sloja, je od izuzetnog značaja. Forma za registrovanje poreskih obveznika, prikazana na *Slika 2*, sastoji se od osam celina sa preko dve stotine ulaznih polja, pa je lako zamisliti koliko je vremena i linija kôda potrebno da se u potpunosti napravi dinamičkom.



Slika 2. Forma za registraciju poreskih obveznika

Skorašnji trend u softverskom inžinjerstvu predstavlja Razvoj vođen modelom (eng. Model-Driven Development - MDD), koji je izazvao dosta kontraverzi sa svojim tvrdnjama i obećanjima, koja su napravljena (ali navodno ne i održana) tokom CASE ere [8]. Jedan od najpopularnijih domena u domenski-specifičnih MDD pristupa je Veb inženjerstvo, naročito dizajniranje korisničkog interfejsa zasnovano na modelu. Ispostavilo se da, za razliku od ostalih domena, Veb inženjerstvo ima svoje specifičnosti i da se razlikuje od opštih projekata softverskog inženjerstva [9], [10].

Prvi cilj ovog rada je analiza i specifikacija ključnih stavki za razvoj automatizovanog generatora formi i njegova implementacija. Glavni fokus rada je opisivanje automatskog popunjavanja formi, vođeno modelom zajedno sa lokalizovanim porukama o greškama prikupljenih tokom neuspele validacije podataka forme. Iako ideja o automatskom generisanju rezultujućih formi pomoću XSLT transformacija iz XML definicija formi nije nova, [11], [12], ona jeste nova u aplikacijama zasnovanih na XForms formama. Jasno razdvajanje modela od vizuelnih komponenti formi, čini XForms jezik idealnim kandidatom za razvijanje formi.

Za opis forme (model, navigacija, pravila i izračunavanja), može se koristiti jedan od sledećih načina (ili njihova kombinacija):

1. *AutoWeb System*, zasnovanim na HDM-lite jeziku (Hypermedia Design Model) – varijanti Veb jezika za dizajniranje HDM-a, predložen od strane Fraternalija i Paolinija [13], koji omogućava automatizaciju generisanja zahtevnijih Veb aplikacija po pitanju količine i složenosti podataka.

2. Prilagođenim *Skladištem Meta-podataka o dokumentima* koji sadrži meta-podatke poslovnih dokumenata sistema i njihova preslikavanja na strukturu sistema. Jedna od njegovih komponenti, model ekrana korisničkog interfejsa, opisuje polja forme, akcije i validaciona pravila, koristeći slogove relacione baze. Prezentacioni meta-podaci se generišu zajedno sa specijalizovanim tabelama za podatke, opisane konceptualnim modelom, na sličan način kao u [13]. U oba slučaja, informacije se koriste u toku izvršavanja i na osnovu njih se Veb forme generišu dinamički.

3. *Korišćenjem UML-a i ER notacije*, u istom maniru kakav je definisan od strane Izvršnog UML-a (eng. Executable UML) [14] i [15].

Generator formi treba da omogući lako dodavanje novih formi za različite korisničke kanale: veb forme, MIDP, itd. Takav generator treba da:

- smanji vreme kompletnog ciklusa pravljenja formi, od procesa analize poslovnih zahteva, statičkih formi, do dinamičkih formi na kraju,
- skrati vreme potrebno za implementaciju servisnog sloja za obradu podataka sa forme,
- generiše kôd koji garantuje konzistentnost i formalnu korektnost dizajna poslovnih formi i specifikacije poslovnih pravila.

Generator formi, zasnovan na formalnom opisu atributa formi, generiše komponente korisničkog interfejsa za veb kanale, API servisnog sloja, poslovnu logiku, objekte za pristup podacima (eng. Data Acees Objects – DAO) i osnovu za komponente formi (*Slika 3*). Generator bi trebalo da bude u stanju da izvršava sledeće servise:

- reagovanje na korisničke akcije vezane za dodavanja novih formi kao i promene, brisanje i arhiviranje postojećih formi,
- generisanje servisnog API-a za podršku korisničkom interfejsu za različite korisničke kanale.

Slika 3. Konceptualni model rezultata generatora formi

Komponente generisane od strane generatora formi, trebalo bi da izvršavaju sledeće akcije:

- reagovanje na korisničke akcije,
- validaciju formi i generisanje izveštaja o eventualnim greškama,
- procesuiranje poslate forme, u slučaju uspešne validacije.

Opis forme definisan je korišćenjem XML tehnologija, u ovom slučaju na osnovu XForms 1.1 specifikacije [16]. Sistem koji će koristiti generator formi treba da omogući prikupljanje podataka iz skupa predefinisanih izvora podataka. U cilju popunjavanja formi podacima iz izvora podataka, treba definisati transformacione skriptove. U te svrhe, za opis transformacija, moguće je koristiti standardizovane i dobro dokumentovane tehnologije (XSLT [17]). Na primer, u kontekstu povraćaja poreza, moguće je definisati poslovna pravila kako na serverskoj strani (ciljajući Java/Spring poslovni sloj i Spring/JSP UI komponente), tako i na klijentskoj. Sva pravila su sastavni deo poslovnog toka i mogu da se primenjuju nad modelom podataka forme i modelom podataka iz izvora podataka. Moguće je definisati sledeće akcije na klijentskoj strani:

- izračunavanje vrednosti međusobno zavisnih polja na formi na osnovu podataka,
- zahtevanje dodatnih podataka od servera na osnovu korisnikovog unosa,
- definisanje listi vrednosti za pregled (eng. look-up), npr. za izbor podataka iz padajuće liste predefinisanih vrednosti,
- definisanje kontrolnih tipova korisničkog interfejsa,
- omogućavanje/onemogućavanje polja na osnovu korisničkog unosa ili podataka sa servera,
- skrivanje/otkrivanje polja na osnovu korisničkog unosa ili podataka sa servera.

Sledeća pravila moguće je definisati na serverskoj strani:

- prepopulaciju modela podataka forme korišćenjem izvora podataka sa servera,
- ekstrahovanje podataka iz modela podataka forme i generisanje poziva sistemskih servisa na osnovu korisničkih akcija.

Narednim dijagramom prikazana je konceptualna interna struktura generatora:

Slika 4. Konceptualna interna struktura generatora formi

Ovako napravljen generator formi, u svojoj strukturi i upotrebi, ne treba da bude sličan bilo kojoj drugoj robustnoj i "teškoj" platformi kakva je Oracle Forms [18]. Sa gledišta korišćenja, platforma koja prethodi generatoru treba da omogući vizuelno pravljenje XForms formi na sledeći način:

- Pravljenje izgleda forme prostim prevlačenjem i postavljanjem komponenti korisničkog interfejsa (eng. drag & drop).
- Mogućnost uvoženja unapred definisane XML šeme, generisanje i pravljenje XML modela na osnovu nje, kao i povezivanje modela sa komponentama korisničkog interfejsa.
- Definisanje atributa stavki modela, kroz koje je moguće definisati vidljivost, dostupnost i druga različita ograničenja nad modelom.
- Definisanje izraza za izračunavanje.
- Definisanje različitih strategija za prikaz grešaka u slučaju kada je serverska validacija izvršena neuspešno nakon slanja podataka sa forme, pomoću obrađivača događaja ili na neki drugi način.

- Mogućnost postavljanja (eng. deploy) forme na server, omogućavajući na taj način brzu proveru korektnosti izgleda forme, izračunavanja i validacije.
- Definisanje validacije.
- Mogućnost arhiviranja formi.

Predloženi generator formi, testiran je za različite vrste formi. Najsveobuhvatnije i najreprezentativnije su bile forme za registraciju poreskih obveznika. Takve forme predstavljaju idealan primer za testiranje i reprezentaciju zato što su sastavljene od brojnih delova sa preko dve stotine ulaznih polja i sadrže puno klijentske validacione logike i pravila izračunavanja.

U tezi je takođe razmatran koncept procesa validacije i njegova automatizacija, kao jednog od najvažnijih delova poslovne logike. Cilj istraživanja je formiranje Semantičkog modela [19], dovoljno izražajnog i širokog za opisivanje i izvršavanje validacionih pravila sa jedne strane, i definisanje Domenski Specifičnog Jezika validacije (eng. Domain Specific Language – DSL) (videti [19]) kojim se popunjava Semantički model, sa druge. DSL, opisan u nastavku teze čine Unutrašnji DSL (eng. Internal DSL) napisan u programskom jeziku Java, kao i Spoljašnji DSL (eng. External DSL), sa sintaksom pogodnom za zapis validacionih pravila od strane sistemskih analitičara, koji se zatim parsiraju i prevode u Semantički model. Na taj način se postiže željena automatizacija.

2 XML i XML - zasnovani jezici

XML (eng. eXtensible Markup Language) jezik pruža programski neutralan način za opisivanje i razmenu podataka između različitih uređaja, sistema i aplikacija. Fleksibilan je i proširiv, daje mogućnost dodavanja novih informacija i dopušta postojanje različitih tipova podataka u okviru jednog dokumenta. Podaci u XML-u su predstavljeni u čitljivom i samoopisujućem formatu da bi se prilagodili raznolikim poslovnim potrebama. Servisno-orijentisana arhitektura (SOA), integracija poslovnih aplikacija (EAI), Web servisi, Atom i RSS izvori podataka itd. se oslanjaju na XML, zbog čega XML predstavlja jedan od važnijih, ako ne i najvažniji, format za čuvanje i razmenu informacija.

Osnovni gradivni blokovi XML-a su *elementi, atributi i imenovani prostori* (eng. namespace). Elementi se definišu pomoću etiketa. Sadržaj elementa može biti tekst, drugi elementi, ili oba. Dodatne informacije o elementu se mogu zadati pomoću atributa, zadavanjem imena i vrednosti atributa unutar etikete elementa. Imenovani prostori služe za izbegavanje dvosmislenosti i kolizija među imenima elemenata i atributa različitog porekla. Najčešće se asociraju sa elementima i atributima kroz tzv. prefiks, na taj način definišući njihovo kvalifikovano ime.

Sintaksa XML-a je jednostavna i vrlo stroga. Neka od sintaksnih pravila su da XML dokument mora da ima jedinstveni koreni elementi, takav da su svi ostali elementi njegova deca (unutar njega); nazivi elemenata i atributa su osetljivi na veličinu slova; elementi moraju biti ispravno ugnježdeni i moraju imati zatvorene etikete. Za XML dokument, koji je u skladu sa XML sintaksom, se kaže da je dobro formiran.

Struktura i sadržaj jednog XML dokumenta se mogu definisati pomoću DTD-a (Document Type Definition) ili XML šeme. Pomoću njih se može specifirati koji elementi (i atributi) se mogu pojaviti u dokumentu, u kakvom redosledu, kakva su njihova ograničenja i drugo. Za XML dokument, koji je u skladu sa pridruženim DTDom ili XML šemom, se kaže da je valjan.

Najčešće operacije koje se izvršavaju nad XML dokumentom su izdvajanje podskupa podataka smeštenih u okviru njega i transformacija celog ili jednog dela dokumenta u XHTML ili drugi XML dokument. Za svrhe pretrage napravljen je veliki broj upitnih jezika, među kojima je najvažniji XQuery, dok je za transformacije XML dokumenta definisan XSLT jezik.

2.1. XML šema

XML šema [20] predstavlja formalizaciju ograničenja nad strukturom i sadržajem XML dokumenata izraženih u formi različitih pravila. Osnovna namena XML šeme je da deklariše elemente i definiše njihove tipove.

Primer. Neka je plan otplate kredita (eng. Payment Schedule) definisan jedinstvenim referentnim brojem, procentom naknade za kašnjenje isplate (latePaymentPenaltyPercentage), i tipom koji može biti kredit za prava za dozvolu gradnje (eng. Building Rights) ili prava za iznajmljivanje zemljišta (eng. Land Rental). Tip kredita definisan je datumom početka otplate (feeCalculationDate), trajanjem izraženom u godinama (durationInYears), iznosom zasnovanim na porezu (taxBasedAmount) i godišnjom naknadom (yearlyFee). Za atribute modela važe sledeća ograničenja:

- referentni broj je string dužine 13 karaktera, čiji su svi karekteri numerički,
- procenat naknade za kašnjenje je realan broj u intervalu [0.01, 999999.99], zadat na dve decimale najviše,
- trajanje otplate je ceo broj u intervalu [1, 100] godina,
- godišnja naknada je realan broj u intervalu [0.01, 999999.99], zadat na dve decimale najviše,
- iznos zasnovan na porezu je realan broj u intervalu [0.01, 999999.99], zadat na dve decimale najviše.

XML dokument kojim bi se predstavio jedan plan otplate, mogao bi da bude definisan u skladu sa sledećom XML šemom:

```
<xsd:schema xmlns:xsd=http://www.w3.org/2001/XMLSchema
            xmlns:tns="http://pras.org/finance"
            targetNamespace="http://pras.org/finance"
            elementFormDefault="qualified">

   <xsd:element name="PaymentSchedule" type="tns:PaymentScheduleStructure" />

   <xsd:complexType name="PaymentScheduleStructure">
            <xsd:sequence>
                  <xsd:element name="referenceNumber"
                               type="tns:ReferenceNumberType" />
                  <xsd:element name="latePaymentPenaltyPercentage"
                               type="tns:AmountType" />
                  <xsd:choice>
                        <xsd:element name="buildingRight"
                                     type="tns:YearlyPayableStructure" />
                        <xsd:element name="landRental"
                                     type="tns:YearlyPayableStructure" />
                  </xsd:choice>
            </xsd:sequence>
      </xsd:complexType>

      <xsd:complexType name="YearlyPayableStructure">
            <xsd:sequence>
               <xsd:element name="feeCalculationStartDate" type="xsd:date" />
               <xsd:element name="durationInYears">
                  <xsd:simpleType>
                        <xsd:restriction base="xsd:positiveInteger">
                              <xsd:maxInclusive value="100" />
                        </xsd:restriction>
                  </xsd:simpleType>
```

```
            </xsd:element>
            <xsd:element name="taxBaseAmount" type="tns:AmountType" />
            <xsd:element name="yearlyFee" type="tns:AmountType" />
        </xsd:sequence>
    </xsd:complexType>

    <xsd:simpleType name="ReferenceNumberType">
        <xsd:restriction base="xsd:string">
            <xsd:length value="13" />
            <xsd:pattern value="([0-9])*" />
        </xsd:restriction>
    </xsd:simpleType>

    <xsd:simpleType name="AmountType">
        <xsd:restriction base="xsd:decimal">
            <xsd:fractionDigits value="2" />
            <xsd:totalDigits value="8" />
            <xsd:minInclusive value="0.01" />
            <xsd:maxInclusive value="999999.99" />
        </xsd:restriction>
    </xsd:simpleType>

</xsd:schema>
```

Iz primera se vidi da XML šema predstavlja jedan XML dokument, čiji elementi dele zajednički, predefinisani imenovani prostor (http://www.w3.org/2001/XMLSchema).

Svaki element i atribut asociran je sa *tipom*, koji definiše prirodu podataka koji element ili atribut sme da sadrži. Elementi se deklarišu pomoću `xsd:element` etikete, u kojoj je neophodno navesti naziv i tip elementa. Atributi se na sličan način deklarišu pomoću `xsd:attribute` etikete. Deklaracije elemenata i atributa može biti globalna ili lokalna. Ako je definicija globalna, tada je smeštena u gornjem nivou kao dete `xsd:schema` elementa. Globalna deklaracija mora da sadrži imena koja su jedinstvena unutar cele šeme. Po imenu se element ili atribut može upotrebiti u definiciji kompleksnog tipa. Sa druge strane, lokalna deklaracija se pojavljuje samo unutar definicije kompleksnih tipova i kao takva ne može se ponovo iskoristiti na drugom mestu. U prethodnom primeru, element *PaymentSchedule* je globalan, dok je npr. element *durationInYears* lokalan.

2.1.1. Tipovi

Tip elementa se koristi da opiše i proveri valjanost elementa i vrednosti njegovih atributa. XML šemom definisana su dva tipa podataka: prost i složen. Prost tip opisuje elemente koji imaju tekstualni sadržaj i koji ne sadrže druge elemente, niti atribute. U prethodnom primeru *yearlyFee* je jedan od takvih elemenata. Složeni tip opisuje elemente koji mogu imati decu-elemente i/ili atribute. Slično deklaraciji elemenata i atributa, deklaracija tipa takođe može biti globalna ili lokalna. Globalni tipovi su imenovani, što omogućava da im se pristupa po imenu, dok su lokalni tipovi neimenovani.

XML šemom zadat je skup predefinisanih prostih tipova podataka. Predefinisani tipovi su dizajnirani tako da budu nezavisni od proizvođača i specifičnog programskog jezika. Neki od prostih ugrađenih tipova su: `xsd:string`, `xsd:double`, `xsd:long`, `xsd:int`, `xsd:date`, `xsd:boolean`, itd.

XML šemom definisana su dva specijalna elementa koja omogućavaju nasleđivanje tipova:

- `xsd:restriction` element omogućava da se već postojeći tip napravi restriktivnijim. U prethodnom primeru *ReferenceNumberType* tip definisan je na taj način,
- `xsd:extension` element omogućava proširivanje već postojećeg kompleksnog tipa. Prosti tipovi se ne mogu proširiti.

Dozvoljene vrednosti svakog od prostih tipova mogu se dodefinisati nametanjem jednog ili više ograničenja (eng. facets). U prethodnom primeru je tako tip *AmountType* ograničen tako da njegove vrednosti mogu imati najviše dve decimale, maksimalno 12 cifara i nalaze se u zatvorenom intervalu [0.01, 999999.99]. XML šema, u zavisnosti od tipa podatka, definiše sledeća ograničenja:

- nabrojivu listu dozvoljenih vrednosti pomoću `xsd:enumeration` elementa,
- tačnu dužinu stringa korišćenjem `xsd:length` elementa, ili opseg dužine pomoću `xsd:minLength` i `xsd:maxLength` elemenata,
- interval za vrednosti numeričkog tipa, pomoću `xsd:minInclusive` / `xsd:minExclusive` i `xsd:maxInclusive` / `xsd:maxExclusive` elemenata,
- ukupan broj značajnih cifara i broj decimalnih mesta korišćenjem `xsd:totalDigits` i `xsd:fractionDigits` elemenata,
- obrasce formatiranja (eng. formatting patterns) pomoću `xsd:pattern` elementa. U prethodnom primeru za tip *ReferenceNumberType* uvedeno je ograničenje oblika `<xsd:pattern value="([0-9]*)" />`, koje označava da svi karakteri vrednosti tog tipa moraju biti cifre. XML šema ima svoju sintaksu za regularne izraze, koja je najsličnija onoj koju ima Perl.

Složeni tip se koristi za elemente koji mogu imati decu elemente i/ili atribute. Svaki složeni tip može imati jedan od navedenih tipova sadržaja:

- prost - predstavlja tip čiji elementi mogu imati samo tekst kao sadržaj,
- sadržaj sastavljen samo od elemenata – tip čiji elementi mogu imati samo decu elemente, bez teksta,
- izmešani – sadrži i tekst i decu elemente,
- prazan – tip čiji elementi nemaju niti decu, niti tekst.

Osnovna razlika između prostog tipa i složenog tipa sa prostim sadržajem je ta što složeni tip može sadržati atribute.

Svaki deklarisan element se podrazumevano može pojaviti tačno jedanput u XML dokumentu. Ova osobina se može promeniti upotrebom `xsd:minOccurs` and `xsd:maxOccurs` atributa u deklaraciji elementa. Redosled i pojavljivanje elemenata definišu se upotrebom jednog od `xsd:sequence`, `xsd:choice` ili `xsd:all` elementa. Prvi označava da se elementi u dokumentu moraju zadati onim redosledom u kojem su deklarisani, drugi označava da tačno jedan element od ponuđenih može biti zadat, dok treći označava da se elementi mogu pojaviti u bilo kom redosledu.

2.1.2. Imenovani prostori

Pravljenjem šeme formira se i prilagođeni rečnik, za koji može da se definiše imenovani prostor. Takav imenovani prostor naziva se ciljni (eng. target namespace). Ciljni imenovani prostor definiše se zadavanjem `targetNamespace` atributa `xsd:schema` elementa, što je ilustrovano prethodnim primerom. Definisanjem ciljnog imenovanog prostora svi globalni tipovi i elementi koji su deklarisani u šemi, postaju deo tog imenovanog prostora. Ova podrazumevana osobina se može promeniti postavljanjem vrednosti `elementFormDefault` atributa na “qualified”, čime i lokalni elementi postaju deo ciljnog imenovanog prostora. Što je `elementFormDefault` atribut za elemente, to je `attributeFormDefault` za atribute. Podrazumevana vrednost “unqualified” označava da su samo globalni atributi deo ciljnog imenovanog prostora, dok vrednost “qualified” obuhvata i lokalne. Za određivanje toga da li je specifični element deo ciljnog imenovanog prostora ili ne, može se koristiti `form` atribut `xsd:element` elementa, sa vrednostima “qualified” i “unqualified”.

Jedna bitna osobina ciljnih imenovanih prostora je da ih generatori koda, koji na osnovu XML šeme generišu klase ciljnog jezika, obično koriste kao naziv paketa u koji se smeštaju generisane klase.

2.2. XPath – XML Path jezik

XPath 2.0 [21] je jezik za procesiranje vrednosti koje su u skladu sa XQuery/XPath modelom podataka opisanom u nastavku rada. Osnovna namena XPath-a je dohvatanje čvorova XML stabla, korišćenjem takozvanih “putanja” za navigaciju kroz hijerarhijsku strukturu XML dokumenta.

Putanje koje se koriste u XPath-u sastoje se od jednog ili više “koraka“ razdvojenih simbolom “/” ili “//“. Svaki korak pre završnog koraka daje sekvencu čvorova koji se koriste kao kontekst-čvorovi u koraku koji sledi. Svaki korak se ponavlja po jednom za svaki kontekstni čvor koji je nastao kao proizvod prethodnog koraka. Vrednost izraza je sekvenca stavki koja sadrži ili čvorove ili atomske vrednosti (nikad oba).

Početak XPath izraza definiše početnu tačku putanje. Početni simbol "/" označava da putanja počinje od korena dokumenta, dok simbol "//" označava početak sekvence od korena dokumenta i svih njegovih potomaka.

Svaki naredni korak može biti po nekoj od osa ili filter-izraza. Korak po osi se sastoji od tri dela: opcione **ose** koja predstavlja pravac u kojem treba izvršiti pomeranje kroz XML dokument ili fragment, **testa čvora** koji definiše kriterijum koji se koristi za selektovanje čvorova i nula ili više **predikata** koji filtriraju sekvencu koja je rezultat prethodnog koraka. Rezultat koraka po osi predstavlja sekvencu čvorova. Svakom od čvorova se dodeljuje kontekstna pozicija koja odgovara poziciji u sekvenci. Kontekstna pozicija omogućava dohvatanje svakog od čvorova po poziciji.

XPath definiše 13 osa, među kojima su najčešće korišćene:

- child – za navigaciju čvorova koji su deca tekućeg kontekst čvora, ujedno predstavlja i podrazumevanu osu ako se ni jedna druga ne navede,
- attribute - za navigaciju kroz atribute tekućeg kontekst čvora,
- parent – za dohvatanje roditelj čvora tekućeg kontekst čvora,
- self – za navigaciju do samog kontekst čvora.

Testiranje čvora je moguće na jedan od tri načina: po imenu (kvalifikovanom ili nekvalifikovanom), tipu čvora i tzv. "džoker" testiranje.

Testiranje po imenu predstavlja najčešći tip testa kojim se selektuju čvorovi sa datim imenom. Imena su osetljiva na velika i mala slova. Izraz poput `/x/y/@z` počinje od korenog čvora dokumenta, nastavlja kroz čvor `x`, zatim kroz čvor `y`, koji je dete čvor čvora `x`, do atribut-čvora sa nazivom "z".

Testiranje po tipu čvora moguće je na jedan od sledećih načina:

- node() – odgovara svakom čvoru,
- text() – odgovara svakom tekstualnom čvoru,
- comment() – odgovara svakom komentar-čvoru,
- processing-instruction() – odgovara svakoj procesorskoj instrukciji,
- element() – odgovara svakom elementu,
- attribute() – odgovara svakom atributu,
- document-node() – odgovara svakom dokument-čvoru.

2.2.1. XQuery/XPath model podataka

Model podataka XQuery 1.0 i XPath 2.0 jezika zasnovan je na XML Infoset-u - apstraktnom skupu podataka XML dokumenta. Njime se definišu sve dozvoljene vrednosti izraza u XQuery, XPath i XSLT jezicima. Svi navedeni jezici su zatvoreni u odnosu na pomenuti model. Za razliku od Infoset-a, model je proširen:

- podrškom za XML šemu,
- mogućnošću prezentacije kolekcije dokumenata i složenih vrednosti,
- podrškom za tipizirane atomske tipove,
- podrškom za uređene, heterogene sekvence.

Kao što je već pomenuto, svaka instanca modela je **sekvenca**. Sekvenca je uređena kolekcija nula ili više stavki. Sekvence ne mogu biti ugnježdene.

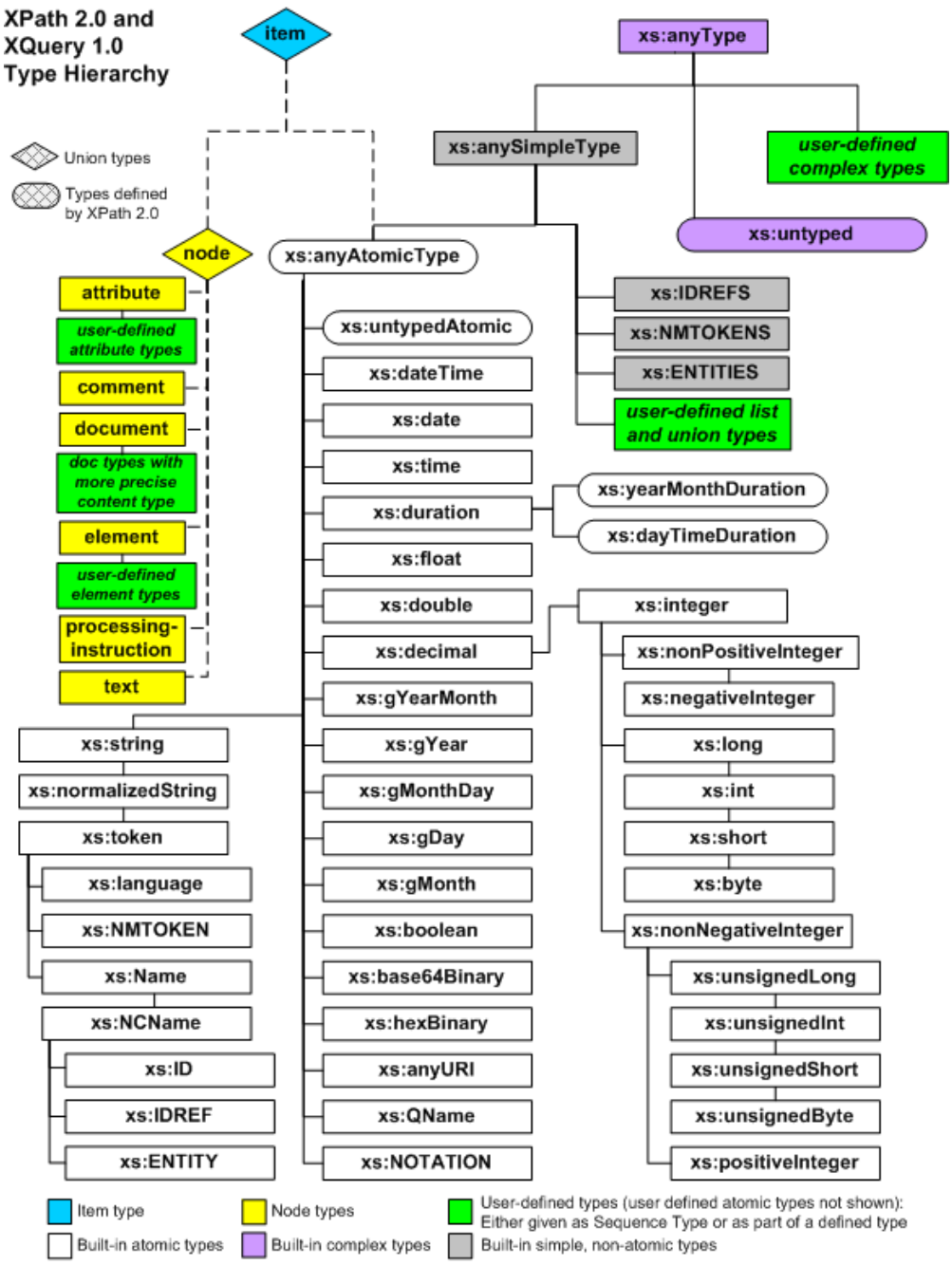
Stavka je ili čvor ili atomska vrednost. Čvor može biti tipa: attribute(), comment(), document(), element(), namespace(), processing-instruction() i text(). Svi navedeni tipovi su potomci apstraktnog tipa node(). Čvorovi formiraju stablo koje se sastoji od korena i čvorova koji su direktno ili indirektno dostižni iz korena. Svaki čvor pripada tačno jednom stablu i svako stablo sadrži tačno jedan čvor.

Drvo čiji je koreni čvor dokument naziva se **dokument**.

Drvo čiji koreni čvor nije dokument naziva se **fragment**.

Atomska vrednost je jedna od vrednosti definisane odgovarajućim atomskim tipom (takozvani prostor vrednosti tipa). Atomski tip je primitivni prosti tip ili tip definisan restrikcijom iz drugog atomskog tipa. U pomenutom modelu postoje 23 primitivna prosta tipa, od kojih je 19 preuzeto iz XML šeme, a dodato je pet novih tipova: xs:untyped, xs:untypedAtomic, xs:anyAtomicType, xs:dayTimeDuration i xs:yearMonthDuration.

Model podržava dobro-formirane XML dokumente u skladu sa prostorima imena 1.0 i 1.1, DTD validne dokumente i dokumente validne u skladu sa nekom XML šemom.



Slika 5. Hijerarhija tipova XQuery 1.0 / XPath 2.0 modela podataka. Izvor: http://www.w3.org/TR/xpath-datamodel/

2.2.2. Operatori

XPath 2.0 nudi veliki broj operatora koji se mogu koristiti u izrazima. Neki od njih su:

- matematički +, -, *, div, idiv, mod,
- relacioni =, !=, < (<), > (>), <= (≤), >= (≥),

- logički - and, or, not,
- uslovni if-then-else, choose-when-otherwise,
- iterator kroz sekvence – for operator,
- kvantifikatori – some i any,
- operatori nad tipovima – instance of, castable as, cast as i treat as.

2.3. XSLT

XSLT (Extensible Stylesheet for Transformations) (videti [22]) predstavlja fleksibilan i moćan jezik za transformisanje XML dokumenata u isti, ili neki drugi tip dokumenta, npr. HTML, PDF, SVG, Java kôd, tekstualni fajl, itd. Tekuća verzija XSLT-a 2.0 predstavlja W3C preporuku (neku vrstu standarda). XSLT dokumentom se definiše transformacija, koja se zatim izvršava od strane XSLT procesora.

Neke od karakteristika XSLT jezika su:

- transformacije se definišu u obliku XML dokumenta pomoću tzv. pravila obrazaca (eng. template rules). Pravilo se primenjuje na podstablo dokumenta koji odgovara obrascu, za koje je pravilo definisano. Obrasci poklapanja zadaju se pomoću XPath jezika. Pravila obrazaca sastoje se iz tri dela: *obrasca* koji se koristi za određivanje čvorova nad kojima se pravilo primenjuje, opcionog *skupa parametara* i *konstruktora sekvenci* koji za rezultat daje sekvence čvorova ili atomičkih vrednosti.
- Dizajniran je tako da u njemu ne postoje bočni efekti: više pravila transformacija može da se primeni simultano, bez bojazni da će jedno pravilo uticati na vrednosti promenljivih od kojih zavisi drugo pravilo. Ključna stvar je u tome što promenljivoj u XSLT-u ne može biti promenjena vrednost koja joj je prvobitno dodeljena. Ova karakteristika nastala je pod uticajem funkcionalnih programskih jezika.
- Umesto ciklusa, XSLT koristi iteraciju i rekurziju.
- Počev od verzije 2.0, XSLT ima podršku za XML šemu, što znači da su svi čvorovi i promenljive tipizirane.
- Izgrađen je nad modelom podataka XML šeme, kao i nad XPath 2.0 modelom podataka.

U zavisnosti od toga da li XSLT procesor treba da obradi dokument koji sadrži prilagođene tipove definisane od strane neke XML šeme, razlikuju se dva tipa procesora: procesori svesni šeme (eng. schema-aware) i osnovni procesori. Procesor svestan šeme može da validira vrednosti čvorova tokom procesiranja, u skladu sa definisanom šemom.

U poglavlju 5., navedena je i detaljno razmatrana XSLT transformacija koja dinamički na formu dodaje greške nastale prilikom validacije.

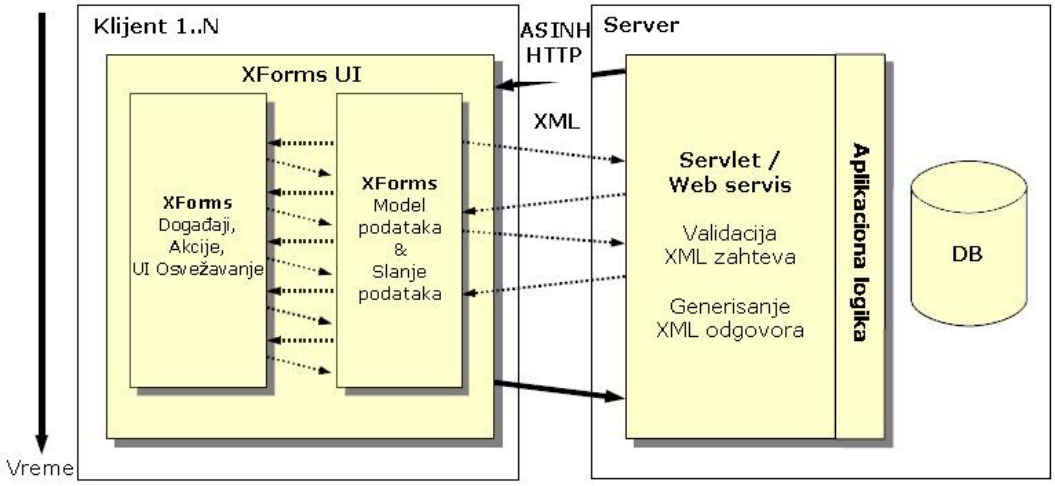
3 XForms

XForms jezik za označavanje je XML-zasnovani jezik za razvijanje veb formi. U odnosu na XHTML forme, XForms obezbeđuje viši nivo u samom pristupu, razdvajajući svrhu forme od prezentacije i koristi deklarativni stil za opis operacija u aplikacijama zasnovanim na formama [23]. XForms omogućava laku ponovnu iskoristivost komponenti forme, jaku validaciju tipova podataka i eliminiše nepotrebno kontaktiranje servera u slučaju nekorektnih podataka. Pruža nezavisnost forme od same platforme na kojoj se forma predstavlja - za jednu definiciju forme moguće je definisati tzv. XForms procesore za različite tipove uređaja poput PDA, mobilnih telefona, itd. Takođe, XForms-om se znatno smanjuje potreba za JavaScript-om.

Prema [16], [24] i [25], XForms jezikom forma je podeljena na tri glavna sloja:

- *Instanca*, sadrži samo podatke sa forme, bez bilo kakvog opisa prezentacionog dela ili validacije. Koristi se za čuvanje i slanje podataka između klijenta i servera. Definisana je ubliku XML segmenta unutar dokumenta koji ga sadrži, ili kao spoljašnji XML entitet na koji se referiše pomoću XLink-a [26].
- *Model*, sadrži različite instance i pored njih definiše validaciju, ograničenja, izračunavanja i neophodne meta-podatke kojima se definiše komunikacija sa Veb serverom. Validacija se izvršava kroz pridruženu XML šemu. Ograničenja i izračunavanja se definišu pomoću XPath 1.0 jezika.
- *Korisnički interfejs* sastavljen od različitih tipova kontrola, kojima se definiše izgled i odnos (povezivanje) sa instancama modela. Ovaj sloj takođe uključuje i napredne funkcionalnosti, kao što su ponavljanje konstrukcija, dinamičko vezivanje kontrola korisničkog interfejsa, itd.

U odnosu na HTML, veći deo obrade korisničkog interfejsa je prebačen na klijentsku stranu (*Slika 6*), što poboljšava propusnu moć aplikacije i daje bolju iskoristivost. Takođe, zbog činjenice da su podaci razdvojeni od korisničkog interfejsa, lakše je napraviti modularni softver koji je kao takav lakši za održavanje. XForms model interakcije razdvaja logiku korisničkog interfejsa od aplikativne logike, i može biti izvršen u celosti na klijentu od strane XForms procesora. XForms jezik omogućava navođenje asihnrone komunikacije sa serverom, obezbeđujući iskoristivost korisničkog interfejsa dok se sam asinhroni zahtev obrađuje.



Slika 6. XForms model interakcije

XForms ne predstavlja samostalni tip dokumenta, već je dizajniran da bude sastavni deo drugih XML-zasnovanih jezika, kao što su XHTML, Synchronised Multimedia Integration Language (SMIL) [27] ili Scalable Vector Graphics (SVG) [28].

3.1. Model

Model u XForms specifikaciji opisuje šta forma predstavlja i koje podatke sadrži. Definiše se `<model>` elementom, i sadrži podatke instance modela, zajedno sa pridruženim ograničenjima. Ograničenja nad XML instancom mogu biti statička, izražena kroz XML šemu, pogodna za definisanje onda kada se pravila neće menjati tokom interakcije korisnika. Sa druge strane, model može da uključi i tzv. *atribute stavki modela* (eng. Model Item Properties) – dinamička ograničenja napisana u XPath-u, evaulirana od strane XForms procesora. Model element se obično navodi unutar `<head>` elementa XHTML dokumenta (dela dokumenta koji nema vizuelnu reprezentaciju).

Instanca modela definiše se pomoću `<instance>` elementa, kao dete `<model>` elementa. Podaci u modelu se mogu iskoristiti za definisanje inicijalnih podataka forme, a takođe mogu biti promenjeni kroz kontrole korisničkog interfejsa sa kojima su povezani.

Sledećom sekvencom je definisan model kojim se opisuje registraciona forma za registraciju poreskih obveznika:

```
<xf:model id="registrationForm">
  <xf:instance xmlns="" id="registrationForm-instance">
    <registrationForm>
      <registrationNumber />
      <fullName />
      <shortName />
      <partnerCompany />
```

```
      <numberOfPartners />
      <address>
        <town>
          <code />
          <name />
        </town>
        <streetName />
        <streetNumber />
        <streetLetter />
        <entrance />
        <homeNumber />
        <homeLetter />
        <shopBooth />
        <contact>
          <phone />
          <fax />
          <email />
        </contact>
      </address>
    </registrationForm>
  </xf:instance>
</xf:model>
```

Atributi stavki modela zadaju se u okviru **<bind>** elementa. Za prethodni primer ima smisla definisati sledeća ograničenja:

- uneti registracioni broj mora da bude pozitivan ceo broj dužine 8, obavezan za unos

```
<xf:bind nodeset="/registrationForm/registrationNumber"
required="true()" id="registrationNumber" constraint=". castable as
xs:positiveInteger and string-length(.) = 8" />
```

- ukoliko je izabrano da je kompanija, koja se registruje partnerska, u tom slučaju neophodno je precizirati broj partnera, u suprotnom ne.

```
<xf:bind nodeset="/registrationForm/numberOfPartners"
required="/registrationForm/partnerCompany='true'"
readonly="/registrationForm/partnerCompany='false'"
id="numberOfPartners"
constraint="(/registrationForm/partnerCompany='true' and string-
length(.) > 0 and . castable as xs:positiveInteger) or
(not(/registrationForm/partnerCompany='true') and string-length(.) = 0)"
/>
```

Atribut ”constraint” predstavlja predikat koji treba da je zadovoljen da bi se čvor instance modela, sa kojim je ograničenje povezano, smatrao validnim [16]. Vrednost ovog atributa je XPath logički izraz

Ostali atributi <bind> elementa su:

- **relevant** – predstavlja XPath logički izraz, sa podrazumevanom vrednošću "true", kojim se određuje da li je čvor dostupan ili ne.
- **required** – odeređuje da li je vrednost određenog elementa obavezna ili ne. Isto ograničenje može biti definisano XML šemom, ali je kao takvo statičko i uvek na snazi. Atributom stavki modela može se definisati obaveznost unosa određenog polja na osnovu unosa korisnika (dinamički).
- **calculate** – predstavlja XPath izraz koji se evaulira i dodeljuje čvoru, obično na osnovu vrednosti koje su unete od strane korisnika.
- **readonly** – korisnik može ili ne može da promeni vrednost atributa instance onda kada je određeni uslov ispunjen / neispunjen. Ovim atributom se takođe zadaje XPath logički izraz.

XForms model takođe sadrži jedan ili više **<submission>** elemenata za slanje podataka, kojim se definišu podaci o tome: šta se šalje, gde se šalje i kako se šalje. Anatomija <submission> elementa je sledeća:

- **binding** atributom definiše se koji deo modela se šalje prilikom slanja podataka. Ukoliko se ovaj element izostavi, podrazumevano se šalje u celosti prva instanca definisana u modelu.
- **action** atributom se definiše lokacija na koju se šalju podaci. Action atribut je ekvivalent action atributu HTML forme.
- **method** atributom definiše se metoda kojom se šalju podaci. Moguće vrednosti su: post, put, get, multipart-post, form-data-post, urlencoded-post.
- **mode** atributom određuje se da li se komunikacija između klijenta i servera odvija sinhrono (synchronous) ili asinhrono (asynchronous).
- **encoding** atributom se definiše na koji način se koduju podaci sa forme pre slanja
- **replace** atribut omogućava veb pretraživaču i serveru koji "umeju" da procesiraju XForms inteligentno delimično slanje podataka i osvežavanje podataka. Moguće vrednosti ovog atributa su:
    - **all** - odgovor servera zamenjuje celu tekuću stranu
    - **instance** - odgovor servera zamenjuje samo podatke određene instance
    - **none** - odgovor servera se tretira samo kao potvrda

3.2. Tipovi podataka

XForms specifikacijom podržani su svi tipovi XML šeme osim: duration, ENTITY, ENTITIES, NOTATION.

XForms definiše takođe nekoliko tipova u svom prostoru imena: listItem, listItems, dayTimeDuration, yearMonthDuration, email, credit-card.

XForms u izrazima izračunavanja i povezivanja koristi XPath tipove podataka: boolean, string, numeric i node-set. Tip "object" se koristi za označavanje tipa povratne vrednosti i može biti bilo koji od četiri prethodno nabrojana tipa.

3.3. Kontrole korisničkog interfejsa

XForms kontrole korisničkog interfejsa definišu izgled modela. Vrednosti koje se prikazuju u njima vezane su za čvorove XForms modela. Kontrola korisničkog interfejsa se povezuje sa modelom na jedan od sledeća dva načina [29]:

- **bind** atributom čija vrednost predstavlja identifikator prethodno definisanog "bind" elementa
  ```
  <xf:input bind="registrationNumber"  />
  ```
- **ref** atributom čija vrednost predstavlja XPath putanju do čvora sa kojim se kontrola povezuje
  ```
  <xf:input ref="/registrationForm/registrationNumber"  />
  ```

Način na koji se kontrola prikazuje zavisi od tipa podataka i atributa stavki čvora modela sa kojim je kontrola povezana. Na primer, ista `<input>` kontrola može biti prikazana kao tekstualno polje ili kao kontrola za izbor datuma u slučaju da je tip podataka u čvoru `xsd:date`. Ako je kontrola povezana sa nevalidnim čvorom, može se prikazati poruka o grešci. Takođe, u slučaju da je čvor označen samo za čitanje (eng. read-only), vrednost u kontroli se ne može izmeniti.

Veza između kontrole i modela je više-na-jedan, i važi jedno od sledećih pravila:

- delovi modela ne moraju da imaju vizuelnu reprezentaciju,
- delovi modela mogu da se iskoriste samo za izlaz korišćenjem "output" elementa,
- delovi modela se mogu izračunati, ali ne i direktno uneti,
- delovi modela mogu biti pridruženi višestrukim kontrolama.

Kontrole korisničkog interfejsa su jako slične XHTML kontrolama unutar formi i uključuju: tekstualna polja, padajuće liste, radio dugmad, dugmad, itd. Unutar svake od kontrola moguće je navesti dodatne elemente koji određuju način na koje se kontrole prikazuju:

- *label* predstavlja labelu obaveznu za sve tipove kontrola,
- *alert* predstavlja poruku o grešci ukoliko uneta vrednost u polju nije validna,
- *hint* je poruka koja je nagoveštaj korisniku šta određena kontrola predstavlja,

- *help* ukoliko je naveden, prikazuje se korisniku u obliku ikone koja sadrži znak pitanja. Kada korisnik pritisne mišem pomenutu ikonu, prikazuje mu se odgovarajuća definisana poruka. Poruka može biti definisana u obliku XHTML-a čime se dodatno može vizuelno dočarati,
- *incremental* ukoliko je naveden definiše slanje događaja na svaku promenu vrednosti kontrole.

Prethodni model, kojim se definiše registraciona forma, bi vizuelno mogao da se opiše na sledeći način:

```
<xf:group ref="/registrationForm">
  <xf:input ref="registrationNumber">
    <xf:label>Registration number:</xf:label>
    <xf:alert>Please specify valid 8-digit registration number</xf:alert>
  </xf:input>

  <xf:select1 ref="partnerCompany" appearance="full">
    <xf:label>Partner company?:</xf:label>
    <xf:alert>Mandatory field</xf:alert>
    <xf:item>
      <xf:label>Yes</xf:label>
      <xf:value>true</xf:value>
    </xf:item>
    <xf:item>
      <xf:label>No</xf:label>
      <xf:value>false</xf:value>
    </xf:item>
  </xf:select1>

  <xf:input ref="numberOfPartners">
    <xf:label>Nr. of partners:</xf:label>
    <xf:alert>Number of partners is mandatory when company is partner
              company
    </xf:alert>
  </xf:input>
  ...
<xf:group>
```

Definisana `xf:group` kontrola ne predstavlja kontrolu korisničkog interfejsa, već je zgodna za grupisanje kontrola i njihovo povezivanje sa modelom kroz "ref" atribut, koji u tom slučaju sadrži putanju relativnu u odnosu na "ref" atribut xf:group kontrole. Kao što je već napomenuto, "input" uglavnom predstavlja tekstualno polje, osim kada je tipom podataka preciziran tip xsd:date. "Select1" kontrola, u zavinosti od vrednosti "appearance" atributa, predstavlja radio dugmad, padajuću listu ili listu. Rezultujuća forma prikazana je na *Slika 2*.

Za sve kontrole korisničkog interfejsa karakteristično je i sledeće:

- korisnički izgled se, kao i u XHTMLu, definiše pomoću kaskadnih stilova (CSS). Svaka komponenta iz tog razloga sadrži ”class” atribut.
- definisanje određenih DOM2 događaja i upravljača događajima, koji ih obrađuju.

Povezivanje kontrola korisničkog interfejsa sa modelom omogućava i dinamičke zavisnosti u XPath izrazima povezivanja. Na primer:

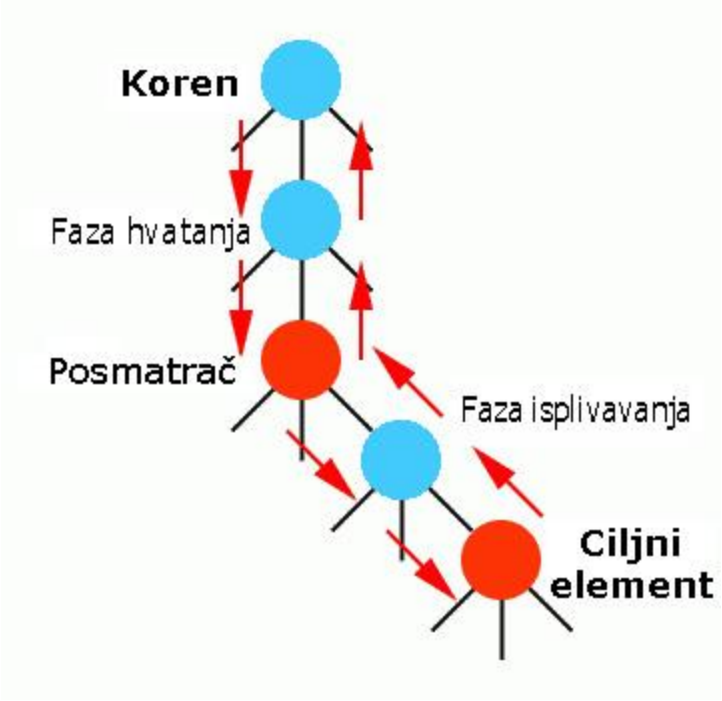
```
1: <select1 ref=’/ui/curr’ /> (glavni deo forme)
2: <input ref=’person[@id=’/ui/curr’]/birth’ /> (detalji)
```

Tekstualno polje zadato u liniji 2. je povezano sa rođendanom osobe, ali specifirani predikat određuje dodatnu osobinu: select1 kontrola u liniji 1. određuje osobu čiji je rođendan moguće izmeniti u određenom momentu. Ovo omogućava pravljenje tzv. glavnog dela forme sa detaljima (eng. master-detail). Proces promene trenutno vezanog čvora korisničkog interfejsa naziva se ponovno povezivanje (eng. *rewiring*), i izvršava se automatski kada se procesira xforms-refresh događaj.

3.4. XForms događaji i akcije

Model događaja (eng. event model) u XForms jeziku zasnovan je na DOM2 specifikaciji događaja, nad kojom je zasnovan i model događaja u JavaScript-u. Novina u XForms jeziku je mogućnost deklarativnog registrovanja obrađivača događaja (eng. event handler) u skladu sa specifikacijom XML događaja. Pomenuta specifikacija definiše događaje i njihovu obradu na sledeći način:

Događaj (eng. event) formalno predstavlja asinhronu pojavu koja je pridružena elementu u XML dokumentu. Kada se događaj desi, prosleđuje se nadole kroz stablo dokumenta do elementa nad kojim se događaj desio (koji je prouzrokovao događaj) koji se naziva još i ciljni element (eng. target) događaja. Ova faza prosleđivanja događaja naziva se faza *hvatanja* (eng. capture). Narednu fazu predstavlja tzv. faza *isplivavanja* (eng. bubble) u kojoj se događaj prosleđuje u obrnutom smeru, od ciljnog elementa ka korenu dokumenta. U toku obe faze svaki od elemenata, kroz koji događaj prolazi, može da se registruje kao tzv. posmatrač (eng. observer) i može da izvrši određenu akciju, stopira dalje propagiranje događaja ili stopira izvršavanje podrazumevane akcije vezane za događaj. Sledeća slika ilustruje obradu događaja.



Slika 7. Faze životnog ciklusa DOM2 događaja

Događaji u XForms-u mogu da se svrstaju u četiri kategorije [24]:

- *događaji životnog ciklusa* – predstavljaju događaje koji se tiču postavljanja i gašenja XForms mašinerije (eng. engine),
- *notifikacije* - predstavljaju indikatore da se nešto dogodilo, poput fokusiranja polja, polje je postalo validno / nevalidno, itd,
- *interakcije* - događaji koji uzrokuju neku vrstu obrade,
- *error* – indikatori greške.

Lista događaja definisana XForms specifikacijom predstavljena je sledećom tabelom.

Ime događaja	Prekidiv?	Isplivava?	Ciljni element
Događaji inicijalizacije			
xforms-model-construct	Ne	Da	`model`
xforms-model-construct-done	Ne	Da	`model`
xforms-ready	Ne	Da	`model`
xforms-model-destruct	Ne	Da	`model`
Događaji interakcije			
xforms-previous	Da	Ne	form kontrola
xforms-next	Da	Ne	form kontrola

Ime događaja	Prekidiv?	Isplivava?	Ciljni element
xforms-focus	Da	Ne	form kontrola
xforms-help	Da	Da	form kontrola
xforms-hint	Da	Da	form kontrola
xforms-rebuild	Da	Da	`model`
xforms-refresh	Da	Da	`model`
xforms-revalidate	Da	Da	`model`
xforms-recalculate	Da	Da	`model`
xforms-reset	Da	Da	`model`
xforms-submit	Da	Da	`submission`
Događaji notifikacija			
DOMActivate	Da	Da	form kontrola
xforms-value-changed	Ne	Da	form kontrola
xforms-select	Ne	Da	`item` ili `case`
xforms-deselect	Ne	Da	`item` ili `case`
xforms-scroll-first	Ne	Da	`repeat`
xforms-scroll-last	Ne	Da	`repeat`
xforms-insert	Ne	Da	`instance`
xforms-delete	Ne	Da	`instance`
xforms-valid	Ne	Da	form kontrola
xforms-invalid	Ne	Da	form kontrola
DOMFocusIn	Ne	Da	form kontrola
DOMFocusOut	Ne	Da	form kontrola
xforms-readonly	Ne	Da	form kontrola
xforms-readwrite	Ne	Da	form kontrola
xforms-required	Ne	Da	form kontrola
xforms-optional	Ne	Da	form kontrola
xforms-enabled	Ne	Da	form kontrola
xforms-disabled	Ne	Da	form kontrola
xforms-in-range	Ne	Da	form kontrola
xforms-out-of-range	Ne	Da	form kontrola
xforms-submit-done	Ne	Da	`submission`
xforms-submit-error	Ne	Da	`submission`
Indikatori grešaka			
xforms-binding-exception	Ne	Da	Bilo koji element koji sadrži binding

Ime događaja	Prekidiv?	Isplivava?	Ciljni element
			izraz
xforms-link-exception	Ne	Da	`model`
xforms-link-error	Ne	Da	`model`
xforms-compute-exception	Ne	Da	`model`

Tabela 1. Lista događaja definisana XForms specifikacijom. Za svaki događaj definisan je ciljni element (eng. target), kao i to da li je događaj prekidiv i da li isplivava (eng. bubbles)

Akcija predstavlja način na koji se odgovara na događaj. Neke od najvažnijih akcija definisanih XForms jezikom su:

- *message* - definiše poruku koja se prikazuje korisniku u kojoj je moguće zadati nekoliko nivoa "nametljivosti", od obične sugestivne do modalne poruke. Sledećom sekvencom se definiše akcija koja prikazuje modalnu poruku nakon inicijalizacije XForms procesora:
```
<xf:message level="modal" ev:event="xforms-ready">
Hello!
</xf:message>
```
- *setvalue* - postavlja vrednost čvora instance modela. Sledećom sekvencom inicijalizuje se vrednost partnerske radnje registracionog dokumenta na "false":
```
<xf:setvalue ref="registrationForm/partnerCompany" value="false"
             ev:event="xforms-ready" />
```
- *dispatch* - šalje događaj preciziran vrednošću "name" atributa do elementa navedenim kao vrednost "target" atributa
- *send* - šalje podatke sa forme, emitujući pri tom xforms-submit događaj

Za grupisanje akcija koristi se `<action>` element. Akcije se u tom slučaju izvršavaju po redu kako su navedene.

3.5. O razvoju aplikacija zasnovanih na XForms specifikaciji

Postoje različiti načini za razvijanje veb aplikacija zasnovanih na XForms specifikaciji [23]. Svaka ima svoje prednosti i mane:

1. **Izvorna podrška od strane veb pretraživača**: ako User Agent izvorno podržava XForms, tada se XHTML + XForms opis korisničkog interfejsa, zajedno sa XML šemom, prima kao statički dokument od strane servera. Instanca modela se može dinamički napraviti od strane servera i serijalizovati kao XML. Slično, podaci sa forme se serijalizuju u formi XML-a prilikom slanja na server i obrađuju se na serverskoj strani aplikacije. Na žalost niti jedan od vodećih veb pretraživača Firefox, IE, Google Chrome ili Opera, nema izvornu podršku za XForms. Jedini takav pretraživač je XSmiles, veb pretraživač otvorenog koda, implementiran u programskom jeziku Java.

- **Prednosti**: vrlo malo kašnjenje korisničkog interfejsa; relativno lak razvoj serverske strane; pretraživač je jedina pouzdana komponenta (što je i inače po definiciji – ako korisnik želi da vidi sadržaj na odgovarajućoj veb adresi, treba da se uzda u to da će ga pretraživač korektno prikazati); malo opterećenje mreže.
- **Mana**: podržan samo od strane eksperimentalnih veb pretraživača

2. **Dodatak (eng. plugin) veb pretraživača**: primer takvog dodatka je formsPlayer za Interner Explorer, kao i Mozilla XForms projekat za Mozillu Firefox. Kada se dodatak instalira, priča je ista kao i u prethodnom slučaju.
   - **Prednosti**: iste kao i u predhodnom slučaju.
   - **Mane**: podržan samo od strane nekoliko vodećih pretraživača; pouzdanu komponentu, osim pretraživača, predstavlja i proizvođač samog dodatka.
3. **Implementacije zasnovane na AJAXu**: kako svi vodeći veb pretraživači imaju izvornu podršku za AJAX, moguće je napraviti podršku za XForms pomoću njega. Na ovaj način je moguće napraviti klijentsku stranu u XForms-u, bez instaliranja dodatnog softvera na klijentu.
   - **Prednosti**: relativno laka implementacija serverske strane; pretraživač je jedina pouzdana komponenta.
   - **Mane**: promenljivo kašnjenje korisničkog interfejsa; veće opterećenje mreže zbog transfera XForms biblioteke zajedno sa opisom korisničkog interfejsa.
4. **Transformacije na serverskoj strani**: XForms + XHTML opis korisničkog interfejsa se transformiše u običan XHTML ili HTML na serverskoj strani. Slanje može biti izvršeno ili AJAX-om ili slanjem HTML forme i transformacijom u XML na serveru. Ovaj pristup koristi se od strane Orbeon i Chiba procesora formi.
   - **Prednosti**: pretraživač je jedina pouzdana komponenta.
   - **Mane**: promenljivo kašnjenje korisničkog interfejsa; komplikovano razvijanje serverske strane; veće opterećenje mreže.

3.6. XPath i XForms

XPath izrazi u tekućoj verziji XForms-a (1.1) zasnovani su na XPath 1.0 specifikaciji. XForms koristi XPath za:

- adresiranje čvorova instanci modela u izrazima povezivanja,
- za predstavljanje ograničenja nad atributima stavki modela,
- za izvršavanje različitih računskih operacija.

Za vreme evaluacije svaki XPath izraz mora biti sintaksno ispravan, u suprotnom odgovorajući događaj, koji označava grešku, biće prosleđen (xforms-compute-exception ili xforms-binding-exception).

3.7. XML šema i XForms

Kao što je već napomenuto, tipovi podataka u XForms-u zasnovani su na tipovima podataka definisanih u XML šemi, sa nekoliko navedenih odstupanja i dodatno definisanih tipova.

XML dokument, koji opisuje model i koji se procesira od strane XForms-a, u većini slučajeva je definisan XML šemom. Ako takva šema postoji, ona može da se iskoristi tako što se na nju ukazuje iz XForms modela, čime bi informacije o tipovima podataka i ograničenjima bile dostupne XForms procesoru.

XML šema predstavlja jednu od centralnih pojmova ove teze i automatizacije generisanja formi, jer se njome opisuje struktura XML modela forme i na osnovu nje se mogu generisati Java klase modela u koje se preslikava XML model forme, a koji je zatim pogodan za dalju obradu.

3.8. HTML5 i XForms

W3C grupa je 1998. godine prestala svoj rad na HTML-u i počela je da radi na XHTML 1.0 specifikaciji, koja je ubrzo postala zvanična preporuka 2002. godine. XHTML je u početku bio jako uspešan, naročito zbog toga što je njime predviđeno da HTML dokumenti budu u skladu sa XML sintaksom. Međutim, niti jedan od veb pretraživača nije imao striktnu podršku za proveru XML sintakse, definisane standardom. Upravo je pooštrenje provera trebalo da bude definisano narednom XHTML 2.0 specifikacijom. Njome je takođe bilo predviđeno definisanje XForms specifikacije novim standardom za forme, XFrames specifikacije novim standardom za HTML okvire (eng. frame), itd. Takođe, mnoge "sumnjive" i loše stvari iz HTML-a su bile odbačene, narušavajući kompatibilnost sa prethodnim verzijama HTML-a (eng. backward compatibility). Upravo je narušavanje kompatibilnosti i uvođenje tehnologija koje su zahtevale od veb programera njihovo učenje i razumevanje bio razlog da grupa ljudi iz Opere, Mozille i Apple-a formiraju *Web Hypertext Application Technology Working* grupu, skr. WHATWG, sa idejom da prošire HTML (umesto da ga zamene), na takav način da se ne naruši kompatibilnost unazad. Rad na, kako će se ispostaviti, novom standardu započeli su 2004. godine i nazvali su ga **HTML5**. Iako je HTML5 specifikacija još u fazi izrade, svi vodeći veb pretraživači se trude da je implementiraju, u manjoj ili većoj meri. U međuvremenu, 2009. godine W3C grupa je objavila da prestaje sa razvojem XHTML 2.0 specifikacije.

Kako se cela evolucija HTML-a, prihvatanje HTML5 specifikacije kao novog standarda i odbacivanje XHTML 2.0 reflektuje na budućnost XForms specifikacije? Odgovor je **nikako**. XForms 1.1 specifikacija je postala zvanična preporuka nakon što je W3C grupa odlučila da obustavi rad na XHTML 2.0 specifikaciji. Takođe, W3C grupa nije napustila rad na XForms specifikaciji: aktivno se radi na 1.2 i 2.0 specifikaciji, koje donose mnoga poboljšanja.

HTML5 je uneo novine u definiciji forme, pre svega, za ulazna polja forme definisani su novi tipovi: number, date, email, pattern, tel, itd, koji definišu skup raspoloživih vrednosti polja i vizuelni prikaz. Nad njima je moguće definisati ograničenja, poput najmanje i najveće vrednosti, formata datuma, obrasca, itd. Nad svakim poljem se izvršava validacija (osim ako eksplicitno nije ukinuta), i onemogućava se njeno slanje ukoliko su podaci u formi nevalidni.

U poređenju sa XForms formama, HTML5 forme su i dalje veoma inferiorne: njima se ni blizu ne mogu definisati ograničenja koja se mogu definisati XML šemom (statička) i XPath izrazima (dinamička). Takođe, model, kontrole korisničkog interfejsa i definicija obrađivača događaja i dalje nisu razdvojene, otežavajući na taj način bilo kakvu automatizaciju.

Kako HTML5 specifikacija definiše dva načina serijalizacije, prvi u obliku HTML-a i drugi u obliku XML-a (tzv. XHTML5), i dalje je validno i moguće koristiti XForms forme u okviru HTML5 dokumenta. Smatra se da je XForms specifikacija jedna od najboljih koja je proistekla iz W3C grupe, a istovremeno i jedna od najmanje prihvaćenih, uglavnom zbog svoje kompleksnosti i zavisnosti od drugih XML-zasnovanih jezika. XForms forme, zbog svoje arhitekture i jasne razdvojenosti komponenata forme, predstavljaju i dalje idealan izbor u aplikacijama u kojima je automatizacija generisanja formi od presudnog značaja.

4 Validacija

Najčešću grešku u implementaciji validacione logike predstavlja njena implementacija isključivo na klijentskoj strani. U tom slučaju, prilikom dodavanja novog tipa klijenta, celokupnu validaciju je potrebno duplirati. Problem je još izraženiji u slučaju veb aplikacija i korisničkog interfejsa u obliku XHTML-a. Validacija je tada implementirana u JavaScript-u pod kontrolom veb pretraživača, koji je pak pod kontrolom korisnika. Kada se JavaScript onemogući, pristup aplikaciji postaje nekontrolisan. U slučaju veb aplikacija grešku u dizajnu predstavlja i smeštanje validacije u veb sloj. Kao i u prethodnom slučaju, ako se tip klijenta promeni, npr. kroz veb servise, Swing aplikaciju sa RPC pozivima, Ajax sa DWR-om takođe kroz RPC, i sl., validaciona logika se mora duplirati i ne može biti ponovno iskorišćena. Neke od najpoznatijih Java veb biblioteka poput Struts i Struts 2 [30] su u ovom pogledu loše dizajnirane i forsiraju smeštanje validacije u veb sloj.

Iz prethodno navedenog sledi da je veoma bitno izdvojiti validacionu logiku u posebnu komponentu, koja se zatim može iskoristiti u poslovnom (servisnom) sloju i po potrebi u veb sloju (u slučaju veb aplikacija), ili na samom klijentu u slučaju aplikacija sa bogatim grafičkim korisničkim interfejsom. Prethodna konstatacija važna je iz dva razloga:

- kako servisni sloj izlaže funkcionalnost i implementira poslovnu logiku sistema, to se mora garantovati validnost argumenata metoda servisa u skladu sa unapred definisanim specifikacijama i
- validaciona logika može biti ponovno iskorišćena.

U slučaju veb aplikacija veoma je važan odziv aplikacije prema korisniku i prikaz informacija o greškama što je pre moguće, idealno, pre slanja samih podataka na server. Najčešće rešenje, koje se primenjuje u ovom slučaju, je dupliranje osnovne validacije i u JavaScriptu i na serverskoj strani. Problem kod ovakvog pristupa predstavlja pisanje složenijeg i obimnijeg koda. Takođe, održavanje kompleksnog JavaScript koda predstavlja pravu noćnu moru.

Baš iz tog razloga predloženi generator formi pomoću XForms jezika predstavlja dobit jer se validacija i ograničenja definisani XML šemom i XPathom mogu iskoristiti za generisanje serverske validacije. Na taj način su klijentska i serverska validacija sinhronizovane, a vreme potrebno za pisanje koda se drastično smanjuje.

4.1. Koncepti validacije

Koncepti kojih se drže većina popularnih validacionih biblioteka su sledeći:

- validacija se izvršava od strane odgovarajućeg *Validatora*, koji sadrži svu validacionu logiku i koji se vezuje za određenu klasu domenskog modela čije se instance validiraju. Validator sam po sebi ne izbacuje nikakvu vrstu izuzetka ukoliko objekat nije validan, već "skuplja" informacije o tome šta nije u redu, ostavljajući nekom drugom izbor o tome šta treba raditi sa prikupljenim greškama.
- validator prikuplja i čuva greške u instanci klase specijalno dizajnirane za te svrhe, na osnovu koje se može izvršavati dalje ispitivanje i obrada greške i određivanje daljeg toka korišćenja. Martin Fowler u [1] naziva takav objekat *Notification*.
- greške su povezane sa samom instancom koja se validira od strane validatora. Postoje dva tipa grešaka: **globalne** (objektne) greške koje se vezuju za sam objekat pod validacijom u globalnom smislu i koje predstavljaju informaciju o tome da objekat ne valja sam po sebi. Drugu vrstu čine tzv. **greške nad poljima** koje označavaju nevalidnost određenih atributa objekta pod validacijom. Na osnovu objekta koji čuva informacije o greškama moguće je ispitati postojanost određene greške nad samim objektom ili nad nekim od njegovih atributa.
- svaka greška sadrži odgovarajući **kôd,** koji jedinstveno određuje grešku u okviru konteksta (globalnom/nad poljem), i na osnovu kojeg se može odrediti odgovarajuća lokalizovana poruka, koja će biti prikazana korisniku u slučaju greške. Druga stavka je veoma bitna jer omogućava lokalizaciju poruka o greškama, koje se zatim mogu prikazati korisniku na formi na odgovarajućem mestu.
- greške nepoklapanja tipova između određenog polja (atributa) i stvarne vrednosti atributa (npr. u slučaju slanja parametara HTTP zahteva kao String-a koji ne predstavlja numeričku vrednost i koji se vezuje za celobrojni atribut klase), se obrađuju konzistentno i na isti način kao validacione greške. U ovakvom slučaju se semantička validacija, koja uključuje vrednost datog atributa, obično ne razmatra pošto vrednost sama po sebi nije sintaksno ispravna.

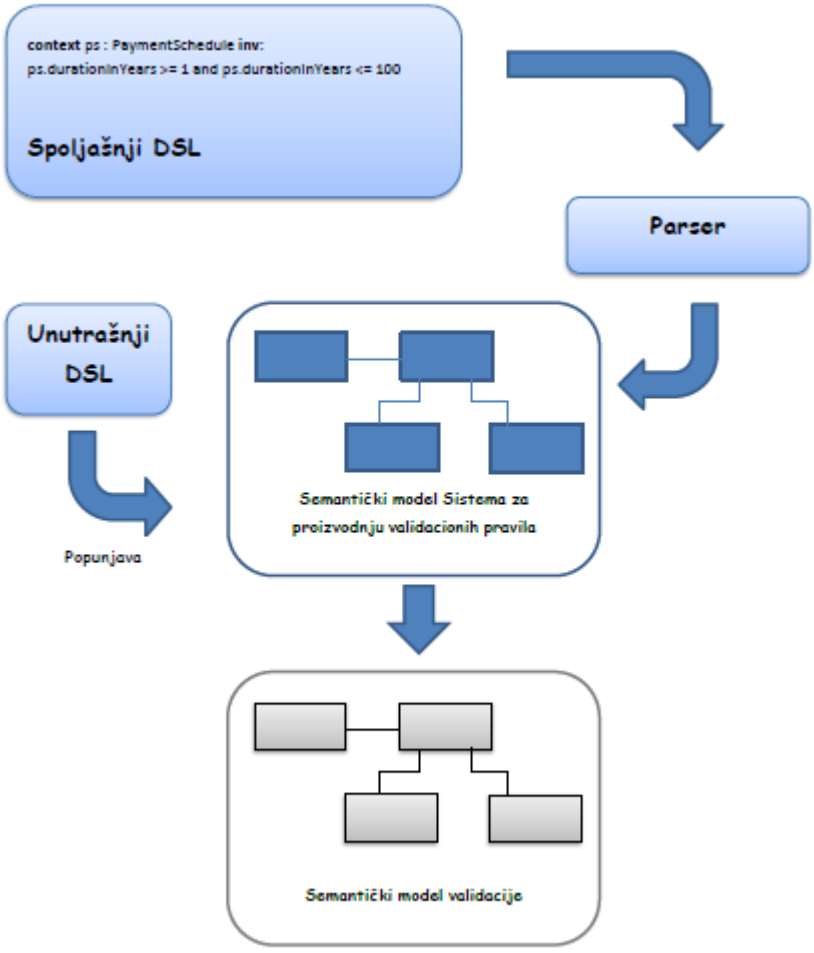
Prvi od ciljeva istraživanja je dizajniranje *Semantičkog modela validacije*, koji treba da podržava prethodno opisane koncepte sa jedne strane, dok sa druge treba da je dovoljno širok da obuhvati aspekte koji nisu obuhvaćeni od strane niti jedne od validacionih biblioteka i koji će biti izneti u nastavku ovog rada.

Drugi je izrada Semantičkog modela *Sistema za proizvodnju validacionih pravila* (eng. Production Rule System of Validation Rules). Kako sâm proces validacije uključuje niz *if-then-else* naredbi, na osnovu kojih se određena greška prijavljuje u slučaju nekorektnosti podataka, to se prirodno kao model nameće uprošćen Sistem za proizvodnju pravila. U opštem slučaju Sistem za proizvodnju pravila se sastoji od skupa pravila, od kojih svako ima uslov i akciju koja treba da se izvrši u zavisnosti od logičke vrednosti koja se dobija proverom uslova. Najsloženiji deo Sistema za proizvodnju pravila predstavlja logika za izvršavanje pravila [19], pošto određene akcije mogu da promene stanje modela nad kojim se

izvršavaju pravila. U tom slučaju primenjuju se različiti algoritmi i strategije za izvođenje i izvršavanje pravila. Jedan od najpoznatijih algoritama je Reteov algoritam [31]. U slučaju validacije, akcije Sistema za proizvodnju pravila ne menjaju stanje modela i svedene su na prosto prijavljivanje grešaka, pa nema potrebe za komplikovanim strategijama za određivanje redosleda kojim se pravila izvršavaju. Preciznije, menja se jedino stanje objekta u kojem se greške prikupljaju. Međutim, ta činjenica ne utiče bitno na logiku izvršavanja pravila.

Treći cilj predstavlja definisanje unutrašnjeg DSL-a "preko" Semantičkog modela Sistema za proizvodnju validacionih pravila, kojim je moguće lako i intuitivno definisanje pravila i popunjavanje Semantičkog modela.

Na kraju, cilj istraživanja predstavlja definisanje spoljašnjeg DSL-a sa sintaksom pogodnom za zapis validacionih pravila od strane sistemskih analitičara, na osnovu kojeg se takođe popunjava Semantički model.



Slika 8. Tok procesiranja unutrašnjeg i spoljašnjeg DSL-a

Automatizacija se dobija na taj način što se validacija izvršava učitavanjem i parsiranjem definicija validacionih pravila zapisanih pomoću spoljašnjeg DSL-a, prevođenjem u Semantički model sistema za proizvodnju validacionih pravila, zatim prevodenjem u Semantički model validacije i na kraju njegovom interpretacijom.

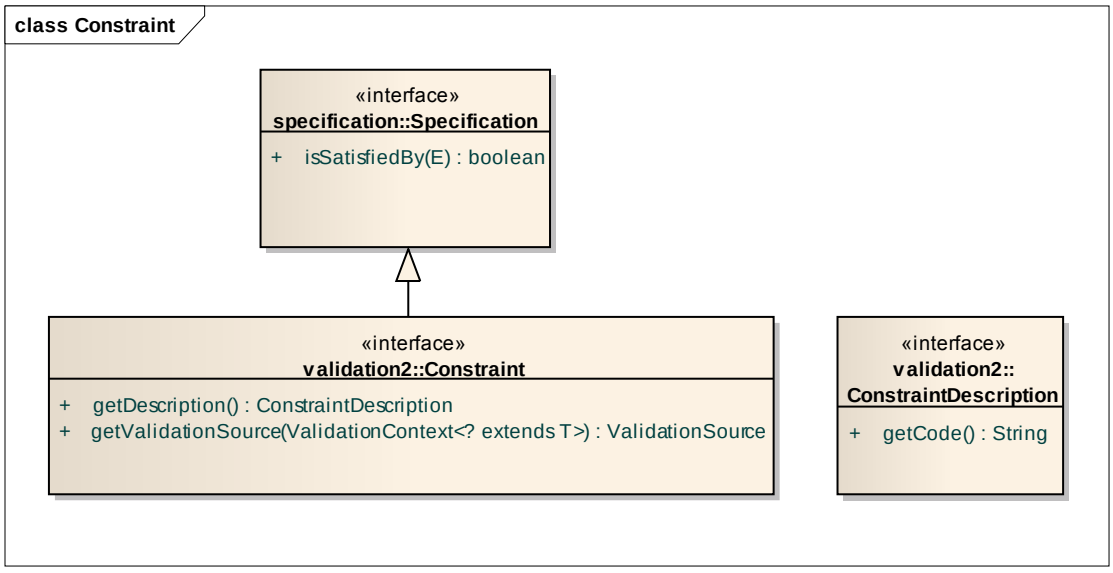
4.2. Semantički model validacije

Semantički model validacije, osim prethodno opisanih koncepata, trebalo bi da omogući:

- definisanje različitih nivoa (težine) grešaka. Pojedina poslovna pravila se mogu smatrati manje važnim i u slučaju njihove nezadovoljenosti, korisnika treba samo upozoriti i omogućiti mu izvršavanje započete akcije ako odluči da ignoriše upozorenja. Težine grešaka su zgodne i u slučaju kada se definišu strategije za njihovu obradu ili strategije za izvršavanje validacije, npr. postojanje greške višeg nivoa, nad određenim atributom, može usloviti ignorisanje provera ostalih pravila nad istim atributom ili nad drugim zavisnim atributima.
- definisanje različlitih strategija za ispitivanje grešaka: nad određenim atributom, kodom greške, tipom atributa, itd.
- definisanje različitih strategija za određivanje kodova grešaka, u zavisnosti od osnovnog koda greške i konteksta u kojem je greška prijavljena. Na taj način lokalizovane poruke o grešci, koje su povezane sa odgovarajućim kodom i kontekstom, mogu biti ponovno iskorišćene.
- kontekst u kojem se validacija izvršava od strane validatora treba da je hijerarhijski, jer se na taj način dobija uniformna struktura pogodna za prijavljivanje i ispitivanje grešaka, kao i mogućnost ponovne iskoristivosti validatora.

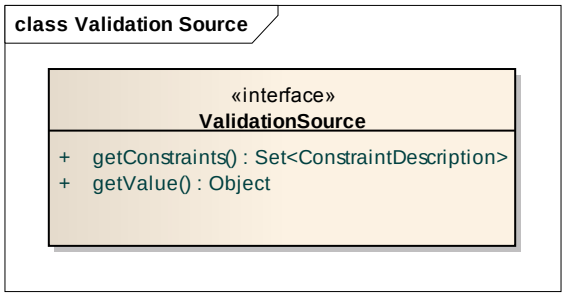
Programski interfejs (eng. Application Programming Interface - API) semantičkog modela validacije čine:

Ograničenja (eng. constraints), koja su definisana opisom i logikom ispitivanja da li određena instanca zadovoljava ograničenje ili ne.



Slika 9. Dijagram klasa modela ograničenja

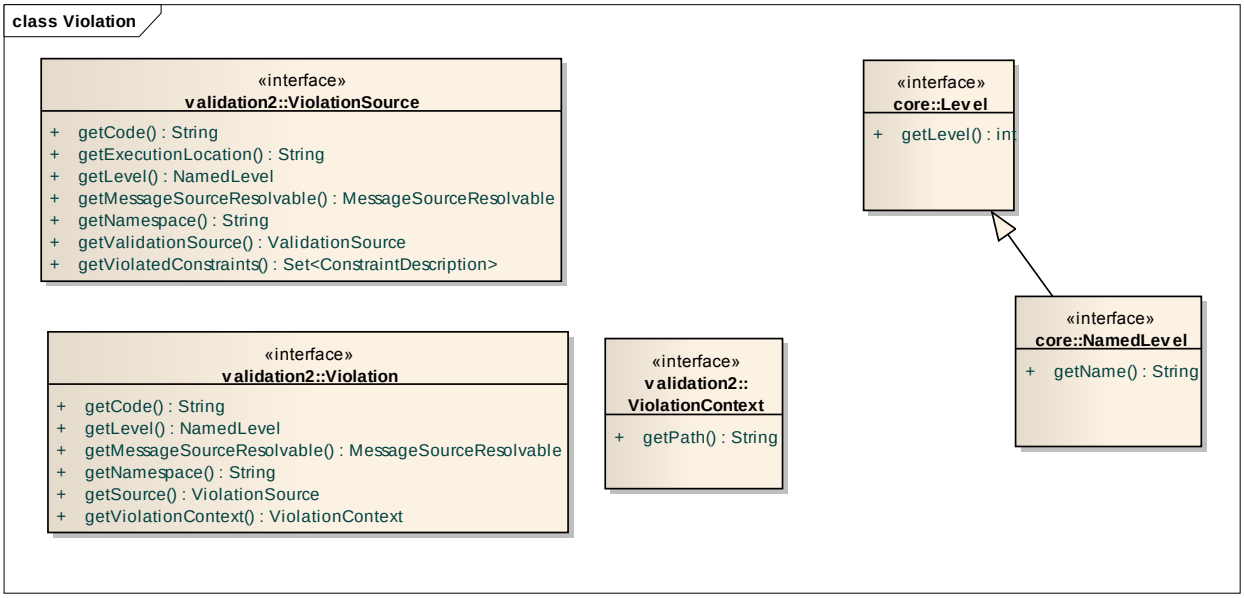
Izvor validacije (eng. validation source) je deo modela ili sistema za koji se definišu ograničenja koja treba da su zadovoljena



Slika 10. Klasni dijagram izvora validacije

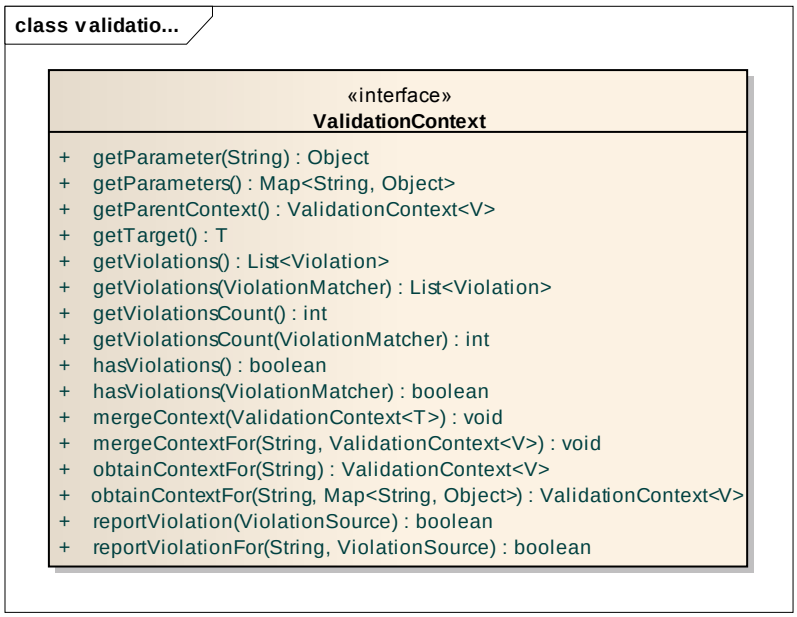
U slučaju da izvor validacije ne zadovoljava neki od definisanih ograničenja, on tada postaje **izvor prekršaja** (eng. Violation Source)**.** Za izvor prekršaja karektiristični su: kôd prekršaja, nivo (težina), izvor validacije i skup opisa ograničenja koja su narušena. Na osnovu koda izvora prekršaja moguće je odrediti lokalizovanu poruku o grešci. Za jedan izvor prekršaja moguće je prijavljivanje više **prekršaja** (eng. violation), definisanim kodom, nivoom (težinom), izvorom prekršaja nad kojim je utvrđen i kontekstom u okviru kojeg se prekršaj desio. Na primer, u slučaju da klasa *Osoba* kao atribut ima instancu klase *Adresa*, čiji je atribut "ulica" obavezan, sa stanovišta adrese kontekst je "ulica", dok je sa stanovišta osobe kontekst "adresa.ulica". U oba slučaja izvor prekršaja je "ulica". Ponavljanje koda, nivoa i određivanja lokalizovanih poruka u izvoru prekršaja i samom prekršaju daje određenu fleksibilnost. Naime, instanca klase koja je zadužena za prijavljivanje prekršaja (validacioni kontekst opisan u nastavku) može da promeni kôd ili nivo greške dobijene iz izvora prekršaja i dodeli prekršaju. Na taj način izvor greške, koji je u jednom slučaju okarakterisan kao

fatalan, može da bude promenjen i nad njim može biti definisan prekršaj slabije težine, ili obrnuto, u zavisnosti od poslovnih pravila.



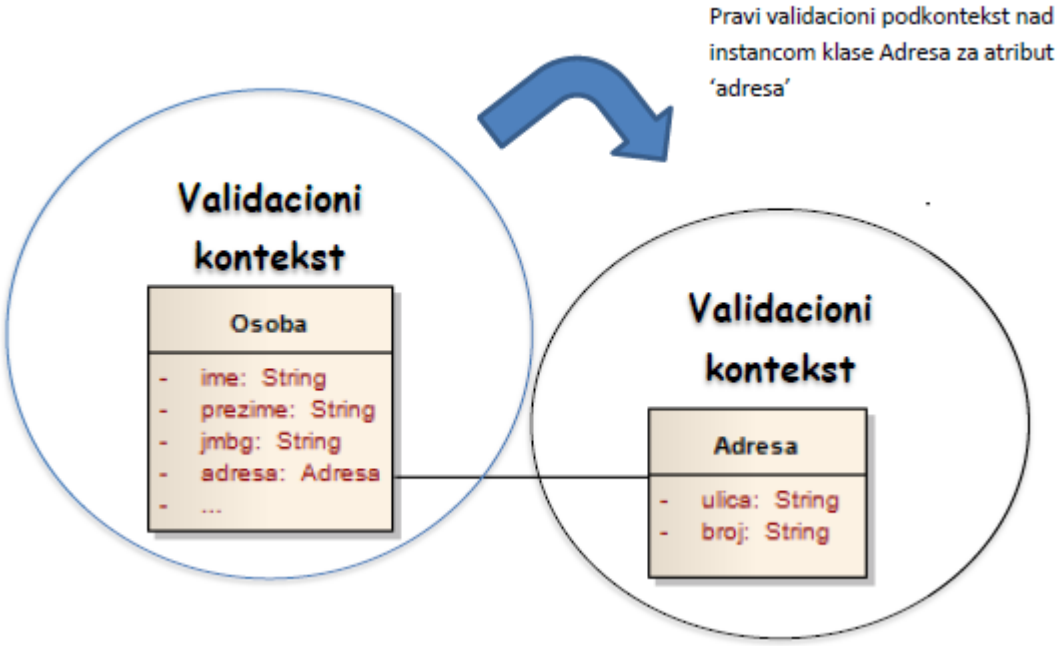
Slika 11. Dijagram klasa prekršaja

Validacioni kontekst (eng. ValidationContext) je centralni deo API-a koji služi za prikupljanje prekršaja i njihovo ispitivanje. Kontekst je povezan sa objektom nad kojim se vrši validacija (eng. target) i definiše metode kako za prijavljivanje prekršaja, tako i za ispitivanje njihovog broja, dohvatanje svih prijavljenih prekršaja ili prekršaja koji odgovaraju nekoj specifikaciji (koji su u skladu sa specifikacijama prekršaja, opisanih u nastavku).



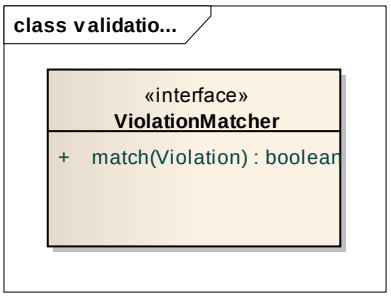
Slika 12. Klasni dijagram validacionog konteksta

Jedna od najvažnijih odlika validacionog konteksta je da je hijerarhijski i može da proizvede podkontekste (decu kontekste), koji su povezani sa podobjektima ciljnog objekta osnovnog konteksta. Ova osobina je važna jer omogućava ponovnu iskoristivost validatora koji izvršava validaciju i prijavljuje prekršaje podkontekstima. Prijavljivanje prekršaja u podkontekstu automatski povlači i prijavljivanje prekršaja u otac-kontekstu, sa izmenjenim kontekstom prekršaja. Na primer, u prethodno opisanom primeru neka je definisan validator adrese (*AdresaValidator*), koji greške prijavljuje u validacionom kontekstu, čiji je ciljni objekat instanca klase *Adresa* i u kojem je opisana sva validaciona logika. Potrebno je definisati validator za instance klase *Osoba*, koja je ciljni objekat validacionog konteksta nad kojim validator operiše. Da se logika validacije adrese, koju osoba poseduje, ne bi duplirala, *OsobaValidator* može pomoću osnovnog konteksta da napravi podkontekst za pripadajuću adresu i da nad njim pozove *AdresaValidator*. U tom slučaju napravljeni podkontekst bi prijavio sve prekršaje ocu-kontekstu.



Slika 13. Hijerarhijska struktura validacionog konteksta, koja oslikava hijerarhijsku kompoziciju objektnog modela

Pomoću **specifikacija prekršaja** (eng. Violation Matcher) moguće je ispitati validacioni kontekst u cilju dobijanja informacija o prijavljenim greškama. U najosnovnijem slučaju specifikacija može da se definiše tako da se proverava da li nad određenim atributom ima prijavljenih prekršaja ili prekršaja sa određenim kodom. Ta informacija, u slučaju prijavljenih grešaka, može da se iskoristi npr. u prezentacionom sloju u kojem komponenta korisničkog interfejsa, koja je povezana sa tim atributom, može biti drugačije prikazana (npr. crvenom bojom) i uz nju mogu biti prikazane lokalizovane poruke o grešci.



Slika 14. Dijagram klasa specifikacija prekršaja

Na primer, sledećom sekvencom,

```
ViolationMatcher warningMatcher =
                              ViolationMatchers.parse("imp:weight = 0");
List<Violation> warnings = context.getViolations(warningMatcher);
```

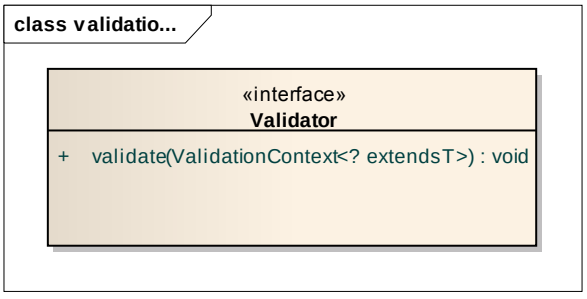
se iz konteksta dobijaju sve greške čija je težina jednaka 0 (upozorenja), dok se sledećom specifikacijom prekršaja

```
ViolationMatchers.parse("code = error.message and imp:weight >= 5
                            and not source:kind = rel");
```

dobijaju sve greške čiji je kod jednak "error.message", težina veća ili jednaka 5 i čiji izvor validacije nije relacioni, tj. ne uključuje izvor sastavljen od atributa objekta pod validacijom, koji su u nekoj vrsti relacije.

Implementacija specifikacije prekršaja može, kao što je u ovom slučaju, biti izvršena definisanjem posebne gramatike i njenog parsera.

Sva validaciona logika, njena provera i prijavljivanje prekršaja, izvršava se od strane *Validatora*. Metoda za proveru validnosti objekta kao argument prima validacioni kontekst, kojem se, u slučaju narušenih poslovnih pravila, prijavljuju prekršaji. Kao što je već napomenuto, validacioni konteksti su hijerarhijski, što omogućava da validatori koji ih koriste budu usko specifirani (modularni) i ponovno iskoristivi u različitim slučajevima korišćenja.



Slika 15. Dijagram klasa validatora

Primer: neka je plan otplate za iznajmljivanje zemljišta (eng. Land Rental) definisan datumom početka otplate (feeCalculationDate), trajanjem izraženom u godinama (durationInYears), iznosom zasnovanim na porezu (taxBasedAmount) i godišnjom naknadom (yearlyFee). Poslovna pravila koja treba da su ispunjena prilikom unošenja podatka o iznajmljivanju zemljišta su:

- datum početka otplate je obavezan,
- trajanje otplate je obavezno i mora biti u intervalu [1, 100] godina,
- godišnja naknada je obavezna i mora biti u intervalu [0.01, 999999.99] i zadata na dve decimale najviše,
- iznos zasnovan na porezu, ako je zadat mora biti zadat na dve decimale najviše i takođe u intervalu [0.01, 999999.99],
- ukupan iznos jednak je proizvodu godišnje naknade i trajanja u godinama i ukoliko je definisan ne sme preći vrednost 999999.99

Korišćenjem prethodno opisanog semantičkog modela validacije, validaciona logika, koja je deo *LandRentalValidator* – a, može biti zapisana na sledeći način:

```
if(landRental.getFeeCalculationStartDate() == null) {
   context.reportViolationFor(
            "feeCalculationStartDate",
            new SimpleViolationSourceBuilder(
                  "error.missing",
                  DefaultViolationLevel.CRITICAL,
                  new ObjectValidationSource(
                      landRental.getFeeCalculationStartDate(),
                      LocalDate.class)
            ).violated(/* opis ogranicenja */)
             .build());
}

if(landRental.getDurationInYears() == null) {
   context.reportViolationFor("durationInYears", /* izvor narusavanja
                                                    validnosti */);
}
else {
      if(landRental.getDurationInYears() < DURATION_IN_YEARS_MIN_VALUE
         ||
         landRental.getDurationInYears() > DURATION_IN_YEARS_MAX_VALUE) {
         context.reportViolationFor("durationInYears",
                /* izvor narusavanja validnosti */);
   }
}

if(landRental.getYearlyFee() == null) {
   context.reportViolationFor("yearlyFee",
         /* izvor narusavanja validnosti */);
}
else {
      if(landRental.getYearlyFee().compareTo(YEARLY_FEE_MIN_VALUE) < 0
         ||
         landRental.getYearlyFee().compareTo(YEARLY_FEE_MAX_VALUE) > 0) {
```

```
            context.reportViolationFor("yearlyFee", /* izvor narusavanja
                                                       validnosti */);
      }

      if(landRental.getYearlyFee().scale > DEFAULT_SCALE) {
            context.reportViolationFor("yearlyFee", /* izvor narusavanja
                                                       validnosti */);
      }
}

if (landRental.getDurationInYears() != null &&
      landRental.getYearlyFee() != null &&

landRental.getDurationInYears().multiply(landRental.getYearlyFee())
              .compareTo(TOTAL_PAYMENT_AMOUNT_MAX_VALUE) > 0) {
   context.reportViolation(/* izvor narusavanja validnosti */);
}
```

U navedenom primeru, u slučaju da datum početka otplate nedostaje, kontekstu se prijavljuje greška nad odgovarajućim atributom (feeCalculationStartDate), kodom (error.missing) i nivoom prioriteta (DefaultViolationLevel.CRITICAL). U slučaju nevalidnosti trajanja u godinama i godišnjeg iznosa otplate, na sličan način se prijavljuju greške. Zbog preglednosti, izostavljeno je pravljenje izvora narušavanja validnosti. Poslednja provera ukupnog plana otplate predstavlja proveru da li je proizvod godišnje naknade i trajanja otplate zadat i da li je prekoračio dozvoljen iznos. U ovom slučaju moguće je prijaviti globalnu grešku nad celom instancom klase *LandRental*, ili relacionu grešku nad atributima klase, koji učestvuju u izračunavanju.

Iz primera se vidi da je Semantički model validacije širok i nešto komplikovaniji za upotrebu zbog toga što treba da omogući definisanje različitih vrsta grešaka i njihovo ispitivanje.

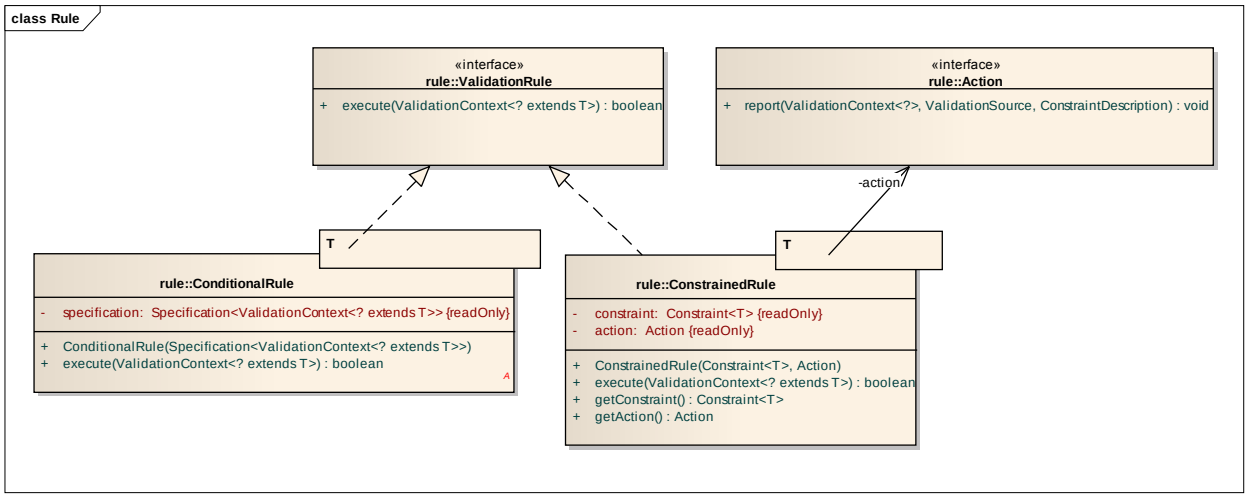
4.3. Semantički model sistema za proizvodnju validacionih pravila

Poslovna pravila se definišu kroz seriju validacionih pravila, pa se za njihovo definisanje može iskoristiti *sistem za proizvodnju validacionih pravila*. U ovakvom sistemu, kao što je već napomenuto, pravilo se sastoji od logičkog uslova i akcije koja predstavlja prijavljivanje greške ukoliko uslov nije zadovoljen.

Semantički model sistema za proizvodnju validacionih pravila čine:

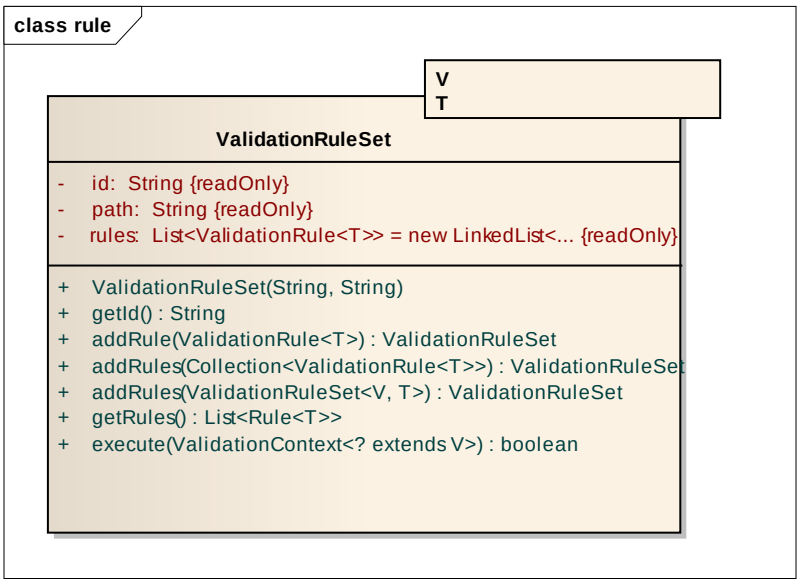
Validaciona pravila (eng. ValidationRule) definišu metodu za izvršavanje unutar validacionog konteksta. Metoda vraća logičku vrednost *true* onda kada je pravilo izvršeno uspešno ili *false*, u suprotnom. Postoje dva vrsta pravila: **uslovno pravilo** (eng. Conditional Rule) i **pravilo sa ograničenjem** (eng. Constrained Rule). U slučaju da je validaciona logika oblika "*ako važi pravilo A i pravilo, B tada mora da važi pravilo C*", tada su pravila A i B uslovna pravila, dok je pravilo C pravilo sa ograničenjem. Slično za validacionu logiku oblika "za matični broj osobe moraju da važe pravila A, B i C", sva tri pravila A, B i C su sa ograničenjem. Za pravila sa ograničenjem karakteristična je akcija (eng. Action) prijave

prekršaja u datom kontekstu nad izvorom validacije sa opisom ograničenja ukoliko je narušeno.



Slika 16. Dijagram klasa validacionih pravila i akcija u sistemu za proizvodnju validacionih pravila

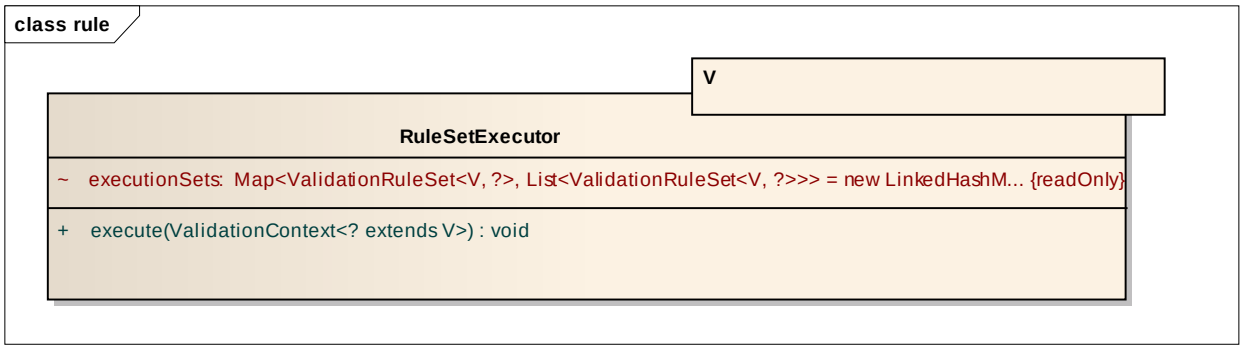
Validaciona pravila se mogu logički i fizički grupisati u **uređeni skup pravila** (eng. ValidationRuleSet). Uređen skup pravila se vezuje za određenu instancu modela ili neki od njegovih atributa, može se identifikovati jedinstvenim identifikatorom pogodnim za njegovo označavanje i definiše metode za dodavanje pravila, kao i za njihovo izvršavanje. U zavisnosti od strategije, skup pravila je uspešno izvršen onda kada su sva njegova pravila izvršena uspešno ili ukoliko je uspešno izvršeno bar jedno njegovo pravilo.



Slika 17. Klasni dijagram skupa validacionih pravila

Sistem za proizvodnju pravila obično svu kontrolu i logiku izvršavanja pravila smešta u komponentu koja se zove **aparat za izvršavanje pravila** (eng. Rule Engine) ili **izvršilac pravila** (eng. Rule Executor) [19]. Pošto su akcije pravila svedene na prijavljivanje grešaka i ne menjaju stanje modela koji je pod validacijom, uz zapažanje da pojedini skupovi pravila mogu biti zavisni od drugih, to je izvršilac pravila implementiran na sledeći način:

- skupovi pravila i zavisnosti među njima predstavljeni su acikličnim usmerenim grafom, u kojem ivica usmerena od čvora A ka čvoru B označava zavisnost skupa pravila u čvoru B od skupa pravila u čvoru A. Graf je interno predstavljen pomoću liste povezanosti (eng. adjacency list).
- definisani skupovi pravila se izvršavaju tako što se graf prvo sortira topološki, dajući na taj način listu svih skupova pravila u kojoj nezavisna pravila prethode zavisnim. Dobijena lista se obilazi od početka i redom se izvršavaju pravila. Ukoliko pravilo nije uspešno izvršeno, proverava se da li postoje pravila koja su od njega zavisna i ako postoje ignorišu se u nastavku.



Slika 18. Klasnii dijagram aparata za izvršavanje pravila

Validaciona logika, definisana u primeru semantičkog modela validacije, može da se, kroz semantički model sistema za proizvodnju validacionih pravila, definiše na sledeći način:

```
RuleSet<LandRental, LocalDate> fcsd
      = new RuleSet< LandRental, LocalDate>("feeCalculationStartDate");
fcsd.addRule(new ConstrainedRule<LocalDate>(new
                                          NotNullConstraint<LocalDate>()));

RuleSet<LandRental, Integer> diy1
      = new RuleSet<LandRental, Integer>("durationInYears");
diy1.addRule(new ConstrainedRule<Integer>(new NotNullConstraint<Integer>()));

RuleSet<LandRental, Integer> diy2
      = new RuleSet<LandRental, Integer>("durationInYears");
diy2.addRule(
      new ConstrainedRule<Integer>(
          new MinInclusiveConstraint<Integer>(DURATION_IN_YEARS_MIN_VALUE)))
    .addRule(
      new ConstrainedRule<Integer>(
          new MaxInclusiveConstraint<Integer>(DURATION_IN_YEARS_MAX_VALUE)));
```

```
RuleSet<LandRental, BigDecimal> yf1
      = new RuleSet<LandRental, BigDecimal>("yearlyFee");
yf1.addRule(
      new ConstrainedRule<BigDecimal>(new NotNullConstraint<BigDecimal>()));

RuleSet<LandRental, BigDecimal> yf2
      = new RuleSet<LandRental, BigDecimal>("yearlyFee");
yf2.addRule(
     new ConstrainedRule<BigDecimal>(
         new MinInclusiveConstraint<BigDecimal>(YEARLY_FEE_MIN_VALUE)))
   .addRule(
     new ConstrainedRule<BigDecimal>(
         new MaxInclusiveConstraint<BigDecimal>(YEARLY_FEE_MAX_VALUE)))
   .addRule(
     new ConstrainedRule<BigDecimal>(new ScaleConstraint(DEFAULT_SCALE));

RuleSet<LandRental, LandRental> tpa1
      = new RuleSet<LandRental, LandRental>(".");
tpa1.addRule(
      new ConstrainedRule<LandRental>(
          new MaxInclusiveFunctionalConstraint<LandRental, BigDecimal>
              (TOTAL_PAYMENT_AMOUNT_MAX_VALUE,
               new UnaryFunction<LandRental, BigDecimal>() {
                    public BigDecimal apply(LandRental arg) {
                       return arg.getYearlyFee().multiply(
                              new BigDecimal(arg.getDurationInYears()));
}})));
```

Za navedene skupove pravila definišu se njihove međusobne zavisnosti, npr. skup *tpa1* je zavisan od skupa koji sadrži pravila *diy1* i *yf1* i ukoliko taj skup pravila nije uspešno primenjen, tada ne treba primenjivati pravila iz skupa *ta2*. Ovako struktuiranim pravilima kompletira se i primenjuje izvršilac pravila.

4.4. Unutrašnji domenski specifičan jezik (DSL) sistema za proizvodnju validacionih pravila

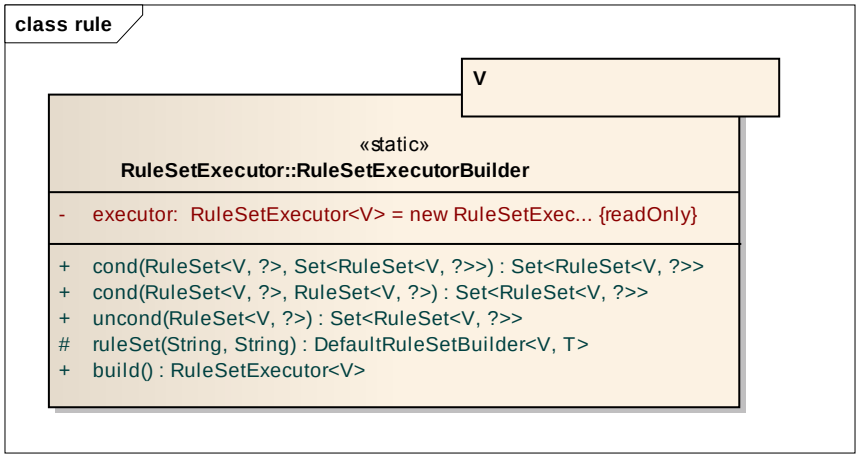
Domenski specifičan jezik (eng. Domain Specific Language - DSL) predstavlja programski jezik ograničene izražajnosti usmerene na određeni domen [19]. Postoje dva glavna tipa DSL – a: **unutrašnji** i **spoljašnji**. *Unutrašnji DSL* je deo nekog većeg opšteg programskog jezika, kao što su Java, Lisp, Ruby ili C#, koji koristi podskup mogućnosti programskog jezika na određeni način. Kao rezultat, unutrašnji DSL bi trebalo da izgleda kao pod-jezik opšteg programskog jezika, tako što je prilagođen ciljnom domenu. *Spoljašnji DSL* predstavlja jezik razdvojen od programskog jezika u kojem se aplikacija razvija. Obično ima svoju specifičnu sintaksu prilagođenu ciljnom domenu, ali može biti opisan i sintaksom nekog drugog jezika, npr. XML-a. Skript napisan u spoljašnjem DSL-u se obično parsira i prevodi u programski jezik u kojem se aplikacija razvija.

Korist, koja se dobija korišćenjem DSL-a, je višestruka. Prva je ta da je kôd napisan pomoću DSL-a lakši za čitanje, razumevanje i održavanje, što povećava produktivnost programera. Drugo, DSL je razumljiv i od strane eksperta u domenu za koji se DSL pravi. Na

taj način programeri i eksperti dele zajednički jezik, koji je čitljiv i razumljiv i jednima i drugima čime se olakšava komunikacija između njih. Ova odlika je veoma važna jer je komunikacija između programera i klijenata obično "usko grlo" u razvoju aplikacija.

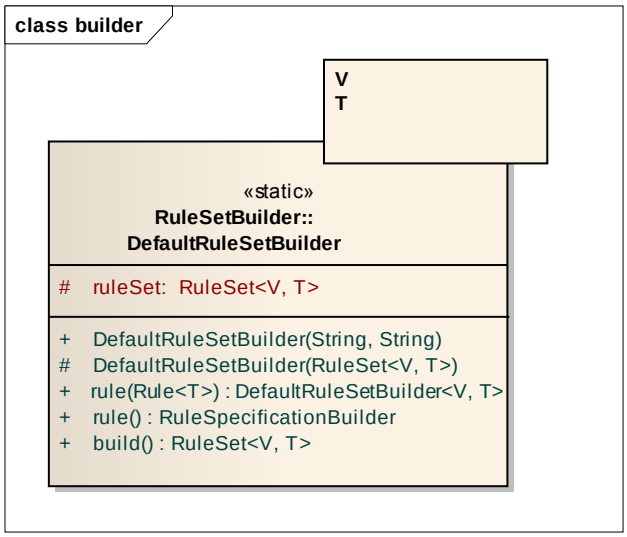
Unutrašnji DSL se obično implementira kombinacijom sledećih tehnika [19]: **graditelj izraza** (eng. Expression Builder) objekat ili familija objekata koji pružaju tzv. tečni interfejs, koji je dizajniran sa ciljem da pruži čitljivost i razumljivost izraza. U svakom momentu graditelj treba da nudi samo one metode koje su dozvoljene u trenutnom kontekstu [32], [33]. Ova tehnika se obično kombinuje sa tehnikom **povezivanja metoda** (eng. Method Chaining) u kojoj metoda kao rezultat vraća tekući objekat, na taj način omogućavajući sekvencijalni poziv druge metode nad njim; **sekvence funkcija** (eng. Function Sequence) predstavlja kombinaciju funkcija u nizu izraza, koje se obično definišu na globalnom nivou, ili na nivou nekog prostora imena (ili klase ili paketa); **ugnježdene funkcije** predstavljaju kompoziciju funkcija ugnježdavanjem njihovih poziva kao argumenata drugih funkcija; **delokrug objekta** (eng. Object Scoping) označava smeštanje DSL koda unutar jednog objekta koji sadrži i metode i podatke neophodne za izgradnju modela. Implementira se obično pravljenjem podklase date klase, čije se metode zatim koriste u delokrugu odvojenom od onog u kojem je definisan.

Za pravljenje skupova pravila koja treba da se izvrše, i uspostavljanje veza između njih, definisan je **graditelj izvršioca pravila** (eng. Rule Set Executor Builder).



Slika 19. Klasni dijagram graditelja izvršioca pravila

Graditeljem je moguće definisati pravila koja se izvršavaju uslovno (eng. conditionally) ili bezuslovno (eng. unconditionally). Graditelj je implementiran pomoću progresivnog interfejsa: metoda *ruleSet* kao rezultat daje graditelja skupa pravila (eng. Rule Set Builder), preko koga se pravila lako dodaju pomoću *tečnog interfejsa*.



Slika 20. Klasni dijagram graditelja skupa pravila

Tehnika *delokruga objekta* omogućava upotrebu graditelja izvršioca pravila na intuitivan način. U Java-i delokrug objekta se implementira korišćenjem dvostruke skupovne zagrade, na taj način se prvom zagradom pravi unutrašnja anonimna klasa koja je podklasa ciljne klase, dok se drugom definiše klasni blok za inicijalizaciju koji sadrži logiku pravljenja. Na taj način se validaciona logika, opisana u prethodnom poglavlju, može definisati na sledeći način:

```
RuleSetExecutorBuilder<LandRental> builder
      = new RuleSetExecutorBuilder<LandRental>() {{
            uncond(ruleSet("feeCalculationStartDate")
                        .rule(mustHave())
                        .build());

            cond(ruleSet("durationInYears")
                        .rule(mustHave())
                        .build(),
                  ruleSet("durationInYears")
                        .rule(minInclusive(DURATION_IN_YEARS_MIN_VALUE))
                        .rule(maxInclusive(DURATION_IN_YEARS_MAX_VALUE))
                        .build());

            cond(ruleSet("yearlyFee")
                        .rule(mustHave())
                        .build(),
                  ruleSet("yearlyFee")
                        .rule(minInclusive(YEARLY_FEE_MIN_VALUE))
                        .rule(maxInclusive(YEARLY_FEE_MAX_VALUE))
                        .rule(scale(DEFAULT_SCALE))
                        .build());

RuleSet<LandRental, LandRental> tpa =
      ruleSet(".")
          .rule()
          .withConstraint(
```

```
                new MaxInclusiveFunctionalConstraint<LandRental, BigDecimal>(
                    TOTAL_PAYMENT_AMOUNT_MAX_VALUE,
                   new UnaryFunction<LandRental, BigDecimal>() {
                       public BigDecimal apply(LandRental arg) {
                            return arg.getYearlyFee()
                           .multiply(new BigDecimal(arg.getDurationInYears()));
                }})).build();

            cond(ruleSet("durationInYear")
                      .rule(exists())
                      .build(),
                 tpa);
            cond(ruleSet("yearlyFee")
                      .rule(exists())
                      .build(),
                 tpa);

}};
RuleSetExecutor<LandRental> executor = builder.build();
```

U navedenom kodu, bezuslovno pravilo za proveru zadatosti datuma početka otplate, zadaje se pozivom metode *uncond(ruleSet)*; metodom *ruleSet(String)* započinje se sa pravljenjem skupa pravila u koji se pravilo dodaje pozivom metode *rule*. Pravljenje skupa pravila završava se pozivom metode *build()* iz graditelja skupa pravila. Uslovna pravila se prave pozivom metode *cond(ruleSet, ruleSet)*.

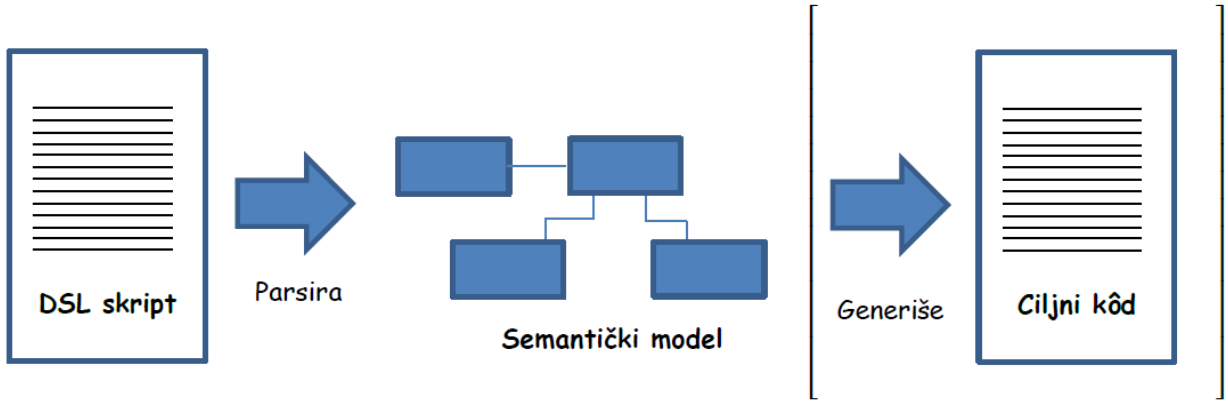
Za pravljenje pravila, koja su najčešće u upotrebi, koristi se *sekvenca statičkih funkcija*, definisanih u klasi *Rules*. Java-in statički uvoz (eng. import) metoda, raspoloživ od verzije 1.5, omogućava njihovu upotrebu na čitljiv način. Tako *mustHave()* pravi pravilo sa *NotNullContraint* ograničenjem, dok npr. *exists()* pravi uslovno pravilo sa *NotNullSpecification* specifikacijom.

Još jedan zgodan način za opis pravila predstavljaju **anotacije**. Anotacije predstavlju podatke o elementima programa kao što su klase, metode ili atributi, koji mogu biti procesirani tokom kompajliranja ili izvršavanja programa. Anotacije su po ugledu na .NET uvedene u Java-u 1.5. Tokom poslednjih godina, Java zajednica predložila je specifikaciju validacije klasa koje su u skladu sa Java Beans specifikacijom [34]. Specifikacijom su definisane anotacije poput *@NotNull, @Null, @Min, @Max, @Size*, itd. kojima se zadaju ograničenja. Moguće je realizovati procesor anotacija koji bi procesirao ovakve anotacije i koji bi napravio i popunio semantički model opisan u ovom poglavlju.

4.5. Spoljašnji DSL sistema za proizvodnju validacionih pravila

Spoljašnji DSL, kao što je već navedeno, predstavlja jezik koji je razdvojen od glavnog jezika kojim je aplikacija napisana. Obično spoljašnji DSL ima sintaksu prilagođenu domenu na koji se odnosi, ali može imati i sintaksu drugog jezika, među kojima je XML najčešći izbor. Korišćenje spoljašnjeg DSL-a omogućava jasnu razdvojenost DSL skripta od parsera i

semantičkog modela: skript je napisan posebnim jezikom i preveden od strane parsera u semantički model. Arhitektura procesiranja spoljašnjeg DSL-a prikazana je na sledećoj slici:



Slika 21. Arhitektura procesiranja spoljašnjeg DSLa

Dobijeni semantički model se može izvršiti ili se na osnovu njega može generisati kôd koji se odvojeno kompajlira i izvršava. Druga konstatacija je važna zbog toga što se, u slučaju validacije na osnovu semantičkog modela, može generisati npr. JavaScript validacija primenljiva na klijentskoj strani.

Često, međukorak između parsiranja i semantičkog modela predstavlja Sintaksno stablo, koje je usko povezano sa sintaksom DSL skripta.

U zavisnosti od formata i sintakse spoljašnjeg DSL-a, primenjuju se različite tehnike parsiranja. Takve tehnike predstavljaju tzv. sintaksne analize. Pre obavljanja sintaksne analize, primenjuje se leksička analiza teksta – deljenje teksta u tokene.

U svrhu definisanja validacionih ograničenja, mogu se koristiti trenutno raspoloživi jezici, kao što su XML šema, jezik za definisanje podataka (eng. Data Definition Language – DDL) ili jezik za definisanje ograničenja objekata (eng. Object Constraint Language – OCL), zajedno sa unifikovanim jezikom za modeliranje (eng. Unified Modeling Language – UML).

4.5.1. XML šema

Ograničenja koja se definišu nad prostim tipovima XML šeme mogu se iskoristiti i prevesti u ograničenja semantičkog modela validacije, što je ilustrovano narednom tabelom:

XML šema ograničenje	Ograničenje semantičkog modela
`minInclusive` / `minExclusive`	`MinInclusiveConstraint` / `MinExclusiveConstraint`
`maxInclusive` / `maxExclusive`	`MaxInclusiveConstraint` / `MaxExclusiveConstraint`
`length` / `minLength` / `maxLength`	`LengthConstraint` / `MinLengthConstraint` / `MaxLengthConstraint`

`totalDigits / fractionDigits`	`TotalDigitsConstraint / ScaleConstraint`
`pattern`	`PatternMatchingConstraint`

Tabela 2. Pregled ograničenja definisanih XML šemom i njima pridruženih ograničenja semantičkog modela validacije

Parser XML šeme, koji treba da se implementira i koji izdvaja ograničenja, treba da podrži i prepoznaje samo deo sintakse koju XML šema nudi. Preciznije, od interesa su samo definicije tipova atributa, pa je prevođenje svih ograničenja trivijalno.

Model nad kojim se definiše semantički model validacije dobija se prethodnim generisanjem iz XML šeme, od strane jedne od postojećih biblioteka (JAXB, JiBX, itd.).

Generisana ograničenja služe za proveru samo osnovnih pravila, pošto se XML šemom ne može opisati poslovna logika. Stoga, celokupna validacija treba da se implementira slojevito, proverom prvo osnovnih pravila automatski generisanim iz XML šeme, a zatim i proverom preostalih poslovnih pravila.

4.5.2. DDL

Jezik za definiciju podataka relacionih baza se takođe može iskoristiti za generisanje osnovnih validacionih pravila. DDL-om je moguće definisati sva ograničenja koja su definisana XML šemom, osim ograničenja obrasca formatiranja.

Nedostatak ovog pristupa je taj što se DDL-om opisuje relacioni model, a ne objektni model kojim aplikacija manipuliše. Međutim, kako su alati za mapiranje relacionog modela u objektni i obrnuto, najčešće zastupljeni u modernim poslovnim aplikacijama (tzv. Object-Relational mapping – ORM), to se definicija mapiranja može iskoristiti za dobijanje informacija o objektnom modelu.

4.5.3. OCL

OCL jezik [35] predstavlja tekstualnu i deklarativnu notaciju kojom se definišu ograničenja ili pravila primenljiva nad UML modelom, npr. klasnim dijagramima. Zasnovan je na matematičkoj teoriji skupova i predikatskoj logici i njime se mogu predstaviti relevantne informacije o sistemu koji se modeluje i koje se inače ne mogu predstaviti pomoću dijagrama. Kombinacijom UML-a i OCL-a može se opisati precizan model koji uključuje detaljne odlike dizajna zajedno sa semantikom, poput invarijanti klasa, preduslova i postuslova [36]. OCL je standardizovan od strane Object Management Grupe.

OCL je deklarativni, strogo tipizirani jezik, kojim se opisuju pravila i čiji izrazi ne menjaju stanje sistema. Svi OCL izrazi definišu se nad odgovarajućim kontekstom. Njima je moguće definisati početne vrednosti atributa, pravila izvođenja, nove atribute i operacije koji nisu definisani UML dijagramom, itd. Poslednje pravilo, iz primera u poglavlju 4.2 “ukupan iznos jednak je proizvodu godišnje naknade i trajanja u godinama i ukoliko je definisan ne sme preći vrednost 999999.99”, u OCLu može da se zapiše na sledeći način:

```
context LandRental
inv:
 (not yearlyFee.oclIsUndefined() and not durationInYears.oclIsUndifined())
     implies  yearlyFee * durationInYears <= 999999.99
```

Prethodnim primerom definisana je tzv. **invarijanta** – logički izraz koji mora biti zadovoljen od strane svih instanci tipa nad kojim je definisan. Kontekst u primeru predstavlja instanca klase LandRental.

Preduslovi i postuslovi takođe predstavljaju tipove ograničenja koji se mogu definisati OCL-om. Preduslov je logički izraz koji mora biti zadovoljen pre izvršavanja određene operacije (metode), dok je postuslov logički izraz koji mora biti zadovoljen nakon njenog izvršavanja.

Prevođenje OCL ograničenja u semantički model validacije je mehaničko i odnosi se samo na prevođenje invarijanti. Osnovni tipovi podataka definisani od strane OCL-a se mogu prevesti u tipove podataka u Java-i na sledeći način:

OCL tip	**Java tip**
Boolean	boolean
Integer	int
Real	double (float)
String	String
OclAny	Object
OclType	Class

Tabela 3. Tabela osnovnih tipova podataka definisanih OCL specifikacijom i njima odgovarajućih tipova u programskom jeziku Java

Prevođenje se može prilagoditi, tako što se tip `Real` može prevesti u klasu `BigDecimal`, po potrebi. Vrednost `OclVoid` se prevodi u `null` vrednost. Na sličan način se i tipovi kolekcija, definisani u OCL-u, mogu prevesti u odgovarajuće tipove u Java-i. Prevođenje operacija definisanih OCL-om kao i jezičkih kontrukcija je trivijalno. Posebnu pažnju potrebno je obratiti na logičke operatore i izraze, pošto OCL koristi trovalentnu logiku koja uključuje nedostajuću vrednost, stoga je potrebno u kodu izvršiti proveru na nedostajuću vrednost da ne bi dolazilo do ispaljivanja izuzetaka prilikom izvršavanja programa (npr. treba dodati `NotNullConstraint`, pre provere drugih ograničenja). Takođe, treba obratiti pažnju na relacion operator provere jednakosti. OCL-om definisano je samo poređenje po vrednosti, dok u Javi postoji i logičko i fizičko poređenje objekata. Iako za ovaj problem ne postoji univerzalno rešenje, najčešće se prevođenje izvršava tako što se za primitivne tipove proverava jednakost po vrednosti, dok se referentni tipovi porede primenom metode `equals`.

U [36], [37] i [38] se mogu naći nekoliko implementacija koje prevode OCL ograničenja u Java kôd, uglavnom prevodeći ih u aspekte AspectJ [39] jezika. Svaka od predloženih implementacija se bavi time u kojoj fazi izvršavanja treba dodati provere

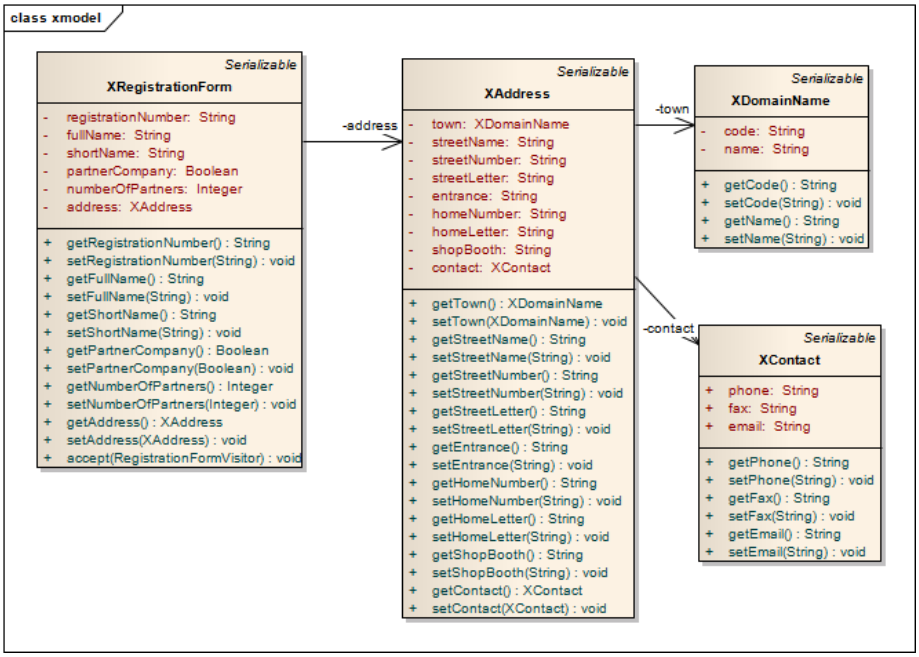
ograničenja: prilikom poziva metoda koje menjaju stanje objekta (tzv. setteri) ili nakon potpunog konstruisanja celog objekta. Takođe se razmatraju i strategije koje treba primeniti u slučaju kada ograničenje nije zadovoljeno. U kontekstu validacije, kao što je već navedeno, ograničenja se proveravaju prilikom same validacije, dok akcije predstavljaju prosto prijavljivanje greške.

Među raspoloživim alatima za definisanje OCL ograničenja nad različitim vrstama modela i njihovo prevođenje u Java kôd najpoznatiji je Dresden OCL [40]. Dresden OCL-om, izrazi se mogu definisati nad modelom definisanim UML-om, XML šemom ili XMI dokumentom, i zatim se mogu prevesti u AspectJ aspekt.

Snagu OCL jezika predstavlja njegova izražajnost i mogućnost zadavanja složenih provera. Nedostatak, u kontekstu prevođenja u semantički model validacije, predstavlja nemogućnost zadavanja koda greške, kao ni ostalih relevantnih informacija, npr. prioriteta grešaka, argumenata validacionih poruka, itd., stoga je neophodno izvršiti njegovu dopunu i prilagođavanje.

5 Detalji realizacije generatora formi

Tradicionalno, najviši sloj veb aplikacija implementira se korišćenjem Model-View-Controller (MVC) obrasca. U njemu, pogled (eng. view) predstavlja tekuću stranicu koju korisnik vidi i na kojoj se prikazuju podaci iz modela. Korisničke interakcije na strani prihvata i obrađuje kontroler, koji zatim intereaguje sa poslovnim objektima i u zavisnosti od akcije, prosleđuje odgovarajući model pogledu [41]. Pogled se zatim iznova prikazuje korisniku. Na ovaj način uloge svake od komponenti obrasca su jasno razdvojene: model je nosilac informacija koje se prikazuju na pogledu, dok kontroler upravlja tokom procesa i snadbeva pogled modelom. Generator formi primenjen je na veb aplikaciji koja koristi Spring MVC [42] kao veb biblioteku, Orbeon [43] kao XForms procesor i Saxon [44] kao XSLT 2.0 processor. Za generisanje klasa koje odgovaraju XML modelu forme, kao i za izvršavanje transformacija instanci modela u XML model i obrnuto, korišćena je JiBX [45] biblioteka. Na primer, objektni Java model za XML dokument koji predstavlja instancu modela registracione prijave iz poglavlja 3.1, prikazan je na *Slika 22*



Slika 22. Dijagram klasa modela registracione prijave

Razdvojenost modela od kontrola korisničkog interfejsa u XForms jeziku, omogućava automatizaciju generisanja formi. Praznina između veb dizajnera i programera može biti potpuno premošćena tako što se pravljenje dinamičkih formi u celosti može automatizovati na sledeći način: tokom analize problema, tim analitičara prvo definiše XML šemu kojom se opisuje XML model forme. U ovoj fazi definišu se različita poslovna pravila i izračunavanja, kako na

klijentskoj strani (na samoj formi), tako i na serverskoj. Forma napravljena na ovaj način može biti dodatno prilagođena i vizuelno poboljšana od strane veb dizajnera, preraspoređivanjem kontrola korisničkog interfejsa i definisanjem različitih CSS stilova. Kao takva, forma dobija svoj završni izgled.

Definisanje XML šeme daje trostruku dobit: njome se definišu izgled, tipovi i ograničenja nad instancom modela koja se prikazuje na formi, na osnovu nje se na serverskoj strani generišu Javine [46] klase koje odgovaraju XML modelu, a takođe se i njenim parsiranjem dobija osnovni sloj serverske validacije opisane u poglavlju 4.5.1.

Inicijalizacija instance modela na formi postiže se serijalizacijom (eng. *marshalling*) instance Javine klase u XML, koja se zatim umeće XSLT transformacijom na mesto prethodno definisane instance modela (definisane od strane analitičara). XML dokument, koji opisuje instancu modela, umeće se u rezultujući dokument korišćenjem XSLT funkcije `document(object)`, koja kao argument prihvata URL do dokumenta koji se umeće. Javina standardna biblioteka definiše interfejs `javax.xml.transform.URIResolver`, čija se metoda `resolve(String href, String base)` poziva onda kada XSLT procesor naiđe na funkciju `document(object)`, tokom procesiranja XSLT transformacije. Kako Spring-ov kontroler na pogled postavlja model u obliku mape (kataloga), u kojoj je ključ naziv instance u modelu a vrednost sama instanca, to se XSLT procesoru mora postaviti specijalna implementacija `URIResolver`-a, koja je zadužena za serijalizovanje objekta u XML dokument i njegovo uvoženje. Odgovarajući deo koda bi mogao da izgleda ovako:

```
if (source instanceof Source) {
      return (Source) source;
} else if (source instanceof Document) {
      return new DOMSource(((Document) source).getDocumentElement());
} else if (source instanceof Node) {
      return new DOMSource((Node) source);
} else if (source instanceof Reader) {
      return new StreamSource((Reader) source);
} else if (source instanceof InputStream) {
      return new StreamSource((InputStream) source);
} else if (source instanceof Resource) {
      Resource resource = (Resource) source;
      return new StreamSource(resource.getInputStream(), resource
                              .getURI().toASCIIString());
} else {
      for (Marshaller marshaller : marshallers) {
            if (marshaller.supports(source.getClass())) {
                  ByteArrayOutputStream baos = new ByteArrayOutputStream();
                  marshaller.marshal(source, new StreamResult(baos));
                  return new StreamSource(new
                                 ByteArrayInputStream(baos.toByteArray()));
            }
      }
      throw new IllegalArgumentException("Value '" + source
                              + "' cannot be converted to XML Source");
}
```

Standardni izvori XML dokumenata se obrađuju na jednostavan način. U ostalim slučajevima, proverava se da li se među JiBX klasama (klase koje prevode instance Java klasa u XML dokumenta – tzv. *maršalerima*) postoji neka koja podržava konverziju određene instance modela u XML dokument. Ako takav maršaler postoji, pomoću njega se dati objekat serijalizuje i zatim umeće u rezultujući dokument. U suprotnom se prijavljuje greška. U opštem slučaju moguće je definisati maršaler koji prevodi instance modela bez obzira na tip (klasu), prevodeći svaki atribut klase u XML element sa istim imenom, dok se specijalnim slučajevima, radi bolje kontrole, mogu definisati prilagođeni maršaleri. Prethodna konstatacija je važna zbog toga što navedena implementacija `URIResolver`-a treba samo da se proširi definicijom novog maršalera, ukoliko je potrebno, bez ikakvog menjanja kôda.

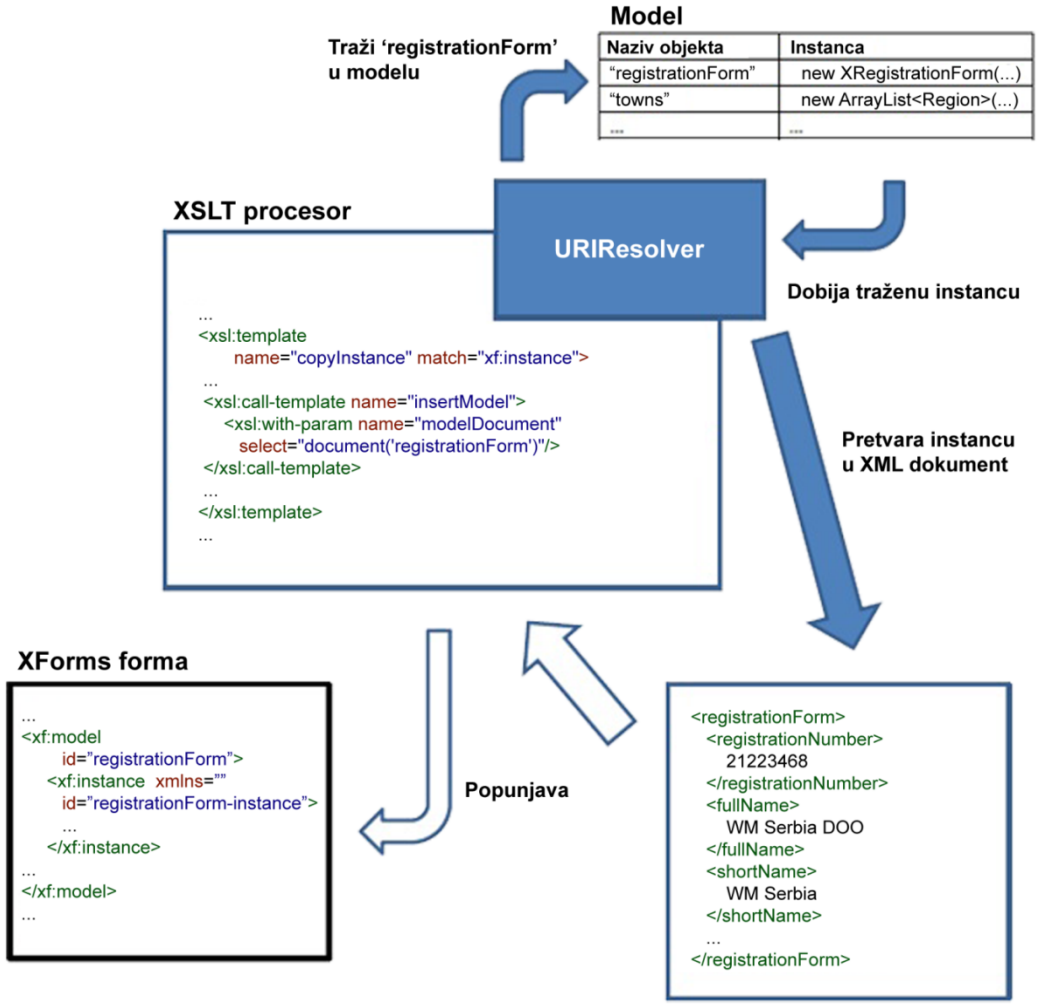
U ovom momentu važno je napomenuti da su prilikom razvijanja XForms forme uvedene određene konvencije imenovanja, radi lakše automatizacije. Na primer, po konvenciji, identifikatori instanci modela završavaju se sa *'-instance'* (npr. instanca registracione forme ima identifikator 'registrationForm-instance'). Imajući ovo na umu, deo XSLT transformacije kojim se umeće instanca modela na formu, definisan je na sledeći način:

```
<xsl:param name="modelDocumentsNames" as="xs:string*" />
<xsl:template name="copyAny" match="node()|@*">
   <xsl:copy>
      <xsl:apply-templates select="node()|@*" />
   </xsl:copy>
</xsl:template>

<xsl:template name="copyInstance" match="xf:instance">
   <xsl:variable name="instanceName"
                 select="substring-before(@id, '-instance')" />
   <xsl:choose>
      <xsl:when test="exists(index-of($modelDocumentsNames, $instanceName))">
         <xsl:call-template name="insertModel">
            <xsl:with-param name="modelDocument"
                            select="document($instanceName)"/>
         </xsl:call-template>
      </xsl:when>
      <xsl:otherwise>
         <xsl:copy-of select="." />
      </xsl:otherwise>
   </xsl:choose>
</xsl:template>

<xsl:template name="insertModel">
   <xsl:param name="modelDocument" />
   <xsl:copy>
      <xsl:apply-templates select="@*" />
      <xsl:copy-of select="$modelDocument" />
   </xsl:copy>
</xsl:template>
```

Elementom `xsl:param` definisan je globalni parametar transformacije sa imenom *'modelDocumentNames'*, čiju vrednost predstavlja niz identifikatora instanci modela koji treba da se zamene XSLT transformacijom forme. U primeru registracione prijave, instanca modela sa nazivom 'registrationForm-instance' je dinamička i trebalo bi da se zameni validnom instancom pomoću transformacije. Sa druge strane, instanca koja sadrži informacije o gradovima i koja služi samo kao referentni podatak, treba da ostane kakva jeste. Šablon 'copyInstance' primenjuje se nad instacama modela (`xf:instance`), uklanja '-instance' sufiks i proverava da li je takva instanca označena za zamenu. Ako jeste, pomoću već opisane funkcije `document(object)` i registrovanog `URIResolver`-a preuzima se XML dokument sa instancom modela i njime se zamenjuje već postojeća instanca. U suprotnom, instanca ostaje kakva jeste. Zamena instance izvršava se pomoću XSLT funkcija `xsl:copy` i `xsl:copy-of`. Prva daje veću kontrolu prilikom kopiranja, ali zahteva i precizniju definiciju pošto kopira sadžaj tekućeg čvora bez kopiranja atributa i dece čvorova, dok se drugom funkcijom kopira sve [47]. Tok procesiranja i umetanja modela na formu prikazan je na *Slika **23***.



Slika 23. Tok procesiranja XSLT transformacije i umetanja intance modela u XForms formu

U slučaju kada je na formi potrebno, osim podataka, prikazati i greške koje su nastale nakon neuspešne validacije, situacija se znatno komplikuje. Prikupljene greške nalaze se u instanci klase `ValidationContext`, opisane u poglavlju 4.2., koja se na sličan način kao i instanca modela serijalizuje u XML dokument i kao takva se daje na raspolaganje XSLT transformaciji. Sledećom XML šemom definisani su XML dokumenti koji sadrže informacije o greškama. Šema nije u skladu sa semantičkim modelom validacije "1-1", zbog toga što generator formi može da se koristi odvojeno od njega:

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
                  xmlns:tns="http://www.pras.org/Validation/Errors"
                  targetNamespace="http://www.pras.org/Validation/Errors"
                  elementFormDefault="qualified">

      <xsd:complexType name="ErrorsType">
            <xsd:choice minOccurs="1" maxOccurs="unbounded">
                  <xsd:element name="error" type="tns:ErrorType" />
                  <xsd:element name="errorGroup" type="tns:ErrorGroupType" />
            </xsd:choice>
      </xsd:complexType>

      <xsd:complexType name="BaseErrorType">
            <xsd:all>
                  <xsd:element name="code" type="tns:ErrorCodeType" />
                  <xsd:element name="message" type="tns:ErrorMessageType" />
                  <xsd:element name="level" type="tns:ErrorLevelType" />
                  <xsd:element name="namespace" type="xsd:string"
                                                minOccurs="0" />
                  <xsd:element name="details" type="tns:ErrorDetailsType"
                                               minOccurs="0"/>
            </xsd:all>
      </xsd:complexType>

      <xsd:complexType name="ErrorGroupType">
            <xsd:sequence>
                  <xsd:element name="error" type="tns:BaseErrorType"
                                             maxOccurs="unbounded" />
            </xsd:sequence>
            <xsd:attribute name="path" type="tns:XpathType" use="required"/>
      </xsd:complexType>

      <xsd:complexType name="ErrorType">
            <xsd:complexContent>
                  <xsd:extension base="tns:BaseErrorType">
                        <xsd:attribute name="path" type="tns:XpathType"
                                       use="required"/>
                  </xsd:extension>
            </xsd:complexContent>
      </xsd:complexType>

      <xsd:simpleType name="XpathType">
            <xsd:restriction base="xsd:string" />
      </xsd:simpleType>
```

```
	<xsd:simpleType name="ErrorCodeType">
		<xsd:restriction base="xsd:string" />
	</xsd:simpleType>

	<xsd:simpleType name="ErrorMessageType">
		<xsd:restriction base="xsd:string" />
	</xsd:simpleType>

	<xsd:simpleType name="ErrorLevelType">
		<xsd:restriction base="xsd:string" >
			<xsd:enumeration value="WARN" />
			<xsd:enumeration value="CRITICAL" />
			<xsd:enumeration value="FATAL" />
		</xsd:restriction>
	</xsd:simpleType>

	<xsd:complexType name="ErrorDetailsType">
		<xsd:sequence minOccurs="0" maxOccurs="unbounded">
			<xsd:any processContents="lax" />
		</xsd:sequence>
	</xsd:complexType>

	<xsd:element name="errors" type="tns:ErrorsType" >
		<xsd:unique name="uniquePath">
			<xsd:selector xpath="./tns:error | ./tns:errorGroup" />
			<xsd:field xpath="@path" />
		</xsd:unique>
	</xsd:element>
</xsd:schema>
```

Šemom su definisana dva glavna tipa elemenata: osnovni (`ErrorType`) i grupišući (`ErrorGroupType`). Svaki od tipova definiše podelemente čije je značenje uglavnom objašnjeno u semantičkom modelu validacije, kao što su kôd, poruka i nivo greške, kao i putanja do čvora modela koji je nevalidan. Novinu predstavljaju detalji greške, koji mogu da se iskoriste onda kada je potrebno uz grešku prikazati detaljnije informacije. Grupišući tip je zgodan za korišćenje u slučaju kada se uz jedan čvor modela prijavljuje više grešaka, čime se eliminiše ponavljanje putanje do čvora modela. Primer XML dokumenta, koji sadrži informacije o greškama validacije, izgleda kao u sledećem prikazu:

```
<?xml version="1.0" encoding="UTF-8"?>
<errors xmlns="http://www.pras.org/Validation/Errors">
	<error path="registrationNumber">
		<code>error.not_unique</code>
		<message>Firma sa unetim registracionim brojem je već zavedena
			kao poreski obveznik</message>
		<level>CRITICAL</level>
	</error>

	<error path="address.town.name">
		<code>error.missing</code>
		<message>Adresa sedišta firme je obavezna</message>
		<level>CRITICAL</level>
	</error>
```

```
</errors>
```

Informacije o greškama je tada moguće prikazati tako što se XSLT transformacijom svaka greška uparuje po XPath lokaciji sa odgovarajućim čvorom modela, odnosno kontrolom korisničkog interfejsa sa kojom je čvor povezan. Nad takvom kontrolom, greške mogu biti indukovane na tri načina:

1. Prosleđivanjem `xforms-invalid` događaja (ili nekog drugog, za te svrhe prilagođenog događaja), koji bi zatim bio obrađen pomoću obrađivača događaja prethodno definisanog od strane analitičara. Obrađivači bi, u zavisnosti od tipa greške na određenom čvoru, prikazali lokalizovane poruke o grešci. Prednosti ovakvog pristupa su sledeće: prosleđivanjem `xforms-invalid` događaja, XForms procesoru se naglašava koji čvor modela je nevalidan; takođe, na ovaj način se omogućava analitičarima da, u fazi razvoja na osnovu greške, preciziraju lokalizovane poruke i odgovarajuće akcije. Tenutno je ovakav pristup neprimenljiv zbog nedostataka DOM2 specifikacije događaja i nemogućnosti prosleđivanja argumenata (konteksta) događaju (u ovom slučaju argumenata i koda greške). Iz tog razloga, ne postoji način da se obrađivaču događaja prosledi informacija o kakvoj grešci je reč, već samo to da se greška desila. Postojeći nedostatak je ispravljen u najavljenoj DOM3 specifikaciji događaja. Takođe, u najavljenoj XForms 2.0 specifikaciji u planu je dodavanje kontekstnih informacija UI događajima (http://www.w3.org/MarkUp/Forms/wiki/XForms_2.0), pa će tada i predloženi pristup biti primenljiv.
2. Slično prethodnom, prosleđivanjem `xforms-invalid` događaja za svaku grešku koja se desila, sa tom razlikom što se XSLT transformacijom dodaje `xf:switch` element, kojim se prikazuju/skrivaju različiti delovi forme. Identifikatori takvih elemenata jednaki su kodovima grešaka. Svaki identifikator jedinstveno određuje deo forme koji treba da se prikaže. Mana ovakvog pristupa je ta, što se, u zavisnosti od broja grešaka, opterećuje XForms procesor i komplikuje sama forma.
3. Generisanjem odgovarajućeg XHTML fragmenta XSLT transformacijom, kojim bi se na osnovu koda (ili kodova) greške prikazala porukom (porukama) o grešci. Ovakav pristup ne opterećuje XForms procesor i ne komplikuje formu, ali je uz generisani fragment potrebno dodati i atribute stavki modela kojima bi se procesoru naznačili nevalidni čvorovi modela. Time bi se prouzrokovalo prosleđivanje `xforms-invalid` događaja nad povezanim kontrolama korisničkog interfejsa.

U slučaju kada postoje validacione greške, XML dokument sa instancama modela, prosleđen kao argument XSLT transformacije, sadrži instancu modela forme i instancu sa greškama, obe unutar `<bind>` elementa. Ovakav model je zgodniji za obradu od strane procesora, nego u slučaju da su instance definisane odvojeno. Pri tom, elementi grešaka pripadaju imenovanom prostoru sa prefiksom ’err’; instanca modela koja se šalje sa forme ne pripada ni jednom imenovanom prostoru, dok ostali elementi pripadaju imenovanom prostoru sa

prefiksom ’bnd’. Dokument sa greškama se transformacijom umeće kao jedna od instanci modela forme. Deo XSLT transformacije, koji obavlja umetanje instance modela grešaka i koji se poziva onda kada postoje greške prilikom izvršavanja serverske validacije, ima XML dokument sa instancom modela kao parametar i podeljen je na tri dela:

Prvi deo šablona sa imenom ’insertErrors’ dodaje formi instancu ’error-instance’. XForms instance ne smeju da pripadaju ni jednom imenovanom prostoru, a kako je u XHTML formama obično zadat podrazumevani imenovani prostor (`xmlns="http://www.w3.org/1999/xhtml"`), to je uz instancu modela neophodno navesti `xmlns=""`.

```
<xf:instance xmlns="" id="error-instance">
   <xsl:apply-templates select="$modelDocument/bnd:bind/err:errors"
                        mode="copy-errors" >
      <xsl:with-param name="modelRootName"
                      select="name($modelDocument/bnd:bind/bnd:model/*)" />
   </xsl:apply-templates>

</xf:instance>
```

Iz ’insertErrors’ šablona, primenjuju se naredna dva. Prvi umeće koreni ’errors’ element, bez imenovanog prostora (tj. sa lokalnim imenom), primenom funkcije local-name(). Drugi generiše ’kontejner’ elemente (grupe grešaka ili greške sa definisanim ’path’ atributom, čija je vrednost prepravljana tako da sadrži apsolutnu putanju do čvora modela koji je nevalidan, radi lakšeg uparivanja sa vrednošću ’ref’ atributa odgovarajuće kontrole korisničkog interfejsa). Sadržaj kontejnera se zatim kopira primenom ’copyError’ šablona.

```
<xsl:template name="copyErrors" match="err:errors" mode="copy-errors" >
   <xsl:param name="modelRootName" required="yes" />
   <xsl:element name="{local-name()}">
      <xsl:apply-templates mode="copy-errors">
         <xsl:with-param name="modelRootName" select="$modelRootName" />
      </xsl:apply-templates>
   </xsl:element>
</xsl:template>

<xsl:template name="copyErrorWithPath" match="err:errorGroup |
                                                 err:error[@path]"
              mode="copy-errors">
   <xsl:param name="modelRootName" required="yes" />
   <xsl:element name="{local-name()}">
      <xsl:attribute name="path"
                     select="xslext:getAbsolutePath($modelRootName, @path)" />
      <xsl:apply-templates mode="copy-errors" />
   </xsl:element>
</xsl:template>

<xsl:template name="copyError" match="err:*" mode="copy-errors">
```

```
   <xsl:element name="{local-name()}">
      <xsl:copy-of select="@*" />
      <xsl:apply-templates mode="copy-errors" />
   </xsl:element>

</xsl:template>
```

Izvršavanjem transformacije dobija se instanca sa greškama, slična polaznom XML dokumentu sa greškama, iz kojeg je izbačen imenovani prostor i putanje zamenjene apsolutnim.

Drugi deo šablona odnosi se na obaveštavanje XForms procesora o kontrolama korisničkog interfejsa koje su povezane sa nevalidnim čvorovima modela. To se postiže dodavanjem atributa stavki modela svakom čvoru nad kojim je prijavljena greška. Nevalidni čvorovi modela uparuju se sa greškama po apsolutnoj putanji.

```
<xsl:for-each select="$modelDocument/bnd:bind/err:errors/err:errorGroup
                      |
                      $modelDocument/bnd:bind/err:errors/err:error[@path]">
   <xsl:variable name="modelRootName" as="xs:string"
                 select="name($modelDocument/bnd:bind/bnd:model/*)" />
   <xsl:variable
            name="absolutePath" as="xs:string"
            select="xslext:getModelRootAbsolutePath($modelRootName, @path)" />
   <xf:bind nodeset="{$absolutePath}"
            constraint="count(instance('error-instance')
                              /error[@path='{$absolutePath}']) = 0
                       and
                       count(instance('error-instance')
                              /errorGroup[@path='{$absolutePath}']) = 0" />

</xsl:for-each>
```

Transformacijom se za svaki grupišući element greške i svaki osnovni tip greške, koji ima definisan ’path’ atribut, dodaje po jedan atribut stavki modela, sa sledećim ograničenjem:

```
count(instance('error-instance')/error[@path='{$absolutePath}']) = 0
and
count(instance('error-instance')/errorGroup[@path='{$absolutePath}'])= 0
```

U ograničenju se proverava da li postoje prijavljene greške nad specifinčnim skupom čvorova. Pomoću dodatih atributa stavki modela, nevalidni čvorovi modela se prijavljuju XForms procesoru. Za takve čvorove, procesor će ispaliti xforms-invalid događaj nad kontrolama korisničkog interfejsa koji su sa njima povezani. Na primer, ako postoje prijavljene greške nad registracionim brojem i punim nazivom firme, tada će transformacijom biti umetnute sledeće dve stavke atributa modela:

```
<xf:bind nodeset="/registrationForm/registrationNumber"
         constraint="
```

```
    count(instance('error-instance')
            /error[@path='/registrationForm/registrationNumber']) = 0
    and
    count(instance('error-instance')
            /errorGroup[@path='/registrationForm/registrationNumber']) = 0" />

<xf:bind nodeset="/registrationForm/fullName"
          constraint="
    count(instance('error-instance')
            /error[@path='/registrationForm/fullName']) = 0
    and
    count(instance('error-instance')
          /errorGroup[@path='/registrationForm/fullName']) = 0" />
```

Atributi stavki modela se izračunavaju i proveravaju prilikom svake promene stanja forme. Kako je slanje podataka sa forme onemogućeno dokle god je forma nevalidna, neophodno je obrisati instancu sa greškama da bi ograničenja definisana dodatim stavkama atributa modela bila zadovoljena. To se postiže dodavanjem obrađivača `xforms-submit` događaja nad `xf:submission` kontrolom:

```
  <xf:delete ev:event="xforms-submit" nodeset="instance(error-instance)/*" />
```

Obrađivač uklanja instancu sa greškama iz modela zbog čega se logički izraz u definiciji ograničenja evaulira u vrednost "tačno". Na taj način se forma označava validnom i omogućava se njeno slanje.

Treći deo šablona odnosi se na samo prikazivanje grešaka. Šablonom se prepisuju sve kontrole korisničkog interfejsa i proverava se da li postoje greške prijavljene nad čvorom modela povezanim sa tekućom kontrolom preko 'ref' atributa. Poklapanje se vrši upoređivanjem vrednosti 'ref' atributa kontrole sa vrednošću 'path' atributa `error` ili `errorGroup` elementa inastance grešaka. Ako takva greška postoji, XHTML fragment se umeće transformacijom (u ovom slučaju `<div>` element sa određenim stilom). Pomoću `xf:repeat` XForms elementa, prolazi se kroz čvorove instance sa greškama koje odgovaraju tekućoj kontroli korisničkog interfejsa i za svaki od njih prikazuje se sadržaj message elementa, koji predstavlja lokalizovanu poruku o grešci.

```
<xsl:template name="copyControl" match="xf:input | xf:textArea | xf:range |
                                        xf:select | xf:select1 | xf:secret">
  <xsl:call-template name="copyAny" />
  <xsl:variable name="path" as="xs:string">
     <xsl:value-of separator="/">
        <xsl:sequence select="ancestor::xf:group/@ref" />
        <xsl:sequence select="@ref" />
     </xsl:value-of>
  </xsl:variable>
  <div class="msg-error" xmlns="http://www.w3.org/1999/xhtml">
     <xf:repeat nodeset="instance('error-instance')/error[@path='{$path}']
```

```
                        |
                        instance('error-instance')
                             /errorGroup[@path='{$path}']/error">
        <xf:output ref="message" />
        <br xmlns="http://www.w3.org/1999/xhtml" />
      </xf:repeat>
   </div>

</xsl:template>
```

Putanja kontrole do čvora modela se u navedenom šablonu definiše `path` promenljivom. Prilikom određivanja putanje, neophodno je proveriti da li kontrola pripada grupišućem `xf:group` elementu. U tom slučaju apsolutna putanja do čvora dobija se dodavanjem njegove relativne putanje na ’ref’ atribut `xf:group` elementa. Vrednost atributa ’nodeset’ `xf:repeat` elementa predstavlja XPath izraz koji kao rezultat daje skup čvorova kroz koji se iterira. Kontekst svake iteracije predstavlja tekući čvor (`error` element). Telo `xf:repeat` elementa se tada izvršava za svaki čvor ponaosob. Elementom `<xf:output ref="message" />` se prosto prikazuje tekst `message` elementa tekućeg čvora.

Na prethodno opisan način zaokružen je automatizovan proces generisanja dinamičkih formi, od inicijalizacije forme, do prikaza grešaka validacije prilikom slanja podataka sa forme. Definisanje novih formi ne zahteva nikakvu promenu, niti koda niti XSLT transformacija, pošto se na osnovu njih ceo proces umetanja modela i informacija o greškama i obaveštavanje XForms procesora o greškama izvršava automatski, bez obzira na instancu modela koja se sa forme šalje.

6 Zaključak

U poslovnim aplikacijama, pravljenje formi za unos podataka i implementacija poslovne logike, u vidu izračunavanja i provere niza validacionih pravila, predstavlja njihov neizostavan deo, koji je po pravilu vrlo komplikovan. Baš iz tog razloga, proces automatizacije validacije i generisanja formi važan je jer smanjuje vreme kompletnog ciklusa pravljenja formi, smanjuje vreme potrebno za implementaciju validacije i garantuje konzistentnost i formalnu korektnost dizajna poslovnih formi i specifikacije poslovnih pravila. U ovoj tezi, detaljno je obrađen problem validacije i predložen je model koji omogućava njenu automatizaciju. Takođe je u radu opisan generator formi koji formu pravi dinamičkom, bez obzira na njen izgled i model koji se na njoj prikazuje.

Rad prati i praktični deo u vidu validacione biblioteke napisane u programskom jeziku Java, koji se sastoji od semantičkog modela validacije, semantičkog modela sistema za proizvodnju validacionih pravila, kao i unutrašnjeg DSL-a za definisanje ograničenja. Svrha unutrašnjeg DSL-a je da omogući lak i intuitivan način zadavanja ograničenja i akcija koje se izvršavaju onda kada ograničenja nisu zadovoljena. U radu su takođe iznete prednosti i mane nekih od raspoloživih jezika za definisanje ograničenja: XML šeme, jezika za definisanje podataka (DDL) kao i jezika za definisanje ograničenja (OCL). Za pomenute jezike su razvijeni i parseri koji prevode ograničenja zapisana u njima u semantički model validacije, koji se zatim interpretira i pomoću kojeg se izvršava validacija.

Generator formi, predstavljen i opisan u ovom radu, zasnovan je na dokazanim i preporučenim XML-zasnovanim tehnologijama: XML šemi, XSLT transformacijama, XPath i XForms jezicima. Podela forme na različite slojeve u XForms jeziku, i pre svega jasno razdvajanje modela podataka od kontrola korisničkog interfejsa, omogućava automatizaciju celokupnog procesa generisanja formi. Model formi zadat je u obliku XML dokumenta, koji se na serverskoj strani nakon slanja prevodi u objektni model, a zatim se u povratku ponovo prevodi u XML. Iako celokupan proces zahteva dodatno vreme za procesiranje XML-a, XML biblioteke i XSLT procesori su poslednjih godina toliko napredovali da je to vreme neosetno. Predložen generator formi testiran je praktično nad različitim vrstama formi. Jedna od tih formi je i forma za registrovanje poreskih obveznika Ona predstavlja idealan primer za testiranje i prezentaciju zbog toga što se sastoji od osam celina i preko dve stotine ulaznih polja. Za celokupan proces, koji uključuje: slanje podataka sa forme, vraćanje nazad i prikaz validacionih grešaka, potrebno je oko tri sekunde, što je u skladu sa preporukama iskoristivosti veb aplikacija. Rezultati bi bili i bolji da se dodatno vreme ne troši za procesiranje forme od strane Orbeon XForms procesora, koji je konfigurisan kao zasebna veb aplikacija. Procesor ugrađen u samu aplikaciju bi mogao da se koristi za dostizanje boljih rezultata, ili još bolje, procesor ugrađen u sam veb pretraživač.

U ovom radu takođe smo predstavili različite strategije za prikazivanje lokalizovanih poruka o greškama, sakupljenih prilikom neuspele serverske validacije. Nedostaci XPath 1.0 i

DOM2 specifikacije događaja, nad kojima je XForms jezik izgrađen, predstavljaju glavnu prepreku u implementaciji generatora formi. To je takođe bio i jedan od razloga, između ostalih, zašto smo izabrali manje intutitivnu strategiju za prikaz validacionih grešaka. XSLT transformacije, koje se koriste u generatoru, bi trebalo dodatno unaprediti onda kada sledeća XForms specifikacija postane zvanična preporuka.

6.1. Dalje istraživanje

Dopuna sintakse OCL jezika, ili još bolje, definisanje prilagođenog jezika koji bi bio pogodan za zapis ograničenja i detalja o greškama koje se prijavljuju onda kada ograničenja nisu zadovoljena, predstavilo bi glavnu dobit i omogućilo automatizaciju validacije u potpunosti. Pomoću prilagođenog jezika, koji bi bio razumljiv i ekspertima u domenu za koji se određena aplikacija razvija i analitičarima i programerima, moguće je definisati validaciona pravila koja se zatim pomoću parsera prevode u semantički model validacije. Jezik bi trebalo da omogući lakšu komunikaciju između klijenata i programera, koja najčešće predstavlja usko grlo u razvoju aplikacija. Analitičari u fazi analize slučajeva korišćenja mogu da definišu validaciona pravila i detalje grešaka: lokalizovane poruke o grešci, težinu greške, itd. Automatizacija se postiže tako što se validaciona pravila zapisana u tekstualnom formatu prosto ‘učitaju’ i interpretiraju, bez menjanja koda.

Dalje istraživanje obuhvata i uključivanje opisa poslovnih procesa uz definiciju forme, pomoću XML jezika za definiciju procesa (XML Process Definition Language – XPDL [48]), i opisa autorizacije pomoću ARBAC02 [49] modela. Time bi se zaokružio i automatizovao ceo poslovni proces, od definicije modela i formi, pa do akcija, poslovnih pravila i validacije.

7 Literatura

[1] *M. Fowler, "Patterns of Enterprise Application Architecture", Addison-Wesley, 2003*

[2] *Brown et al., "Enterprise Java Programming with IBM Websphere", Addison Wesley, 2001*

[3] *Java Platform Micro Edition, J2ME at http://www.oracle.com/technetwork/java/javame/index.html*

[4] *XHTML 1.0: The Extensible Hyper Text Markup Language W3C Recommendation at http://www.w3.org/TR/xhtml1/, 2000.*

[5] *A. Girgensohn and A. Lee. Seamless integration of interactive forms into the Web Computer Networks and ISDN Systems, 1997, Vol. 29, Pages 1531–1542.*

[6] *Java Platform Enterprise Edition – J2EE, http://download.oracle.com/javaee/*

[7] *Oracle Designer, http://www.oracle.com/technetwork/developer-tools/designer/overview/index.html*

[8] *R. Gitzel, A. Korthaus and M. Schader. Using established Web Engineering knowledge in model–driven approaches Science of Computer Programming, 2007, Vol. 66, Pages 105–124.*

[9] *R. Glass. Some heresy regarding Software Engineering IEEE Software, (JulyAugust) (2004).*

[10] *Y. Deshpande and S. Hansen. Web Engineering: Creating a discipline among disciplines IEEE MultiMedia (April) (2001) 82-87.*

[11] *Maverick MVC framework at http://mav.sourceforge.net/, (accessed June 2012).*

[12] *Apache Software Foundation: The Apache Cocoon Project at http://cocoon.apache.org/, (accessed June 2012).*

[13] *P. Fraternali and P. Paolini. Model–driven development of Web Applications: The autoweb system ACM Transactions on Information Systems 28 (4) (2000) 323-382.*

[14] *S. Mellor and M. Balcer. Executable UML – A Foundation for Model–Driven Architecture Addison-Wesley, 2002.*

[15] *A. Torres, R. Galante and M. S. Pimenta. Towards a UML profile for model-driven object-relational mapping 2009 XXIII Brazilian Symposium on Software Engineering.*

[16] *XForms 1.1 W3C Recommendation at http://www.w3.org/TR/xforms11/, 2009.*

[17] *XSLT Version 2.0 W3C Recommendation at http://www.w3.org/TR/xslt20, 1999.*

[18] *Oracle Forms at http://www.oracle.com/technetwork/developer-tools/forms/overview/index.html/, (accessed June 2012).*

[19] *M. Fowler, "Domain Specific Languages", Addison-Wesley Professional, 2010*

[20] *XML Schema W3C Recommendation at http://www.w3.org/TR/xmlschema-0/, 2004.*

[21] *XML Path Language (XPath) Version 1.0 W3C Recommendation at http://www.w3.org/TR/xpath/, 1999.*

[22] *XSLT Version 2.0 W3C Recommendation at http://www.w3.org/TR/xslt20, 1999.*

[23] *M. Honkala. Web User Interaction a Declarative Approach Based on XForms. Ph.D. thesis, Helsinki University of Technology, 2006*
[24] *M. Dubinko. XForms Essentials. O'Reilly, 2003.*
[25] *M. Honkala and P. Vuorimaa. XForms in X-Smiles. Journal of World Wide Web, Internet and Web Information Systems, Kluwer, 2001, Vol. 4, No. 3.*
[26] *XML Linking Language (XLink) Version 1.0 W3C Recommendation at http://www.w3.org/TR/xlink/, 2001*
[27] *Synchronized Multimedia Integration Language (SMIL 2.0) W3C Recommendation at http://www.w3.org/TR/SMIL2/, 2001.*
[28] *Scalable Vector Graphics (SVG) 1.1 W3C Recommendation at http://www.w3.org/TR/SVG/, 2011.*
[29] *T. V. Raman. XForms: XML Powered Web Forms. Addison Wesley, 2003.*
[30] *Apache Struts 2, http://struts.apache.org/2.2.1/index.html*
[31] *C. Forgy, "Rete: A Fast Algorithm for the Many Pattern / Many Object Pattern Match Problem", IEEE Computer Society Press, Los Alamitos, CA, USA*
[32] *S. Freeman, N. Pryce, "Evolving an Embedded Domain-Specific Language in Java", OOPSLA, 2006*
[33] *J. Kabanov, R. Raudjarv, "Embedded Typesafe Domain Specific Languages for Java", Proceedings of the 6th international symposium on Principles and practice of programming in Java, 2008*
[34] *JSR 303: Bean Validation, http://jcp.org/en/jsr/detail?id=303*
[35] *OMG, "UML 2.0 OCL Specification", Object Management Group, Oct. 2003.*
[36] *Y. Cheon, C. Avila, S. Roach, C. Munoz, "Checking Design Constraints at Run-time Using OCL and AspectJ", International Journal of Software Engineering, vol. 3, no. 1, pp. 5–28, Dec. 2009.*
[37] *Y. Cheon, C. Avila, "Automating Java Program Testing Using OCL and AspectJ", 7th International Conference on Information Technology, pp. 1020-1025, 2010.*
[38] *A. Hamie, "Translating the Object Constraint Language into the Java Modeling Language," Proceedings of the ACM Symposium on Applied Computing, Nicosia, Cyprus, March 14-17, 2004, pp. 1531–1535, 2004.*
[39] *AspectJ, http://www.eclipse.org/aspectj/*
[40] *Dresden OCL Toolkit: http://dresden-ocl.sourceforge.net*
[41] *R. Johnson. Expert One-on-One Design and Development Wrox Press, 2003.*
[42] *Spring framework at http://www.springsource.org/.*
[43] *Orbeon forms at http://www.orbeon.com/.*
[44] *Saxon at http://saxon.sourceforge.net/.*
[45] *JiBX at http://jibx.sourceforge.net/.*
[46] *J. Gosling, B. Joy, G. Steele, and G. Bracha. Java (TM) Language Specification, The (Java (Addison-Wesley)). Addison-Wesley Professional, 2005.*
[47] *D. Tidwell. XSLT, 2nd edition O'Reilly, 2008.*

[48] *XML Process Definition Language - XPDL at http://www.wfmc.org/xpdl.html.*

[49] *S. Oh et al. An Effective Role Administration Model Using Organization Structure. ACM Transactions on Information and System Security, Vol. 9, No. 2, May 2006, Pages 113–137.*