Univerzitet u Novom Sadu

Fakultet tehničkih nauka

Željko Vuković

Modelom vođena semantička integracija poslovnih aplikacija

Doktorska disertacija

Mentor:
prof. dr Gordana Milosavljević

Novi Sad, 2019.

УНИВЕРЗИТЕТ У НОВОМ САДУ ● **ФАКУЛТЕТ ТЕХНИЧКИХ НАУКА**
21000 НОВИ САД, Трг Доситеја Обрадовића 6

КЉУЧНА ДОКУМЕНТАЦИЈСКА ИНФОРМАЦИЈА

Редни број, **РБР**:		
Идентификациони број, **ИБР**:		
Тип документације, **ТД**:		Монографска публикација
Тип записа, **ТЗ**:		Текстуални штампани документ
Врста рада, **ВР**:		Докторска дисертација
Аутор, **АУ**:		Жељко Вуковић
Ментор, **МН**:		Др Гордана Милосављевић
Наслов рада, **НР**:		Моделом вођена семантичка интеграција пословних апликација
Језик публикације, **ЈП**:		Српски
Језик извода, **ЈИ**:		Српски
Земља публиковања, **ЗП**:		Република Србија
Уже географско подручје, **УГП**:		Војводина
Година, **ГО**:		2019.
Издавач, **ИЗ**:		Факултет техничких наука
Место и адреса, **МА**:		Нови Сад, Трг Доситеја Обрадовића 6
Физички опис рада, **ФО**: (поглавља/страна/ цитата/табела/слика/графика/прилога)		9/114/140/6/27/0/0
Научна област, **НО**:		Електротехничко и рачунарско инжењерство
Научна дисциплина, **НД**:		Примењене рачунарске науке и информатика
Предметна одредница/Кључне речи, **ПО**:		Интеграција пословних апликација, моделом управљан развој софтвера, семантички веб, језици специфични за домен
УДК		
Чува се, **ЧУ**:		Библиотека Факултета техничких наука, Трг Доситеја Обрадовића 6, 21000 Нови Сад
Важна напомена, **ВН**:		
Извод, **ИЗ**:		У склопу докторске дисертације извршено је истраживање везано за аутоматизацију интеграције пословних апликација. Приказани приступ комбинује модел структуре интерфејса апликација које се интегришу са формалним описом њихове семантике, датим у виду онтологије. На основу ових извора, обавља се аутоматско мапирање међу елементима интерфејса који се интегришу, као и детекција и разрешавање семантичких конфликата. Развијена је практична имплементација предложеног приступа, која је коришћена за верификацију теоријских разматрања, а укључује адаптиван радни оквир и језик специфичан за домен. Предложени приступ је верификован на два реална интеграциона сценарија и једним експериментом.
Датум прихватања теме, **ДП**:		28.2.2019.
Датум одбране, **ДО**:		
Чланови комисије, **КО**:	Председник:	Др Горан Сладић, ванредни професор
	Члан:	Др Владимир Димитриески, доцент
	Члан:	Др Владимир Вујовић, доцент
	Члан:	Др Игор Дејановић, ванредни професор
	Члан, ментор:	Др Гордана Милосављевић, ванредни професор

Потпис ментора

UNIVERSITY OF NOVI SAD ● **FACULTY OF TECHNICAL SCIENCES**
21000 NOVI SAD, Trg Dositeja Obradovića 6

KEY WORDS DOCUMENTATION

Accession number, **ANO**:		
Identification number, **INO**:		
Document type, **DT**:		Monographic publication
Type of record, **TR**:		Textual material
Contents code, **CC**:		PhD thesis
Author, **AU**:		Željko Vuković
Mentor, **MN**:		Gordana Milosavljević, PhD
Title, **TI**:		Model based semantic enterprise application integration
Language of text, **LT**:		Serbian
Language of abstract, **LA**:		Serbian
Country of publication, **CP**:		Republic of Serbia
Locality of publication, **LP**:		Vojvodina
Publication year, **PY**:		2019.
Publisher, **PB**:		Faculty of Technical Sciences
Publication place, **PP**:		Novi Sad, Trg Dositeja Obradovića 6
Physical description, **PD**: (chapters/pages/ref./tables/pictures/graphs/appendixes)		9/114/140/6/27/0/0
Scientific field, **SF**:		Electrical engineering and computing
Scientific discipline, **SD**:		Applied computer science and informatics
Subject/Key words, **S/KW**:		Enterprise application integration, model driven engineering, semantic web, domain specific languages
UC		
Holding data, **HD**:		Library of Faculty of Technical Sciences, Trg Dositeja Obradovića 6, 21000 Novi Sad
Note, **N**:		
Abstract, **AB**:		This thesis presents a research in the field of automation of enterprise application integration. The approach combines structural models of interfaces of the applications being integrated with a formal specification of their semantics, given in form of an ontology. Using information from these sources, automated interface mapping is performed, along with detection and resolution of semantic conflicts. A practical implementation of the presented approach was developed and used to verify theoretical considerations. The implementation includes an adaptive framework and a domain specific language. The proposed approach has been verified on two real-world integration scenarios and one experiment.
Accepted by the Scientific Board on, **ASB**:		
Defended on, **DE**:		
Defended Board, **DB**:	President:	Goran Sladić, PhD, Associate professor
	Member:	Vladimir Dimitrieski, PhD, Assistant professor
	Member:	Vladimir Vujović, PhD, Assistant professor
	Member:	Igor Dejanović, PhD, Associate professor
	Member, Mentor:	Gordana Milosavljević, PhD, Associate professor

Mentor's signature

Mojoj Smiljani.

Zahvaljujem se:

svojim roditeljima, na svim odricanjima, ljubavi i pažnji,

svojoj dragoj Smiljani i njenoj porodici,

svojim iskrenim prijateljima,

svojim kolegama,

PI Informatik GmbH iz Berlina na gostoprimstvu i pomoći,

komisiji na pažljivom čitanju i korisnim komentarima i

svim svojim profesorima, na svemu što su nas naučili i naveli nas da sami naučimo.

Posebnu zahvalnost dugujem:

dr Gordani Milosavljević, svojoj mentorki i

dr Nikoli Milanoviću, pokretaču istraživačkog projekta iz kog je potekla ova disertacija, koji je, kao gostujući profesor, bio i moj savetnik na doktorskim studijama.

Predgovor

Postepeno uvođenje računara u poslovanje, imalo je za posledicu da se softver koji je pokrivao određene aspekte poslovanja nabavljao ili razvijao u etapama. Neretko, pojedinačna softverska rešenja razvijana su bez uvida u ostala softverska rešenja koja su u tom trenutku bila u upotrebi u istoj kompaniji. Posledica ovoga je da su podaci, koji su bili neophodni za razne namene, morali biti uneti u više različitih aplikacija. Još važnije, ti podaci su morali biti i ažurirani na više mesta ukoliko bi došlo do promene. Na taj način dolazilo je do pojave onoga što će kasnije biti nazvano silosima informacija. Funkcionalno povezani podaci nalazili su se fizički na odvojenim sistemima.

Baze podataka su viđene kao rešenje ovog problema. Umesto da svaka aplikacija skladištiti podatke na sebi svojstven način, sve aplikacije mogu koristiti jednu bazu podataka. Međutim, u praksi ponekad nije moguće menjati način na koji aplikacije čuvaju podatke. Jedno rešenje za ovaj problem je i takozvani reinženjering. Umesto postojećih pojedinačnih aplikacija, izrađuju se nove aplikacije, dizajnirane tako da budu međusobno interoperabilne, ili pak jedna aplikacija, koja predstavlja integralni informacioni sistem. Nove aplikacije preuzimaju funkcionalnost starih, kao i njihove podatke. Osnovni nedostatak reinženjeringa je što se od kompanija zahteva da se odreknu postojećih rešenja u koje su uložena sredstva za nabavku i održavanje, na koje su korisnici navikli i, što je najvažnije, koja su dokazana u praksi.

Alternativa reinženjeringu je integracija poslovnih aplikacija. Cilj integracije je da se omogući da nezavisno razvijani programski paketi funkcionišu kao povezana celina. Integracija poslovnih aplikacija je zrela oblast, za koju postoje dostupni oprobani komercijalni alati i koja se intenzivno koristi u praksi. Izazove u oblasti integracije predstavljaju: heterogenost aplikacija, širina neophodnih znanja o raznim korišćenim tehnologijama za izradu aplikacija koje se integrišu, kao i o poslovnim domenima i procesima za koje se one koriste, dugotrajnost i visoka cena ovog procesa, kao i pojava tehničkih i semantičkih konflikata. Tehnički konflikti su posledica različitih načina na koje aplikacije skladište i razmenjuju podatke, dok su semantički konflikti posledica različitog značenja tih podataka u različitim aplikacijama.

U [48] data je formalna karakterizacija i specifikacija radnog okvira za razmenu konteksta (Context Interchange Framework, **COIN**), koji predstavlja posredničku strategiju za pristup podacima, u kojoj se semantički konflikti između heterogenih sistema otkrivaju i rešavaju upotrebom kontekstnog posrednika. Svrha posrednika je da poredi kontekste bilo koja dva sistema koji učestvuju u razmeni podataka. Korišćenjem formalizama uvedenih kroz COIN radni okvir semantika pojedinačnog konteksta se može koristiti za rezonovanje

o semantičkim razlikama heterogenog sistema u kom taj kontekst učestvuje. Pregled arhitektura i tehnologija za integraciju distribuiranih poslovnih aplikacija, uz prikaz njihovih prednosti i nedostataka dat je u [59]. U [80] dat je radni okvir, baziran na modelima, za analizu konflikata i kompoziciju na nivou komponenti. Jedna platforma i metodologija za integraciju bazirana na meta-modelima, definisana u okviru projekta Bizycle, predstavljena je u [76]. Ova platforma podržava usku saradnju inženjera koji razvijaju integraciono rešenje sa domenskim ekspertima. Podržana je poluautomatizovana analiza konflikata. Primer definisanja ontologija za komponente dostupne na mreži jezikom OWL prikazan je u [130]. Dodavanjem klasa postojećoj ontologiji integracija je implementirana u vremenskom opsegu od nekoliko sati. Pristup integraciji koji kombinuje korišćenje veb servisa i ontologija nazvan ODSOI (Ontology-Driven Service-Oriented Integration) dat je u [67]. Autori sugerišu pogodnu topologiju servisa i ontologija, uz viziju radnog okvira za integraciju. Još jedan pristup za integraciju poslovnih informacionih sistema, pod nazivom Highway, predstavljen je u [75]. U [5] razmatrana je detekcija semantičkih konflikata u veb servisima i servisno orijentisanim arhitekturama (SOA) uopšte, u slučajevima kada se razmenjuju heterogeni podaci. Metodologija integracije poslovnih aplikacija u [25] daje smernice uz isticanje raznih pogleda na interoperabilnost: poslovni, procesni, pogled u odnosu na ljudske resurse, tehnološki, pogled u odnosu na znanje i u odnosu na značenje. U [43] dat je predlog za integraciju informacija baziran na ontologijama, uz mapiranje lokalne ontologije na globalnu. Ontologije su korišćene za detekciju i rešavanje konflikata u [112], uz praktičnu implementaciju nazvanu CREAM (Conflict Resolution Environment for Autonomous Mediation). Doktorska teza [34] prikazuje radni okvir za integraciju heterogenih tehničkih prostora, zasnovan na principima razvoja softvera vođenog modelima (RSVM).

Iako su razni pristupi imali viziju pokrivanja svih potreba vezanih za informacionu podršku poslovnih subjekata, u praksi se i dalje javlja potreba za korišćenjem različitih softverskih rešenja usko specijalizovanih za određeni domen primene. Kako bi se izbeglo redundantno unošenje i ažuriranje podataka neophodnih u više ovakvih aplikacija, moguće je izvršiti njihovu integraciju. Za ručni postupak izrade integracionog rešenja postoji više zrelih metodologija i alata. Ipak, ručna integracija ostaje kompleksan, skup i dugotrajan proces, koji zahteva ekspertizu kako iz oblasti same integracije, tako i poznavanje svih aplikacija i tehnologija koji se integrišu, kao i dobro shvatanje poslovnog domena koje integrisano rešenje treba da podrži. Automatizacija procesa integracije može dovesti do skraćenja vremena neophodnog za razvoj integracionog rešenja, kao i do smanjenja troškova. Kako u ručnom, tako i u automatizovanom procesu integracije, značajan izazov predstavljaju tehnički i semantički konflikti. Tehnički konflikti se odnose na različite načine predstavljanja, čuvanja i prenošenja podataka u različitim sistemima. Semantički konflikti su posledica različitog značenja koje ti podaci mogu imati za različite aplikacije. Automatsko razrešavanje tehničkih konflikata nije trivijalno, ali je u velikoj meri rešeno. Automatsko rešavanje semantičkih konflikata je polje za koje postoji značajan obim istraživanja i za koje postoji prostor za nova rešenja. Na polju detekcije i razrešavanja semantičkih konflikata izdvajaju se rešenja koja za opis semantike sistema koji učestvuju u integraciji koriste tehnike semantičkog veba. Izuzetna širina mogućih tehnologija i procesa koji se mogu javiti u praksi dovela je do toga da rešenja koja pokušavaju da u startu pokriju sve moguće scenarije budu izuzetno kompleksna, što za posledicu ima nedostatak zrelih alata za njihovu praktičnu primenu. Jedan način za prevazilaženje ovog problema je istraživanje mogućnosti izrade fleksibilne i proširive arhitek-

ture, koja bi omogućila da se specifične potrebe određenih integracionih scenarija podmire namenski izrađenim komponentama. Ukoliko bi ove komponente bile takve da se mogu ponovo koristiti, rezultat bi bila progresivno lakša izrada narednih integracionih rešenja.

Cilj istraživanja je definisanje procesa i komponenti koje omogućuju delimičnu ili potpunu automatizaciju otkrivanja mapiranja između elemenata interfejsa aplikacija i podsistema koji su predmet integracije poslovnih aplikacija i delimičnu ili potpunu automatizaciju otkrivanja semantičkih konflikata nad uočenim mapiranjima. Rešenje omogućava mapiranje heterogenih i disparatnih oblika interfejsa, pri čemu u integraciji može učestvovati N [1..] interfejsa koji pripadaju M [1..] aplikacija (gde je $M>1 \vee N>1$).

Pregled sadržaja i strukture teze

Uvodno poglavlje daje opšti pregled polja i predmeta istraživanja. U ovom poglavlju izložene su hipoteze istraživanja. Naredna poglavlja su organizovana na sledeći način.

Poglavlje 2 daje teorijske osnove koje su neophodne za razumevanje oblasti integracije. U ovom poglavlju su definisane vrste integracije, uz pregled nekih mogućih kategorizacija po različitim osnovama: broju i tipu atomičkih delova koji se integrišu, sintaktičkom pristupu i semantičkom pristupu. S obzirom da rešenje predstavljeno u tezi koristi neke od principa modelom upravljanog razvoja softvera, dat je kratak opis i ove oblasti. Definiše se šta su modeli, kako se koriste u inženjerstvu uopšte i kako pomažu pri dizajnu i implementaciji softverskih rešenja. Opšti jezici za modelovanje osmišljeni su tako da se njima mogu opisati problemi iz bilo kog domena primene i rešenja koja koriste bilo koju tehnologiju ili arhitekturu. Jezici specifični za domen, sa druge strane, fokusiraju se samo na neku usku oblast i koriste koncepte i konstrukte te oblasti, bliske krajnjim korisnicima. Ovo poglavlje daje osnove i ove dve klase jezika, kao i obrazloženje kako se modeli na visokom nivou apstrakcije, dati bilo opštim, bilo jezikom za modelovanje specifičnim za domen, mogu koristiti za dobijanje izvornog koda za željenu tehnologiju. Na kraju, ovo poglavlje daje uvod u oblast semantičkog veba, prikaz pojma ontologije i ideje korišćenja semantičkog rezonovanja kako bi se iz specifikacije poznatih činjenica mašinski izvukli zaključci.

Poglavlje 3 daje pregled naučne i stručne literature iz oblasti integracije i prikazuje postojeće pristupe njenoj automatizaciji, uz poseban osvrt na semantičke pristupe, odnosno one koji koriste tehnologije semantičkog veba i one koji koriste principe modelom upravljanog razvoja softvera.

Poglavlje 4 uvodi pristup automatizaciji semantičke integracije poslovnih aplikacija dat u ovoj tezi. Predstavljen je proces mapiranja, koji sa jedne strane polazi od modela strukture interfejsa aplikacija koje se integrišu, a sa druge strane od formalnog opisa semantike tih sistema, datih objedinjenom ontologijom. Korisnik zatim anotira elemente srukturnog modela elementima ontologije. Ovaj, semantikom obogaćen model, radni okvir koristi za uspostavljanje mapiranja između elemenata ulaznih i izlaznih interfejsa, kao i za detekciju i razrešavanje semantičkih konflikata. U poglavlju su dati neki mogući kriterijumi koji se mogu koristiti za detekciju kandidata za ovakvo mapiranje, kao i kriterijumi za detekciju

konflikata. Ovi kriterijumi u radnom okviru predstavljaju nezavisne komponente, koje se po potrebi mogu uključivati ili modifikovati.

Kako bi se radni okvir, čija je arhitektura opisana u poglavlju 4, mogao testirati i evaluirati, razvijena je njegova praktična implementacija. Ova implementacija je opisana u poglavlju 5. Kao bi se skratilo vreme neophodno za implementaciju, kao podloga je korišćen alat za integraciju otvorenog koda, Talend Open Studio. Ovaj alat sadrži sve neophodne mehanizme za opis strukture aplikacija koje se integrišu, konektore za razne vrste protokola, mehanizme za generisanje izvornog koda izvršivog integracionog rešenja, kao i podršku za ručno mapiranje elemenata interfejsa aplikacija koje se integrišu. Time je rad na implementaciji našeg rešenja sveden na ono što zaista jeste predmet ovog istraživanja, a to je automatizacija pomenutog mapiranja.

Arhitektura radnog okvira definisanog u prethodna dva pomenuta poglavlja je takva da je moguće po potrebi naknadno razviti komponente koje su prilagođene problemu koji se javi prilikom nekog konkretnog integracionog scenarija. Način na koji se ove komponente implementiraju zahteva korišćenje jezika opšte namene, u ovom slučaju Jave i, pored poznavanja arhitekture ovog radnog okvira, poznavanje i arhitekture platforme na kojoj je on implementiran, a to su Eclipse Rich Client Platform i aplikacija Talend Open Studio. Kako bi se omogućilo opisivanje komponenti radnog okvira na način koji je nezavisan od platforme implementacije, razvijen je jezik specifičan za domen, nazvan SAIL - Semantic Application Integration Language. Dizajn, odnosno meta-model i implementacija ovog jezika opisani su u poglavlju 6.

U poglavlju 7 prikazana je empirijska i eksperimentalna evaluacija radnog okvira za automatizovano mapiranje. Korišćena su dva praktična integraciona scenarija: (1) procedura prikupljanja medicinskih analiza klijenata doma za negu starih lica i njihovo prosleđivanje informacionom sistemu bolnice i (2) integracija portala za vođenje projekata sa SAP informacionim sistemom. Za oba scenarija je ranije postojala ručno izrađena implementacija, a zatim je integracija izvedena našim radnim okvirom, uz poređenje rezultata. Eksperimentalna evaluacija izvedena je kroz skup integracionih scenarija koje su učesnici u eksperimentu mapirali prvo ručno, a zatim automatski, pomoću našeg radnog okvira. Na osnovu prikupljenih rezultata izvedeni su zaključci.

Sadržaj

Skraćenice

AIAG Automotive Industry Action Group. 7

AIDX Aviation Information Data Exchange. 7, 8

AIRM ATM Information Reference Model. 7, 8

ANSI American National Standards Institute. 82

API Application programming interface. 9, 16, 22, 36, 37, 41, 60

ASCII American Standard Code for Information Interchange. 19

ATM Air Traffic Management. 7

BAPI Business Application Programming Interface. 90–92, 94

BOD Business Object Documents. 7

BPEL Business Process Execution Language. 28

BPM business process management. 6

BPML Business Process Modeling Language. 7

CDS Cell directory service. 9

CIM Computation Independent Model. 27

COIN Context Interchange Framework. i, 23

COM Component Object Model. 10

CORBA Common Object Request Broker Architecture. 10

CPS Cyber-Physical Systems. 32

CREAM Conflict Resolution Environment for Autonomous Mediation. ii, 31

CRM Customer relationship management. 38

CSV Comma-separated values. 36, 38, 81–84, 86, 89

DAML DARPA Agent Markup Language. 20

DCE Distributed computing environment. 9

DL Description Logic. 18, 20, 22, 31, 34, 35

DSL Domain-specific language. 14

DSM Domain-specific model. 14

DTD Document Type Definition. 7

EAI enterprise application integration. 6, 33

ECTL Extract, Cleanse, Transform and Load. 32

ELF Executable and Linkable Format. 60

ER Entity relationship (diagram). 35

ERP Enterprise Resource Planing. xvii, 35

ESB Enterprise Service Bus. 6, 28, 59

ETL Extract, Transform, Load. 10

FDR Failures-Divergences Refinement. 27

FTP File Transfer Protocol. 38, 39, 41

GUI graphical user interface. 60

HTTP Hypertext Transfer Protocol. 21, 24, 39, 83, 91

HTTPS Hypertext Transfer Protocol Secure. 91

IDE Integrated Development Environment. 60

IDL Interface Definition Language. 10

IEC International Electrotechnical Commission. xv, 25, 26

IIOP Internet InterORB Protocol. 10, 24

IMAP Internet Message Access Protocol. 38

IP Internet Protocol. 90, 93

IRI International Resource Identifier. 19

ISO International Organization for Standardization. 83

JDBC Java Database Connectivity. 9

JSD Domain Specific Language – DSL. 14, 15, 27–29, 33, 37, 39, 51, 55, 84

JSON JavaScript Object Notation. 20

LDAP Lightweight Directory Access Protocol. 38

LSL Link Specification Language. 36

LTSA Labelled Transition System Analyser. 27

MDA Model-driven architecture. 27

MDE Model-driven engineering. 14

MIME Multipurpose Internet Mail Extensions. 83

MOF Meta Object Facility. 16

MOM Message-oriented middleware. 11

MSMQ Microsoft Message Queuing. 39

OAG Open Applications Group. 7

OBDA Ontology-Based Data Access. 34, 35

ODBC Open Database Connectivity. xv, 1, 9, 10

ODSOI Ontology-Driven Service-Oriented Integration. ii, 28

OIC Oracle Integration Cloud. 41

OIL Ontology Inference Layer. 20

OMG Object Management Group. 10

ORB Object request broker. 10

OSI Open Systems Interconnection. 83

OWL Web Ontology Language. ii, 17, 20, 22, 28, 31, 35, 36, 51, 60

PIM Platform Independent Model. 27

POJO Plain Old Java Object. 85

POP3 Post Office Protocol version 3. 38

PSM Platform Specific Model. 27

RCP (Eclipse) Rich Client Platform. 60, 67

RDF Resource Description Framework. 3, 17, 19–21, 36, 60

RDFS RDF Schema. 20, 22

REST Representational state transfer. 36, 38, 41

RMI Remote method invocation. 10

RPC Remote Procedure Call. 6, 9

RSVM Razvoja softvera vođen modelima. ii, 32

SAD National Institute of Standards and Technology. 41

SAIL Semantic Application Integration Language. iv, xvi, xix, 51, 67, 72, 73, 78, 80

SCROL Semantic Conflict Resolution Ontology. 31

SDK Software development kit. 33, 34

SMTP Simple Mail Transfer Protocol. 38, 39, 91

SOA Servisno orijentisane arhitekture. ii, 6, 29

SOAP Simple Object Access Protocol. 36, 38, 39, 41, 90, 91

SPARQL SPARQL Protocol and RDF Query Language. 21, 36, 72

SQL Structured Query Language. xvi, 1, 9, 32, 38, 84, 86, 87

SQS Amazon Simple Queue Service. 38

STAR Standards in Automotive Retail. 7

SUBP Sistem za upravljanje bazama podataka. 1, 2

SWRL Semantic Web Rule Language. 17

SWT Standard Windget Toolkit. 60

TCP Transmission Control Protocol. 90

TOS Talend Open Studio. xvi, 54, 55, 59–62, 86, 87, 92–94

TP Tehniki prostor. 32, 33

UML Unified Modeling Language. xvii, 7–9, 16, 35

URI Uniform Resource Identifier. 19, 37

URL Uniform Resource Locator. 19, 21

WCF Windows Communication Foundation. 39

WSDL Web Services Description Language. xvi, xix, 29, 39, 91–93

XMI XML Model Interchange. xvii, 7, 8, 16

XML Extensible Markup Language. 3, 7, 16, 20, 22, 31–33, 36, 38, 48, 49, 61, 83, 90–92, 94

XSD XML Schema Definition. 31, 39

XSLT eXtensible Stylesheet Language Transformations. 31, 40

Slike

Tabele

Listinzi

1

Uvod i motivacija

Poslovna informatika se rodila iz mogućnosti, koja se otvarala pred kompanijama, da upotrebom elektronskih računara određene poslove učine bržim, efikasnijim i manje podložnim greškama. Ovi poslovi su se do tada obavljali ručno, ili eventualno uz upotrebu mehaničkih pomagala. Međutim, kako su računari i sve što je bilo vezano za njihovu upotrebu bili skupi, poveravani su im samo pojedini zadaci. Kako su tehnologije postajale pristupačnije (a time, ključno, isplativije), sve više aspekata poslovanja dobijali su računarsku podršku. Od računovodstvenih zadataka, kojima su se bavili pioniri njihove upotrebe, računari su uvođeni u korespodenciju, lance nabavke, planiranje, projektovanje, proizvodnju.

Ovakvo, postepeno uvođenje računara u poslovanje, imalo je za posledicu da se softver koji je pokrivao određene aspekte poslovanja nabavljao ili razvijao u etapama. Neretko, softverska rešenja koja su pokrivala određeni vid poslovanja, razvijana su bez uvida u ostala softverska rešenja koja su u tom trenutku bila u upotrebi u istoj kompaniji. Posledica ovoga je bila da su određeni podaci, koji su bili neophodni za razne namene, morali biti uneti u više različitih aplikacija. Još važnije, ti podaci su morali biti i ažurirani na više mesta ukoliko bi došlo do promene. Na taj način dolazilo je do pojave onoga što će kasnije biti nazvano *silosima informacija*. Funkcionalno povezani podaci nalazili su se na fizički odvojenim sistemima.

Baze podataka su viđene kao rešenje ovog problema. Umesto da svaka aplikacija skladišti podatke na sebi svojstven način, sve aplikacije mogu koristiti jednu bazu podataka. Moguće je projektovati šemu baze podataka tako da obuhvati potrebe svih podstistema. Ukoliko su aplikacije već projektovane tako da koriste bazu podataka, moguće je izvršiti integraciju njihovih pojedinačnih šema u jedinstvenu [82]. Međutim, u praksi ponekad nije moguće menjati način na koji postojeće aplikacije čuvaju podatke. Proizvođači aplikacije mogu proceniti da im se takve promene ne isplate. U nekim slučajevima, kompanije koje su razvile softver više ne postoje. Tehničke ili pravne prepreke mogu sprečiti kompaniju koja je korisnik da ona načini izmene. Čak i kada sve aplikacije koriste baze podataka, one mogu koristiti sisteme za upravljanje bazama podataka (SUBP) različitih proizvođača. Iako postoje standardizovani načini za pristup SUBP, kao što su Open Database Connectivity (ODBC) i Structured Query Language (SQL), u praksi postoje značajne razlike, zbog kojih prepravka

aplikacije koja je projektovana da koristi određeni SUBP na korišćenje drugog (ili čak druge verzije istog) ne predstavlja trivijalan poduhvat. Na kraju, za projektovanje integrisane šeme baze podataka neophodno je poznavati detaljnu specifikaciju pojedinačnih šema na konceptualnom nivou, za šta je dokumentacija često nedostupna, a iz fizičke šeme se ne može u potpunosti sagledati (na primer, u nekim aplikacijama su ograničenja implementirana samo na nivou korisničkog interfejsa, što svakako nije preporučena praksa, ali, nažalost, ni redak slučaj).

Jedno rešenje za ovaj problem je takozvani **reinženjering**. Umesto postojećih pojedinačnih aplikacija izrađuju se nove aplikacije, dizajnirane tako da budu međusobno interoperabilne, ili pak jedna aplikacija, koja predstavlja integralni informacioni sistem. Nove aplikacije preuzimaju funkcionalnost starih, kao i njihove podatke.

Integracija je alternativa reinžinjeringu. Cilj integracije je povezivanje različitih funkcionalnosti gotovih aplikacija, bez potrebe za njihovom izmenom. Integraciono rešenje automatizuje usklađivanje uskladištenih podataka među ovim aplikacijama. Pored toga, omogućava izgradnju složenih poslovnih procesa, u kojima svaka aplikacija obavlja deo funkcionalnosti za koji je zadužena, dok se korisniku obezbeđuje jedinstven interfejs kroz koji može upravljati izvršavanjem procesa. Sam proces integracije može biti izuzetno složen i zahtevan. Inženjeri koji sprovode integraciju moraju imati znanja o širokom spektru tehnologija, protokola i formata. Neretko, u istom integracionom scenariju mogu učestvovati aplikacije koje koriste moderne, aktuelne tehnologije zajedno sa aplikacijama koje koriste zastarele tehnologije. Pored poznavanja tehnologija kojima su aplikacije razvijene, neophodno je i detaljno poznavanje funkcionalnosti samih aplikacija.

Danas postoje alati koji delimično olakšavaju neke od aspekata integracije. Ovi alati obično imaju mnoštvo raspoloživih gotovih konektora - komponenti koje omogućavaju razmenu podataka ili pozivanje funkcionalnosti određenim formatom ili protokolom. Za mnoge od ovih formata moguća je i automatska ekstrakcija strukture, odnosno šeme koju koristi neka aplikacija. Dostupni alati olakšavaju i testiranje, izvršavanje i nadgledanje rada integracionog rešenja.

Deo procesa integracije koji i dalje predstavlja prevashodno manuelni posao jeste uspostavljanje mapiranja između interfejsa različitih aplikacija. Automatizacija procesa mapiranja, kao i automatizacija detekcije grešaka koje se pri tom mogu javiti i njihovog rešavanja mogli bi pojednostaviti celokupan proces integracije, učiniti ga jeftinijim i pouzdanijim. Da bi se mapiranje moglo obaviti sa adekvatnim nivoom pouzdanosti, kao i da bi se mogli efikasno utvrditi konflikti koji postoje u načinu na koji različite aplikacije tumače podatke i funkcionalnosti kojima operišu, neophodno je pored strukture interfejsa poznavati i njihovu semantiku. Iz tog razloga, pristup prikazan u ovoj disertaciji baziramo na ideji da se strukturni model interfejsa na pogodan način obogati formalnom definicijom svoje semantike.

***

Druga polovina devedesetih godina donosi usvajanje i ekspanziju veba (World Wide Web) kao opsežnog izvora informacija i medijuma za deljenje znanja i podataka. Arhitektura veba

polazila je od niza uslova [39], koji su doprineli kako njegovom usvajanju, tako i održanju kao aktuelne platforme u nastupajućim godinama:

- lakoća korišćenja - kako pri konzumiranju, tako i pri kreiranju sadržaja,
- proširivost - moraju postojati načini kojima se prvobitna implementacija distribuiranog sistema može naknadno prilagoditi i unaprediti,
- skalabilnost,
- uniformnost interfejsa i
- podrška evoluciji zahteva.

Iz pepela *dot-com mehura*[1] iznikao je Veb 2.0. Iako novi veb nije predstavljao i novi standard u tehničkom smislu, glavna novina u pogledu korišćenja veba, bilo je sveprisutno uključivanje korisnika u doprinos sadržaju. Korisnici su doprinosili na razne načine: od deljenja utisaka o pročitanim knjigama ili pogledanim filmovima, do pisanja enciklopedijskih članaka. *Tagovanjem* multimedijalnih sadržaja, ljudi pomažu u njihovoj klasifikaciji, a *lajkovanjem* izdvajaju sadržaje koji su im interesantni. Statičke stranice zamenjene su dinamičkim aplikacijama, indeksi su zamenjeni pretragama. Aplikacije mogu koristiti podatke koje prikupe od korisnika kako bi agregirale određene rezultate i pružile korisnicima povratnu informaciju koja je za njih od interesa.

Međutim, iako interakcija korisnika i aplikacija može biti od uzajamne koristi, podaci koje ostavljaju korisnici dostupne su samo u okviru određene aplikacije. Iako raznim tehnikama druge aplikacije mogu dobiti podatke koje određena aplikacija pruža korisnicima, one nisu svesne *značenja* tih podataka. Problem „silosa informacija", koji je ranije pomenut u kontekstu poslovne informatike javlja se i na vebu.

Semantički veb [13] zamišljen je sa ciljem da se omogući aplikacijama da pored podataka mogu saznati i semantiku, odnosno - značenje, tih podataka. Načini za reprezentaciju znanja ranije su izučavani u oblasti računarske inteligencije. Uz XML, koji omogućava da se elementi dokumenta označe i struktuiraju, Resource Description Framework (RDF), omogućava da se iskaže njihovo *značenje*. Ontologija, u kontekstu semantičkog veba, predstavlja dokument koji formalno definiše odnose među pojmovima.

Cilj ovog istraživanja je pronalaženje načina da se proces integracije poslovnih aplikacija, kao i detekcija i razrešavanje semantičkih konflikata koji se pri tome javljaju, u mogućoj meri automatizuju. Cilj automatizacije je povećanje efikasnosti samog procesa, radi bržeg i jeftinijeg postizanja integracije, kao i smanjenje mogućnosti za nastanak grešaka koje se javljaju pri ručnoj integraciji. Polazeći od ovih ciljeva, kao i od aktuelnog stanja oblasti, formiramo sledeće hipoteze, uz važenje jedne prethodne pretpostavke:

Pretpostavka 1: Postoji jedinstvena ontologija, formalni zapis semantike, koja opisuje sve aspekte relevantne za integraciju svih sistema čija se integracija obavlja.

[1] Takozvani Dot-com mehur predstavlja period na početku XX veka, kada je zabeležen ektreman rast vrednosti kompanija koje su svoje poslovanje bazirale na ponudi usluga putem Interneta, a zatim i naglog pada njihove vrednosti i gašenja većine ovih kompanija.

Hipoteza 0: Moguće je razviti radni okvir koji podržava mehanizme koji, uz pomoć opisa semantike datog u ontologiji, koja je definisana u *pretpostavci 1*, omogućavaju automatizovanu detekciju mapiranja između elemenata interfejsa koji učestvuju u integraciji na osnovu zadatih kriterijuma mapiranja, kao i automatizovano otkrivanje semantičkih konflikata između datih interfejsa, na osnovu zadatih kriterijuma konflikata.

Hipoteza 1: Kriterijumi mapiranja i kriterijumi konflikata, pomenuti u Hipotezi 0, mogu biti razvijeni kao nezavisne komponente, čime se korisniku omogućava dodavanje, modifikacija i uklanjanje datih kriterijuma.

Hipoteza 2: Moguće je kreirati jezik specifičan za domen, kojim se mogu opisati kriterijumi za mapiranje i kriterijumi za otkrivanje konflikata.

Dakle, opseg ovog istraživanja ograničavamo na ispitivanje mogućnosti korišćenja formalnog zapisa semantike svih sistema koji učestvuju u integracionom scenariju, kako bi se automatizovalo uspostavljanje mapiranja među interfejsima aplikacija koje se integrišu. Za potpunu automatizaciju celog procesa integracije, neophodno bi bilo automatizovati i ekstrakciju, spajanje i poravnanje ontologija za svaki pojedinačan sistem, što je predmet istraživanja druge disertacije. Za potrebe stvaranja uslova za evaluaciju ovog istraživanja, ovi koraci su obavljani ručno.

2

Teorijske osnove

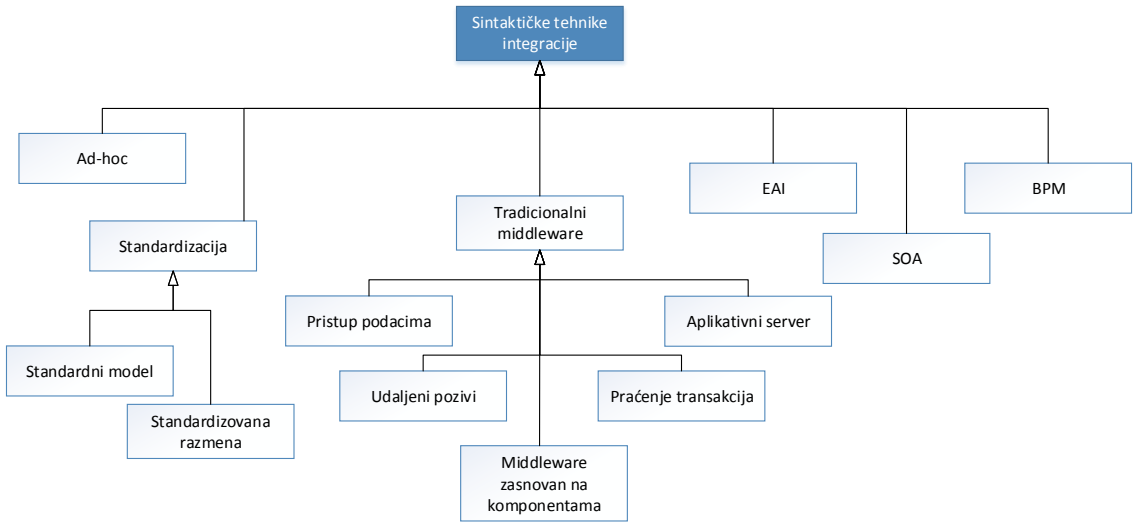
U ovom poglavlju prikazane su oblasti integracije poslovnih aplikacija, kao i tehnika i tehnologija koje su služile kao polazna tačka za istraživanje prikazano u ovoj disertaciji. Najpre je u sekciji 2.1 dat pregled raznih mogućih vrsta integracije. Zatim je, u sekciji 2.2, predstavljen razvoj softvera upravljan modelom. Modeli se u pristupu integraciji prikazanom u poglavlju 4 koriste za specifikaciju strukture interfejsa aplikacija koje se integrišu. Na kraju, u sekciji 2.3, opisane su osnove semantičkog veba; tehnologije koja se u našem pristupu koristi kako bi se strukturalni model interfejsa obogatio semantikom, što pomaže u traženju kandidata za mapiranje i pri detekciji i rešavanju semantičkih konflikata.

2.1 Vrste integracije

Kategorije po tipu i broju atomičkih delova

Četiri kategorije metoda sistemske integracije mogu se klasifikovati po tipu i broju atomičkih delova sistema koji se integrišu [50, 96]:

- **Vertikalna integracija**. Metodi iz ove grupe integrišu delove sistema po osnovu njihove funkcionalnosti. Integrišu se samo neophodni delovi sistema, što u prvi mah rezultuje brzom i jeftinom integracijom. Mana ovakvih metoda je nemogućnost ponovnog korišćenja integrisanih entiteta, pa je kasnije dodavanje novih entiteta skupo i komplikovano.
- **Zvezdasta integracija**. Delovi sistema se integrišu svako sa svakim. Poznata i pod nazivima *point-to-point* i špageti integracija. Ovakve metode su fleksibilne i jednostavne za implementaciju u pogledu svakog para ponaosob, ali kompleksnost integracije raste eksponencijalno s brojem delova koji se integrišu. Dodatni izazov za ovakve metode predstavlja integracioni scenario gde je potrebno da više delova istovremeno sarađuje kako bi se agregirala određena funkcionalnost.



Slika 2.1: Tipologija sintaktičkih tehnika integracije po [66]

- **Horizontalna integracija**. Metode horizontalne integracije se oslanjaju na postojanje komponente koja ima ulogu prosleđivanja poruka između delova sistema. Svaki deo sistema se time integriše samo jednom - sa posredničkom komponentom. Ulogu ove komponente može obavljati Enterprise Service Bus (ESB). U ovakvim načinima integracije moguće je zameniti neki od delova drugim koji pruža sličnu funkcionalnost, čak i kada novi deo ima drugačiji interfejs. U idealnom slučaju, ova zamena je transparentna za ostale delove sistema.
- **Zajednički format podataka**. U ovakvim metodama, usvaja se zajednički format podataka, nezavisan od formata pojedinačnih aplikacija koje se integrišu. Za svaki deo koji se integriše, kreira se adapter koji vrši mapiranje na zajednički format.

Sintaktički pristupi integraciji

U sintaktičke pristupe integraciji mogu se svrstati: ad-hoc tehnike, tehnike zasnovane na standardizaciji, tradicionalni *middleware*, pristupi zasnovani na alatima za integraciju poslovnih aplikacija (enterprise application integration, EAI), pristupi zasnovani na tehnikama za upravljanje poslovnim procesima (business process management, BPM) i pristupi zasnovani na tehnikama servisno orijentisanih arhitektura (service-oriented architectures, SOA) [66]. Pregled sintaktičkih tehnika integracije dat je na slici 2.1.

Ad-hoc integracija. Ad-hoc tehnike se zasnivaju na implementaciji adaptera, pri čemu se za svaki interfejs koji učestvuje u integraciji piše adapter ka svim ostalim interfejsima sa kojima je neophodna komunikacija. Obično su adapteri pisani na programskim jezicima opšte namene [66], kada se za realizaciju koriste mehanizmi koji su na raspolaganju na jeziku i u bibliotekama koje se koriste za implementaciju adaptera, kao što je Remote Procedure

Call (RPC), ali moguće je i korišćenje alata za integraciju [115]. U smislu podele date u sekciji 2.1, na ovaj način se implementiraju point-to-point, odnosno zvezdaste metode.

Tehnike zasnovane na standardima. Tehnike zasnovane na standardima se baziraju na ideji da bi aplikacije koje treba integrisati trebalo da podrže određene standarde formata poruka koje se razmenjuju, formata podataka ili modela procesa koji se odvijaju. Ovakve tehnike primenljive su u slučajevima kada sve aplikacije koje se integrišu pripadaju određenom domenu (npr. zdravstvo, proizvodnja, avionika, obrazovanje), odnosno kada je moguće doneti pomenute standarde za datu oblast. Navodimo nekoliko primera takvih standarda.

- **XML** (eXtensible Markup Language) [18] namenjen je struktuiranom zapisivanju dokumenata i podataka u obliku stabla. Za specifikaciju modela dokumenta može se koristiti stariji jezik DTD (Document Type Definition) ili XML Schema [109, 110] koji ispravlja njegove nedostatke. U praksi se javljaju i dokumenti koji nemaju pridružen model. XML se koristi i kao način zapisivanja za druge standarde, poput **ebXML** (e-Business XML), **XMI** (XML Metadata Interchange) formata za razmenu UML dokumenata [103], BPML i drugih.
- **RosettaNet** je standard namenjen razmeni informacija između poslovnih subjekata (Business-to-business, B2B). Omogućava usaglašavanje procesa među učesnicima u lancu nabavke na globalnom nivou [15]. Propisuje ga istoimeni konzorcijum.
- **HL7** je standard za komunikaciju u oblasti zdravstva. Detaljnije je opisan u sekciji 7.1
- **IDEF3** namenjen je za opis sekvenci aktivnosti u poslovnim procesima [86].
- **BPML** - Business Process Modeling Language [17] je još jedan jezik za opis poslovnih procesa.
- **UML** - Unified modelling language, jezik opšte namene za specifikaciju softverskih i drugih sistema [105],
- **AIRM** - ATM Information Reference Model [22] je UML model namenjen obezbeđivanju semantičke interoperabilnosti evropske mreže za upravljanje vazdušnim saobraćajem (Air Traffic Management, ATM)[1], inicijalno propisan od strane organizacije Eurocontrol.
- **AIDX** - Aviation Information Data Exchange (AIDX) [3] je standard za razmenu XML poruka o informacijama u vezi sa letovima među aerodromima, aviokompanijama i drugim pružaocima usluga iz oblasti vazdušnog saobraćaja, propisan od strane Međunarodne asocijacije za vazdušni prevoz (IATA)[2].

Namena navedenih standarda je da omogućavaju razmenu podataka ili poruka između aplikacija različitih korisnika i proizvođača, ukoliko te aplikacije podržavaju dati standard. Jasno je da se problem javlja ukoliko aplikacije ne podržavaju neki zajednički standard. Na primer, **STAR** (Standards in Automotive Retail) i **AIAG** (Automotive Industry Action

[1] http://airm.aero
[2] https://www.iata.org/publications/Pages/info-data-exchange.aspx

Group) su dva standarda iz oblasti automobilske industrije. Oba baziraju interfejse na istom horizontalnom standardu **BOD** (Business Object Documents), kog propisuje Open Applications Group (OAG)[3]. Iako imaju istu bazu i sličan domen primene, za međusobnu komunikaciju po ova dva standarda potrebna je integracija (jedan primer semantičke integracije ova dva standarda dat je u [8]). Slična relacija može se uvideti između navedenih standarda **AIRM** i **AIDX**, koji služe istoj industriji, ali za različite namene i koriste različite formate.

Tako, čak i kada neke dve aplikacije po specifikaciji podržavaju određeni standard, u praksi se mogu javiti problemi pri pokušaju razmene podataka. Na primer, [83] analizira mogućnost međusobne razmene modela između pet alata za UML modelovanje, koji podržavaju XML Model Interchange 2.0 (XMI) format za izvoz i uvoz modela: (1) Borland Together Architect 2006 for Eclipse, (2) EclipseUML Free Edition, (3) IBM Rational Software Architect 6.0, (4) MagicDraw Community Edition version 10.5 i (5) Altova UModel 2006. Rezultati razmene između svakog para ovih aplikacija (uključujući ponovni uvoz u istu) prikazani su u tabeli 2.1. Zbog različitog tumačenja standarda, razlika u implementaciji ili grešaka prilikom implementacije standarda, u 48% (12/25) pokušaja razmena modela među heterogenim alatima bila je neuspešna.

Tabela 2.1: Rezultati razmene modela korišćenjem iste verzije XMI (✓ - uspešno, ✗ - neuspešno). Izvor: [83]

Izvoz \ Uvoz	Borland	Eclipse	Rational	MagicDraw	UModel
Borland	✓	✓	✓	✗	✗
Eclipse	✓	✓	✓	✗	✗
Rational	✓	✓	✓	✗	✗
MagicDraw	✗	✗	✗	✓	✓
UModel	✗	✗	✗	✓	✓

Dodatni problem može predstavljati činjenica da se standardi vremenom menjaju. Iako alati podržavaju isti standard, verzije koje podržavaju se mogu razlikovati. Tabela 2.2 pokazuje, mahom bezuspešne, pokušaje da se u pet ranije navedenih UML alata uvezu modeli koji su izvezeni iz alata koji podržavaju stariju verziju XMI formata. U svega 4% (2/45) uvezen je nenarušen model, dok je u dodatnih 9% (4/45) zabeležen delimičan uspeh (neki od problema koji su se javljali su: gubitak veza, nepotpuni dijagrami, izmenjeni nazivi atributa, pojava dodatnih klasa).

[3] https://oagi.org

Tabela 2.2: Prikaz kompatibilnosti unazad pet UML alata. Izvor: [83]

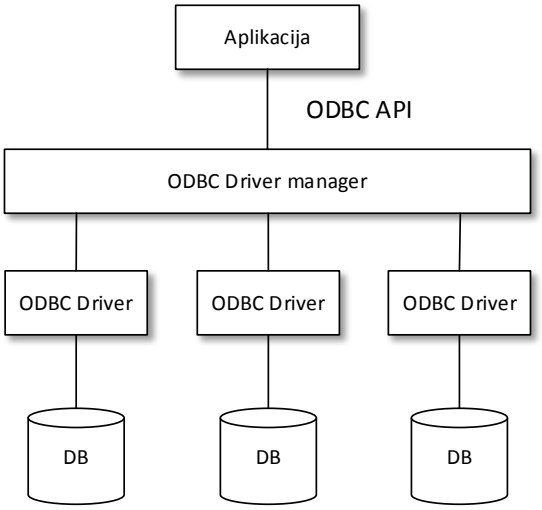
Izvoz \ Uvoz	Borland	Eclipse	Rational	MagicDraw	UModel
ArgoUML	✗	✗	✗	✗	✗
Fujaba	✓	✗	✗	✗	✗
Umbrello	✗	✗	✗	✗	✗
Artisan	✗	✗	✗	✗	✗
Poseidon	✗	✗	✗	✓	✗
Rhapsody	✗	✗	✗	✗	✗
Rose 1.0	✗	✗	✗	✗	✗
Rose 1.1	✗	✗	✗	✗	✗
Visio	✗	✗	✗	✗	✗

Middleware *Middleware* se uopšteno može definisati kao skup zajedničkih usluga, koje omogućavaju aplikacijama i krajnjim korisnicima da razmenjuju podatke i informacije, obično putem računarske mreže. Može se reći i da je *middleware* softver koji vrši medijaciju između aplikacije i mreže, upravljajući međusobnim vezama disparatnih aplikacija koje se nalaze na heterogenim računarskim platformama[4]. Middleware se može klasifikovati u sledeće kategorije [66, 81]:

- mehanizmi pozivanja udaljenih procedura,
- *middleware* orijentisan ka podacima (data-oriented middleware),
- *middleware* orijentisan ka komponentama (component-oriented middleware),
- *middleware* orijentisan ka porukama (message-oriented middleware),
- *middleware* orijentisan ka servisima (service-oriented middleware) i
- aplikativni serveri i monitori transakcija.

DCE-RPC *Distributed computing environment* (DCE) [124] je skup standarda i tehnologija koje razvija The Open Group[5] koji predstavljaju infrastrukturu za razvijanje distribuiranih sistema. DCE je zasnovan na pozivima udaljenih procedura (RPC). Koristi klijent-server arhitekturu i pruža servise niskog nivoa. Serveri implementiraju telo procedura, a interfejse izvoze u globalni hijerarhijski *cell directory service* (CDS). Klijenti pozivaju CDS tražeći određenu funkciju i, nakon autentifikacije i autorizacije dobijaju podatke neophodne za pozivanje procedure na serveru. Ova procedura je transparentna za korisnika.

***Middleware* orijentisan ka podacima** Jedan primer *middleware*-a orijentisanog ka podacima su ODBC (Open Database Connectivity) API, namenjen pristupu bazama podataka, kao i njegov ekvivalent za Javu, JDBC (Java Database Connectivity) [56]. Cilj ovih specifikacija je da omoguće radnje sa bazom podataka putem SQL jezika, nezavisno od konkretnog sistema za upravljanje bazama podataka, kao što je prikazano na slici 2.2.



Slika 2.2: Pristup aplikacije bazi podataka posredstvom ODBC.

Još jedan primer middleware-a orijentisanog ka podacima su ETL (Extract, Transform, Load) alati [134]. Ovi alati se koriste kako bi se podaci dobavili (ekstrahovali) iz raznih izvora, zatim validirali, transformisali (očistili, deduplicirali, agregirali, deagregirali, itd) i učitali u skladište podataka (data warehouse), nakon čega je moguće nad njima sprovesti analize [84].

Middleware orijentisan ka komponentama Ova vrsta middleware-a namenjena je povezivanju distribuiranih komponenti [66]. S obzirom da se najčešće implementira u objektno-orijentisanim okruženjima, naziva se i *objektno-orijentisani* middleware. Jedan primer ove klase middleware-a je **CORBA** (Common Object Request Broker Architecture). CORBA [28] je standard razvijen od strane Object Management Group (OMG). Broker (Object request broker, ORB) je mehanizam putem kog objekti upućuju zahteve i dobijaju odgovore, nezavisno od toga da li je drugi objekat na istom računaru ili na mreži [126]. Mehanizam je transparentan za klijenta. Interfejsi se definišu korišćenjem IDL (Interface Definition Language), deklarativnog jezika koji je nezavisan od programskog jezika koji se koristi za implementaciju. Brokeri komuniciraju međusobno koristeći Internet Inter-ORB Protocol (IIOP). Neki od jezika za koje postoji CORBA implementacija su C, C++, Smalltalk, Ada'95, Java, COBOL, Modula-3, Perl i Python [135]. Nepotpune implementacije i greške u implementacijama brokera dovele su do raznih problema u praktičnoj upotrebi ove tehnologije. U [61] konstatuje se da je CORBA prešla put od vrhunske tehnologije u svojim ranim danima, preko popularnog middleware-a, do relativne zastarelosti. Još neki primeri ove vrste middleware-a su Microsoft [D]COM ([Distributed] Component Object Model) i Sun (sada Oracle) Java RMI.

[4] https://web.archive.org/web/20120629211518/http://www.middleware.org/whatis.html
[5] http://www.opengroup.org/dce/

***Middleware* orijentisan ka porukama** *Message-oriented middleware* (MOM) namenjen je izgradnji slobodno spregnutih (loosley coupled) integracionih rešenja. Sistemi koriste *poruke* kako bi razmenjivali pakete podataka. Poruke se mogu razmenjivati sinhrono ili asinhrono, što je češće slučaj. Osnovni koncepti ove paradigme su [62]:

- **Kanal** - definisan put kojim se prenose poruke. Aplikacije šalju poruke u određeni kanal i primaju poruke iz određenog kanala.
- **Poruka** - atomični paket podataka koji se prenosi putem kanala.
- **Dostava u više koraka** (multi-step delivery) - nakon što jedna aplikacija pošalje poruku, pre nego što je neka druga aplikacija primi, nad porukom mogu biti preduzete određene radnje - validacija, transformacija, preusmeravanje, logovanje, itd.
- **Rutiranje** - pojedinačni koraci u obradi neke poruke mogu se učiniti nezavisnim jedni od drugih korišćenjem rutera, komponenti koje primaju poruku na jednom kanalu i vrše njeno ponovno slanje na nekom drugom kanalu, pri čemu se kanal u koji će poruka biti preusmerena bira na osnovu definisanih uslova.
- **Transformacija** - ukoliko aplikacije koje komuniciraju ne koriste isti oblik podataka, mogu se koristiti prevodioci (Message Translator), komponente koje prevode poruke iz jednog formata u drugi.
- **Pristupne tačke** (endpoints) - komponente koje predstavljaju most između ugrađenih interfejsa aplikacije i sistema za razmenu poruka.

Po [66], postoje tri osnovna vida implementacije MOM:

- direktna razmena (*message passing*) [113] - je najjednostavnija organizacija, gde se poruke šalju u jednom smeru, bez uspostave konekcije (na primer, korišćenjem UNIX sistemskih funkcionalnosti kao što je pipeline za ulančavanje i povezivanje procesa ili sličnih funkcionalnosti u drugim operativnim sistemima),
- redovi čekanja (*message queueing*) [116] - proširuju direktnu razmenu, uvodeći perzistenciju poruka i automatsko ponovno slanje u slučaju da primalac nije odmah na raspolaganju i
- objava i pretplata (*publish/subscribe*) [85] - učesnici u komunikaciji mogu objaviti vrstu sadržaja koju proizvode, a drugi učesnici se mogu pretplatiti na takav sadržaj; omogućena je komunikacija više-ka-više, pri čemu učesnici postaju nezavisni jedni od drugih.

2.1.1 Klasifikacija semantičkih konflikata

U [98] data je klasifikacija semantičkih konflikata u heterogenim sistemima. Klasifikacija je izvedena po tri dimenzije: imenovanje, apstrakcija i nivo heterogenosti. Naglašava se potreba za semantičkim usklađivanjem strana koje učestvuju u komunikaciji. Utvrđeno je da detekcija konflikata zavisi od raspoloživog podskupa celokupnog šematskog (strukturnog)

i semantičkog znanja o sistemima. Kao primeri klasa semantičih konflikata navode se: strukturalne i prezentacione razlike, neusklađeni domeni i konflikti imenovanja. Neki primeri ovih klasa konflikata su:

- Konflikti imenovanja:
  - sinonimi - različiti pojmovi koji imaju isto značenje,
  - homonimi - isti ili slični pojmovi koji nemaju isto značenje,
  - nepovezani pojmovi - su pojmovi koji postoje u nekom od sistema koji se integriše, a nisu ni homonimi ni sinonimi sa pojmovima u ostalim sistemima, pa se njihova veza ne može utvrditi na osnovu imenovanja.
- Konflikti nivoa apstrakcije:
  - generalizacija - konflikt se javlja kada jedan koncept učestvuje u dva sistema na različitom nivou apstrakcije (npr. klasa Kompanija iz jednog sistema je generalizacija klase Klijent iz drugog sistema),
  - agregacija - konflikt se javlja kada se jedna instanca iz jednog sistema mapira na više instanci u drugom sistemu.

Pet klasa heterogenosti definisane su u [71]: nekompatibilnost domena, nekompatibilnost definicije entiteta, nekompatibilnost vrednosti podataka, nekompatibilnost nivoa apstrakcije i semantičke nejednakosti. U [107, 24] data je podela na dve vrlo široke kategorije konflikata: konflikte (vrednosti) podataka i konflikte šeme.

2.2 Modelom upravljan razvoj softvera

Kreiranje modela sistema koji se razvija, pre kreiranja krajnjeg proizvoda, često je u raznim inženjerskim disciplinama. Inženjerski model je selektivni prikaz nekog sistema, koji predstavlja, precizno i koncizno, suštinske odlike tog sistema sa određene tačke gledišta.

Modeli se mogu koristiti kako bi se bolje **razumelo** određeno ponašanje sistema. Modeli se mogu upotrebljavati i u svrhu **komunikacije** i dokumentovanja. Model građevinskog objekta, bilo da je izrađen od čvrstog kartona ili renderovan na računaru, omogućava arhitektama da krajnjim korisnicima bolje predstave svoju zamisao, kao i da lakše objasne razloge koji stoje iza određenog aspekta rešenja. Fizički model prostornog rasporeda atoma u molekulu može omogućiti predavaču hemije da lakše objasni razloge zbog kojih se dato jedinjenje ponaša na određeni način.

Modeli mogu poslužiti da se uz manje troškove i za kraće vreme proveri i **predvidi** ponašanje nekog sistema (vozila, letelice, građevinskog objekta) u širokom dijapazonu uslova u kojima se taj sistem može naći u toku korišćenja. Na primer, model novog tipa aviona prvo se testira u aerotunelima, laboratorijskim postrojenjima koja omogućavaju da se utvrdi kako se konstrukcija krila, tela i drugih elemenata aviona ponaša u raznim aerodinamičkim uslovima - pri raznim brzinama kretanja, raznim položajima aviona u odnosu na smer kretanja

i slično. Ovakvi testovi na modelima omogućavaju da se utvrdi da li je konstrukcija aviona bezbedna za dalje ispitivanje u punoj veličini i sa ljudskom posadom. Isti testovi omogućavaju da se poredi ponašanje više različitih rešenja i da se njihovom postupnom modifikaciom dođe do optimuma u određenim kriterijumima, kao što je potrošnja goriva pri brzini krstarenja i bezbedan let pri brzini sletanja. Slično, model upravljačke kabine aviona u realnoj veličini, omogućava da se još u ranim fazama razvoja novog tipa aviona ispita kako položaj raznih upravljačkih komandi i instrumenata utiče na bezbednije i efikasnije obavljanje letačkih zadataka.

Na kraju, ukoliko je model dovoljno detaljan, može poslužiti i kao **specifikacija** na osnovu koje se izrađuje krajnji proizvod [121].

Inženjerski modeli su **deskriptivni** modeli ukoliko služe za bolje razumevanje pojava, komunikaciju i predviđanje pojava i ponašanja. Modeli su **preskriptivni** ukoliko mogu služiti kao specifikacija za izgradnju krajnjeg sistema [118]. Jedan model može u isto vreme biti i deskriptivni i preskriptivni model.

Jezici za modelovanje

Jezici za modelovanje su računarski jezici namenjeni konstruisanju modela sistema i njihovog okruženja. Kada govorimo o jezicima namenjenim modelovanju softverskih sistema, prva generacija ovih jezika mahom je služila za kreiranje samo deskriptivnih modela. Ovi modeli služili su samo kao neformalne skice i kao deo tehničke dokumentacije, ali nisu bili formalno definisani, pa je tumačenje moglo biti dvosmisleno. Time ne samo da nisu bili dovoljan izvor za implementaciju modelovanih sistema, već su mogli biti i izvor pogrešne komunikacije. Njihova jednostavnost ujedno može da stvori i lažnu sliku o tome da i druga strana verovatno na isti način shvata značenje korišćenih elemenata jezika (simbola, veza). Time je i deskriptivna uloga neformalnih modela ugrožena. Ovu odliku neformalnih modela opisao je Bertrand Meyer: *[...] the good thing about bubbles and arrows, as opposed to programs, is that they never crash* [91].

Formalni jezici za modelovanje treba da imaju razumljive i precizne semantičke osnove, mogu se formalno, matematički, analizirati, nedvosmisleni su i sadrže dovoljno detalja da pored deskriptivne imaju i preskriptivnu ulogu.

Jezik za modelovanje se sastoji od:

- **apstraktne sintakse**:
  - koncepata jezika (npr. Sistem, Klasa, Paket, Veza, Račun) i
  - pravila za kombinovanje jezičkih koncepata
- **konkretne sintakse**, koja služi za predstavu i zapisivanje:
  - grafički simboli, ključne reči i
  - mapiraju se na koncepte apstraktne sintakse

- **semantike**, koja određuje značenje koncepata jezika:
  - semantički domen i
  - semantičko mapiranje domena na koncepte jezika.

Jedan jezik može imati više konkretnih sintaksi. Za isti jezik moguće je razviti i tekstualnu i grafičku notaciju [33].

Model kao osnova za razvoj softvera

Inženjerstvo upravljano modelima (Model-driven engineering, MDE) model tretira kao deo implementacije softverskog sistema. Postoje dva osnovna načina na koji se modeli na visokom nivou apstrakcije mogu koristiti kao direktna osnova za dobijanje krajnjeg proizvoda. Prvi način je da se model određenim automatizovanim postupkom, korišćenjem alata, prevede (transformiše) u kod pisan programskim jezikom opšte namene. Drugi način je da se sam model izvršava, odnosno interpretira. Pored transformacije u jezik opšte namene, modeli se mogu transformisati i u druge modele.

Deskriptivni modeli, kreirani u svrhu dokumentovanja i komunikacije, neretko nastaju *post festum*. Čak i kada model prethodi implementaciji, često se dešava da, nakon što se implementacija izmeni (zbog uočenih nedostataka ili izmenjenih zahteva) model ne bude ažuriran. Takav model može postati i prepreka za razumevanje funkcionisanja sistema.

Nasuprot tome, preskriptivni model u svetlu MDE je deo implementacije. Od modela se za kratko vreme dobija *živa* aplikacija koju je moguće probati i testirati. Ukoliko dođe do izmene zahteva ili se uoče nedostaci, sledeća iteracija kreće od izmene modela do njegove ponovne transformacije u izvršiv oblik ili direktnog izvršavanja modela. Ukoliko se uoči da nastali problem nije moguće opisati i realizovati raspoloživim jezikom za modelovanje, taj jezik se može izmeniti, ili se može za dati aspekt sistema odabrati drugi jezik.

Jezici specifični za domen

Jezici specifični za domen - JSD (Domain-specific language, DSL) umesto opštih pojmova, prilagođenih rešavanju širokog spektra programskih problema (klase, objekti, paketi, metode i sl.) koriste koncepte koji su bliski oblasti konkretnog problema (npr. radnik, račun, knjiga, uređaj). Sintaksa ovih jezika može biti tekstualna, tabelarna, grafička ili kombinovana [136].

Modelovanje specifično za domen (Domain-specific model, DSM) je metodologija zasnovana na MDE i JSD. Domen od interesa može biti tehnički (horizontalni), što se odnosi na određeni aspekt razvoja: korisnički interfejs, perzistencija, ponašanje, bezbednost, ili funkcionalni (vertikalni), što se odnosi na neku određenu oblast primene za koju se razvija rešenje: bankarstvo, medicina, robotika, proizvodna industrija.

U [133, 136, 73] navode se neke od prednosti korišćenja JSD i DSM.

- **Uključivanje domenskih eksperata**. Ukoliko su koncepti jezika i način njihovog zapisivanja bliski načinu na koji se ti koncepti inače koriste u određenoj oblasti, olakšana je komunikacija i saradnja između onih koji razvijaju softver i onih koji poseduju znanje iz date oblasti.
- **Povećana produktivnost**. Kada se JSD jednom razvije, njegovo korišćenje ubrzava razvoj aplikacija iz datog domena. Istraživanja [73] pokazuju da se produktivnost može povećati i do 1000 %.
- **Povećanje kvaliteta**. S obzirom da je jezik vezan za domen, pravila koja važe u domenu mogu biti uključena u alate koji proveravaju validnost modela.
- **Lakše učenje jezika**.
- **Povećana čitljivost modela**.

Isti izvori navode i izazove povezane sa korišćenjem JSD-ova.

- **Cena razvoja**. Korišćenju jezika prethodi proces dizajna i implementacije, kao i razvijanje propratnih alata. Balansiranje resursa neophodnih za razvoj jezika i dobiti koja proizlazi iz njegovog budućeg korišćenja treba da uključi analizu mogućnosti za ponovno korišćenje jezika. Ukoliko sam domen nije dobro dokumentovan, proces razvoja JSD-a će pored navedenih prednosti vezanih za softversko rešenje imati i pozitivan nusprodukt u vidu formalne dokumentacije domena od interesa.
- **Neophodno znanje**. Ljudi koji razvijaju jezik moraju biti upoznati sa oblastima dizajna jezika, alatima i formalizmima za specificiranje sintakse, kao i alatima za transformisanje ili interpretaciju jezika.
- **Zavisnost od alata**. U razvoju jezika se često koristi mnoštvo alata. Održavanje jezika može biti otežano ukoliko prestane održavanje nekog od korišćenih alata. Takva mogućnost je smanjena, ali i dalje prisutna ukoliko se koriste alati otvorenog koda.

Na kraju, napomenimo činjenicu da će svaka greška ili nedostatak, koji je uveden kroz alat koji vrši transformaciju ili izvršavanje modela, biti propagiran na sve softverske sisteme za čije razvijanje je korišćena data verzija alata. Iz ovoga proističe da alati koji se koriste, ali i ciljni kod koji se generiše moraju biti podrobno pokriveni testovima. Time će i generisani sistemi biti takođe pokriveni testovima. Na kraju, ukoliko se neko neželjeno ponašanje zapazi na jednom od sistema za koje je generisan kod, ispravke se lakše mogu uvesti u sve sisteme u čijem razvoju je korišćen isti JSD i skup alata.

Generisanje koda

U ovoj sekciji opisani su uobičajeni načini za transformaciju modela u izvorni kod, bilo da se radi o modelu koji je definisan tekstualnim ili grafičkim jezikom.

Prvi korak je **čitanje modela**. Ukoliko je za modelovanje razvijen poseban alat, čitanje modela se može obavljati direktno, iz radne memorije alata. Ukoliko je model zadat

tekstualnim jezikom, izvorni fajlovi se parsiraju. Ukoliko je u pitanju UML model, on se može čitati kreiranjem dodatka (plug-in) za alat kojim je model kreiran, ukoliko sam alat to podržava. Ovakvi alati uobičajeno poseduju API kojim je moguće pročitati i manipulisati elementima modela u radnoj memoriji, u internoj reprezentaciji tog alata. Ukoliko alat ne podržava ovaj način dobavljanja modela, model se može izvesti iz alata u XML Metadata Interchange (XMI) formatu [103]. XMI standard se može koristiti za perzistenciju i razmenu bilo kojih jezika koji su bazirani na Meta Object Facility (MOF) [104, 65].

Nakon čitanja, model se **analizira**. U ovom koraku moguće je sprovesti validaciju modela.

Nakon sprovedene analize, model se može **transformisati** u oblik koji je bliži ciljnoj platformi. Ovime se olakšava dalja manipulacija i generisanje koda, a komponenta koja vrši samo generisanje se pojednostavljuje. Rezultat ove transformacije može biti interni za alat, a može biti i drugi model, kojim je moguće dalje nezavisno manipulisati.

Sledeći korak je **generisanje izvornog koda** za odabrani ciljni jezik i platformu. U ovu svrhu se obično koriste **obrađivači šablona** (template engine). Šablon sadrži tekst koji će se naći u ciljnoj datoteci, kao i označena mesta na koja se upisuje tekst koji je rezultat konkretnih elemenata modela. Svaki konkretan obrađivač šablona (Free marker, Jinja, T4 Templates, itd.) ima definisan jezik kojim se pišu šabloni. Ovi jezici mogu pored iskaza za umetanje prostih podataka sadržati i petlje, grananja, a neki obrađivači šablona nude i napredne mogućnosti poput nasleđivanja šablona. Listing 2.1 sadrži FreeMarker šablon koji rezultuje Java klasom popunjenom poljima iz modela.

```
1  package ${class.typePackage};
2
3  ${class.visibility} class ${class.name} {
4  <#list properties as property>
5    <#if property.upper == 1 >
6        ${property.visibility} ${property.type} ${property.name};
7      <#elseif property.upper == -1 >
8        ${property.visibility} Set<${property.type}> ${property.name}
9          = new HashSet<${property.type}>();
10     <#else>
11       <#list 1..property.upper as i>
12       ${property.visibility} ${property.type} ${property.name}${i};
13     </#list>
14     </#if>
15 </#list>
16 }
```

Listing 2.1: Primer FreeMarker šablona

U praksi se često javlja potreba da neki deo softverskog sistema bude ručno izmenjen u odnosu na generisani kod, na primer ukoliko je teško ili neisplativo određenu funkcionalnost realizovati u sklopu jezika za modelovanje i u pratećim alatima. Za ovakve slučajeve neophodno je osmisliti i primeniti mehanizam za kombinovanje ručno pisanog i generisanog koda. Osnovni cilj ovog mehanizma treba da bude očuvanje ručno implementiranih delova nakon ponovnog generisanja koda na osnovu (izmenjenog) modela. Uporedni prikaz raznih mehanizama ove vrste dat je u [52].

2.3 Semantički veb

Veb (svetska mreža, World Wide Web) sadrži veliku količinu dokumenata, mahom u obliku hiperteksta. Hipertekst, putem hiperlinkova, omogućava onome ko čita jedan takav dokument, da dođe do povezanog sadržaja koji se nalazi na drugom mestu u istom ili u drugom dokumentu. Većina podataka, sadržanih u ovim dokumentima, dostupna je u formatima koji su prilagođeni čitanju i korišćenju od strane ljudi, ali su teški za mašinsko čitanje i obradu. Motivacija za nastanak semantičkog[6] veba bila je uvođenje strukture i standardizovane reprezentacije *značenja* dostupnog sadržaja [12]. Na ovaj način, softverski agenti bi bili u mogućnosti da sakupe podatke iz raznih izvora i da na osnovu tih podataka obavljaju poslove za koji zahtevaju informacije dostupne na raznim mestima na vebu. Druga posledica raspoloživosti vezivanja semantike za mašinski čitljive podatke jeste da različiti agenti mogu da razmenjuju podatke međusobno, iako ni jedan od agenata koji učestvuju u ovoj razmeni nije dizajniran sa eksplicitnim znanjem o ostalim agentima. Iz ovoga se vidi jasna sličnost sa osnovnim problemom koji se javlja u integraciji aplikacija i motivacija da se mehanizmi prvobitno namenjeni primeni u semantičkom vebu iskoriste u oblasti integracije. U semantičkom vebu važe sledeće pretpostavke:

- *nejedinstvenost imena* - isti resurs može imati različita imena u različitim kontekstima i
- *otvoreni svet* - nove informacije mogu postati dostupne u bilo kom trenutku.

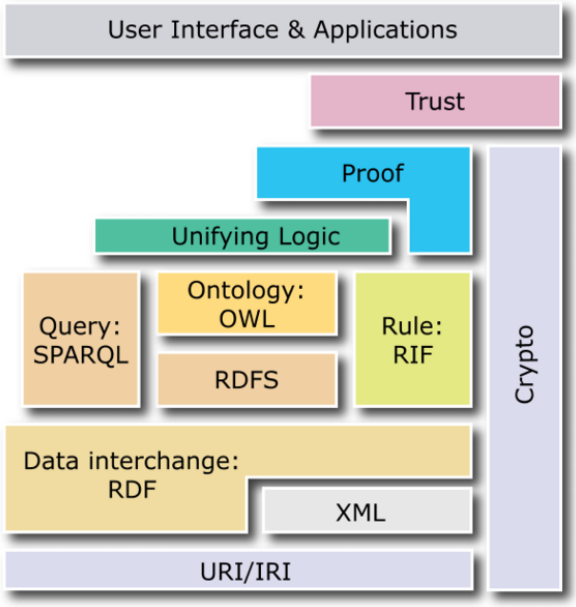
Ontologija

Termin ontologija pozajmljen je iz filozofije, gde označava disciplinu koja izučava prirodu bivstvovanja, postojanja, nastajanja, bitka, osnovne kategorije postojanja i odnose između tih kategorija [19]. U računarske nauke prvi put je uveden na polju računarske inteligencije u [53], gde Gruber definiše ontologije kao *eksplicitne specifikacije konceptualizacije*. Ontologija je definisana još, od strane Borsta, i kao *formalna specifikacija deljene konceptualizacije* [16]. Pod pojmom *konceptualizacije* podrazumeva se apstraktan pogled na svet koji se želi predstaviti. *Eksplicitno* u Gruberovoj definiciji znači da su svi elementi ontologije eksplicitno definisani. *Formalna* specifikacija podrazumeva postojanje jezika, sa formalno definisanom sintaksom i semantikom, koji se može koristiti za specifikaciju ontologije, čime se omogućava da ona bude mašinski čitljiva, da se može mašinski obrađivati, prevoditi, interpretirati i izvršavati. Jezici za definisanje ontologija, korišćeni u semantičkom vebu su: deskriptivna logika, F-logika, RDF, OWL i SWRL (Semantic Web Rule Language).

Ontologije se mogu klasifikovati po nivou opštosti [54] na:

- **ontologije višeg nivoa** (Upper Ontologies) definišu opšte koncepte realnog sveta, nezavisno od domena (npr: prostor, vreme),

[6] semantika - nauka o značenju



Slika 2.3: Stek tehnologija semantičkog veba. Izvor: wikimedia

.

- **domenske ontologije** (Domain Ontologies) specijalizacijom koncepata datih u ontologijama višeg nivoa, definišu rečnik koncepata specifičnih za određenu oblast,
- **ontologije zadatka** (Task Ontologies) specijalizacijom koncepata datih u ontologijama višeg nivoa, definišu rečnik pojmova potrebnih za opisivanje generičkih zadataka ili aktivnosti i
- **aplikativne ontologije** (Application Ontologies) opisuju konkretne aplikacije, pri čemu koncepti ontologije odgovaraju ulogama entiteta iz domena te aplikacije.

Deskriptivna logika (Description Logic, DL) je podskup logike prvog reda i predstavlja familiju formalizama namenjenih za predstavu znanja. DL definiše *koncepte* (unarne predikate) i *uloge* (binarne relacije). *Konstruktori* omogućavaju da se korišćenjem atomičkih koncepata i uloga izgrade kompleksni koncepti. Skup terminoloških aksioma naziva se TBox. Skup asertivnih formalizama naziva se ABox.

F-logika (Frame Logic) je jezik za reprezentaciju znanja i formalizama za ontologije [74]. Zasniva se na kombinovanju konceptualnog modelovanja sa objektno-orijentisanim jezicima. Definiše kompaktnu, deklarativnu sintaksu.

RDF Osnovna jedinica zapisa podataka u RDF-u su uređene trojke (*triple*). Svaku trojku čine:

- **subjekat** - *stvar* na koju se tvrdnja odnosi,
- **predikat** - osobina subjekta i
- **objekat** - vrednost predikata.

Za identifikovanje resursa koriste se internacionalni identifikatori resursa (International Resource Identifier, IRI). IRI je generalizacija uniformnog identifikatora resursa (Uniform Resource Identifier, URI) koja dozvoljava korišćenje karaktera van ASCII[7] standarda [35]. Jedna podvrsta URI-ja su uniformni lokatori resursa (Uniform Resource Locator, URL), koji pored identifikovanja resursa govore i gde se resurs nalazi [79]. IRI se u RDF može naći u svakom od članova trojke.

Trojke je moguće predstaviti i korišćenjem usmerenih grafova. Čvorovi grafa predstavljaju subjekte i objekte, dok označene grane usmerene od čvora subjekta do čvora objekta predstavljaju predikate. Vizuelnu reprezentaciju čvorova takvog grafa i veza između čvorova moguće je raspoređivati na automatizovan način tako da bude pogodna za lako čitanje i razumevanje [132].

Identifikatori koji pripadaju prostoru imena RDF čine standardne identifikatore.

Predikat `rdf:type` označava da objekat treba tretirati kao tip subjekta.

Primer:

```
mpi:Ferrari rdf:Type mpi:Car
```

Predikat `rdf:Property` označava da prvi član trojke treba tretirati kao predikat.

[7] American Standard Code for Information Interchange je nastao kako bi se olakšao prenos teksta teleprinterima, koji su do uvođenja ovog standarda koristili različite sisteme kodovanja u zavisnosti od proizvođača. Standard je izgrađen oko engleskog alfabeta. Za tekstove pisane ovim alfabetom, rešava mnoge praktične probleme, poput lakog pretvaranja malih u velika slova i obratno. Međutim, za pisanje i razmenu teksta koji sadrži simbole van engleskog alfabeta, neophodno je koristiti određenu kodnu stranu, odnosno šemu koja mapira neophodne dodatne karaktere na mesta manje korišćenih karaktera iz gornje polovine ASCII tabele. Ovakvo rešenje, međutim, zahteva da obe strane u komunikaciji znaju koja je kodna strana korišćena, jer se to ne može zaključiti iz samog prenetog teksta. Ove probleme rešava Unicode, koji uz to omogućava i korišćenje pisama i simbola koji ne mogu da se mapiraju na proširene ASCII karaktere, čija je dužina ograničena na 8 bita.

Primer:

```
mpi:isFrontWheelDrive rdf:type rdf:Property
```

Jezik RDF Schema RDF Schema (RDFS) [20, 55] definiše kako treba interpretirati grafove koje definiše RDF, odnosno pruža rečnik za modelovanje podataka. Osnovni koncepti i apstraktna sintaksa definisani su u [29]. Semantika jezika je definisana u [58]. Neke od konkretnih sintaksi su Turtle [11], TriG [23] i JSON-LD [72]. RDFS je proširenje RDF i definisan je korišćenjem RDF sintakse. Definiše mehanizme za opisivanje grupa srodnih resursa i njihovih međusobnih odnosa. Način na koji RDFS definiše klase i osobine entiteta obrnut je u odnosu na uobičajeni pristupa objektno-orijentisane paradigme. Umesto definisanja klase putem osobina koje ona sadrži, RDFS definiše osobine po tome kojim klasama resursa mogu da pripadaju. Jedna posledica ovakvog pristupa je da definicija postojećeg resursa može biti proširena u bilo kom trenutku i od strane bilo koga, što se uklapa u pretpostavku o *otvorenom svetu*.

OWL Web Ontology Language Web Ontology Language (OWL) [32] je jezik namenjen definisanju pojmova i njihove međusobne povezanosti. Ovakva reprezentacija rečnika pojmova i veza među njima naziva se ontologija. Jezik je nastao kao unapređenje jezika DAML+OIL (DARPA Agent Markup Language + Ontology Inference Layer) koji je imao istu namenu. OWL je deo steka koji sačinjavaju XML, XML-S, RDF i RDF-S. U odnosu na njih, OWL omogućuje definisanje disjunktnosti klasa, kardinaliteta, jednakosti, proširuje skup dostupnih tipova osobina klasa i omogućava dodatno opisivanje tih osobina.

OWL nudi veliku ekspresivnost. Velika ekspresivnost jezika donosi sa sobom i neke poteškoće u pogledu obrade elemenata iskazanih jezikom [38]. Stoga, definisana su tri podskupa sa različitim nivoima restrikcije ekspresivnosti:

- **OWL-Lite** omogućava kreiranje hijerarhije klasifikacije i samo jednostavnih ograničenja,
- **OWL-DL** omogućava maksimalnu ekspresivnost do nivoa održane izračunljivosti,
- **OWL Full** pruža potpunu sintaktičku slobodu, ali ne garantuje izračunljivost.

Navodimo neke osnovne koncepte OWL-Lite jezika [89]. Prefiksi ukazuju na to da su ovi koncepti preuzeti iz drugih jezika, poput RDF i RDF-S.

- `Class`. Klasa okuplja grupu *individua* koje imaju zajednički skup *osobina*
- `Individual`. Individue su instance klase.
- `rdfs:subClassOf`. Omogućava kreiranje hijerarhije klasa.
- `rdf:Property`. Osobina je binarna relacija, a u OWL-u su definisane dve podklase:

  - `owl:ObjectProperty`. Osobina koja ukazuje na odnos dve individue.
  - `owl:DatatypeProperty`. Osobina koja ukazuje na odnos individue i tipa podataka

- `rdfs:subPropertyOf`. Osobine, poput klasa, takođe mogu formirati hijerarhije.
- `rdfs:domain`. Domen ograničava na koje individue osobina može da se primeni.
- `rdfs:range`. Raspon ograničava koje individue mogu biti postavljene kao vrednost osobine.

SPARQL (SPARQL Protocol and RDF Query Language) je jezik za zadavanje upita nad RDF ontologijama [57]. SPARQL protokol, koji služi za izvršavanje SPARQL operacija zasnovan je na HTTP protokolu. SPARQL Endpoint je URL putem kog se mogu zadavati SPARQL upiti. Upit se sastoji iz zaglavlja i tela. Telo se sastoji od kolekcije RDF izraza. Postoje sledeći tipovi upita:

- `SELECT` upiti se koriste za dobavljanje sirovih vrednosti u formi tabele,
- `CONSTRUCT` rezultuje RDF grafom nastalim od elemenata dostupnih endpoint-u za koje je tačan uslov upita,
- `ASK` upiti rezultuju odgovorom da li je tvrdnja upita tačna ili netačna (odnosno, da li postoji trojka koja ga zadovoljava),
- `DESCRIBE` upit rezultuje trojkom koja ispunjava uslov upita

Postoje i CREATE, INSERT, UPDATE i DELETE upiti, namenjeni ažuriranju elemenata baze znanja dostupne endpoint-u.

Semantičko rezonovanje

Jedna od ključnih posledica formalne prirode definisanja semantike u semantičkom vebu je mogućnost semantičkog rezonovanja ili rasuđivanja (*semantic reasoning*). Rezonovanje podrazumeva izvođenje logičkih posledica iz skupa činjenica ili aksioma. Postoje brojni softverski alati i biblioteke koji omogućavaju automatizovan proces semantičkog rezonovanja. Pored izvođenja zaključaka, isti alati se obično mogu koristiti i za validaciju ontologija [106]. Postoje i alati koji za rasuđivanje koriste skup zadatih pravila. Ovakav alat naziva se *rule engine*. Moguće je i kombinovanje ova dva pristupa [49]. Iako manipulacija samim ontologijama izlazi van okvira ovog istraživanja, ukratko ćemo prikazati nekoliko implementacija ovih alata, kao osvrt na procese koji se mogu koristiti, kako bi se zadovoljila naša osnovna pretpostavka - postojanje ontologije koja opisuje aplikacije koje se integrišu.

Pellet

Pellet [108, 127] je jedna implementacija *semantic reasoner*-a. Nastao je iz neophodnosti da se podrže neki osnovni principi semantičkog veba, ali i neke praktične potrebe koje se na njemu javljaju. Na primer, neki njegovi prethodnici, iako su implementirali efikasnije algoritme, nisu poštovali pretpostavku nejedinstvenosti imena, nisu omogućavali rad sa XML Schema tipovima podataka i slično. Pellet, između ostalih, podržava sledeće funkcionalnosti:

- analizu i oporavak ontologije - proverava da li ontologija zadovoljava ograničenja koja nameće OWL DL i pomoću niza heuristika pokušava da predloži načine na koji se OWL Full ontologija može svesti na DL restrikciju,
- rezonovanje nad tipovima podataka - omogućava validaciju tipova definisanih XML Schema mehanizmima,
- implikaciju i
- konjuktivne ABox upite.

Jena

Apache Jena je slobodan radni okvir otvorenog koda za rad sa ontologijama u programskom jeziku Java. Njegove osnovne funkcionalnosti opisane su kasnije, u 5.1. Jena uključuje nekoliko internih implementacija *resoner*-a [8]

- tranzitivni - omogućava skladištenje i iteraciju nad strukturama koje sadrže klase i njihove osobine,
- RDFS - implementira konfigurabilan skup RDFS implikacija,
- OWL, OWL Mini i OWL Micro - nepotpune implementacije rasuđivanja nad OWL-Lite i
- generički - rasuđivač koji operiše nad pravilima definisanim od strane korisnika.

Pored ugrađenih, Jena poseduje i generički API za razvoj novih implementacija *reasoner*-a, kao i mogućnost uključivanja eksternih implementacija, poput Pelleta i drugih.

[8] https://jena.apache.org/documentation/inference

3

Pregled trenutnog stanja u oblasti

U prvom delu ovog poglavlja prikazani su klasični pristupi integraciji, pristupi koji koriste principe modelom vođenog inženjerstva, pristupi koji koriste principe semantičkog veba, kao i oni koji kombinuju više pristupa. Neka od navedenih rešenja su namenjena prevashodno za integraciju podataka, što je posebna oblast, ali se njihove ideje mogu koristiti i u integraciji aplikacija.

U drugom delu poglavlja prikazani su alati za spajanje i mapiranje (poravnanje) ontologija. Kako je pretpostavka za upotrebu radnog okvira prikazanog u poglavlju 4 postojanje jedinstvene (objedinjene) ontologije, koja se odnosi na sve aplikacije koje se integrišu, ovi alati su od značaja u slučaju da postoje ontologije koje opisuju pojedinačne aplikacije.

3.1 Pristupi integraciji

Formalna karakterizacija i specifikacija radnog okvira za razmenu konteksta - Context interchange framework, COIN [48] predstavlja posredničku strategiju za pristup podacima u kojoj se semantički konflikti između heterogenih sistema otkrivaju i rešavaju upotrebom kontekstnog posrednika. Svrha posrednika je da poredi kontekste bilo koja dva sistema koji učestvuju u razmeni podataka. Korišćenjem formalizama uvedenih kroz COIN radni okvir semantika pojedinačnog konteksta se može koristiti za rezonovanje o semantičkim razlikama heterogenog sistema u kom taj kontekst učestvuje.

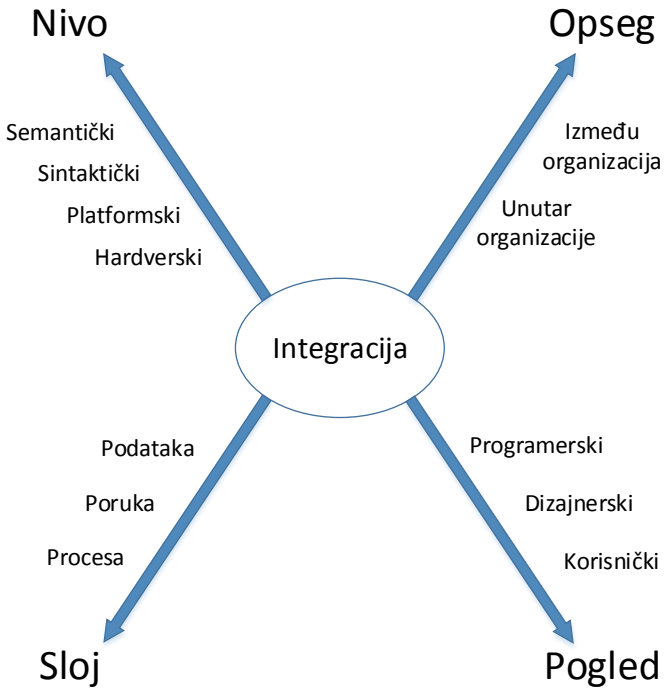
Pregled arhitektura i tehnologija za integraciju distribuiranih poslovnih aplikacija, uz prikaz njihovih mana i vrlina, dat je u [59]. Pod distribuiranom aplikacijom podrazumeva se aplikacija koja se nalazi na više od jednog računara u mreži. Naglašava se da, pored činjenice da su aplikacije koje komuniciraju mrežom heterogene i sama komunikaciona mreža je često heterogena i povezuje razne vrste uređaja različitim medijumima i protokolima. Navodi se da poslovni subjekti koji uspešno sprovedu integraciju svojih informacionih sistema imaju jasnu

prednost zbog bolje mogućnosti da se podaci kojima kompanija raspolaže iskoriste, čime se stvaraju šanse za veću efikasnost i profit. Ističe se i pojava da se tehnike za integraciju poslovnih aplikacija više ne koriste samo unutar jedne organizacije, već i za povezivanje aplikacija koje koriste različite organizacije.

Tehnologije za integraciju autori dele na:

- **Integraciju komunikacionog sloja** - uspostavlja se razmena podataka između aplikacija korišćenjem protokola poput HTTP ili IIOP.
- **Integraciju podataka** - prenos ili povezivanje podataka koji se nalaze u heterogenim izvorima podataka - na različitim računarima, pod različitim operativnim sistemima i sistemima za upravljanje bazama podataka. Obično se prepoznaju izvorna šema, ciljna šema i mapiranje između njih. Izazov koji se pri tom javlja je neophodnost razumevanja velikih i komplikovanih šema, kao i potrebu za održavanjem mapiranja ukoliko se jedna od šema promeni.
- **Itegraciju sloja poslovne logike** - može se podeliti na osnovnu koordinaciju, funkcionalne interfejse, poslovne protokole i nefunkcionalne osobine. Obično se za olakšavanje ove vrste integracije koriste middleware rešenja, koja omogućavaju da se elementi visokog nivoa, apstrakcije izoluju od stalnih promena na hardverskom nivou, u operativnim sistemima, protokolima, itd.
- **Integraciju prezentacionog sloja** - glavni cilj ovakve integracije je kreiranje korisničkog interfejsa koji omogućava upotrebu različitih aplikacija sa jednog mesta, dajući utisak da, inače nezavisne, aplikacije čine jednu celinu.

Još jedna podela vidova integracije data je u [66]. Autori daju pregled sintaktičkih i semantičkih pristupa integraciji. Klasifikacija integracionih stilova organizovana je u odnosu na četiri dimenzije, kao što je prikazano na slici 3.1. Po dimenziji **nivoa** integracija može biti hardverska, platformska, sintaktička i semantička. Integracija na nivou hardvera, na primer omogućava premošćavanje razlika u pogledu fizičkih veličina koje se koriste za predstavljanje signala za prenos ili skladištenje (struja, napon, intenzitet svetla ili radio talasi), vrste modulacije, brzine prenosa i slično. Platformska integracija razrešava način na koji različite platforme tumače ove signale - dužinu reči, redosled bita u reči, redosled reči u slogovima i drugo. Sintaktička integracija se bavi i organizacijom struktura podataka koje koriste sistemi koji se integrišu, dok sintaktička uzima u obzir i značenje, odnosno smisao tih struktura. Po dimenziji **nivoa**, integracija se može obavljati unutar neke organizacije ili između različitih organizacija. Po dimenziji **sloja**, mogu se razmatrati (1) samo podaci koji su ulaz ili izlaz neke aplikacije, (2) poruke koje aplikacija može da razmenjuje sa drugim aplikacijama ili samom sobom ili (3) čitavi procesi u kojima aplikacije učestvuju. Po nivou **pogleda**, integracija se može posmatrati sa aspekta onoga ko vidi rezultat integracije: programer, dizajner (arhitekta softvera) ili krajnji korisnik.

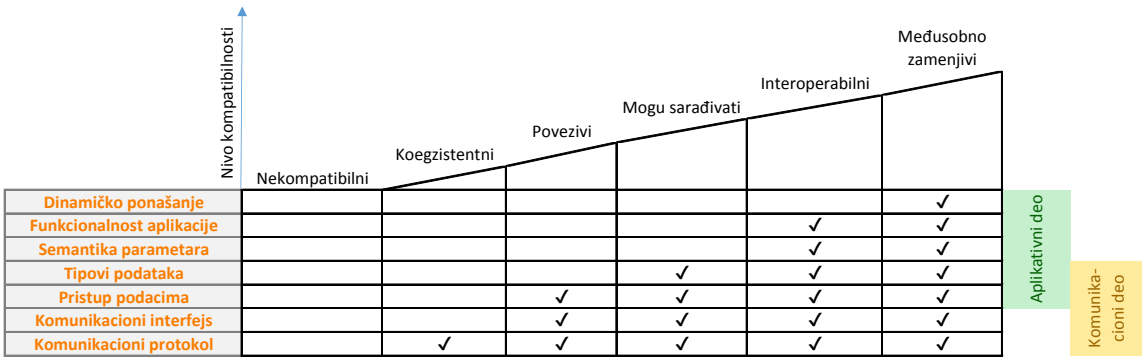


Slika 3.1: Podela vidova integracije u četiri dimenzije po [66]

Standard IEC 61804-1 definiše nivoe interoperabilnosti među sistemima. Prikaz ovih nivoa dat je na slici 3.2. Sistemi se posmatraju u odnosu na komunikacione protokole koje koriste, komunikacione interfejse, način pristupa podacima, tipove podataka, semantiku parametara koji se razmenjuju, funkcionalnost aplikacije i dinamičko ponašanje aplikacije. Prva četiri uslova spadaju u komunikacioni deo, dok uslovi od trećeg do sedmog spadaju u aplikativni deo. Za sisteme koji nisu kompatibilni ni po jednom od ovih osnova se kaže da su u celini nekompatibilni, dok se za sisteme koji su kompatibilni po svim osnovima kaže da su međusobno zamenljivi. Za efikasnu integraciju minimalno je potrebno da sistemi budu povezivi, odnosno kompatibilni po protokolima, komunikacionim interfejsima i pristupu podacima.

Pored rešenja koja imaju za cilj kreiranje univerzalnog pristupa za integraciju aplikacija, postoji i niz usko specijalizovanih. Na primer, alat *Dashboard* [87] namenjen je integraciji, validaciji i vizualizaciji različitih sistema za obradu prirodnih jezika (*Natural Language Processing*), koji su distribuirani i izvršavaju se na heterogenim platformama.

Radni okvir NEGOSEIO [68] ima za cilj uspostavljanje operabilnosti na nivou servisa. Za integraciju koja uzima u obzir razumevanje semantike i poslovne logike koristi referentne ontologije. Predložena metodologija uključuje sledeće korake: (1) prikupljanje znanja, (2) kreiranje modela, (3) kreiranje servisa, (4) puštanje servisa u rad i (5) adaptaciju servisa. Ovi koraci se obavljaju u svakom poslovnom subjektu čiji se sistemi međusobno integrišu.



Slika 3.2: Nivoi interoperabilnosti po IEC 61804-1

Reč je, dakle, o pristupu koji podrazumeva njihovu međusobnu kolaboraciju i kooperaciju. Interoperabilnost se postiže ugovaranjem neophodnih izmena u sklopu pete faze. Referentna ontologija pomaže da se ustanove ove neophodne izmene. Njeno formiranje obavlja se po MENTOR metodologiji, koja obuhvata sledeće faze:

- formiranje leksikona,
  - prikupljanje pojmova,
  - izgradnja pojmovnika (glossary),
  - izgradnja leksikona sinonima (thesaurus),
- formiranje referentne ontologije,
  - prikupljanje ontologija,
  - harmonizacija ontologija i
  - mapiranje ontologija.

Iako se ne bavi direktno integracijom aplikacija, [80] proučava kompozicione konflikte koji se javljaju u sistemima baziranim na komponentama. Jedna od osnovnih ideja ovakvog načina izgradnje sistema jeste mogućnost ponovnog korišćenja komponenti. U praksi, tehničke neusaglašenosti i razlike u specifikacijama mogu dovesti do pojave nekompatibilnosti, koje se mogu svrstati u jednu od tri kategorije: (1) konflikte tipa, (2) konflikte ponašanja i (3) konflikte osobina. Pod osobinama autori podrazumevaju opis strukture i ponašanja na višem nivou apstrakcije, nezavisno od tehničkih detalja i mehanizama komunikacije. Opisan je radni okvir koji podržava:

- obradu komponenti specificiranih različitim standardima i tehnologijama,
- identifikaciju konflikata,
- prevođenje kanoničkih komponenti u konkretne tehnologije i
- ugrađivanje u proces razvoja softvera.

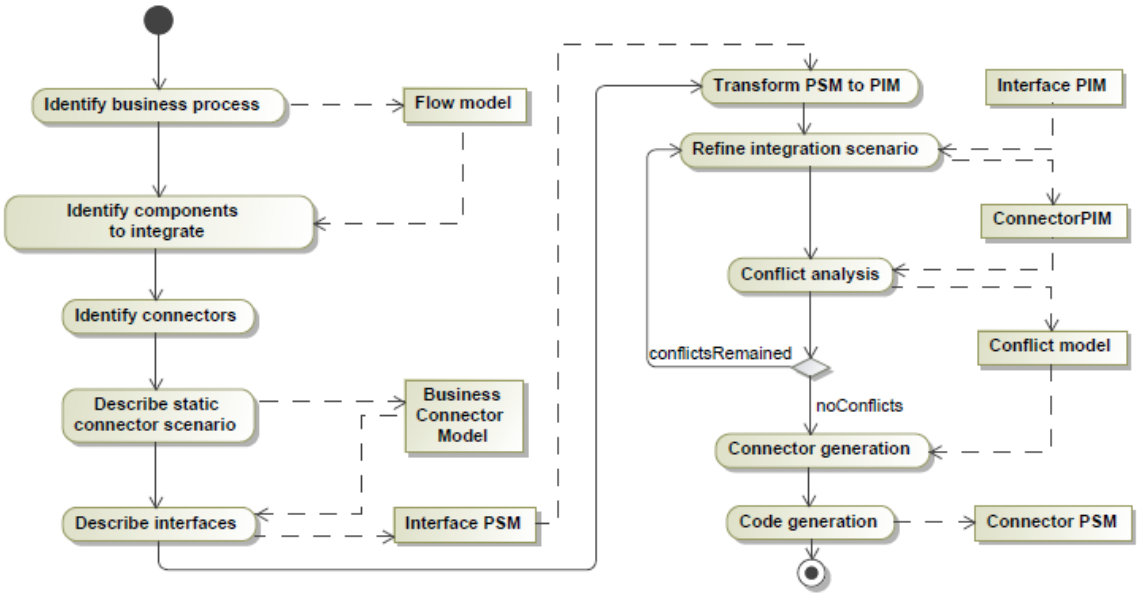
Autori koriste Model-driven architecture (MDA) principe transformacije između modela nezavisnog od platforme (Platform Independent Model, PIM) i modela za konkretnu platformu (Platform Specific Model, PSM). Za provere nad modelima koristi se deduktivna baza znanja, modeli se proveravaju korišćenjem proširenja F-logike zvanog Triple, dok se konflikti tipa i ponašanja detektuju eksternim alatima Haskell, LTSA (Labelled Transition System Analyser)[1], FDR (Failures-Divergences Refinement)[2], fc2tools[3] i Aldebaran[4].

Projekat BIZYCLE [76, 92] predstavlja platformu za interoperabilnost i integraciju softvera i podataka. Platforma je zasnovana na principima modelom upravljanog razvoja softvera. Osnovni pristup se zasniva na rešavanju integracionih problema na višim nivoima apstrakcije. U ovu svrhu koriste se modeli na Computation Independent Model (CIM), Platform Independent Model (PIM) i Platform Specific Model (PSM) nivoima. Značajno je napomenuti da se redosled transformacija ovih modela razlikuje u odnosu na standardni MDA pristup. Kako je MDA namenjen razvoju softvera, polazi se od modela na najvišem nivou apstrakcije - CIM, na osnovu kog se razvija PIM i na kraju PSM. Međutim, pošto je ova platforma namenjena integraciji, polazna osnova su konkretan softverski proizvod ili konkretni podaci, na konkretnoj platformi. Na osnovu njih se izrađuje PSM, od kog se kreiraju modeli na višim nivoima apstrakcije, a oni se zatim koriste za analizu scenarija i generisanje rešenja. Polaznu osnovu čini i poslovni proces u kom učestvuju komponente koje se integrišu i njegova definicija je sastavni deo integracionog scenarija. Celokupan tok razvoja integracionog rešenja dat je na slici 3.3. BIZYCLE platformu čine četiri osnovna dela:

- skup MDA alata,
- repozitorijum - služi za skladištenje modela, meta-modela, specifikacija, ograničenja, dokumentacije, šablona, pravila za transformaciju, konfiguracije i njihovih metapodataka,
- analizator konflikata i
- generator koda konektora.

U [125] opisan je izvršiv JSD za integraciju sistema zasnovanu na razmeni poruka. Jezik omogućava definisanje strukture, tipova i podržanih vrsta poruka, kao i upotrebu integracionih šablona. Uključeni su obrađivači poruka (*Aggregator, Content Enricher, Filter, Content-based Router, Splitter, Timer* i *Transformer*), kanali za prenos poruka (*Point-to-Point* i *Publish-Subscribe*) i šabloni razmena poruka (*Out-Only, Robust Out-Only, Out-In* i *Out-Optional-In*). Omogućeno je i definisanje izraza koji opisuju ponašanje šablona. Integracioni scenario definisan ovim jezikom prolazi kroz niz transformacija: prvo se pretvara

[1] https://www.doc.ic.ac.uk/ jnm/LTSdocumention/LTSA.html
[2] http://www.cs.ox.ac.uk/projects/fdr/
[3] http://www-sop.inria.fr/meije/verification/
[4] https://cadp.inria.fr/



Slika 3.3: Tok specifikacije i generisanja integracionog rešenja na BIZYCLE platformi. Izvor: [76]

u Business Process Execution Language (BPEL) proces, zatim se on transformiše u Java apstraktno sintaksno stablo, zatim se ono pretvara u Java izvorni kod koji se kompajlira i uvezuje sa ostalim komponentama i izvršava na Sun GlassFish ESB.

Primer korišćenja ontologija za komponente dostupne na mreži u cilju njihove integracije prikazan je u [130]. U prikazanoj studiji slučaja integrisani su različiti izvori koji daju geoinformacione podatke od interesa jedinicama na terenu. Dodavanjem klasa postojećoj ontologiji, integracija je implementirana u vremenskom opsegu od nekoliko sati. U ovom primeru cilj nije bila automatizacija, niti je razvijan radni okvir, već su ontologije korišćene ručno. Ipak, i ovakav pristup pruža korisna saznanja u pogledu korišćenja OWL ontologija u cilju poboljšanja procesa integracije.

Pristup integraciji koji kombinuje korišćenje veb servisa i ontologija nazvan ODSOI (Ontology-Driven Service-Oriented Integration) dat je u [67]. Autori sugerišu pogodnu topologiju servisa i ontologija, uz viziju radnog okvira za integraciju. Radni okvir omogućava i *dinamičku* integraciju, što znači da je spajanje servisa ciljnih poslovnih informacionih sistema moguće za vreme izvršavanja (runtime).

Jezik specifičan za domen integracije poslovnih informacionih sistema, pod nazivom Highway, predstavljen je u [75]. U pitanju je interni JSD, baziran na jeziku Clojure. Highway

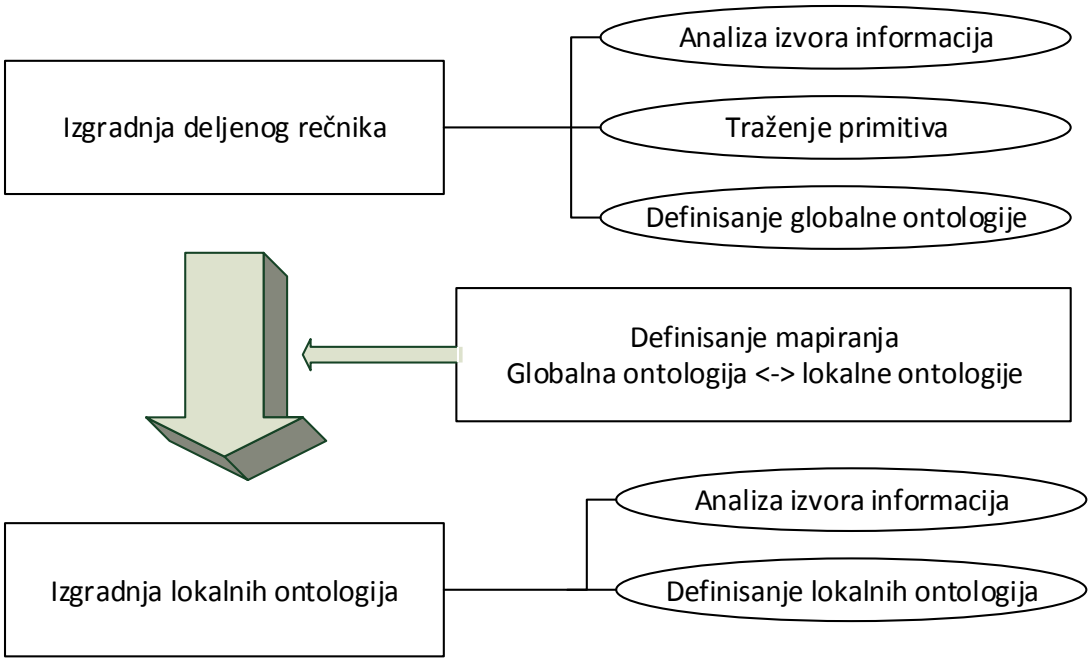
suštinski predstavlja još jednu konkretnu sintaksu[5] za Apache Camel platformu, na koju se prevodi. Jezik olakšava razvoj korisnicima koji su navikli na funkcionalne jezike i omogućava korišćenje ove paradigme u kontekstu razvoja integracionih rešenja.

Deseto poglavlje knjige [26] predstavlja prikaz arhitektonskih obrazaca za semantičku integraciju, zasnovanih na ontologijama.

U [5] razmatrana je detekcija semantičkih konflikata u veb servisima i servisno orijentisanim arhitekturama (SOA) uopšte, u slučajevima kada se razmenjuju heterogeni podaci. Kao primer semantičkog konflikta, navodi se slučaj dva javna veb servisa za dobavljanje geodemografskih podataka, koji adrese (koje su za dati domen podatak od primarne važnosti) prezentuju na drugačiji način. Ističe se da, iako je za SAOP veb servise dostupna precizna WSDL definicija strukture poruka, ona nije dovoljna za uspostavljanje mapiranja i detekciju konflikata između servisa koji različito tumače podatke. Rešenje koje predlažu autori koristi dve komponente: (1) ontologiju, koja pruža semantičke interpretacije, reprezentacije i strukturu poruka i (2) klasifikaciju semantičkih konflikata, kao vodilju za identifikaciju vrste semantičkog konflikta. Koristi se modifikovana klasifikacija nivoa heterogenosti poruka (Message Level Heterogeneities) iz [97]. Rešenje je nezavisno od domena primene.

Pristup integraciji podataka zasnovan na ontologijama razmatran je u [21]. Postupak obuhvata faze prikazane na slici 3.4: (1) izgradnju deljenog rečnika, (2) izgradnju lokalnih ontologija i (3) definisanje mapiranja među konceptima globalnih i lokalnih ontologija definisanih u prve dve faze. Prva faza dalje obuhvata sledeće korake: (1a) analizu izvora informacija, (1b) traženje pojmova, odnosno *primitiva* i (1c) izgradnju globalne ontologije. Druga faza obuhvata korake: (2a) analizu izvora informacija i (2b) definisanje lokalnih ontologija. Svaka od ovih faza ima za cilj otkrivanje heterogenosti među podacima i načina za njeno premošćavanje.

[5]Sama Apache Camel platforma definiše nekoliko ugrađenih načina za specificiranje ruta: Camel JSD, Java JSD, Spring, Rest JSD, Scala JSD.



Slika 3.4: Faze izgradnje i mapiranja ontologija pri integraciji podataka po [21]

Metodologija integracije poslovnih aplikacija IRIS data u [25] daje smernice za proces integracije u velikim poslovnim sistemima, uz isticanje raznih pogleda na interoperabilnost: poslovni, procesni, pogled u odnosu na ljudske resurse, tehnološki, pogled u odnosu na znanje i u odnosu na značenje (semantiku). Metodologija definiše faze, aktivnosti, zadatke, način njihovog izvršavanja i učesnike. Predviđene su sledeće faze uz pripadajuće aktivnosti:

1. konceptualna definicija
   (a) strateška definicija
   (b) definicija procesa kolaboracije
2. modelovanje kolaboracije
   (a) globalno modelovanje
   (b) semantičko poravnanje
   (c) modelovanje scenarija kolaboracije
3. dijagnostika i predlozi za unapređenje
   (a) merenje sposobnosti interoperabilnosti i definicije projekata za poboljšanje interoperabilnosti
   (b) modelovanje scenarija kolaboracije
4. razvoj

   (a) razvoj projekata interoperabilnosti

5. implantacija[sic]

   (a) zajedničke radne sesije sa korisnicima

   (b) implantacija tehnološke platforme

6. izvršavanje i monitoring

   (a) monitoring sistema

Doktorska teza [43] izučava interoperabilnost velikih i kompleksnih skupova podataka iz domena zdravstva. Za integraciju se koristi pristup baziran na ontologijama. Lokalne ontologije pojedinih izvora podataka mapiraju se na globalnu ontologiju. Poseban akcenat se stavlja na podmirivanje potreba korisnika na radnim mestima gde je neophodna saradnja sa drugim organizacionim jedinicama ili ustanovama (boundary spanning roles). Ovo čini celokupan pristup pogodan i za primene u drugim domenima, osim zdravstvenog.

U [112] data je ontologija za razrešavanje semantičkih konflikata (Semantic Conflict Resolution Ontology, SCROL). Ontologija je namenjena integraciji heterogenih baza podataka i rešavanju konflikata na nivou šeme. Razvijena je i praktična implementaciju alata koji koristi predloženi pristup. Alat je nazvan CREAM (Conflict Resolution Environment for Autonomous Mediation).

Arhitektura za semantičke standarde poslovnih aplikacija predstavljena u [8] oslanja se na automatizovano prevođenje XML šema u ontologije na OWL-DL jeziku. Cilj pristupa je da se omogući automatska provera kompatibilnosti ograničenja i pravila različitih standarda. Ovi standardi su obično zadati kao XML šeme, uz eksterna sintaktička ograničenja i pravila data jezicima poput Schematron-a. Tradicionalni način za integraciju ovih standarda se sastoji od (1) ručne identifikacije sintaktičkih i semantičkih podudaranja i konflikata i njihovom rešavanju, (2) kreiranju eXtensible Stylesheet Language Transformations (XSLT) transformacije sa izvornog na ciljni format i obrnuto, (3) primene transformacije i (4) validacije testom ekvivalencije. Validacija i postupak nisu trivijalni, jer postoje slučajevi da su neki elementi standarda semantički jednaki, a sintaktički različiti (npr. različit redosled, različit način zapisa vremenskih odrednica itd). Pretpostavka za korišćenje automatizovanog pristupa jeste da postoje ontologije koje opisuju pojedine standarde, da standardi koriste istu terminologiju, te da je moguće načiniti objedinjenu ontologiju. Postupak primene metodologije se sastoji od 11 koraka, od kojih neki u određenim slučajevima zahtevaju ručne intervencije, a obuhvataju prevođenje XML Schema Definition (XSD) šeme u OWL ontologiju, validacije zasebnih ontologija svakog standarda, formiranje zajedničke ontologije i njene provere i korišćenja alata za rezonovanje. Autori zaključuju da je korišćenje tehnologija semantičkog veba

moguće u kontekstu integracije poslovnih aplikacija, da su te tehnologije zrele, ali da **nedostaju dovoljno robusni alati koji podržavaju proširivanje dodacima specijalizovanim za konkretne kompleksne zahteve koji se tipično javljaju u industriji**.

Doktorska teza [34] prikazuje radni okvir za integraciju heterogenih tehničkih prostora, zasnovan na principima razvoja softvera vođenog modelima (RSVM). Tehnički prostor (TP, technical space) definisan je kao „radni kontekst koji obuhvata skup pridruženih koncepata, znanja, resursa, potrebnih veština i alata“. Cilj radnog okvira je automatizacija dela procesa integracije softvera vezanog za Industriju 4.0 i srodne oblasti, sajber-fizičke sisteme (Cyber-Physical Systems, CPS) i Internet stvari (Internet of Things, IoT). Prikazan je namenski jezik (jezik specifičan za domen) za modelovanje i pristup zasnovan na RSVM principima, čijom primenom je moguće rešiti probleme heterogenosti u integraciji tehničih prostora. Pokazano je da je moguće kreirati strukturu tipa grafa, pomoću koje se predstavljaju šeme podataka (meta-modeli) tronivovskih tehničkih prostora, koja sadrži veze sa originalnim elementima šema podataka. Teza sadrži analizu i pregled postojećih alata za integraciju. Alati su analizirani sa aspekta licenci za korišćenje, načina distribucije i aktuelnosti održavanja, domena primene, načina na koji se mapiranje obavlja i jezika kojim se formiraju izrazi za mapiranje, mogućnosti ponovnog korišćenja delova integracionih rešenja, mogućnosti i načina izvršavanja rešenja, kao i mogućnosti proširenja alata. Analizirani su alati:

- opšti alati za mapiranje:
  - Altova MapForce,
  - AnalytiX Mapping Manager,
  - FME Desktop,
  - OPC Router,
  - Open Mapping Software,
  - MetaDapper,
  - MuleSoft Anypoint Studio i
  - Vorto;
- alati za mapiranje XML-a:
  - Liquid XML Studio i
  - Stylus Studio;
- Extract, Cleanse, Transform and Load (ECTL) alati:
  - Adeptia Integration Suite,
  - CloverETL,
  - Informatica PowerCenter,
  - Karma,
  - Microsoft SQL Server Integration Services,
  - OpenRefine,
  - Oracle Data Integrator i
  - Talend Studio.

U pogledu pristupa mapiranju, alati su podeljeni na direktne (67%) i indirektne (33%). Kod direktnih, korisnik vidi samo izvorni i ciljni tehnički prostor, dok je tehnički prostor samog alata sakriven. Kod indirektnog pristupa, uvodi se medijatorski tehnički prostor, pa korisnik prvo mora izvorni i ciljni tehnički prostor da mapira na medijatorski, a zatim se unutar njega kreiraju transformacije. U pogledu načina zadavanja mapiranja, identifikovane su četiri vrste sintakse: grafička, tekstualna, tabularna i zasnovana na konfiguraciji. Od analiziranih alata 72% podržava samo jednu vrstu sintakse. Najčešće je podržana grafička sintaksa (56%), dok sintaksu zasnovanu na konfiguraciji podržava 11% alata, a tekstualnu samo jedan alat (6%). Tabularna sintaksa ni u jednom alatu nije bila jedina podržana, već se koristi u kombinaciji sa nekom drugom vrstom. U pogledu ponovnog korišćenja, analizirane su mogućnosti vezane za tri vrste koncepata: ponovno korišćenje korisnički definisanih funkcija (što podržava 61% alata), ponovno korišćenje kreiranih mapiranja (28%) i ponovno korišćenje specifikacije (28%).

Pristup izložen u doktorskoj tezi [34] je i praktično implementiran u alatu nazvanom AnyMap. Proces izrade integracionog rešenja ovim pristupom sastoji se od: (1) uvoza TP, (2) specifikacije mapiranja i (3) generisanja integracionih adaptera. Uvoz TP može biti obavljen ručno ili poluautomatski, ekstrakcijom šeme iz skupova podataka. Faza specifikacije mapiranja može biti obavljena ručno, poluautomatski ili potpuno automatski. Za automatizaciju ovog koraka na raspolaganju su dva algoritma: (1) ponovno korišćenje ranije kreiranih mapiranja, koja se čuvaju u repozitorijumu alata i (2) algoritam poravnanja. Algoritam poravnanja poredi parove ulaznih i izlaznih elemenata i za svaki par pronalazi verovatnoću u intervalu [0,1) da ti elementi treba da budu mapirani. Verovatnoća se računa kao prosečna vrednost verovatnoće koju daju više algoritama za računanje sličnosti među datim parom elemenata. Alat nema mogućnost zadavanja semantike interfejsa koji se mapiraju, pa time ni mogućnost detekcije semantičkih konflikata.

U [128, 42] predstavljen je Guaraná JSD, grafički jezik specifičan za domen dizajniranja EAI rešenja. Osnovni koncepti jezika, delom inspirisani integracionim šablonima iz [62], su: (1) **gradivni blok** (building block), komponenta koja može da prima i šalje poruke, a sastoji se od zadataka, (2) **zadatak**, element koji čita poruku sa jednog slota, obrađuje je i šalje na sledeći slot, (3) **slot**, element koji se koristi unutar gradivnih blokova kako bi omogućio razmenu poruka između različitih zadataka, kao i između zadataka i portova, (4) **port**, apstrakcija komunikacije spoljne aplikacije i gradivnih blokova i (5) **integraciona veza**, koja se interno koristi za transport poruka između gradivnih blokova. Na osnovu specifikacije integracionog rešenja izrađenog ovim jezikom, moguće je generisati programski kod implementacije rešenja. Referentna implementacija generiše Microsoft Windows Workflow Foundation[6] radne tokove, XML konfiguracione datoteke i pomoćne C# klase.

U [41] izložen je Guaraná SDK, koji se sastoji od radnog okvira i skupa pomoćnih alata. Cilj autora bio je da razviju okruženje koje predstavlja implementaciju integracionih šablona

[6] https://msdn.microsoft.com/en-us/library/jj684582.aspx

iz [62], ali je lakše za održavanje u odnosu na postojeća, naročito u pogledu adaptivnog održavanja, odnosno mogućnosti prilagođavanja novim kontekstima. U tom pogledu, predloženi SDK su poredili sa postojećim alatima Apache Camel[7] i Spring Integration[8], koji takođe predstavljaju implementacije pomenutih šablona. Za procenu lakoće održavanja su korišćene statističke metrike: broj paketa, broj klasa, broj interfejsa, broj metoda u klasama i interfejsima, broj linija koda, broj linija koda po metodi, broj parametara u metodi, nivo kohezije klasa po Henderson-Sellers LCOM* metodi [60] i cirkularna složenost (cyclomatic complexity) [88]. Rezultati ovih metrika znatno su povoljniji u korist Guaraná alata. Bitno je istaći da se ova merenja odnose na same alate, a ne na integraciona rešenja izvedena pomoću njih.

Još jedan dodatak Guaraná familije je alat koji omogućava simulaciju rada projektovanog integracionog rešenja [117]. Simulacija konceptualnog modela omogućena je korišćenjem Petrijevih mreža [95]. Prevođenje iz Guaraná konceptualnog modela u stohastičku Petrijevu mrežu predstavlja horizontalnu, egzogenu transformaciju [119, 90] (transformaciju na istom nivou apstrakcije, gde su polazni i ciljni jezik različiti). Simulacija omogućava identifikovanje karakteristika integracionog rešenja, bez neophodnosti za izvršavanjem na realnim sistemima ili replikama takvih sistema, čime se celokupan proces razvoja čini jeftinijim. Ovim je omogućeno poređenje različitih rešnja, kao i identifikacija mogućih problema ili grešaka u dizajnu pre izrade prototipa i konačne implementacije.

Korišćenje ontologije za semantičku integraciju podataka prisutno je i u oblasti analize velike količine podataka (*big data*). U [31] autori definišu radni okvir za realizaciju pristupa zvanog Ontology-Based Data Access (OBDA), uvedenog u [111]. *OBDA specfikacija* $\mathcal{J}$ je definisana kao trojka

$$\langle \mathcal{O}, \mathcal{S}, \mathcal{M} \rangle, \tag{3.1}$$

gde je:

- $\mathcal{O}$ - ontologija, obično definisana kao *Description Logic TBox* [10],
- $\mathcal{S}$ - šema relacija i
- $\mathcal{M}$ - mapiranje $\mathcal{S}$ na $\mathcal{O}$, odnosno niz tvrdnji, gde svaka predstavlja relaciju između nekog upita nad šemom i nekog upita nad ontologijom.

Za jezik ontologije u radnom okviru bira se *DL-Lite*$_A$, kao najekspresivniji iz *DL-Lite* familije (pri čemu se ističe da je ekspresivnost ovog jezika načelno dovoljna da se obuhvate koncepti

[7] http://camel.apache.org/
[8] https://spring.io/projects/spring-integration

Tabela 3.1: Problemi prilikom uvođenja ERP rešenja [131]

Tip problema	Udeo
Troškovi projekta veći od planiranih	66%
Kašnjenje projekta	58%
Neusaglašenost sa poslovnim strategijama	42%
Otpor zaposlenih ka promenama	42%
Nesuglasice sa konsultantima	38%
Unutrašnje nesuglasice	34%
Nesuglasice sa dobavljačima	30%

Entity relationship (ER) dijagrama i UML dijagrama klasa), ili *DL-Lite*$_R$, koji je osnova i OWL 2 QL profila. Jezik mapiranja je definisan tvrdnjama oblika

$$\phi(x) \rightsquigarrow A(f(x)) \qquad \phi(x) \rightsquigarrow P(f_1(x_1), f_2(x_2)) \tag{3.2}$$

Radni okvir dalje definiše način na koji se formiraju korisnički upiti, mehanizam formiranja odgovora na upite, mehanizam prerade upita (query rewrite) u odnosu na ontologiju i mehanizam prerade upita u odnosu na mapiranje. Kao jednu od prepreka za trenutno korišćenje radnog okvira, autori navode **nedostatak prikladnih alata** i metodologija, naročito za definisanje mapiranja, navodeći da postojeći i predloženi alati za mapiranje šema nisu adekvatni za OBDA.

Sistemi za planiranje poslovnih resursa (Enterprise Resource Planing, ERP) zamišljeni su kao integralno rešenje koje pokriva aspekte finansija, zaposlenih (*ljudski resursi*), nabavke, proizvodnje, distribucije i prodaje, čime bi zamenili ranije korišćene aplikacije, koje su bile zadužene za svaki od ovih aspekata poslovanja ponaosob. [46] Nakon desetak godina praktične primene raznih ERP aplikacija, autori su sakupili iskustva poslovnih korisnika u vezi sa ovim sistemima [131]. Poteškoće u vezi sa uvođenjem i eksploatacijom koje su navodili ispitanici sumirane su u tabeli 3.1. Takođe, navedeno je da 58% kompanija nije integrisalo ERP sa postojećim sistemima, od čega je 23% navelo da je pokušaj integracije bio neuspešan, a 71% nisu ni pokušali integraciju, ocenjujući je kao "kompleksan, skup i dugotrajan proces".

3.2 Alati za spajanje i mapiranje ontologija

U [45] dat je pregled alata za spajanje i mapiranje ontologija. Od 97 pronađenih alata, izdvojeno je sedam, koje su autori okarakterisali kao *pogodne za praktičnu upotrebu* u svom projektu INTER-IoT. Kriterijumi odabira su sledeći: dostupnost veb sajta i datum

njegovog poslednjeg ažuriranja, broj publikacija i datum poslednje publikacije, dostupnost izvornog koda i dokumentacije, korišćene tehnologije i mogućnost korišćenja različih formata podataka, reference korišćenja alata u akademskim i komercijalnim projektima i skalabilnost. Iako to nije bio kriterijum, primećuje se da je većina alata razvijana na Javi. Izdvojeni su sledeći alati:

LogMap [70] je razvijen na Oksfordskom univerzitetu. Alat omogućava da se iz komandne linije, ili putem Ajax veb interfejsa, za zadate ontologije u formatima koje podržava OWL API [63] dobije kao izlaz poravnanje između klasa, osobina i instanci. Omogućava proveru konzistentnosti primenom Daulin-Galijer (Dowling-Gallier) algoritma. [120]

Alignment API [30] definiše format za zapisivanje poravnanja ontologija korišćenjem RDF, kako bi se omogućilo njihovo skladištenje, razmena i deljenje. Zamišljen je kao nezavisan od alata, što potvđuje lista od oko stotinu alata kompatibilnih sa ovim API-jem. API definiše funkcije za odabir algoritama za mapiranje, prevođenje upita, pretragu postojećih poravnanja, njihovu manipulaciju, prikaz u raznim jezicima itd. Referentna implementacija, *Alignment Server*, omogućava pozivanje ovih funkcija putem SOAP ili Representational state transfer (REST) veb servisa, kao i putem FIPA jezika za komunikaciju agenata. [40]

Silk Framework [137] se razlikuje od ostalih alata, po tome što ne radi na nivou šema, već na nivou podataka i omogućava pronalaženje veza između njih. Korisnik definiše kriterijume veze kroz XML, CSV, RDF ili Silk-LSL, podaci se dobavljaju SPARQL upitom, a rezultati se mogu snimiti zasebno ili spojiti sa postojećim podacima.

COMA [9] predstavlja fleksibilan radni okvir za mapiranje ontologija, potekao sa Univerziteta u Lajpcigu. Alat iterativno koristi raspoložive algoritme za određivanje mapiranja. Korisnik može po potrebi da uključuje i isključuje ove algoritme, da svakom rezultatu dodeljuje nivo pouzdanosti, kao i da obriše rezultujuće mapiranje ili ručno doda novo.

AgreementMaker [37] je još jedan radni okvir u kom se kriterijumi mapiranja mogu proširivati, konfigurisati i dodavati. Neki od implementiranih mapera koriste i spoljašnje servise, poput leksičke baze WordNet [93]. Nije podržano mapiranje individua, već samo klasa i osobina. Ulazne ontologije mogu biti u OWL, OBO ili SKOS formatu, a rezultat u Alignment API formatu.

S-Match [47] je radni okvir koji omogućava ekstrakciju ontologija iz kataloga, stabla, konceptualnih modela i drugih struktura, a zatim vrši semantičko poređenje i mapiranje ovih ontologija. Koristi se u više od deset projekata.

OntoBuilder [44] je skup alata koji omogućavaju ekstrakciju ontologija na osnovu veb stranica i sukcesivno mapiranje na jedinstvenu ontologiju. Dostupan je grafički prikaz mapiranja, koja korisnik može da pregleda i odabere.

3.3 Alati za integraciju

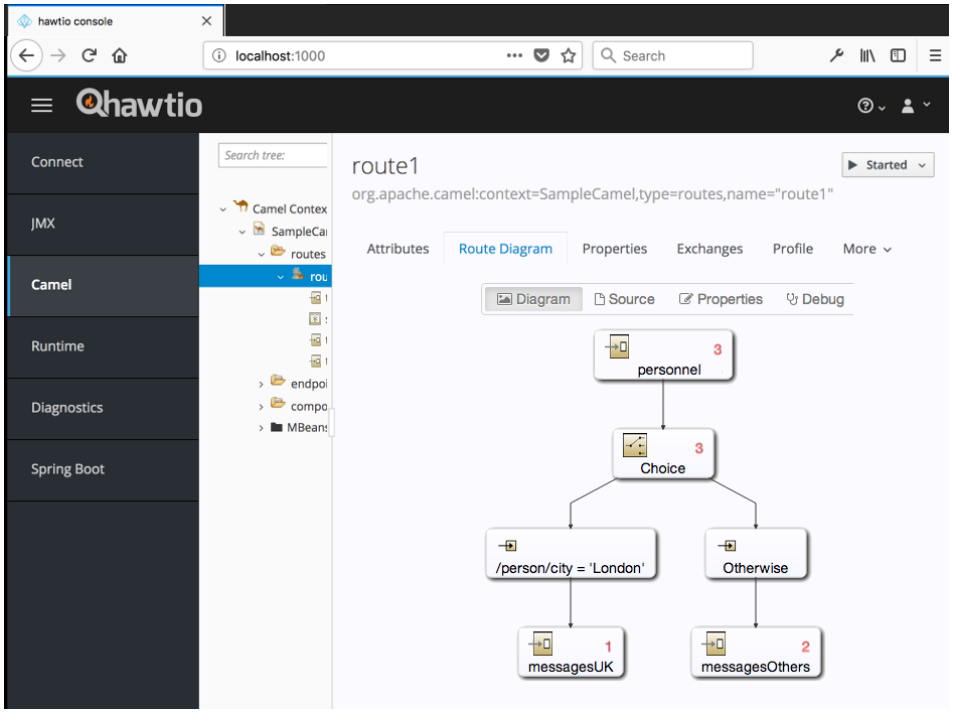
U ovom delu biće prikazani neki alati za integraciju koji se često koriste u praksi. Rezime poglavlja daje pregled analiziranih alata uz fokus na proces mapiranja koji je prisutan kao sastavni deo celokupnog procesa integracije u svim alatima i čija automatizacija predstavlja predmet ovog istraživanja.

3.3.1 Apache Camel

Apache Camel je radni okvir namenjen izradi integracionih rešenja. Može se reći da Camel velikim delom predstavlja implementaciju integracionih šablona opisanih u [62]. Ovaj radni okvir je modularan, može se proširiti dodacima (plugin), otvoren (pod Apache licencom), sa dobrom podrškom aktivnih korisnika i dobrim resursima za učenje, poput [64]. Može raditi samostalno, biti ugrađen u druge aplikacije ili radne okvire, izvršavati se u okviru aplikativnih servera ili koristiti njihove usluge (JBoss, Tomcat, Spring, MINA, itd). Camel nije zamišljen kao Enterprise Integration Bus. Razvoj integracionog rešenja u Camel-u sastoji se od izgradnje ruta koje povezuju različite Camel komponente: Endpoint, Processor, Producer, Consumer, Message Translator, itd. Rute se definišu u jednom od raspoloživih internih JSD-ova: Java, Spring, Blueprint, Groovy, Scala. Svaki od ovih JSD-ova ima konkretnu sintaksu koja je osmišljena tako da se najbolje uklopi u okruženje ili jezik u okviru kojih se koristi. Na primer, Camel Java JSD za specifikaciju ruta koristi ulančane pozive metoda (tzv. fluid API), Camel Spring JSD koristi isti format kojim se inače specificiraju Spring komponente. Camel koristi načelo *konvencija umesto konfiguracije* – za svaku komponentu podrazumevana su podešavanja koja odgovaraju najvećem broju slučajeva korišćenja. Ukoliko je potrebno promeniti neke od parametara, to se može učiniti u okviru URI-ja kojim se komponenta i identifikuje. Za velik broj formata na raspolaganju su komponente koje vrše konverziju iz jednog formata u drugi. Za formate koji nisu podržani, kao i za implementiranje poslovne logike koja se ne može drugačije realizovati, moguće je razviti sopstvene komponente koje se zatim dalje povezuju rutama i koriste u okviru integracionog rešenja. Komponente mogu implementirati neki od Camel interfejsa ili biti realizovane kao bean-ovi, kako bi mogle biti korišćene i van Camel-a. Camel poseduje sopstveno okruženje za testiranje razvijenih rešenja, bazirano na JUnit radnom okviru. Ono uključuje i gotove mockup komponente, logere, kao i alate za praćenje i presretanje poruka. Na raspolaganju su i alati za upravljanje radom rešenja u eksploataciji.

Apache Camel nudi široke mogućnosti za implementaciju integracionog rešenja. Koncepti internog JSD-a se dobro mapiraju na jezik integracionih šablona. Iako je kod pisan ovim jezikom lako čitljiv, mogu se javiti poteškoće u sagledavanju implementacija komplikovanijih scenarija. Ovo je svojstveno za sve tekstualne reprezentacije složenih struktura. Od pomoći mogu biti i alati poput JBoss Camel Developer Tools ili Hawtio[9] (prikazan na slici 3.5), koji omogućavaju vizuelizaciju Camel ruta u vidu grafičkih dijagrama.

[9] https://hawt.io/

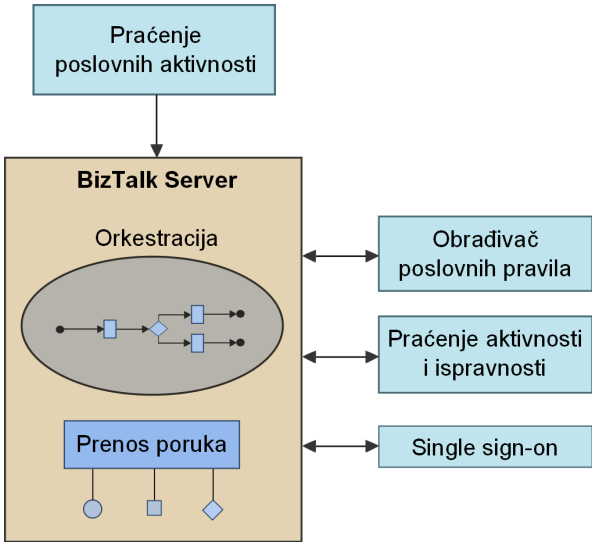


Slika 3.5: Vizualizacija rute u alatu Hawtio

3.3.2 Talend Open Studio

Aplikacija Talend Open Studio je deo familije proizvoda namenjenih raznim vidovima integracije. Osnovne verzije ove aplikacije i pratećih alata dostupni su pod Apache slobodnom licencom i besplatne su za korišćenje. Postoje i dve edicije dostupne uz pretplatu: *Entry Level* i *Platform Edition*, koje uključuju podršku i dodatne alate za razvoj, administraciju, monitoring, pripremu podataka, merenje kvaliteta podataka, grupni rad i redundantno izvršavanje integracionog rešenja. Aplikacija je dobro dokumentovana i ima veliku bazu korisnika. Sve edicije, uključujući besplatnu, sadrže više od 900 predefinisanih konektora [10]. Ovi konektori omogućavaju uvoz i izvoz podataka ka i od širokog spektra poslovnih paketa, SaaS (Software as a Service) rešenja, raznih protokola, sistema za upravljanje bazama podataka i formata datoteka, među kojima su: SAP Business Suite, Sage X3, Sugar CRM (Customer relationship management), CentronicCRM, Microsoft Dynamics; Marketo, Salesforce Wave, NetSuite; Amazon S3, Google Drive, Amazon SQS (Amazon Simple Queue Service), Elastic Search, JIRA; Microsoft SQL Server, PostgeSQL, Informix, MySQL; REST, SOAP; XML, CSV, XLS; E-mail (POP3/SMTP/IMAP), FTP, SFTP, LDAP. Jedan primer Big Data integracije korišćenjem paketa Talend Open Studio prikazan je u [94].

[10] Spisak predefinisanih konektora, kategorizovanih po verziji i ediciji: https://www.talendforge.org/components/index.php



Slika 3.6: Pregled BizTalk arhitekture

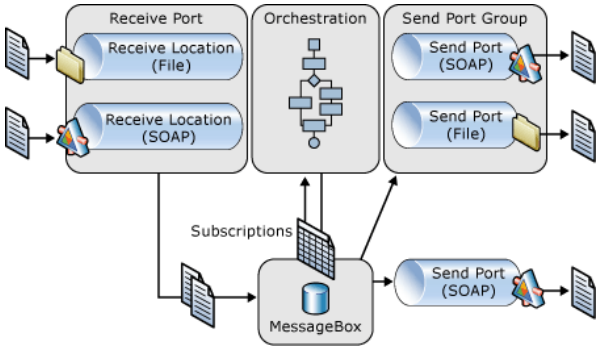
3.3.3 Microsoft BizTalk

Microsoft BizTalk je komercijalna platforma namenjena integraciji poslovnih aplikacija[11]. Arhitektura BizTalk servera zasniva se na publish-subscribe obrascu [101]. Pregled arhitekture dat je na slici 3.6. BizTalk omogućava implementaciju poslovnih procesa orkestracijom, upravljanje definisanim procesima i njihovo praćenje, poseduje mehanizme za postizanje visoke dostupnosti, konzolu za upravljanje artefaktima, kao i alate za povezivanje sa drugim poslovnim entitetima (business-to-business). Podaci se primaju i šalju u vidu poruka. Samo povezivanje infrastrukture sa krajnjim aplikacijama koje se integrišu obavlja se putem adaptera. U BizTalk arhitekturi, adapter je komponenta koja implementira mehanizam isporuke poruka korišćenjem nekog komunikacionog standarda ili formata. Nativni adapteri, koji se isporučuju uz aktuelnu verziju BizTalk server-a su: FILE, FTP, HTTP, MQSeries, MSMQ, POP3, SMTP, SOAP, Windows Sharepoint Services, WCF-WSHttp, WCF-BasicHttp, WCF-NetTcp, WCF-NetMsmq, WCF-NetNamedPipe, WCF-Custom i WCF-CustomIsolated. Moguć je i razvoj drugih adaptera korišćenjem BizTalk Adapter radnog okvira[12]. Životni ciklus poruke prikazan je na slici 3.7.

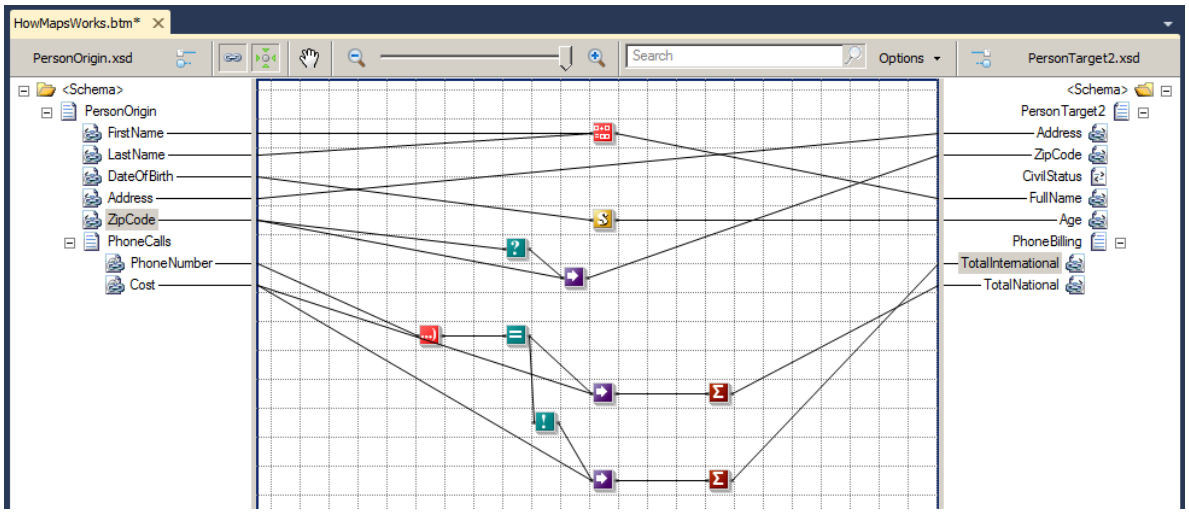
Za razvoj integracionih rešenja koja se izvršavaju na BizTalk serveru kreiraju se projekti upotrebom integrisanog razvojnog okruženja Visual Studio. BizTalk projekat može sadržati sledeće delove[13].

Orkestraciju Orkestracija je reprezentacija poslovnih procesa. Može biti opisana jezicima opšte namene iz .NET familije, poput Visual C# ili Visual Basic .NET ili jezikom XLANG/s. XLANG/s je JSD namenjen opisu procesa. Koristi konstrukte standarda poput XSD, XSD i WSDL i poseduje podršku za direktno korišćenje .NET objekata i poruka. [102] Uvodi

[11] https://www.microsoft.com/en-us/cloud-platform/biztalk
[12] https://docs.microsoft.com/en-us/biztalk/core/what-is-the-adapter-framework
[13] https://docs.microsoft.com/en-us/biztalk/core/adding-project-items



Slika 3.7: Životni ciklus poruke u BizTalk okruženju. Izvor: docs.microsoft.com



Slika 3.8: Radno okruženje alata BizTalk Mapper. Izvor: social.technet.microsoft.com

podršku i za konstrukte visokog nivoa, kao što su: poruka, port, korelacija ili veza servisa.

Šeme Šema predstavlja definiciju strukture dokumenta ili poruke. Može imati kompozitnu strukturu, odnosno sadržati druge šeme.

Mape Mapa daje relaciju između polja neke dve šeme. Mapiranje se obavlja XSLT jezikom. Za lakšu manipulaciju, na raspolaganju je alat BizTalk Mapper. Ovaj alat omogućava grafički prikaz i manipulaciju vezama između šema i njihovim međusobnim transformacijama. Korisnički interfejs alata prikazan je na slici 3.8.

Protočne obrade Protočna obrada obuhvata infrastrukturu koja definiše i povezuje jednu ili više faza u obradi poruka koje šalje ili prima BizTalk server.

3.3.4 Oracle Integration Cloud

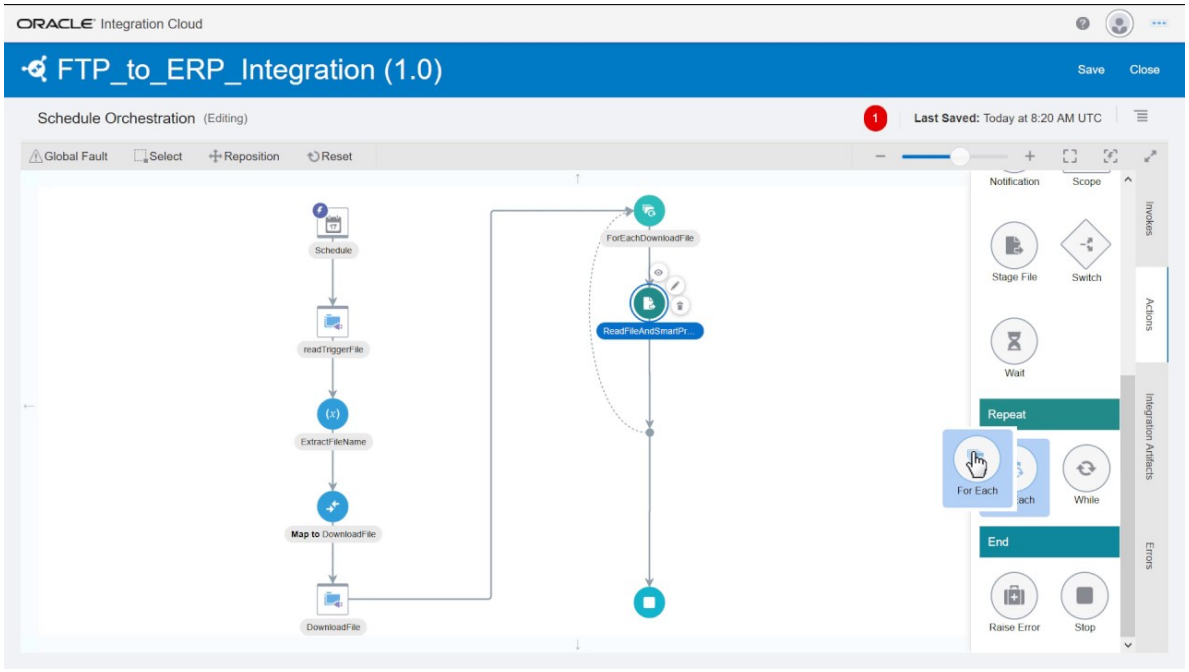
Oracle Integration Cloud (OIC) je platforma za integraciju aplikacija i automatizaciju procesa *u oblaku*. Pod ovime, podrazumevamo da se sama platforma izvršava u oblaku, da se njom upravlja u oblaku i da je namenjena integraciji prevashodno aplikacija koje su i same u oblaku, mada se može koristiti i za integraciju ovih aplikacija sa tradicionalnim rešenjima, koja se izvršavaju na lokalnoj infrastrukturi (*on-premises*). Izgled dela korisničkog interfejsa prikazan je na slici 3.9. OIC je deo šire familije platformi i radnih okvira za aplikacije u oblaku koje nudi Oracle [77]. Nacionalni institut za standarde i tehnologiju SAD (National Institute of Standards and Technology[14]) definiše računarstvo u oblaku na sledeći način:

> Računarstvo u oblaku je model koji omogućava sveprisutan, praktičan mrežni pristup na zahtev zajedničkom skupu konfigurabilnih računarskih resursa (npr. mrežama, serverima, skladištima, aplikacijama i servisima) koji se mogu brzo obezbediti i objaviti uz minimalan napor i minimalnu interakciju sa pružaocem usluga.

Brojna istraživanja govore o ekspanziji aplikacija u oblaku, kao i o sve širem prihvatanju ovakvih rešenja od strane kompanija. Skyhigh navodi da se broj aplikacija u oblaku utrostručio od 2013 do 2016. LogicMonitor i Forbes predviđaju da će se do 2020. godine 83% radnih zadataka koje kompanije obavljaju biti skladišteno u oblaku. Sa aspekta integracije, naročito je interesantna procena koju iznosi Skyhigh, a koja govori da prosečna kompanija koristi preko 1400 različitih servisa u oblaku, a da svaki zaposleni koristi prosečno 36 ovakvih servisa. Iz ovoga je jasno da je potreba za integracijom i kod ove vrste aplikacija i dalje izražena, a time i potreba za plaftormama poput OIC.

Ka svakoj aplikaciji koja se integriše, u OIC se kreira *konekcija*. Komunikacija sa aplikacijom se odvija putem *adaptera*. Adapteri implementiraju protokole i formate za razmenu podataka i poruka. Postoji skup predefinisanih adaptera za standardne protokole, a dodatni adapteri se mogu razviti ili dobaviti iz baze dostupnih adaptera. Postoje četiri vrste adaptera: SaaS, tehnološki, društveni i on-premises [4]. SaaS (Software as a Service) adapteri služe za povezivanje sa aplikacijama koje se izvršavaju u oblaku i nude API. Tehnološki adapteri implementiraju protokole za industrijski standardne protokole poput SOAP, REST i (S)FTP. Društveni (social) adapteri omogućavaju komunikaciju sa društvenim mrežama poput Facebook-a, LinkedIn-a ili Twittera i ličnim aplikacijama poput GMail-a, Google kalendara, SurveyMonkey i drugim. On-premises adapteri omogućavaju komunikaciju sa tradicionalnim rešenjima koja se izvršavaju na lokalnoj infrastrukturi kompanije. Integracijom sa njima, kreiraju se rešenja koja predstavljaju *hibridni oblak*. Iz ove grupe dostupni su adapteri za razne standardne formate datoteka, konekcije na sisteme za upravljanje bazama podataka i rasprostranjena rešenja za poslovne aplikacije, poput Oracle E-Business Suite, SAP i slične.

[14] https://www.nist.gov/



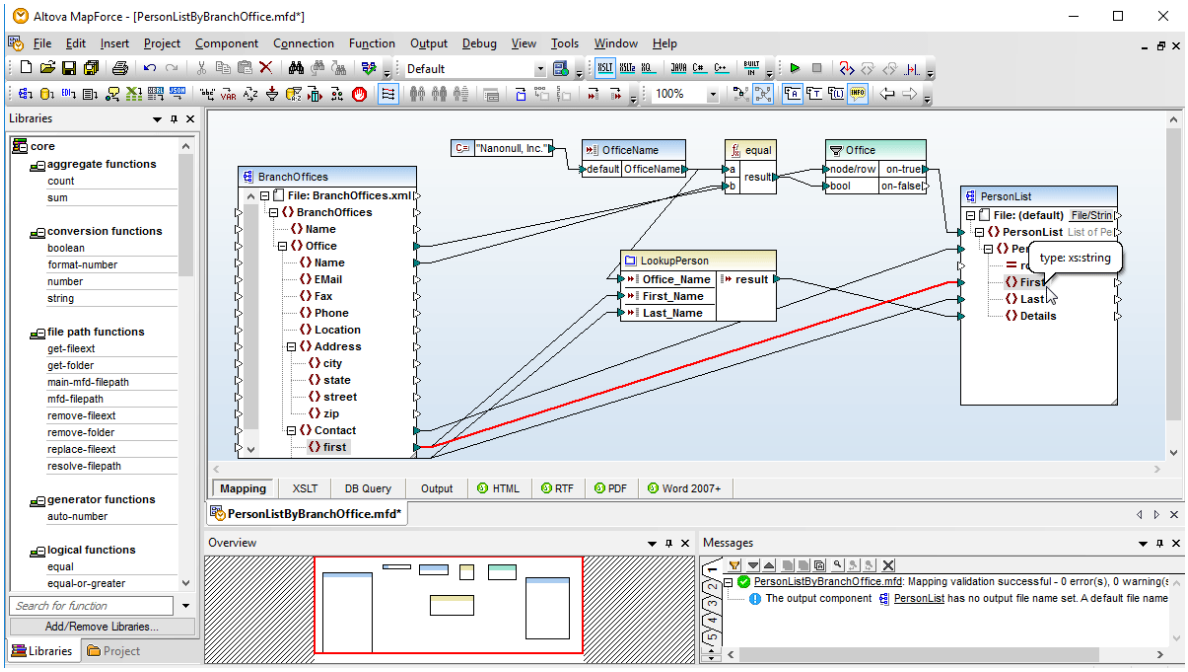
Slika 3.9: Kreiranje orkestracije u Oracle Integration Cloud

3.3.5 Rezime pregleda alata za integraciju

U ovom poglavlju predstavljena su četiri alata koji predstavljaju primere dostupnih alata za integraciju. Prvi, Apache Camel je radni okvir koji implementira integracione šablone i može služiti za programiranje integracionih rešenja na visokom nivou apstrakcije, korišćenjem konstrukata ovog tehničkog domena. Ostali predstavljeni alati - Talend Open Studio, Microsoft BizTalk i Oracle Integration Cloud su primeri komercijalnih integracionih platformi, koje uključuju dodatne alate koji pomažu pri razvijanju i održavanju rešenja, kao i implementaciju okruženja kojima se data rešenja mogu izvršavati i pratiti. Ovi alati uključuju podršku za grafički prikaz i manipulaciju komponentama rešenja, testiranje, validaciju i drugo. Zajedničko svim platformama je i da nude velik broj gotovih konektora, odnosno komponenti koje omogućavaju jednostavnu razmenu podataka ili poruka sa poznatim formatima datoteka, protokolima i aplikacijama. Velik broj ovih konektora podržava i automatsku ekstrakciju strukture šeme datog ulaznog ili izlaznog interfejsa.

Pored četiri prikazane platforme, slične suštinske funkcionalnosti nude i paketi poput: Altova MapForce, AquaLogic, AnalytiX Mapping Manager, BEA WebLogic Workshop, FME Desktop, IBM Rational Data Architect, OPC Router, Open Mapping Software, MetaDapper, MuleSoft Anypoint Studio i Vorto. Njihovi uporedni prikazi analizirani su u [34, 6, 14]. Zajednički korak u razvoju integracionog rešenja kod svih analiziranih alata jeste uspostavljanje mapiranja među elementima ulaznih i izlaznih intefejsa koji učestvuju u integracionom scenariju. Ovaj korak predstavlja manuelni zadatak, čije rešavanje zahteva detaljno poznavanje strukture, ali i načina rada i značenja svakog od ovih elemenata. Još jedan primer tipičnog izgleda korisničkih interfejsa za ručno mapiranje dat je na slici 3.10.

Automatizacija mapiranja dovela bi do značajnog olakšanja celog procesa integracije i



Slika 3.10: Korisnički interfejs za mapiranje u Altova MapForce. Izvor: www.altova.com

doprinela njegovoj eventualnoj potpunoj automatizaciji. Automatizacija detekcije i razrešavanja semantičkih konflikata koji se mogu javiti prilikom mapiranja povećala bi valjanost i pouzdanost integracionog rešenja u celini. Arhitektura radnog okvira, koji predstavlja jedan pristup rešavanju ovog problema, data je u narednom poglavlju.

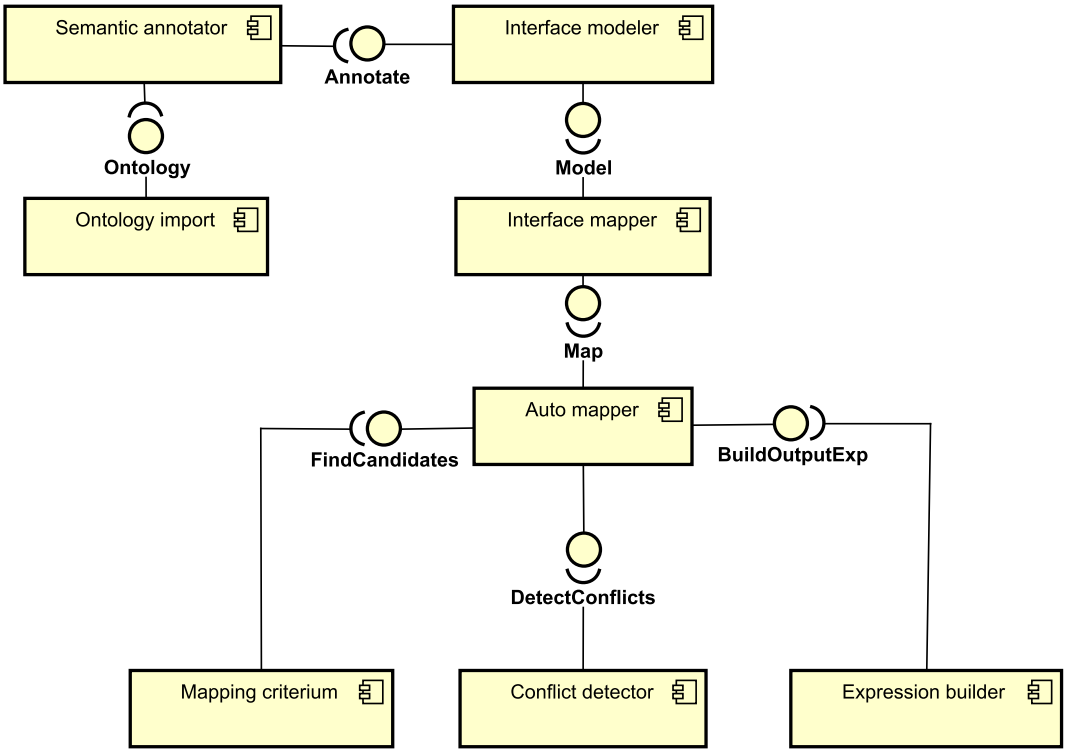
4

Radni okvir za automatizaciju integracije

Cilj radnog okvira je da obezbedi okruženje u kom je moguće ostvariti automatsko ili poluautomatsko mapiranje interfejsa aplikacija koje učestvuju u integracionom scenariju, detektovanje tehničkih i semantičkih konflikata i njihovo rešavanje [139]. Za proces automatizovanog mapiranja potrebni su:

- formalna specifikacija strukture interfejsa aplikacija koje učestvuju u scenariju i koja definiše:
  - nazive elemenata interfejsa,
  - tip podataka svakog elementa interfejsa i
  - ograničenja elementa interfejsa (dužina, preciznost, da li je obavezan, itd.);
- formalna specifikacija semantike svih interfejsa i njihovih elemenata i
- formalna specifikacija integracionog scenarija.

Komponente radnog okvira prikazane su na dijagramu 4.1. Komponenta `Interface modeler` zadužena je za kreiranje i čuvanje modela strukture svih interfejsa svih aplikacija koje se integrišu. Komponenta `Semantic annotator` služi da omogući korisniku da anotira elemente strukturnog modela interfejsa semantikom, odnosno elementima ontologije, čija definicija je uvezena komponentom `Ontology import`. Komponenta `Interface mapper` omogućava da se kreira, prikaže, ručno izmeni i sačuva specifikacija mapiranja među elementima interfejsa. Komponenta `Auto mapper` obavlja proces automatskog mapiranja, detekcije i razrešavanja konflikata. Implementacije komponente `Mapping criterium` definišu uslov koji određuje da li treba uspostaviti mapiranje među nekim parom ulaznih i izlaznih elemenata interfejsa. Neki od ovih kriterijuma prikazani su u sekciji 4.2. Komponenta `Conflict detector` definiše uslove tehničkih i semantičkih konflikata. Primeri ovih uslova prikazani su u sekciji 4.3. Na kraju, komponenta `Expression builder` određuje kako će biti sačinjen izraz koji definiše kako se obavlja mapiranje ukoliko u njemu učestvuje više ulaznih ili više izlaznih interfejsa (da li će njihove vrednosti biti spojene, sabrane ili na neki drugi način obrađene).



Slika 4.1: Dijagram komponenti radnog okvira

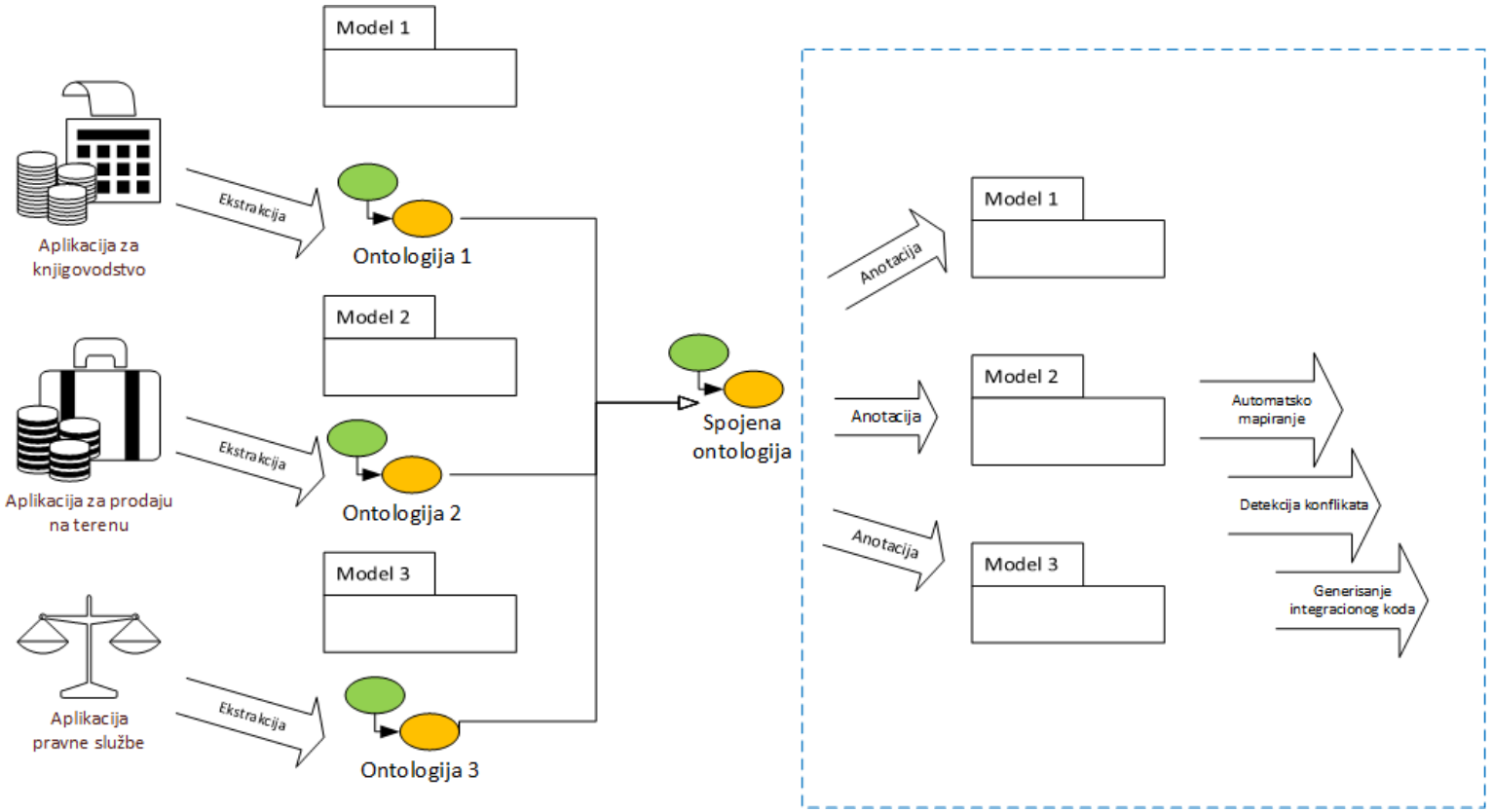
4.1 Proces mapiranja

Preduslov za proces automatskog mapiranja interfejsa je postojanje ontologije koja opisuje elemente tih interfejsa. Korisnik učitava ovu ontologiju i anotira (označava) elemente interfejsa elementima ontologije, pri čemu svaki element interfejsa može biti označen jednim ili više elementa ontologije i na taj način semantički opisan. U mapiranju mogu istovremeno da učestvuju više ulaznih i više izlaznih interfejsa. Šematski prikaz celokupnog procesa dat je na slici 4.2.

Sam proces automatskog mapiranja podeljen je u sledeće faze:

- traženje kandidata za mapiranje,
- detekcija i razrešavanje konflikata,
- formiranje izraza koji definišu mapiranje i
- generisanje integracionog koda.

Tok ovih faza prikazan je na dijagramu 5.1.



Slika 4.2: Šematski prikaz procesa integracije

.

4.2 Kriterijumi mapiranja

Proces traženja kandidata za mapiranje odvija se iterativnim prolaskom kroz sve elemente svih ulaznih interfejsa i sve elemente svih izlaznih interfejsa, tj. njihovim poređenjem svakog sa svakim. Za svaki ovakav par jednog elementa ulaznog i jednog elementa izlaznog interfejsa proverava se da li zadovoljava neki od raspoloživih *kriterijuma mapiranja*. Svaki kriterijum za mapiranje je implementiran kao komponenta prikazanog radnog okvira za automatsko mapiranje. Ovime je omogućeno da se po potrebi uvode novi kriterijumi, proširuju ili isključuju postojeći.

Navodimo neke kriterijume koji su implementirani i testirani u toku razvijanja praktičnog prototipa radnog okvira. U tabeli 4.1 navodimo pregled ovih kriterijuma, a zatim svaki opisujemo ponaosob.

Tabela 4.1: Pregled kriterijuma za mapiranje

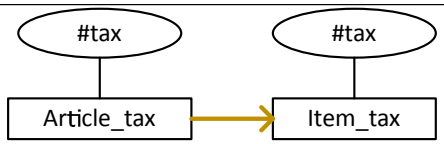
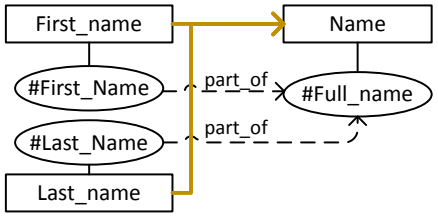
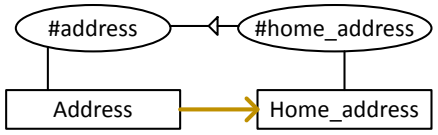
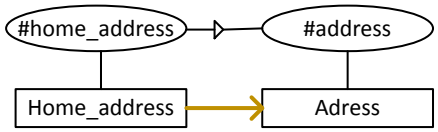
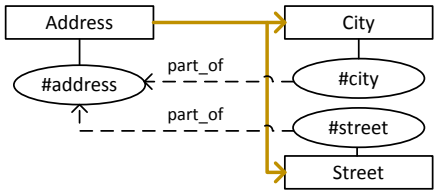
Naziv	**Kratak opis**	**Grafički prikaz**
Jednako ime	Elementi interfejsa imaju isti naziv	
Jednak XPath	Ulazni i izlazni elementi su XML čvorovi koji imaju isti XPath izraz	
Jednaka anotacija	Elementi interfejsa su označeni istim elementom ontologije	
Agregacija	Dva ili više izlaznih elemenata su označeni elementima ontologije koji su definisani kao deo (`part_of`) elementa ontologije kojim je anotiran element izlaznog interfejsa	
Generalizacija	Element ulaznog interfejsa je anotiran elementom ontologije koji je potklasa elementa kojim je anotiran element izlaznog interfejsa	
Specijalizacija	Element izlaznog intefejsa je anotiran elementom ontologije koji je potklasa elementa ontologije kojim je anotiran element ulaznog interfejsa	
Razdvajanje	Dva ili više elementa izlaznog interfejsa su anotirani elem. ontologije koji se sastoji od elemenata ontologije kojim je anotiran jedan elem. ulaznog interfejsa	

Tabela 4.1: Pregled kriterijuma za mapiranje

Zabrana mapiranja	Elem. ulaznog interfejsa i element izlaznog interfejsa su anotirani elementima ontologije koji su u njoj definisani kao različiti	
Legenda		Interface element Ontology element Mapping Ontology predicate Annota on

4.2.1 Jednako ime

Kandidat za mapiranje će biti formiran ukoliko element ulaznog i element izlaznog interfejsa imaju jednako ime. Pri tom se opciono uzimaju u obzir razlike u malim i velikim slovima, načinu predstavljanja razmaka (razmak, donja crta, kamilja notacija) i dijakritičkim znacima.

4.2.2 Jednaka XPath putanja

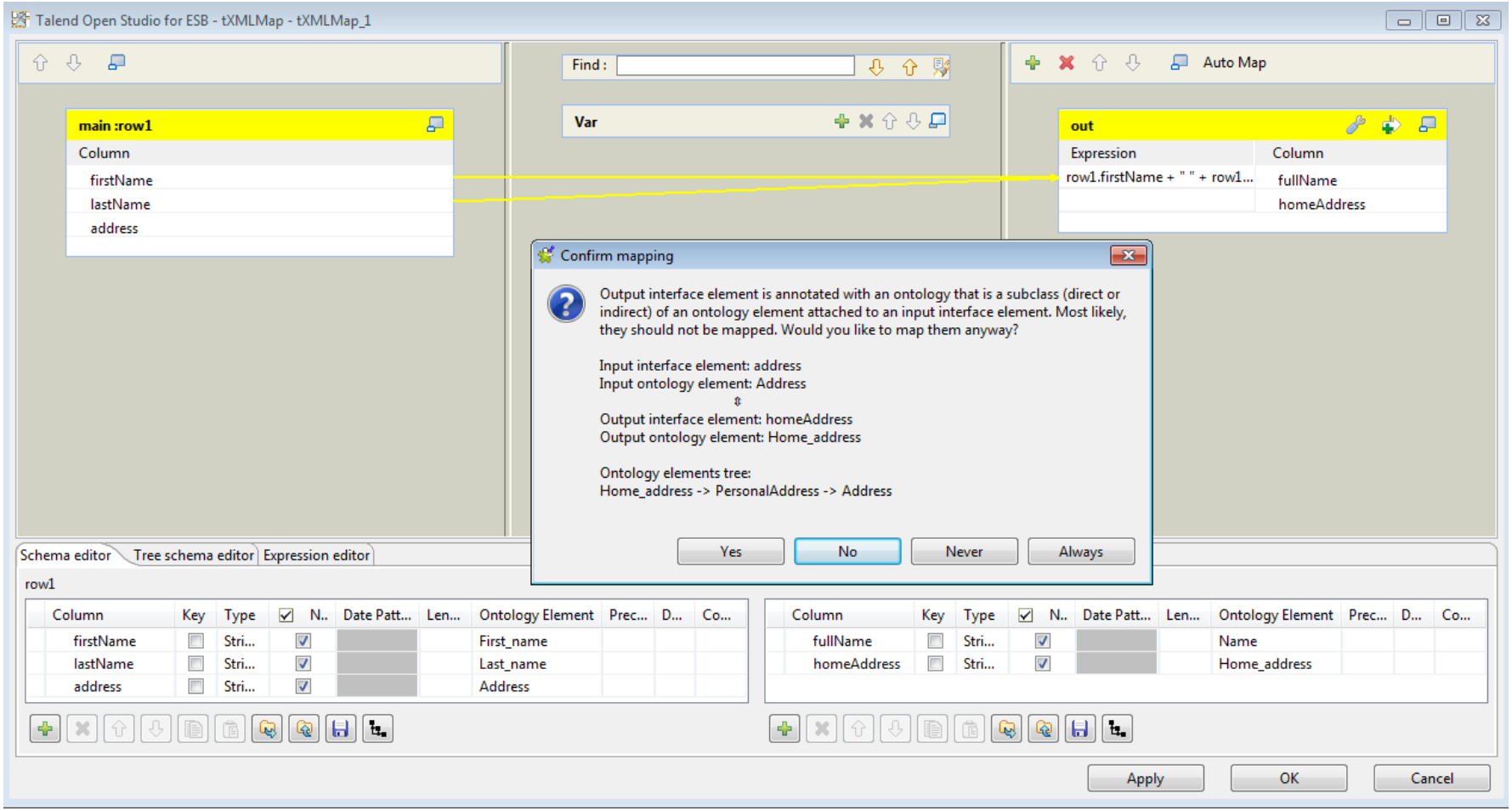
Ukoliko su ulazni i izlazni interfejs deo XML šeme, elementi će postati kandidati za mapiranje ukoliko imaju jednaku XPath putanju.

4.2.3 Specijalizacija

Neka je I1 element ulaznog interfejsa, a O1 element izlaznog interfejsa. U ontologiji su definisane klase A i B, pri čemu je B potklasa A, kao što je prikazano na Listingu 4.1. I1 je anotiran sa A, a O1 je anotiran sa B. Drugim rečima, element izlaznog interfejsa je anotiran elementom ontologije koji prestavlja *specijalizaciju* elementa ontologije kojim je anotiran element ulaznog interfejsa. U ovom slučaju, postoji semantička veza ovih elemenata interfejsa, ali nismo sigurni da li ona treba da rezultuje kandidatom za mapiranje, pa korisniku nudimo mogućnost da, putem dijaloga, odabere da li ovi elementi treba da budu mapirani. Dijalog za odabir prikazan je na slici 4.3.

```
Ontology(
  Class(A partial)
  Class(B partial)
  SubClassOf(B A))
```

Listing 4.1: Definicija specijalizacije u ontologiji



Slika 4.3: Traženje potvrde od strane korisnika

Napominjemo da je ovo samo jedan način na koji se može tumačiti situacija kada smo prepoznali da postoji odnos specijalizacije između nekih elemenata ulaznog i izlaznog interfejsa. Može postojati, na primer, kontekst u kom ovakav odnos elemenata ulaza i izlaza uvek treba tretirati kao mapiranje. U sekciji 6 ćemo pokazati kako se, korišćenjem JSD SAIL, može definisati ponašanje alata u ovakvim i drugim situacijama.

4.2.4 Generalizacija

Neka je I1 element ulaznog interfejsa, O1 element izlaznog interfejsa, a A i B klase ontologije, takve da je A potklasa B, kao što je definisano na listingu 4.2. Element interfejsa I1 je označen sa A, dok je O1 označen sa B. Dakle, element izlaznog interfejsa je označen elementom ontologije koji predstavlja generalizaciju elementa ontologije pridruženog elementu ulaznog interfejsa. Ukoliko je I1 jedini element izlaza anotiran kao potklasa B, mapiranje će biti uspostavljeno automatski između I1 i O1. Ukoliko na ulaznoj strani postoji više elemenata koji su potklasa B, nije moguće automatski odlučiti koji element će biti mapiran, pa se korisniku ostavlja mogućnost da odabere da li će I1 biti mapiran na O1.

```
Ontology(
  Class(A partial)
  Class(B partial)
  SubClassOf(A B))
```

Listing 4.2: Generalizacija u ontologiji

Primer: neka na ulazu postoji polje *kućnaAdresa*, a na izlazu polje *adresa*. Adresa je generalizacija kućne adrese. Ukoliko na ulazu ne postoji ni jedana druga adresa, može se automatski uspostaviti mapiranje *kućneAdrese* na *adresu*. Međutim, ukoliko na ulazu postoji i polje *adresaNaPoslu*, ne možemo znati koja od ovih adresa treba da bude mapirana na jedino polje izlaza koje nosi semantiku adrese, pa odluku ostavljamo korisniku.

4.2.5 Jednako anotiranje

Neka je I1 element ulaznog interfejsa, a O1 element izlaznog interfejsa. Ako su oba ova elementa označena istim elementom ontologije A, biće uspostavljeno mapiranje I1 na O1.

4.2.6 Agregacija

Neka su dati elementi ulaznog interfejsa I1 i I2 i element izlaznog interfejsa O1. Ulazni elementi I1 i I2 su anotirani elementima ontologije A i B, dok je O1 anotiran elementom ontologije C. U ontologiji je definisano da se C sastoji od A i B [1]. Uspostavlja se mapiranje

[1] Jezik OWL sam ne poseduje mehanizam kojim se mogu opisati veze celina-deo. Ukoliko je neophodno uključiti takvu vezu, može se uključiti neka mereološka ontologija koja je definiše. Za potrebe ovog

I1 na O1 i I2 na O1. Dakle, O1 je agregacija I1 i I2, što znači da dve ulazne vrednosti treba da se spoje u jednu izlaznu vrednost. Na koji način će to spajanje biti implementirano (konkateniranjem, sabiranjem, ...) razrešava se kasnije, u fazi konstrukcije izraza (4.4).

```
Ontology(
 Class(C partial)
 Class(A partial
  restriction(part:partOf_directly someValuesFrom(C)))
 Class(B partial
  restriction(part:partOf_directly someValuesFrom(C)))
)
```

Listing 4.3: Definicija agregacije u ontologiji

4.2.7 Tranzitivna agregacija

Neka su I1 i I2 elementi ulaznog interfejsa, a O1 element izlaznog interfejsa. Elementi I1 i I2 su anotirani elementima ontologije A i B, dok je O1 anotiran elementom ontologije C. C je označen kao potklasa D. A i B su u tranzitivnom odnosu agregacije sa C, pa će I1 i I2 biti mapirani na O1.

```
Ontology(
 Class(D partial)
 Class(C partial)
 SubClassOf(D C)

 Class(A partial
  restriction(part:partOf_directly someValuesFrom(D)))
 Class(B partial
  restriction(part:partOf_directly someValuesFrom(D)))
)
```

Listing 4.4: Ontologija koja definiše agregaciju ka nadklasi

4.2.8 Razdvajanje

Neka je I1 element ulaznog interfejsa, a O1 i O2 elementi izlaznog interfejsa. I1 je anotiran sa A, O1 sa B, a O2 sa C. U ontologiji je definisano da su B i C delovi A. Uspostavlja se mapiranje I1 sa O1 i I2 sa O1. U fazi konstruisanja izraza sadržaj I1 će biti podeljen (npr. na razmacima ili zarezima, ukoliko je u pitanju tekstualni sadržaj), a delovi dodeljeni O1 i O2.

Primer: ulazni interfejs sadrži polje *adresa*, dok izlazni interfejs sadrži polja *grad*, *ulica* i *broj*.

primera i testiranja okruženja za automatsko mapiranje korišćena je jedna takva ontologija dostupna na http://www.w3.org/2001/sw/BestPractices/OEP/SimplePartWhole

4.2.9 Zabrana mapiranja

Može se javiti potreba da se određeni elementi interfejsa označe tako da ih ne treba mapirati, iako možda zadovoljavaju neki od aktivnih kriterijuma za mapiranje. Stoga uvodimo kriterijum za zabranu mapiranja. U listingu 4.5 data je ontologija koja definiše klase A i B i njihove individue a i b, za koje je definisano da su različite. Elementi a i b se mogu koristiti za anotiranje elemenata interfejsa koji ne treba da budu mapirani.

```
Ontology(
 Class(A)
 Class(B)
 ClassAssertion(A a)
 ClassAssertion(B b)
 DifferentIndividuals(a b))
```

Listing 4.5: Ontologija koja definiše da su a i b različiti, pa ih ne treba mapirati

4.3 Detekcija i razrešavanje konflikata

Nakon faze traženja kandidata za mapiranje, sledi faza detekcije i razrešavanja konflikata. Kao i u slučaju kriterijuma za mapiranje, i kriterijumi i načini za razrešavanje konflikata predstavljaju proširive komponente, pa se mogu definisati novi, kao i modifikovati ili isključiti postojeći. U nastavku navodimo neke primere konflikata.

4.3.1 Višestruko mapiranje po istom osnovu

Ukoliko je u prethodnoj fazi element izlaznog interfejsa mapiran na više ulaznih interfejsa po istom osnovu, korisniku će biti prezentovan dijalog u kom može da odabere koja od tih mapiranja treba zadržati. Pod istim osnovom smatramo da su mapiranja rezultat istog kriterijuma i da su u okviru tog kriterijuma uslovi isti za sva mapiranja.

Primer: Neka su I1 i I2 elementi ulaznog interfejsa, a O1 element izlaznog interfejsa i neka su svi anotirani istim elementom ontologije A. Na osnovu kriterijuma iste anotacije, biće uspostavljena dva mapiranja, I1 na O1 i I2 na O1. U ovom slučaju nismo sigurni da li I1 i I2 imaju istu semantiku, pa ih treba konkatenirati, ili su greškom dva ulazna elementa anotirana istim elementom ontologije, pa se korisniku ostavlja odluka šta učiniti u ovom slučaju. Ukoliko nije u pitanju greška, korisnik može anotirati elemente kao što je opisano u kriterijumu agregacije, kako ovaj konflikt ne bi bio ponovo prijavljen, a semantika bila jasnije izražena.

4.3.2 Višeznačna specijalizacija ili generalizacija

Kao što je navedeno u kriterijumu specijalizacije (4.2.3), kada je element izlaznog interfejsa anotiran elementom ontologije koji je opštiji od elementa ontologije kojim je anotorian element ulaznog interfejsa, ne može se sa sigurnošću zaključiti da li mapiranje treba da bude uspostavljeno, pa se odluka ostavlja korisniku.

4.3.3 Nekompatibilnost tipova

Prilikom konstrukcije izraza za mapiranje, kao što će biti detaljnije izloženo u sekciji 4.4, uzimaju se u obzir tipovi podataka svakog od elemenata interfejsa koji učestvuju u mapiranju. Ukoliko se ulazni tip ne može direktno dodeliti izlaznom tipu u skladu sa pravilima ciljnog jezika na kom će se izvršavati integraciono rešenje, što je u našem slučaju Java, biće prijavljena greška porukom u konzoli i kao grafički simbol  pored Talend *job* komponente u kojoj je detektovan problem. Izvorna TOS aplikacija poseduje komponentu *tConvertType*, koja se može postaviti pre *tMap* komponente za mapiranje i koja služi za prevođenje podataka ulaznog interfejsa u format koji je kompatibilan sa formatom elementa izlaznog interfejsa. U slučaju da je detektovana nekompatibilnost tipova ulaznog i izlaznog interfejsa između kojih je utvrđeno da treba uspostaviti mapiranje, *tAutoMap* komponenta, koju smo uveli, automatizuje postopak uvođenja i povezivanja *tConvertType* komponente.

4.4 Konstrukcija izraza za mapiranje

Nakon što su pronađeni kandidati za mapiranje i razrešeni eventualni konflikti među njima, neophodno je konstruisati izraze koji će služiti za dodelu vrednosti elemenata ulaznih interfejsa elementima izlaznog interfejsa. Ovi izrazi postaju deo izvornog koda integracionog rešenja, pa u slučaju TOS platforme treba da predstavljaju validne Java izraze. Na primer, ukoliko ulazni interfejs ima polja firstName i lastName, a izlazni interfejs ima polje fullName, u prethodnim fazama je utvrđeno da se firstName i lastName mapiraju na fullName. U izvornom kodu integracionog rešenja želimo liniju poput:

```
output.fullName = input.firstName + " " + input.lastName;
```

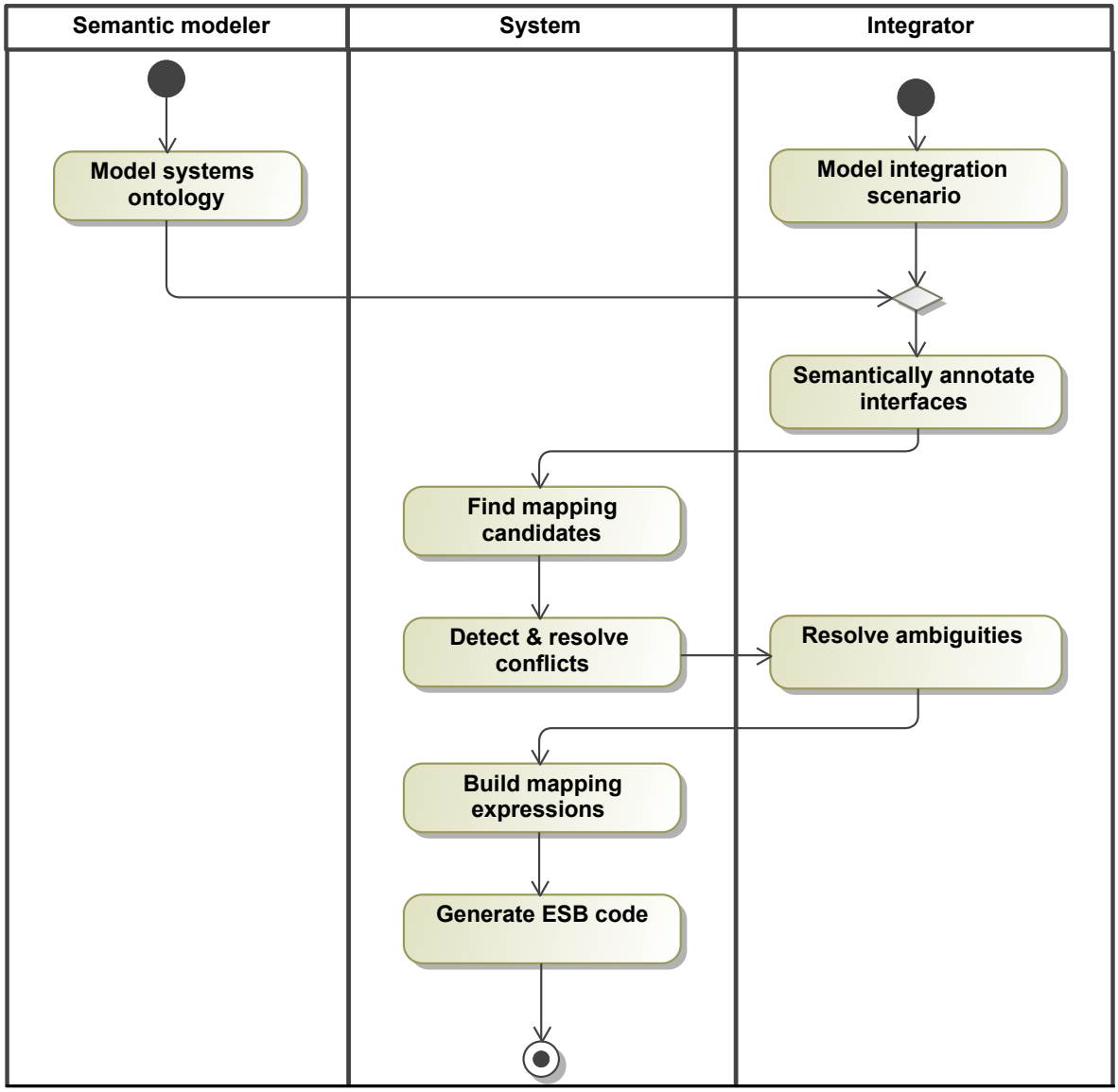
Pri tom, generator koda ugrađen u TOS će za svaki element izlaznog interfejsa generisati po jednu dodelu, pa mi treba da obezbedimo samo desnu stranu dodele u okviru komponente za mapiranje. Za jednostavna mapiranja 1-na-1, konstrukcija ovog izraza je trivijalna i on će sačinjavati samo pun naziv elementa ulaznog interfejsa. Sa druge strane, ukoliko je u pitanju mapiranje više ulaznih na jednu izlaznu vrednost (više-na-1), te ulazne vrednosti treba na neki način agregirati, odnosno spojiti. Ukoliko su sve ulazne vrednosti tekstualne, može se izvršiti konkateniranje, kao u gornjem primeru. Ukoliko su sve ulazne vrednosti numeričke, može se konstruisati izraz koji ih sumira. Ovo su ujedno i podrazumevana ponašanja automatskog

mapera u ovim slučajevima. Ukoliko korisnik želi drugačije ponašanje, može u dijalogu komponente za mapiranje da izmeni svaki od predloženih izraza. Takođe, podrazumevano ponašanje automatskog mapera se može izmeniti proširivanjem komponente za konstrukciju izraza, ili zadavanjem njene specifikacije putem JSD-a opisanog u 6. Kod mapiranja 1-na-više, u *job* se dodaje ugrađena TOS komponenta *cSplitter*, čiji izlaz se dovodi na ulaz *tAutoMap* komponente. U okviru cSplitter komponente je moguće zadati način na koji će ulazna vrednost biti podeljena na delove.

5

Referentna implementacija radnog okvira

Kako bi se omogućilo praktično testiranje do sada navedenih teorijskih razmatranja, razvijena je referentna implementacija radnog okvira datog u poglavlju 4. Oslonac na postojeće aplikacije i alate omogućio je da ova implementacija bude izvedena u relativno ranoj fazi istraživanja, čime je omogućena kontinuirana evaluacija i poređenje novih ideja i pristupa. U ovom poglavlju opisano je kako je izvedena ova praktična implementacija alata za automatizovano mapiranje interfejsa koji učestvuju u integraciji i detekciji semantičkih konflikata među njima. Dijagram aktivnosti celokupnog procesa integracije dat je na slici 5.1. Radni okvir obavlja ulogu naznačenu kao *System* na ovom dijagramu, omogućava učitavanje ontologije koja je rezultat koraka koji obavlja uloga *Semantic modeler* i omogućava anotiranje strukturnog modela elementima semantike, kao i prikaz rezultata i prikupljanje ručnih izmena od strane *Integratora*.



Slika 5.1: Proces integracije. Uloge: *Semantic modeller* - osoba koja kreira semantički opis sistema; *System* - radni okvir za mapiranje; *Integrator* - osoba koja razvija integraciono rešenje

.

5.1 Podloga i podrška

Radni okvir kao ulaz uzima strukturne modele interfejsa i ontologiju koja definiše semantiku tih intefejsa, radi mogućnosti detekcije elemenata koje treba mapirati i mogućnosti detekcije semantičkih konflikata. Kako bi korisnik na efikasan način definisao ove ulazne podatke, neophodno je razviti pogodne alate koji omogućavaju njihovo kreiranje i manipulaciju. Kako već postoji mnoštvo raspoloživih alata koji obavljaju ove funkcionalnosti, zarad bržeg dobijanja prototipa koji možemo koristiti za testiranje teorijskih razmatranja datih u prethodnim poglavljima, koristili smo date alate kao osnovu. Time je potreba za implementacijom svedena samo na one zadatke koji predstavljaju inovaciju i uži predmet ovog istraživanja. Ova sekcija daje osvrt na alate i radne okvire koji su korišćeni kao podloga i podrška implementaciji našeg radnog okvira.

Talend Open Studio

Za aplikaciju kojom će biti kreiran strukturni model interfejsa, korišćena je aplikacija TOS, već prikazana u sekciji 3.3.2. Osnovni razlog za odabir ovog alata je raspoloživost izvornog koda. Zahvaljujući tome, delovi alata su lako mogli biti po potrebi prilagođeni našim potrebama. Pored pomenute funkcionalnosti, TOS daje mogućnost definisanja i orkestracije procesa kojim se implementira rešenje integracionog scenarija, kao i generisanje izvornog koda rešenja, koji je izvršiv na Talend ESB Runtime platformi.

TOS poseduje mogućnost zadavanja specifikacije elemenata za sve ulazne i izlazne interfejse. Ova aplikacija poseduje i komponente za obradu podataka i omogućava međusobno ulančavanje ovih komponenti i ulaznih i izlaznih konektora. Na taj način, aplikacija ispunjava i specificiranje celokupnog rešenja integracionog scenarija. Jedna od komponenti za obradu je i komponenta tMap, koja omogućava mapiranje više ulaznih interfejsa na više izlaznih interfejsa. Mapiranje se obavlja tako što se za svaki element izlaznog interfejsa unosi naziv elementa ulaznog interfejsa. Mapiranje je moguće zadati i prevlačenjem mišem elementa ulaznog interfejsa na element izlaznog interfejsa. Oba načina zadavanja mapiranja rezultuju i vizuelnom reprezentacijom mapiranja, što omogućava lakše snalaženje korisniku. Komponenta poseduje i *Auto Map* opciju, koja je namenjena automatskom mapiranju. Međutim, ova ugrađena funkcionalnost povezuje samo elemente ulaznih i izlaznih interfejsa koji se identično zovu (nije dozvoljena ni razlika u malim i velikim slovima).

Iako je ugrađena funkcionalnost automatskog mapiranja u TOS aplikaciji prilično ograničena, sama aplikacija i njena komponenta tMap predstavljaju dobru osnovu za razvijanje prototipa radnog okvira koji koristi ovde izložene principe. Ono što nedostaje TOS aplikaciji, kako bi se mogli stvoriti preduslovi za automatsko mapiranje zasnovano na semantičkom opisu interfejsa jeste mogućnost zadavanja semantike, pa je i ovo predstavljalo neophodnu modifikaciju.

Eclipse Rich Client Platform

Sam TOS je razvijen kao Eclipse Rich Client Platform aplikacija. S obzirom da se izvšava kao deo TOS-a, i naš radni okvir koristi elemente ove platforme.

Eclipse Rich Client Platform (RCP) [87], između ostalog, uključuje:

- Standard Windget Toolkit (SWT), koji Java GUI (graphical user interface) aplikacijama omogućava korišćenje nativnih korisničkih kontrola [100],
- OSGi [7] radni okvir za razvoj softvera zasnovanog na komponentama,
- Equinox p2 [78], za upravljanje međuzavisnostima, distribuciju i instalaciju komponenti i
- jezgro platforme, koje omogućava definisanje raznih edicija aplikacije (Eclipse koristi termin *branding*), kao i da se krajnja Java aplikacija pokreće iz nativne izvršive datoteke za ciljni operativni sistem umesto `java -jar` komandom. Na primer, za Windows će biti kreirana datoteka u Portable Executable (PE) formatu sa `.exe` ekstenzijom, za Linux će biti kreirana datoteka u Executable and Linkable Format (ELF) formatu.

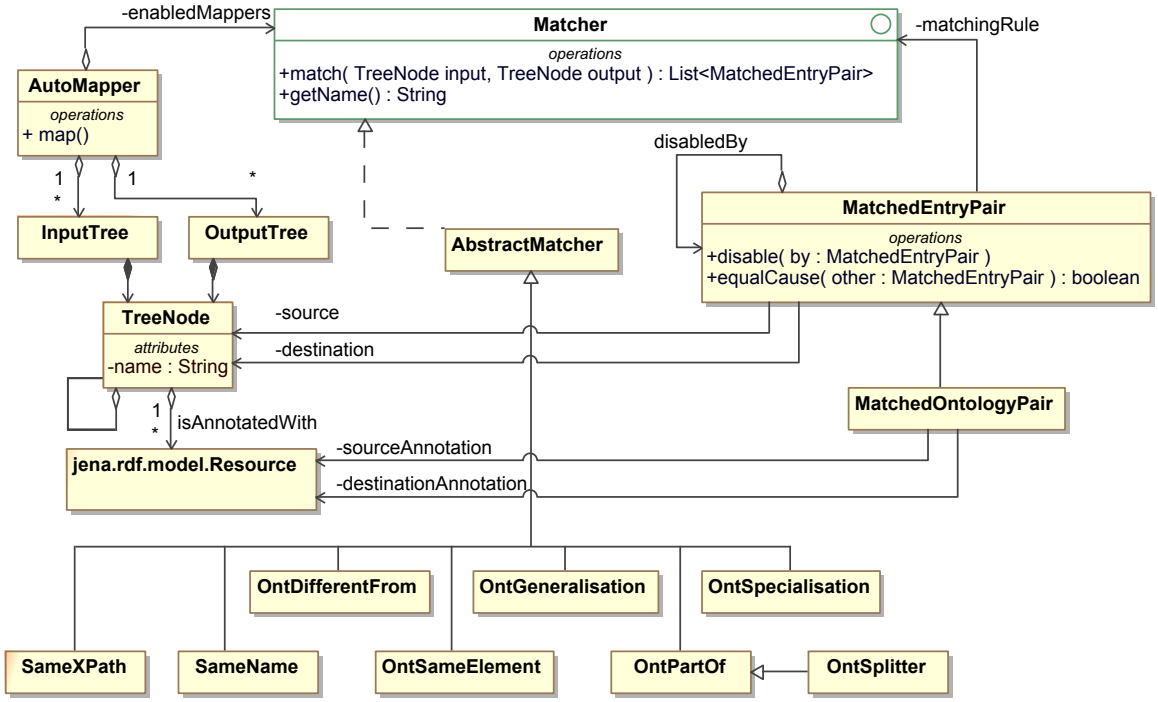
Pored platforme, TOS koristi i Generic Workbench (paket `org.eclipse.ui`) Eclipse IDE (paket `org.eclipse.ui.ide`) komponente, čime se postiže da izgledom i funkcionalnošću podseća na Eclipse integrisano razvojno okruženje.

Apache Jena

Apache Jena je slobodan radni okvir otvorenog koda namenjen izradi aplikacija vezanih za semantički veb i Linked Data [69]. Jena se može koristiti za čitanje i pisanje RDF stabala u raznim formatima. Radni okvir poseduje i OWL API, koji može poslužiti za interakciju sa alatima za rezonovanje. U referentnoj implementaciji našeg radnog okvira, Apache Jena je korišćena za za učitavanje ontologije.

5.2 Arhitektura radnog okvira

Slika 5.2 sadrži dijagram klasa dela radnog okvira zaduženog za pronalaženje kandidata za mapiranje. Prikazana je infrastruktura koja služi za definisanje modela interfejsa aplikacija koje se integrišu, klasa AutoMapper koja implementira mehanizam obilaska, klase koje omogućavaju da se strukturni model anotira elementima ontologije, interfejs koji omogućava da se definišu komponente koje predstavljaju implementaciju kriterijuma za mapiranje, kao i nekoliko njegovih konkretnih implementacija koje predstavljaju kriterijume date u 4.2. U daljem tekstu ovi delovi arhitekture će biti pobliže opisani.



Slika 5.2: Dijagram klasa `Matcher` komponenti

Strukturni model interfejsa

Za omogućavanje definisanja strukture elemenata ulaznih i izlasnih interfejsa koji se integrišu zadužene su klase `InputTree`, `OutputTree` i `TreeNode`. Pomoću ovih klasa definišu se strukture tipa stabla. Klase InputTree i OutputTree predstavljaju korene stabla i može ih biti više, čime je podržano da i po više od jednog ulaznog i izlaznog interfejsa učestvuju u integraciji. Svaka od ovih klasa poseduje kolekciju instanci klase `TreeNode`, kojom se modeluje jedan element interfejsa. Ova klasa sadrži sledeće atribute:

- naziv - koristi se izvorni naziv iz aplikacije koja se integriše,
- tip,
- dužinu i preciznost,
- komentar - sadrži verbalni opis uloge ovog elementa interfejsa i može se koristiti za generisanje dokumentacije,
- rekurzivnu vezu na drugu instancu klase `TreeNode`, čime su podržane kompozitne strukture i
- vezu na instancu klase `Resource` - ova klasa modeluje element ontologije, pa se upravo ovom vezom dodeljuje semantika strukturnom modelu.

Napominjemo da su ove tri klase izvorno deo TOS-a[1]. Definisanje strukture interfejsa jeste jedna od funkcionalnosti koje preuzimamo iz ovog programskog paketa. Naša suštinska modifikacija se sastoji u dodavanju reference na resurse ontologije, čime se omogućava da se model semantički obogati. Interesantan je i pohvalan način na koji su ove, ali i druge modelske klase implementirane u TOS-u. Naime, prvo je sačinjen ECore meta-model, na osnovu kog se generiše Java izvorni kod modelskih klasa. Generisanje skeleta aplikacije svakako nije novina, ali dobra praksa koja je korišćena je da se ovaj ECore model nalazi na istom repozitorijumu i istoj lokaciji kao i izvorni kod. Time se omogućava i ohrabruje da i članovi zajednice, koji unose svoje modifikacije, to čine na samom ECore modelu, čime on ostaje ažuran sa izvornim kodom. Ovakav postupak ispraćen je i prilikom naših izmena.

Klasa Resource

Ova klasa je deo radnog okvira Jena, koji se koristi za čitanje i manipulaciju ontologijama. Ona i njene naslednice mogu predstavljati bilo koji deo ontologije: klasu, individuu, osobinu, itd.

Klasa AutoMapper

Klasa `AutoMapper` implementira osnovni algoritam koji obilazi stabla ulaznih i izlaznih interfejsa i za svaki par njihovih elemenata koristi raspoložive kriterijume za mapiranje, koji su implementacija interfejsa `Matcher`. Pseudo kod algoritma dat je u listingu 5.1. U proceduri Mapiraj, pretpostavlja se da svaki kandidat za mapiranje poseduje proceduru `kandidati(u,i)` koji za određeni par interfejsa daje odgovor da li oni predstavljaju kandidata za mapiranje ili ne. Takođe, pretpostavlja se i postojanje procedure `proveri(u, i, K, U, I)`, koja po osnovu nekog kriterijuma za detekciju konflikata odlučuje da li dati kandidat za mapiranje ostaje na važnosti ili će biti ukinut.

U implementaciji, rezultat rada prve iteracije kroz parove ulaznih i izlaznih elemenata su kandidati za mapiranje, predstavljeni instancama klase `MatchedEntryPair`. Ovi kandidati se dalje prosleđuju raspoloživim detektorima konflikata. Nakon razrešavanja eventualnih konflikata, ostaju samo oni kandidati koji zaista treba da se mapiraju. Na osnovu njih se konstruišu izrazi za mapiranje.

[1] Unutar TOS-a postoje zasebne klase koje modeluju XML interfejse i druge tipove interfejsa. Za ovo postoje implementacioni razlozi, zbog drugačijeg tretmana u pogledu ostalih komponenti TOS-a. Kako u pogledu pronalaženja mapiranja ove dve vrste modela nema suštinskih razlika sa aspekta našeg radnog okvira, ovaj detalj je izostavljen iz opisa arhitekture.

```
procedura Mapiraj
  ulaz: U - stablo ulaznih interfejsa,
      I - stablo izlaznih interfejsa,
      KMap - skup raspoloživih kriterijuma mapiranja,
      KKonf - skup raspoloživih kriterijuma konflikata
  izlaz: M - skup mapiranja
{
  K - skup kandidata za mapiranje
  za svaki u iz U.elementi{
    za svaki i iz I.elementi{
      za svaki km iz KMap{
        ako su km.kandidati(u, i){
          dodaj (u, i) u K
        }
      }
    }
  }
  za svaki (u, i) iz K {
    za svaki kk iz KKonf {
      r = kk.proveri(u, i, K, U, I)
      ako r{
        dodaj (u, i) u M
      }
    }
  }
}
```

Listing 5.1: Algoritam za traženje mapiranja

Interfejs Matcher

Interfejs `Matcher` implementiraju sve komponente koje predstavljaju implementaciju kriterijuma za mapiranje, poput onih datih u 4.2. Osnovna metoda interfejsa je metoda `match`, koja prima dve instance klase `TreeNode`, od kojih jedna predstavlja element ulaznog interfejsa, a druga element izlaznog interfejsa. Metoda vraća kolekciju instanci klase `MatchedEntryPair`, koje predstavljaju kandidate za mapiranje. Dakle, omogućeno je da jedan par ulaznih i izlaznih interfejsa po jednom kriterijumu proizvede više mapiranja. Svaki kriterijum treba da implementira i metodu `getName`, koja vraća deskriptivno ime kriterijuma, nemenjeno za prikazivanje korisniku.

Klasa AbstractMatcher

`AbstractMatcher` je apstraktna klasa koja implementira interfejs `Matcher` i sadrži pomoćne metode koje mogu biti od koristi konkretnim implementacijama. Između ostalog, implementira metodu `getName` tako da vrati naziv klase, čime se dobija njeno podrazumevano ponašanje. Ukoliko ime klase koja implementira kriterijum nije dovoljno deskriptivno, ovo ponašanje može da se redefiniše.

Klasa MatchedEntryPair

Klasa `MatchedEntryPair` predstavlja jednog kandidata za mapiranje. Koristi se kao povratna vrednost metode `match` interfejsa `Matcher`. Instanca ove klase ima dve veze na instance klase `TreeNode`, koje nose informaciju o tome koji ulazni i izlazni element interfejsa učestvuju u ovom mapiranju. Instanca nosi i informaciju o tome na osnovu kog kriterijuma je nastala. U fazi detekcije konflikata kandidati mogu biti u međusobnom konfliktu. Komponenta za razrešavanje konflikata tada može odlučiti da jedan ili oba kandidata poništi pozivom metode `disable`. Metoda `equalCause` određuje da li su dva kandidata za mapiranje nastala iz istog razloga, što se koristi u fazi detekcije konflikata.

Klasa MatchedOntologyPair

Klasa `MatchedOntologyPair` nasleđuje klasu `MatchedEntryPair` i predstavlja kandidata za mapiranje koji je nastao kao rezultat kriterijuma koji u obzir uzima i semantiku. Osnovnim atributima nasleđene klase dodaje i dve reference na klasu `Resource`, koje označavaju na osnovu kog elementa ontologije je detektovano mapiranje.

Klasa AbstractOntologyMatcher

`AbstractOntologyMatcher` je klasa koja pomaže pri implementaciji kriterijuma za mapiranje koji koriste semantičko anotiranje strukturalnog modela elementima ontologije. Svaki

element interfejsa može biti anotiran nijednim, jednim ili sa više elemenata ontologije. Ova klasa implementira metodu `match` tako da za par ulaznih i izlaznih elemenata interfejsa koji se porede, uzima njihove semantičke anotacije i iterira ih svaku sa svakom. Za svaki par anotacija, zatim, poziva apstraktnu metodu `matchOntologyElements`, u kojoj klase koje će naslediti AbstractOntologyMatcher testiraju da li je konkretan kriterijum koji implementiraju zadovoljen. Izvorni kod ove klase prikazan je na listingu 5.2.

```
public abstract class AbstractOntologyMatcher extends
    AbstractMatcher<MatchedOntologyPair> {
  protected Model unionModel =
   OntologyStore.getInstance().getOntModel();

  @Override
  public Collection<MatchedOntologyPair> match(TreeNode input, TreeNode output) {
    List<MatchedOntologyPair> ret = new ArrayList<MatchedOntologyPair>();
    if (unionModel != null) {

      for (String outOntologyElement:output.getOntologyElements()) {

        for (String inOntologyElement : input.getOntologyElements()) {
          Resource inRes =
            unionModel.getResource(inOntologyElement);
          Resource outRes =
            unionModel.getResource(outOntologyElement);

          matchOntologyElements(input, output, ret, inRes, outRes);
        }
      }
    }
    return ret;
  }

  protected abstract void matchOntologyElements(TreeNode input,
      TreeNode output, List<MatchedOntologyPair> ret, Resource inRes,
      Resource outRes);

}
```

Listing 5.2: Izvorni kod klase AbstractOntologyMatcher

Klasa OntologyGeneralisation

Kao primer implementacije jednog konkretnog kriterijuma za mapiranje, navodimo klasu `OntologyGeneralisation`, koja se odnosi na kriterijum opisan u 4.2.4. Klasa nasleđuje klasu `AbstractOntologyMatcher`. U implementaciji metode `matchOntologyElements` (listing 5.3), pristupa se resursima, odnosno elementima ontologije kojima su anotirani ulazni i izlazni element interfejsa koji su prosleđeni metodi. Ukoliko resurs kojim je anotiran ulazni element ima za nadklasu (superklasu) resurs kojim je anotiran izlazni element, tada ovaj par elemenata ispunjava uslove kriterijuma mapiranja koji se ispituje. U tom slučaju, u rezultate se dodaje kandidat za mapiranje koji sadrži ovaj par ulaznih i izlaznih elemenata.

```
@Override
protected void matchOntologyElements(TreeNode input, TreeNode output,
    List<MatchedOntologyPair> ret, Resource inRes, Resource outRes) {
  if (outRes.canAs(OntClass.class)
      && inRes.canAs(OntClass.class)) {

    OntClass clOut = outRes.as(OntClass.class);
    OntClass clIn = inRes.as(OntClass.class);

    if (clIn.hasSuperClass((clOut))) {

      MatchedOntologyPair match = new MatchedOntologyPair(
          input, output, this, clIn.getURI(),clOut.getURI());
      ret.add(match);
      System.out.println("Found ontology Generalisation: "
          + match);
    }
  }
}
```

Listing 5.3: Metoda klase OntologyGeneralisation koja implementira kriterijum 4.2.4

6

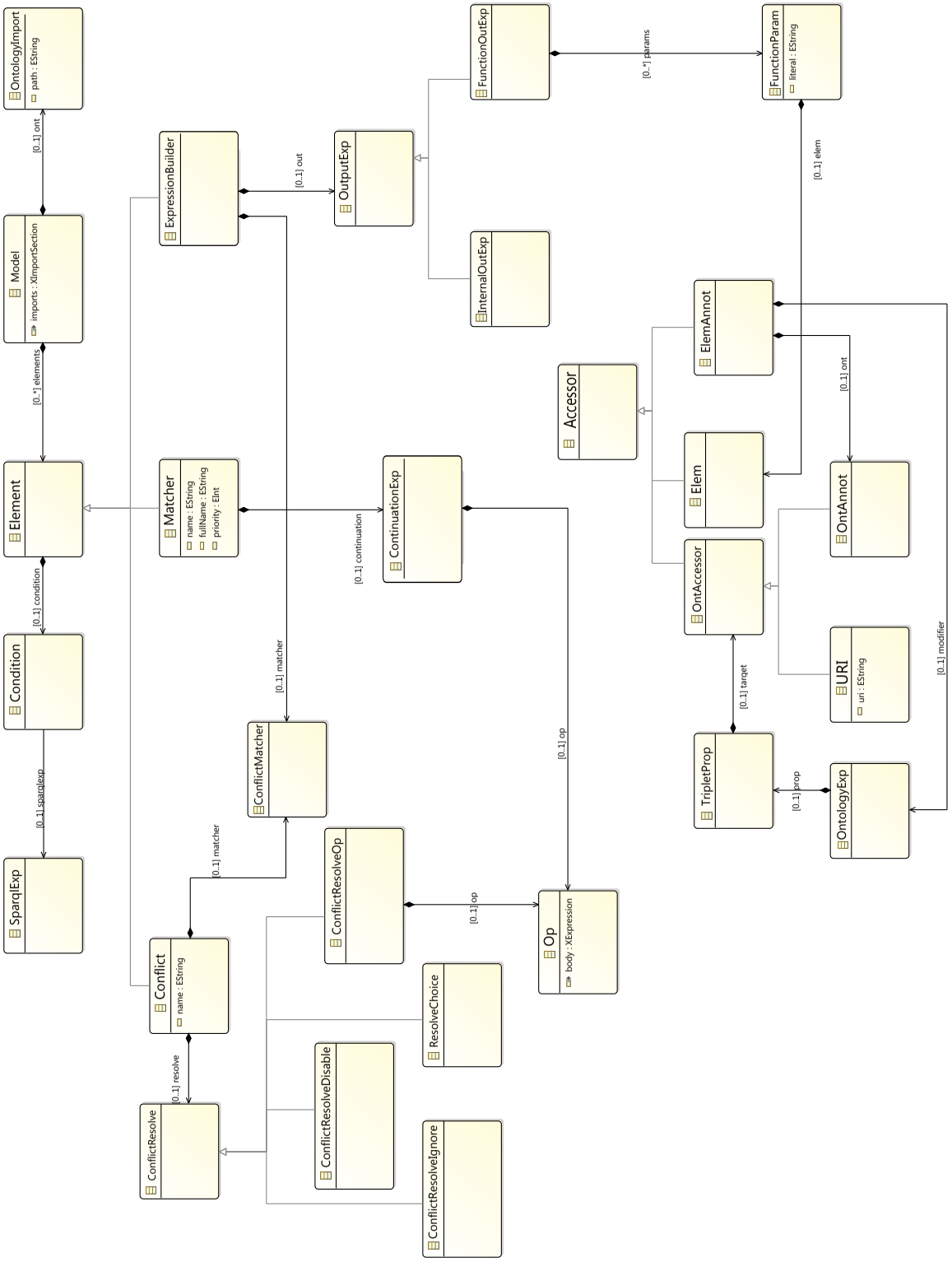
SAIL jezik specifičan za domen

Prikazani radni okvir za automatsko mapiranje, detekciju i rešavanje konflikata moguće je proširivati i prilagođavati potrebama konkretnog integracionog scenarija ukoliko je to potrebno. Detektovanje kandidata za mapiranje, detektovanje i razrešavanje konflikata i izgradnju izraza za mapiranje obavljaju zasebne potkomponente, od kojih se svaka može izmeniti, isključiti ili dodati nova. Međutim, za realizaciju izmena komponenti i razvoja novih, neophodno je poznavanje programskog jezika Java i Eclipse RCP ekosistema u kom su razvijeni aplikacija Talend Open Studio, kao i prototip prikazanog radnog okvira za mapiranje. Radni okvir za automatsko mapiranje mogao bi se implementirati i nezavisno od ove aplikacije (kao što je ranije pomenuto, ona je korišćena samo kako bi se ubrzao razvoj prototipa i izbeglo implementiranje neophodne infrastrukture), kao i na nekom drugom jeziku opšte namene (Python, C#, Pharo, itd). Međutim, i takva implementacija bi i dalje bila zavisna od konkretne izabrane tehnologije, odnosno i dalje bi za proširivanje komponenti za automatsko mapiranje bilo neophodno poznavanje njene arhitekture.

Kako bi se omogućilo opisivanje komponenti radnog okvira za automatsku integraciju na način nezavisan od tehnologije njihove implementacije, dizajniran je i razvijen jezik specifičan za domen nazvan **SAIL**: Semantic-Aided Integration Language [138]. Za jezik je razvijen i propratni skup alata koji olakšava korišćenje jezika, kao i generator izvornog koda komponenti. U nastavku ovog poglavlja biće prikazan dizajn jezika, kao i primer njegovog korišćenja.

6.1 SAIL Meta-model

Meta-model jezika SAIL na apstraktan način opisuje njegovu strukturu, nezavisno od konkretne sintakse i njene implementacije. Korišćen je ECore meta-meta-model. Kompletan meta-model dat je u obliku stabla na slici 6.1.



Slika 6.1: Meta-model jezika SAIL

Meta-klasa *Model*

Model predstavlja korenski kontejner, koji sadrži druge elemente modela, kojima se opisuju komponente radnog okvira. Model sadrži kolekciju uvezenih ontologija, kao instance klasa *OntologyImport*.

Meta-klasa *OntologyImport*

Koristi se za uvoz ontologije koju mogu da koriste komponente za pronalaženje kandidata za mapiranje ili semantičkih konflikata. Poseduje atribut *path*, kojim se navodi putanja do datoteke u kojoj je definisana ontologija.

Meta-klasa *Element*

Element je apstraktna meta-klasa, koju nasleđuju klase koje mogu biti elementi specifikacije komponenti radnog okvira. Element može biti *Matcher*, *Conflict* ili *ExpressionBuilder*.

Meta-klasa *Matcher*

Koristi se za definiciju komponente za specifikaciju kriterijuma za pronalaženje kandidata za mapiranje. Poseduje sledeće atribute:

- name - identifikator,
- fullName - deskriptivni naziv (opciono),
- condition - uslov koji definiše da li će neki par elemenata biti kandidat za mapiranje; instanca je meta-klase *Condition*,
- continuation - veza sa instancom meta-klase *ContinuationExp*, kojom se određuje ponašanje nakon što se za neki par elemenata ustanovi da predstavlja kandidata za mapiranje po prethodno datom uslovu (opciono, podrazumevano: nastavlja se poređenje tog para po drugim uslovima) i
- priority - ceo broj kojim se može odrediti prioritet, odnosno redosled kojim će biti testirani kriterijumi mapiranja, pri čemu veći broj označava veći prioritet (opciono).

Meta-klasa *ContinuationExp*

Služi za definisanje ponašanja nakon što se za neki par elemenata ustanovi da predstavlja kandidat za mapiranje po određenom kriterijumu. Mogući su sledeći ishodi:

- *continue (looking)* - nastavlja se poređenje ovog para elemenata korišćenjem ostalih raspoloživih kriterijuma,
- *next (pair)* - prelazi se na poređenje sledećeg para elemenata i
- do { ... } - zadaje se operacija koja će biti izvršena korišćenjem ciljnog jezika opšte namene (podrazumevano Java), pri čemu su dostupni komponente i servisi radnog okvira.

Meta-klasa *Conflict*

Koristi se za definisanje uslova za detekciju konflikata, kao i načina za njegovo razrešavanje. Poseduje sledeće atribute:

- name - identifikator,
- matcher - pomoću ovog atributa moguće je specificirati da se ovaj kriterijum za detekciju konflikata primenjuje samo na kandidate koji su rezultat određenih kriterijuma za mapiranje (opciono, podrazumevano se konflikt primenjuje na sve kandidate),
- condition - uslov koji definiše kriterijum za detekciju ovog konflikta; instanca je meta-klase *Condition* (opciono),
- causeMultiplicity - koristi se da bi se specificiralo da li se konflikt odnosi na slučajeve kada je par elemenata postao kandidat po više kriterijuma za mapiranje (opciono) i
- resolve - veza sa instancom meta-klase *ConflictResolve*, kojom se specificira način na koji će konflikt biti razrešen.

Za definiciju konflikta mora biti naveden ili atribut *condition* ili atribut *causeMultiplicity*.

Meta-klasa ConflictResolve

Služi za specificiranje načina na koji će konflikt biti razrešen. Može biti jedno od sledećeg:

- ConflictResolveDisable - kandidat za mapiranje se ukida i neće se uzimati dalje u obzir,
- ConflictResolveIgnore - konflikt se ignoriše,
- ResolveChoice - korisniku će biti prepušteno da odluči da li će kandidat za mapiranje biti uzet u obzir i
- ConflictResolveOp - pokreće se procedura zadata iskazima ciljnog jezika opšte namene (podrazumevano Java), pri čemu su dostupni komponente i servisi radnog okvira.

Meta-klasa ExpressionBuilder

Koristi se za definisanje načina na koji će, od nekog kandidata za mapiranje, ili grupe kandidata za mapiranje, biti konstruisan izraz ciljnog jezika ili platforme, koji će postati deo integracionog rešenja. Moguće je navesti i uslov pod kojim će se ova definicija primenjivati. Poseduje sledeće atribute:

- matcher - pomoću ovog atributa moguće je odrediti da se ova specifikacija konstruisanja izraza za mapiranje primenjuje samo na kandidate koji su rezultat određenih kriterijuma za mapiranje (opciono, podrazumevano se konflikt primenjuje na sve kandidate),
- condition - uslov koji definiše kriterijum koji mora biti zadovoljen da bi se ova specifikacija izraza za mapiranje korisila za određeni par kandidata za mapiranje; instanca je meta-klase *Condition* (opciono) i
- out - sama specifikacija konstrukcije izraza, instanca je meta-klase *OutputExp*.

Meta-klasa OutputExp

Koristi se za definisanje izgradnje izraza za mapiranje. Način može biti jedan od sledećih:

- konkatenacija - vrednost izlaznog elementa se dobija konkatenacijom svih ulaznih elemenata,
- razdvajanje (split) - vrednost izlaznih elemenata se dobija podelom jedne ulazne tekstualne vrednosti ili
- specifikacija zadata izrazima ciljne platforme, pri čemu je moguće pristupati elementima parova koji se mapiraju.

Meta-klasa Condition

Koristi se u specifikacijama komponenti kako bi se definisao uslov pod kojim se ta komponenta koristi. Prva vrsta uslova su oni koji se sastoje od jednog iskaza, koji kao rezultat daje Bulovsku vrednost. Uslov je zadovoljen ukoliko je iskaz tačan. Iskaz može biti:

- literal - direktno navedena tekstualna ili brojevna vrednost,
- element modela, ili
- upit nad ontologijom zadat instancom klase *SparqlExp*.

Druga vrsta uslova je poređenje. Poređenje se sastoji iz dve strane koje su iskazi i operatora poređenja. Na raspolaganju su sledeći operatori:

- CO_LT - prva strana je manja od druge,
- CO_GT - prva strana je veća od druge,
- CO_EQ - strane su jedanke,
- CO_NE - strane nisu jednake,
- CO_SUBCLAS_OF - prva strana je potklasa druge,
- CO_SUPERCLASS_OF - prva strana je nadklasa druge,
- CO_PART_OF - prva strana je deo druge i
- CO_DIFFERENT_FROM - prva i druga strana ne predstavljaju istu instancu.

Uslov može biti i konjunkcija ili disjunkcija više uslova.

Meta-klasa *SparqlExp*

Omogućava da se definiše korišćenje SPARQL upita nad uvezenim ontologijama. Poseduje sledeće atribute:

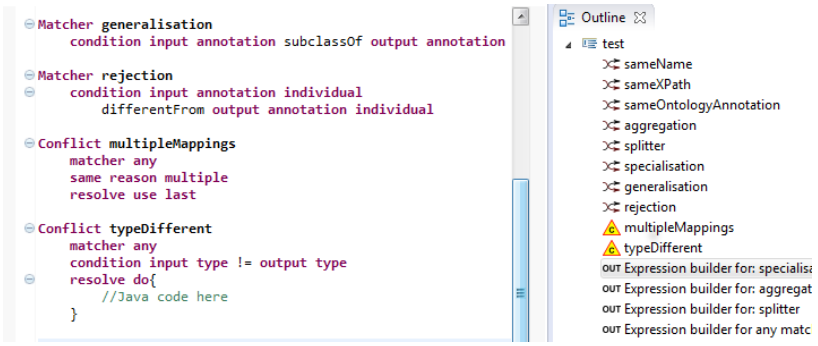
- query - tekst SPARQL upita, pri čemu je moguće koristiti vrednosti iz modela, tako što se identifikator elementa modela navodi u vitičastim zagradama i
- condition - uslov koji treba da bude zadovoljen da bi rezultat ovog upita bio smatran tačnim kada se koristi u definiciji instance meta-klase *Condition*.

6.2 Implementacija jezika

Jezik SAIL je implementiran pomoću radnog okvira Xtext[1], koji je deo Eclipse ekosistema. Za zadatu gramatiku Xtext generiše editor koji podržava bojenje sintakse, dovršavanje izraza, sintaksnu proveru (uz mogućnost implementiranja semantičkih provera), kao i pregled strukture koda pisanog ciljnim jezikom (layout) [36], kao što je prikazano na slici 6.2. Transformaciju izvornog koda razvijenog jezika u iskaze nekog ciljnog jezika moguće je implementirati korišćenjem jezika Xtend[2] i obrađivača šablona Xpand[3]

Gramatika jezika data je u listingu 6.1. Koncepti jezika zasnovani su na komponentama razvojnog okvira za automatsku integraciju i njihovim podešavanjima, kao i izrazima za pristup elementima interfejsa i elementima ontologije.

[1] https://eclipse.org/Xtext
[2] https://www.eclipse.org/xtend/
[3] http://wiki.eclipse.org/Xpand



Slika 6.2: SAIL editor i outline alati

```
grammar rs.ac.uns.ftn.informatika.Sail
  with org.eclipse.xtext.xbase.Xbase

generate sail

Model:
  imports=XImportSection?
  (ont=OntologyImport)?
  elements+=Element*
;

Element:
  Matcher
  | Conflict
  | ExpressionBuilder
;

OntologyImport:
  'Ontology' path=STRING
;

/* Matcher */
Matcher:
  'Matcher' name=ID
  ('full' 'name' fullName=STRING)?
  condition=Condition
  (continuation=ContinuationExp)?
  ('priority' priority=INT)?
;


ContinuationExp:
  'when' 'found'
  (cont=KeepLookingEnum | op=Op)
;

Op:
  'do' body=XBlockExpression
;

```

```
KeepLookingEnum:
  CNT_KEEP_LOOKING='continue' ('looking')?
  | CNT_NEXT_PAIR='next' ('pair')?
;

/* Conflict detection and resolving */
Conflict:
  'Conflict' name=ID
  (matcher=ConflictMatcher)?
  (condition=Condition)?
  (causeMultiplicity=ConflictCauseMultiplicity)?
  resolve=ConflictResolve
;

ConflictMatcher:
  'matcher'
  kind=ConflictMatcherKind
;

ConflictMatcherKind:
  ConflictMatcherSpecific
  | ConflictMatcherAny
;

ConflictMatcherSpecific returns ConflictMatcherKind:
  matchers+=[Matcher]
  (',' matchers+=[Matcher])?
;

ConflictMatcherAny returns ConflictMatcherKind:
  anyMatcher?='any'
;

ConflictCauseMultiplicity:
  'same' 'reason'
  multiplicity=ConflictCauseMultiplicityEnum
;

ConflictCauseMultiplicityEnum:
  ML_MULTIPLE='multiple'
  | ML_SINGLE='single'
;

ConflictResolve:
  'resolve' (
    ConflictResolveDisable
    | ConflictResolveIgnore
    | ConflictResolveOp
    | ResolveChoice
  )
;

ConflictResolveDisable:
  resolve='disable'
;

```

```
98  ConflictResolveIgnore:
      resolve='ignore'
100 ;

102 ConflictResolveOp:
      op=Op
104 ;

106 ResolveChoice:
      'use' use=UseTypeEnum
108 ;

110 UseTypeEnum:
      UT_FIRST='first'
112     | UT_LAST='last'
        | UT_USER='user' 'choice'
114     | UT_PRIORITY='priority';

116 Condition:
      'condition' expr=CompareExp
118 ;

120 /* Expression builder */
    ExpressionBuilder:
122   'OutExpression'
      matcher=ConflictMatcher
124   (condition=Condition)?
      'out' out=OutputExp
126 ;

128 OutputExp:
      InternalOutExp
130   | FunctionOutExp
    ;
132
    InternalOutExp:
134   InternalExpTypeEnum
      elem=Accessor 'with'
136   withStr=STRING
      ('null' ('substitute')? nullSub=STRING)?
138   ('order' elems+=OrderElem
        (',' elems+=OrderElem)*)?;
140
    OrderElem:
142   name=STRING (ignore?='ignore')?;

144 InternalExpTypeEnum:
      IE_CONCATENATE='concatenate'
146     | IE_SPLIT='split'
    ;
148
    FunctionOutExp:
150   'function'
      fName=FunctionNameEnum
152   '('
```

```
  => (params+=FunctionParam
    (',' params+=FunctionParam)*)?
  ')'
;

FunctionParam:
  elem=Elem
    | literal= (STRING | Number)
;

FunctionNameEnum:
  FN_SUM='SUM'
  | FN_AVG='AVG'
  // etc...
;

/**** Comparison */
CompareOr:
  left=CompareAnd
  ('or' right+=CompareAnd)*
;

CompareAnd:
  left=CompareExp
  ('and' right+=CompareExp)*
;

CompareExp:
  left=CompareSide
  (op=CompareOp right=CompareSide)?
;

CompareSide:
  {CompareSide}
  ((quantifier=Quantifier)? accessor=Accessor)
  | direct=(STRING | Number )
  | sparql=SparqlExp;

CompareOp:
  CO_LT='<'
  | CO_GT='>'
  | CO_EQ='='
  | CO_NE='!='
  | CO_SUBCLAS_OF='subclassOf'
  | CO_SUPERCLASS_OF='superclassOf'
  | CO_PART_OF='partOf'
  | CO_DIFFERENT_FROM='differentFrom'
;

Quantifier:
  EVERY='every' | EXISTS='exists'
;

/*  Interface accessors */
Accessor:
  Elem
```

```
    | ElemAnnot
    | OntAccessor
;

Elem:
    {Elem}
    (side=InputOutputEnum)?
    (pair?='pair')?
    (part=ElemPartEnum)?
;

ElemPartEnum:
    EP_NAME='name'
    | EP_TYPE='type'
    | EP_LEN='len'
    | EP_XPATH='xPath'
    | EP_COUNT='count';

ElemAnnot:
    side=InputOutputEnum
    ont=OntAnnot
    (modifier=OntologyExp)?;

InputOutputEnum:
    IO_INPUT='input'
    | IO_OUTPUT='output';

/*  Ontology related */
OntAccessor:
    OntAnnot | \acrshort{uri};

OntAnnot:
    {OntAnnot} 'annotation';

\acrshort{uri}:
    'ontURI' '(' uri=STRING ')';

OntologyExp:
    kind=OntElemKindEnum (prop=TripletProp)?;

OntElemKindEnum:
    OE_CLASS='class' | OE_INDIVIDUAL='individual';

TripletProp:
    'has' part=TripletPartEnum target=OntAccessor;

TripletPartEnum:
    TP_SUBJECT='subject'
    | TP_PREDICATE='predicate'
    | TP_OBJECT='object';

/* \acrshort{sparql} */
SparqlExp:
    SparqlQuery 'returns' condition=SparqlReturn;

SparqlReturn:
```

```
  SparqlReturnAny
266 | SparqlReturnRegEx
  | SparqlBooleanReturn;
268
SparqlBooleanReturn:
270 boolean=('true' | 'false');

272 SparqlReturnRegEx:
  regEx=STRING;
274
SparqlReturnAny:
276 {SparqlReturnAny} 'any';

278 SparqlQuery:
  '\acrshort{sparql}' query=STRING;
```

Listing 6.1: Gramatika jezika SAIL

Izrazi za pristup elementima okruženja

Svaki izraz se podrazumevano odnosi na par ulaznih i izlaznih interfejsa koji se trenutno obrađuju. Ulazni interfejs se identifikuje ključnom reči *input*, a izlazni ključnom reči *output*. Ulazni i izlazni interfejsi imaju sledeća svojstva, kojima se može pristupiti navođenjem iza ključne reči koja se odnosi na interfejs, uz razdvajanje tačkom:

- `name` - naziv elementa interfejsa,
- `type` - tip podataka elementa interfejsa,
- `length` - dozvoljena dužina sadržaja,
- `count` - broj elemenata interfejsa i
- `annotation` - semantičke anotacije pridružene elementu.

Uslovni izrazi

Uslovni izraz počinje ključnom reči `condition`, iza koje se navodi izraz koji opisuje kada je uslov ispunjen, a sastoji se od kombinacije izraza za pristup elementima, literala, operatora za poređenje (`=,<,>,<=,=>,<>`) i konjunkcije, disjunkcije ili negacije (`and, or, not`).

Kriterijum mapiranja

Definicija kriterijuma mapiranja počinje ključnom reči `Matcher`, nakon koje se navodi identifikator kriterijuma. Moguće je, zatim, navesti i puno ime kriterijuma iza ključnih reči `full name`, unutar navodnika. Sledi uslovni izraz koji treba da bude zadovoljen da bi par elemenata ulaznog i izlaznog bili smatrani kandidatom za mapiranje. Ukoliko neki par

zadovolji uslov za mapiranje po ovom kriterijumu, možemo hteti da se nastavi provera po ostalim kriterijumima za isti par ili da se pređe na sledeći par - u zavisnosti od semantike kriterijuma. Ovo se navodi pomoću konstrukcije `when found continue` [`looking`] ili `when found next` [`pair`].

Detektor konflikata

Definicija komponente za detekciju konflikata počinje ključnom reči `Conflict`, nakon koje se navodi identifikator detektora. Možemo odabrati da detektor gleda kandidate za mapiranje koje su proizveli samo određeni kriterijumi za mapiranje. Ovo se postiže navođenjem ključne reči `matcher`, iza koje sledi spisak identifikatora kriterijuma mapiranja. Ukoliko detektor važi za bilo koji kriterijum mapiranja, navodimo `matcher any`. Zatim se navodi uslovni izraz konflikta. Način razrešavanja konflikta navodi se nakon ključne reči `resolve`. Načini razrešavanja su sledeći:

- `ignore` - konflikt se ignoriše,
- `none` - konflikt se prijavljuje, ali se ne razrešava automatski,
- `disable` ili `drop` - kandidat za mapiranje se odbacuje,
- `use first` - u slučaju više kandidata za mapiranje koristi se prvi,
- `use last` - u slučaju više kandidata za mapiranje koristi se poslednji,
- `user choice` - u slučaju više kandidata za mapiranje korisniku se prikazuje dijalog za izbor,
- `do` - navodi se blok Java koda koji će biti izvršen, ili
- `call` - poziva se eksterna komponenta.

Konstruktori izraza za mapiranje (expression builder)

Konstruktor izraza za mapiranje počinje ključnom reči `OutExpression`. Iza ključne reči `matcher` opciono se mogu navesti identifikatore kriterijuma za mapiranje za koje se primenjuje ovaj konstruktor. Zatim se navodi uslovni izraz koji odlučuje da li će ovaj konstruktor biti korišćen. Ukoliko više konstruktora izraza po ovim kriterijumima mogu biti korišćeni za određeno mapiranje, biće korišćen konstruktor koji je poslednji naveden. Način na koji će biti izgrađen izraz navodi se iza ključne reči `out`. Na raspolaganju su sledeći izrazi:

- `concatenate [with "`*`<delimiter>`*`"]` - vrednosti elemenata ulaznih interfejsa se konkateniraju, uz opciono navođenje stringa koji će biti korišćen za razdvajanje,
- `split "`*`<delimiter>`*`"` - ulazni string se razdvaja, a delovi dodeljuju elementima izlaznog interfejsa,
- `function` - korišćenje ugrađenih funkcija poput `SUM()` i `AVG()`,
- `ignore` - preskače određeni element interfejsa i

- `null substitute "<string>"` - omogućava da se nedostajuće vrednosti zamene određenim stringom.

Kao primer SAIL specifikacije navodimo listing 6.2 u kom su definisane neke od komponenti navedenih u 4.2. Ovakva definicija korišćena je za testiranje implementacije jezika, poređenjem ponašanja izvršivih komponenti, dobijenih njenom transformacijom, sa ranije ručno razvijenim komponentama.

```
Matcher sameName
  full name "Same name"
  condition input name = output name
  and input type = output type
Matcher sameOntologyAnnotation
  condition input annotation = output annotation
Matcher aggregation
  condition output annotation partOf input annotation
Matcher splitter
  condition input annotation partOf output annotation
Matcher specialisation
  condition input annotation
    superclassOf output annotation
Matcher generalisation
  condition input annotation subclassOf output annotation
Matcher rejection
  condition input annotation individual
    differentFrom output annotation individual
Conflict multipleMappings
  matcher any
  same reason multiple
  resolve use last
Conflict typeDifferent
  matcher any
  condition input type != output type
  resolve do{
    System.out.println("Java code here"); }
OutExpression
  matcher specialisation, rejection
  condition pair count > 1
    and every pair type = "String"
  out concatenate input name with " "
OutExpression
  matcher any
  condition pair count > 1
  and every pair type = "Number"
  out function SUM(output name)
OutExpression
  matcher aggregation
  condition output name = "address"
  out concatenate input name with "\n"
  order "street", "number", "zip" ignore, "city"
OutExpression
  matcher splitter
  out split with "," null substitute ""
```

Listing 6.2: Primer korišćenja SAIL jezika

7

Evaluacija

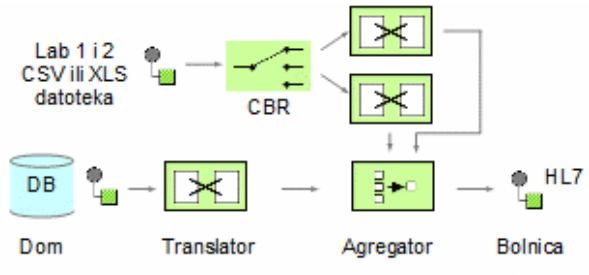
Validnost prikazanog radnog okvira i mogućnost primene razvijene prototipske implementacije testirani su na dva realna integraciona scenarija, kao i jednim eksperimentom. Za oba scenarija je prethodno postojala ručna implementacija integracionog rešenja. U oba slučaja uspešno je postupkom prikazanim u poglavlju 4 i alatom prikazanom u 5 dobijeno rešenje koje funkcionalno ne odstupa od ranije ručno razvijenog. Sprovedeni eksperiment kroz nekoliko konstruisanih zadataka poredi vreme neophodno da se interfejsi mapiraju ručno i upotrebom alata za automatsko mapiranje.

7.1 Scenario 1: dom za negu starih

Ova sekcija prikazuje prvi od dva razmatrana integraciona scenarija. Prvo je izložen sam scenario, a zatim su predstavljeni protokoli i formati podataka koji koriste aplikacije koje se integrišu. Slede opisi ručne implementacije i implementacije radnim okvirom predstavljenim u prethodna dva poglavlja. Na kraju, data je diskusija o izazovima uočeni u toku razvijanja rešenja ovog scenarija.

7.1.1 Scenario

Dom za negu starih lica poseduje informacioni sistem zasnovan na komercijalno raspoloživom softverskom paketu. Klijenti ove ustanove upućuju se periodično u lokalnu bolnicu na redovan godišnji pregled. Za potrebe pregleda urađene su analize krvi u dve različite laboratorije. Jedna je rezultate poslala kao CSV, a druga kao XLS datoteke. Opšte podatke o svakoj osobi, koji se nalaze u bazi podataka aplikacije za evidenciju o korisnicima doma, potrebno je spojiti sa rezultatima analiza i poslati bolnici u obliku HL7 poruka, poput one koja je prikazana na listingu 7.1. Podaci se šalju bolnici samo ukoliko rezultati analiza sadrže vrednosti koje odstupaju od referentnih, pa je potrebna analiza specijaliste. Šema scenarija prikazana je na slici 7.1.



Slika 7.1: Integracioni scenario razmene rezultata krvi

.

Kako je broj korisnika velik, neophodno je automatizovati proces obrade rezultata i slanja na dodatnu analizu.

```
MSH|^~\&|HCM|SAP|TestSis|Nexus|
20130218080959||MFN^M05|01166422|P|2.3||
NP97I0|
MFI|LOC|||||NE|
MFE|||||
LOC||||Regal-Eagle-Hospital^^
0000010133|Thesis-Str.33-52^^
Bluffoonia^^14193^BF|030/8955-0~030/8955-5055|
MFE|||||


MSH|^~\&|TestSis|Nexus|HCM|SAP|
20130826233743.528+0200||ACK^M05|501|P|
2.3|
MSA|AA|01166422|
```

Listing 7.1: Primer razmene poruka u HL7 formatu

7.1.2 Formati i protokoli od interesa u scenariju

U prvom scenariju javljaju se poruke u HL7, CSV i XLS formatima.

HL7 Standard HL7 primenjuje se u domenu informacionih sistema zdravstvene zaštite. Standardom rukovodi Health Level Seven International, neprofitna organizacija, akreditovana od strane ANSI (American National Standards Institute), koja se bavi donošenjem radnih okvira i pridruženih standarda za razmenu, integraciju, deljenje i dobavljanje podataka iz oblasti zdravstva u elektronskoj formi.[1] Broj sedam u nazivu standarda i organizacije odnosi se na sedmi (aplikativni) nivo ISO/OSI (International Organization for Standardization Open Systems Interconnection) modela mrežne komunikacije. Komunikacija se odvija

[1] http://www.hl7.org/about

razmenom poruka. Poruka može biti u XML formatu ili u tzv. *pipe-and-hat* formatu, svojstvenom za HL7. Struktura poruka je hijerarhijska i poruke se sastoje od segmenata, polja, komponenti i podkomponenti. U pipe-and-hat formatu, određeni karakteri se koriste za razdvajanje delova poruke. Svi karakteri za razdvajanje, osim karaktera za razdvajanje segmenta, koji je uvek heksadecimalna vrednost 0x0D, mogu se konfigurisati u sklopu MSH segmenta (Message Header, zaglavlje poruke). Najčešće se koriste sledeći karakteri[2]:

0x0D	razdvajanje segmenta
|	razdvajanje polja, *pipe* (cev)
^	razdvajanje komponenti, *hat* (šešir)
&	razdvajanje podkomponenti
~	razdvajanje ponovljenog polja

CSV Comma-Separated Values je format u kom se polja odvajaju određenim karakterom, najčešće zarezom. Formalni opis ovog formata dat je u dokumentu RFC4180 [122]. Kako je naznačeno i u samom dokumentu, on ne predstavlja propisani standard za ovaj format, već ga samo dokumentuje i pruža informacije o njegovom korišćenju Internet zajednici. Jednostavnost formata i činjenica da koristi tekstualni zapis čine ga lakim za obradu i lako prenosivim na razne sisteme i platforme. Istovremeno, s obzirom da ne postoji formalan standard koji ga definiše, podaci koji su zapisani ovim formatom mogu biti protumačeni na različite načine. Navodimo neke moguće uzroke ovakvih višeznačnosti:

- jedan zapis nalazi se u jednoj liniji, pri čemu način na koji se odvajaju linije zavisi od operativnog sistema (CR, CRLF, LFCR, LF),
- za odvajanje polja unutar jednog zapisa u upotrebi su razni karakteri: razmak, zarez, tačka-zarez, tab; unutar datoteke ili poruke ne postoji zaglavlje koje definiše koji karakter se koristi za odvajanje (ova informacija se može prenositi eksterno, npr. u okviru HTTP zaglavlja, kao deo MIME (Multipurpose Internet Mail Extensions)pojo oznake, ukoliko se CSV sadržaj prenosi ovim protokolom)
- prva linija u datoteci može, ali ne mora sadržati nazive polja,
- polja mogu, a ne moraju biti uokvirena navodnicima i
- ne postoji formalno definisan način za predstavljanje NULL vrednosti, npr. drugo polje u zapisu

  aaa„bbb

  može se tumačiti i kao prazan string i kao NULL.

XLS je format datoteke koji koristi aplikacija Microsoft Excel kao podrazumevani format do verzije 2007. Zapis u datoteci je binaran. U pitanju je vlasnički format, pa dostupnost i legalnost alata za programsko čitanje i upisivanje u ovakve datoteke zavisi od proizvođača. Postiji detaljna opisna specifikacija formata[3].

[2] http://healthstandards.com/blog/2007/09/24/hl7-separator-characters/
[3] https://docs.microsoft.com/en-us/openspecs/office_file_formats/ms-xls

7.1.3 Ručna implementacija

Integraciono rešenje je implementirano radnim okvirom Apache Camel. Za specifikaciju ruta je korišćen Camel-ov Java JSD. Podaci dobijeni od laboratorija stižu kao datoteke CSV ili XLS formata i snimaju se u direktorijum nazvan RESULTS_INBOX. Čim neka datoteka pristigne u navedeni direktorijum, čita se i pretvara u Camel poruku. *File* komponenta potom premešta datoteku u drugi folder, čime je označava kao obrađenu. Zatim se poruka prosleđuje *Content-based Router*-u, koji na osnovu tipa datoteka, utvrđenog iz zaglavlja, poruku dalje šalje na rutu za obradu datog tipa. Deo Camel rute za čitanje datoteka i rutiranje na osnovu tipa prikazan je na listingu 7.2.

```
from("file:" + RESULTS_INBOX)
.routeId("file")
  .choice()
  .when(header("CamelFileName")
  .endsWith(".xls"))
    .to("direct:XLSTestResults")
  .when(header("CamelFileName")
  .endsWith(".csv"))
    .to("direct:CSVTestResults");
```

Listing 7.2: Apache Camel ruta za čitanje datoteka i slanje na dalju obradu na osnovu tipa datoteke

Za čitanje iz baze podataka korišćena je SQL Camel komponenta, kao što je prikazano na listingu 7.3. Prilikom zadavanja upita za SQL komponentu, moguće je koristiti imenovane parametre. Naziv imenovanog parametra navodi se iza kombinacije znakova :#, a na to mesto biće umetnuta vrednost pronađena u polju zaglavlja poruke sa istim nazivom.

Potrebno je prvo na osnovu imena, prezimena i datuma rođenja (što su podaci raspoloživi u laboratorijskim rezultatima) pronaći unos vezan za ovog korisnika u tabeli `PFLEGEBED` (nem. postelja za brigu). Vrednost primarnog ključa `Z_PF` za pronađeni unos postavlja se u zaglavlje poruke, što se postiže upotrebom procesora `MapBodyToHeader`. Svrha ovog procesora je da jedan red tabele dobijen od SQL komponente mapira u zaglavlje tako što će kreirati parametar zaglavlja sa nazivom jednakim nazivu kolone i vrednošću jednakoj vrednosti u bazi. Vrednost ključa `Z_PF` prosleđuje se uskladištenoj proceduri koja prikuplja podatke o pacijentu iz sedam tabela i smešta ih u jedan red privremene tabele `Export`. Ova uskladištena procedura je deo aplikacije za vođenje evidencije o klijentima.

```
from("direct:loadDb").routeId("Database")
.process(new PatientHeaders())
.to("sql: " + makePatientQuery())
.process(new FirstItemToBody())
.process(new MapBodyToHeader())
.to("sql:exec [dbo].[ExportiereDebitor_Event] :#Z_PF "
 + "?dataSource=db")
.setBody(simple(""))
.to("sql:select top(1) * "
  + "from [DemoDBVivAmbulant].[dbo].[Export]"
  + "where Pflegebed_Z_PF = :#Z_PF ?dataSource=db")
.process(sqlToBean);
```

Listing 7.3: Pronalazak vrednosti primarnog ključa u tabeli baze podataka na osnovu identifikacionih podataka pronađenih u rezultatima analize krvi

Podaci iz laboratorije i opšti podaci o klijentu spajaju se primenom *Aggregator*-a, kao što je prikazano na listingu 7.4. Zatim se proverava da li u rezultatu postoje vrednosti koje su van referentnih opsega. Ukoliko postoje povišene ili povećane vrednosti neke analize, poruka se prosleđuje komponenti koja dobijene POJO klase pretvara u HL7 objektni model, a zatim, korišćenjem HAPI biblioteke, konstruiše string u HL7 pipe-and-hat formatu i šalje ga bolnici na analizu.

Pored definisanja prikazanih ruta, koje rukovode procesom koji se odvija prilikom prispeća svake analize, bilo je neophodno razviti i nekoliko Camel procesora i pratećih komponenti. Ovi procesori, poput `PatientBeanToHL7` zaduženi su za mapiranje i konverziju podataka iz jednog oblika u drugi.

```

from("direct:aggregate").routeId("aggregator")
.to("log:test?level=DEBUG&showHeaders=true")
.aggregate(header("idString"),
new EnrichPatientResults()).completionSize(2)
.process(new ResultsAlertToHeader())
.choice()
.when(header(ResultsAlertToHeader.HAS_ALERTS)
   .isEqualTo(true))
   .to(DIRECT_HOSPITAL);

from(DIRECT_HOSPITAL)
.process(new PatientBeanToHL7())
.to("mina2:tcp://localhost:333?sync=true
  &codec=#hl7codec");
```

Listing 7.4: Agregacija i prosleđivanje ukoliko rezultati odstupaju od referentnih vrednosti

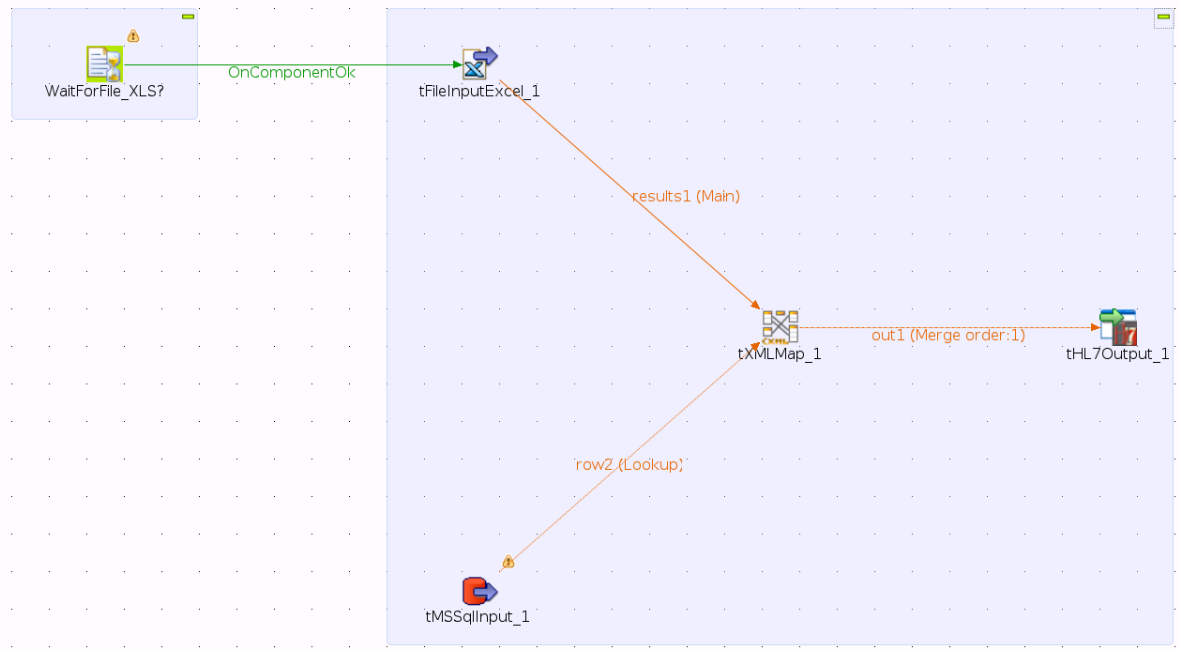
7.1.4 Implementacija prikazanim radnim okvirom

Nakon opisa ručne implementacije radnim okvirom Apache Camel, predstavljamo implementaciju koja je realizovana radnim okvirom za automatizovano mapiranje predstavljenim u poglavljima 4 i 5. Za potrebe opisa semantike sistema obuhvaćenih integracionim scenariom izrađena je ontologija. Za izradu ontologije korišćen je alat Protégé[4]. Zatim je kreiran TOS projekat koji opisuje integracioni scenario. U okviru projekta kreiran je TOS *Job*, prikazan na slici 7.2, kojim se obavlja agregacija podataka iz XLS i SQL izvora. Sličan Job kreiran je i za slučaj da je ulaz datoteka u CSV formatu.

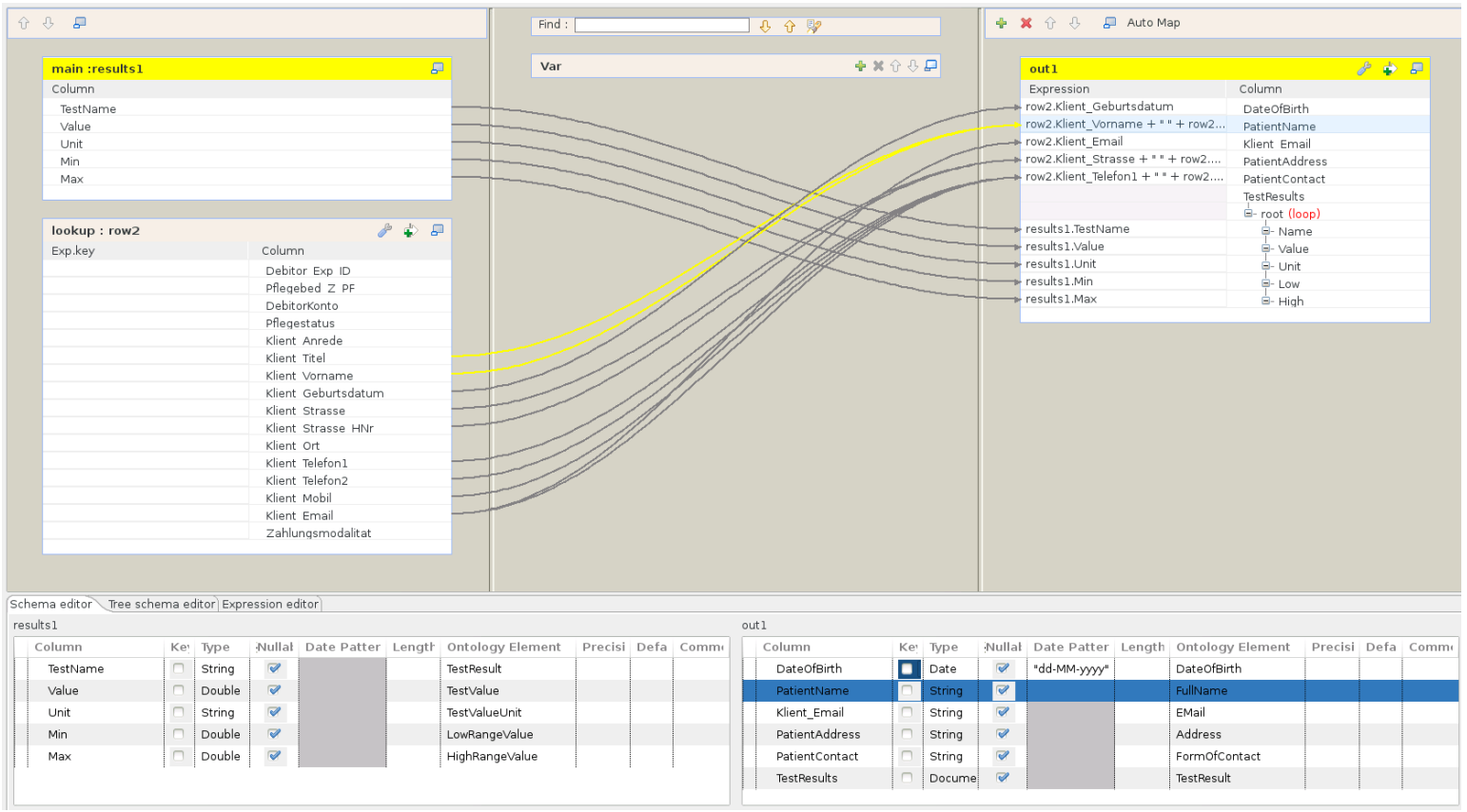
Proces agregacije započinje komponentom koja čeka da se u određenom direktorijumu pojave nove XLS datoteke. Pojava datoteka okida događaj koji aktivira `tFileInputExcel` ulazni konektor. Konektor čita podatke iz prispelih datoteka. U okviru `tFileInputExcel` komponente definisana je i šema Excel tabele, odnosno lokacije redova i kolona odakle se podaci čitaju. Paralelno sa čitanjem datoteka iz prvog izvora, obavlja se i upit nad tabelom baze podataka u kojoj se nalazi rezultat uskladištene procedure koja sakuplja detalje o korisniku doma. Ovo obavlja ulazni konektor `tMSSqlInput`, u okviru čijih podešavanja je definisana i šema date tabele. Definisanje šeme obavljeno je automatski, čitanjem metapodataka sa servera baze podataka.

Izlazni konektor `tHL7Output` zadužen je za slanje objedinjenih rezultata serveru bolnice u HL7 formatu. U okviru ove komponente definisana je šema HL7 dokumenta. Samo mapiranje dva ulazna na jedan izlazni interfejs obavlja komponenta `tXMLMap`, u okviru koje funkcioniše implementirani mehanizam za automatizovano mapiranje i detekciju konflikata. Izgled korisničkog interfejsa ove komponente, nakon završetka procesa mapiranja vidi se na slici 7.3. Grafički prikaz mapiranja vidljiv je u gornjem delu interfejsa. Ovaj prikaz omogućava korisniku kontrolu rezultata, kao i eventualne ručne intervencije nakon završetka automatskog procesiranja. U donjem delu komponente vidljiv je prikaz detalja selektovanog ulaznog i izlaznog interfejsa. U dijalogu za definisanje šeme korisnik može anotirati neki element interfejsa elementom ontologije, što se čini klikom na dugme u koloni Ontology Element. Isto se može učiniti na dijalozima svakog zasebnog konektora, uz definiciju strukture njegove šeme. Umesto u dijalogu svake od šema, anotiranje može biti obavljeno i u dijalogu komponente `tMap`, kojom se kontroliše mapiranje.

[4] https://protege.stanford.edu - Protégé - besplatan alat i radni okvir otvorenog koda namenjen uređivanju ontologija i razvoju inteligentnih sistema



Slika 7.2: TOS *Job* koji agregira XLS i SQL ulaze u HL7 izlaz



Slika 7.3: Mapiranje koje je automatski dobijeno radnim okvirom

7.2 Scenario 2: portal za vođenje projekata

Ovaj integracioni scenario je deo softverskog rešenja za praćenje projekata, razvijenog u okviru kompanije PI Informatik GmbH iz Berlina. Za scenario je postojalo rešenje ručno implementirano u ovoj kompaniji. Razvili smo drugo rešenje, korišćenjem ovde prikazanog prototipa radnog okvira za automatsko mapiranje i funkcionalno ga uporedili sa prethodnim.

7.2.1 Scenario

Portal za vođenje projekata nudi SOAP veb servis koji omogućava dobavljanje informacija o nekom projektu. Podaci o projektima jednog dela većeg poslovnog sistema se nalaze u starijoj verziji sistema, koja koristi SAP okruženje. Ovo starije rešenje se i dalje aktivno koristi, a podaci ažuriraju, pa prosti uvoz podataka iz starije u noviju verziju nije adekvatno rešenje. Neophodno je sprovesti integraciju. Kada SOAP servis primi zahtev, potrebno je pozvati BAPI funkciju starog sistema koja izvozi podatke o projektima, pretvoriti ih u XML oblik i kao takve vratiti SOAP odgovorom.

7.2.2 Formati i protokoli od interesa u scenariju

Remote Function Call Remote Function Call (RFC) je protokol za komunikaciju između različitih SAP sistema, kao i za omogućavanje pozivanja funkcija SAP aplikacija od strane drugih aplikacija, kao što je opisano u [1]. Postoje tri verzije, odnosno tri načina korišćenja SAP RFC:

- synchronous RFC (sRFC), koji koristi sinhronu komunikaciju,
- Transactional RFC (tRFC), koji podrazumeva asinhronu komunikaciju i garantuje transakcioni integritet,
- Queued RFC (qRFC), koji radi poput tRFC, ali vodi računa o redosledu operacija i
- Background RFC (bgRFC), koji omogućava i sinhroni i asinhroni način rada, kao i definisanje međuzavisnosti različitih redova čekanja operacija.

U svim verzijama RFC za prenos mogu koristiti CPI-C[5] ili TCP/IP protokol stek.

SAP BAPI BAPI (Business Application Programming Interface) je interfejs za integraciju SAP aplikacija sa drugim aplikacijama. Koriste ga i različite komponente SAP aplikacija za međusobnu saradnju i komunikaciju. Eksterne aplikacije mogu pozivati BAPI funkcije koristeći RFC (Remote Function Call) protokol. *SAP business object* (poslovni objekat) u

[5]Common Programming Interface for Communications (CPI-C) je protokol za međusobnu komunikaciju programa koji se izvršavaju na različitim sistemima, razvijen od strane IBM-a [2]

SAP sistemima predstavlja reprezentaciju nekog poslovnog entiteta (npr. narudžbina, klijent, banka). U skladu sa osnovnim principima objektno-orijentisane paradigme [129], obuhvata podatke - u vidu atributa i ponašanje - u vidu metoda. Interakcija sa SAP poslovnim objektima od strane drugih aplikacija ostvaruje se putem BAPI-ja. [1] Uz BAPI funkcije koje dolaze uz aplikaciju moguće je definisati i nove. Kako bi se pozvao neki BAPI metod, neophodno je poznavati sledeće:

- naziv BAPI-ja i
- detalje BAPI interfejsa, koji obuhvataju:
  - ulazne parametre,
  - izlazne parametre i
  - tabele.

Tabele predstavljaju rezultat izvršavanja BAPI-ja i kroz njih se može dobaviti vrednost instanci poslovnih objekata koji su u obuhvaćeni BAPI pozivom, kao i ažurirati ove vrednosti.

SOAP SOAP[6] je protokol namenjen razmeni struktuiranih informacija u decentralizovanom, distribuiranom okruženju. Oslanja se na XML i definiše radni okvir za razmenu poruka koji je nezavisan od jezika implementacije, kao i od transportnog protokola (može se koristiti npr. HTTP, HTTPS, SMTP, itd). [99] Najčešće je korišćen za implementaciju veb servisa.

WSDL Web Services Description Language (WSDL) je jezik za formalnu specifikaciju mrežnih servisa [27]. Ovim jezikom se, u XML formatu, mogu zadati opisi pristupnih tačaka (endpoint) servisa, kao i poruka koji se njima razmenjuju. Jezik je nezavisan od formata poruka i mrežnih protokola koji se koriste za prenos, ali se najčešće koristi za opis SOAP servisa preko HTTP. Deo WSDL definicije servisa koji učestvuje u ovom integracionom scenariju prikazan je na listingu 7.5.

```
<?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions name="SI_request_PSP_Portal"
  targetNamespace="http://www.pi-informatik.de/io"
  xmlns:rfc="urn:sap-com:document:sap:rfc:functions"
  xmlns:p1="http://www.pi-informatik.de/io"
  xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/">
  <wsdl:documentation />
  <wsdl:types>
    <xsd:schema targetNamespace="urn:sap-com:document:sap:rfc:
    functions"
      xmlns="urn:sap-com:document:sap:rfc:functions"
      xmlns:xsd="http://www.w3.org/2001/XMLSchema">
      <xsd:element name="BAPI_PROJECT_GETINFO">
        <xsd:complexType>
```

[6]U verziji 1.1 specifikacije, ime protokola je *SOAP (Simple Object Access Protocol)*. U verziji 1.2, naziv protokola je *SOAP* i napominje se da više ne predstavlja skraćenicu.

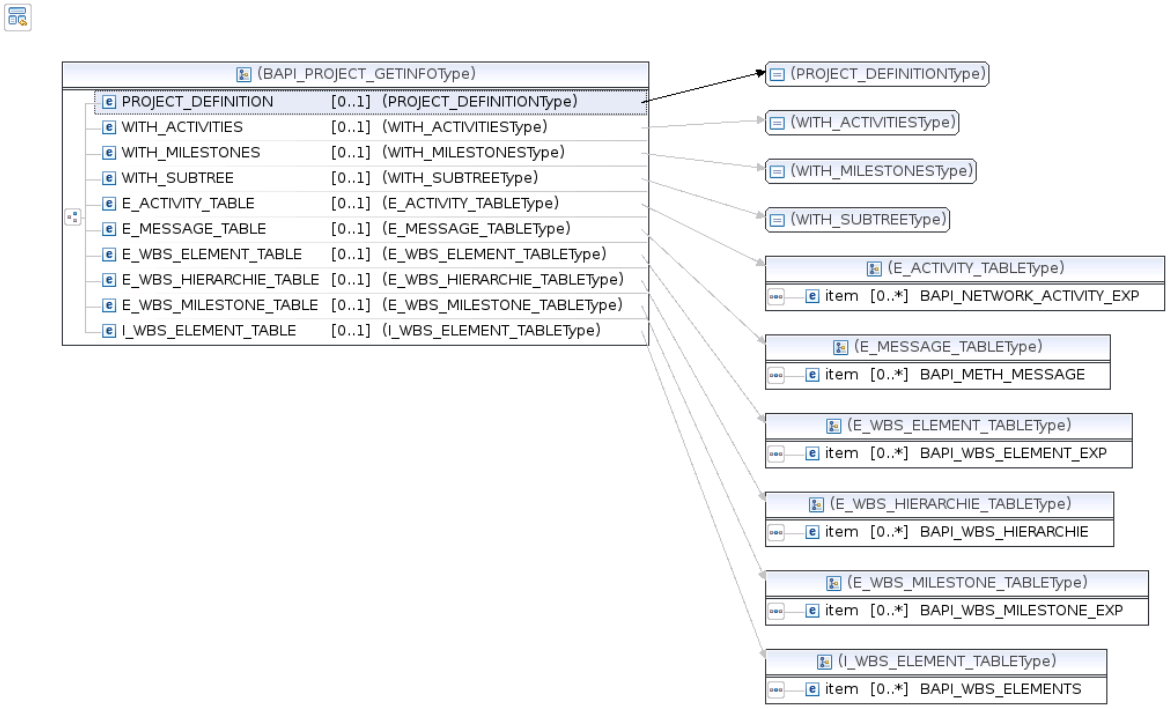
```
14          <xsd:all>
              <xsd:element name="PROJECT_DEFINITION" minOccurs="0">
16              <xsd:simpleType>
                  <xsd:restriction base="xsd:string">
18                  <xsd:maxLength value="24" />
                  </xsd:restriction>
20              </xsd:simpleType>
              </xsd:element>
22            <xsd:element name="WITH_ACTIVITIES" minOccurs="0">
                <xsd:simpleType>
24                <xsd:restriction base="xsd:string">
                    <xsd:maxLength value="1" />
26                </xsd:restriction>
                </xsd:simpleType>
28            </xsd:element>
              <xsd:element name="WITH_MILESTONES" minOccurs="0">
30              <xsd:simpleType>
                  <xsd:restriction base="xsd:string">
32                  <xsd:maxLength value="1" />
                  </xsd:restriction>
34              </xsd:simpleType>
              </xsd:element>
36            <xsd:element name="WITH_SUBTREE" minOccurs="0">
                <xsd:simpleType>
38                <xsd:restriction base="xsd:string">
                    <xsd:maxLength value="1" />
40                </xsd:restriction>
                </xsd:simpleType>
42            </xsd:element>

44        ...

46          </xsd:all>
          </xsd:complexType>
48      </xsd:element>
    ...
```

Listing 7.5: Deo WSDL definicije veb servisa koji učestvuje u scenariju 2

7.2.3 Implementacija

Implementacija integracionog rešenja za ovaj scenario počinje definisanjem servisa. Ovo je u TOS-u moguće učiniti uvozom WSDL datoteke ili ručno, kroz korisnički intefejs konektora. U oba slučaja, korisnik pored izvorne, tekstualne XML predstave, dobija grafičku predstavu pristupnih tačaka servisa, kao i formata poruka. Kako je WSDL datoteka bila na raspolaganju za ovaj scenario, ista je uvezena, a grafički prikaz formata zahteva BAPI_PROJECT_GETINFO prikazan je na slici 7.4. U okviru WSDL-a definiše se i adresa

i port na kojoj će servis očekivati zahteve.



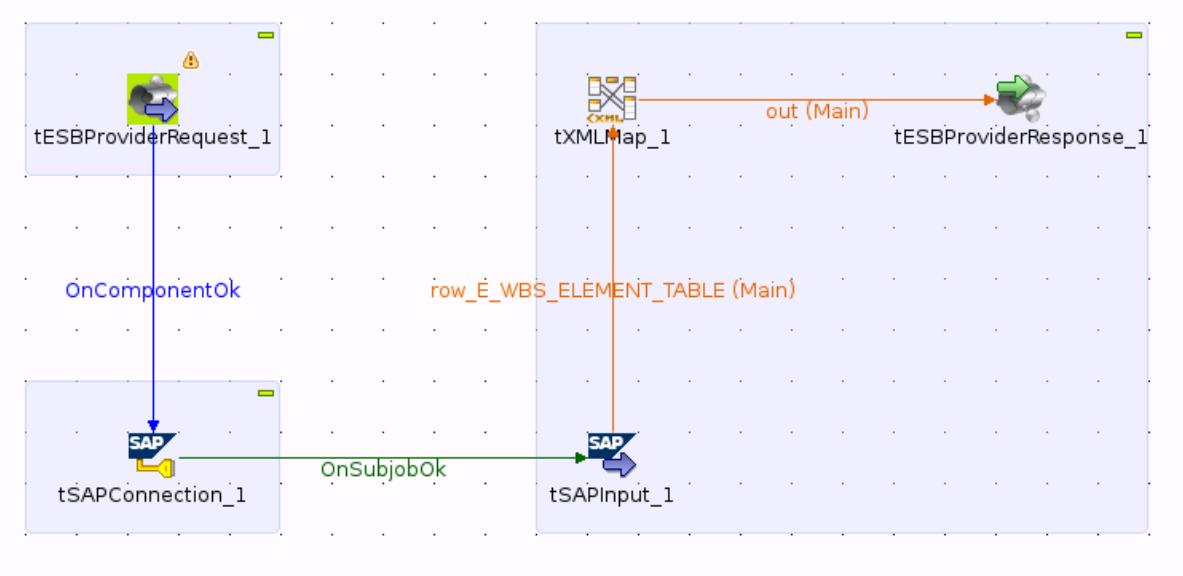
Slika 7.4: Grafički prikaz dela WSDL definicije servisa

.

Nakon definicije strukture veb servisa, prelazi se na definisanje procesa koji se odvija kada do ovog servisa dođe zahtev. U TOS, ovo se obavlja kreiranjem takozvanog *job*-a, koji može sadržati ulazne i izlazne komponente, kao i komponente procesore između njih. Proces ovog rešenja prikazan je na slici 7.5.

Proces počinje komponentom **tESBProviderRequest**, koja služi za prijem zahteva ka veb servisu. Ukoliko servis primi ispravan zahtev, okida se događaj *OnComponentOk*. Ovaj događaj je doveden na komponentu **tSAPConnection**, koja po prijemu događaja inicira konekciju ka SAP aplikaciji. Za konekciju se definiše:

- Server Type - Application Server ili Message Sever,
- Client - jedan SAP sistem može opsluživati više kompanija ili organizacionih jedinica koje na neki način predstavljaju zasebnu celinu (npr. teritorijalno), pa se svakoj dodeljuje trocifren *Client* kod; različitim kodovima mogu se odvojiti i razvojno, test i produkciono rešenje, koja se izvršavaju na istom sistemu,
- Userid - korisničko ime,
- Password - lozinka za pristup,
- Language - dvoznačni kod jezika sistema kom se pristupa (npr. EN za engleski),
- Host name - IP ili simbolička adresa za pristup sistemu i
- System number - broj instance u opsegu 00 do 99, koji se podešava prilikom instalacije SAP sistema.

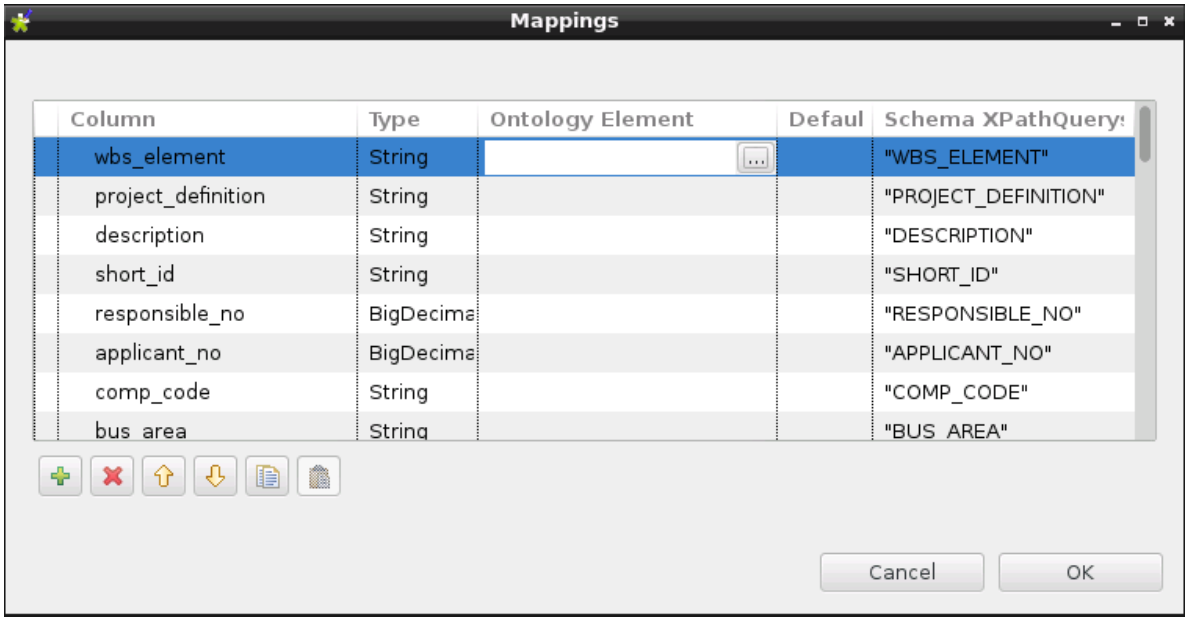


Slika 7.5: Grafički prikaz TOS *job*-a koji povezuje veb servis i SAP

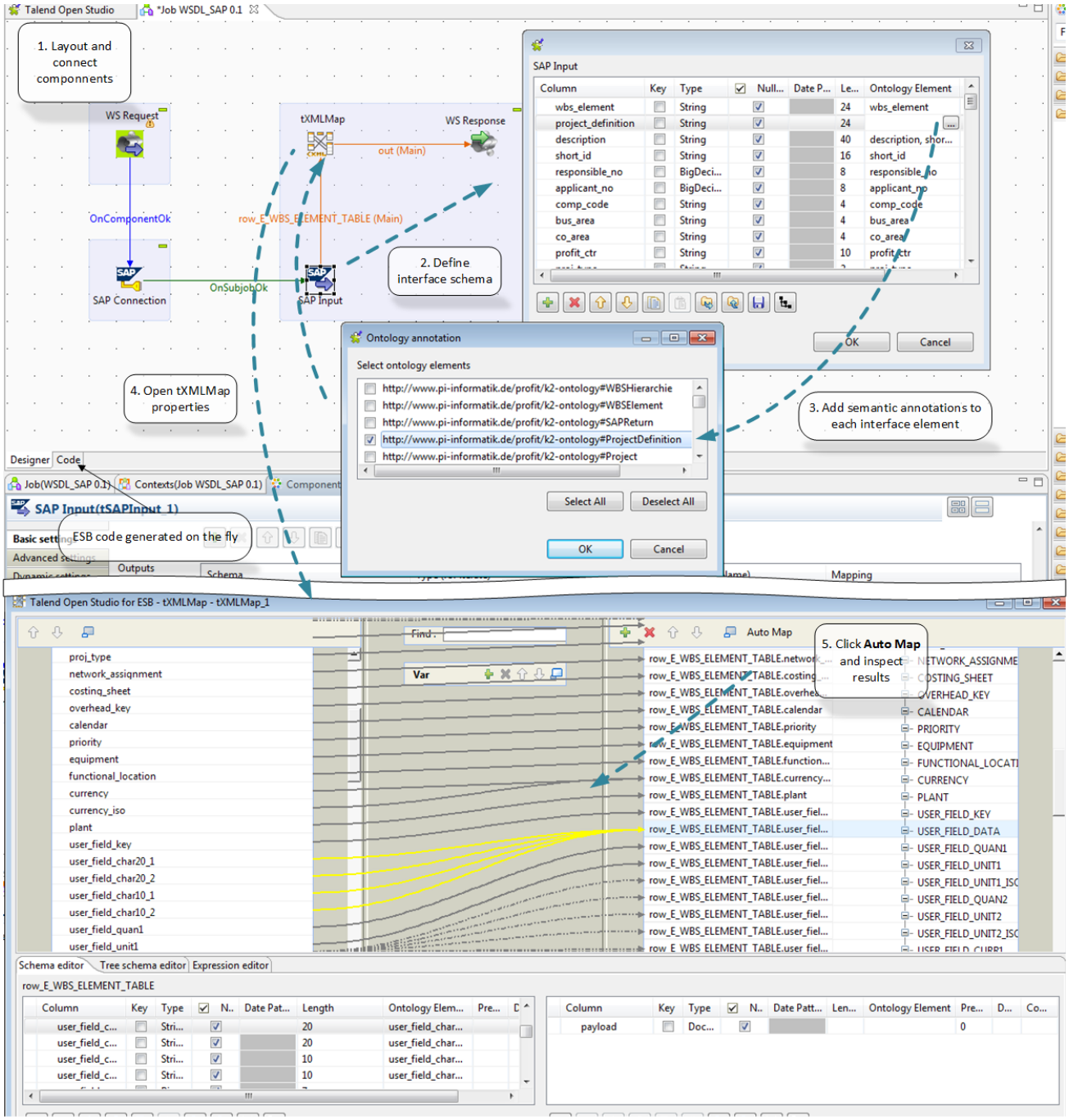
Ukoliko `tSAPConnection` komponenta uspešno ostvari konekciju sa SAP sistemom, okida događaj `OnSubjobOk`. Ovaj događaj je doveden na komponentu **`tSAPInput`**, koja služi za dobavljanje podataka iz SAP sistema pozivanjem BAPI funkcije. Kada dobije signal da je konekcija uspešna, ova komponenta je podešena da poziva funkciju `BAPI_PROJECT_GETINFO`. Prilikom poziva, prosleđuje se parametar `PROJECT_DEFINITION` postavljen na vrednost `"I+O PROJEKTDEFINITION"`. Izlaz funkcije je SAP tabela pod nazivom `E_WBS_ELEMENT_TABLE`, koja sadrži podatke o projektima. U okviru komponente `tSAPInput` definiše se struktura izlaznih parametara, uključujući strukturu ove tabele. Dijalog za zadavanje strukture tabele prikazan je na slici 7.6. Klikom na dugme u koloni *Ontology Element* ovog dijaloga, dobija se mogućnost anotiranja svake kolone tabele elementom ontologije. Moguće je pridružiti i više elemenata ontologije jednoj koloni tabele, koja u ranije uvedenoj terminologiji predstavlja jedan element ulaznog interfejsa.

Komponenta **`tMap`** služi za rutiranje i transformaciju. U sklopu ove komponente može se definisati kako se proizvoljan broj ulaznih interfejsa mapira na proizvoljan broj izlaznih interfejsa. Ovo je komponenta koja je najviše modifikovana kako bi bio implementiran predstavljeni radni okvir. Dijalog ove komponente podeljen je na tri dela. U levom delu se zadaju ulazni interfejsi, u desnom izlazni, dok se u središnjem delu mogu definisati pomoćne promenljive i izrazi za transformaciju. Interfejsi mogu predstavljati i složene strukture i moguće je kroz njih iterirati.

U slučaju scenarija koji trenutno obrađujemo, ulazni interfejs je tabela čiji svaki red predstavlja jedan projekat, a izlazni interfejs je XML struktura koja će biti prosleđena kao telo odgovora veb servisa i koja sadrži informacije o svim traženim projektima. Pregled svih koraka neophodnih za dobijanje automatskog mapiranja, kao i njegov rezultat u slučaju ovog scenarija, prikazani su na slici 7.7.



Slika 7.6: Definicija kolona tabele E_WBS_ELEMENT_TABLE i kontrola za anotiranje svake kolone ove tabele elementima ontologije



Slika 7.7: Koraci potrebni za dobijanje automatskog mapiranja sa rezultatima

7.3 Eksperimentalna evaluacija

Cilj ove eksperimentalne evaluacije je poređenje vremena neophodnog za ručno mapiranje interfejsa sa vremenom potrebnim za mapiranje istih interfejsa korišćenjem razvijene komponente za automatsko mapiranje. Za ručno mapiranje korišćen je ranije opisani softverski alat Talend Open Studio, a za automatsko mapiranje korišćen je naš prototip razvojnog okvira opisanog u [140], gde su prvobitno objavljeni i rezultati ovog eksperimenta. U eksperimentu su učestvovale dve grupe studenata treće i četvrte godine fakulteta, pri čemu ukupan broj studenata na datim godinama iznosi oko 200. Izračunato je da je, za ukupnu populaciju ove veličine, neophodno da u eksperimentu učestvuje bar 83 studenata, kako bi se zadovoljio nivo od 95 % sigurnosti, uz odabranu marginu greške koja iznosi 5 %, za očekivanu raspodelu 90 % rezultata. Ukupno je u eksperimentu učestvovalo 93 studenata, pa je ovaj kriterijum zadovoljen. Učesnici su podeljeni u dve grupe:

- grupa od 46 studenata, koji ručno mapiraju interfejse - kontrolna grupa i
- grupa od 47 studenata, koji mapiraju interfejse pomoću komponente za automatsko mapiranje.

Kao nultu hipotezu eksperimenta uzimamo tvrdnju da ne postoji razlika u vremenu potrebnom da se mapiranje obavi ručno i vremenu potrebnom da se mapiranje obavi pomoću komponente za automatsko mapiranje. Pretpostavka je da postoji ontologija koja opisuje interfejse koji će biti mapirani i vreme potrebno za izradu ove ontologije se ne uzimaju u obzir pri merenju vremena neophodnom za mapiranje. Svi učesnici u eksperimentu su zadatke obaljali pod istim okolnostima (u istoj prostoriji), na istim virtuelnim mašinama, koje su se izvršavale na jednakim hardverskim konfiguracijama.

Obe grupe su imale za zadatak da mapiraju četiri para interfejsa (svaki par ponaosob), pri čemu je kompleksnost intefejsa različita. Mereno je i beleženo vreme potrebno za mapiranje svakog para. Ispitanici iz grupe koja je koristila alat za automatsko mapiranje imali su i dodatni zadatak, u kom su elemente interfejsa anotirali elementima ontologije, s obzirom da je ovo preduslov za rad automatskog mapera. Kako bi rezultati eksperimenta bili validni, neophodno je da svi ispitanici jednako poznaju semantiku interfejsa koji se mapiraju. Kako bi ovaj uslov bio ispunjen, umesto stvarnih sistema i interfejsa, korišćena su slova engleskog alfabeta i njihov redni broj u alfabetu.

Prvi zadatak. Ulazni intefejs sadrži 26 slova engleskog alfabeta, poređanih po redosledu iz alfabeta. Izlazni interfejs, takođe, sadrži 26 slova engleskog alfabeta, ali poređanih na slučajan način. Potrebno je svako slovo ulaznog interfejsa mapirati na isto slovo u izlaznom interfejsu (npr. povezuje se *A* sa *A*, *B* sa *B*, itd).

Drugi zadatak. Ulazni intefejs sadrži 26 slova engleskog alfabeta, poređanih po redosledu iz alfabeta. Nazivi elemenata izlaznog interfejsa predstavljaju redne brojeve slova u alfabetu, poređane na slučajan način. Zadatak je spojiti svako slovo sa svojim rednim brojem (*A* se mapira na *Letter1*, *B* se mapira na *Letter2*, itd).

Treći zadatak. U ovom zadatku, elementi ulaznog interfejsa su ponovo slova, a elementi

Tabela 7.1: Rezultati eksperimenta. $\bar{t}$ - prosečno vreme, σ - standardna devijacija

	Ručno				Auto Map				
Zadatak	1	2	3	4	Anotiranje	1	2	3	4
$\bar{t}$ [m:s]	2:12	2:33	3:02	3:47	8:17	0:07	0:13	0:28	0:21
σ [m:s]	0:41	1:33	0:59	0:59	1:54	0:05	0:08	0:20	0:11

izlaznog interfejsa se sastoje od nekoliko slova. Svako slovo se mapira na sve parove koji sadrže to slovo. Na primer, *A*, *D* i *F* treba mapirati na *A_D_F*. Postoji 20 ovakvih parova.

Četvrti zadatak. U poslednjem zadatku, elementi izlaznog interfejsa su kombinacija slova i rednih brojeva slova. Potrebno je mapirati, na primer, *A*, *B* i *F* na element *Letter1_B_Letter6*. Ovaj zadatak je takav da tokom automatskog mapiranja proizvodi nekoliko konflikata tipa *Višestruko mapiranje po istom osnovu* (vidi 4.3.1), koje korisnik treba da razreši odgovorom na dijalog nalik onom na slici 4.3.

Proseci zabeleženih vremena prikazani su u tabeli 7.1.

Zaključak eksperimentalne evaluacije

Ukoliko posmatramo prosečna vremena potrebna za obavljanje četiri prikazana zadatka, vidljiva je jasna prednost automatskog postupka mapiranja, čak i u slučaju kada se od korisnika očekuje „pomoć“ u vidu razrešavanja konflikata. Statistička analiza potvrđuje ovakav zaključak. Nad prikupljenim podacima urađen je Velčov (Welch) t-test za dva uzorka, a rezultat je p-vrednost manja od 10^{-16}, pa se nulta hipoteza testa može odbaciti sa visokom sigurnošću.

Ukoliko se uzme u obzir i anotiranje interfejsa semantikom, vidi se da ovaj preduslov za automatsko mapiranje iziskuje dodatno vreme. Iz ovoga se može zaključiti da je, u pogledu skraćenja vremena implementacije integracionog rešenja, korišćenje prikazane komponente i postupka za automatsko mapiranje pogodno za slučajeve gde se isti elementi javljaju u većem broju interfejsa i gde je broj takvih interfejsa veći. U slučaju malog broja interfejsa, koji ne sadrže elemente koji se ponavljaju, ručni pristup može dati bolje rezultate u pogledu vremena implementacije.

8

Zaključak

Integracija poslovnih aplikacija je proces kojim se uspostavlja razmena podataka ili funkcionalnosti između dve ili više aplikacija. Motiv za sprovođenje integracije je da se izbegne potreba da se isti podaci ručno unose i ažuriraju na više mesta, kao i da se omogući da pojedinačne aplikacije učestvuju u kompleksnim poslovnim procesima, tako da svaka obavlja one funkcionalnosti za koje je razvijena. Sam proces integracije može biti kompleksan i dugotrajan, pa samim tim i skup. Istraživanje koje je sproveo Forrester Research pokazalo je da Fortune-1000 kompanije troše 35 % svog budžeta za održavanje softvera za integraciju svojih sistema [114].

Za sprovođenje integracije neophodno je široko znanje, koje obuhvata sve tehnologije, formate i protokole koji su korišćeni od strane pojedinačnih aplikacija, kao i različitih tehnika same integracije. Alati za integraciju pomažu u tehničkom pogledu. Ovi alati poseduju velik broj ulaznih i izlaznih konektora, koji omogućavaju lakši rad sa mnogim formatima i protokolima. U nekim slučajevima moguća je i automatska ekstrakcija šeme podataka koju koriste aplikacije nad datim formatima. Ono što je takođe moguće upotrebom ovih alata je i automatska konverzija različitih načina predstavljanja, zapisa i prenosa podataka. Ove razlike predstavljaju sintaktičke konflikte među aplikacijama koje se integrišu. Drugu klasu konflikata predstavljaju razlike u pogledu značenja podataka u različitim aplikacijama. Ovakve razlike nazivamo semantičkim konfliktima. Cilj ovog istraživanja bio je pronalaženje mehanizma kojim bi se omogućilo automatsko mapiranje između interfejsa različitih aplikacija koje učestvuju u određenom integracionom scenariju, kao i automatska detekcija semantičkih konflikata među njima.

U ovoj tezi prikazan je radni okvir za semantičku integraciju poslovnih aplikacija. Dati radni okvir omogućava korišćenje ontologija kao formalne definicije semantike aplikacija koje se integrišu, koja se kombinuje sa modelima strukture podataka i ponašanja, čime se omogućuje poluautomatizovano uspostavljanje mapiranja između elemenata interfejsa datih aplikacija. Tok procesa integracije, opisan u sekciji 4.1 podrazumeva: (1) izradu strukturnog modela integracionog rešenja, (2) učitavanje ontologije koja opisuje semantiku elemenata interfejsa aplikacija koje se integrišu, (3) anotiranje, odnosno povezivanje elemenata ontologije sa elementima strukturnog modela, (4a) automatizovanu detekciju kandidata za

međusobno mapiranje izlaznih i ulaznih elemenata interfejsa i (4b) detekciju semantičkih konflikata, uz mogućnost automatizovanog razrešenja nekih konflikata, (5) grafički prikaz rezultata mapiranja korisniku, uz mogućnost ručne izmene i (6) generisanje koda izvršivog integracionog rešenja. Implementacija ovog radnog okvira testirana je na više realnih integracionih scenarija, od kojih su dva detaljno analizirana u poglavlju 7. Za potrebe svakog scenarija kreirana je ontologija koja opisuje semantiku aplikacija koje se integrišu. Za svaki interfejs kreirane su šeme podataka interfejsa, odnosno model koji specificira strukturu podataka. Kreiran je i opis svakog od scenarija, koji definiše tok podataka i ponašanje svake od aplikacija u okviru scenarija. Svaki od elemenata interfejsa anotiran je elementom iz ontologije koji ga opisuje. Na osnovu ovoga, radni okvir je na osnovu raspoloživih kriterijujma za mapiranje (prikazanih u 4.2) uspešno automatski uspostavio mapiranja između ulaznih i izlaznih interfejsa. Ovime je **u potpunosti ispunjena *hipoteza 0***.

Kriterijumi za mapiranje predstavljaju specifikaciju uslova koji treba da budu ispunjeni kako bi bilo uspostavljeno mapiranje između dva ili više elemenata interfejsa aplikacija koje se integrišu. Slično, kriterijumi za detekciju konflikata predstavljaju zasebne komponente. Arhitektura radnog okvira je takva da svaki od ovih kriterijuma predstavlja zasebnu komponentu. Ovo omogućava da se kriterijumi po potrebi uključuju ili isključuju od strane korisnika, da se specifikacija kriterijuma menja, kao i da se definišu i implementiraju potpuno novi kriterijumi koji odgovaraju nekom konkretnom integracionom scenariju. Svaki novi razvijeni kriterijum može se, po potrebi, ponovo koristiti za druge scenarije. Ovime su **ispunjeni uslovi *hipoteze 1***.

Iako arhitektura radnog okvira omogućava da se po potrebi kriterijumi mapiranja i detekcije prilagođavaju konkretnim scenarijima, za razvijanje komponenti kojima se oni implementiraju, neophodno je koristiti jezik opšte (u slučaju referentne implementacije, u pitanju je jezik Java) namene i poznavati arhitekturu kako radnog okvira za automatizovanu konfiguraciju, tako i arhitekturu aplikacije za integraciju (Talend Open Studio) u okviru koje se referentna implementacija izvršava. Kako bi se pojednostavilo razvijanje komponenti radnog okvira, razvijen je jezik specifičan za domen, zvani Semantic Application Integration Language. Jezik omogućava da se opisno definišu komponente radnog okvira: kriterijumi za mapiranje, kriterijumi za detekciju konflikata, načini njihovog razrešavanja, kao i načini izgradnje izraza koji definišu konačno mapiranje elemenata interfejsa. Jezik je evaluiran kreiranjem specifikacije svih komponenti koje su prethodno ručno implementirane i korišćene prilikom evaluacije samog radnog okvira. Ovime su **ispunjeni uslovi *hipoteze 2***.

Za kvantitativno poređenje vremena neophodnog za ručno mapiranje elemenata interfejsa aplikacija koje se integrišu sa vremenom neophodnim da se isto mapiranje obavi razvijenim sistemom za automatizovano mapiranje, eksperiment je konstruisan tako da su zadaci nezavisni od poslovnog domena primene. Rezultati, prikazani u poglavlju 7 pokazuju da je sprovođenje razvijenog postupka isplativo kada je očekivano da će određeni interfejs učestvovati u više scenarija, kao i u slučaju da je struktura interfejsa složena ili broj elemenata velik.

Pored eksperimentalne evaluacije, praktična upotreba razvijenog alata je ispitana kroz dva realna realna integraciona scenarija. Za svaki od ovih scenarija, rešenje dobijeno upotrebom pristupa i alata prikazanog u ovoj tezi, bilo je funkcionalno ekvivalentno prethodno

ručno razvijenom integracionom rešenju.

Potreba za integracijom je široko rasprostranjena, a samim tim mogućnost primene razvijenog pristupa je široka. Automatizacija bar dela procesa integracije ima za cilj značajno smanjenje troškova, bržu realizaciju i smanjenje grešaka prilikom integracije.

Na primer, informacioni sistem vatrogasno-spasilačke službe u Srbiji sastoji se od dve do pet (u zavisnosti od konkretne područne jedinice) aplikacija. Aplikacije su uvođene u etapama i izražena je visoka heterogenost. U prvi mah, početkom devedesetih godina prošlog veka, izrađivane su od strane entuzijasta, pojedinaca. Kasnije su, u nekoliko navrata, uvođene namenski kreirane aplikacije rađene po zahtevima nadležnog Ministarstva. U trenutnom stanju, iste podatke neophodno je ažurirati u više aplikacija. Integracija ovog sistema omogućila bi efikasniji utrošak vremena korisnika, što bi u ovom slučaju značilo da oni imaju više vremena da se posvete konkretnim zadacima iz oblasti rada svoje službe. Olakšani unos bi uz objedinjeni prikaz omogućio da se podaci koriste u realnom vremenu, kao reprezent aktivnih i raspoloživih sredstava i ljudstva, što bi uz dosadašnji, statistički, aspekt korišćenja informacionog sistema dodatno povećalo efikasnost i unapredilo bezbednost građana i imovine.

Dalja istraživanja mogu teći u pravcu eventualne potpune automatizacije celokupnog procesa integracije. U svetlu metodologije koju smo ovde koristili, to bi zahtevalo automatizaciju dobijanja ontologija svakog sistema koji se integriše, automatizaciju spajanja pojedinačnih ontologija u jednu, kao i automatizaciju anotacije strukturalnog modela semantikom iz ove spojene ontologije. Jedan izvor za ekstrakciju ontologija može biti tehnička i korisnička dokumentacija. Pregled ovakvih metoda, koje kao ulaz koriste struktuiran i nestruktuiran tekst dat je u [51]. Pored ovog, kao izvor se može koristiti korisnički interfejs aplikacije, kao i podaci. Primer ovakvih metoda je patent [123].

Kao što je ranije pomenuto, procedura automatskog mapiranja obavlja se tako što se koristi niz kriterijuma za mapiranje, od kojih svaki odlučuje da li će neki par ulaznih i izlaznih elemenata interfejsa biti kandidat za mapiranje. Više kriterijuma može dati potvrdan odgovor za određeni par elemenata. Svi oni čine skup *razloga* za mapiranje datog para. Da li će mapiranje zaista biti uspostavljeno odlučuje se u sledećoj fazi, gde se obavlja detekcija i razrešavanje konflikata. Ukoliko je neki par kandidat samo po jednom kriterijumu, mapiranje se uspostavlja. Ukoliko je par kandidat po više kriterijuma, po osnovu zadatih definicija konflikata dobija se razrešenje da li kandidat ostaje na snazi ili se odbacuje. Ukoliko ni jedan kriterijum konflikata ne daje razrešenje za dati skup kriterijuma mapiranja, odluka se prepušta korisniku. Postoji i kriterijum zabrane mapiranja (4.2.9), kojim se eksplicitno određuje da neki par ne treba da bude mapiran. Ukoliko se pojavi zajedno sa bilo kojim drugim kriterijumom u skupu razloga za mapiranje nekog kandidata, kriterijum zabrane ima prioritet i taj par sigurno neće biti mapiran. Neka rešenja, poput [9] imaju drugačiji, *fuzzy* pristup ovom procesu. U ovakvim pristupima, svaki kriterijum mapiranja daje normalizovanu ocenu u intervalu [0,1] da li neki par treba da rezultuje mapiranjem. Kaže se i da ova vrednost predstavlja ocenu *sličnosti* datih elemenata. Nakon dobijanja rezultata po svakom raspoloživom kriterijumu, računa se prosečna vrednost ovih ocena, a mapiranje se uspostavlja ukoliko rezultujuća vrednost pređe određeni prag. Težinski faktori mogu biti pridruženi svakom od kriterijuma, čime se uticaj njegove ocene može favorizovati ili umanjiti. Pristup

koji koristi alat AnyMap [34] proširuje statistički vid detekcije mapiranja time što čuva vrednosti ocena sličnosti u repozitorijumu za ponovno korišćenje i koristi ih u slučaju da je naknadno potrebno razmatrati mapiranje nekih od ovih elemenata. Arhitektura našeg radnog okvira dovoljno je fleksibilna da može da podrži implementaciju i ovakvog načina odluke o mapiranju elemenata. Dovoljno bi bilo učitati implementacije kriterijuma za mapiranje koji su konstruisani da daju rasplinute vrednosti i implementirati kriterijum detekcije i razrešenja konflikata koji njihove vrednosti objedinjuje na željeni način. Možda još značajnije, bilo bi moguće realizovati hibridni pristup koji kombinuje deterministički i statistički, gde se svaki koristi za klasu kriterijuma za koju je pogodniji.

U pogledu klasifikacije integracionih rešenja po [96, 50] datih u sekciji 2, prikazana implementacija radnog okvira generiše horizontalno integraciono rešenje. Po potrebi, implementacija se može izmeniti tako da, bez izmene ulazne specifikacije, generiše i drugu vrstu arhitekture.

Pri razmatranju praktične upotrebe bilo kog tehničkog, pa i ovog, pristupa, ne smeju se zanemariti ni bezbednosni i pravni aspekti. U pogledu prvog, integracija uopšte je na kritičnoj poziciji. Prilikom preuzimanja podataka i funkcionalnosti iz više aplikacija, mora se voditi računa o tome nad kojim od resursa koji učestvuju u interakciji ima prava korisnik koji im pristupa kroz integraciono rešenje. Pri tom, nije isključen scenario u kom korisnik nema pravo pristupa ni jednoj od pojedinačnih aplikacija koje se integrišu, ali ima prava nad rezultatom neke obrade u kojoj one učestvuju. Razmotrimo primer takvog scenarija iz medicinskog domena. Naučni istraživači imaju potrebu da analiziraju statističke podatke o određenim bolestima. Kako bi podaci bili relevantni, treba da obuhvate podatke iz više medicinskih centara. Svaki od tih centara koristi različit sistem za vođenje podataka o pacijentima. Kako bi se skratilo vreme do izvođenja zaključaka iz prikupljenih podataka i mogućnosti za pravovremeno reagovanje, na primer u slučaju pojave određenih epidemija, odlučeno je da se sprovede integracija. Korisnici ovog sistema moraju imati uvid u statistiku koja proizilazi iz podataka o pacijentima, ali ne smeju imati uvid u njihove pojedinačne kartone.

Pravni aspekti mogu biti od značaja ne samo u trenutku izvršavanja i korišćenja integracionog rešenja, već i u samom procesu integracije. Ovaj, kao i drugi pristupi automatizaciji integracije teže mogućnosti da određeni produkti procesa integracije budu ponovo korišćeni u slučaju da u budućnosti učestvuju u drugom integracionom scenariju. Jedan od takvih artefakata pristupa koji se bazira na uključivanju semantike jesu upravo ontologije koje opisuju sisteme koji se integrišu. Ovim ontologijama, suštinski se beleži nečije znanje o datim sistemima. Sa pravnog aspekta, može se postaviti, međutim, pitanje prava intelektualne svojine nad ovim znanjem i eventualnih ograničenja u njegovom korišćenju. Još jedno od pitanja koje se može javiti u trenutku razvijanja integracionog rešenja jeste i međusobna kompatibilnost licenci svake od komponenti i aplikacija koje se javljaju u samom scenariju ili se razmatra njihovo uvođenje u rešenje kako bi se olakšao njegov razvoj.

Možemo zaključiti da bi još jedan pravac daljeg razvoja predstavljao nalaženje načina da se modeluju pravni i bezbednosni aspekti aplikacija koji učestvuju u integraciji, kao i krajnjeg integracionog rešenja. Ovi modeli bi predstavljali još jedan od ulaza informacija za automatizovan proces integracije. Cilj njihovog razmatranja je da se na vreme detektuju konflikti

ove prirode, kako bi se izbegle njihove posledice prilikom kasnijeg korišćenja integracionog rešenja.

Bibliografija

[1] *SAP Documentation, https://help.sap.com*.

[2] *CPI-C Reference*, 1998.

[3] *Air Transport & Travel Industry XML Implementation Guide, Aviation Information Data Exchange (AIDX)*. International Air Transport Association, 2014.

[4] *Implementing Oracle Integration Cloud Service: Understand everything you need to know about Oracle's Integration Cloud Service and how to utilize it optimally for your business*. Packt Publishing, 2017.

[5] Ibrahim Ahmed Al-Baltah, Abdul Azim Abdul Ghani, Wan Nurhayati Wan Ab Rahman, and Rodziah Atan. Semantic conflicts detection of heterogeneous messages of web services: challenges and solution. *Journal of Computer Science*, 10(8):1428, 2014.

[6] Bogdan Alexe, Wang-Chiew Tan, and Yannis Velegrakis. Comparing and evaluating mapping systems with stbenchmark. *Proceedings of the VLDB Endowment*, 1(2):1468–1471, 2008.

[7] OSGi Alliance. *Osgi service platform, release 3*. IOS Press, Inc., 2003.

[8] Nenad Anicic, Nenad Ivezic, and Albert Jones. An architecture for semantic enterprise application integration standards. In *Interoperability of Enterprise Software and Applications*, pages 25–34. Springer, 2006.

[9] David Aumueller, Hong-Hai Do, Sabine Massmann, and Erhard Rahm. Schema and ontology matching with COMA++. In *Proceedings of the 2005 ACM SIGMOD international conference on Management of data*, pages 906–908. Acm, 2005.

[10] Franz Baader, Diego Calvanese, Deborah McGuinness, Peter Patel-Schneider, and Daniele Nardi. *The description logic handbook: Theory, implementation and applications*. Cambridge university press, 2003.

[11] Beckett David, Berners-Lee Tim, Prud'hommeaux Eric, and Carothers Gavin. RDF 1.1 Turtle, 2014.

[12] T Berners-Lee, J Hendler, and O Lassila. The semantic web. *Scientific american*, 2001.

[13] Tim Berners-Lee, James Hendler, and Ora Lassila. The semantic web. *Scientific american*, 284(5):34–43, 2001.

[14] Philip A Bernstein, Sergey Melnik, and John E Churchill. Incremental schema matching. In *Proceedings of the 32nd international conference on Very large data bases*, pages 1167–1170. VLDB Endowment, 2006.

[15] Wai Fong Boh, Christina Soh, and Steven Yeo. Standards development and diffusion: A case study of rosettanet. *Commun. ACM*, 50(12):57–62, December 2007.

[16] Willem Nico Borst and WN Borst. Construction of engineering ontologies for knowledge sharing and reuse. 1997.

[17] BPMI BPML. Business process modeling language 1.0. *Business Process Management Initiative*, 2001.

[18] Tim Bray, Jean Paoli, C Michael Sperberg-McQueen, Eve Maler, and François Yergeau. Extensible markup language (XML). *World Wide Web Journal*, 2(4):27–66, 1997.

[19] Karin Breitman, Marco Antonio Casanova, and Walt Truszkowski. *Semantic web: concepts, technologies and applications*. Springer Science & Business Media, 2007.

[20] Dan Brickley and Ramanathan Guha. RDF vocabulary description language 1.0: RDF schema. W3C recommendation, W3C, February 2004. http://www.w3.org/TR/2004/REC-rdf-schema-20040210/.

[21] Agustina Buccella, Alejandra Cechich, and Nieves Rodríguez Brisaboa. An ontology approach to data integration. *Journal of Computer Science & Technology*, 3, 2003.

[22] Felix Burgstaller, Dieter Steiner, Michael Schrefl, Eduard Gringinger, Scott Wilson, and Sam Van Der Stricht. AIRM-based, fine-grained semantic filtering of notices to airmen. In *Integrated Communication, Navigation, and Surveillance Conference (ICNS), 2015*, pages D3–1. IEEE, 2015.

[23] Carothers Gavin and Seaborne Andy, editors. *RDF 1.1 TriG, RDF Dataset Language*. W3C Recommendation, 2014.

[24] Marion G Ceruti and Magdi N Kamel. Preprocessing and integration of data from multiple sources for knowledge discovery. *International Journal on Artificial Intelligence Tools*, 8(02):157–177, 1999.

[25] Ricardo Chalmeta and Verónica Pazos. A step-by-step methodology for enterprise interoperability projects. *Enterprise Information Systems*, 9(4):436–464, 2015.

[26] Yannis Charalabidis. *Revolutionizing Enterprise Interoperability through Scientific Foundations*. IGI Global, 2014.

[27] Erik Christensen, Francisco Curbera, Greg Meredith, Sanjiva Weerawarana, et al. Web services description language (WSDL) 1.1, 2001.

[28] OMG CORBA and IIOP Specification. Object management group. *Joint revised submission OMG document orbos/99-02*, 1999.

[29] Cyganiak Richard, Wood David, Lanthaler Markus, and Klyne Graham. RDF 1.1 Concepts and Abstract Syntax, 2014.

[30] Jérôme David, Jérôme Euzenat, François Scharffe, and Cássia Trojahn dos Santos. The alignment API 4.0. *Semantic web*, 2(1):3–10, 2011.

[31] Giuseppe De Giacomo, Domenico Lembo, Maurizio Lenzerini, Antonella Poggi, and Riccardo Rosati. Using ontologies for semantic data integration. In *A Comprehensive Guide Through the Italian Database Research Over the Last 25 Years*, pages 187–202. Springer, 2018.

[32] Mike Dean and Guus Schreiber. OWL web ontology language reference. W3C recommendation, W3C, February 2004. http://www.w3.org/TR/2004/REC-owl-ref-20040210/.

[33] Igor Dejanović, Maja Tumbas Živanov, Gordana Milosavljević, and Branko Perišić. Comparison of Textual and Visual Notations of DOMMLite Domain-Specific Language. In *Proceedings of the Advances in Databases and Information Systems*, pages 20–24, Novi Sad, 2010.

[34] Vladimir Dimitrieski. *Model-Driven Technical Space Integration Based on a Mapping Approach*. PhD thesis, University of Novi Sad, Faculty of Technical Sciences, 2018.

[35] M. Duerst and M. Suignard. RFC 3987: Internationalized Resource Identifiers (IRIs). RFC 3987 (Proposed Standard), see `http://www.ietf.org/rfc/rfc3987.txt`, January 2005.

[36] Moritz Eysholdt and Heiko Behrens. Xtext: implement your language faster than the quick and dirty way. In *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion*, pages 307–309. ACM, 2010.

[37] Daniel Faria, Catia Pesquita, Emanuel Santos, Isabel F Cruz, and Francisco M Couto. Agreementmakerlight: a scalable automated ontology matching system. In *Proceedings of the 10th International Conference on Data Integration in the Life Sciences (DILS 2014)*, pages 29–32, 2014.

[38] Matthias Felleisen. On the expressive power of programming languages. *Science of computer programming*, 17(1-3):35–75, 1991.

[39] Roy T Fielding and Richard N Taylor. Principled design of the modern web architecture. *ACM Transactions on Internet Technology (TOIT)*, 2(2):115–150, 2002.

[40] ACL Fipa. Fipa acl message structure specification. *Foundation for Intelligent Physical Agents, http://www. fipa. org/specs/fipa00061/SC00061G. html (30.6. 2004)*, 2002.

[41] Rafael Z Frantz and Rafael Corchuelo. A software development kit to implement integration solutions. In *Proceedings of the 27th Annual ACM Symposium on Applied Computing*, pages 1647–1652. ACM, 2012.

[42] Rafael Z Frantz, Antonia M Reina Quintero, and Rafael Corchuelo. A domain-specific language to design enterprise application integration solutions. *International Journal of Cooperative Information Systems*, 20(02):143–176, 2011.

[43] Michel Gagnon. Ontology-based integration of data sources. In *Information Fusion, 2007 10th International Conference on*, pages 1–8. IEEE, 2007.

[44] Avigdor Gal, Giovanni Modica, and Hasan Jamil. Ontobuilder: Fully automatic extraction and consolidation of ontologies from web sources. In *Data Engineering, 2004. Proceedings. 20th International Conference on*, page 853. IEEE, 2004.

[45] Maria Ganzha, Marcin Paprzycki, WiesŞaw PawŞowski, PaweŞ Szmeja, Katarzyna Wasielewska, and Giancarlo Fortino. Tools for ontology matching practical considerations.

[46] Nicola Gibson, Christopher P Holland, and Ben Light. Enterprise resource planning: a business approach to systems development. In *Systems Sciences, 1999. HICSS-32. Proceedings of the 32nd Annual Hawaii International Conference on*, pages 9–pp. IEEE, 1999.

[47] Fausto Giunchiglia, Aliaksandr Autayeu, and Juan Pane. S-match: an open source framework for matching lightweight ontologies. *Semantic Web*, 3(3):307–317, 2012.

[48] Cheng Hian Goh. *Representing and reasoning about semantic conflicts in heterogeneous information systems*. PhD thesis, Citeseer, 1996.

[49] Christine Golbreich. Combining rule and ontology reasoners for the semantic web. In *International Workshop on Rules and Rule Markup Languages for the Semantic Web*, pages 6–22. Springer, 2004.

[50] Beth Gold-Bernstein and William Ruh. *Enterprise integration: the essential guide to integration solutions*. Addison Wesley Longman Publishing Co., Inc., 2004.

[51] Asunción Gómez-Pérez and David Manzano-Macho. An overview of methods and tools for ontology learning from texts. *Knowl. Eng. Rev.*, 19(3):187–212, September 2004.

[52] Timo Greifenberg, Katrin Hölldobler, Carsten Kolassa, Markus Look, Pedram Mir Seyed Nazari, Klaus Mueller, Antonio Navarro Perez, Dimitri Plotnikov, Dirk Reiss, Alexander Roth, et al. A comparison of mechanisms for integrating handwritten and generated code for object-oriented programming languages. In *2015 3rd International Conference on Model-Driven Engineering and Software Development (MODELSWARD)*, pages 74–85. IEEE, 2015.

[53] TR Gruber. A translation approach to portable ontology specifications. *Knowledge acquisition*, 1993.

[54] Nicola Guarino. *Formal ontology in information systems: Proceedings of the first international conference (FOIS'98), June 6-8, Trento, Italy*, volume 46. IOS press, 1998.

[55] Ramanathan Guha and Dan Brickley. RDF schema 1.1. W3C recommendation, W3C, February 2014. http://www.w3.org/TR/2014/REC-rdf-schema-20140225/.

[56] Graham Hamilton, Rick Cattell, and Maydene Fisher. *JDBC database access with Java: a tutorial and annotated reference*. Addison-Wesley Longman Publishing Co., Inc., 1997.

[57] Steve Harris, Andy Seaborne, and Eric Prud'hommeaux. SPARQL 1.1 query language. *W3C recommendation*, 21(10), 2013.

[58] Hayes Patrick and Patel-Schneide Peter. RDF 1.1 Semantics, 2014.

[59] Wu He and Li Da Xu. Integration of distributed enterprise applications: a survey. *Industrial Informatics, IEEE Transactions on*, 10(1):35–42, 2014.

[60] Brian Henderson-Sellers. *Object-oriented metrics: measures of complexity*. Prentice-Hall, Inc., 1995.

[61] Michi Henning. The rise and fall of CORBA. *Queue*, 4(5):28–34, June 2006.

[62] Gregor Hohpe and Bobby Woolf. *Enterprise integration patterns: Designing, building, and deploying messaging solutions*. Addison-Wesley Professional, 2004.

[63] Matthew Horridge and Sean Bechhofer. The owl api: A java api for owl ontologies. *Semantic Web*, 2(1):11–21, 2011.

[64] Claus Ibsen and Jonathan Anstey. *Camel in Action*. Manning Publications Co., Greenwich, CT, USA, 1st edition, 2010.

[65] Željko Ivković, Renata Vaderna, Milorad Filipović, Gordana Milosavljevic, and Igor Dejanovic. Implementacija podloge za saradnju KROKI alata sa alatima za UML modelovanje opšte namene. In *YU INFO 2014 Zbornik radova*, pages 411–416. Društvo za informacione sisteme i računarske mreže, 01 2014.

[66] Saïd Izza. Integration of industrial information systems: from syntactic to semantic integration approaches. *Enterprise Information Systems*, 3(1):1–57, 2009.

[67] Saïd Izza, Lucien Vincent, and Patrick Burlat. A framework for semantic enterprise integration. In *Interoperability of enterprise software and applications*, pages 75–86. Springer, 2006.

[68] Ricardo Jardim-Goncalves, Carlos Coutinho, Adina Cretan, Catarina Ferreira da Silva, and Parisa Ghodous. Collaborative negotiation for ontology-driven enterprise businesses. *Computers in Industry*, 65(9):1232–1241, 2014.

[69] Apache Jena. A free and open source java framework for building semantic web and linked data applications. *Available online: jena. apache. org/(accessed on 28 April 2015)*, 2015.

[70] Ernesto Jiménez-Ruiz and Bernardo Cuenca Grau. Logmap: Logic-based and scalable ontology matching. In *International Semantic Web Conference*, pages 273–288. Springer, 2011.

[71] Vipul Kashyap and Amit Sheth. Semantic and schematic similarities between database objects: a context-based approach. *The VLDB Journal—The International Journal on Very Large Data Bases*, 5(4):276–304, 1996.

[72] Gregg Kellogg, Markus Lanthaler, and Manu Sporny. JSON-ld 1.0. W3C recommendation, W3C, January 2014. http://www.w3.org/TR/2014/REC-json-ld-20140116/.

[73] Steven Kelly and Juha-Pekka Tolvanen. *Domain-Specific Modeling: Enabling Full Code Generation*. Wiley, 2008.

[74] Michael Kifer, Georg Lausen, and James Wu. Logical foundations of object-oriented and frame-based languages. *Journal of the ACM (JACM)*, 42(4):741–843, 1995.

[75] Vitomir Kovanovic and Dragan Djuric. Highway: a domain specific language for enterprise application integration. In *Proceedings of the 5th India Software Engineering Conference*, pages 33–36. ACM, 2012.

[76] Ralf Kutsche, Nikola Milanovic, Gregor Bauhoff, Timo Baum, Mario Cartsburg, Daniel Kumpe, and Jürgen Widiker. BIZYCLE: Model-based interoperability platform for software and data integration. *Proceedings of the MDTPI at ECMDA*, 430, 2008.

[77] Tom Laszewski and Prakash Nauduri. *Migrating to the cloud: Oracle client/server modernization*. Elsevier, 2011.

[78] Daniel Le Berre and Pascal Rapicault. Dependency management for the eclipse ecosystem: eclipse p2, metadata and resolution. In *Proceedings of the 1st international workshop on Open component ecosystems*, pages 21–30. ACM, 2009.

[79] Berners T. Lee, L. Masinter, and M. Mccahill. RFC 1738: Uniform resource locator (URL). http://www.ietf.org/rfc/rfc1738.txt, 1994.

[80] Andreas Leicher, Susanne Busse, and Jörn Guy Süß. Analysis of compositional conflicts in component-based systems. In *Software Composition*, pages 67–82. Springer, 2005.

[81] Apprenda Library. Types of middleware. `https://apprenda.com/library/architecture/types-of-middleware/`. Dobavljeno: 19.10.2018.

[82] Ivan Luković. *Integracija šema modula baze podataka informacionog sistema*. PhD thesis, University of Novi Sad, Faculty of Technical Sciences, 1996.

[83] Björn Lundell, Brian Lings, Anna Persson, and Anders Mattsson. UML model interchange in heterogeneous tool environments: an analysis of adoptions of XMI 2. In *International Conference on Model Driven Engineering Languages and Systems*, pages 619–630. Springer, 2006.

[84] Tim A Majchrzak, Tobias Jansen, and Herbert Kuchen. Efficiency evaluation of open source ETL tools. In *Proceedings of the 2011 ACM Symposium on Applied Computing*, pages 287–294. ACM, 2011.

[85] A Mathur, Robert W Hall, Farnam Jahanian, Atul Prakash, and Craig Rasmussen. The publish/subscribe paradigm for scalable group collaboration systems. *Ann Arbor*, 1001(313):48109, 1995.

[86] Richard J Mayer, Christopher P Menzel, Michael K Painter, Paula S Dewitte, Thomas Blinn, and Benjamin Perakath. Information integration for concurrent engineering (IICE) IDEF3 process description capture method report. Technical report, KNOWLEDGE BASED SYSTEMS INC COLLEGE STATION TX, 1995.

[87] Jeff McAffer, Jean-Michel Lemieux, and Chris Aniszczyk. *Eclipse rich client platform*. Addison-Wesley Professional, 2010.

[88] Thomas J McCabe. A complexity measure. *IEEE Transactions on software Engineering*, (4):308–320, 1976.

[89] Deborah McGuinness and Christopher Welty. OWL web ontology language guide. W3C recommendation, W3C, February 2004. http://www.w3.org/TR/2004/REC-owl-guide-20040210/.

[90] Tom Mens and Pieter Van Gorp. A taxonomy of model transformation. *Electronic Notes in Theoretical Computer Science*, 152:125–142, 2006.

[91] Bertrand Meyer. UML: The positive spin. *American Programmer*, 1997.

[92] Nikola Milanovic, Ralf Kutsche, Timo Baum, Mario Cartsburg, Hatice Elmasgünes, Marco Pohl, and Jürgen Widiker. Model&metamodel, metadata and document repository for software and data integration. In *International Conference on Model Driven Engineering Languages and Systems*, pages 416–430. Springer, 2008.

[93] George A Miller. Wordnet: a lexical database for english. *Communications of the ACM*, 38(11):39–41, 1995.

[94] Richard Millham. Integrating heterogeneous data for big data analysis. *Handbook of Research on Cloud Infrastructures for Big Data Analytics*, page 263, 2014.

[95] Tadao Murata. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77(4):541–580, 1989.

[96] Judith M Myerson. *Enterprise systems integration*. CRC Press, 2001.

[97] Meenakshi Nagarajan, Kunal Verma, Amit P Sheth, and John A Miller. Ontology driven data mediation in web services. *International Journal of Web Services Research (IJWSR)*, 4(4):104–126, 2007.

[98] Channah F Naiman and Arison M Ouksel. A classification of semantic conflicts in heterogeneous database systems. *Journal of Organizational Computing and Electronic Commerce*, 5(2):167–193, 1995.

[99] Henrik Frystyk Nielsen, Marc Hadley, Jean-Jacques Moreau, Martin Gudgin, Anish Karmarkar, Noah Mendelsohn, and Yves Lafon. SOAP version 1.2 part 1: Messaging framework (second edition). W3C recommendation, W3C, April 2007. http://www.w3.org/TR/2007/REC-soap12-part1-20070427/.

[100] Steve Northover and Mike Wilson. *Swt: the standard widget toolkit, volume 1*. Addison-Wesley Professional, 2004.

[101] Mandi Ohlinger, Kent Sharkey, and Saisang Cai. Microsoft biztalk runtime architecture. 2017.

[102] Mandi Ohlinger, Kent Sharkey, and Saisang Cai. XLANG-s language. 2017.

[103] OMG. *XML Metadata Interchange (XMI)*. OMG, 2015.

[104] OMG. *Meta Object Facility (MOF) 2.5.1*. OMG, 2016.

[105] OMG. *Unified Modeling Language (UML) 2.5.1*. OMG, 2017.

[106] Zoltan Papp and George Exarchakos. *Runtime Reconfiguration in Networked Embedded Systems*. Springer, 2016.

[107] Jinsoo Park and Sudha Ram. Information systems interoperability: What lies beneath? *ACM Transactions on Information Systems (TOIS)*, 22(4):595–632, 2004.

[108] Bijan Parsia and Evren Sirin. Pellet: An owl dl reasoner. In *Third International Semantic Web Conference-Poster*, volume 18, 2004.

[109] XML Schema Part. 1: Structures, 2001.

[110] XML Schema Part. 2: Datatypes, 2001.

[111] Antonella Poggi, Domenico Lembo, Diego Calvanese, Giuseppe De Giacomo, Maurizio Lenzerini, and Riccardo Rosati. Linking data to ontologies. In *Journal on data semantics X*, pages 133–173. Springer, 2008.

[112] Sudha Ram and Jinsoo Park. Semantic conflict resolution ontology (SCROL): An ontology for detecting and resolving data and schema-level semantic conflicts. *Knowledge and Data Engineering, IEEE Transactions on*, 16(2):189–202, 2004.

[113] Steven P Reiss. Connecting tools using message passing in the field environment. *IEEE software*, (4):57–66, 1990.

[114] Forrester Research. Reducing integration costs, 2001.

[115] Arthur Alexander Reyes, José R Espino, Vijai Mohan, and Monica Nadkar. Ad hoc software interfacing: enterprise application integration (EAI) when middleware is overkill. In *Computer Software and Applications Conference, 2003. COMPSAC 2003. Proceedings. 27th Annual International*, pages 570–575. IEEE, 2003.

[116] Gérard Ribière. *Communication et traitement en mode message avec MQSeries*. Ed. Techniques Ingénieur, 1997.

[117] Fabricia Roos-Frantz, Manuel Binelo, Rafael Z Frantz, Sandro Sawicki, and Vítor Basto Fernandes. Using petri nets to enable the simulation of application integration solutions conceptual models. In *ICEIS (1)*, pages 87–96, 2015.

[118] Walt Scacchi. Process models in software engineering. *Encyclopedia of software engineering*, 2002.

[119] Douglas C Schmidt. Model-driven engineering. *COMPUTER-IEEE COMPUTER SOCIETY-*, 39(2):25, 2006.

[120] Maria Grazia Scutella. A note on dowling and gallier's top-down algorithm for propositional horn satisfiability. *The Journal of Logic Programming*, 8(3):265–273, 1990.

[121] Bran Selic. What will it take? a view on adoption of model-based methods in practice. *Software & Systems Modeling*, 11(4):513–526, 2012.

[122] Yakov Shafranovich. Common Format and MIME Type for Comma-Separated Values (CSV) Files. RFC 4180, October 2005.

[123] Ronald Sheffer, Mark Morsch, and Michelle Wieczorek. Automated clinical indicator recognition with natural language processing, March 5 2015. US Patent App. 14/478,454.

[124] John Shirley. *Guide to writing DCE applications*. O'Reilly & Associates, Inc., 1992.

[125] Michael Shtelma, Mario Cartsburg, and Nikola Milanovic. Executable domain specific language for message-based system integration. In *Model Driven Engineering Languages and Systems*, pages 622–626. Springer, 2009.

[126] Jon Siegel and Dan Frantz. *CORBA 3 fundamentals and programming*, volume 2. John Wiley & Sons New York, NY, USA:, 2000.

[127] Evren Sirin, Bijan Parsia, Bernardo Cuenca Grau, Aditya Kalyanpur, and Yarden Katz. Pellet: A practical owl-dl reasoner. *Web Semantics: science, services and agents on the World Wide Web*, 5(2):51–53, 2007.

[128] Hassan A Sleiman, Abdul W Sultán, Rafael Z Frantz, Rafael Corchuelo, et al. Towards automatic code generation for EAI solutions using DSL tools. In *JISBD*, pages 134–145, 2009.

[129] Mark Stefik and Daniel G Bobrow. Object-oriented programming: Themes and variations. *AI magazine*, 6(4):40, 1985.

[130] Suzette Stoutenburg, Leo Obrst, Deborah Nichols, Paul Franklin, Ken Samuel, and Michael Prausa. Ontologies in OWL for rapid enterprise integration. *issues*, 6:7, 2007.

[131] Marinos Themistocleous, Zahir Irani, Robert M O'Keefe, and Ray Paul. ERP problems and application integration issues: An empirical survey. In *System Sciences, 2001. Proceedings of the 34th Annual Hawaii International Conference on*, pages 10–pp. IEEE, 2001.

[132] Renata Vaderna. *Algoritmi i jezik za podršku automatskom raspoređivanju elemenata dijagrama*. PhD thesis, University of Novi Sad, Faculty of Technical Sciences, 2018.

[133] Arie Van Deursen, Paul Klint, and Joost Visser. Domain-specific languages: An annotated bibliography. *ACM Sigplan Notices*, 35(6):26–36, 2000.

[134] Panos Vassiliadis. A survey of extract–transform–load technology. *International Journal of Data Warehousing and Mining (IJDWM)*, 5(3):1–27, 2009.

[135] Steve Vinoski. CORBA: integrating diverse applications within distributed heterogeneous environments. *IEEE Communications magazine*, 35(2):46–55, 1997.

[136] Markus Voelter, Sebastian Benz, Christian Dietrich, Birgit Engelmann, Mats Helander, Lennart C. L. Kats, Eelco Visser, and Guido Wachsmuth. *DSL Engineering - Designing, Implementing and Using Domain-Specific Languages*. dslbook.org, 2013.

[137] Julius Volz, Christian Bizer, Martin Gaedke, and Georgi Kobilarov. Silk-a link discovery framework for the web of data. *LDOW*, 538, 2009.

[138] Željko Vuković, Nikola Milanović, Renata Vaderna, Igor Dejanović, and Gordana Milosavljević. SAIL: A domain-specific language for semantic-aided automation of interface mapping in enterprise integration. In Ioana Ciuciu, Hervé Panetto, Christophe Debruyne, Alexis Aubry, Peter Bollen, Rafael Valencia-García, Alok Mishra, Anna Fensel, and Fernando Ferri, editors, *On the Move to Meaningful Internet Systems: OTM 2015 Workshops*, volume 9416 of *Lecture Notes in Computer Science*, pages 97–106. Springer International Publishing, 2015.

[139] Željko Vuković, Nikola Milanović, and Gregor Bauhoff. Prototype of a framework for ontology-aided semantic conflict resolution in enterprise integration. In *ICIST 2015*, pages 257–260. Society for Information Systems and Computer Networks, 2015.

[140] Željko Vuković, Nikola Milanović, Renata Vaderna, Igor Dejanović, Gordana Milosavljević, and Vuk Malbaša. Semantic-aided automation of interface mapping in enterprise integration with conflict detection. *Information Systems and e-Business Management*, 15(2):305–322, 2017.