Radni okvir za vizualno modeliranje i formalnu verifikaciju izvršavanja poslovnoga procesa temeljen na modelima s dvosmjernim preslikavanjem

Matković, Jelena

Sveučilište u Zagrebu

FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

JELENA MATKOVIĆ

Radni okvir za vizualno modeliranje i formalnu verifikaciju izvršavanja poslovnoga procesa temeljen na modelima s dvosmjernim preslikavanjem

DOKTORSKI RAD

Mentor: Prof. dr. sc. Krešimir Fertalj

Zagreb, 2019.

University of Zagreb

FACULTY OF ELECTRICAL ENGINEERING AND COMPUTING

Jelena Matković

Framework for visual modelling and formal verification of business process execution based on two-way mapping models

DOCTORAL THESIS

Professor Krešimir Fertalj, PhD

Zagreb, 2019.

Doktorski rad izrađen je na Sveučilištu u Zagrebu Fakultetu elektrotehnike i računarstva, na Zavodu za primijenjeno računarstvo.

Mentor: Prof. dr. sc. Krešimir Fertalj

Doktorski rad ima: 133 stranice

Doktorski rad br.:_____

O mentoru

Krešimir Fertalj je redoviti profesor na Zavodu za primijenjeno računarstvo Fakulteta elektrotehnike i računarstva Sveučilišta u Zagrebu gdje predaje nekoliko računarskih kolegija na preddiplomskom, diplomskom i poslijediplomskom doktorskom studiju. Diplomirao je, magistrirao i stekao doktorat iz Računarstva na istom fakultetu. Njegov stručni i znanstveni interes je u području automatiziranog programskog inženjerstva, složenih informacijskih sustava i upravljanja projektima. Vodio je dva znanstvena projekta Ministarstva znanosti, obrazovanja i sporta Republike Hrvatske te više desetaka istraživačkih projekata za gospodarstvo. Objavio je više od 150 znanstvenih i stručnih radova. Bio je mentor na više od 140 diplomskih radova te sedam doktorskih disertacija.

Prof. Fertalj član je stručnih udruga ACM i IEEE. Suradnik je Akademije tehničkih znanosti Hrvatske (HATZ), gdje je obnašao dužnost tajnika Odjela informacijskih sustava od 2009. do 2013. godine. Dobio je Medalju Personalne uprave Ministarstva obrane Republike Hrvatske.

About the Supervisor

Krešimir Fertalj is a full professor at the Department of Applied Computing at the Faculty of Electrical Engineering and Computing, University of Zagreb and lectures a couple of computing courses on undergraduate, graduate and doctoral studies. He graduated and received his PhD degree in Computing at the same Faculty. His professional and scientific interest is in computer-aided software engineering, complex information systems and in project management. He led two scientific projects for the Ministry of Science, Education and Sports of the Republic of Croatia and dozens of research projects with the industry. He has published more than 150 scientific and technical papers. He was a mentor to students for more than 140 graduate theses and seven PhD dissertations.

Prof. Fertalj is a member of professional associations ACM and IEEE. He is associate of the Croatian Academy of Engineering (HATZ), where he served as secretary of the Department of Information Systems from 2009 to 2013. He has received a Medal of Human Resources Administration of the Croatian Ministry of Defence.

Ponajprije zahvaljujem svome mentoru profesoru Krešimiru Fertalju na ogromnoj podršci, ohrabrenju i pomoći tijekom izrade doktorskog rada. Također želim zahvaliti i svojoj obitelji na svoj podršci i ohrabrenju koju mi je pružila u ključnim trenutcima pisanja doktorskog rada.

Sažetak

Web Service Business Process Execution Language (WS-BPEL) je dominantan standard za integraciju postojećih web usluga u cjelinu kojom se podupire izvršavanje poslovnog procesa. Cilj integracije nije samo omogućiti razmjenu poruka između učesnika poslovnog procesa, nego razmjenu poruka realizirati tako da se ona izvršava točno u trenutcima i na način kako bi se odvijali koraci stvarnog poslovnog procesa. WS-BPEL koristi na XML-u zasnovan jezik koji podržava web tehnologije. Budući da WS-BPEL specifikacija ne određuje službeni standard za vizualno modeliranje procesa, javila se ideja za razvojem vlastitog modela za vizualno modeliranje procesa. Osim vizualnog modeliranja, kao potrebna pokazala se formalna verifikacija izvršavanja budući da se unutar WS-BPEL procesa mogu pojaviti paralelna izvršavanja čime ponašanje sustava zbog brzine paralelnih komponenti postaje potencijalno nesigurno. Ovaj doktorski rad razradio je korake modela faznog modeliranja WS-BPEL poslovnog procesa na način da su ključne faze razvoja WS-BPEL procesa vezane za vizualno modeliranje i formalnu verifikaciju, a tu je i faza statičke provjere izvršnog modela procesa prema pravilima definiranim WS-BPEL specifikacijom. Za vizualno modeliranje predložen je standard Business Process Model and Notation (BPMN) koji je u pozadini također zasnovan na jeziku XML. Definiran je algoritam za dvosmjerno automatsko preslikavanje između standarda BPMN i WS-BPEL. Za prikazivanje formalnog modela odabrana je sintaksa Process Meta Language (*Promela*). Cilj formalne verifikacije je provjera rukovanja varijablama kao ključnim elementima poslovnog procesa. Osim predloženog faznog modela za razvoj WS-BPEL procesa i pripadajućih algoritama, u doktorskom radu je definiran i radni okvir s prijedlozima konkretnih alata koji su korišteni za realizaciju modela i pripadnih algoritama. Verifikacija algoritama korištenih u modelu i radnom okviru je izvršena nad hipotetičkim poslovnim procesima.

Ključne riječi: uslužno orijentirane arhitekture, WS-BPEL, integracija usluga, faze razvoja poslovnog procesa, vizualno modeliranje, formalna verifikacija sustava, BPMN, konačni automati.

Extended abstract

FRAMEWORK FOR VISUAL MODELLING AND FORMAL VERIFICATION OF BUSINESS PROCESS EXECUTION BASED ON TWO-WAY MAPPING MODELS

Service-oriented architectures (SOA) represent the evolution of software development which has evolved after maturity of object-oriented architectures. Object-oriented architectures have introduced objects as reusable pieces of code while service-oriented architectures have moved one step further in the process of software reuse and have introduced possibilities of application reuse. Applications expose their functionalities as services in the standardized way (WSDL interfaces) and they can be integrated into more complex functionalities, i.e. business processes. Web Service Business Process Execution Language (WS-BPEL) is a predominant standard used for the integration of existing web services into executions used for the automation of business processes. WS-BPEL syntax is XML based and as already stated WS-BPEL processes communicate with WSDL services using SOAP protocol for communication. Services integrated into WS-BPEL processes are called partners and they are called when necessary. Process executions are similar to flow charts where partners are called when necessary. WS-BPEL standard offers a rich set of constructs, like conditional executions, event-driven executions, error handlings, compensation executions, and so on thus contained process behavior can become extremely complex.

Since the WS-BPEL standard is based on XML syntax and no official standards nor approaches are offered for visual modeling of WS-BPEL processes, this resulted in an idea to develop a proprietary model that can be used for visual modeling of WS-BPEL processes which is one of the scientific contributions of the paper.

Constructs offered by WS-BPEL specification can produce processes containing parallel executions which is a possible uncertainty in process execution because of the unpredictable execution speed of parallel components. The possibility of parallelism inside of WS-BPEL processes resulted in a need to develop formal methods that can check all possible executions of WS-BPEL processes. Formal checking presented in this dissertation is focused on the detection of parallel manipulation of process variables.

WS-BPEL specification has defined a set of specific rules that must be verified by static analysis of the process before the process is deployed. This resulted in a need to develop an

algorithm to automate static analysis of WS-BPEL processes which is the third algorithm of the scientific contribution of this dissertation.

Algorithms and phases of WS-BPEL process development (visual modeling, formal verification, and static analysis) have resulted in the need to develop a model for phase development of WS-BPEL processes and a corresponding proposal for an implementation framework. Identified needs: algorithms for visual modeling, formal verification, and static analysis, as well as the model for phased development of WS-BPEL processes and its implementation framework, represent the scientific contributions of this dissertation.

WS-BPEL standard is presented in chapter 2 where WS-BPEL offered constructs and related standards are introduced. In chapter 3 phased model for the development of WS-BPEL processes as one of the scientific contributions, is presented. Phases of the model integrate all the algorithms (visual modeling, formal verification, and static analysis) into the coherent whole. The first phase of the model is related to the visual development of the WS-BPEL process. Visual model, as output of the first phase, is transformed into WS-BPEL code and brought into the second phase where manual upgrade of the resulting WS-BPEL code is performed to make it executable. The third phase of the model is related to performing formal verification of the WS-BPEL code to find parallel variable manipulations or readings of uninitialized variables. If violations are found, correction of the WS-BPEL code is performed as a part of the fourth phase and the same cyclic process of formal verification and correction is performed until the resulting WS-BPEL code is satisfactory. The fifth and possibly the final phase of the model is related to the static analysis of the WS-BPEL code. Changed WS-BPEL code from any phase can be transformed into the visual model if changes affect the visual representation of the WS-BPEL process. Phases of the model are interconnected so the development of WS-BPEL processes is a complete process and it was the intention of the scientific contribution related to the model. When considering phased model development, it is assumed that the WS-BPEL process is made up of the existing web services, i.e. determining functionalities that will be exposed as web services is not part of the proposed phased model. It is a prerequisite that the services already exist before their integration into WS-BPEL processes is started.

The first phase of the model presented in chapter 3 is related to visual modeling of processes while chapter 4 of the dissertation introduces standard used for visual modeling and bidirectional algorithm for transformation between visual and execution WS-BPEL model.

Business Process Model and Notation (BPMN) standard is used for visual modeling of WS-BPEL processes because of its relation to business processes in general and because of its high level of standardization and a wide selection of offered BPMN tools. As part of chapter 4, BPMN executions (named as visual activities) are paired up with the corresponding WS-BPEL activities to show how they are visually presented in BPMN graphs. Since the BPMN standard is XML based as well as WS-BPEL standard, XML background code of visual activities and the XML code of the corresponding WS-BPEL activities are paired up. Functions for bidirectional transformations between XML BPMN graphs and XML WS-BPEL graphs are introduced. There is a gap between mechanisms of connecting elements inside of BPMN and WS-BPEL graphs. Sequence flows are used to connect sequential BPMN elements whilst WS-BPEL graphs are block-based. The difference between connection mechanisms was a challenge that had to be overridden when constructing bidirectional functions for transformations of sequence BPMN elements into populated WS-BPEL sequence activity and vice versa. Another challenge was the transformation of nested visual activities. Nesting to an arbitrary depth had to be supported by bidirectional transformation algorithms. BPEL flow activity was also a challenge since it can contain complex synchronized behavior. A proposal for visual modeling of BPEL flow activity was made but the transformation to XML code is left to be manual. The automatic bidirectional transformation of BPEL flow activity can be considered as a task for future work.

Chapter 5 has introduced approaches for automatic static analysis of WS-BPEL code if chosen categories of rules are satisfied in the process. Automatic analysis can improve the development of WS-BPEL processes since its XML structure can be quite complex and manual static analysis can be error-prone. A static analysis of WS-BPEL processes is performed in the final phase of the proposed phased model for the development of WS-BPEL processes presented in chapter 3.

Chapter 6 has introduced steps for the formal checking of WS-BPEL processes. Promela is used as a syntax for presenting formal models expressed as finite-state automata. Elements of WS-BPEL processes included in formal models are all parallel executions inside of processes as well as all behavior that performs variable manipulations. To generate Promela's formal models more efficiently, the generation of symbolic models based on source WS-BPEL processes is proposed. Symbolic models serve as transitions towards a successful generation of final formal models and they systematically list all elements of the WS-BPEL process (and their corresponding details) that are relevant for formal verification to transform them into the formal

model according to the predefined rules. Formal verification presented in this dissertation inspects if there are parallel variable writings and readings or attempts of using uninitialized or variables of inadequate types. If there are such violations in executions of the formal model, they are easily located in the symbolic model which helps to locate them more easily in the WS-BPEL process to correct its behavior. A scientific basis upon which formal verification of Promela code is performed is s synchronous product of finite-state automata. Checked WS-BPEL code, expressed as Promela code, is one finite-state automata (formed as an asynchronous interleaving product of parallel components of WS-BPEL process) and it is the first input into the synchronous product upon which verification is performed. The second input to the synchronous product is a negation of wanted behavior. Violations are found if there is a common behavior of two input automata. Scientific basis upon which formal verification is performed does not represent scientific contribution since it is something already known, but the scientific contributions are the identification of process elements relevant to the formal verification, forming a symbolic model as a transition towards the formal model, forming of the formal model, and expressing wanted/unwanted behavior. Formal verification is performed in the third phase of the proposed phased model for the development of WS-BPEL processes presented in chapter 3.

Chapter 7 has introduced the implementation framework of the proposed phased model for the development of WS-BPEL processes presented in chapter 3. Tools for the implementation of proposed algorithms for visual modeling, formal verification, and static analysis are introduced as part of the implementation framework. It is concluded that any visual BPMN modeling tool providing access to the background XML code can be used for visual modeling of WS-BPEL processes since BPMN XML code is an input to the BPMN-BPEL transformation algorithm. Language-Integrated Query for C# (LINQ for C#) is used as a technology for the implementation of a bidirectional algorithm between BPMN and BPEL models, even though it is concluded that any programming language with constructs for XML document manipulation can be used for the implementation of bidirectional BPMN-BPEL transformation algorithm. eXtensible Stylesheet Language Transformations technology (XSLT) is also listed as a possible technology that can be used for the implementation of the BPMN-BPEL transformation algorithm but it is concluded that it is relatively complex when comparing it to LINQ for C# or any similar technology. The spin tool is used for the implementation of formal checking. Formal models, written in Promela syntax can be made using any textual tool. They are inputs to the formal verification performed by the Spin tool.

Spin is the only alternative to use in the process of formal verification since it is a tool specially designed to operate with Promela syntax. Scientific contribution is the implementation of a bidirectional BPMN-BPEL transformation algorithm using LINQ for C# technology while operating with BPMN visual editors (for visual modeling) and Spin (for formal verification) is not since they are not proprietary tools. Analysis and proposals for possible tools usage also represent scientific contributions.

Chapter 8 has tested the algorithms for the development of WS-BPEL processes presented in chapter 3. The BPMN-BPEL transformation algorithm was verified on an abstract WS-BPEL process as part of this chapter. Contained execution of an abstract BPMN process was transformed into corresponding WS-BPEL activities based on XML mappings defined in chapter 4. Formal verification was also verified on an abstract WS-BPEL process and it has detected all parallel variable manipulations that were present in the process. The symbolic model of the abstract WS-BPEL process was constructed according to the rules defined in chapter 6 towards the formal model upon which formal verification was performed.

Scientific contributions of the dissertation with its proposed model for the phased development of WS-BPEL processes can be seen as a basis for future work to optimize it or to include other phases of development like design of services which would include: collecting user requests, identification of functionalities aggregated in services, and building services. The current model starts with the first phase of visual modeling where it is assumed that integration services already exist and their construction is not covered by any phase the model.

Keywords: service-oriented architectures, WS-BPEL, web service integration, development phases of business processes, visual modelling, formal verification, BPMN, finite state automata.

Sadržaj

1. Uvod

Uslužno orijentirane arhitekture (SOA, engl. *service-oriented architectures*) [1] s uvođenjem usluga i njihovog reiskorištavanja, predstavljaju evoluciju razvoja nakon objektno orijentiranih arhitektura koje su uvele objekte i njihovo višestruko izvršavanje. Može se reći da su usluge svojevrsna evolucija objekata, gdje ne samo dijelovi programskog koda, nego cijele aplikacije postaju predmet ponude, iskorištavanja i integracije. Prema [2] ponovna upotrebljivost (engl. *reusability*) usluga je jedna od glavnih dobiti uslužno orijentiranih arhitektura. Prema [3] s uslužno orijentiranim arhitekturama napravljen je pomak od sustava kojima su u fokusu ljudi prema sustavima kojima su u fokusu aplikacije što podrazumijeva da se komunikacija između aplikacija može odvijati direktno poput komunikacije između web preglednika i poslužitelja. Budući da su uslužno orijentirane arhitekture uvele nove paradigme i ponašanja, *SOA Reference Model* [4] definirao je referentni model uslužno orijentiranih arhitektura definiravši rječnik osnovnih pojmova koji se provlače kroz ovaj tip arhitektura, dok je *OASIS Reference Architecture* [5] definirala osnovne pojmove vezane za SOA ekosistem, SOA arhitekturu i implementaciju.

Pojava usluga, kao ponuđenih jedinica za integraciju u veće jedinice, odnosno u poslovne procese, sa sobom je uvela standard Web Service Description Languge Standard (WS-BPEL, u daljnjem tekstu BPEL). BPEL je standard, razvijen od strane OASIS grupe [6], a baziran na jeziku XML, što podrazumijeva da su sve njegove konstrukcije opisane vlastitom XML *Schemom* [7]. Postoje brojna okruženja za izvršavanje BPEL koda [8], [9], [10], to jest okruženja koja interpretiraju njegovu XML sintaksu i izvršavaju je (engl. *BPEL engines*). BPEL je standard namijenjen integraciji web usluga (koje će se zvati partnerima) u poslovni proces. Standard oslikava poslovni proces pozivanjem partnera kada je to potrebno, dok je samo izvršavanje procesa između poziva partnera najsličnije dijagramu toka izvršavanja sa sljedećim ponuđenim tipovima izvršavanja: uvjetna izvršavanja, izvršavanja aktivirana događajem, izvršavanja u slučaju pojave predefinirane pogreške, alternativna izvršavanja u slučaju potrebe poništavanja prethodno obavljenog izvršavanja i slično. Detaljniji pregled ponuđenih konstrukcija od strane BPEL standarda objašnjen je u poglavljima 2 i 4.

1.1. Motivacija

Budući da je namjena BPEL standarda integracija partnera u poslovni proces i budući da nerijetko u tome učestvuju partneri koji su vlasnici poslovnih procesa, često s nižom razinom

tehničkog znanja, potrebno je pronaći način da se njima približi integracija partnera u poslovni proces, a to se može postići preciznom vizualizacijom modeliranja poslovnog procesa. Za vizualnu izgradnju BPEL procesa u ovom radu predložen je standard *Business Process Model and Notation* (BPMN) [11] koji je široko prihvaćen upravo u području vizualnog oblikovanja procesa. BPMN posjeduje standardizirane vizualne konstrukcije koje u pozadini imaju svoje XML ekvivalente pa je ideja procese modelirati vizualno, a pozadinski BPEL kod generirati XML transformacijom između BPMN i BPEL.

Druga poteškoća u procesu modeliranja BPEL procesa vezana je za verifikaciju ispravnosti izvršavanja BPEL procesa jer neki proces može posjedovati paralelna asinkrona izvršavanja čime se smješta u domenu konkurentnih sustava. Konkurentni sustavi zbog asinkronog paralelnog izvršavanja mogu posjedovati potencijalno opasna, odnosno nepredvidiva ponašanja, koja za poslovni proces mogu biti važna, a koja se mogu otkriti tek slučajnim izvršavanjem instance poslovnog procesa [12]. Zbog toga se pojavila potreba za formalnom verifikacijom izvršavanja s ciljem preventivnog pronalaska potencijalno opasnih ponašanja prije nego se poslovni proces stavi u produkciju i slučajno otkrije njegovo neželjeno ponašanje. Formalna verifikacija prolazi kroz cijeli prostor stanja sustava, tj. kroz sva njegova izvršavanja čime se neželjeno ponašanje može otkriti prije njegovog postavljanja u realno okruženje.

Treća poteškoća prilikom modeliranja BPEL procesa vezana je za obvezu poštivanja pravila definiranih od strane BPEL specifikacije prilikom modeliranja BPEL procesa, a koja se prema specifikaciji moraju provjeriti statičkom analizom prilikom modeliranja. Neka od pravila sistemske su prirode i namijenjena otkrivanju od strane BPEL izvršnih okruženja i neće biti uvrštena u statičku provjeru, a biti će uvrštena sva ostala koja nisu te prirode. Korisnost ovoga algoritma leži u tome da uvrštena pravila, koja će se statički provjeravati, postaju neovisna o provjeri od strane izvršnog okruženja, to jest algoritmom se mogu provjeriti prije nego se proces postavi u bilo koje izvršno okruženje čime se postiže neovisnost od izvršnog okruženja.

Svi navedeni algoritmi pripadaju različitim fazama razvoja BPEL procesa i potrebno ih je uklopiti u jednu cjelinu temeljem čega se javlja i potreba za preciznim definiranjem modela za razvoj BPEL procesa koji bi obuhvatio sve faze kroz koje se izvode gornji algoritmi.

Gornje identificirane poteškoće i potrebe su bile motiv za definiranje teme doktorske disertacije i znanstvenih doprinosa.

1.2. Znanstveni doprinos

Cilj doktorskog rada jeste detaljno proučiti BPEL standard i proces modeliranja poslovnog procesa BPEL standardom te ga uklopiti u predloženi fazni model kroz koji će se provući algoritmi za vizualno modeliranje, formalnu verifikaciju i statičku provjeru pravila.

Očekivani znanstveni doprinos rada sadržan je u sljedećem:

- Model koji omogućuje fazni razvoj poslovnog procesa prikazanog u WS-BPEL notaciji, počevši od oslikavanja tijeka izvršavanja poslovnog procesa do formalne provjere.
- Algoritmi dvosmjernog preslikavanja između WS-BPEL koda i odabranih formalnih specifikacija za vizualno modeliranje i verifikaciju te algoritam za provjeru usklađenosti WS-BPEL koda naspram pravila definiranih WS-BPEL specifikacijom.
- Radni okvir kojim se razvoj WS-BPEL poslovnog procesa implementira kroz faze definirane predloženim modelom i korištenjem predloženih algoritama.
- Prototip za verifikaciju predloženog modela, algoritama i okvira nad reprezentativnim hipotetičkim poslovnim procesima.

1.3. Struktura rada

Rad je strukturiran kroz sljedeća poglavlja. Drugo poglavlje prikazuje osnovne konstrukcije definirane BPEL standardom, njegovu XML strukturu, namjenu standarda i s njim povezane standarde. U trećem poglavlju je predstavljen model za fazni razvoj poslovnih procesa u sklopu čijeg uvodnog dijela je prikazan i osvrt na rad drugih autora iz područja modeliranja poslovnih procesa BPEL-om i njihove formalne verifikacije. U četvrtom poglavlju je predložen algoritam dvosmjernog preslikavanja između BPMN i BPEL (u daljnjem tekstu algoritam preslikavanja BPMN-BPEL), u sklopu kojega su predstavljene i XML konstrukcije korištenih standarda budući da predloženi algoritam funkcionira na XML razini. U petom poglavlju je opisan algoritam za statičku provjeru BPEL procesa naspram pravila definiranih BPEL specifikacijom. Šesto poglavlje opisuje teorijsku osnovu formalne verifikacije na kojoj se zasniva formalna provjera BPEL procesa nakon čega slijedi prikaz predloženog algoritma za formalnu verifikaciju, to jest prikazano je što će se provjeravati, koje su konstrukcije bitne za formalnu provjeru te kako se iz BPEL procesa dolazi do formalnog modela. Sedmo poglavlje predstavlja radni okvir kojim je implementiran model za fazni razvoj BPEL procesa. Osmo poglavlje verificira algoritme korištene u modelu i radnom okviru za fazni razvoj procesa. Rad

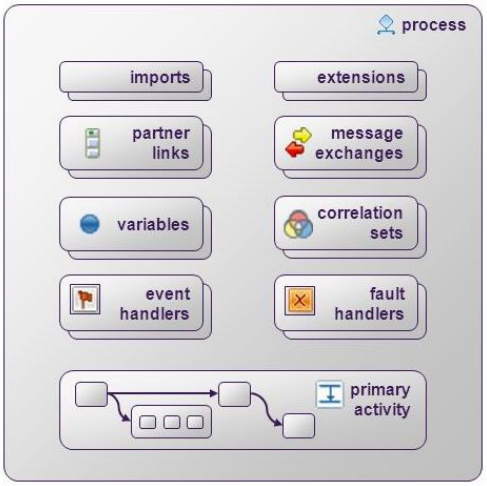
završava kritičkim osvrtom na rezultate doktorskog rada, prijedlogom budućih aktivnosti te završnim pregledom rada.

2. Standard BPEL

2.1. Osnovno o standardu

Standard BPEL je, kao što je to već u uvodnom poglavlju rečeno, namijenjen integraciji Web usluga (partnera) u poslovni proces. Funkcionalnost integracije partnera/usluga spada u područje uslužno orijentiranih arhitektura, koje se definiraju kao arhitekture namijenjene udovoljenju zahtjeva raspodijeljenog računarstva čiji je cilj integrirati labavo povezane (engl. *loosely coupled*), standardizirane (engl. *standardized)* i o protokolu implementacije neovisne (engl. *implementation independent*) komponente [13]. Kako je u [13] navedeno, ovakve komponente obično komuniciraju međusobnim pozivima, najčešće na način aktiviran događajem ili asinkrono s ciljem integracije u poslovni proces.

Upravo je BPEL standard jedan iz domene standarda čiji je cilj integracija partnerskih usluga u poslovni proces na prethodno spomenuti način. Partneri su usluge koje mogu biti implementirane u bilo kojoj tehnologiji dokle god su njihova sučelja prikazana na standardiziran način korištenjem Web Service Description Language (WSDL) standarda [14] s kojim BPEL proces zna komunicirati. WSDL sučelja izlažu metode usluga koje se mogu pozvati, a poruke se šalju kodirane Simple Object Access Protocol (SOAP) protokolom [15] (*SOAP over HTTP*). I sam BPEL proces je izložen kao jedna WSDL usluga u čijem sučelju su izložene one metode kojima se kreira nova instanca poslovnog procesa njihovim pozivanjem od strane partnera. Integracija usluga u poslovni proces korištenjem BPEL standarda u SOA svijetu se nerijetko naziva orkestracijom te se nerijetko navodi kao jedna od funkcionalnosti koju Enterprise Service Bus (ESB) mora imati [16]. Slika 1 prikazuje osnovne elemente jednog BPEL procesa.



Slika 1: Osnovni elementi BPEL procesa

U ovom poglavlju su popisane i semantički, sa stajališta funkcionalnosti, analizirane suštinske konstrukcije ponuđene BPEL standardom, a koje služe za izgradnju jednog BPEL procesa. Kategorije ponuđenih konstrukcija su sljedeće:

- *konstrukcije za komunikaciju procesa s partnerima ili za definiciju te komunikacije* (`invoke`, `reply`, `receive`, `partnerLink`),
- *konstrukcije za kontrolu toka izvršavanja procesa* (`sequence`, `if`, `while`, `repeatUntil`, `pick`, `flow`, `forEach`),
- *varijable i konstrukcije za manipulaciju sa varijablama* (`variables`, `assign`, `property`, `propertyAlias`),
- *konstrukcija za stvaranje blokova unutar glavnog procesa* (`scope`),
- *konstrukcije reakcije na događaje* (`eventHandlers`),
- *konstrukcije za izbacivanje pogreške i reakcije na pojavu iste* (`throw`, `rethrow`,`errorHandlers`),
- *konstrukcije za završetak* (`terminationHandlers`),
- *konstrukcije za poništavanje ishoda uspješno završenog izvršavanja i konstrukcije pozivanja istih* (`compensationHandlers`, `compensate`, `compensateAll`),
- *konstrukcije za definiranje instanci procesa* (`correlationSets`).

Ponuđene konstrukcije su svrstane u kategorije na temelju sličnosti njihove namjene iako ih sama specifikacija standarda nije kategorizirala na ovaj način. Neke od konstrukcija se definiraju unutar glavnog BPEL XML elementa `process`, dok sve one, koje se vezuju za

definiciju izvršavanja procesa, se smještaju unutar primarne BPEL aktivnosti, a to je obično BPEL aktivnost `sequence` (slika 1). Slijedi semantički opis navedenih BPEL konstrukcija.

U konstrukcije za komunikaciju procesa s partnerima spadaju sljedeće BPEL aktivnosti:

- aktivnost `invoke` za pozivanje operacije partnera i prihvat odgovora od iste u sklopu koje se specificira WSDL `portType` partnera, naziv pozivane operacije, varijabla čija se vrijednost šalje kao ulazni parametar pozivane operacije i varijabla u koju se sprema odgovor operacije ukoliko isti postoji,
- aktivnost `receive` za pozivanje izložene operacije procesa od strane partnera u sklopu koje se specificira WSDL `portType` procesa, naziv pozivane operacije, varijabla čija se vrijednost šalje kao ulazni parametar pozivane operacije, `bool` atribut kojim se definira da li se ovom aktivnošću kreira instanca poslovnog procesa te opcionalno atribut kojim se uparuje aktivnost `reply` kojom se šalje odgovor prema partneru. receive aktivnost igra važnu ulogu u BPEL procesu budući da se instanca procesa može kreirati samo pozivanjem početne izložene operacije od strane partnera,
- aktivnost `reply` za slanje odgovora partneru koji je prethodno pozvao izloženu operaciju procesa kroz aktivnost receive i u sklopu koje se specificira WSDL `portType` partnera, naziv pozivane operacije, varijabla čija se vrijednost šalje kao ulazni parametar pozivane operacije, te opcionalno atribut kojim se specificira aktivnost `receive` kojom se primio poziva partnera na koji se šalje odgovor,
- `partnerLink` konstrukcije se mogu zamisliti kao instance WSDL `partnerLinkType` konstrukcija i one definiraju koju operaciju implementira partner, a koju implementira proces da bi međusobno mogli komunicirati.

U konstrukcije kontrole toka izvršavanja procesa spadaju sljedeće BPEL aktivnosti koje se uglavnom poklapaju sa standardnim konstrukcijama kontrole toka u drugim jezicima:

- aktivnost `sequence` za slijedno izvršavanje sadržanih BPEL aktivnosti,
- aktivnost `if` za uvjetno izvršavanje sadržanih BPEL aktivnosti,
- aktivnost `while` za ponavljano izvršavanje sadržane BPEL aktivnosti dok je uvjet izvršavanja zadovoljen (uvjet izvršavanja se ispituje prije izvršavanja sadržane aktivnosti),

- aktivnost `repeatUntil` za ponavljano izvršavanje sadržane BPEL aktivnosti dok je uvjet izvršavanja zadovoljen (uvjet izvršavanja se ispituje nakon izvršavanja sadržane aktivnosti),
- aktivnost `pick` za izvršavanje sadržanih BPEL aktivnosti nakon pojave pripadnog događaja, gdje događaj može biti vremenski okidač ili prijem poruke,
- aktivnost `flow` za paralelno ili sinkronizirano izvršavanje sadržanih BPEL aktivnosti,
- aktivnost `forEach` za paralelno ili serijsko izvršavanje sadržane scope aktivnosti predefinirani broj puta, te koja može imati prijevremeni uvjet završetka.

Budući da je aktivnost `flow` nestandardna aktivnost koja se ne susreće kod drugih programskih jezika i koja unutar sebe može sadržavati sinkronizacijski međuovisna izvršavanja, slijedi detaljniji pregled mehanizma sinkronizacije unutar aktivnosti `flow`. Sinkronizacijski međuovisno izvršavanje sadržanih aktivnosti teoretski može biti kompleksno budući da je velik broj mogućih scenarija. Međuovisno izvršavanje implementirano je pomoću sinkronizacijskih poveznica između aktivnosti (engl. `links`) na sljedeći način:

- svaka aktivnost može biti izvor jedne ili više sinkronizacijskih poveznica,
- svaka aktivnost može biti odredište jedne ili više sinkronizacijskih poveznica,
- svaka sinkronizacijska poveznica ima svoj uvjet nastavka (engl. `transitionCondition`) koji se ispituje nakon što aktivnost završi i kojim poveznica, koja izlazi iz te aktivnosti, poprima svoj status,
- svaka aktivnost, koja ima jednu ili više sinkronizacijskih poveznica kao svoje odredište, posjeduje uvjet spajanja (engl. `joinCondition`) koji je `boolean` konstrukcija statusa ulaznih poveznica i koji se ispituje kada svi statusi ulaznih poveznica budu poznati i aktivnost dođe na svoj red izvođenja te čija vrijednost (`true/false`) određuje da li će aktivnost biti izvođena ili ne. Svaka aktivnost posjeduje atribut `supressJoinFailure` čija vrijednost (`true/false`) određuje da li će se izbaciti pogreška ukoliko vrijednost uvjeta spajanja bude `false` i aktivnost se temeljem toga ne izvrši. Ukoliko je vrijednost ovoga atributa `true`, to podrazumijeva da se pogreška, koja simbolizira neispunjenje uvjeta izvršavanja, neće izbaciti i sve poveznice koje izlaze iz predmetne neizvršene aktivnosti poprimaju status `false`. Ukoliko je pak vrijednost ovoga atributa `false`, to podrazumijeva izbacivanje pogreške o neispunjenju uvjeta izvršavanja koja se potom obrađuje od strane pripadajuće kontrole obrade pogreške (engl. `errorHandler`).

BPEL varijable služe za prihvat poslanih poruka od partnera, za kontrolu toka na temelju njihovih vrijednosti, za slanje odgovora partneru, za međupohranu rezultata i slično. Uz njih se specificiraju ime te tip. Mogu biti definirane na temelju WSDL `messageType` mehanizma, te na temelju `xsd:type` ili `xsd:element` mehanizma. `Property` i `propertyAlias` konstrukcije su mehanizmi kojima se u BPEL procesu omogućava definicija sinonima za određeni tip varijabli.

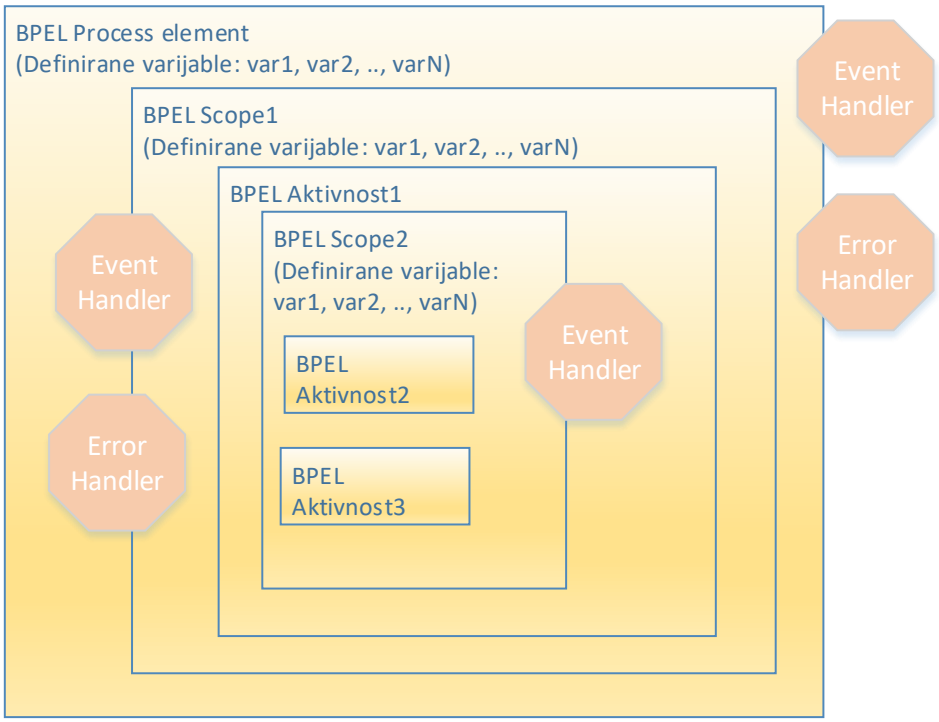
BPEL aktivnost `assign` služi za inicijalizaciju ili mijenjanje vrijednosti varijablama.

Aktivnost `scope` je BPEL aktivnost koja unutar sebe kao ugniježđenu može sadržavati bilo koju BPEL aktivnost, vlastite varijable vidljive samo unutar nje, te pridružene specijalne BPEL konstrukcije za obradu događaja (engl. `eventHandlers`), pogreški (engl. `errorHandlers`), alternativno djelovanje - kompenzaciju (engl. `compensationHandlers`) i završetak (engl. `terminationHandlers`), to jest ona može služiti kao vlastiti blok izvršavanja unutar procesa. Aktivnost `scope` se dakle može zamisliti kao autonomna jedinica unutar procesa. Na slici 2 je vizualno demonstrirana logika BPEL aktivnosti `scope`.

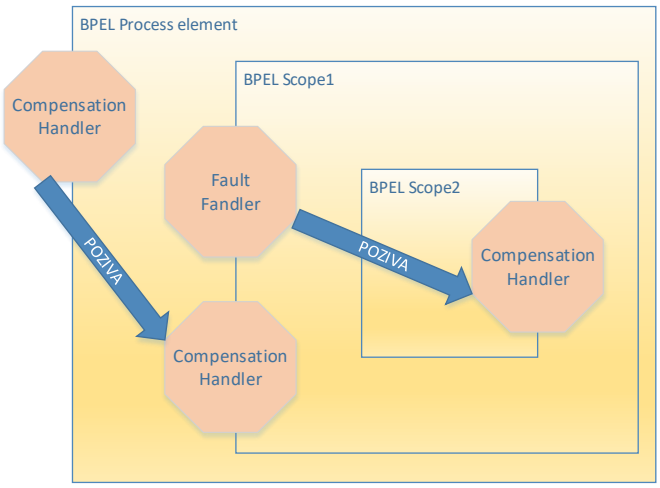
S BPEL aktivnošću `scope` se povezuju BPEL konstrukcije za obradu događaja. One mogu biti dvojake, to jest mogu biti reakcija na prijem poruke (`onMessage`) ili reakcija na dosegnuti vremenski trenutak ili vremensko trajanje (`onAlarm`) i povezuju se s aktivnostima `scope` na način da su prikačene za njih i paralelno se izvršavaju s njima ukoliko se dogode.

BPEL konstrukcije za obradu pogreški se izvršavaju ukoliko dođe do pojave pogreške unutar aktivnosti `scope` čime se izvršavanje aktivnosti `scope` zaustavlja i poziva se pripadna konstrukcija za obradu pogreške. Dakle, konstrukcije za obradu pogreški se ne izvršavaju paralelno s aktivnostima `scope`.

BPEL konstrukcije za alternativno djelovanje prethodno uspješno završene `scope` aktivnosti se mogu izvršiti samo ukoliko je aktivnost `scope` prethodno uspješno završila i one se pozivaju iz konstrukcija za obradu pogreške ili alternativno djelovanje roditeljskih aktivnosti `scope` radi poništavanja prethodno uspješno završene BPEL aktivnosti `scope` (slika 2).



Slika 2: Logika BPEL aktivnosti `scope`



Slika 3: Pozivanje BPEL aktivnosti `compensationHandler`

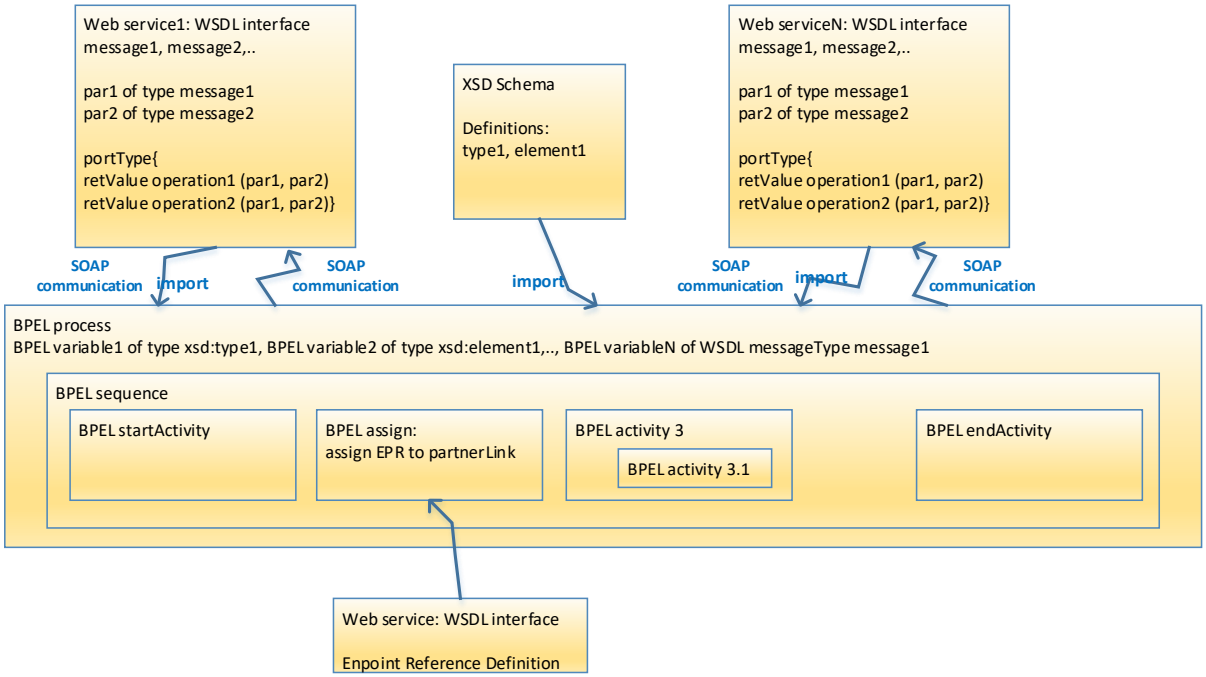
2.2. Veza s drugim standardima

U prethodnom dijelu poglavlja već je spomenuta veza između BPEL standarda i WSDL standarda za opisivanje sučelja usluga te je spomenut SOAP standard razmjene poruka između procesa i partnera. U ovom dijelu poglavlja će se detaljnije analizirati veza koju BPEL proces ima s WSDL standardom, prikazat će se struktura WSDL sučelja te će se izdvojiti konstrukcije WSDL sučelja bitne za BPEL proces.

WSDL sučelja na dogovoreni način opisuju sve potrebne informacije bitne za komunikaciju s izloženom uslugom/partnerom, a u koje spadaju sljedeće:

- WSDL `message` – struktura koja se sastoji od dijelova i kojom se opisuje poruka koju primaju operacije izložene od strane partnera, povratna vrijednost izložene operacije u slučaju normalnog odgovora i povratna vrijednost u slučaju pogreške,
- WSDL `portType` – struktura koja objedinjuje skup operacija u cjelinu,
- `partnerLinkType` - struktura koja objedinjuje jednu ili dvije `portType` konstrukcije i koja služi za definiranje uloga koje usluge (partneri), koje međusobno komuniciraju, moraju implementirati za uspješnu komunikaciju,
- WSDL `service` – struktura koja objedinjuje jedan ili više WSDL `port` elemenata, pri čemu `port` element sadržava informacije o komunikacijskom protokolu koji usluga koristi i o adresi na kojoj je usluga dostupna.

Standard koji je također povezan s BPEL standardom je Web Service Addressing Standard (WS-ADDRESSING) [17] standard čija se konstrukcija referenca krajnje točke (engl. *end-point reference*) za opisivanje adrese na kojoj je usluga dostupna, može koristiti za dinamičko povezivanje BPEL procesa s konkretnim ponuditeljima usluga pri čemu se podrazumijeva da je ponuditelj pripadajućeg tipa definiranog u BPEL procesu, to jest da implementira potrebni `portType`. S definicijom BPEL procesa direktno se povezuju i XSD dokumenti u kojima su definirani tipovi na temelju kojih se definiraju varijable u BPEL procesu, a koji se uvezuju u definiciju BPEL procesa. Slika 4 vizualno prikazuje blokovsku XML logiku BPEL standarda te njegovu povezanost s drugim standardima.

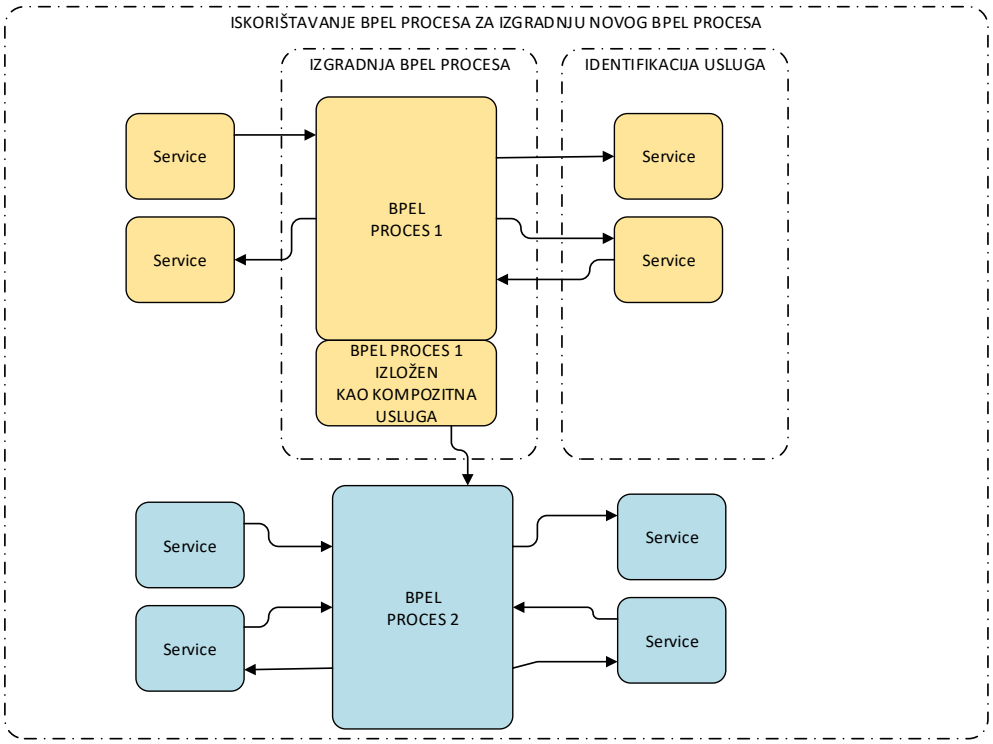


Slika 4: Veza BPEL standarda s drugim standardima

3. Model za fazni razvoj BPEL procesa

3.1. Model faznog razvoja

Što se tiče razvoja sustava vezanih za uslužno orijentirane arhitekture, postoje brojni pristupi. U [18][19] su se autori fokusirali na identifikaciju funkcionalnosti koja će biti izražena kao usluga pri čemu su kao osnovni kriterij naveli zahtjeve poslovnog procesa. Oni su se fokusirali na razvoj metodologije za identifikaciju funkcionalnosti usluge na temelju zahtjeva poslovnog procesa, a koje će se kasnije integrirati u željeni poslovni proces te koje će se kao autonomne jedinice moći ponovno iskoristiti u drugim poslovnim procesima. U [20] su se autori fokusirali na sigurnost usluga. U [21] je fokus na metodologiji kojom se usluge mogu identificirati na temelju UML dijagrama slučajeva korištenja (UML *use-case*) i analize poslovnog procesa. Svi ovi navedeni primjeri su bili fokusirani na identifikaciju usluga, no nigdje fokus nije bio na razradi metodologije za integraciju već postojećih usluga u poslovni proces što bi se moglo shvatiti kao faza koja dolazi nakon uspješno definirane faze identifikacije i implementacije pojedinačnih usluga. Može se reći da nakon identifikacije pojedinačnih usluga ili odlukom o iskorištavanju već postojećih usluga slijedi proces izgradnje jedne kompozitne usluge na temelju postojećih, gdje se ta kompozitna usluga naziva poslovni proces i kao takva se opet izlaže prema svijetu kao jedna nova usluga čime se otvara mogućnost za njeno iskorištavanje za izgradnju neke još kompleksnije usluge, odnosno novog poslovnog procesa. Slika 5 prezentira ovu ideju, odnosno mogućnost ponovnog iskorištavanja.



Slika 5: Faza izgradnje BPEL procesa u odnosu na druge faze

U ovom poglavlju naglasak nije na identifikaciji temeljnih usluga ili njihovoj identifikaciji, niti na mogućnostima ponovnog iskorištavanja kompozitnih usluga/procesa, već na razradi modela ili metodologije za integraciju već postojećih usluga u poslovni proces kako bi se njegova izgradnja kompletirala na najbolji mogući način i kako bi to bio jedan cjelovit proces u čijim koracima bi učestvovali heterogeni izvođači: od vlasnika poslovnih procesa koji poznaju kako isti funkcionira, preko razvojnih timova zaduženih za razvoj i implementaciju do tima za testiranje poslovnog procesa.

Prijedlog modela koja bi omogućio fazni razvoj BPEL poslovnog procesa prikazan je na slici 6 pri čemu su faze definirane na sljedeći način:

1. Faza vizualnog modeliranja procesa:

    I. Modeliranje vizualnog modela procesa odabranim standardom za vizualno modeliranje, a u ovom slučaju BPMN standardom [11], do definirane razine vizualnog modeliranja, korištenjem dogovorenih pravila za vizualno modeliranje, to jest korištenjem utvrđenih BPMN uzoraka slijeda izvršavanja koji odgovaraju pojedinačnim BPEL aktivnostima. U ovoj fazi se dakle generira dobro formirani BPMN model pri čemu „*dobro formiran*

model“ podrazumijeva da su korišteni dogovoreni uzorci koje će algoritam preslikavanja BPMN-BPEL moći prepoznati,

II. Korištenje algoritma za automatsko preslikavanje iz vizualnog modela procesa u izvršni BPEL model procesa, što se svodi na korištenje algoritma preslikavanja BPMN-BPEL, pri čemu se dobiva potencijalno nepotpun izvršni XML BPEL kod (koji je potrebno ručno dopuniti konstrukcijama koje nedostaju), fragmenti pripadnih WSDL sučelja i XSD dokumenata),

2. Nadogradnja izvršnog modela procesa:

   I. Ručna nadogradnja izvršnog BPEL modela procesa konstrukcijama koje nedostaju u izvršnom BPEL modelu ukoliko iste nisu uključene razinom vizualnog modeliranja procesa i ručna nadogradnja pripadnih WSDL sučelja i pripadnih XSD dokumenata čiji fragmenti su također generirani algoritmom preslikavanja BPMN-BPEL,

   II. Preslikavanje izmjena iz izvršnog BPEL modela u vizualni BPMN model korištenjem algoritma preslikavanja BPEL-BPMN ukoliko dođe do izmjene izvršnog BPEL modela uslijed bilo kojeg razloga,

3. Formalna provjera procesa:

   I. Označavanje elemenata BPEL procesa na način da se na temelju definiranih oznaka može generirati pravilan simbolički model BPEL procesa. Simbolički model BPEL procesa na dogovoreni, standardiziran način prikazuje one elemente BPEL izvršnog modela koji su relevantni za formalnu provjeru i njegova svrha jeste olakšavanje generiranja formalnog modela,

   II. Pretvaranje simboličkog modela u formalni model procesa temeljem definiranih pravila pretvaranja između elemenata simboličkog modela i elemenata formalnog modela,

   III. Formalna provjera željenog svojstva nad formalnim modelom,

   IV. Lociranje pogrešaka, ukoliko istih temeljem formalne provjere ima, u formalnom modelu i potom na temelju simboličkog modela, njihovo lociranje u izvršnom BPEL modelu,

4. Ispravak izvršnog BPEL modela procesa:

    I. Ispravak izvršnog BPEL modela procesa sukladno pronađenim pogreškama u koraku 3.IV,

    II. Preslikavanje izmjena iz izvršnog BPEL modela u vizualni BPMN model korištenjem algoritma preslikavanja BPMN-BPEL ukoliko ih ima,

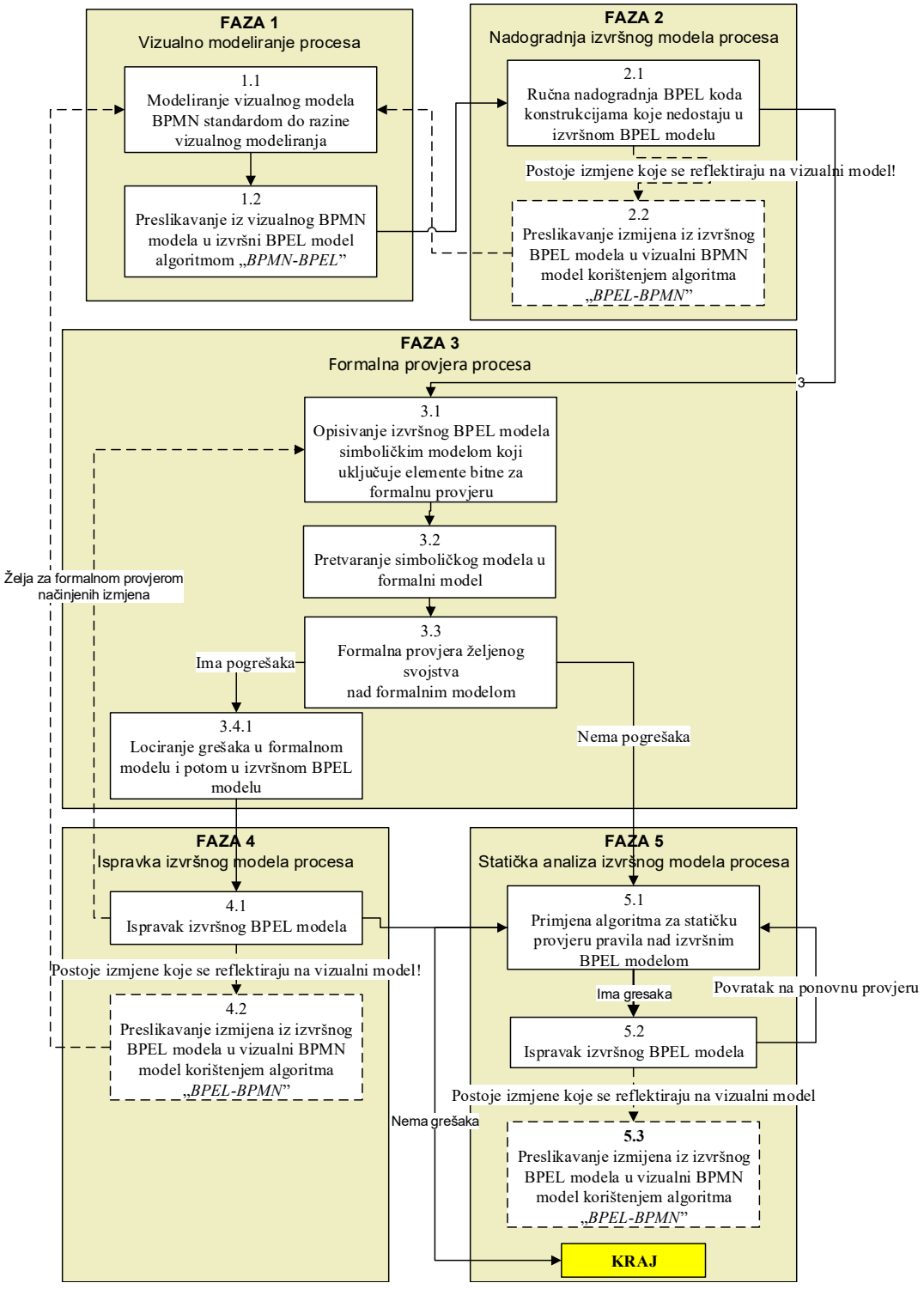
5. Statička analiza izvršnog modela procesa:

    I. Primjena algoritma za statičku provjeru odabranih pravila nad izvršnim BPEL modelom (pravila definiranih BPEL specifikacijom) i ispravak izvršnog BPEL modela ukoliko algoritam naiđe na nedosljednosti koje se moraju ispraviti,

    II. Preslikavanje izmjena iz izvršnog BPEL modela u vizualni BPMN model korištenjem algoritma preslikavanja BPEL-BPMN ukoliko ih ima.

3.2. Faze modela

Slijedi detaljniji prikaz koraka predloženog faznog modela, dok će u narednim poglavljima biti detaljnije objašnjeni algoritmi koji se koriste u modelu. Slika 6 prikazuje navedene korake predloženog modela te poveznice među njima.



Slika 6: Faze modela za fazni razvoj BPEL procesa

Budući da se BPEL povezuje s integracijom usluga/partnera u poslovne procese, u prvu fazu razvoja moraju biti uključeni vlasnici poslovnih procesa koji poznaju korake poslovnog procesa, to jest poznaju način na koji usluge moraju biti integrirane. Vlasnici poslovnih procesa

u suradnji s tehničkim timom bi učestvovali u ovoj prvoj fazi oslikavanja vizualnog kostura poslovnog procesa koji zbog svoje vizualne prirode posjeduje veću ekspresivnost nego bilo koji drugi oblik oslikavanja procesa.

Kako je već rečeno, model za fazni razvoj procesa kreće od već pripremljenih usluga koje će biti uključene u poslovni proces, to jest ovaj model se ne bavi identifikacijom funkcionalnosti koje će se izložiti kao pojedinačne usluge nego kreće od njihove integracije u novu kompozitnu uslugu.

Kao standard za modeliranje predložen je BPMN standard zbog njegove uske povezanosti s poslovnim procesima. Službeno je BPMN standard predstavljen kao standard za vizualno oslikavanje procesa. Postoje brojna okruženja koja omogućavaju modeliranje korištenjem ovoga standarda [22][23][24] te je zbog široke uporabe i standardizacije i odabran za korištenje u prvoj fazi modela.

BPMN standard posjeduje bogate konstrukcije kojima se mogu oslikavati poslovni procesi. Oslikavanje BPEL procesa korištenjem BPMN standarda neće biti u omjeru 1:1 što podrazumijeva da ne postoje gotove, pojedinačne BPMN konstrukcije za svaki pojedinačni oblik BPEL aktivnosti no dogovorenom kombinacijom nekoliko BPMN aktivnosti se može postići dogovor za vizualno oslikavanje svake pojedinačne BPEL aktivnosti. U Preslikavanje između BPMN i BPEL definirano je koji će se dijelovi BPEL procesa prikazati vizualno, to jest definirana je razina vizualnog modeliranja. Kao što se u tom poglavlju vidi, većina BPEL aktivnosti obuhvaćena je vizualnim modelom, osim nekih specifičnih konstrukcija kao što su: `property`, `propertyAlias`, `partnerlink` i `correlationSet`. U sklopu istog poglavlja definiran je i pseudokod algoritma preslikavanja BPMN-BPEL te su uparena XML preslikavanja između dijelova BPMN grafa i XML koda pripadne BPEL aktivnosti. Zamisao je da preslikavanja između vizualnog BPMN i izvršnog BPEL modela budu automatska čime faza vizualnog modeliranja ne bi bila teret, nego olakšica.

Vještine, kojima bi osoba koja radi na procesu u prvoj fazi vizualnog modeliranja morala ovladati, osim poznavanja odabranog BPMN alata, jesu i definirani BPMN uzorci iz poglavlja četiri, a koji odgovaraju pojedinačnim BPEL aktivnostima budući da BPMN graf mora sadržavati isključivo dogovorene uzorke ponašanja kako bi se isti uspješno mogli preslikati u pripadni BPEL proces, to jest vizualni model mora biti valjan sa stajališta algoritma preslikavanja BPMN-BPEL.

Analizirajući postojeći rad na području pomoćnog modeliranja BPEL procesa, autori u [25] koriste formalni, vlastiti jezik *BliteC* koji omogućava olakšano modeliranje BPEL procesa i automatsko preslikavanje u izvršni BPEL model. U [26][27] su se autori fokusirali na model razvijen UML dijagramom aktivnosti i na njegovo preslikavanje u BPEL proces. U [28] su se autori fokusirali na pristupe preslikavanju iz BPMN modela u BPEL model. U [29] autor također sistematski analizira postojeći rad na preslikavanjima između BPMN i BPEL modela i zaključuje da postoji veliki jaz između prirode standarda. U [30] su se autori fokusirali na identificiranje pojedinačnih obrazaca radnog toka (engl. *workflow patterns*) i na mogućnostima njihovog prikazivanja BPMN standardom. U [31] su se autori fokusirali na vizualnu izgradnju BPEL procesa koristeći sukladno definiranim pravilima valjane BPMN dijagrame koje su nazvali *Business Process Diagram* (BPD). BPD dijagrami se potom raščlanjuju na komponente pri čemu su za svaku identificiranu komponentu definirana preslikavanja u pripadne BPEL aktivnosti. Definirana je funkcija koja BPD dijagrame inkrementalno raščlanjuje na komponente i vrši njihovo preslikavanje u pripadne BPEL aktivnosti. Preslikavanje je definirano samo iz smjera BPMN su BPEL, ali ne i obratno. U [32] su autori predstavili prijedlog preslikavanja konkretnog procesa rezervacije putovanja iz smjera BPMN u BPEL kroz koji su naveli uparene vizualne BPMN konstrukcije i pripadajući BPEL kod za određene BPEL aktivnosti i scenarije izvršavanja koji se pojavljuju unutar procesa uzetog kao primjer.

Druga faza razvoja vezana je za pripremu izvršnog BPEL modela za implementaciju u korišteno izvršno BPEL okruženje. Postoje brojna izvršna BPEL okruženja [8][9][10] i sva bi trebala biti dosljedna BPEL specifikaciji [3], to jest na identičan način interpretirati BPEL kod. Izlazni BPEL model iz prve faze je nepotpun i potrebno ga je statički pregledati, to jest dodati sve konstrukcije koje nisu uključene fazom vizualnog modeliranja, a koje su neophodne za produkcijsko izvršavanje.

Ova faza bi se trebala izvoditi od strane tehničkog tima koji poznaje direktno XML *Schemu* BPEL standarda ili alate za njeno uređivanje. Osim nadogradnje XML koda BPEL standarda, u ovoj fazi se trebaju nadograditi ili pripremiti i pripadni fragmenti povezani s BPEL procesom: WSDL sučelja partnera i procesa i pripadni XSD dokumenti koji se uvoze u BPEL proces.

U drugoj fazi je stvoren i prostor za izmjene nad BPEL kodom koje mogu utjecati na semantiku izvršavanja poslovnog procesa, to jest u suradnji s vlasnicima poslovnog procesa mogu se vršiti i značajnije izmjene nad izvršavanjem poslovnog procesa koje bi se korištenjem algoritma preslikavanja BPEL-BPMN reflektirale nazad u vizualni model. Algoritam preslikavanja

BPEL-BPMN preslikava cjelovit BPEL proces do razine obuhvaćene fazom vizualnog modeliranja. Novonastali vizualni model zamjenjuje postojeći vizualni model prethodno kreiran u fazi 1. Izmjene nad BPEL kodom mogu biti posljedica grešaka u fazi vizualnog modeliranja no one mogu nastati i kao posljedica naknadnih, željenih izmjena nad procesom.

Treća faza vezana je za provjeru željenih svojstava BPEL procesa nad svim njegovim izvršavanjima. Klasične tehnike testiranja, kod sustava koji posjeduju paralelna izvršavanja, je dosta težak proces, možda čak i nemoguć budući da je teško predvidjeti brzinu izvršavanja paralelnih komponenti [12]. Budući da BPEL procesi mogu sadržavati paralelna izvršavanja, kandidat su za formalnu provjeru sustava. U Formalna provjera BPEL procesa je opisan algoritam za preslikavanje u odabrani formalni model za provjeru odabranih željenih svojstava, a u doktoratu su to provjere manipulacija nad varijablama budući da su varijable prepoznate kao ključni nosioci informacija poslovnog procesa i od ogromnog je značaja činjenica da u procesu postoji paralelno ažuriranje i/ili čitanje varijabli. Kod nekih sustava postojanje putanje s pogreškom nije toliko kritično, pa se nakon što se ona pojavi sustav može ispraviti, no kod nekih sustava je vrlo bitno da budu isporučeni „bez pogreške“ jer na tim sustavima počivaju iznimno osjetljivi procesi poput kontrole letova, kontrole robe u proizvodnji, signalizacije i slično. Formalna provjera prezentirana u doktoratu otvorila je mogućnost za daljnji rad u tom području na način da se sustav može formalizirati na više načina obzirom na vrstu željenih svojstava koja se žele provjeriti.

Na području formalne verifikacije sustava postoji mnogo znanstvenog doprinosa. U [33] [34] su se autori fokusirali na translaciju BPEL konstrukcija u Petrijeve mreže i iskorištavanje postojećih metodologija analize Petrijevih mreža za ispitivanje svojstava BPEL procesa. U [35] su autori razvili vlastiti referentni model nazvan *RMWSComposition* za hvatanje dijelova kompozicija usluga i predstavili su formalne metode za verifikaciju modela. U [36] su autori razvili vlastitu metodu *Event-B* za verifikaciju BPEL procesa. U [37] autori koriste *π-calculus* za formalnu verifikaciju BPEL procesa.

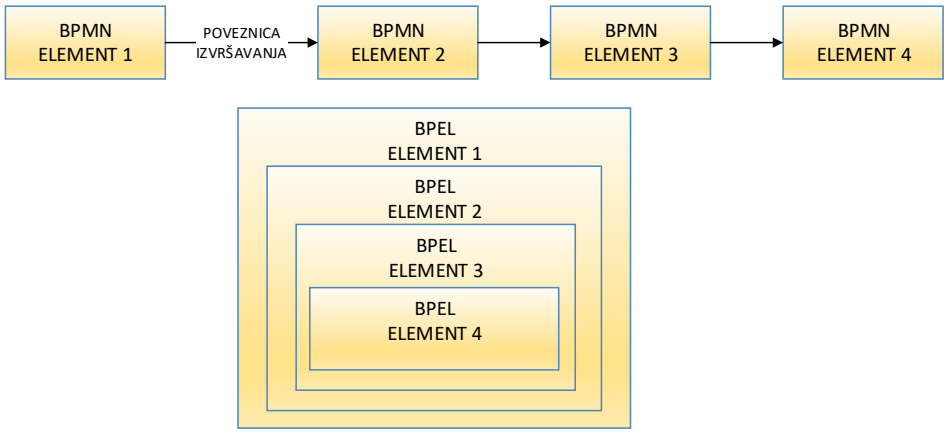
Nakon lociranja potencijalno opasnih izvršavanja u izvršnom BPEL modelu, može doći do izmjena istog no ne mora nužno ukoliko se detektirana ponašanja formalne provjere ne ocijene „opasnima“ za poslovni proces. Kao i u drugoj fazi, ukoliko dođe do izmjena nad izvršnim modelom, a koja se reflektiraju nazad na vizualni model, algoritam preslikavanja BPEL-BPMN preslikava izmijenjeni BPEL proces u vizualni model s kojim se zamjenjuje postojeći vizualni model.

BPEL specifikacija [3] definirala je skup pravila koja moraju biti zadovoljena od strane procesa, a koja se statičkom analizom procesa trebaju provjeriti. Neka od pravila su sistemske prirode namijenjena provjeri od strane odabranog BPEL izvršnog okruženja, dok su neka neovisna od BPEL izvršnog okruženja te se algoritmom statičke provjere mogu provjeriti nad izvršnim BPEL modelom čime se stječe neovisnost razvoja BPEL procesa od bilo koje izvršne BPEL infrastrukture.

4. Preslikavanje između BPMN i BPEL

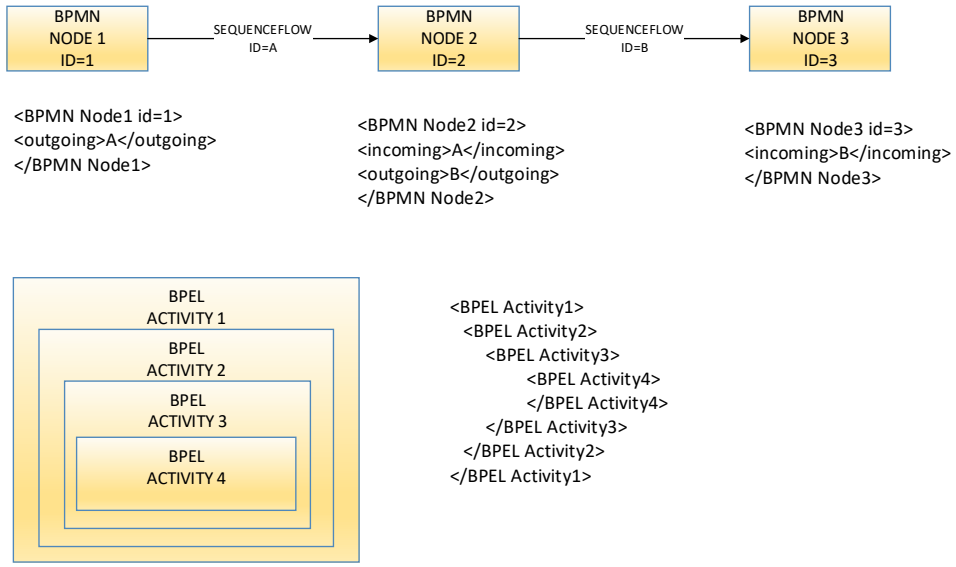
BPMN standard je odabran za vizualno modeliranje zbog standardizacije, velikog broja dostupnih alata za modeliranje, široke prihvaćenosti, povezanosti s procesima, i povezanosti same BPMN specifikacije s BPEL specifikacijom. U pozadini je također baziran na XML standardu stoga je dobar kandidat za razvoj prototipa algoritma automatskog dvosmjernog preslikavanja s BPEL standardom budući da se preslikavanje izvršava nad pripadnim XML *Schema*ma. Dosta je rada na području preslikavanja između modela BPMN i BPEL standarda no većina njih se na kraju složila kako je struktura standarda dosta različita i kako nije laka zadaća napraviti neki prihvatljiv pristup koji bi optimalno to odradio i uključio što veći broj BPEL konstrukcija [29].

Iako su i BPMN i BPEL standardi bazirani na XML standardu, važno je naglasiti razliku između logike povezivanja elemenata unutar BPMN grafa i BPEL grafa [29]. Povezani BPMN elementi (koji čine BPMN slijed unutar BPMN grafa) povezani su poveznicama izvršavanja `sequenceFlow`, dok su povezani BPEL elementi (koji čine BPEL slijed unutar BPEL grafa) sadržani jedni u drugima, to jest BPMN standard je baziran na blokovima kao većina programskih jezika. Slika 7 simbolično prikazuje razliku povezivanja elemenata u BPMN i BPEL grafovima.



Slika 7: Slijedna BPMN naspram blokovska BPEL logika povezivanja

Na XML razini povezivanja su definirana na sljedeći način (slika 8).



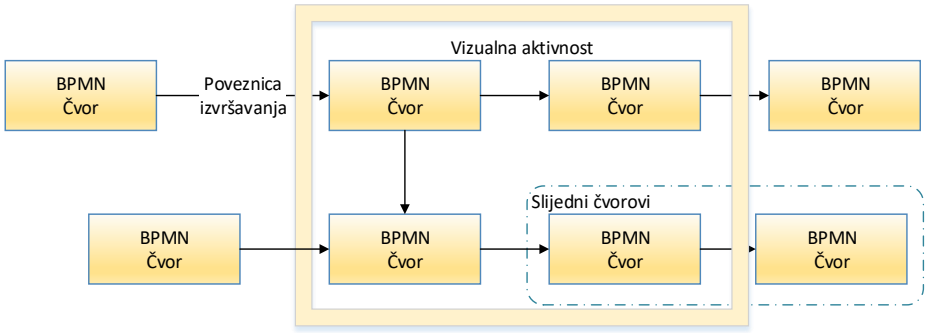
Slika 8: Slijedna BPMN naspram blokovska BPEL logika (XML razina)

Terminologija koja će se koristiti u tekstualnom opisu preslikavanja između BPMN i BPEL standarda je sljedeća:

- jedan element BPMN grafa nazvan je čvor,
- strelice u BPMN grafu, koje povezuju čvorove, nazvane su poveznicama izvršavanja,
- skup povezanih čvorova, koji se kao cjelina mogu preslikati u jednu BPEL aktivnost, nazvani su "vizualna aktivnost",
- jedan element BPEL grafa nazvan je aktivnost,
- BPMN graf nazvan je BPMN izvršavanje,
- BPEL graf nazvan je BPEL izvršavanje,
- poveznica, koja vizualno ulazi u BPMN čvor, nazvana je dolaznom poveznicom,
- poveznica, koja vizualno izlazi iz BPMN čvora, nazvana je odlaznom poveznicom,
- terminologija „BPMN element koji nema dolaznih poveznica“ odnosi se na element koji vizualno nema poveznicu koja ulazi u njega, dok je na XML razini to element koji nema dijete element `incoming`,
- terminologija „BPMN element koji nema odlaznih poveznica“ odnosi se na element koji vizualno nema poveznicu koja izlazi iz njega, dok je na XML razini to element koji nema dijete element `outgoing`,
- terminologija „slijedni element X elementu Y“ odnosi se na elemente X i Y koji su vizualno direktno povezani poveznicama izvršavanja, dok na XML razini element X

ima dijete element `outgoing` čiji je sadržaj jednak sadržaju dijete elementa `incoming` elementa Y (slika 9).

Slika 9 vizualno označava definirano imenovanje u dokumentu.



Slika 9: Vizualno oslikavanje prihvaćenog BPMN imenovanja

4.1. Algoritam preslikavanja BPMN-BPEL

Preslikavanje između BPMN i BPEL standarda mora biti dvosmjerno. U ovom poglavlju je prikazan opis preslikavanja iz smjera BPMN standarda u BPEL standard te će se za taj smjer preslikavanja koristiti terminologija BPMN-BPEL.

4.1.1. Koraci preslikavanja

Algoritam preslikavanja BPMN-BPEL se izvršava kroz sljedeće korake.

1) Svi elementi BPMN izvršavanja smješteni su sukladno sljedećoj strukturi:
   `<definitions>..korijenski elementi..`
   `<process>..elementi procesa..</process>..</definitions>`,

   dok su svi elementi BPEL izvršavanja smješteni sukladno sljedećoj strukturi:
   `<process>..korijenski elementi..`
   `<sequence>..elementi procesa..</sequence>..</process>`.

2) Algoritam preslikavanja BPMN-BPEL prvo vrši pronalazak i preslikavanje BPMN koda u BPEL varijable, WSDL poruke, WSDL operacije i WSDL priključke koji se neće nalaziti unutar BPEL `sequence` elementa no dio su BPEL procesa. BPEL

varijable spadaju u korijenske elemente koji se smještaju kao djeca elementi direktno unutar BPEL elementa `process`.

U korijenske elemente spadaju i aktivnosti `*handler` zakačene za BPEL aktivnost `process` te se one također mogu otkriti u ovom koraku ili pak nakon popunjavanja glavne BPEL aktivnosti `sequence`.

Osim prethodno navedenog, u korijenske elemente spadaju i BPEL `partnerLink` konstrukcije no one nisu obuhvaćeni razinom vizualnog modeliranja. Sve što nije obuhvaćeno razinom vizualnom modeliranja popunjavat će se ručno u BPEL procesu.

Rezultat koraka 2:

```
<process>
<variables><variable1/>..</variableN></variables>
</process>
```

3) Nastavak rada algoritma preslikavanja BPMN-BPEL vezan je za kreiranje BPEL elementa `sequence` unutar kojega će se blokovski slijedno smještati sve ostale BPEL aktivnosti (elementi) kreirane algoritmom. BPEL element `sequence` smješta se unutar BPEL elementa `process`.

Rezultat koraka 3:

```
<process>
<variables><variable1/>..</variableN></variables>
<sequence>..</sequence >
</process>
```

4) Nakon kreiranja glavnog XML BPEL elementa `sequence` slijedi njegovo popunjavanje. Popunjavanje elementa `sequence` izvršava se prolaskom kroz djecu elemente BPMN elementa `process` izvršavajući sljedeće korake:
   i. prepoznavanje i generiranje početnih BPEL aktivnosti,
   ii. pamćenje poveznice koja izlazi iz početne vizualne aktivnosti,
   iii. otkrivanje BPMN elementa koji slijedi iza početne vizualne aktivnosti i provjera da li se on sam može preslikati u pripadajuću BPEL aktivnost (element se može zvati čvorom), a ako ne, provjera slijednog izvršavanja i otkrivanje koja BPEL vizualna aktivnost počinje tim elementom i preslikavanje iste u pripadajuću BPEL aktivnost,

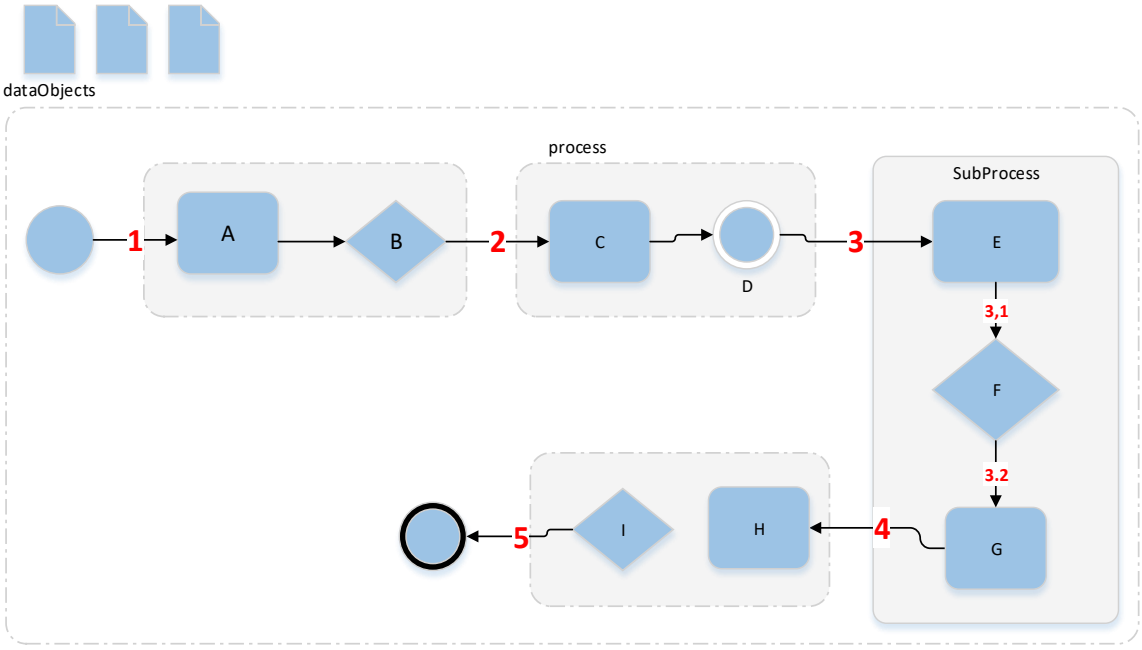
iv. pamćenje poveznice koja izlazi iz neposredno otkrivene vizualne aktivnosti,

v. ponavljanje sa narednim čvorom/vizualnom aktivnošću (povratak na korak iii) dok se ne dođe do čvora koji nema odlaznih poveznica čime se glavna BPEL aktivnost `sequence` smatra popunjenom i preslikavanje smatra završenim.

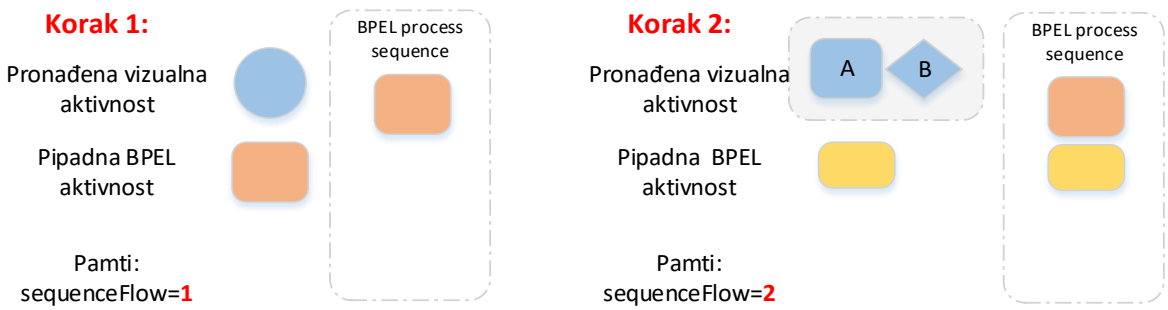
Rezultat koraka 4:

```
<process>
<variables>..<variable1/>..<variableN/></variables>
<sequence>..svi otkriveni BPEL elementi..</sequence>
</process>
```

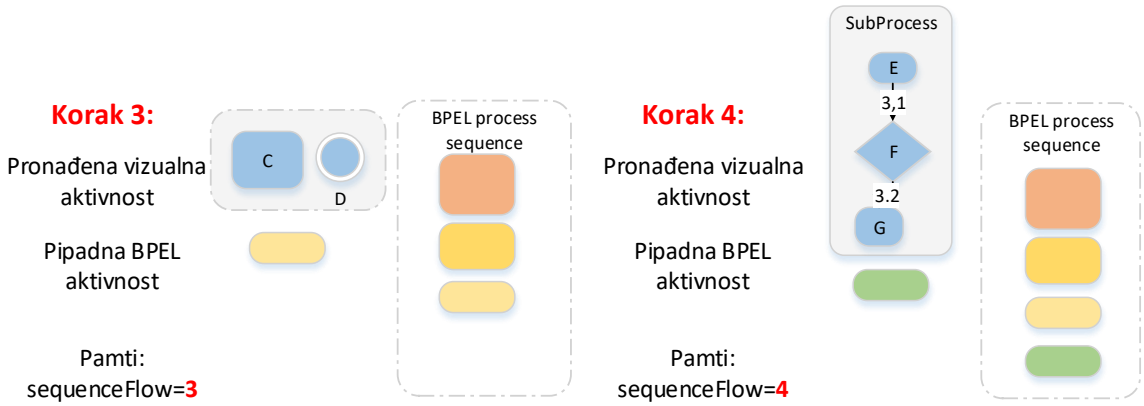
Koraci algoritma preslikavanja BPMN-BPEL (od drugog do četvrtog koraka) vizualno su predstavljeni donjim slikama 10 do 13.



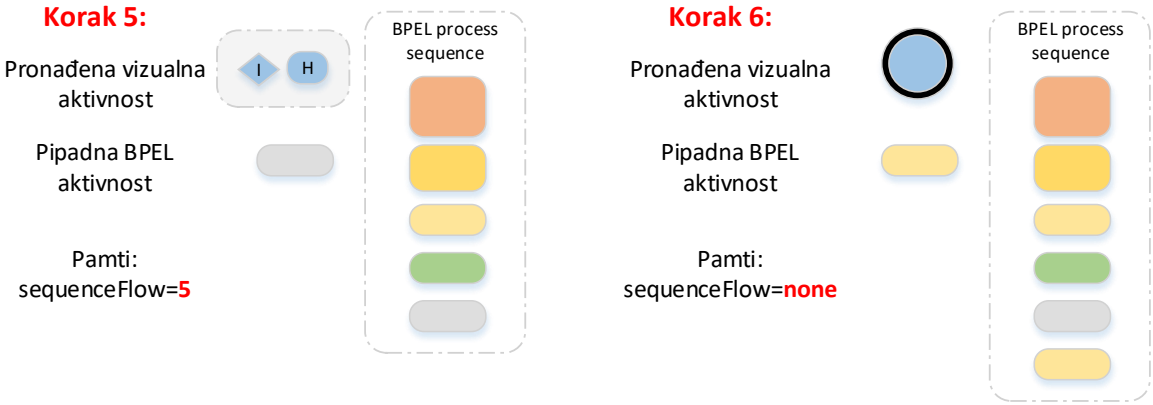
Slika 10: Početak algoritma preslikavanja BPMN-BPEL



Slika 11: Koraci 1 i 2 algoritma preslikavanja BPMN-BPEL

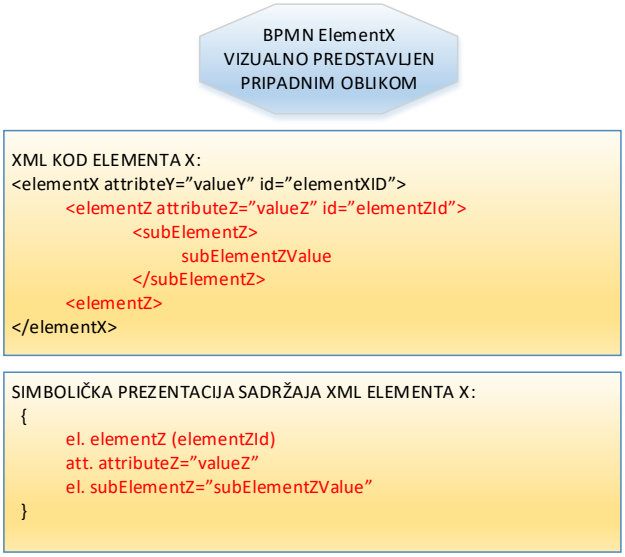


Slika 12: Koraci 3 i 4 algoritma preslikavanja BPMN-BPEL



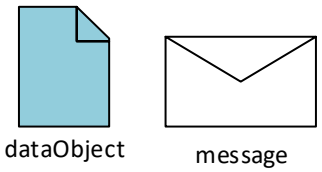
Slika 13: Koraci 5 i 6 algoritma preslikavanja BPMN-BPEL

Prethodni koraci opisno i slikovito su definirali korake algoritma preslikavanja BPMN-BPEL, dok tablice koje slijede, na simbolički način prikazuju BPMN XML kodove pojedinačnih vizualnih BPEL aktivnosti, njihovu vizualnu interpretaciju i pripadni BPEL XML kod. Simbolički opis BPMN XML koda uveden je radi uštede prostora. Slika 14 prikazuje tumačenje korištene notacije.



Slika 14: Simbolika XML prikaza

Preslikavanja ispod prikazuju BPMN XML kod elemenata `dataObject`, `message` i `error` koji se preslikavaju u elemente BPEL `variable` i WSDL `message`. Masnim slovima naglašeni su dijelovi koji se međusobno preslikavaju iz jednoga u drugi u donjem kodu kao i u svim narednim uparenim XML kodovima preslikavanja.



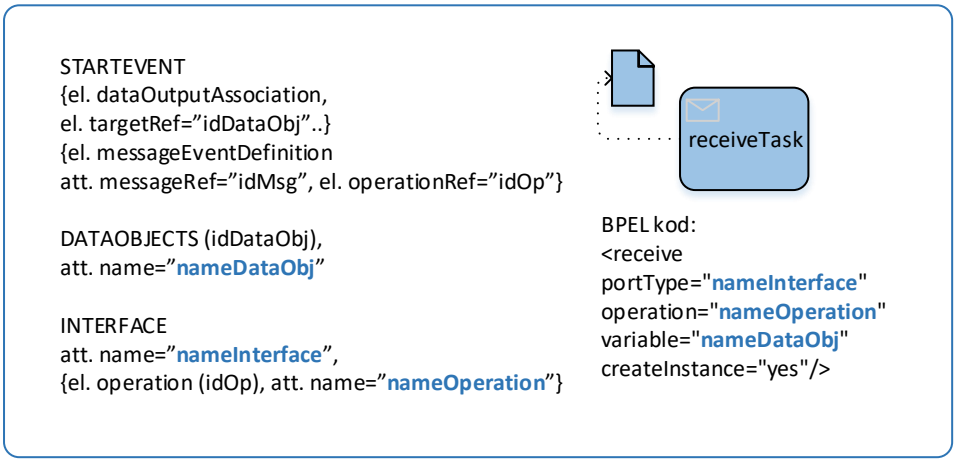
Slika 15: BPEL `dataObject` *i* `message`

```
BPMN DATAOBJECT/MESSAGE/ERROR:
att. name="nameDataObj/nameMessage/nameError"
(att. errorCode="codeError")
att. item/structureRef="idItemDefinition"
ITEMDEFINITION (idItemDefinition)
att. structureRef="nms:elem/msg/type"

BPEL VARIABLE: <variable name="nameDataObj"
element/messageType/type="nms:elem/msg/type"></variable>
WSDL MESSAGE:
targetNamespace=".." xmlns:nms="smth.wsdl"
<message name="nameMessage"></message>
```

Kod ispod prikazuje BPMN XML kod elemenata `operation` i `interface` koji se preslikavaju u WSDL elemente `operation` i `portType`.

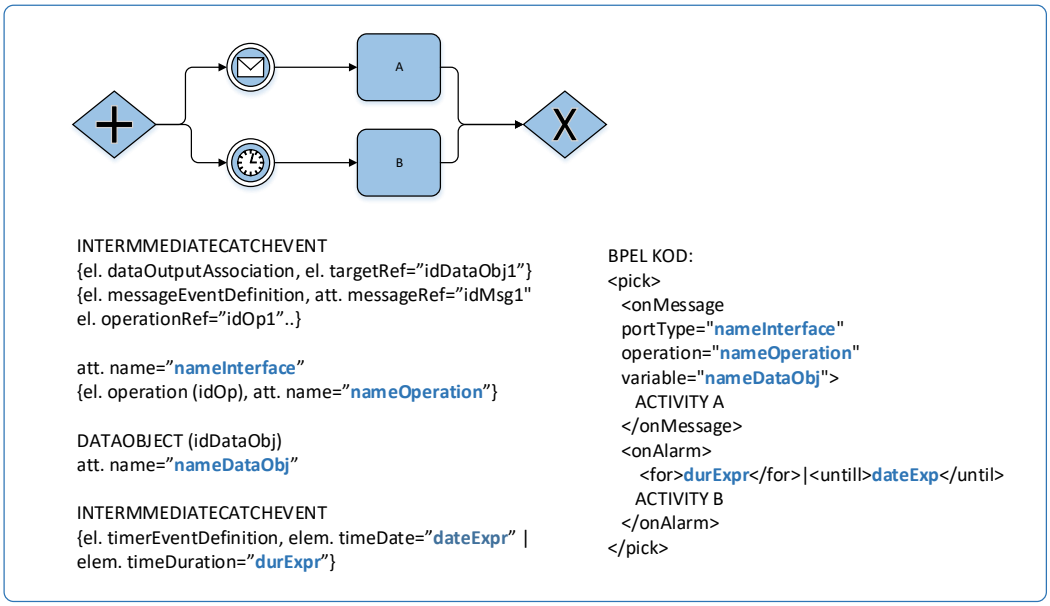
```
BPMN INTERFACE/OPERATION/MESSAGE:
att. name=”nameInterface”
att. implementationRef=”nms:nameInterface”
  {el. operation
  att. name=”nameOperation”
  att. implementationRef=”nms:nameOperation”
  el. in/out/errorMessageRef=”idMsg1/2/3”}
MESSAGE (idMsg1/2/3)
att. name=”nameMsg1/2/3”

WSDL PORTTYPE/OPERATION:
targetNamespace="smth.wsdl" xmlns:nms="smth.wsdl"
<portType name="nameInterface">
  <operation name="nameOperation">
  <input/output/fault message="nameMsg1/2/3"/></operation>
</portType>
```

Slijedom koraka algoritma preslikavanja BPMN-BPEL, nakon kreiranja BPEL varijabli i WSDL elemenata, kreće se s popunjavanjem BPEL aktivnosti `sequence` na način da se prvo kreiraju početne BPEL aktivnosti. Slijede upareni XML kodovi početnih BPEL aktivnosti i njihovih pripadnih BPEL vizualnih interpretacija. Na slici 16 naveden je upareni kod za BPMN čvor `startEvent` koji se preslikava u početnu BPEL aktivnost `receive`, dok u početne BPMN čvorove spada i čvor `receiveTask` (analogan čvoru `startEvent`) i vizualna BPEL aktivnost `pick` koja će biti naknadno objašnjena budući da ona ne mora nužno biti samo početna BPEL aktivnost.



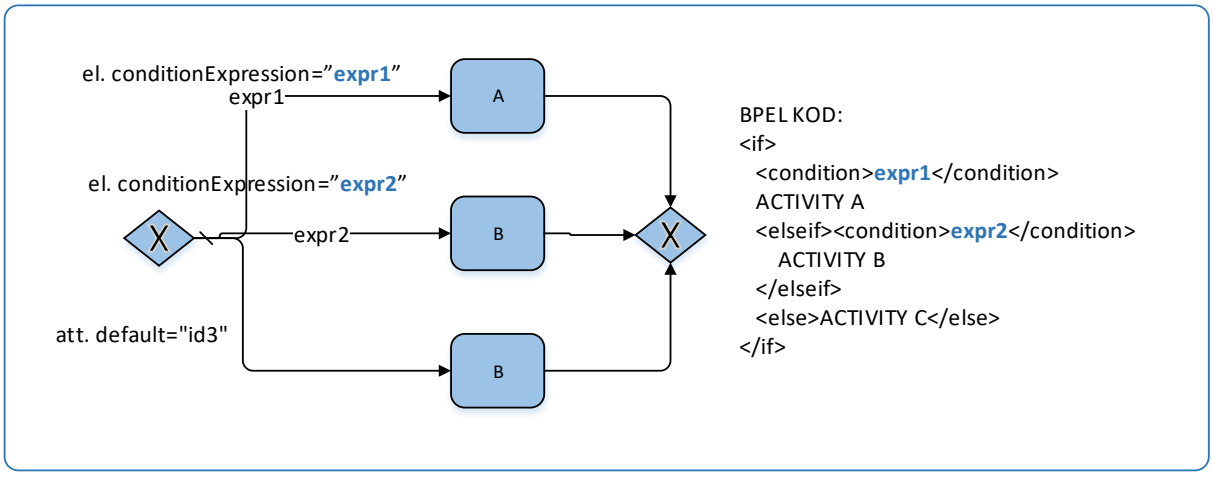
Slika 16: BPEL aktivnost `receive` *u BPMN*

Kao što je u Standard BPEL rečeno, BPEL posjeduje standardne aktivnosti kontrole tijeka izvršavanja procesa koje se susreću kod većine programskih jezika. Slijede upareni XML kodovi između spomenutih BPEL aktivnosti i njihovih BPMN vizualnih inačica. Na slici 17 je prikazana vizualna aktivnost `pick`, s tim da se umjesto čvora `intermmediateCatchEvent` može koristiti i čvor `receiveTask` te ukoliko je prisutan jedan događaj onda nema potrebe za čvorovima `*gateway`.



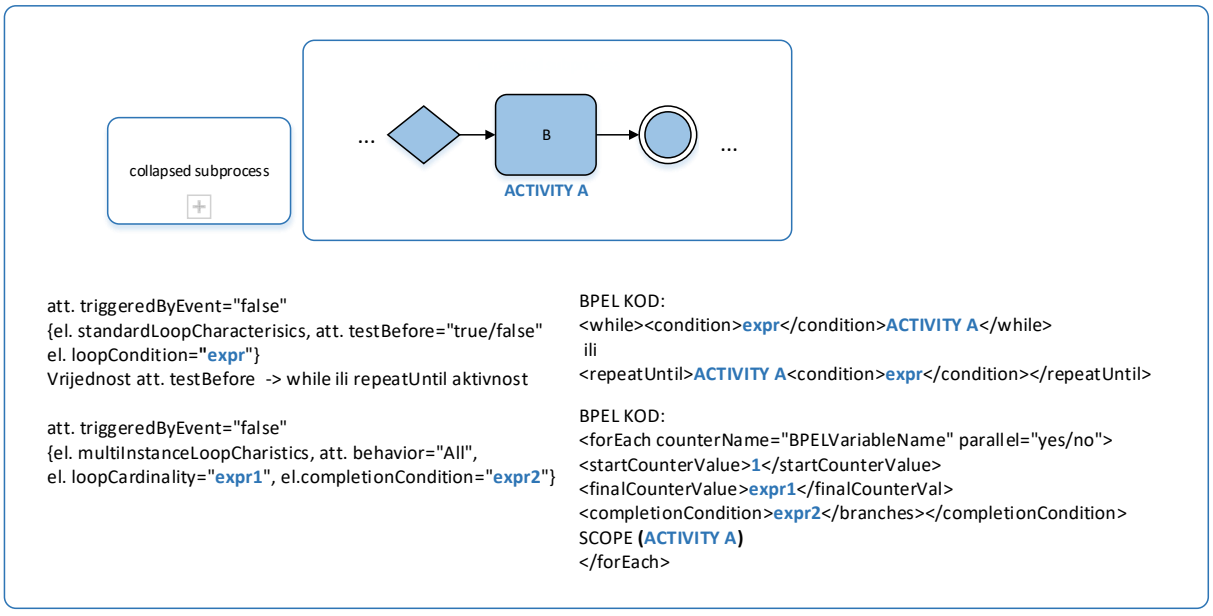
Slika 17: BPEL aktivnost `pick` *u BPMN*

BPEL aktivnost `if` vizualno se predstavlja na način predstavljen slikom 18. Početni element `exclusiveGateway` ima izlazne poveznice od kojih samo jedna smije biti neuvjetovana da bi se zadovoljilo pravilo BPEL aktivnosti `if` o samo jednom sadržanom elementu `else`.



Slika 18: BPEL aktivnosti `if` u BPMN

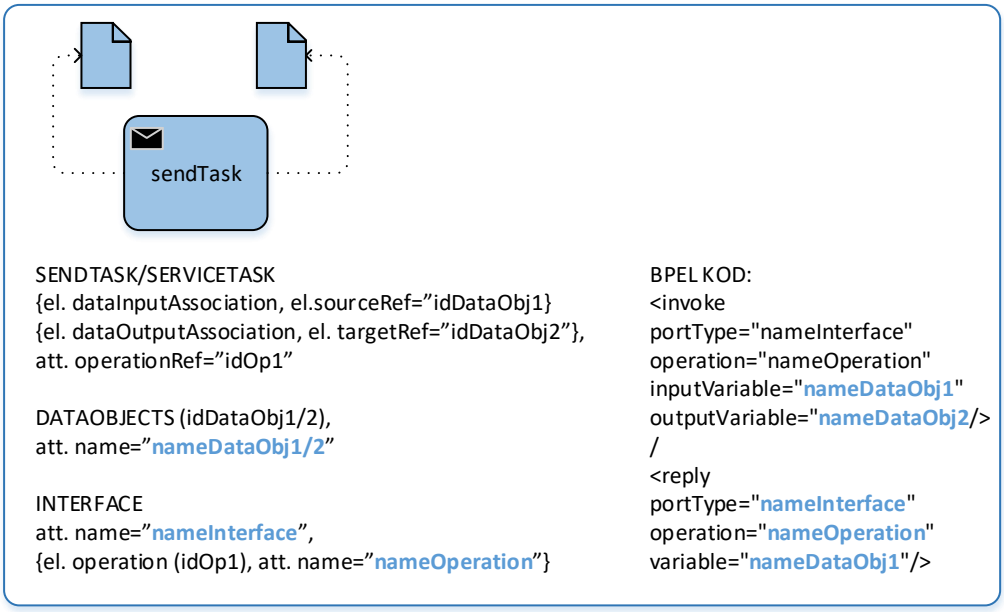
BPMN čvor `subProcess` se može iskoristiti za prikazivanje nekoliko BPEL aktivnosti (`while`, `repeatUntil`, `forEach`) pri čemu se razlikovanje postiže na temelju sadržanih elemenata `standardLoopCharacteristics` i `multiInstanceLoopCaracterostics` (slika 19).



Slika 19: BPEL aktivnosti predstavljene s BPMN `subProcess`

Slika 20 prikazuje BPEL aktivnost `invoke` za komunikaciju s partnerima i njenu BPMN vizualnu interpretaciju, što je u ovom slučaju BPMN čvor `sendTask`. Osim čvora `sendTask`, aktivnost `invoke` se može prikazati i kombinacijom slijednih čvorova `throwEvent` (`messageEventDefinition`) + `catchEvent` (`messageEventDefinition`)[1] ili čvorom `serviceTask` analogno slici 20.

BPEL aktivnost `reply` se može predstaviti BPMN čvorom `sendTask` i čvorom `throwEvent` analogno slici 20, dok se BPEL aktivnost `receive` može predstaviti BPMN čvorom `receiveTask` i čvorom `intermmediateCatchEvent` (`messageEventDefinition`) analogno slici 20.



Slika 20:BPEL aktivnost `invoke` *u BPMN*

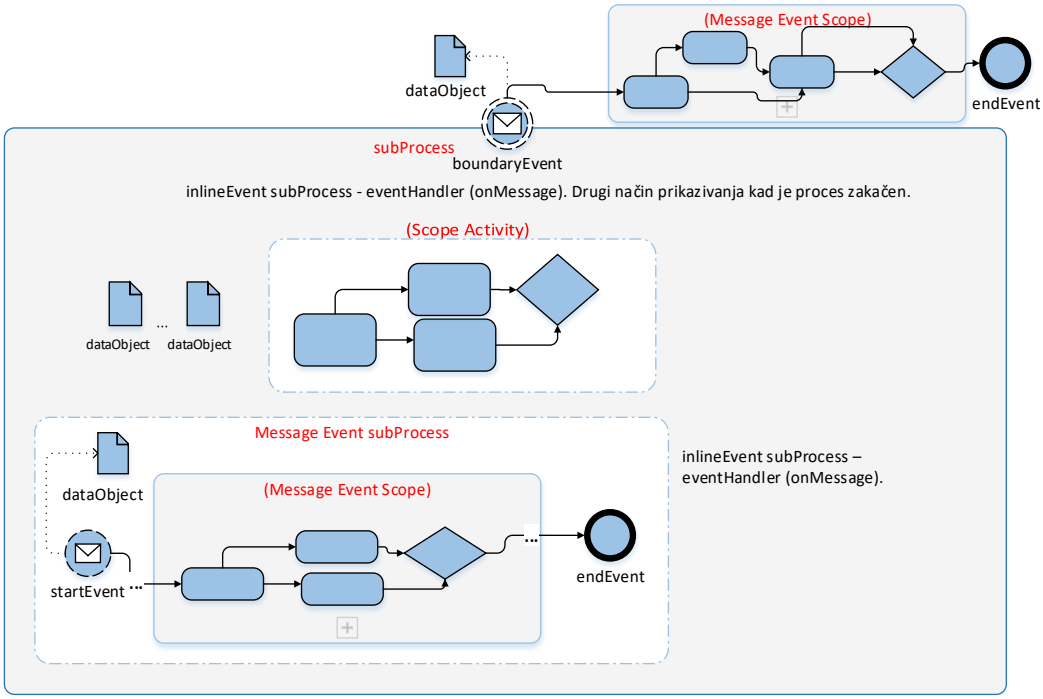
BPEL aktivnost `scope` se također predstavlja čvorom `subProcess` (koji ne sadržava elemente `standardLoopCharacteristics` i `multiInstanceLoopCaracterostics`). Pripadne aktivnosti `*Handler` se prikazuju korištenjem čvorova `inlineEvent subProcess` koji su kao djeca elementi sadržani unutar glavnog čvora `subProcess` pri čemu vrsta početnog čvora `startEvent` određuje o kojoj vrsti aktivnosti `*Handler` se radi. Elementi `inlineEvent subProcess` nisu dio

[1] Umjesto kombinacije `throwEvent(messageEventDefinition)+ catchEvent(messageEventDefinition)`, može se koristiti i kombinacija `throwEvent(messageEventDefinition)+receive`.

regularnog izvršavanja unutar elementa `process/subProcess`, to jest sadržani su kao djeca elementi i vizualno nisu niti s jednom aktivnošću povezani.

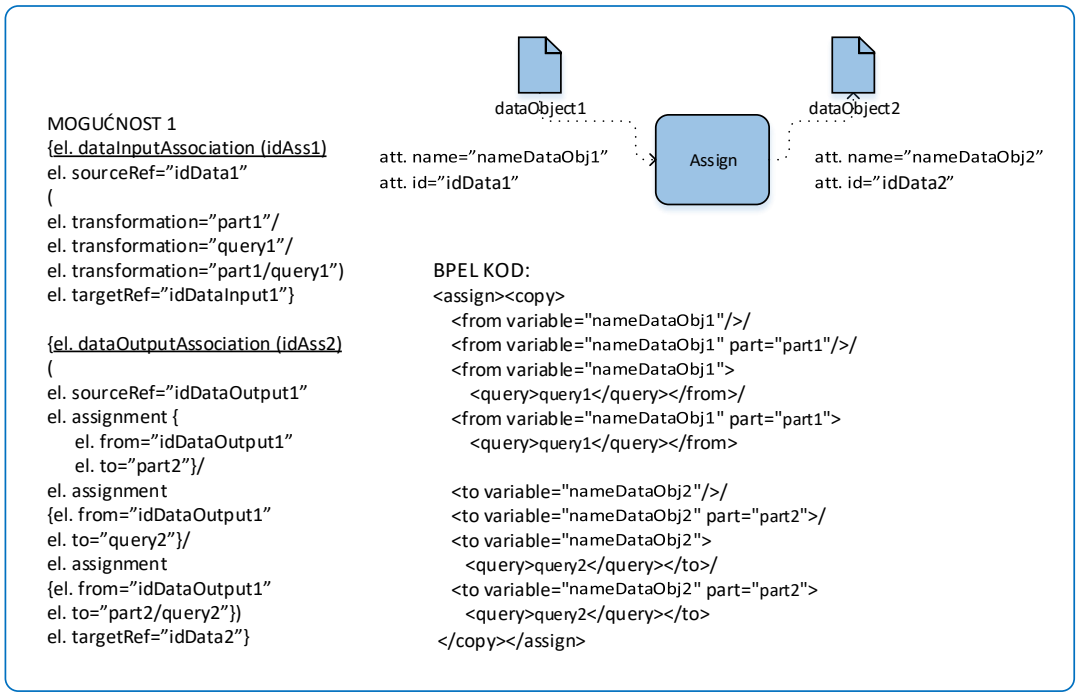
Na slici 21 vizualno je prikazana BPEL aktivnost `scope` s prikačenom aktivnosti `onMessage eventHandler`. BPEL aktivnost `scope` je prikazana pomoću BPEL elementa `subProcess`, dok je pridružena aktivnost `onMessage eventHandler` prikazana pomoću elementa `inlineEvent subProcess` kojoj je početni događaj `startEvent(messageEventDefinition)`. Vrste početnih čvorova `startEvent` mogu biti `timerEventDefinition`, `errorEventDefinition`, `compensateEventDefinition` i oni redom odgovaraju BPEL aktivnostima `onAlarm`, `errorHandler` i `compensationHandler *Handler`.

Aktivnost `compensateScope` jeste aktivnost koja se može pojaviti unutar aktivnosti `compensationHandler` ili `faultHandler` i ona služi za pozivanje aktivnosti `compensationHandler` prikačene za dijete aktivnost koja je oblika `scope`. U vizualnom modelu ona se predstavlja običnom aktivnošću `task` s naznačenim imenom „*callCompensation ChilScopeName*“.



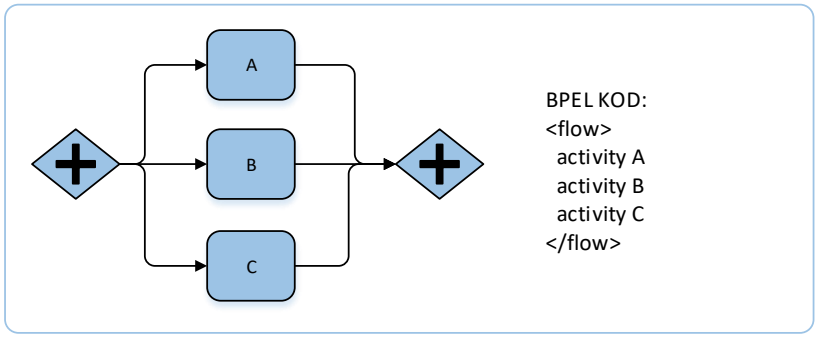
Slika 21: BPEL aktivnost `scope` *s* `onMessage` *u BPMN*

BPEL aktivnost `assign` se može predstaviti običnim čvorom `task` čiji oblici `dataInputAssociation` i `dataOutputAssociation` određuju vrstu elemenata `from/to` aktivnosti `assign`. U obzir nije uzet obzir oblik `from/to` kojim se dodjeljuje vrijednost konstrukciji `partnerLink` budući da ista nije obuhvaćena razinom vizualnog modeliranja.



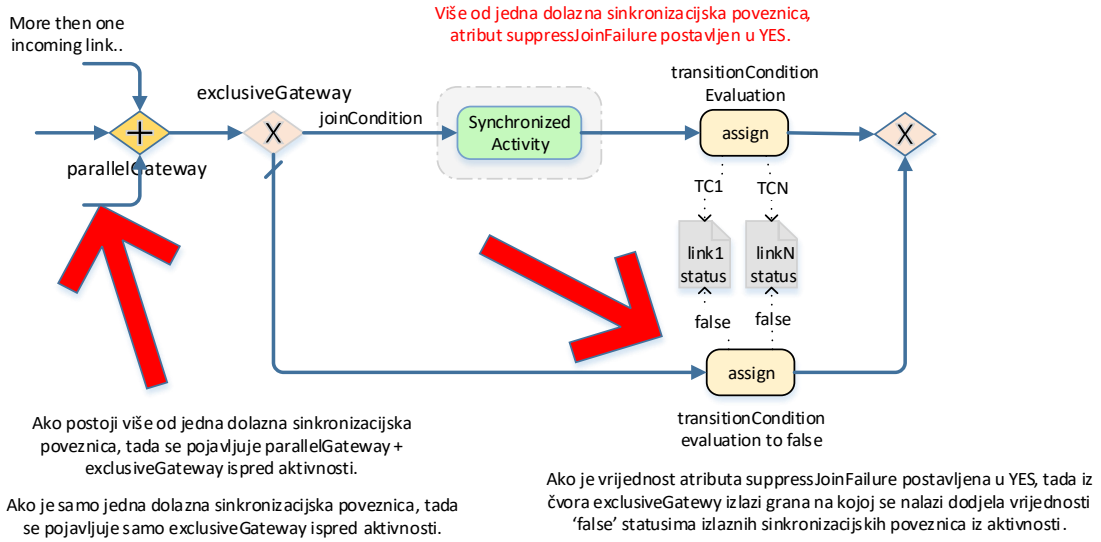
Slika 22: BPEL aktivnost `assign` u BPMN

Ponašanje aktivnosti `flow`, koja nudi paralelizam i sinkronizirano izvršavanje sadržanih aktivnosti, objašnjeno je detaljnije u Standard BPEL. Na slikama 24 do 27 prikazani su načini na koji se mogu prikazivati sinkronizacijski međuovisne aktivnosti unutar BPEL aktivnosti `flow`. Sva paralelna ili sinkronizacijski međuovisna izvršavanja sadržana unutar aktivnosti `flow` u BPMN kodu su ugniježđena između dva elementa `parallelGateway`. Osnovni slučaj izvršavanja unutar aktivnosti `flow` je kada su sadržane aktivnosti paralelne i one se u BPMN kodu predstavljaju sukladno slici 23. Pripadno osnovno izvršavanje se automatski preslikava u BPEL korištenjem algoritma preslikavanja BPMN-BPEL dok se sva sinkronizirana izvršavanja unutar aktivnosti `flow` u BPEL kod preslikavaju ručno na temelju definiranih pravila vizualnog prikaza na slikama 24 do 27.



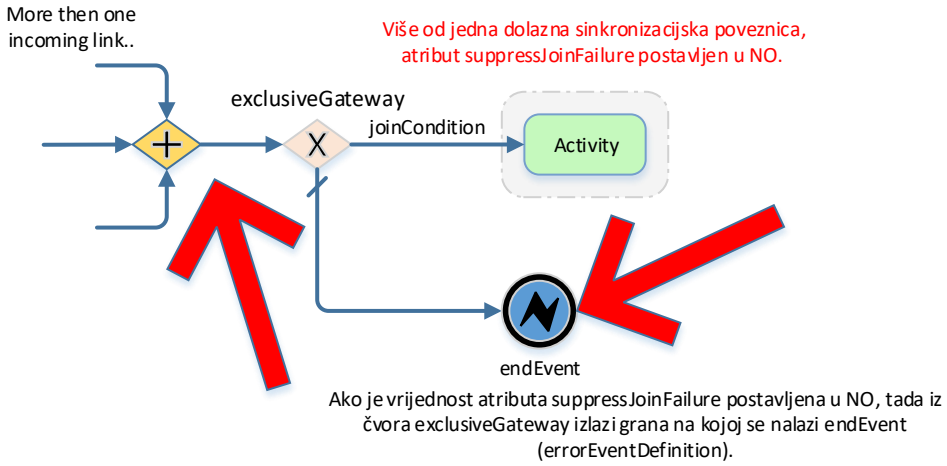
Slika 23: Osnovno izvršavanje BPEL aktivnosti `flow` *u BPMN*

Čvor `ParallelGateway` ispred sinkronizirane aktivnosti na slici 24 služi da statusi svih dolaznih poveznica `link` budu poznati jer sinkronizirana aktivnost tek tada može početi s ispitivanjem uvjeta izvršenja. Čvor `exclusiveGateway` ispred sinkronizirane aktivnosti služi za mogućnost prikazivanja izvršavanja ukoliko uvjet izvršenja ne bude istinit čime statusi svih izlaznih poveznica `link` koje izlaze iz sinkronizirane aktivnosti poprimaju vrijednost `false`.



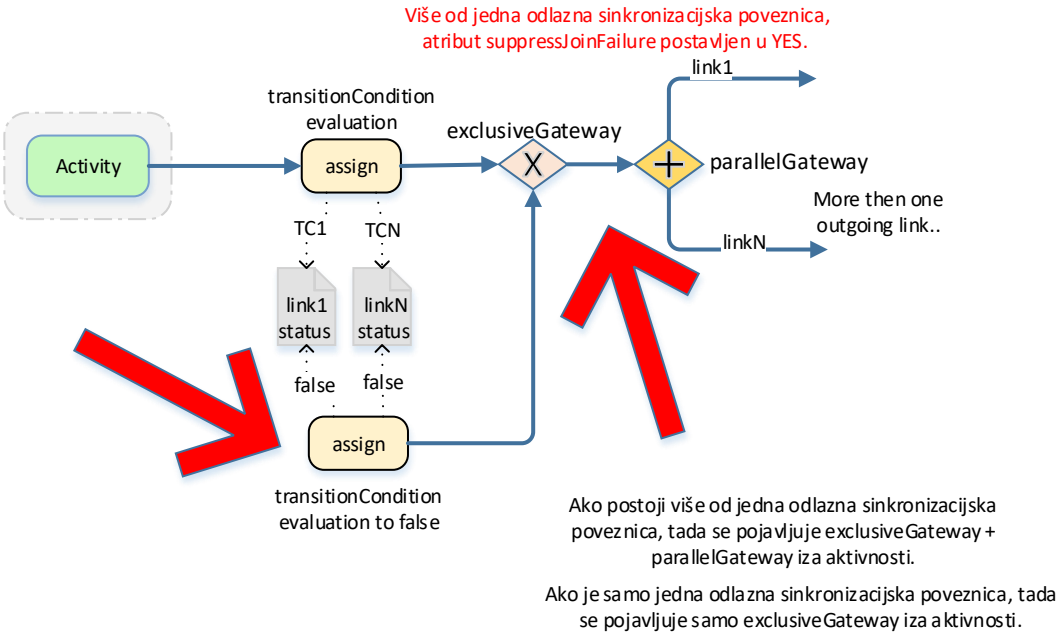
Slika 24: Sinkronizirana aktivnost unutar aktivnosti `flow` *prva mogućnost*

Slučaj sinkronizirane aktivnosti na slici 25 je identičan slučaju na slici 24 s tom razlikom da iz čvora `exclusiveGateway` podrazumijevana grana vodi prema završetku, to jest čvoru `endEvent` kojim se izbacuje pogreška `suppressJoinFailure` o neispunjenju uvjeta izvršenja.



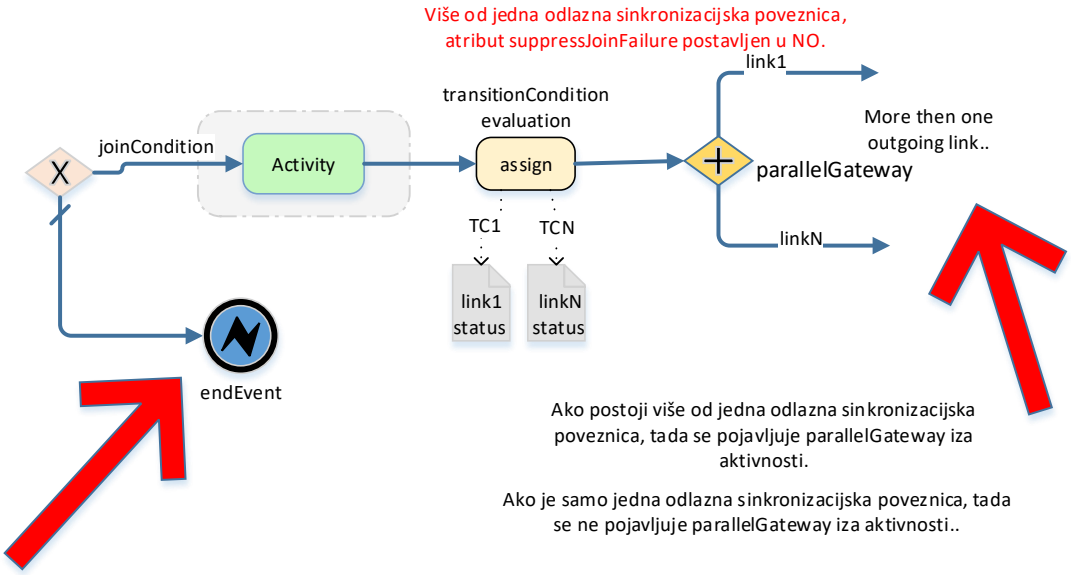
Slika 25: Sinkronizirana aktivnost unutar aktivnosti `flow` *druga mogućnost*

Čvor `exclusiveGateway` iza sinkronizirane aktivnosti na slici 26 služi da se pričekaju međusobno isključujuće grane koje dolaze od različitih postavljanja statusa poveznica `link` koje izlaze iz sinkronizirane aktivnosti. Čvor `parallelGateway` koji potom slijedi služi da se sinkronizirana aktivnost spoji sa svim drugim aktivnostima s kojima je ista povezana putem njenih izlaznih poveznica `link`.



Slika 26: Sinkronizirana aktivnost unutar aktivnosti `flow` *treća mogućnost*

Ukoliko sinkronizirana aktivnost ima atribut `suppressJoinFailure` postavljen u vrijednost `true`, tada iza sinkronizirane aktivnosti slijedi samo čvor `parallelGateway` (slika 27) koji ima istu ulogu kao čvor na slici 26.



Slika 27: Sinkronizirana aktivnost unutar aktivnosti `flow`
četvrta mogućnost

4.1.2. Pseudokod algoritma preslikavanja BPMN-BPEL

Ovo poglavlje predstavlja pseudokod funkcija algoritma preslikavanja BPMN-BPEL. Za tumačenje pseudokoda, potrebno je obratiti pozornost na donju notaciju kojom je pseudokod opisan. Pseudokod opisuje funkcije koje implementiraju preslikavanja uparenih BPMN i BPEL XML kodova iz prethodnog poglavlja 4.1.

Notacija kojom je opisan pseudokod je sljedeća:

- `A` - XML element A (BPMN ili BPEL izvršavanja) se predstavlja kao skup,
- `A(x)` – svi elementi skupa A, a to mogu biti ili XML djeca elementi ili XML atributi,
- `A = {x | P(x)}` – elementi skupa A koji imaju svojstvo P,
- `A = {x | E(x)}` – elementi skupa A koji su XML elementi, (E - "element svojstvo"),
- `A = {x | R(x)}` – elementi skupa A koji su XML atributi (R – "atribut svojstvo"),
- `A.read/create(„name“)` – čitavanje/kreiranje skupa A imena “name”,
- `A.add(E(x))` – dodavanje prethodno kreiranog elementa x (XML element) skupu A,
- `A.add(E(„value“))` – dodavanje elementa x (XML element) skupu A,
- `A.add(R(„name“,“value“))` – dodavanje elementa x (XML atribut) skupu A,
- `A.E/R(„name“)` - dohvaćanje vrijednosti djeteta (XML elementa/atributa) imena “name”,
- `A.value` - postavljanje vrijednosti skupa A (dodavanje „XML Text Node“),
- `A.select(E(„name“))` – izbor djeteta (XML elementa) imena “name”,
- `forEvery{x€A(x)}:action` – izvrši akciju za svaki element x skupa A,
- `forEvery{x€A(x)|P(x)}:action` – izvrši akciju za svaki element x skupa A koji zadovoljava svojstvo P(x),
- `A.search(a€A|P(x),C)` – pretraži skup A za elementima koji zadovoljavaju svojstvo P(x) i rezultate spremi u skup C.

Funkcije algoritma preslikavanja BPMN-BPEL:

- `toBPELprocess/scopeDraw():`
    - popunjava glavnu BPEL aktivnost `sequence` unutar BPEL `process` ili `scope` aktivnosti. Ona poziva funkcije `seqFlowIdFromVisualActivity(*)` radi dohvaćanja elementa

sequenceFlow koji izlazi iz vizualnih BPEL aktivnosti i funkcije toBPELactivity(*)Draw za generiranje pripadnih BPEL aktivnosti kojima se popunjava glavna BPEL aktivnost sequence. Osim popunjavanja aktivnosti sequence, ova funkcija kreira varijable i aktivnosti *Handler koji pripadaju aktivnostima process ili scope,

- toBPELactivity(*)Draw(X,..,A):
  - funkcija koja na temelju ulaznog parametra X generira pripadnu BPEL aktivnost (*) i smješta je kao dijete unutar pripadajuće BPEL aktivnosti A gdje je X prvi čvor BPEL vizualne aktivnosti,
- seqFlowIdFromVisualActivity(*)(X):
  - funkcija koja na temelju ulaznog parametra X vraća id atribut sequenceFlow elementa koji izlazi iz BPEL vizualne aktivnosti (*) gdje je X prvi čvor BPEL vizualne aktivnosti.

Funkcije toBPELactivity(*)Draw ili seqFlowIdFromVisualActivity(*) za svaku pojedinu BPEL aktivnost predstavljenu jednim BPMN čvorom nisu objašnjavanje na razini pseudokoda jer su u Algoritam preslikavanja BPMN-BPEL prikazana preslikavanja između pripadnih XML kodova i smatra se da su ona dovoljna za razumijevanje.

Pseudokodovi funkcija toBPELactivity(*)Draw i seqFlowIdFromVisualActivity(*) vezanih za vizualnu BPEL aktivnost if su objašnjeni budući da je if strukturna aktivnost[2] koja posjeduje grane izvršavanja i kompleksnost koju ona uvodi vezana je za ugnježđivanje jedne strukturne aktivnosti unutar druge do proizvoljnih dubina, a tu je i problem otkrivanja aktivnosti sequence na granama (slika 28) budući da se na granama BPEL aktivnosti if može slijedno nalaziti jedna ali i više aktivnosti do proizvoljnog broja tako da je potrebno omogućiti prepoznavanje bilo kojeg slijednog broja aktivnosti na granama i smjestiti ih u pripadnu aktivnost sequence. Istovjetnu kompleksnost je nametnula BPEL aktivnost pick. Pseudokodovi funkcija vezanih za BPEL aktivnost pick nisu navedeni budući da između if i pick funkcija postoji analogija, to jest u obzir treba uzeti samo različitu strukturu vizualnih BPEL aktivnosti if i pick (Slika 17: BPEL aktivnost pick u BPMN i Slika 18: BPEL aktivnosti if u BPMN).

[2] Strukturnom aktivnošću smatra se bilo koja BPEL vizualna aktivnost koja sadržava grane i predstavljena je sa više BPMN čvorova (if, pick, flow).

Funkcije preslikavanja vizualne BPEL aktivnosti `if`:

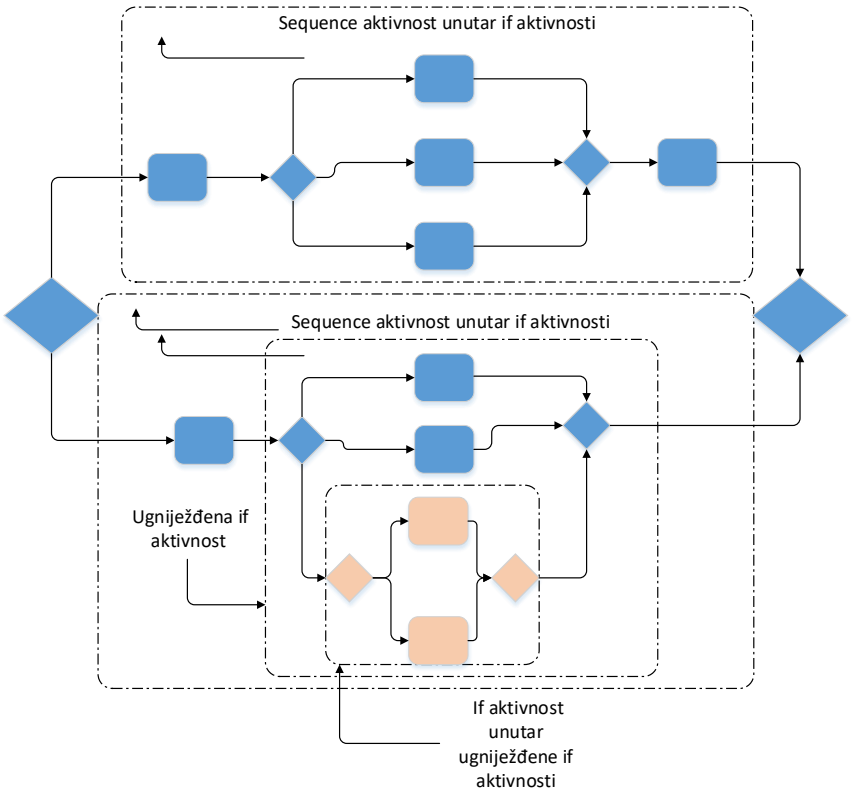
- `seqFlowIdFromIf(X)`:
  - funkcija koja vraća atribut id izlazne `sequenceFlow` poveznice iz vizualne BPEL aktivnosti `if`

    (X = početni `exclusiveGateway` čvor vizualne BPEL aktivnosti `if`),
- `toBPELifDraw(X,G,B)`:
  - funkcija koja generira BPEL aktivnost `if` i dodaje je kao dijete element roditeljskoj BPEL aktivnosti koja je funkciji proslijeđena kao parametar pri čemu roditeljska BPEL aktivnost može biti aktivnost `sequence` unutar aktivnosti `process` ili `scope` ili neka od strukturnih aktivnosti (`if`, `pick` ili `flow`)

    (X = početni `exclusiveGateway` čvor vizualne BPEL aktivnosti `if`,

    G = krajnji `exclusiveGateway` čvor vizualne BPEL aktivnosti `if`,

    B = BPEL aktivnost kojoj se kao dijete element dodaje generirana BPEL aktivnost `if`).



Slika 28: Ugniježđene vizualne aktivnosti `if` *i aktivnost* `sequence` *na aktivnosti* `if`

Slijedi pseudokod funkcije `toBPELprocess/scopeDraw` za popunjavanje BPEL aktivnosti `process` ili `scope`. U dijelovima pseudokoda navedeni su komentari (//) radi pojašnjenja značenja koda.

Tablica 1: Funkcija `toBPELprocess/scopeDraw`

```
toBPELprocess/scopeDraw()

A.create("process"), B.create("sequence");

C.read("BPMNProcess");

string nextSequenceNodeId = „";

//pamti izlaznu poveznicu zadnje otkrivene vizualne BPEL aktivnosti

A.add(E(„variables"));
forEvery{x € C | E(x).type == "dataObject"} :
           A.E(variables).Add(variableDraw(x));
           //pretvaranje: BPMN dataObject-BPEL variable

forEvery{x € C | E(x).isStartCandidate} :

           nextSequenceNodeId = seqFlowIdFromVisualActivity(x,..);

           toStartBPELactivityDraw(x,..,A);

           //pretvaranje: BPMN kod-početne BPEL aktivnosti

while nextSequenceNodeId != "none"

           forEvery{x € C | E(x)} :

               if E(x).E(incoming) == nextSequenceNode

                   nextSequenceNode = seqFlowIdFromVisualActivity(x,..);

                   toBPELactivityDraw(x,..,B);

                   //pretvaranje: BPMN kod-pripadne BPEL aktivnosti

A.add(E(B));
```

Funkcija `toBPELprocess/scopeDraw` prikazana u tablici 1 preslikava otkrivene BPEL vizualne aktivnosti u pripadni BPEL kod i dodaje ih kao djecu elemente unutar BPEL aktivnosti `sequence` pri čemu varijablu `nextSequenceNodeId` koristi za pamćenje atributa `id` izlazne poveznice `sequenceFlow` iz zadnje otkrivene BPEL vizualne aktivnosti. Pamćenje izlazne poveznice `sequenceFlow` je potrebno radi pravilnog dohvaćanja slijedeće vizualne BPEL aktivnosti.

Osim popunjavanja slijeda `sequence`, funkcija `toBPELprocess/scopeDraw` vrši i preslikavanje BPMN čvorova `dataObject` u BPEL elemente `variable`.

Ukoliko je za BPEL aktivnost `process` ili za BPEL `scope` prikačena bilo kakva aktivnost `*Handler`, ona se u BPMN kodu predstavlja dijete elementom `inlineEvent subProcess` (atribut `triggeredByEvent="true"` i nije dio standardnog tijeka izvršavanja) na temelju čijeg početnog događaja se određuje vrsta BPEL aktivnosti `*Handler`. Donji pseudokod demonstrira pronalazak aktivnosti `*Handler` unutar BPMN `process` ili `subProcess` elementa. Donji pseudokod je dio funkcije `toBPELprocess/scopeDraw` ukoliko postoje aktivnosti `*Handler`. Naknadno je naveden budući da nije navođen u opisu funkcije u tablici 1.

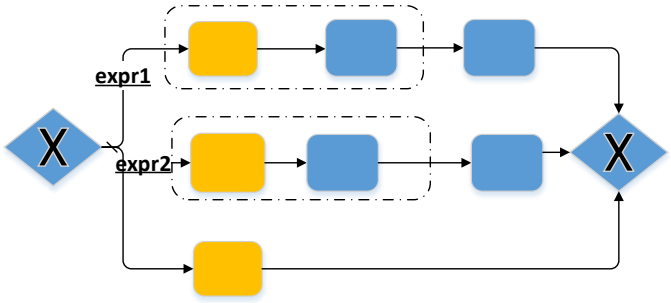
```
F.create("event/compensation/faultHandler(s)");
forEvery{x ∈ C | E(x).type=="subProcess" &&
E(x).R("triggeredByEvent")="true" && !E(x).E("incoming").exists} :
      forEvery{x1 ∈ x | E(x1).type == "startEvent"} :
            forEvery{x2 ∈ x1 | E(x2).type ==
            "message/timer/compensation/errorEventDefinition"} :
      toBPELonMessage/onAlarm/compensation/errorDraw(x,..,F);
A.add(F);
```

Slijedi pseudokod funkcije `seqFlowIdFromIf` koja vraća atribut `id` izlazne poveznice `sequenceFlow` iz vizualne BPEL aktivnosti `if`. Dijelovi pseudokoda koji su u tablici 2 opisani tekstualno će se detaljnije objasniti kroz izdvojene dijelove koda i slike radi njihovog lakšeg razumijevanja.

Tablica 2: Funkcija `seqFlowIdFromIf`

```
seqFlowIdFromIf(X)

X – početni element exclusiveGateway

M.read("BPMNProcess");
A.create("outgoingsInExcGateway");

forEvery {x € X | E(x).type == outgoing} : A.add(x);
//djeca elementi outgoing početnog elementa exclusiveGateway

B.create("firstElementsOnBranches");
//prvi elementi na granama iza početnog elementa exclusiveGateway
kod (1)

E.create("outgoingsInExecutions");
// djeca elementi outgoing prve vizualne aktivnosti na granama
F.create("nextElements");
//elementi koji slijede iza početne vizualne aktivnosti na granama

//pronalazak slijednog elementa svake prve vizualne aktivnosti na granama
kod (2)

bool flag=false;
//značkica da li je pronađen zadnji element exclusiveGateway
check:
//otkrivanje zadnjeg elementa exclusiveGateway ukoliko se slijedni element
dviju početnih vizualnih aktivnosti na granama poklapa (članovi skupa F)
kod (3)

//na svim granama if aktivnosti nalazi se sequence aktivnost->
dodatni prolazak kroz izvršavanja na granama do pronalaska zadnjeg elementa
exclusiveGateway
kod (4)

string outgoingFromBPELIf;
//dohvaćanje atributa id elementa sequenceFlow
koji izlazi iz zadnjeg elementa exclusiveGateway
kod (5)
return outgoingFromBPELIf;
```

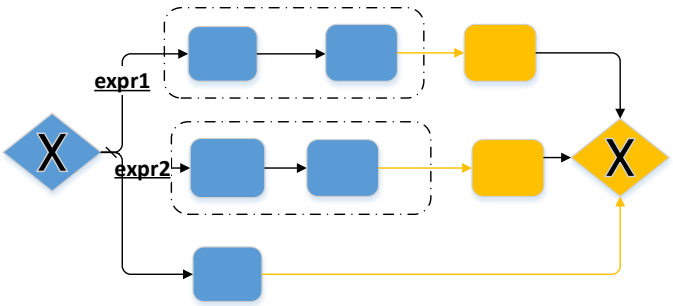
Dijelovi funkcije `seqFlowIdFromIf`, u tablici 2 kratko objašnjeni komentarima i numerički označeni, će biti prikazani i objašnjeni u donjim izlaganjima. Dio funkcije numeriran s (1) u skup B dodaje prve elemente na granama vizualne aktivnosti `if` (slika 29).



Slika 29: Vizualno - `firstElementsOnBranches`

```
B.create("firstElementsOnBranches");
forEvery {a € A | E(a)} :
      M.search(m € M | E(m).R(id) == a.value, C);
      M.search(m € M | E(m).R(id) == C.R(targetRef), D);
      B.add(D);
//prvi elementi na granama iza početnog elementa exclusiveGateway
```

Dio funkcije `seqFlowIdFromIf` numeriran s (2) u skup F dodaje čvorove koji slijede iza prve vizualne aktivnosti na granama (slika 30).



Slika 30: Vizualno - `nextElements`

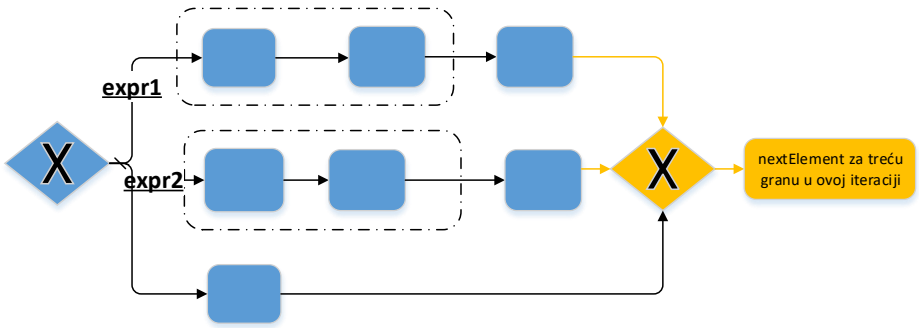
```
F.create("nextElements");
forEvery {b € B | E(b)} : (****)
      if E(b).type == "eventBased/exclusive/parallelGateway" ->
        //strukturne vizualne aktivnosti na granama
              M.search(m € M | E(m).R(id) ==
                seqFlowIdFromPick/If/Flow(b), G);
      else -> M.search(m € M | E(m).R(id) == b.E(outgoing), G);
```

```
        //vizualne aktivnosti na granama predstavljene jednim cvorom
              F.Add(E(*BPMNnodeFollowing(G)));
               //sljedeći BPMN čvor iza prve vizualne aktivnosti
```

Dio funkcije `seqFlowIdFromIf` numeriran s (3) uspoređuje slijedne čvorove iza prvih vizualnih aktivnosti na granama (članove skupa F) te ukoliko se pronađu dva identična čvora, dolazi se do zaključka da je pronađen zadnji element `exclusiveGateway` što nije slučaj za primjer sa slike 30.

```
while (flag==false) :
      forEvery {e € F | E(e)} :
           forEvery {e1 € F | E(e1)} :
              if (e==e1) ->
                  flag=true;
                    G.create("endingExclusiveGateway");
                       G=e; goto next;
           //otkriven zadnji element exclusiveGateway
           //slijedni čvor aktivnosti na dvije grane se poklapa
```

Dio funkcije `seqFlowIdFromIf` numeriran s (4) se izvršava ukoliko nisu pronađena dva ista slijedna čvora iza prvih vizualnih aktivnosti na granama, to jest nije pronađen zadnji element `exclusiveGateway`. Ovaj dio koda pronalazi slijedne vizualne aktivnosti iza prethodno pronađenih vizualnih aktivnosti na granama i čvorove koji slijede neposredno iza novih pronađenih vizualnih aktivnosti kako bi se izvršavanje vratilo ponovno na (3) s ciljem pronalaska zadnjeg elementa `exclusiveGateway`. Izvršavanje (3) bi u prezentiranom slučaju (slika 31) pronašlo zadnji element `exclusiveGateway` budući da je to zajednički slijedni čvor iza vizualnih aktivnosti na prvoj i drugoj grani.

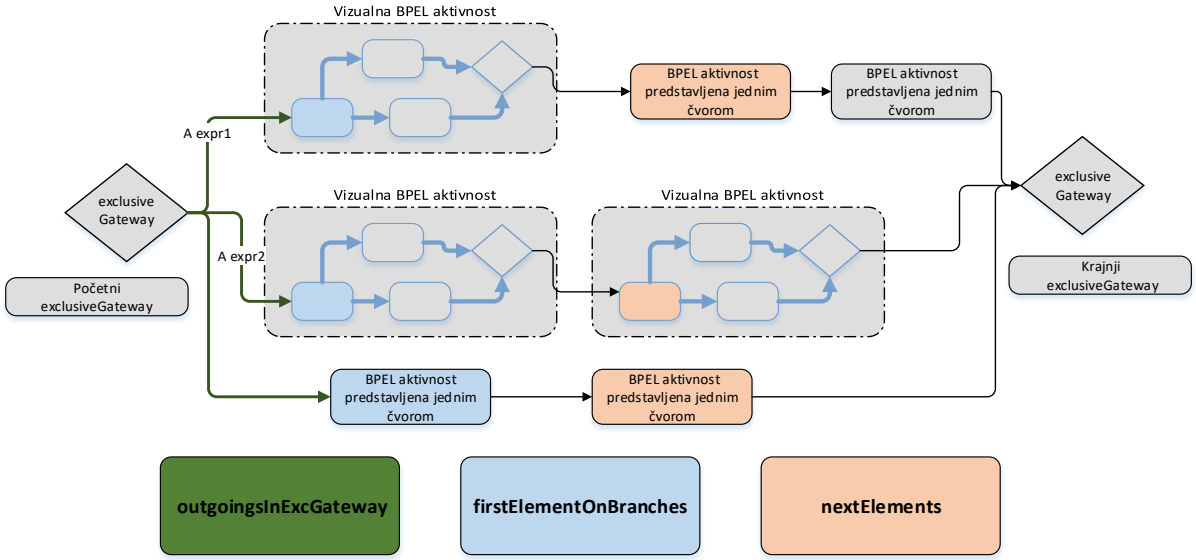


Slika 31: Vizualno - `nextElements` *nakon izvršavanja (4)*

```
if (flag==false) ->
      F1.create("nextElements");
      forEvery {e € F1 | E(e)} :
      (****)analogy(E(b) replacedWith E(e), F replacedWith F1)
      if E(e).type == "eventBased/exclusive/parallelGateway" ->
            //strukturne vizualne aktivnosti na granama
                  M.search(m € M | E(m).R(id) ==
      seqFlowIdFromPick/If/Flow(e), G);
      else -> M.search(m € M | E(m).R(id) == e.E(outgoing), G);
      //vizualne aktivnosti na granama predstavljene jednim čvorom
            F1.Add(E(*BPMNnodeFollowing(G)));
      F=F1;
      goto check;
```

Dio funkcije `seqFlowIdFromIf` numeriran s (5) se izvršava nakon što je pronađen zadnji element `exclusiveGateway` i on vraća atribut `id` njegovog izlaznog elementa `sequenceFlow`.

```
if G.E(outgoing).exists ->
      G.search(m € M | E(m).R(id) == G.E(outgoing), TEMP);
      outgoingFromBPELIf = TEMP(R(„id");
else -> outgoingFromBPELIf = "none";
```



Slika 32: Označavanje elemenata unutar funkcije `seqFlowIdFromIf`

Slijedi pseudokod funkcije `toBPELifDraw` koja generira BPEL aktivnost `if` i dodaje je kao dijete element roditeljskoj BPEL aktivnosti koja je funkciji proslijeđena kao parametar.

Tablica 3: Funkcija `toBPELifDraw`

```
void toBPELifDraw(X, G, B)
X – početni exclusiveGateway element,
G – krajnji exclusiveGateway element,
B - roditeljska BPEL aktivnost unutar koje se dodaje aktivnost if

C.create("if");
M.read(„BPMNProcess");//BPMN element process

D.create("sequenceFlows");
forEvery {a € A | E(a)} :
      M.search(m € M | E(m).R(id) == X.E(outgoing), D);
//pronalazak sequenceFlow koji izlaze iz početnog exclusiveGateway

forEvery {d € D| E(d)} :
      if d.E(conditionExpression).exists ->****
            if counter==0 //prva uvjetna aktivnost
                C.add(E(condition, d.E(conditionExpression));
            elseif counter>0 //naredne uvjetne aktivnosti na granama
                C1.create(„elseif");
                C1.add(E(condition, d.E(conditionExpression));
            E.create("firstNodeOnFirstBranch");
            M.search(m € M | E(m).E(incoming) == d.R(id), E);
            //prvi element na grani
            if(nextElementAfterVisualActivity(E) == G) ->***
            //jedna aktivnost na grani – nema slijeda sequence
                  //generiranje sadržane jedne BPEL aktivnosti
                  //na temelju BPMN izvršavanja na grani: kod (1)
            else ->
            //na grani je niz aktivnosti koji se smješta u sequence
                   //generiranje sadržanih BPEL aktivnosti
                     na temelju slijednih BPMN izvršavanja na grani
                     koji se smještaju u BPEL aktivnost sequence: kod (2)

            else ->
            //else grana – analogija kao s uvjetnim granama (****)
B.add(C); //dodavanje generirane aktivnosti if roditeljskoj BPEL aktivnosti
```

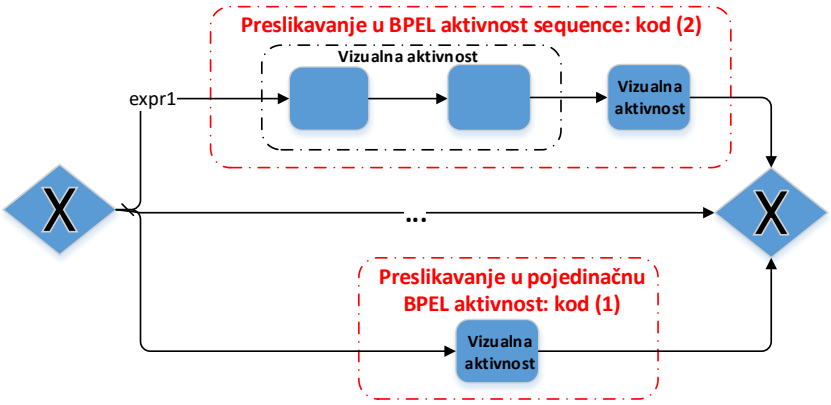
Dijelovi funkcije `toBPELifDraw`, u tablici 3 kratko objašnjeni komentarima i numerički označeni, će biti prikazani i objašnjeni u donjim izlaganjima. Dio funkcije numeriran s (1) generira sadržanu BPEL aktivnost i smješta je unutar prve uvjetne grane ili unutar neke od narednih grana aktivnosti `if`.

```
if E == "eventBased/exclusive/parallelGateway" ->
//ugniježđene aktivnosti pick/if/flow na granama
     M.search(m € M | E(m).R(id) == seqFlowIdFromIf/Pick/Flow(E),
TEMP);
     G1=*BPELactivityFollowing(TEMP);
     toBPELpick/if/flowDraw(E,G1,C/C1);
     //C/C1-ovisno o counter vrijednosti
else -> //ostale BPEL aktivnosti
     toBPELactivity*Draw(E,..,C/C1);
C.add(C1);//ukoliko counter>0
```

Dio funkcije numeriran s (2) generira slijedne sadržane BPEL aktivnosti na granama i smješta ih unutar BPEL aktivnosti `sequence`, a koja se pak smješta unutar prve uvjetne grane ili unutar neke od narednih grana aktivnosti `if`. Kod (2) se dakle izvršava ukoliko postoji slijed od nekoliko vizualnih aktivnosti na grani, za razliku od koda (1) koji se izvršava kada je na grani prisutna samo jedna vizualna aktivnost.

```
S.create("sequence");
while(nextElementAfterVisualActivity(E)!=G) ->
    analogija s kodom(1) (C/C1 replacedWith S)
    E=nextElementAfterVisualActivity(E);
C/C1.Add(S);//C/C1-ovisno o counter vrijednosti
C.add(C1);//ukoliko counter>0
```

Slika 33 demonstrira razliku između slučajeva kada se izvršava dio funkcije `toBPELifDraw` (1) i (2).



Slika 33: Izvršavanje dijelova (1) i (2)

4.2. Algoritam preslikavanja BPEL-BPMN

Algoritam preslikavanja BPEL-BPMN iz BPEL smjera u BPMN smjer se zbog blokovske strukture BPEL procesa izvršava propadajući kroz blokove počevši od glavnog bloka `process`.

Struktura BPEL procesa je sljedeća:

```
<process>..<variables><variable1/>..<variableN/></variables>
  <partnerLinks>..</partnerLinks>[3]
  <faultHandlers><catch faultName=".."></catch>
  ..<catchAll></catchAll></faultHandlers>
  <sequence><startBpelActivity></startBpelActivity>
  ..other BPEL activities..</sequence>
</process>
```

4.2.1. Koraci preslikavanja

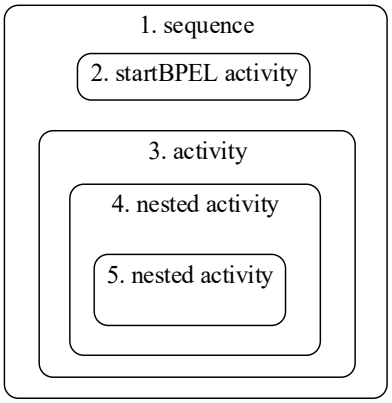
Algoritam preslikavanja BPEL-BPMN se izvršava kroz sljedeće korake.

1) Iz pripadnih WSDL sučelja procesa i partnera izvršavaju se preslikavanja poruka, operacija i priključaka u pripadne BPMN elemente (BPMN `message`, `operation`, `interface`) sukladno opisu u Algoritam preslikavanja BPMN-BPEL koji se kao djeca elementi smještaju unutar BPMN elementa `definitions`. Potom se vrši preslikavanje djece elemenata `fault/eventHandlers` BPEL elementa

[3] Konstrukcije `partnerLink` kao konstrukcije koje opisuju komunikaciju s partnerima nisu uključene u vizualni model i one se ručno dodaju u BPEL kod.

`process` u `inlineEvent` `subProcess` (`startEvent` `error/message/timeEventDefinition`) analogno slici 21 koji se kao dijete element smješta unutar BPMN elementa `process`. Preslikavanje BPEL elemenata `variable` u BPMN elemente `dataObject`, koji se kao djeca elementi smještaju unutar BPEL elementa `process`, također se vrši u ovoj fazi.

2) Drugi korak vezan je za preslikavanje izvršavanja unutar glavne BPEL aktivnosti `sequence`. Preslikavanje se vrši prolaskom kroz djecu elemente slijedno kako se nalaze u dokumentu budući da je poredak bitan (slika 34). Nakon preslikavanja početne BPEL aktivnosti u pripadajući BPMN kod, pamti se atribut `id` izlaznog BPMN elementa `sequenceFlow` kako bi se kod povezao sa slijedećim BPMN izvršavanjem generiranim na temelju sljedeće BPEL aktivnosti. Postupak se ponavlja sve dok se ne izvrši kompletno preslikavanje. Preslikavanje BPEL aktivnosti u pripadni BPMN kod izvršava se na temelju preslikavanja definiranih u Algoritam preslikavanja BPMN-BPEL .



Slika 34: Blokovska struktura BPMN izvršavanja

4.2.2. Pseudokod algoritma preslikavanja BPEL-BPMN

Notacija korištena za opisivanje pseudokoda algoritma preslikavanja BPEL-BPMN identična je korištenoj za opisivanje pseudokoda algoritma preslikavanja BPMN-BPEL.

Funkcije algoritma preslikavanja BPEL-BPMN:

- `toBPMNprocessDraw(..)`:
    - funkcija koja popunjava BPMN element `process` na način da preslikava BPEL aktivnosti unutar glavne BPEL aktivnosti `sequence` (sadržane unutar

BPEL aktivnosti `process`) u glavni BPMN slijed i koja tijekom toga postupka poziva pripadne `toBPMNActivity(*)Draw()` funkcije sukladno pronađenoj vrsti BPEL aktivnosti,

- `toBPMNActivity(*)Draw(..)`:
    - funkcija koja preslikava BPEL aktivnost (*) u pripadno BPMN izvršavanje na temelju preslikavanja definiranih u Algoritam preslikavanja BPMN-BPEL .

Od `toBPMNActivity(*)Draw(..)` funkcija posebno je objašnjen pseudokod BPEL aktivnosti koje mogu biti ugniježđene jedna unutar druge ili koje se preslikavaju u više BPMN čvorova, a to su: BPEL `if`, `scope`, `sequence`, `while`, `pick`.

Ulazni parametri funkcija `toBPMNActivity(*)Draw`:

- A: XML element BPEL aktivnosti koja se preslikava,
- segFlow: identifikator zadnjeg kreiranog BPMN elementa `sequenceFlow` koji ulazi u BPMN inačicu aktivnosti A,
- B: XML kod BPMN elementa u koji će se smještati svi generirani BPMN elementi, a to mogu biti ili BPMN element `process` ili `subProcess`.

Tablica 4: Funkcija `toBPMNprocessDraw`

```
void toBPMNprocessDraw(A)
A – BPEL element process
B.create(„process“), C.create(„definitions“);
string idSequenceFlow=“”;
//pamćenje atributa id zadnjeg kreiranog elementa sequenceFlow

forEvery {e € A.E(„variables“) | E(e)} : BPMNvariableDraw(e, C);
//pretvaranje BPEL variable - BPMN dataObject

forEvery {e € A|E(e).type == “event/faultHandlers“} :
   forEvery {e1 € e|E(e1)} :
   BPMNevent/faultHandlerDraw(e1,B.create(E(„subProcess“)));
   //pretvaranje BPEL *handler - BPMN inlineEvent subProcess

forEvery {e € A.E(„sequence“)|E(e) in document-order} :
      idSequenceFlow=BPMNexecutionDraw(e, idSequenceFlow, B);
C.add(B);
      //pretvaranje BPEL aktivnosti - pripadni BPMN kod
```

Funkcija toBPMNprocessDraw prikazana u tablici 4 se kreće slijedno niz djecu elemente BPEL aktivnosti sequence i preslikava ih u pripadne BPEL vizualne aktivnosti pri čemu varijablu idSequenceFlow koristi za pamćenje atributa id zadnjeg kreiranog elementa sequenceFlow kako bi se napravilo potrebno povezivanje kreiranih slijednih BPEL vizualnih aktivnosti.

Osim preslikavanja sadržaja slijeda sequence, funkcija toBPMNprocessDraw vrši i preslikavanje BPEL elemenata variable u BPMN čvorove dataObject. Sve generirane BPMN elemente funkcija toBPMNprocessDraw smješta kao djecu elemente unutar BPMN elementa process ili unutar elementa definitions ako su u pitanju elementi dataObject. Na identičan način funkcionira i funkcija toBPMNscopeDraw koja na temelju BPEL aktivnosti scope popunjava BPMN element subProcess.

Tablica 5: Funkcija toBPMNscopeDraw

```
string toBPMNscopeDraw(A, segFlow, B)
A – BPEL element scope,
B - BPMN element u koji se dodaje generirani element BPMN subProcess
string idSequenceFlow="";
//pamćenje zadnjeg kreiranog elementa sequenceFlow
C.create(„subProcess");
. . .
//pretvaranje BPEL variable - BPMN dataObject
  sukladno toBPMNprocessDraw funkciji
. . .
//pretvaranje BPEL event/faultHandler - BPMN inlineEvent subProcess
  sukladno toBPMNprocessDraw funkciji
. . .
//pretvaranje BPEL compensationHandler - BPMN inlineEvent subProcess
  sukladno Slika 21: BPEL aktivnost scope s onMessage u BPMN

forEvery {e € A.E(„sequence")|E(e) in document-order} :
      idSequenceFlow=toBPMNexecutionDraw(e, idSequenceFlow, C);
D.create(„sequenceFlow").R(„id") = idSequenceFlow+1;
C.add(E(outgoing, D.R(„id")));
B.add(C,D);
return D.R(„id");
```

Funkcija `toBPMNifDraw` generira vizualnu BPMN interpretaciju BPEL aktivnosti `if` na temelju preslikavanja prikazanog na Slika 18: BPEL aktivnosti `if` u BPMN.

Tablica 6: Funkcija `toBPMNifDraw`

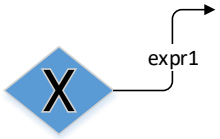
```
string toBPMNifDraw(A,segFlow,B)
C.create(„exclusiveGateway“);
C.add(E(incoming, segFlow));
D.create(„sequenceFlow“);D.Add(R(id,“..“));
D.add(E(conditionExpression,A.E(„condition“)));
C.add(E(outgoing, D.R(„id“)));
string seqFlows[n];
seqFlows[0] = toBPMNActivity(*)Draw(A.second,D.R(„id“),B);
i=1;
forEvery {a € A|E(a).type == “elseif“ OR „else“} :
      F.create(„sequenceFlow“); F.add(R(id,“..“));
      C.add(E(outgoing, F.R(„id“)));
      if (a.type==“elseif“) ->
            F.add(E(conditionExpression,a.E(„condition“)));
            seqFlows[i] = toBPMNActivity(*)Draw(a.second, F.R(„id“),B);
            //pretvaranje sadržane BPEL aktivnosti - pripadni BPMN kod
      else ->
            seqFlows[i] = BPMNActivity(*)Draw(a.first, F.R(„id“),B);
      B.add(F);
      i++;
G.create(„exclusiveGateway“);
forEvery {seq € seqFlows} : G.add(E(incoming, seg));
H.create(„sequenceFlow“); H.Add(R(id,“..“));
G.add(E(outgoing, H.R(„id“)));
B.add(C, D, G, H);
return H.R(„id“);
```

Slijedi detaljniji opis funkcioniranja funkcije `toBPMNifDraw` radi njenog lakšeg shvaćanja. Izdvajat će se dijelovi funkcije popraćeni slikama radi objašnjenja što generiraju.

Korak 1 funkcije:

```
C.create(„exclusiveGateway“);
C.add(E(incoming, segFlow));
D.create(„sequenceFlow“);D.Add(R(id,“..“));
```

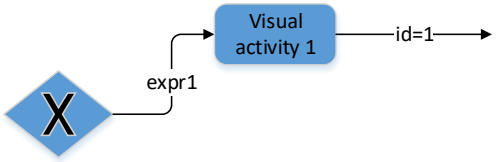
```
D.add(E(conditionExpression,A.E(„condition“)));
C.add(E(outgoing, D.R(„id“)));
```



Slika 35: Rezultat izvršavanja koraka 1

Korak 2 funkcije:

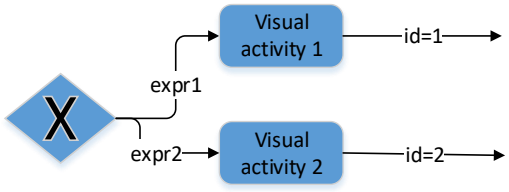
```
seqFlows[0] = BPMNActivity(*)Draw(A.second,D.R(„id“),B);
```



Slika 36: Rezultat izvršavanja koraka 2

Korak 3 funkcije:

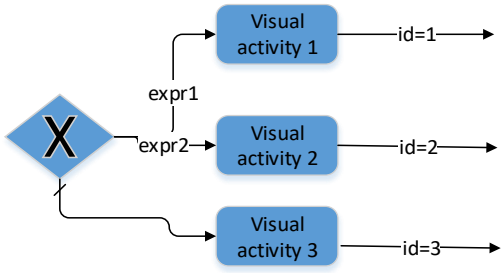
```
i=1;
forEvery {a € A|E(a).type == “elseif“ OR „else“} :
      F.create(„sequenceFlow“); F.add(R(id,“..“));
      C.add(E(outgoing, F.R(„id“)));
      if (a.type==“elseif“) ->
         F.add(E(conditionExpression,a.E(„condition“)));
        seqFlows[i] = BPMNActivity(*)Draw(a.second, F.R(„id“),B);
```



Slika 37: Rezultat izvršavanja koraka 3

Korak 4 funkcije:

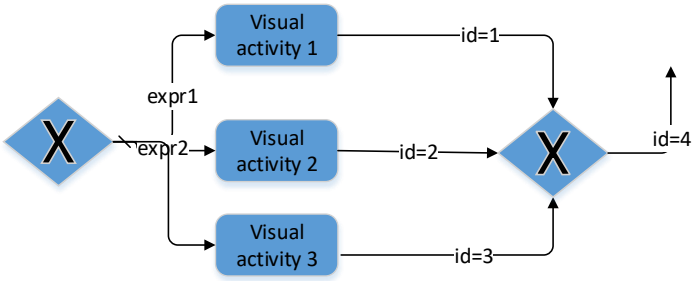
```
else ->
      seqFlows[i] = BPMNActivity(*)Draw(a.first, F.R(„id“),B);
```



Slika 38: Rezultat izvršavanja koraka 4

Korak 5 funkcije:

```
G.create(„exclusiveGateway“);
forEvery {seq € seqFlows} : G.add(E(incoming, seg));
H.create(„sequenceFlow“); H.Add(R(id,“..“));
G.add(E(outgoing, H.R(„id“)));
```



Slika 39: Rezultat izvršavanja koraka 5

Budući da se tijekom pojašnjavanja algoritma preslikavanja BPMN-BPEL spominje pojam ugniježđene BPEL aktivnosti `if`, i u ovom poglavlju je naglašen pojam ugnježđivanja BPEL aktivnosti `if` na razini BPEL XML koda. Tablica 7 prikazuje BPEL XML kod ugniježđene BPEL aktivnosti `if` i načine pozivanja funkcije `toBPMNifDraw` za generiranje BPMN vizualnih interpretacija ugniježđenih aktivnosti `if`.

Tablica 7: Ugniježđene BPEL aktivnosti `if` *(BPEL XML kod)*

```
<if> - glavna if aktivnost (*)
//toBPMNifDraw - vraća "id" izlaznog sequenceFlow elementa i
  prosljeđuje BPMN elementu process ili subProcess
  za nastavak popunjavanja slijeda
      <condition>expr</condition> ..
      <elseif><condition>expr</condition>
            <if> - prva ugniježđena aktivnost if (**)
//BPMNIfDraw – vraća "id" izlaznog sequenceFlow elementa i
  prosljeđuje roditeljskoj if (*) za nastavak generiranja iste
                <condition>expr</condition>
                <if>..</if> - druga ugniježđena aktivnost if (***)
//BPMNIfDraw – vraća "id" izlaznog sequenceFlow elementa i
  prosljeđuje roditeljskoj if (**)za nastavak generiranja iste
                <elseif>...</elseif>
           </if>
           <elseif>...</elseif>
           <else>...</else>
      </elseif>
      <else>...</else>
</if>
```

Funkcija `toBPMNsequenceDraw` generira vizualnu BPMN interpretaciju BPEL aktivnosti `sequence`. Ona slijedno pomoću elemenata `sequenceFlow` povezuje generirane BPEL vizualne aktivnosti.

Tablica 8: Funkcija `toBPMNsequenceDraw`

```
toBPMNsequenceDraw(A,segFlow,B)
string seqFlow1=segFlow;
forEvery {a € A|E(a)} : seqFlow1=BPMNActivity(*)Draw(a, seqFlow1, B);
      //pretvaranje sadržane BPEL aktivnosti - pripadni BPMN kod
return seqFlow1;
```

Funkcija `toBPMNwhileDraw` generira vizualnu BPMN interpretaciju BPEL aktivnosti `while` sukladno preslikavanjima na Slika 19: BPEL aktivnosti predstavljene s BPMN `subProcess`.

Tablica 9: Funkcija `toBPMNwhileDraw`

```
toBPMNwhileDraw(A,segFlow,B)
(analogne funkcije su za BPEL repeatUntil i BPEL forEach)
C.create(„subProcess“); //popuniti potrebnim detaljima sukladno Slika 19:
BPEL aktivnosti predstavljene s BPMN subProcess
string segFlow1=BPMNexecutionDraw(A.second, segFlow, C);
//pretvaranje sadržane BPEL aktivnosti - pripadni BPMN kod
D.create(„sequenceFlow“).R(„id“) = segFlow1+1;
C.add(E(outgoing,segFlow1+1));
B.add(C, D);
return segFlow1+1;
```

Funkcija `toBPMNpickDraw` generira vizualnu BPMN interpretaciju BPEL aktivnosti `pick` sukladno preslikavanjima na Slika 17: BPEL aktivnost `pick` u BPMN.

Tablica 10: Funkcija `toBPMNpickDraw`

```
string toBPMNpickDraw(A,segFlow,B)
C.create(„eventBasedGateway“);
C.add(E(incoming, segFlow));
B.add(C);
G.create(„exclusiveGateway“);
forEvery {a € A|E(a)} :
      D.create(„sequenceFlow“).add(R(id,“..“));
      C.add(E(outgoing, D.R(„id“)));
      if a.type==“onMessage“-> E.create(„intermediateCatchEvent“).
                               add(E(„messageEventDefinition“));
      elseif a.type==“onAlarm“ -> E.create(„intermediateCatchEvent“).
                               add(E(„timerEventDefinition“));
      B.add(E);
      F.create(„sequenceFlow“).add(R(id,“..“));
      E.add(incoming,D.R(„id“));E.add(outgoing,F.R(„id“));
      string seqFlow1 = BPMNActivity(*)Draw(a.first, F.R(„id“),B);
      //pretvaranje sadržane BPEL aktivnosti - pripadni BPMN kod
      G.add(incoming, seqFlow1);
H.create(„sequenceFlow“).add(R(id,“..“));
G.add(outgoing, H.R(„id“));
B.add(G);
return H.R(„id“);
```

Funkcija `toBPMNflowDraw` generira vizualnu BPMN interpretaciju osnovnog ponašanja BPEL aktivnosti `flow` sukladno preslikavanjima na Slika 23: Osnovno izvršavanje BPEL aktivnosti *`flow`* u BPMN u kojem su međuizvršavanja unutar aktivnosti `flow` paralelna. Što se tiče preslikavanja sinkronizacijski međuovisnih izvršavanja unutar aktivnosti `flow`, ona se u BPMN preslikavaju ručno temeljem pravila definiranih na Slika 24: Sinkronizirana aktivnost unutar aktivnosti *`flow`*

prva mogućnostdo Slika 27: Sinkronizirana aktivnost unutar aktivnosti `flow`.

Tablica 11: Funkcija toBPMNflowDraw

```
string toBPMNflowDraw(A,segFlow,B)
C.create(„parallelGateway“);
C.add(E(incoming, segFlow));
B.add(C);
string seqFlows[n];
forEvery {a € A|E(a)} :
      F.create(„sequenceFlow“); F.add(R(id,“..“));
      B.add(F.R(„id“));
      seqFlows[i] = BPMNActivity(*)Draw(a.i, F.R(„id“),B);
      i++;
G.create(„exclusiveGateway“);
forEvery {seq € seqFlows} : G.add(E(incoming, seg));
H.create(„sequenceFlow“); H.Add(R(id,“..“));
G.add(E(outgoing, H.R(id,“..“)));
B.add(G);
return H.R(„id“);
```

5. Statička provjera BPEL procesa

BPEL specifikacija u prilogu B [3] definirala je niz pravila (njih ukupno 85) koja u BPEL procesu moraju biti zadovoljena, a koja se statičkom provjerom procesa moraju provjeriti. Iako su neki od alata za modeliranje BPEL procesa moguće implementirali statičku provjeru spomenutih pravila, u doktorskom radu je definiran pseudokod algoritma na temelju kojeg se na BPEL XML razini bez uporabe bilo kakvog pomoćnog BPEL alata mogu provjeriti spomenuta pravila čime se stvara neovisnost o radnom okruženju. Zbog velikog broja pravila, pseudokodom nisu obrađena sva pravila nego samo ona pravila koja demonstriraju na koji način se pravila mogu otkrivati unutar BPEL XML koda. Ovdje slijede neka od odabranih pravila i pseudokod koji otkriva načine njihove provjere.

Notacija kojom je opisan pseudokod odabranih pravila identična je onoj korištenoj za opis pseudokoda algoritma preslikavanja BPMN-BPEL-BPMN u Preslikavanje između BPMN i BPEL. Slijedi kratki opis spomenute notacije s navedenim samo onim elementima potrebnima za opis pseudokoda u ovom poglavlju:

- `A` - XML element A BPEL izvršavanja se predstavlja kao skup,
- `A(x)` – svi elementi skupa A, a to mogu biti ili XML djeca elementi ili XML atributi,
- `A = {x | P(x)}` – elementi skupa A koji imaju svojstvo P,
- `A = {x | E(x)}` – elementi skupa A koji su XML elementi, (E - "element svojstvo"),
- `A = {x | R(x)}` – elementi skupa A koji su XML atributi (R – "atribut svojstvo"),
- `A.read(„name“)` – učitavanje skupa A imena “name”,
- `A.E/R(„name“)` - dohvaćanje vrijednosti djeteta (XML elemenata/atributa) imena “name”,
- `forEvery{x€A(x)|E(x)}:action` – izvrši akciju za svaki element x skupa A koji zadovoljava svojstvo E(x), to jest koji je po prirodi XML element.

5.1. Pravila i pseudokod statičke provjere procesa

U ovom poglavlju navedena su odabrana pravila statičke provjere s pripadajućim pseudokom koji otkriva njihovo narušavanje u BPEL procesu.

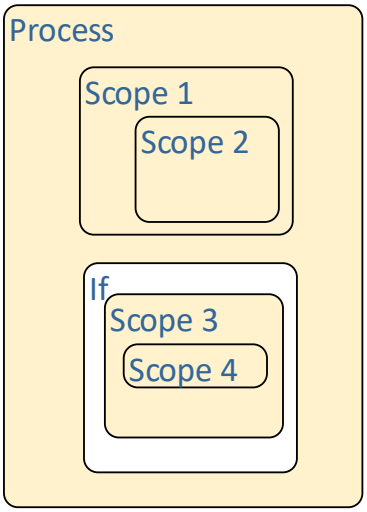
Pravilo 1: „Ukoliko je vrijednost atributa `exitOnStandardFault` aktivnosti `scope` ili `process` postavljena na "`yes`", onda se unutar njih ne smije nalaziti aktivnost `faultHandler` koja hvata neku od standardnih BPEL pogrešaka."

Pseudokod 1:

```
functionError1(A){
      forEvery {a € A.E("faulthandlers")|E(a)} :
            forEvery {b € a|E(b).type=="catch"}
                  if b.R("faultName")=="WS-BPEL standard fault" ->
                  generate (Error1, location: A.R(„Name"));}

recursionFunction1 (A){
forEvery {a € A|E(a)}
      if a.type=="scope/process" ->
            if a.R("exitOnStandardFault")=="yes" ->
                  functionError1(a);
recursionFunction1(a);}

C.read("BPEL XML dokument");
recursionFunction1 (C);
//pozivanje funkcije recursionFunction2 za korijenski element BPEL
process
```

U pseudokodu 1 prikazana je funkcija `functionError1` koja unutar proslijeđenog elementa pretražuje njegov element dijete `faulthandlers` u potrazi za aktivnošću `catch` čiji atribut `faultName="WS-BPEL standard fault"`. Ona se poziva iz funkcije `recursionFunction1` koja unutar svoga proslijeđenog elementa pretražuje svu djecu tipa `scope/process` kojima atribut `exitOnStandardFault="yes"` te u tom slučaju poziva funkciju `functionError1` za provjeru narušenosti pravila 1. Funkcija `recursionFunction1` dalje rekurzivno poziva samu sebe budući da se u BPEL kodu

elementi smještaju blokovski jedan unutar drugog. Rekurzivno pozivanje funkcije je potrebno da se pronađu sve aktivnosti `scope` koje mogu biti ugniježđene jedna unutar druge do nedefiniranog broja dubina (slika 40).




```
recursionFunction1 (B){
forEvery {b € B|E(b)}
      if b.type=="scope/process"->
            action on scope;
recursionFunction1 (b);}

_____________________________
recursionFunction1 (Process)
forEvery prolazi kroz:
1. Scope 1 (action on scope),
2. If.

_____________________________
recursionFunction1 (Scope 1):
forEvery prolazi kroz:
1. Scope 2 (action on scope).

_____________________________
recursionFunction1 (If):
forEvery prolazi kroz:
1. Scope 3 (action on scope).

_____________________________
recursionFunction1 (Scope 3):
forEvery prolazi kroz:
1. Scope 4 (action on scope).
```

Slika 40: Rekurzija funkcije `recursionFunction`

Općenito se za pretraživanje elemenata blokovski smještenih jedan unutar drugog može koristiti uopćeni oblik funkcije `recursionFunction` na sljedeći način. Funkcija `recursionFunction` se kreće kroz sve elemente svih elemenata. Za sva naredna odabrana pravila predstavljena u ovome poglavlju koristit će se analogija kao s pravilom 1.

```
recursionFunction (A){
forEvery {a € A|E(a)}
          recursionFunction(a);}
```

Pravilo 2: „Aktivnost `compensateScope` smije se koristiti samo unutar aktivnosti `faultHandler, compensationHandler` ili `terminationHandler`“.
Budući da postoji jako mnogo pravila [3] koja su oblika kao pravilo 2: „aktivnost *abc* smije se koristiti samo unutar aktivnosti *def*“, ona su logički identična pravilu 2 i pseudokod 2 se može koristiti i za otkrivanje narušenosti tih pravila.

Pseudokod 2:

```
functionError2(A){
     forEvery {a € A|E(a)} :
          if a.type==“compensateScope“ ->
               generate (Error2, location: a.R(„Name“));}

recursionFunction2(A){
forEvery {a € A|E(a)}
          if a.type!="fault/compensation/terminationHandler" ->
               functionError2(a);
recursionFunction2(a);}

B.read("BPEL XML dokument");
recursionFunction2(B);
//pozivanje funkcije recursionFunction2 za korijenski element BPEL process
```

Pravilo 3: „BPEL proces mora sadržavati minimalno jednu aktivnost `receive` ili `pick` s atributom `createInstance="yes"` kojim se kreira nova instanca poslovnog procesa“.

Pseudokod 3:

```
recursionFunction3(A, bool flag){
forEvery {a € A|E(a)}
          if a.type=="receive" or a.type=="pick" ->
               if a.R(„createInstance“)==“yes“ ->
                    flag=true; exit;
recursionFunction3(a,false);
return flag}
```

```
bool startActivityExists;
B.read("BPEL XML dokument");
startActivityExists=recursionFunction3(B, false);
if startActivityExists==false -> generate (Error3);
//pozivanje funkcije recursionFunction3 za korijenski element BPEL process
```

U pseudokodu 3 generirana je rekurzivna funkcija `recursionFunction3` koja se kreće kroz sve elemente unutar proslijeđenog elementa i pretražuje da li postoji element oblika `receive` ili `pick` s atributom `createInstance="yes"`. Ukoliko postoji barem jedan takav element, postavlja se značka `flag` u vrijednost `true` i izlazi se iz funkcije jer je za pravilo dovoljno postojanje samo jednog takvog elementa.

Pravilo 4: „`partnerLink` mora sadržavati `myRole` ili `partnerRole` ili oba i atribut `initializePartnerRole` se ne smije koristiti na onom `partnerLink` koji nema `partnerRole`“.

Pseudokod 4:

```
recursionFunction4(A, bool flag){
     forEvery {a € A|E(a)} :
          if a.type=="partnerLink" ->
              if !a.R(„myRole").exists
          and !a.R(„partnerRole").exists ->
                    flag=false;
                    generate(Error4, location: a.R(„Name");
recursionFunction4(a,false);}

B.read("BPEL XML dokument");
recursionFunction4(B,false);
//pozivanje funkcije recursionFunction4 za korijenski element BPEL process
```

U ovom poglavlju demonstriran je pseudokod nad elementima BPEL procesa za nekoliko odabranih pravila BPEL specifikacije koja se statičkom provjerom moraju provjeriti nakon izgradnje BPEL procesa. Pravila za statičku provjeru definiranih u BPEL specifikaciji ima 95 stoga pa su odabrana neka od navedenih pravila radi demonstracije koncepta statičke provjere koja automatski vrše provjeru nad XML kodom BPEL procesa i daju informaciju o lokaciji pogreške ukoliko je pravilo prekršeno. Na ovaj način ubrzava se razvoj BPEL procesa jer se ne mora vršiti ručna provjera zadovoljenosti svih pravila što bi bio krajnje iscrpljujući proces zbog XML prirode BPEL standarda.

6. Formalna provjera BPEL procesa

Razvoj sustava s kompleksnim izvršavanjem može biti iznimno teška zadaća ukoliko se autor želi osigurati da će sustav na svim mogućim putanjama izvršavanja uvijek raditi onako kako je zamišljeno. Sustav se katkada dulje vrijeme može ponašati ispravno, no u jednom trenutku on se može naći u stanju u kojem željeno obilježje nije zadovoljeno ili se može pojaviti do tada neizvođena putanja koja nije u skladu sa željenim ponašanjem svih mogućih putanja sustava. Sustav koji se ne ponaša uvijek onako kako je autor zamislio ne može se smatrati ispravnim. Dapače, dovoljno je jedno neželjeno stanje ili pak jedna neželjena putanja izvršavanja da se sustav kategorizira kao neispravan.

Skup svih mogućih stanja sustava i prijelaza između stanja, temeljem kojih se generiraju putanje izvođenja sustava, zove se *prostor stanja* sustava. Prostor stanja sustava može biti iznimno velik. Posebno kompleksni jesu sustavi koji u sebi posjeduju paralelna izvršavanja (u daljem tekstu *konkurentni sustavi*) zbog velikog broja mogućih kombinacija brzina izvršavanja paralelnih komponenti (engl. *race condition*) zbog čega se prostor stanja može znatno povećati. Kod takvih sustava jako je teško proći kroz cijeli prostor stanja i najčešće se neželjeno ponašanje otkrije tijekom slučajnog izvršavanja sustava. Kod trivijalnih sustava, čiji je prostor stanja relativno malen, provjera zadovoljenosti željenih obilježja može se ručno provesti, no kod sustava sa većim prostorom stanja to je teže bez automatiziranih metodologija provjere prostora stanja (u daljnjem tekstu *formalna provjera*).

6.1. Mehanizam formalne provjere

Jedna od najpoznatijih metoda formalne provjere je provjera modela (engl. *model checking*) [38][39][40]. Provjera modela počinje generiranjem prostora stanja sustava. S obzirom da prethodno spomenuti pojmovi *'stanje sustava'* i *'putanja izvršavanja sustava'* čine prostor stanja sustava, važno ih je egzaktno definirati. Stanje sustava predstavlja informaciju o trenutnom koraku izvođenja sustava (programskog koda) i o trenutnoj vrijednosti svih varijabli sustava u tom koraku. Samo jedno stanje je obilježeno kao početno stanje, a to je ono stanje u kojem se još niti jedan korak izvođenja sustava nije desio i sve varijable imaju početne vrijednosti. Sustav se u svakom trenutku može nalaziti u samo jednom od svojih mogućih stanja koje se naziva trenutnim stanjem. Iz trenutnog stanja on može preći u neko od mogućih narednih stanja. Putanja izvođenja sustava predstavlja niz mogućih prijelaza sustava iz stanja u stanje. Ona

uvijek počinje iz početnog stanja i završava u nekom od konačnih stanja ukoliko takva stanja postoje. Neki sustavi imaju konačna stanja jer posjeduju samo konačna izvršavanja (engl. *finite behaviour system*) [41][42], dok neki sustavi nemaju konačna stanja jer varijable posjeduju beskonačna izvršavanja (engl. *infinite behaviour system*) [41][42].

Formalnim modelom moguće je egzaktno prikazati cijeli prostor stanja sustava, dakle moguće je prikazati sva moguća stanja sustava i sve moguće putanje njegovog izvršavanja pri čemu se podrazumijeva da je broj stanja sustava, ma koliko velik bio, ipak konačan (engl. *finite state system*). Može se reći da je formalni model slika prostora stanja sustava. Formalni model se simbolički najlakše opisuje konačnim automatima o kojima će biti više u narednom poglavlju, dok ga je vizualno najlakše predstaviti usmjerenim grafom u kojem čvorovi predstavljaju stanja sustava. Čvorovi su susjedni ukoliko sustav iz jednog stanja prelazi u drugo pri čemu usmjereni luk vodi od izvorišnog prema odredišnom stanju. Jedno stanje u grafu je označeno kao početno stanje. Grafovi sustava sa konačnim izvršavanjima ne sadržavaju cikluse (aciklički grafovi) dok grafovi sustava sa beskonačnim izvršavanjima sadržavaju cikluse (ciklični grafovi).

Svrha formalnog modela je omogućiti automatiziranu provjeru nad prostorom stanja sustava (*u daljnjem tekstu formalna provjera*) pretragom nad stanjima ili pretragom nad putanjama s ciljem utvrđivanja da li je željeno obilježje sustava zadovoljeno u svim njegovim stanjima, odnosno na svim njegovim putanjama izvršavanja.

6.1.1. Izgradnja formalnog modela konačnim automatima

Ovim potpoglavljem opisani su koraci izgradnje formalnog modela sustava konačnim automatima (engl. *finite state automata*) [12] (u daljnjem tekstu automati) najčešće korištenima za prikaz formalnog modela sustava. Kako je već u definiciji formalnog modela spomenuto, formalni model predstavlja sliku prostora stanja sustava. Budući da stanje sustava sadržava informaciju o trenutnom koraku izvođenja sustava (programskog koda) i o trenutnoj vrijednosti svih varijabli u tom koraku, teoretski je za svaki korak programskog koda sustava moguće generirati onoliko stanja sustava koliko ima mogućih kombinacija vrijednosti svih varijabli. Neka od tako generiranih stanja naravno neće biti moguća jer varijable sustava u tom koraku izvođenja neće nikada imati pridruženu kombinaciju vrijednosti. No može se zaključiti da, što je veći broj varijabli u sustavu, što je veći broj mogućih vrijednosti za svaku varijablu i što je veći broj koraka u programskom kodu sustava, time se povećava broj mogućih stanja sustava čime se usložnjava pretraga nad stanjima stoga se posebna pozornost treba obratiti na mogućnosti smanjenja prostora stanja sustava.

Pojednostavljeno ponašanje sustava iz kojega su izbačeni elementi nerelevantni za formalnu provjeru i koji sadržava samo one elemente, za koje je prethodno utvrđeno da su relevantni za formalnu provjeru, zvat će se *apstraktni/simbolički model sustava*. On ima znatno manji prostor stanja što je i svrha njegovog postojanja. Izbačeni elementi mogu biti koraci programskog koda, varijable, ili moguće vrijednosti uključenih varijabli. Proces formiranja apstraktnog modela sustava, odnosno proces izbora elemenata ponašanja sustava koji će se uključiti u apstraktni model, odnosno onih koji neće, iznimno je osjetljiv proces kojim je potrebno smanjiti prostor stanja, a pri tom zadržati nepromijenjeno ponašanje sustava koje se želi provjeriti. Ukoliko su u apstraktnom modelu izostavljeni ili izmijenjeni elementi sustava relevantni za formalnu provjeru, kasnija formalna provjera nad formalnim modelom apstraktnog modela neće dati točne rezultate. Dobro formiran apstraktni model sustava predstavlja preduvjet za uspješnu formalnu provjeru.

Apstraktni model sustava simbolično se može predstaviti proširenim automatima (engl. *extended finite state automata*) [12] koji se procesom ekspanzije/izvršavanja (engl. *automata expansion*) prevode u regularne automate (engl. *pure finite state automata*) [12], a koji predstavljaju formalni model apstraktnog modela sustava (u daljnjem tekstu samo formalni model sustava jer će se za formalnu provjeru apstraktni model sustava, to jest pojednostavljeno ponašanje sustava, smatrati referentnim sustavom). Slijedi definicija automata.

Prošireni automati (kojima je opisan apstraktni model sustava) definirani su kao šestorka (S, s_0, D, L, T, F), gdje je [12]:

- S - konačan, neprazan skup stanja sustava,
- s_0 - $s_0 \in S$, početno stanje sustava,
- D – konačni skup imenovanih varijabli (engl. *named data objects*),
- L – konačni skup imenovanih akcija nad varijablama skupa D,
- $T \subseteq S \times L \times S$ funkcija prijelaza između stanja,
- F – skup prihvatljivih stanja; $F \subseteq S$.

Stanje kod proširenih automata nosi samo informaciju o koraku izvođenja procesa. Funkcijom prijelaza iz trenutnog stanja definirano je koji se prijelazi mogu desiti u trenutnom stanju, a čijim izvršavanjem se prelazi u sljedeća stanja. Funkcija prijelaza predstavlja akcije nad varijablama koje se izvršavaju ukoliko je uvjet ispunjavanja istinit. Uvjet ispunjavanja akcije je funkcija vrijednosti varijabli u izvorišnom stanju iz kojeg prijelaz vodi. Izvršenjem akcije

prelazi se u sljedeće stanje čija vrijednost varijabli je rezultat izvršenja akcije. Prošireni automati imaju dakle znatno manje stanja jer njihova stanja nose samo informaciju o koraku izvođenja procesa dok su informacije o vrijednostima varijabli smještene u skup D.

Prošireni automati se vizualno mogu prikazati UML dijagramima stanja (engl. *UML statechart*), gdje stanja dijagrama predstavljaju stanja proširenog automata. Početno stanje je posebno obilježeno, kao i prihvatljiva stanja. Prijelazi između stanja dijagrama, označeni akcijom koja se njima izvršava, predstavljaju prijelaze između stanja proširenog automata.

Regularni automati (kojima je opisan formalni model sustava) definirani su kao petorka ($\sum$, S, s_0, δ, F), gdje je [12]:

- $\sum$ - ulazni alfabet (konačan, neprazan skup simbola),
- S - konačan, neprazan skup stanja sustava,
- s_0 - $s_0 \in S$, početno stanje sustava,
- δ – funkcija prijelaza između stanja; δ: S x $\sum \rightarrow$ S (deterministički automat); δ: S x $\sum \rightarrow$ P(S) (nedeterministički automat),
- F – skup prihvatljivih stanja; $F \subseteq S$.

Stanje kod regularnih automata nosi sve informacije o sustavu: od koraka izvođenja sustava do trenutnog sadržaja svih varijabli u tom koraku, a to su sve one informacije koje su obuhvaćene u definiciji stanja formalnog modela. Prošireni automati prevode se u regularne automate premještanjem informacija iz skupa varijabli D u stanja. To je spomenuti proces ekspanzije proširenih automata u regularne, to jest prevođenja apstraktnog modela u formalni model. Proces ekspanzije počinje tako da se za svako stanje proširenih automata (*predstavlja korak u izvođenju sustava*), osim za početno, napravi onoliko kopija toga stanja koliko ima mogućih kombinacija vrijednosti svih varijabli. Početno stanje je izuzeto, ono ostaje jedinstveno i u njega se za vrijednost varijabli postave njihove početne vrijednosti. Svaka kopija stanja treba imati sve odlazne i dolazne prijelaze kao i originalno stanje u proširenom automatu, nakon čega se ti prijelazi analiziraju te se brišu oni koji nisu izvedivi, to jest u kojima akcija L nije izvediva.

Kao što je već naglašavano, ponašanje konkurentnih sustava zbog većeg prostora stanja teže se može provjeriti. Potreba za formalnom provjerom postoji i kod nekonkurentnih sustava ukoliko je njihovo ponašanje dovoljno kompleksno, no kod konkurentnih sustava gotovo je neizbježna i zbog toga se formalna provjera često i povezuje isključivo s konkurentnim sustavima.

Formalni model konkurentnih sustava u obzir mora uzeti sve moguće kombinacije brzina izvršavanja paralelnih komponenti, a to se pravi na način da se za svaku paralelnu komponentu sustava kreira zaseban apstraktni model koji se predstavi vlastitim proširenim automatom. Paralelno asinkrono izvršavanje paralelnih komponenti simulira se asinkronim proizvodom (engl. *asynchronous product of automata*) proširenih automata komponenti kojim se dobiva prošireni automat konkurentnog sustava u cjelini i on se u formalni model (izražen regularnim automatom) prevodi već spomenutim procesom ekspanzije. Konstrukcija proširenog automata za svaku paralelnu komponentu i računanje asinkronog proizvoda ne provodi se kod nekonkurentnih sustava jer nema potrebe za simulacijom paralelnog izvršavanja. Slijedi opis asinkronog proizvoda proširenih automata.

Asinkroni proizvod dvaju proširenih automata A = {S, s_0, D, L, T, F} i B = {S', s_0', D, L', T', F'} jeste novi prošireni automat {S'', s'', D'', L'', T'', F''} takav da je [12]:

- S'' = S x S',
- S_0'' = (s_0, s_0'),
- D'' = D ∪ D',
- L'' = L ∪ L',
- T'' ⊆ S'' x L'' x S'', gdje ∀ ((n, n'), l, (m,m')) ∈ T' :
  (l∈L ∧ (n,l,m)) ∨ (l∈L' ∧ (n',l,m')),
- F'' = F x F',

gdje je:

- S'' = S x S' – stanja novog automata: trenutna stanja ulaznih automata A i B,
- S_0'' = (s_0, s_0') - početna stanja novog automata: oba ulazna automata A i B su u početnom stanju,
- D'' = D ∪ D' – skup varijabli novog automata: unija varijabli ulaznih automata A i B,
- L'' = L ∪ L' - skup akcija nad varijablama novog automata: unija akcija nad varijablama ulaznih automata A i B,
- T'' ⊆ S'' x L'' x S'', gdje ∀ ((n, n'), l, (m,m')) ∈ T' : (l∈L ∧ (n,l,m)) ∨ (l∈L' ∧ (n',l,m')), - funkcija prijelaza između stanja novog automata podrazumijeva da se korak prijelaza izvršio u oba ulazna automata A i B,

- F'' = F x F' - skup prihvatljivih stanja novog automata: oba ulazna automata A i B su u prihvatljivom stanju.

6.1.2. Provjera formalnog modela

Svrha kreiranja formalnog modela sustava jeste provjera da li je željeno obilježje sustava uvijek zadovoljeno ili ne. Ponašanje koje uvijek zadovoljava željeno obilježje, naziva se željeno ponašanje, dok se ono ponašanje koje barem jednom ne zadovoljava željeno obilježje, naziva neželjeno ponašanje. Proces provjere ponašanja sustava, bilo da je riječ o provjeri zadovoljenosti obilježja u svakom stanju ili na svakoj putanji, započinje generiranjem neželjenog ponašanja mehanizmom negacije željenog ponašanja. Neželjeno ponašanje se prikazuje proširenim automatom koji sadržava univerzalan prostor stanja u kojem je naglašeno neželjeno ponašanje na način da su kao prihvatljiva stanja označena ona stanja u kojima željeno obilježje nije zadovoljeno. Provjera prisutnosti neželjenog ponašanja u ponašanju sustava postiže se pomoću sinkronog proizvoda proširenog automata sustava i proširenog automata njegovog neželjenog ponašanja. Sinkronim proizvodom spomenutih automata dobiva se presjek njihovog ponašanja koji sadržava cijeli prostor stanja sustava u kojem je neželjeno ponašanje (ukoliko je prisutno) naglašeno tako da se može otkriti. Slijedi opis sinkronog proizvoda proširenih automata.

Sinkroni proizvod dvaju proširenih automata A = {S, s_0, D, L, T, F} i B = {S', s_0', D, L', T', F'} jeste novi prošireni automat {S'', s'', D'', L'', T'', F''} takav da [12]:

- S'' = S x S',
- s_0'' = (s_0, s_0'),
- D'' = D ∪ D',
- L'' = L x L',
- T'' ⊆ S'' x L'' x S'', gdje ∀ ((n, n'), (l, l'), (m,m')) ∈ T' :
  (l∈L ∧ (n,l,m)) ∧ (l∈L' ∧ (n',l,m')),
- F'' = F x F',

gdje:

- S'' = S x S' – stanja novog automata: trenutna stanja ulaznih automata A i B,
- S_0'' = (s_0, s_0') - početna stanja novog automata: oba ulazna automata A i B su u početnom stanju,

- D'' = D ∪ D' – skup varijabli novog automata: unija varijabli ulaznih automata A i B,
- L'' = L ∪ L' - skup akcija nad varijablama novog automata: unija akcija nad varijablama ulaznih automata A i B,
- T'' ⊆ S'' x L'' x S'', gdje ∀ ((n, n'), l, (m,m')) ∈ T' : (l∈L ∧ (n,l,m)) ∨ (l∈L' ∧ (n',l,m')), - funkcija prijelaza između stanja novog automata podrazumijeva da se korak prijelaza izvršio ili u ulaznom automatu A ili u ulaznom automatu B,
- F'' = F x F' - skup prihvatljivih stanja novog automata: oba ulazna automata A i B su u prihvatljivom stanju.

Sinkronim proizvodom proširenih automata dobiva se novi prošireni automat koji sadržava samo one prijelaze koji su zajednički i jednom i drugom ulaznom automatu, to jest sinkronim proizvodom dobiva se prošireni automat koji predstavlja sinkronizirano ponašanje ulaznih automata što se slikovito može predstaviti na način da ukoliko su oba automata u nekom stanju, prijelaz u sljedeće stanje je moguć jedino ukoliko je dozvoljen u oba automata. Prihvatljiva stanja proširenog automata sinkronog proizvoda su samo ona stanja u kojima su oba ulazna automata u prihvatljivom stanju. Prošireni automat sinkronog proizvoda se u regularni automat prevodi već spomenutim procesom ekspanzije.

U prethodnim izlaganjima ovog poglavlja na nekoliko mjesta su spominjana obilježja sustava koja moraju biti zadovoljena u svakom stanju i ona koja moraju biti zadovoljena na svakoj putanji. Te dvije kategorije su definirane zbog toga što se različiti algoritmi koriste za otkrivanje neželjenog ponašanja u kojem obilježje sustava nije zadovoljeno u svakom stanju i onoga u kojem obilježje nije zadovoljeno na svakoj putanji.

Ukoliko se nad sustavom postavlja uvjet da željeno obilježje mora vrijediti u svakom njegovom stanju, onda se takva obilježja sustava nazivaju obilježjima sigurnosti (engl. *safety property*) [12]. Željeno ponašanje ovakvih sustava glasilo bi: „*u svakom stanju željeno obilježje mora biti zadovoljeno*“, a suprotno neželjeno ponašanje glasilo bi: „*postoji barem jedno stanje u kojem željeno obilježje nije zadovoljeno*“. Prošireni automat ovakvog neželjenog ponašanja sadržavao bi univerzalni prostor stanja s tim što bi se prihvatljivima označila samo ona stanja u kojima željeno obilježje sustava nije zadovoljeno čime bi se istaklo neželjeno ponašanje. Ukoliko se sva stanja proširenog automata sustava označe kao prihvatljiva, sinkronim proizvodom kao presjekom ponašanja automata sustava i automata gornjeg neželjenog ponašanja dobiva se automat koji sadržava cijeli prostor stanja sustava s naglašenim neželjenim ponašanjem u vidu

postojanja prihvatljivih stanja. Prihvatljiva stanja u sinkronom proizvodu čine presjek prihvatljivih stanja u oba automata. U automatu sustava prihvatljivim su se označila sva stanja, dok su se u automatu neželjenog ponašanja prihvatljivim označila ona stanja u kojima željeno obilježje nije zadovoljeno. Presjek tih stanja jesu stanja u kojima željeno obilježje nije zadovoljeno. Postojanje prihvatljivih stanja u sinkronom proizvodu dokazuje dakle postojanje stanja u kojem željeno obilježje nije zadovoljeno. Algoritam koji se koristi za pronalazak prihvatljivih stanja dosežljivih iz početnog stanja jeste algoritam pretraživanja prostora stanja u dubinu (engl. *depth first search*) [43]. Algoritam pretraživanja u dubinu jeste jedan od algoritama sustavnog obilaska čvorova usmjerenih grafova jer regularni automat nije ništa drugo do usmjereni graf.

Ukoliko se nad sustavom postavlja uvjet da željeno obilježje ne mora vrijediti u svakom stanju, nego da eventualno mora vrijediti na svakoj putanji, takva obilježja sustava nazivaju se obilježjima životnosti (engl. *liveness property*) [12]. Željeno ponašanje ovakvih sustava glasilo bi: „*na svakoj putanji izvođenja željeno obilježje eventualno će biti zadovoljeno*“, a suprotno neželjeno ponašanje glasilo bi: „*postoji putanja izvođenja na kojoj željeno obilježje nije nikada zadovoljeno*“. Prošireni automat ovoga neželjenog ponašanja sadržavao bi univerzalni prostor stanja s istaknutim negativnim ponašanjem prikazanim putanjom na kojoj željeno obilježje nije nikada zadovoljeno. Simulacija takve putanje postiže se na način da se prihvatljivim označi samo ono stanje koja ne zadovoljava željeno obilježje te da se napravi prijelaz iz tog stanja prema njemu samom, a koji se izvršava uz uvjet da željeno obilježje nije zadovoljeno. Ukoliko se sva stanja proširenog automata sustava označe kao prihvatljiva, sinkronim proizvodom automata sustava i automata opisanog neželjenog ponašanja dobiva se automat koji sadržava cijeli prostor stanja sustava s naglašenim neželjenim ponašanjem u vidu postojanja prihvatljivih stanja dosežljivih samih iz sebe. Postojanje takvih stanja u sinkronom proizvodu dokazuje postojanje ograničene ili neograničene putanje u izvođenju sustava na kojoj željeno obilježje nije nikada zadovoljeno. Algoritam koji se koristi za pronalazak prihvatljivih stanja dosežljivih iz početnog stanja i istovremeno dosežljivih samih iz sebe jeste algoritam ugniježđenog pretraživanja prostora stanja u dubinu (engl. *nested depth first search*) [43]. Dokazivanje obilježja životnosti obično se provodi kod sustava s beskonačnim izvršavanjima čiji grafovi automata sadržavaju cikluse. Da bi se ovaj algoritam mogao primijeniti i kod sustava sa ograničenim izvršavanjima čiji grafovi automata ne sadržavaju cikluse, nad konačnim putanjama izvršavanja u automatu sustava primjenjuje se *'stutter ekstenzija*' (engl. *stutter*

extension) kojom se zadnjem stanju u konačnoj putanji izvršavanja dodaje ništavni prijelaz (engl. *nil action*) koji vodi ka njemu opet. Ništavni prijelaz nema nikakav uvjet, niti efekt i on se primjenjuje kako bi se konačna putanja izvršavanja (*beskonačnim izvršavanjem toga prijelaza iz zadnjeg stanja u samog sebe*) pretvorila u beskonačnu.

Teorijska osnova na kojoj su bazirani gornji algoritmi pretrage nad regularnim automatom sinkronog proizvoda je sljedeća [12]. Neka je s $\sum$ označen konačan skup simbola kojima se označavaju prijelazi iz jednog stanja u drugo u regularnom automatu sinkronog proizvoda. Na ovaj način simbolički se mogu prikazati putanje izvršavanja. Skup svih konačnih uređenih nizova sastavljenih od simbola iz skupa $\sum$ označava se s $\sum^*$, dok se skup svih beskonačnih uređenih nizova sastavljenih od simbola iz skupa $\sum$ označava s $\sum^\omega$. Bilo koji konačni ili beskonačni uređeni niz sastavljen od simbola iz skupa $\sum$ (*u daljem tekstu riječ*) može dakle predstavljati jednu putanju izvršavanja automata (engl. *automation run*) gdje izvršavanje počinje iz početnog stanja izvršavajući one prijelaze iz stanja u stanje sukladno nadolazećem simbolu u nizu, naravno uz pretpostavku da pripadajući prijelaz iz trenutnog stanja postoji. Riječ iz skupa $\sum^*$ ili $\sum^\omega$ može biti ili prihvaćena ili ne od strane automata i na toj definiciji se temelje gornji algoritmi.

Konačna riječ iz skupa $\sum^*$ smatra se prihvaćenom od strane automata ukoliko ona predstavlja izvršavanje koje završava u nekom od prihvatljivih stanja F. Konačne riječi mogu biti prihvaćene od strane samo onih automata čiji usmjereni grafovi ne sadržavaju cikluse, a to su dakle automati sustava sa konačnim izvršavanjima. U slučaju automata koji prihvaćaju samo konačne riječi, jezik konačnog automata L ($L \in \sum^*$) jeste skup svih konačnih riječi iz $\sum^*$ prihvaćenih od strane konačnog automata. Ukoliko algoritam pretraživanja u dubinu, koji je koristi za provjeru obilježja sigurnosti nad sinkronim proizvodom, pronađe prihvatljivo stanje koje je dosežljivo iz početnog stanja, algoritam će kao rezultat vratiti konačnu riječ koja završava u tom prihvatljivom stanju, to jest algoritam vraća riječ koja je prihvaćena od strane automata sinkronog proizvoda. Ukoliko se pak ne pronađe niti jedno prihvatljivo stanje dosežljivo iz početnog stanja, to podrazumijeva da ne postoji niti jedna konačna riječ koja može biti prihvaćena od strane automata sinkronog proizvoda, što podrazumijeva da je jezik automata sinkronog proizvoda prazan.

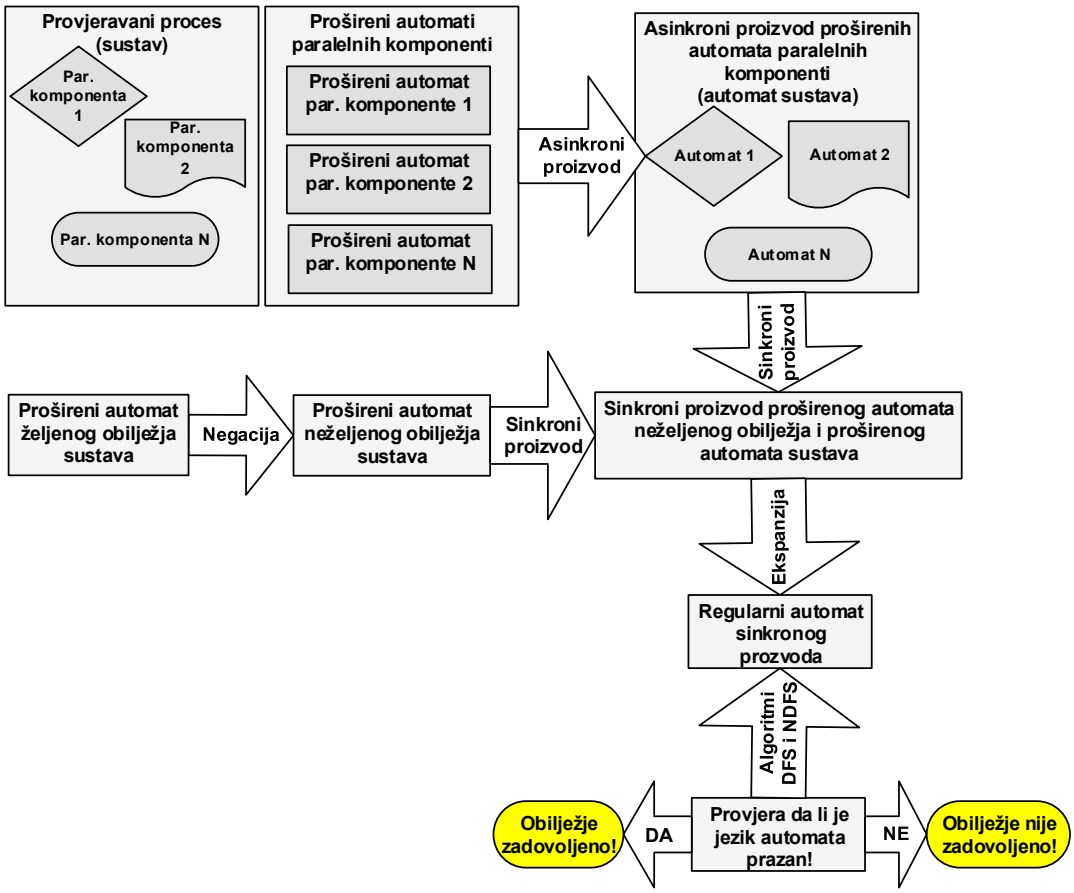
Beskonačna riječ iz skupa $\sum^\omega$ smatra se prihvaćenom od strane automata ukoliko ona predstavlja izvršavanje koje prolazi kroz prihvatljivo stanje koje je dosežljivo samo iz sebe. Beskonačne riječi mogu biti prihvaćene od strane samo onih automata čiji usmjereni grafovi

sadržavaju cikluse, a to su dakle automati sustava sa beskonačnima izvršavanjima. Automati koji simuliraju beskonačna izvršavanja nazivaju se *Büchi automatima*. U slučaju automata koji prihvaćaju beskonačne riječi, jezik automata L ($L \subset \sum^{\omega}$) jeste skup svih beskonačnih riječi iz $\sum^{\omega}$ prihvaćenih od strane *Büchi* automata. Ukoliko algoritam ugniježđenog pretraživanja u dubinu, koji se koristi za provjeru obilježja životnosti nad sinkronim proizvodom, pronađe prihvatljivo stanje koje je dosežljivo iz početnog stanja i iz samoga sebe, algoritam će kao rezultat vratiti beskonačnu riječ koja prolazi kroz to prihvatljivo stanje, to jest algoritam vraća riječ koja je prihvaćena od strane automata sinkronog proizvoda. Ukoliko se pak ne pronađe niti jedno prihvatljivo stanje dosežljivo iz početnog stanja i iz samoga sebe, to podrazumijeva da ne postoji niti jedna beskonačna riječ koja može biti prihvaćena od strane automata sinkronog proizvoda što podrazumijeva da je jezik automata sinkronog proizvoda prazan.

Jezik sinkronog proizvoda dvaju automata jeste presjek jezika ulaznih automata. Budući da u sinkronom proizvodu automata učestvuju dva automata: automat sustava čije se ponašanje provjerava i automat neželjenog ponašanja, slučaj kada je jezik sinkronog proizvoda prazan, podrazumijeva da ne postoje riječi koje su prihvaćene od strane oba automata. To podrazumijeva da se sustav nikada ne posjeduje neželjeno ponašanje. Analogno tomu, slučaj kada jezik sinkronog proizvoda nije prazan, podrazumijeva da postoje riječi koje su prihvaćene od strane oba automata. To podrazumijeva da sustav posjeduje neželjeno ponašanje. Slika 41 prikazuje korake procesa provjere modela.

Neželjeno svojstvo sustava, čije nepostojanje se želi verificirati, može se iskazati *promela never claim* sintaksom [44] ili LTL logikom (engl. *linear temporal logic*) [45]. LTL logika nudi sljedeće konstrukcije za predstavljanje neželjenih/željenih ponašanja:

[] (vremenski operator „*uvijek*“), <> (vremenski operator „*eventualno*“), ! (*bool* operator negacije), U/W (vremenski operator jako/slabo „dok“), V („ili“), && („i“), -> („logička implikacija“), <-> („logička ekvivalencija“).



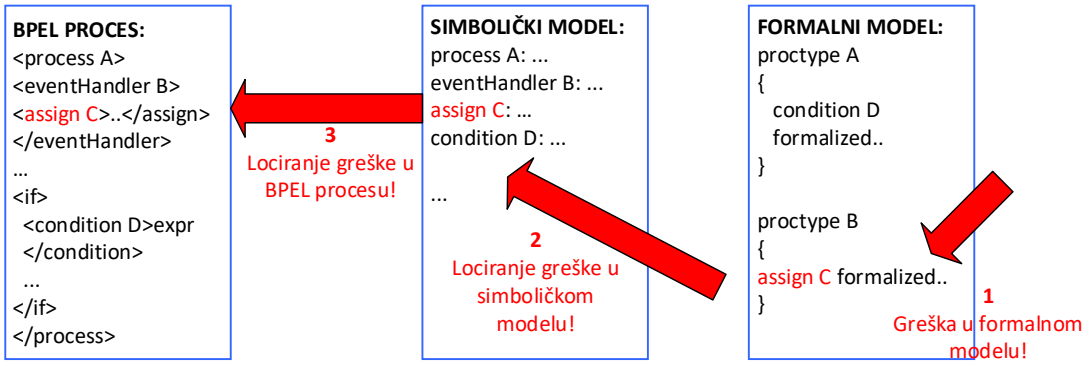
Slika 41: Koraci procesa provjere modela

6.2. Formalna provjera BPEL procesa

Na početku ovog poglavlja definiran je pojam formalnog modela kao slike prostora stanja sustava. Isto tako je objašnjeno da je za provjeru željenih obilježja sustava prolazak kroz svako moguće stanje nerijetko nepotreban i predstavlja samo nepotrebnu potrošnju vremenskih i memorijskih resursa te da je temeljem toga ponašanje sustava poželjno pojednostaviti izbacivanjem onih elemenata kojima se prostor stanja smanjuje, a zadržavanjem onih elemenata ponašanja bitnih za provjeru željenog obilježja, to jest potrebno je definirati apstraktni model sustava koji se izvršavanjem pretvara u njegov formalni model. Osim zbog cilja smanjenja prostora stanja, apstraktni model je potreban i da bi se mogla verificirati željena svojstva, to jest potrebno je na pravilan način simulirati izvršavanje realnog sustava kako bi se u tom apstraktnom izvršavanju kreirali koraci kojima se željena svojstva uistinu mogu provjeriti. U ovom poglavlju opisan je proces izgradnje apstraktnog modela BPEL procesa koji će se u daljnjem tekstu radi jednostavnosti zvati formalni model procesa iako na temelju definicije iz

poglavlja 6.1.1 formalni model tek nastaje procesom ekspanzije/izvršavanja apstraktnog modela.

Proces izgradnje formalnog modela procesa počinje predstavljanjem simboličkog opisa BPEL standarda kojim su simbolički i pojednostavljeno opisani svi elementi procesa koji će se uvrstiti u formalni model. Simbolički opis BPEL standarda koristit će se kroz kasnija objašnjena vezana za formalni model procesa, a on ima i ulogu da se nakon otkrivanja pogreške u formalnom modelu, lokacija pogreške može lako utvrditi na temelju notacije kojom su opisani relevantni elementi BPEL procesa koji su bili uvršteni u formalni model. Nakon predstavljanja simboličkog modela procesa u Simbolički prikaz BPEL standarda, u Obilježja sustava za provjeru će se definirati obilježja sustava koja se nad formalnim modelom procesa mogu provjeravati. Potom je definirano preslikavanje između simboličkog opisa procesa i njegovog formalnog modela izraženog *Promela Meta Level* (*Promela*) sintaksom [46], kao jednom od sintaksi kojima se mogu opisivati prošireni automati. Slika 42 prikazuje vezu između BPEL procesa, njegovog simboličkog i formalnog modela.



Slika 42: Veza između modela formalne provjere

6.2.1. Simbolički prikaz BPEL standarda

Kao što je već rečeno, simbolički model procesa uključit će sve one dijelove procesa koji su relevantni za svojstva koja će se provjeravati formalnom provjerom. Budući da će svojstva formalne provjere biti fokusirana na upravljanje varijablama, u simbolički model procesa uključit će se svi oni dijelovi procesa koji na neki način dotiču varijable, bilo da je riječ o čitanju ili ažuriranju varijabli, a to su sljedeći dijelovi BPEL procesa:

- **scope**[4]: scopeName, variables(variable$_1$,variable$_2$,..,variable$_n$), eventHandlers(eventHandler$_1$,eventHandler$_2$..,eventHandler$_n$[5]), activity,
- **variable**: variableName, type(XSD type, element ili WSDL messageType),
- **WSDL operation**: operationName, messageType(inputMessage), messageType(outputMessage),
- aktivnosti za komunikaciju s partnerima:
  - **invoke**/**reply**/**receive**/**onMessage**: invokeName, operationName, inputVariableName, outputVariableName,
  - **pickMessageEvent**: pickMessageEventName, operationName, variableName, activity,
- aktivnost za ažuriranje varijabli:
  - **assign**: assignName, fromVariableNames(variable$_1$, variable$_2$,..,variable$_n$), toVariableName,
- aktivnosti koje posjeduju izraze referenciranja varijabli:
  - **expression**: expressionName, variables(variable$_1$,variable$_2$,..,variable$_n$),
  - **if**: ifName, expressions(expression$_1$,expression$_2$,..,expression$_n$), variables(variable$_1$,variable$_2$,..,variable$_n$),
  - **while**/**repeatUntil**: while/repeatUntilName, expression, activity,
  - **pickAlarmEvent**: pickAlarmEventName, [expression$_1$(durationExpression)|expression$_2$(deadlineExpression)], activity[6],
  - **forEach**: forEachName, boolean(parallelism),expression$_1$(startCounterValue), expression$_2$(finalCounterValue), expression$_3$(completionCondition), scopeName,

[4] U obzir se neće uzimati aktivnosti: compensatonHandler, terminationHandler i faultHandler jer se vrši formalna provjera normalnog izvršavanja procesa.
[5] Aktivnosti eventHandler će se prikazivati kao zasebni *Promela* procesi koji se paralelno izvršavaju s procesima scope za koje su zakačeni.
[6] activity se u simboličkom modelu uvodi ukoliko sadržana aktivnosti ima doticaj s varijablama (čitanje ili ažuriranje).

- **flow**: flowName, activities(activity$_1$,..,activity$_n$), expressions(expression$_1$..,expression$_n$)[7], synchronizations(synchronization$_1$,..,synchronization$_n$),
  gdje se synchronization definira kao:
  synchronizationName, activity,
  transitionExpressions(TCexpression$_1$,..,TCexpression$_n$),
  synchronizedActivities(activity$_1$..,activity$_n$).

U niz `synchronizations` se za svaku sadržanu aktivnost „A“ unutar aktivnosti `flow` smještaju informacije o aktivnostima koje su izvršavanjem sinkronizacijski ovisne o izvršavanju aktivnosti „A“, a to su one aktivnosti u koje ulaze sinkronizacijske poveznice iz aktivnosti „A“ i koje ne mogu početi s izvršavanjem dok aktivnost „A“ ne završi.
Slijedi primjer BPEL aktivnosti `flow` i pripadni simbolički model radi lakšeg shvaćanja uvedenog pojma `synchronization`.

Primjer BPEL aktivnosti `flow`:

```
<flow A>
   <links><link name="AB"></links>

    <activityA>
      <sources>
            <source linkName="AB">
               <transitionCondition>exprA</transitionCondition>
            </source>
      </sources>
    </activityA>

    <activityB>
      <targets>
           <joinCondition>exprB</joinCondition>
              <target linkName="AB">
      </targets>
    </activityB>
</flow>
```

[7] U niz `synchronizations` se pohranjuju samo izrazi `transitionConditions` budući da oni referenciraju varijable, dok izrazi `joinConditions` ne referenciraju varijable već samo poveznice `link` koje ulaze u predmetnu aktivnost kojoj je `joinCondition` pridružen.

Pripadni simbolički model:

```
flow: A(flowName), activities(activityA, activityB),
expressions(expression1..,expressionn),
synchronizations(exprA),
synchronizationA: Async(synchronizationName), A(activity),
synchronizedActivities(B),
synchronizationB: Bsync(synchronizationName), B(activity),
synchronizedActivities(none)
```

6.2.2. Obilježja sustava za provjeru

Formalnom provjerom provjeravat će se u ovom poglavlju definirana svojstva. Formalna provjera se provodi s pretpostavkom da u procesu postoje asinkrona paralelna izvršavanja. Paralelna asinkrona izvršavanja često mogu biti izvor nepredvidivosti ponašanja procesa zbog nepredvidivosti brzine izvršavanja asinkronih paralelnih komponenti. Primjeri ponuđenih BPEL konstrukcija koja nude paralelna izvršavanja unutar BPEL procesa su sljedeći:

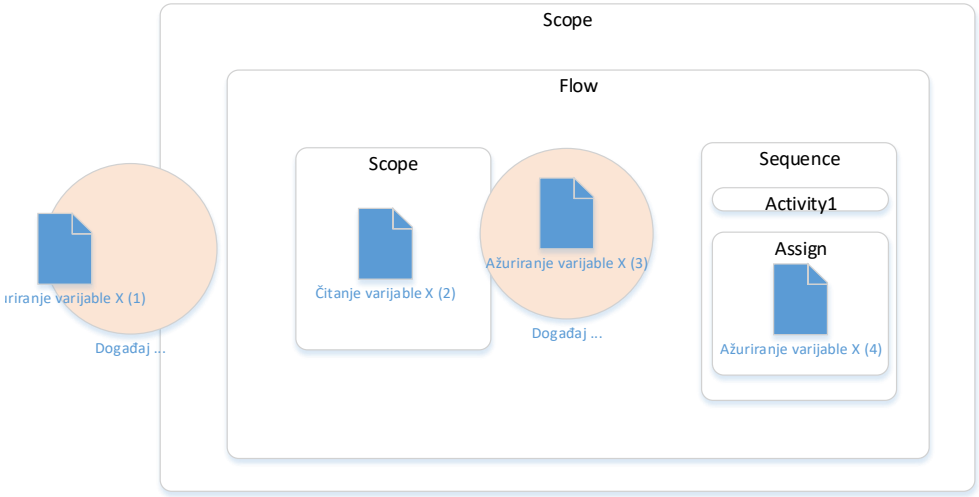
1. Svaka aktivnost koja nije sinkronizirana putem konstrukcija `link`, a nalazi se unutar aktivnosti `flow`, paralelno se izvršava s aktivnostima unutar zajedničke roditeljske aktivnosti `flow`, a mogu se paralelno izvršavati i sa sinkroniziranim aktivnostima unutar roditeljske aktivnosti `flow`,
2. Aktivnost `scope` može imati prikačene aktivnosti za obradu događaja tipa `onEvent` i `onAlarm`. `onEvent` se okida prijemom pripadajuće poruke, dok se `onAlarm` okida pripadajućim vremenskim trenutkom ili trajanjem i one se izvršavaju paralelno s aktivnošću `scope` za koju su prikačeni.

Svojstva, koja se na temelju razine obuhvaćene simboličkim opisom procesa opisanim u prethodnom poglavlju mogu provjeravati formalnom provjerom, vezana su za korištenje varijabli. Fokus formalne provjere je na varijablama i njihovoj uporabi zbog pretpostavke da upravo varijable nose ključne informacije poslovnog procesa i da je ključni čimbenik prepoznavanje nepredvidivosti njihovog korištenja kako se pogrešne informacije ne bi prenosile poslovnim procesom. Formalna provjera prezentirana u ovom radu izvršava sljedeće:

1. Identifikacija svih utrka u paralelnom pristupu varijablama (*ažuriranje+ažuriranje* ili *čitanje+ažuriranje*). Primjeri nekih scenarija paralelnog ažuriranja varijabli su sljedeći:
   a. aktivnost `invoke` kopira podatke iz `inputVariable` u ulazni parametar operacije i šalje ih, to jest poziva operaciju, a paralelno s tim varijabla `inputVariable` se u nekom drugom dijelu procesa ažurira. Brzina dohvata `inputVariable` od strane aktivnosti `invoke` i brzina ažuriranja `inputVariable` odlučuju što će proces poslati svom partneru, odnosno koje će ulazne parametre pozvana operacija primiti. Analogija je s aktivnošću `reply` ili s aktivnošću prijema poruke unutar aktivnosti `onEvent`,
   b. izrazi referenciranja varijabli (engl. `expressions`) (koji se mogu pojaviti kao izraz `transitionCondition` unutar konstrukcija `flow links`), kao izraz `condition` (unutar aktivnosti `while`, `repeatUntil` ili `if`) paralelno se izvršavaju s dijelom procesa koji ažurira istu varijablu koju izrazi referenciraju. Rezultat evaluacije izraza ovisit će o brzini dohvata dijeljene varijable i brzine ažuriranja varijable,
   c. aktivnosti `assign` koje ažuriraju identičnu varijablu nalaze se unutar paralelnih izvršavanja. Brzina izvršavanja spomenutih aktivnosti `assign` određuje koja će biti konačna vrijednost varijable. Konačnu vrijednost varijable postavit će ona aktivnost `assign` koja se zadnja izvršila jer je njena izmjena zadnja zapamćena i nije prebrisana nekom kasnijom izmjenom,
2. Identifikacija pogrešaka koje se nikada ne bi smjele dogoditi (engl. *invariants*):
   a. pokušaj čitanja ne inicijalizirane varijable, a jedan od primjera jeste:
      i. izraz referenciranja varijable (engl. `expression`) pokušava pročitati varijablu koja nije inicijalizirana čime se izvršavanje procesa zaustavlja u neželjenom koraku (engl. *death end*) jer izraz čeka na varijablu koja nikada neće biti inicijalizirana,
   b. korištenje varijable od strane aktivnosti komunikacije s partnerima (`invoke/reply/receive/onMessage`) koja nije definirana na temelju tipa koji njemu pripada, to jest dodjela varijable pogrešnog tipa.

Na temelju kategorizacije svojstava koja se mogu provjeravati formalnim modelom (Provjera formalnog modela), a koja mogu biti ili svojstva sigurnosti (koja uvijek moraju vrijediti) ili

svojstva životnosti (koja eventualno trebaju vrijediti), prethodno definirana svojstva provjere bi spadala u kategoriju svojstava sigurnosti koja uvijek moraju vrijediti.



Slika 43: Paralelno ažuriranje/čitanje BPEL varijabli

Na slici 43 se vidi primjer paralelnog ažuriranja i čitanja varijabli. Prvo paralelno ažuriranje varijable je unutar događaja zakačenog za glavnu aktivnost `scope`, dok je drugo paralelno ažuriranje varijable unutar događaja zakačenog za aktivnost `scope` unutar aktivnosti `flow`. Unutar aktivnosti `flow` se nalazi aktivnost `sequence` unutar koje se nalazi aktivnost `assign` s trećim paralelnim ažuriranjem iste varijable.

```
<scope>
        <eventHandlers>
                <onEvent>
                Ažuriranje varijable X(1)
                </onEvent>
        </eventHandlers>
        <flow>
                <flow>
                        <scope>Čitanje varijable X(2)
                                <eventHandlers>
                                <onEvent>
                                        Ažuriranje varijable X(3)
                                </onEvent>
                                </eventHandlers>
                        </scope>
                </flow>
                <flow>
                        <sequence>
                                <activity1>
                                <assign>
                                        Ažuriranje varijable X(4)
                                </assign>
                        </sequence>
                </flow>
        </flow>
</scope>
```

Slika 44: Paralelno ažuriranje/čitanje BPEL varijabli (XML)

Slika 44 predstavlja BPEL XML kod izvršavanja sa slike 43. Kao što se na slikama 43 i 44 vidi, jako je neintuitivno otkriti paralelna ažuriranja/čitanja varijabli u BPEL procesu kada se on modelira na razini XML koda uzevši u obzir i činjenicu poznavanja semantike BPEL aktivnosti i činjenicu poznavanja koje se BPEL aktivnosti paralelno izvršavaju. Upravo je to bio razlog za definiranje svojstava paralelnog pristupa varijablama kao cilja formalne provjere.

6.2.3. Pretvaranje u formalni model

Pretvaranje simboličkog modela, opisanog u Simbolički prikaz BPEL standarda, u formalni model funkcionira sukladno modelu opisanome u ovom poglavlju. Za prikaz proširenih automata koristit će se sintaksa *Promela* [46]. *Promela* programi se sastoje od procesa, kanala za razmjenu poruka i varijabli. Procesi su globalni objekti. Kanali za razmjenu poruka i varijable se mogu deklarirati ili globalno ili lokalno unutar procesa. Procesima se predstavlja ponašanje dok kanali za razmjenu poruka i varijable predstavljaju okolinu unutar koje se procesi izvršavaju. U jeziku *Promela* nema razlike između uvjeta i koraka izvršavanja (engl. *statements*). Svaki uvjet se također može promatrati kao korak izvršavanja i svaki korak izvršavanja je uvjetan na način da se može izvršiti ili ne. Mogućnost ili nemogućnost izvršavanja koraka unutar procesa jeste jedan od glavnih mehanizama sinkronizacije. Proces

može čekati dok ne dođe vrijeme za izvršavanje njegovog koraka. *Promela* nudi dovoljno konstrukcija kojima se mogu opisati izvršavanja unutar *Promela* procesa: dovoljan tip varijabli za formalnu provjeru (*typename, bit or bool, byte, short, int*), uvjetna izvršavanja (*while, if, do*), *atomic* izraze za izvršavanje koraka u jednom slijedu, skokove na dijelove koda, pokretanje jednog procesa iz drugog. Svaki *Promela* proces odgovara jednoj paralelnoj komponenti, to jest jednom pojedinačnom proširenom automatu koji se asinkrono izvršava s drugim paralelnim komponentama (*Promela* procesima) čime se simulira asinkroni proizvod dvaju proširenih automata opisan u poglavlju 6.1.1. U *Promela* sintaksi se ta pojava naziva *Asynchronous Interleaving Product* [47]. Budući da svaka paralelna komponenta predstavljena zasebnim *Promela* programom jeste prošireni konačni automat, on se korištenjem programa *.dot* [48] može pretvoriti u grafičku interpretaciju. Neželjeno svojstvo predstavljeno *promela never claim* [44] ili LTL sintaksom [45], čija se neprisutnost želi provjeriti, je također zasebni prošireni automat koji se sinkrono izvršava s prošrenim automatom sustava čime se simulira sinkroni proizvod dvaju proširenih automata opisan u Izgradnja formalnog modela konačnim automatima.

Glavna BPEL proces aktivnost `process` će biti prikazana glavnim *Promela* procesom (*init*), dok će svaka identificirana paralelna komponenta unutar procesa biti prikazana zasebnim *Promela* procesom koji će unutar glavnog procesa biti pokrenut u trenutnu kad dođe vrijeme za početak izvršavanja paralelne komponente. Isto tako, svaka aktivnost `scope` bit će također prikazana zasebnim *Promela* procesom zbog prirode te aktivnosti da može imati vlastite varijable i druge elemente vidljive samo unutar nje što je upravo razlog zbog kojega se ona i prikazuje na ovaj način, dakle zbog potrebe skrivanja varijabli i drugih elemenata koji trebaju biti vidljivi samo unutar svoje roditeljske aktivnosti `scope`. Budući da svaka aktivnost scope unutar sebe može imati još sadržanih aktivnosti `scope`, one će se pokretati unutar procesa roditeljske aktivnosti `scope` kada za to dođe vrijeme.
Varijable definirane na razini procesa bit će prikazane kao globalne *Promela* varijable, dok će varijable definirane na razini `scope` biti prikazane kao lokalne *Promela* varijable procesa kojim je predstavljen taj `scope`.
Slijedi algoritam preslikavanja iz simboličkog modela u formalni *Promela* model. Preslikavanja će se obavljati ručno no uzevši u obzir dolje definirane obrasce pretvaranja (definirana pravila), proces postaje egzaktan.

Za svaku varijablu `variableName` simboličkog modela (bilo da je globalna ili lokalna) definira se kompleksna varijabla `variableVar` (ili pojedinačno skup varijabli) koja sadržava sljedeće elemente:

```
typedef  variableVar{
         int variableName;
         int typeId="..";
         bool value=false;
         int accessWrite=0;
         int accessRead=0;}
```

- `int` varijabla `variableName` čija vrijednost pokazuje na jedinstveno ime/šifru varijable[8],
- `int` varijabla `typeId` čija vrijednost pokazuje na jedinstveno ime (identifikator) tipa `qname` ili `messageType` na temelju kojeg je varijabla definirana[9],
- `boolean` varijabla `value` čija vrijednost pokazuje da li je varijabla inicijalizirana ili ne i čija se vrijednost može iskoristiti u provjerama da li postoji pokušaj čitanja ne inicijalizirane varijable. Varijabla `value` inicijalno ima vrijednost `false` jer se u BPEL procesu varijable prilikom deklaracije ne mogu paralelno i inicijalizirati nego tek kasnije tijekom izvršavanja procesa kroz aktivnosti za ažuriranje varijabli (`assign`), kroz pozivanje izložene operacije procesa (`receive`) ili kroz pozivanje partnerske operacije i prihvat odgovora od nje (`invoke`). Prvi put kada se kroz neku od gornjih konstrukcija varijabli pridruži vrijednost, varijabla `value` poprimit će vrijednost `true`. Jednom kada je varijabla inicijalizirana, ona ostaje inicijalizirana,
- `int` varijabla `accessWrite` čija vrijednost služi kao pokazatelj (brojač) da li postoji pojava paralelnog ažuriranja varijabli. Ovaj brojač će imati vrijednost 0 kada varijablu nitko ne ažurira, dok će svaki pristup varijabli s ciljem ažuriranja njene vrijednosti uvećavati taj brojač za 1 i nakon završetka ažuriranja će ga smanjiti za 1. Na ovaj način brojač `accessWrite` će biti pokazatelj koliko paralelnih komponenti trenutno ima pristup varijabli s ciljem ažuriranja njene vrijednosti. Ukoliko je vrijednost brojača u bilo kojem trenutku veća od 0, to podrazumijeva da nad varijablom postoji pojava

[8] Nazivi varijabli i svih drugih konstrukcija mogu se šifrirati nekim vrijednostima `integer` budući da vrijednost `string` ne postoji u *Promela* jeziku.
[9] Identično pravilo vrijedi za šifriranje tipa varijable.

paralelnog ažuriranja čime se stvara utrka (engl. *race condition*) tko će brže ažurirati varijablu i čija će ažuriranje postaviti konačnu vrijednost varijable. Kod utrke ažuriranja varijabli, konačnu vrijednost varijable postavit će komponenta koja je bila najsporija, to jest ona koja je zadnja ažurirala varijablu,

- `int` varijabla `accessRead` čija vrijednost služi kao pokazatelj (brojač) da li postoji pojava paralelnog čitanja varijable. Ovaj brojač će imati vrijednost 0 kada varijabli nitko ne pokušava pročitati vrijednost, dok će svaki sljedeći pristup varijabli radi čitanja njene vrijednosti uvećavati taj brojač za 1 i nakon završetka čitanja će ga smanjiti za 1. Na ovaj način brojač `accessRead` će biti pokazatelj koliko paralelnih komponenti trenutno ima pristup varijabli s ciljem čitanja njene vrijednosti. Ukoliko je vrijednost brojača u bilo kojem trenutku veća od 0, to podrazumijeva da nad varijablom postoji pojava paralelnog čitanja čime se stvara utrka (engl. *race condition*) tko će brže pročitati varijablu. Utrka čitanja varijable je sama po sebi bezopasna i neće se provjeravati njeno postojanje budući da nadmetanje u brzini čitanja ni na koji način ne utječe na tijek izvršavanja procesa.

Formalni model procesa neće prikazivati konkretne vrijednosti varijabli, to jest neće raditi provjere nad konkretnim vrijednostima varijabli, nego će provjeravati:

- postojanje utrke ažuriranja varijable, gdje se utrka ažuriranja varijabli definira kao pojava kada dvije ili više paralelnih komponenti istovremeno želi ažurirati varijablu,
- postojanje utrke čitanja i ažuriranja varijable, gdje se utrka čitanja i ažuriranja varijabli definira kao pojava kada istovremeno postoje barem jedna paralelna komponenta koja želi čitati varijablu i barem jedna paralelna komponenta koja želi ažurirati varijablu,
- pokušaj čitanja neinicijalizirane varijable,
- pokušaj korištenja varijable nepripadajućeg tipa.

Postojanje pristupa varijablama s ciljem čitanja ili pisanja provjeravat će se pomoću varijabli `accessRead` i `accessWrite` čije vrijednosti pokazuje da li netko čita ili ažurira varijablu i koliko je paralelnih komponenti trenutno čita ili ažurira. Analiza vrijednosti spomenutih varijabli je sljedeća:

```
accessRead/Write==0-> nema trenutnog čitanja/ažuriranja varijable
```

`accessRead/Write==1-> 1 paralelna komponenta trenutno čita/ažurira varijablu`
`accessRead/Write>1-> 2 ili više paralelnih komponenti trenutno čitaju/ažuriraju varijablu`

Željeno ponašanje, kojim se postojanje utrke ažuriranja varijable smatra negativnom pojavom, glasi:

`uvijek(accessWrite<=1)` (LTL sintaksa: `[]{accessWrite<=1}`)

Postojanje utrke čitanja i ažuriranja varijable će se provjeravati kombinacijom vrijednosti varijabli `accessRead` i `accessWrite` čije vrijednosti pokazuju koliko paralelnih komponenti trenutno čita, odnosno ažurira vrijednost varijable. Željeno ponašanje, kojim se provjerava postojanje utrke čitanja i ažuriranja varijable, glasi:

`uvijek(ako accessWrite>0 -> accessRead=0)`
(LTL sintaksa: `[]{(accessWrite>0)->(accessRead==0)}`)

Neželjeno ponašanje postojanja utrke čitanja i ažuriranja varijable glasi:

`accessWrite>0 && accessRead>0`

Pokušaj čitanja neinicijalizirane varijable će se provjeravati pomoću varijable `value` čija vrijednost pokazuje da li je varijabla inicijalizirana ili ne. Željeno ponašanje, kojim se provjerava pokušaj čitanja neinicijalizirane varijable, glasi:

`uvijek(ako value=false -> accessRead=0)`
(LTL sintaksa: `[]{(value==false)->(accessRead==0)}`)

Neželjeno ponašanje pokušaja čitanja neinicijalizirane varijable glasi:

`value=false && accessRead>0`

Pristup varijablama definiranim na temelju mehanizma `messageType` neće se provjeravati na razini dijelova, nego će se provjeravati pristup kompletnoj varijabli bilo da se želi pristupiti samo jednom dijelu varijable, bilo da se želi pristupiti cijeloj varijabli, to jest oba slučaja će se tretirati na identičan način. Uvođenje oznake `qName` ili `messageType` se uvodi samo zbog svrhe otkrivanja dodjele pogrešnog tipa varijable aktivnostima za komunikaciju s partnerima što će također biti jedna od provjera sustava. Nakon prikaza formalnog opisa aktivnosti

invoke koji slijedi ispod, bit će prikazan način provjere pogrešno dodijeljene varijable, to jest dodjele pogrešnog tipa varijable.

Svaka aktivnost komunikacije s partnerima invoke, reply, receive, onMessage i pickMessageEvent simboličkog modela u formalnom modelu će biti prikazana na sljedeći način (za primjer je uzeta aktivnost invoke, no analogno se prikazuje svaka od spomenutih aktivnosti komunikacije s partnerima).
Prvenstveno se za svaku aktivnost invoke u formalnom modelu uvodi varijabla WSDL operacije pridružene aktivnosti invoke u BPEL procesu. Sadržana polja varijable (inputMsgType i outputMsgType) predstavljaju simbole tipova (messageType) na temelju kojih su definirane ulazne i povratne poruke operacije.

```
typedef  invokeVarOperation{
         int inputMsgType="..";
         int outputMsgType="..";}
```

Pripadni *Promela* kod koji će se izvršavati unutar *Promela* procesa, kada za izvođenje aktivnosti invoke dođe red, je sljedeći:

```
/*Početak aktivnosti invoke*/
atomic{
inputVariable.accessRead++; outputVariable.accessWrite++;
(input/output)AssignedVariableType=input/outputVariable.typeId;}

/*Kraj aktivnosti invoke*/
atomic{ inputVariable.accessRead--; outputVariable.accessWrite--;}
```

Na gore opisani način izvršavaju se pripadne rezervacije nad varijablama koje aktivnost invoke koristi i koje se mogu kasnije iskoristiti za provjeru paralelnih pristupa varijablama. Uvođenje informacije o pridruženoj operaciji se može iskoristiti za provjeru da li je pravilan tip varijable pridružen aktivnosti invoke. Dakle, ako je aktivnosti invoke pridružena operacija čiji je ulazni parametar messageType „A“, a u procesu joj se pridruži varijabla definirana na temelju tipa „B“, simboličkim opisom procesa i automatiziranim prebacivanjem u formalnu model, otkrit će se kršenje na lakši način naspram statičke analize BPEL XML koda

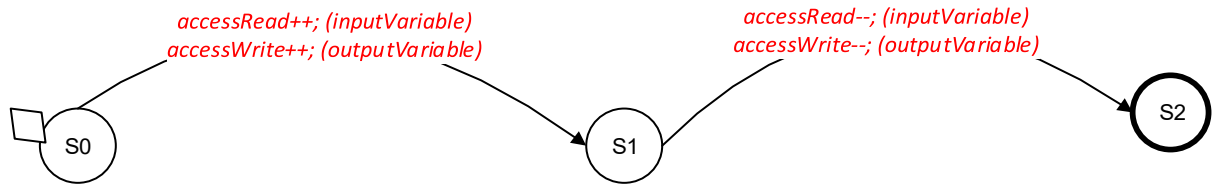
i uočavanja pogreške. U formalnom modelu je apstrahirano slanje poruke partneru i zamišljeno je da se u jednom koraku odradi dodjela vrijednosti ulaznoj varijabli i slanje poruke partneru, kao i prijem odgovora od istoga, to jest kopiranje primljene vrijednosti u izlaznu varijablu. Ukoliko bi se slanje poruke, to jest čitanje iz ulazne varijable i prijem poruke, to jest pisanje u primljenu varijablu obavljalo kroz različite korake, tada bi se razdvojili koraci `inputVariable.accessRead++` i `outputVariable.accessWrite++` koji se sada obavljaju u jednom koraku u sklopu izraza `atomic`.

Željeno ponašanje, kojim se provjerava da li je aktivnosti `invoke` pridružen pravilan tip varijable, bi glasilo:

`uvijek (invokeVarOperation.inputMsgType=inputAssignedVariableType)`
(LTL sintaksa: `[]{invokeVarOperation.inputMsgType==inputVariableType}`).

Na oglednom primjeru simboličkim opisivanjem procesa i generiranjem formalnog modela na temelju simboličkog te kasnijom formalnom provjerom nad formalnim modelom se otkriva da je aktivnosti `invoke` dodijeljen pogrešan tip varijable.

Na slici 45 grafički je prikazan prošireni automat BPEL aktivnosti `invoke` sa stanjima.



Slika 45: Prošireni automat BPEL aktivnosti `invoke`

Stanja BPEL aktivnosti `invoke` su sljedeća:

- `S0 - proces je spreman za izvršavanje prvih akcija aktivnosti invoke:`
  - `inkrementiranje elementa accessWrite i accessRead varijabli inputVariable i outputVariable,`
- `S1 – proces je spreman za izvršavanje akcija:`

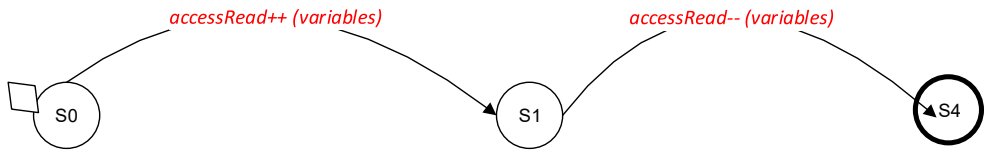
- dekrementiranje elementa accessWrite i accessRead varijabli inputVariable i outputVariable (pretpostavka je da je poruka poslana i uspješno primljena),

- S2 – proces je spreman za izvršavanje slijedne aktivnosti:
  - aktivnost invoke je završila.

Svaki izraz referenciranja varijabli expression simboličkog modela u formalnom modelu će biti prikazan sljedećim *Promela* kodom koji će se izvršavati unutar *Promela* procesa kada za izvođenje izraza expression dođe red i u kojem se vrše rezervacije i otpuštanja čitanja nad varijablama referenciranim od strane izraza:

```
/*Početak expression izraza*/
atomic{variable[1,n].accessRead++;}

/*Kraj expression izraza*/
atomic{variable[1,n].accessRead--;}
```

Na slici 46 grafički je prikazan prošireni automat BPEL izraza expression sa stanjima.



Slika 46: Prošireni automat BPEL izraza expression

Aktivnosti koje koriste izraze referenciranja varijabli expression su: if, while/repeatUntil i pickAlarmEvent.

Svaka aktivnost while/repeatUntil simboličkog modela u formalnom modelu će biti prikazana sljedećim *Promela* kodom koji će se izvršavati unutar *Promela* procesa kada za izvođenje aktivnosti while/repeatUntil dođe red:

```
atomic{
variable[1,n].readAccess++;}
atomic{
variable[1,n].readAccess--;}
containedActivity ako dotiče varijable na neki način;
```

Prethodno opisani kod za izvršavanje aktivnosti `while/repeatUntil` nije uzeo u obzir sadržanu aktivnost unutar `while/repeatUntil` osim ako ista na neki način ne referencira varijable. Prošireni automat prethodno opisanog načina prikazivanja aktivnosti `while/repeatUntil` (ukoliko sadržana aktivnost nije uzeta u obzir) odgovarao bi proširenom automatu `expression` izraza na slici 46.

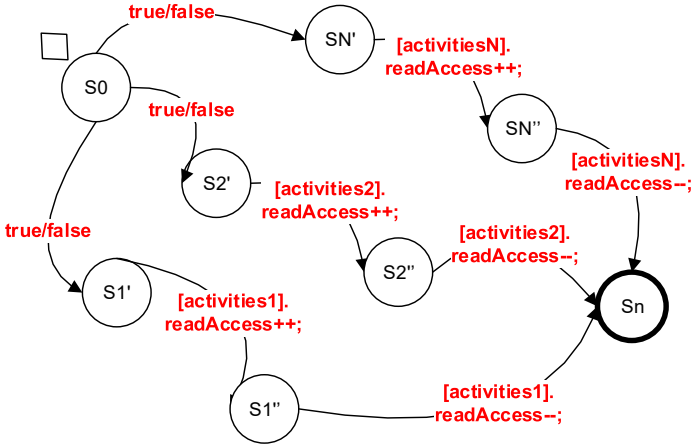
Svaka aktivnost `if` simboličkog modela u formalnom modelu će biti prikazana sljedećim *Promela* kodom koji će se izvršavati unutar *Promela* procesa kada za izvođenje aktivnosti `if` dođe red:

```
if
:: (simlacija:true/false)->
      atomic{/*expression1 počinje*/
      variable[1.1,1.n].readAccess++;}
      atomic{/*expression1 završava*/
      variable[1.1,1.n].readAccess--;}
      containedActivity ako dotiče varijable na neki način;
:: (simlacija:true/false)->
      atomic{/*expressionN počinje*/
      variable[n.1,n.n].readAccess++;}
      atomic{/*expressionN završava*/
      variable[n.1,n.n].readAccess--;}
      containedActivity ako dotiče varijable na neki način;
fi
```

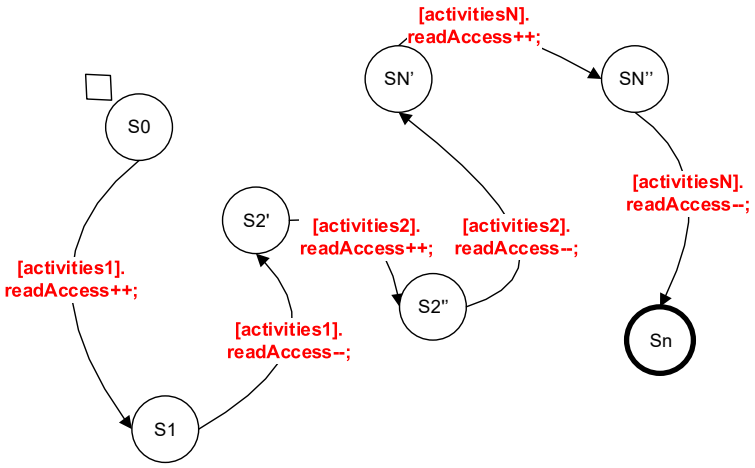
Na gore opisani način se simuliraju ispitivanja uvjeta u aktivnosti `if` budući da ona dotiču varijable s ciljem čitanja. Izvršavanje sadržanih aktivnosti unutar grana aktivnosti `if` nije uzeto u obzir uz pretpostavku da iste ne dotiču varijable no, ukoliko se unutar neke grane aktivnosti `if` pojavi aktivnost koja dotiče varijable, ona se u formalnom modelu prikazuje na svoj pripadni način i izvršava uz pretpostavku da je njen pripadni uvjet izvršavanja istinit.

U stvarnom izvršavanju aktivnosti `if` uvjeti na granama se ne moraju nužno svi ispitati, nego samo oni na koje dođe red do ispunjenja određenog uvjeta. Jedini realni slučaj kada se svi ispitaju jeste ukoliko niti jedan ne bude istinit pa se dođe do grane `else`. Radi provjere paralelnog ažuriranja varijabli, u gornjem formalnom modelu se kreće s pretpostavkom da se svi uvjeti na granama ispituju i sve aktivnosti na pripadnim granama (ukoliko dotiču varijable)

izvršavaju. Prošireni automat prethodno opisanog načina prikazivanja aktivnosti `if` (bez simuliranja izvršavanja sadržanih aktivnosti na granama), ukoliko se simulira izvršavanje uvjeta samo jedne grane, naveden je na slici 47. Slika 48 prikazuje prošireni automat ukoliko se simuliraju izvršavanja na svim granama aktivnosti `if` što je slučaj koji će se provoditi u formalnom modelu.



Slika 47: Prošireni automat BPEL aktivnosti `if` *(1)*



Slika 48: Prošireni automat BPEL aktivnosti `if` *(2)*

BPEL aktivnost `pick` jeste aktivnost koja pokreće izvršavanje ugniježđene aktivnosti na temelju događaja koji se desio pri čemu događaj može biti prijem poruke ili vremenski aktiviran događaj. Već su prethodno spomenuti simbolički načini opisivanja događaja aktivnosti `pick` (`pickAlarmEvent`, `pickMessageEvent`). Pripadni *Promela* kod aktivnosti `pick` u formalnom modelu će biti prikazan na sljedeći način. Za svaku aktivnost

pickMessageEvent u formalnom modelu uvodi se varijabla WSDL operacije pridružene aktivnosti pickMessageEvent na sljedeći način (analogija s aktivnosti invoke):

```
typedef  pickMessageEventOperation[1,N]{
         id inputMsgType="";}
```

Pripadni *Promela* kod, koji će se izvršavati unutar *Promela* procesa kada za izvođenje aktivnosti pickMessageEvent dođe red, također vrši rezervaciju pisanja nad pripadnom varijablom u koju se sprema primljena vrijednost ukoliko je riječ o aktivnosti pickMessageEvent ili rezervacije čitanja nad varijablama koje izraz referencira ukoliko je riječ o sadržanoj aktivnosti pickMessageEvent. Kod je sljedeći:

```
if
:: (true msg1)->
     atomic{/*pickMessageEvent[1,N] počinje-analogija s receive*/
     variable1.accessWrite++;
     assignedVariableType1=variable1.typeId;}
     variable1.accessWrite--;
:: (true time/duration)->
     atomic{/*timeMessageEvent[1,N] počinje-analogija s expression*/
     variable[1,N].accessRead++;}
     atomic{
     variable[1,N].accessRead--;}
     containedActivity ako dotiče varijable na neki način;
fi
```

Aktivnost forEach paralelno ili serijski izvršava sadržanu aktivnost scope predefinirani broj puta gdje se broj izvršavanja sadržane aktivnosti scope izračunava kao razlika izraza finalCounterValue i startCounterValue te koja može imati prijevremeni uvjet završetka completionCondition kojim se izvršavanje sadržanih iteracija aktivnosti scope može prijevremeno završiti.

Pripadni *Promela* kod koji će se izvršavati unutar *Promela* procesa kada za izvođenje forEach aktivnosti dođe red, također vrši rezervacije čitanja nad svima varijablama referenciranima od strane izraza finalCounterValue, startCounterValue i completionCondition ukoliko je prisutan. Uočava se da sa stajališta paralelnog pristupa

varijablama nije bitno da li se aktivnost `forEach` izvršava serijski ili paralelno. Pripadni formalni kod je sljedeći:

```
atomic{/*rezervacije čitanja nad varijablama*/
     variable[1/2/3.1].readAccess++,..,variable[1/2/3.n].readAccess++;}
atomic{/*otpuštanje rezervacija čitanja nad varijablama*/
     variable[1/2/3.1].readAccess--,..,variable[1/2/3.n].readAccess--;}
containedScopeExecution ako dotiče varijable na neki način;
```

Aktivnost `flow` služi za sinkronizacijski međuovisno ili paralelno izvršavanje sadržanih aktivnosti. U Standard BPEL 2 objašnjen je mehanizam sinkronizacijske međuovisnosti izvršavanja unutar aktivnosti `flow`. Budući da je sinkronizacijski međuovisno međuizvršavanje pojava koja se ne susreće standardno, ponovno će se navesti simbolički opis aktivnosti `flow` kako bi se lakše shvatio pripadni formalni *Promela* kod koji će se izvršavati unutar *Promela* procesa kada za izvođenje aktivnosti `flow` dođe red. Pripadni simbolički model iz poglavlja 6.2.1 je sljedeći:

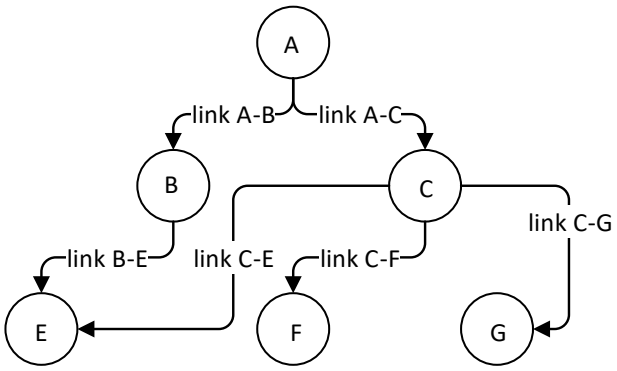
```
flow: flowName, activities(activity1,..,activityn),
expressions(expression1..,expressionn)[10],
synchronizations(synchronization1,..,synchronizationn),
     gdje se synchronization definira kao:
     synchronizationName, activity,
     transitionExpressions (TCexpression1,..,TCexpressionn),
     synchronizedActivities(activity1..,activityn).
```

Kao i kod aktivnosti `if`, i kod aktivnosti `flow` krenut će se s pretpostavkom da se ispituju svi uvjeti prijelaza (`transitionCondition`) što naravno kod realnog izvršavanja procesa ne mora biti slučaj i često i nije no budući da je cilj provjera paralelnog pristupa varijablama i čitanje neinicijaliziranih varijabli, bitno je da se u formalnu provjeru uključe svi pristupi varijablama stoga se u obzir uzimaju ispitivanja svih gore navedenih uvjeta poveznica. Kod aktivnosti `flow` bitno je napravitii poredak sinkroniziranih izvršavanja budući da se neće sve aktivnosti izvršavati u isto vrijeme, neke čak ne mogu ni početi dok njihove prethodne

[10] U niz `expressions` se pohranjuju samo izrazi `transitionConditions` budući da oni referenciraju varijable, dok izrazi `joinConditions` ne referenciraju varijable već samo poveznice `link` koje ulaze u predmetnu aktivnost kojoj je `joinCondition` pridružen.

aktivnosti ne završe tako da će redoslijed izvršavanja aktivnosti biti jednak redoslijedu izvršavanja `joinCondition` i `transitionCondition` uvjeta, odnosno redoslijedu rezervacije nad varijablama u formalnom modelu. Neka je raspored sinkroniziranih izvršavanja aktivnosti `flow` sukladan slici 49. Pripadni simbolički model vezan za `synchronization` bi izgledao:

```
synchronizationA: .., A(activity),
     transitionExpressions (link A-B, link A-C),
     synchronizedActivities(B, C),
synchronizationB: .., B(activity),
     transitionExpressions (link B-E),
     synchronizedActivities(E),
synchronizationC: .., C(activity),
     transitionExpressions (link C-E, link C-F, link C-G),
     synchronizedActivities(E, F, G),
synchronizationE/F/G:.., E/F/G(activity),
synchronizedActivities(none).
```



Slika 49: Sinkronizirana izvršavanja unutar aktivnosti `flow`

Pripadni formalni kod gornje aktivnosti `flow` je sljedeći:

```
executionA ako dotiče varijable na neki način;
atomic{/*rezervacije čitanja nad varijablama*/
     variableTClinkAB/AC[1,n].readAccess++;}
atomic{/*otpuštanje rezervacija čitanja nad varijablama*/
     variableTClinkAB/AC[1,n].readAccess--;}
executionB/C ako dotiču varijable na neki način;
```

```
atomic{/*rezervacije čitanja nad varijablama*/
     variableTClinkBE/CE/CF/CG[1,n].readAccess++;}
atomic{/*otpuštanje rezervacija čitanja nad varijablama*/
     variableTClinkBE/CE/CF/CG[1,n].readAccess--;}

executionE/F/G ako dotiču varijable na neki način;
```

Postupak formalne provjere procesa izvršava se kroz niz dole navedenih koraka kojima se generira formalni model izražen *Promela* konačnim automatima:

- korak 1: označavanje elemenata BPEL procesa relevantnih za formalnu provjeru, a definiranih u Simbolički prikaz BPEL standarda,
- korak 2: izgradnja simboličkog opisa označenih elemenata BPEL procesa sukladno definiranim pravilima u Simbolički prikaz BPEL standarda,
- korak 3: izgradnja *Promela* formalnog modela na temelju simboličkog opisa BPEL procesa iz koraka 2.

Formalna provjera kao rezultat daje informaciju o putanji izvršavanja procesa na kojoj željeno ponašanje nije prisutno [49] te se kretanjem korak po korak kroz generiranu putanju i otkrivanjem točnog koraka pogreške, na temelju simboličkog modela lako može otkriti lokacija pogreške u BPEL procesu. Na taj način je postignuta dvosmjernost formalne provjere što će biti demonstrirano

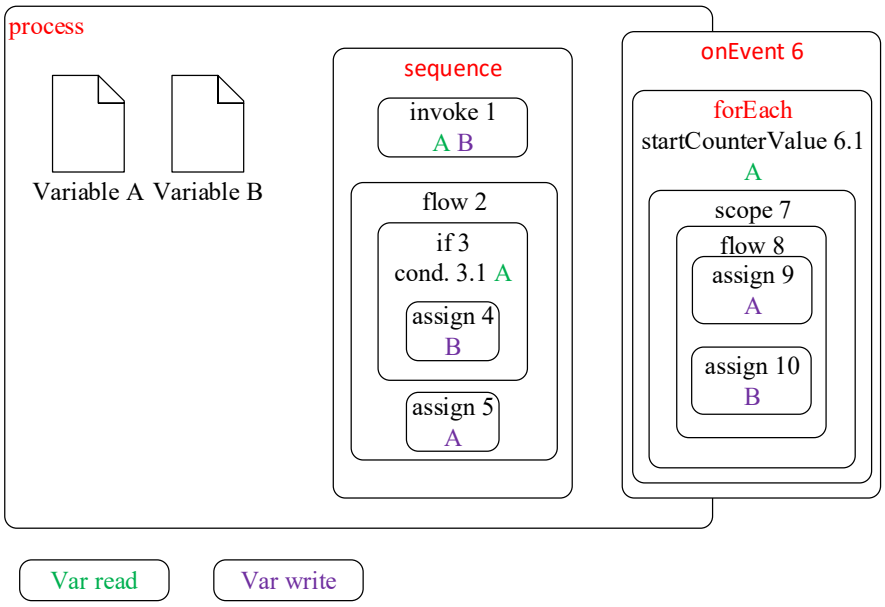
Slijedi primjer BPEL procesa s (brojevima označenim) paralelnim izvršavanjima unutar njega.

Tablica 12: BPEL *proces s paralelnim izvršavanjima*

```
<process>
<variables>
<variable name="A" ../><variable name="B" ../>
</variables>
<sequence>
   <invoke1 portType=".."    operation="Op1"
   inputVariable="A" outputVariable="B"/>
    <flow2>
     <if3>
         <condition3.1>expr reading A</condition>
                  <assign4>writing B</assign4>
         <else>Activity2</else>
     </if3>
     <assign5>writing A</assign5>
    </flow2>
</sequence>
      <eventHandlers>
            <onEvent6>
            <forEach parallel="yes">
                  <startCounterValue6.1>expr reading A
                  </startCounterValue6.1>
                  <finalCounterValue>10</finalCounterValue>
                  <scope7>
                  <flow8>
                 <assign9>writing A</assign9>
                  <assign10>writing B</assign10>
                  </flow8>
                 </scope7>
            </forEach>
            </onEvent6>
      </eventHandlers>
</process>
```

U BPEL procesu u tablici 12 paralelna izvršavanja su naglašena crvenom bojom. S aktivnosti `process` paralelno se može izvršavati aktivnost `onEvent` (`onEvent 6`). Početak njenog izvršavanja (ukoliko se ista aktivira pojavom događaja) može se poklopiti s izvršavanjem aktivnosti `invoke` (`invoke 1`) ili s aktivnosti `flow` (`flow 2`). Istovremenim izvršavanjem

i s jednom i s drugom aktivnosti bi se pojavila pojava paralelnog čitanja/ažuriranja iste varijable. To su dakle dvije mogućnosti. Ukoliko bi se izvršavanje aktivnosti `onEvent 6` poklopilo s izvršavanjem aktivnosti `flow 2`, u tom slučaju bi se moglo poklopiti s izvršavanjem ili aktivnosti `if` (`if 3`, `assign 4`) ili s izvršavanjem aktivnosti `assign` (`assign 5`). To su dakle već tri mogućnosti paralelnog čitanja/ažuriranja iste varijable na samo jednom malom dijelu BPEL procesa. Simbolička slika procesa iz tablice 16 prikazana je na slici 50.



Slika 50: BPEL proces iz tablice 12

Slijedi simbolički opis BPEL procesa iz tablice 12 koji sustavno popisuje paralelna izvršavanja i izvršavanja koja referenciraju varijable (brojevima označena u procesu) sukladno dogovorenom načinu opisivanja u Simbolički prikaz BPEL standarda:

```
process: variables(A, B), eventHandlers(onEvent6),
activities(invoke1, flow2)
flow: flow2(name), activities(if3, assign5)
if: if3(name), expressions(expression 3.1)
expression: expression3.1(name), activities(assign4)
assign: assign5(name), A(write,toVariableName)
assign: assign4(name), B(write,toVariableName)
eventHandlers: onEvent6(name), activities(forEach)
forEach: forEach(name), boolean(parallelism,true),
expression(startCounterValue6.1), activities(scope7)
scope: scope7(name), activities(flow8)
```

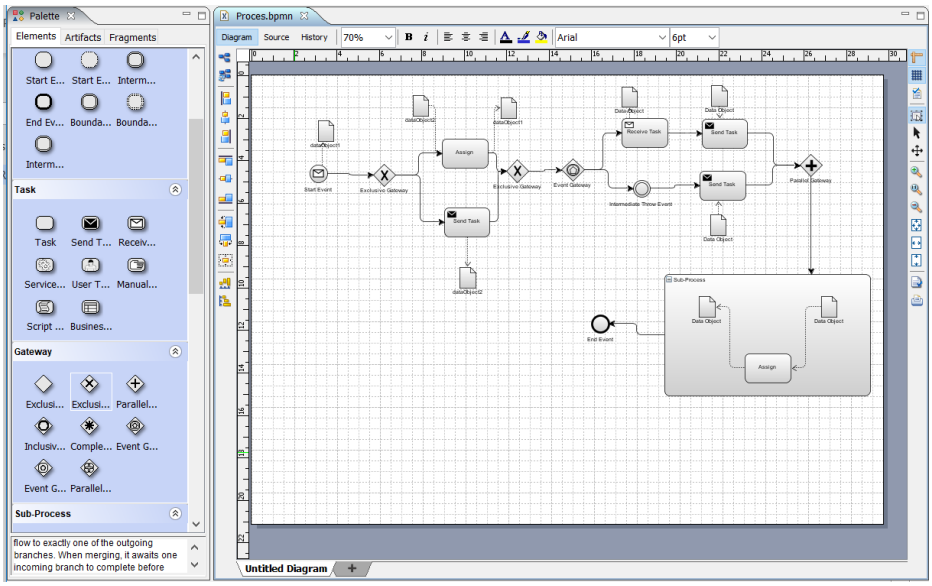
```
flow: flow8(name), activities(assign9, assign10)
assign: assign9(name), A(write,toVariableName)
assign: assign10(name), B(write,toVariableName).
```

BPEL proces sa slike i njegov simbolički model će se u Testiranje pseudokodova modela prevesti u formalni model prikazan *Promela* sintaksom koji će otkriti paralelna ažuriranja varijabli i navesti u kojem koraku formalnog modela se ona događaju.

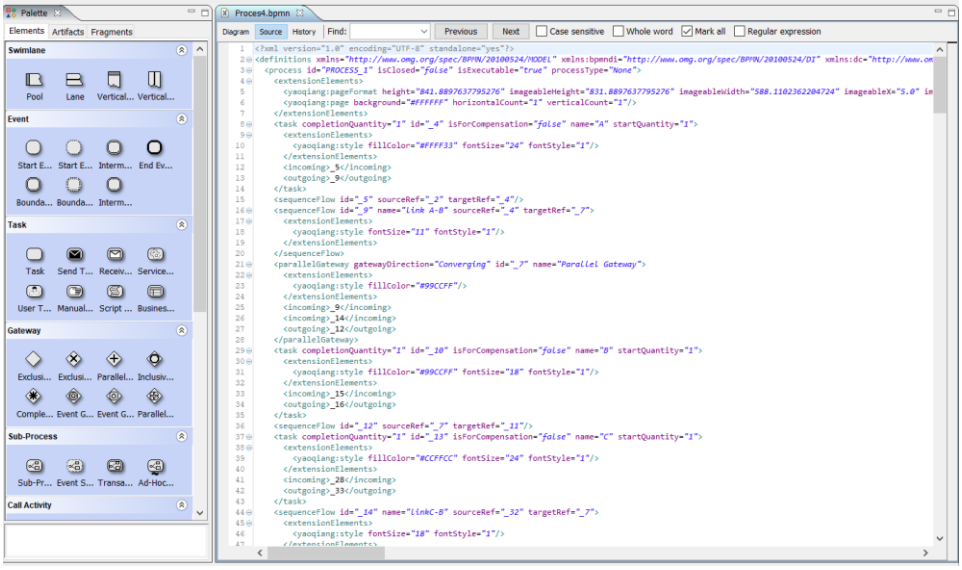
7. Radni okvir za razvoj BPEL procesa

7.1. Vizualno modeliranje BPEL procesa

Prva faza modela faznog razvoja procesa vezana je za vizualno modeliranje BPEL procesa BPMN standardom korištenjem pravila za vizualno predstavljanje BPEL aktivnosti i pripadnih WSDL konstrukcija prikazanih u Preslikavanje između BPMN i BPEL . Iako BPMN standard posjeduje brojne alate za vizualno modeliranje [22][23][24], za modeliranje u ovom radnom okviru odabran je alat *Open Source Yaoqiang BPMN Editor* [23] jer dozvoljava pristup pozadinskom XML BPMN kodu ukoliko se na XML kodu želi direktno odraditi željena izmjena umjesto na vizualnom dijelu BPMN modela, a direktan pristup pozadinskom BPMN XML kodu je bio i neizbježan budući da algoritam preslikavanja BPMN-BPEL iz vizualnog BPMN modela u izvršni BPEL model funkcionira na XML razini stoga je kao ulaz algoritmu potrebno proslijediti *.bpmn dokument koji sadržava XML kod vizualnog BPMN modela. Slika 51 prikazuje sučelje za vizualno modeliranje alata *Yaoqiang BPMN Editor*, a slika 52 način pristupa pozadinskom XML BPMN kodu. Da bi se vizualni BPMN model uspješno preslikao u izvršni BPEL model, potrebno je da bude „*valjan sa stajališta algoritma preslikavanja BPMN-BPEL*“, to jest da svi uzorci ponašanja unutar BPMN koda budu strukturirani na način da se mogu uspješno preslikati u neku od BPEL aktivnosti. stoga je zadaća na onome koji vizualno modelira proces poznavanje definiranih pravila budući da ne postoji nikakvo ograničenje nad *Yaoqiang BPMN Editor* alatom da se prilikom vizualnog modeliranja kreiraju proizvoljni BPMN uzorci.



Slika 51: Vizualno sučelje alata Yaoqiang BPMN Editor



Slika 52: Pristup pozadinskom XML kodu - Yaoqiang BPMN Editor

Nakon što je vizualni BPMN model završen, sprema se kao BPMN dokument. BPMN dokument je izlaz iz prvog dijela prve faze faznog razvoja modela (korak 1.1, Slika 6: Faze modela za fazni razvoj BPEL procesa).

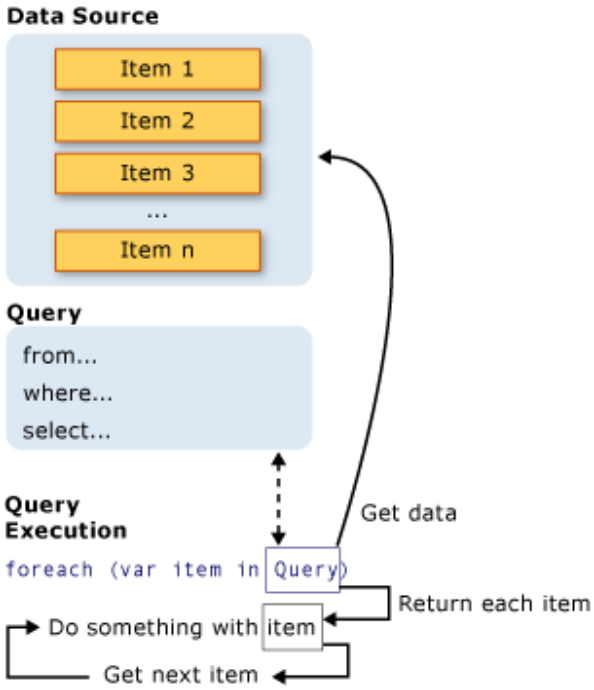
Sljedeći korak (korak 1.2, Slika 6: Faze modela za fazni razvoj BPEL procesa), je pretvaranje sadržaja BPMN dokumenta (BPMN XML koda) u izvršni BPEL dokument (BPEL XML kod). BPMN dokument se može spremiti kao XML dokument da bi bio prikladan ulaz za algoritam

preslikavanja BPMN-BPEL čiji su pseudokod i uparena XML preslikavanja opisani u Preslikavanje između BPMN i BPEL .

7.2. Implementacija algoritma preslikavanja BPMN-BPEL

Za implementaciju algoritma preslikavanja BPMN-BPEL u radnom okviru koristio se alat *Visual Studio Express 2017* [50]. Implementacija algoritma rađena je u programskom jeziku C# [51], korištenjem tehnologije *LINQ to XML* [52] koja se s ponuđenim konstrukcijama za izvlačenje podataka iz izvora (XML dokumenta) i kreiranje novih podataka (XML elemenata, atributa) pokazala dovoljno bogatom za pretvaranje jedne XML *Scheme* (BPMN) u drugu XML *Schemu* (BPEL). Osim korištene tehnologije, mogla se koristiti i W3C Document Object Model (DOM) tehnologija [53], ali je procijenjeno da je *LINQ to XML* tehnologija jednostavnija [54]. Programski jezik C# je odabran zbog prijašnjeg iskustva no mogao se odabrati neki drugi .NET programski jezik (Visual Basic [55]) ili bilo koji drugi programski jezik koji posjeduje funkcije za rad s XML sadržajem. Algoritmi pseudokoda objašnjeni u Preslikavanje između BPMN i BPEL se mogu implementirati funkcijama bilo kojeg programskog jezika koje nude mogućnosti: kreiranja/učitavanja XML dokumenta, kreiranja/čitanja XML elementa, kreiranja/čitanja XML atributa, izabiranja djece XML elemenata na temelju uvjeta koji zadovoljavaju (*tip elementa, posjedovanje/neposjedovanje djece određenog tipa, posjedovanje/neposjedovanje atributa određenog tipa i slično*). Kandidat za transformaciju XML *Schema* na samom početku rada je bio i XSLT [56] standard no zbog njegove striktnosti i relativne kompleksnosti (zaključci nakon analize XSLT standarda) i veće kontrole ukoliko se razvije vlastiti pseudokod, koji se za budući rad može dodatno prilagođavati i optimizirati, XSLT nije uzet u obzir.

Korištenje tehnologije LINQ (XML), odnosno njenih mogućnosti za implementaciju algoritma preslikavanja BPMN-BPEL, bit će prikazani na primjeru preslikavanja BPMN-BPEL XML koda iz Preslikavanje između BPMN i BPEL. Prije spomenute demonstracije, prikazat će se mogućnosti LINQ (XML) tehnologije. Svaki LINQ upit se sastoji od tri dijela: dohvaćanje izvora podataka, kreiranje upita, izvršavanje upita (slika 53).



Slika 53: Tri koraka izvršavanja LINQ upita

Dohvaćanje izvora podataka vrši se vrlo jednostavnom konstrukcijom kojom se učitava *.xml dokument: `XElement source = XElement.Load(@"*.xml").`

Kreiranje upita se izvršava na sljedeći način:

```
IEnumerable<XElement> selectedElements =
from el in sourceDocument.Elements("elementType")
where condition
select el;
//sva djeca elementi „elementType“ korijenskog elementa
source dokumenta
```

Navigacija nad elementima upita se izvršava na sljedeći način:

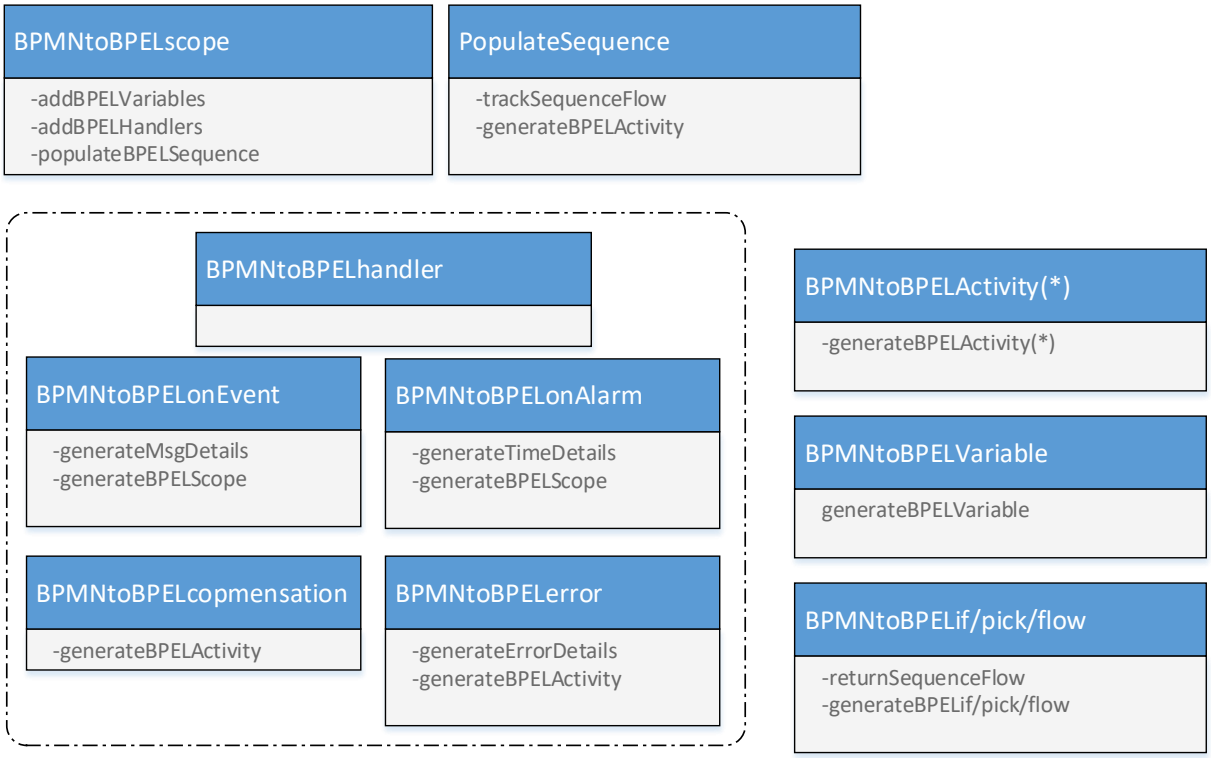
```
foreach (XElement child in selectedElements.Elements())
{if (child.Name == "XY") action
//izvršavanje akcija nad djecom XY elementa selectedElements elementa
```

Konstrukcije ponuđene od strane LINQ (XML) tehnologije dovoljne za implementaciju algoritma preslikavanja BPMN-BPEL-BPMN su sljedeće:

- `XElement element = new XElement("name")` – kreiranje XML elementa,

- `element.Add(element1)` – dodavanje XML elementa elementu,
- `XDocument destination = new XDocument(element)`- kreiranje *xml dokumenta,
- `destination.Save("*.xml")` – spremanje *xml dokumenta,
- `element.SetAttributeValue("variable","value")` – kreiranje XML atributa i dodavanje XML elementu,
- `element.Elements()/Element("name")` – izdvajanje djece elemenata XML elementa.

Slijedi prikaz agregiranih klasa (funkcija) korištenih tijekom implementacije algoritma preslikavanja BPMN-BPEL u LINQ (XML) tehnologiji.



Slika 54: Dijagram klasa algoritma preslikavanja BPMN-BPEL

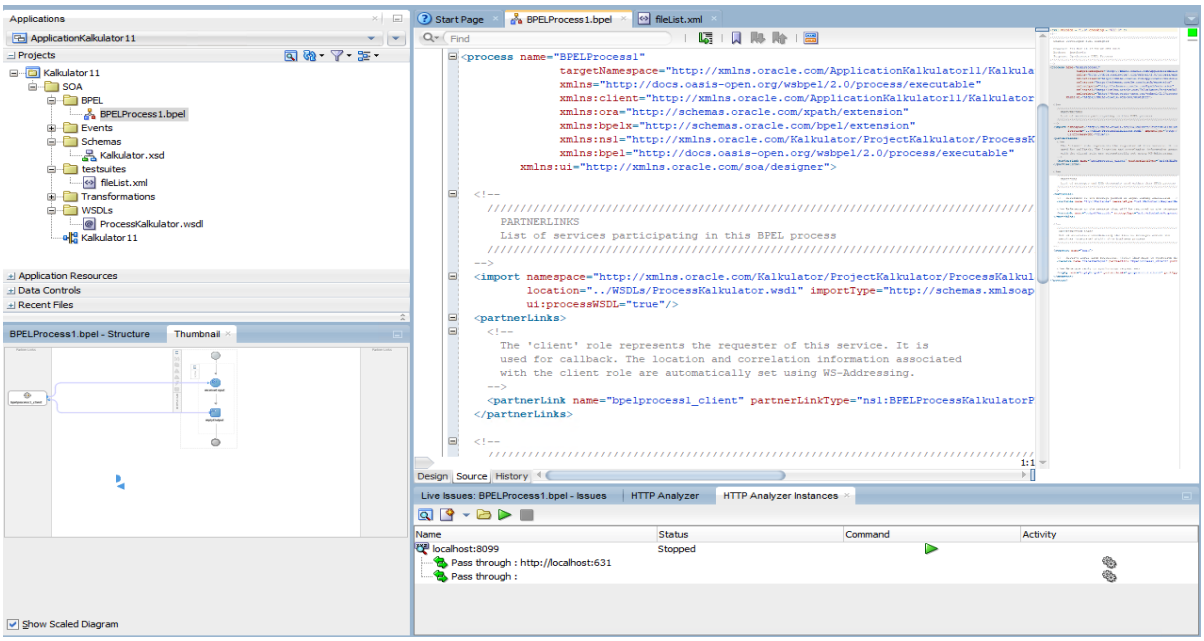
Na slici 54 prikazane su klase za popunjavanje BPEL elementa `process`. Opis klasa, pripadnih metoda koje implementiraju i međusobne ovisnosti je sljedeći:

- `BPMNtoBPELscope` – klasa koja se koristi za generiranje BPEL aktivnosti `scope/process` na temelju BPMN elementa `subProcess`. Ona unutar sebe implementira sljedeće metode/instanciranja:
  - `addBPELVariables` – metoda koja instancira klasu `BPMNtoBPELVariable` za svaki od pronađenih elemenata `dataObject` unutar BPMN elementa `subProcess`,
  - `addBPELHandlers` – metoda koja instancira neku od klasa `BPMNtoBPELhandler(*)` za svaki od pronađenih elemenata `inline event subProcess` unutar BPMN elementa `subProcess` pri čemu se vrsta instancirane klase određuje na temelju vrste čvora `startEvent` elementa `inline event subProcess`,
  - `populateBPELSequence` – instanciranje klase `populateBPELSequence` koja pak instancira neku od klasa `BPMNtoBPELActivity(*)` za generiranje pripadnih BPEL aktivnosti ili klasu `BPMNtoBPELif/pick/flow` za generiranje BPEL aktivnosti `if/pick/flow` koji se dodaju u glavni slijed `sequence` BPEL aktivnosti `scope/process` i koja unutar sebe uvijek pamti izlazni `sequenceFlow` zadnje otkrivene vizualne aktivnosti,
- `BPMNtoBPELhandler(*)` – klasa koja generira pripadnu BPEL aktivnost `*handler` i koja unutar sebe opet kreira instance klasa `BPMNtoBPELscope`, `BPMNtoBPELActivity(*)` ili `BPMNtoBPELif/pick/flow` za generiranje sadržanih BPEL aktivnosti unutar BPEL aktivnosti `*handler`,

Nakon što funkcije algoritma preslikavanja BPMN-BPEL generiraju dijelove WSDL sučelja (`message`, `operation` i `portType`) i pripadni BPEL XML kod koji se kao izlaz sprema u izlazni XML dokument, generirani dokumenti se dalje ručno nadograđuju s potrebnim konstrukcijama koje nisu obuhvaćene razinom formalne provjere ili se mogu uređivati nekim od alata za kreiranje BPEL procesa (korak 1.2, Slika 6: Faze modela za fazni razvoj BPEL procesa).

7.3. Uređivanje BPEL XML koda

U radnom okviru, za doradu generiranih WSDL sučelja i BPEL XML koda iz prve faze modela, korišten je alat *Oracle JDeveloper 12c Studio Edition Version* [57]. U tom alatu se kreira prazan BPEL dokument u koji se kopira generirani BPEL XML kod iz prve faze. Identičan postupak se vrši s pripadnim WSDL i XSD dokumentima u koje se ručno kopira generirani kod nastao na temelju preslikavanja pripadnih BPMN elemenata u WSDL `message`, `operation` i `portType` i u dijelove XSD elemenata.

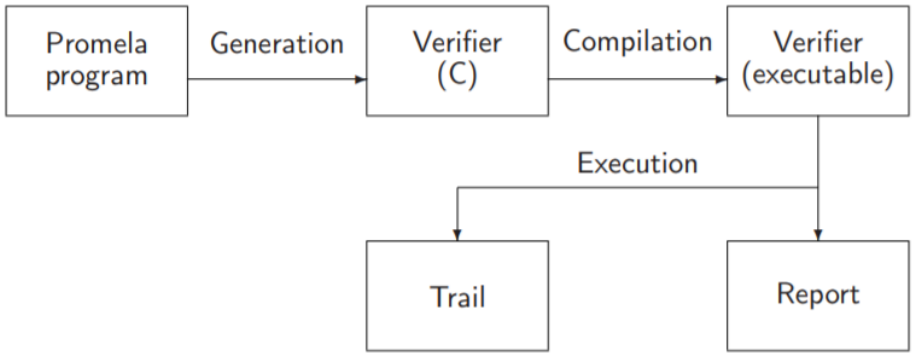


Slika 55: Oracle JDeveloper sučelje za uređivanje BPEL XML koda

Nakon što je izvršni BPEL model, to jest njegov XML kod dorađen sa svim elementima potrebnim za izvršavanje, on se radi potreba testiranja može pokrenuti korištenjem integriranog BPEL izvršnog servera koji se isporučuje kao komponenta alata *JDeveloper*. Ukoliko se pak BPEL proces želi postaviti u neko izvršno okruženje s ciljem da bude dostupan kao autonomna usluga ispoljena svojim WSDL sučeljem, može se postaviti (engl. *deploy*) u samostalni *WebLogic Server* čija je komponenta *Oracle BPEL Process Manager* [10] zadužena za interpretaciju BPEL izvršnog koda.

7.4. Formalna verifikacija izvršavanja

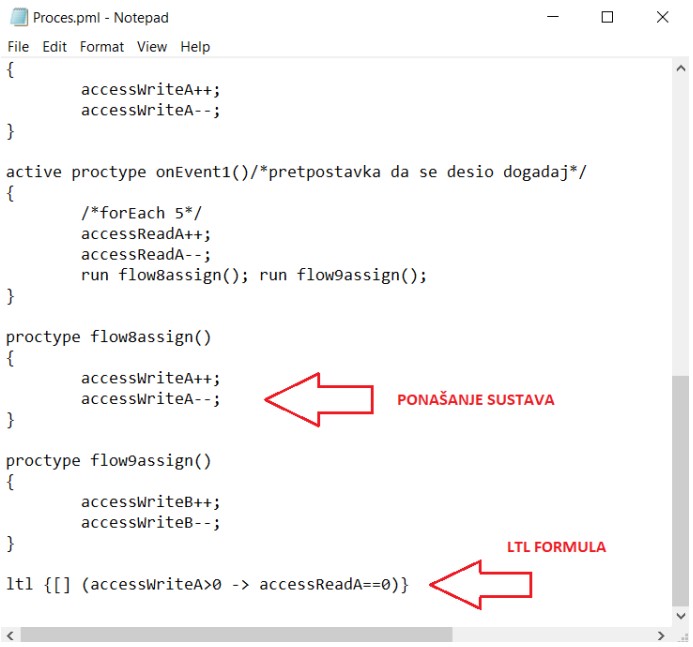
Treća faza modela za fazni razvoj BPEL procesa je vezana za formalnu verifikaciju izvršavanja BPEL procesa. Prvi korak treće faze (korak 3.1, slika 6) simbolički opisuje BPEL proces sukladno pravilima definiranim u Simbolički prikaz BPEL standarda. Za tu svrhu nije potreban nikakav poseban alat osim običnog tekstualnog uređivača kojim se može otvoriti BPEL XML kod te se numerički jednoznačno označiti potrebni elementi BPEL XML koda bitni za formalnu provjeru (sukladno tablici 12) koji će na temelju dogovorenog simboličkog označavanja činiti simbolički model procesa. Pretvaranje simboličkog modela (na temelju pravila definiranih u Pretvaranje u formalni model) u formalni model predstavljen *Promela* sintaksom se također vrši uporabom običnog uređivača teksta. Generirani formalni model procesa sprema se u formatu PML i služi kao ulaz u program *iSpin* [58] kojim se izvršava formalna verifikacija. *iSpin* jeste Tcl/Tk [59] skripta i predstavlja grafičko sučelje programa *spin* [60] razvijenog od strane Bell Labs [61]. Za uspješno izvršavanje verifikacije potrebni su C-*compiler* i C-*preprocessor* budući da *spin* na temelju ulaza (PML dokumenta koji sadržava prošireni automat sustava i neželjeno ponašanje izraženo *promela never claim* ili LTL sintaksom, slika 57) generira C izvršni program čijim se izvršavanjem provodi provjera ispravnosti svih ponašanja procesa (sustava) naspram neželjenog ponašanja te koja kao izlaz generira izvješće o provjeri zajedno s putanjom izvršavanja na kojoj je prisutno neželjeno ponašanje ukoliko takvo postoji (slika 56).



Slika 56: Spin arhitektura [41]

Spin osim iscrpne verifikacije cijelog ponašanja sustava, to jest svih njegovih ponašanja, može provoditi i nasumične simulacije izvršavanja. Tijekom svakog koraka simulacije prikazuje se trenutni korak programa koji se izvršio i vrijednosti varijabli nakon izvršenog koraka (engl.

state vector). Kod procesa verifikacije, ukoliko postoje putanje izvršavanja koje sadržavaju neželjena ponašanja, *spin* generira korake neželjenog ponašanja u izlazni dokument (engl. *error trail*). Otkriveno neželjeno ponašanje se naknadno može simulirati kako bi se otkrilo u kojim dijelovima procesa se točno pojavljuje. Budući da je za generiranje formalnog modela odabrana *Promela* sintaksa, nije bilo mogućnosti izbora korištenja nekog drugog alata osim *spin* alata jer je to sintaksa opisivanja proširenih automata razvijena za interpretaciju isključivo *spin* alatom. Alat *spin* sa svim njegovim opisanim funkcionalnostima ne predstavlja vlastiti doprinos, doprinos je smisleno oblikovan formalni model (PML ulaz) temeljem pravila definiranih u poglavlju 6.2.3 kojim se može vršiti verifikacija prisutnosti neželjenog BPEL ponašanja. Analizom ponuđenih metodologija za formalnu provjeru *Promela* je odabrana zbog dobrih referenci, dovoljno ponuđenih materijala za njeno razumijevanje i dobre dokumentacije vezane za instalaciju i uporabu pripadajućih alata. U Model za fazni razvoj BPEL procesa je naveden pregled rada autora iz područja formalne verifikacije te pregled ponuđenih metodologija za formalnu verifikaciju (provjeru modela) koji su se eventualno također mogli koristiti s tim da bi se proces izgradnje formalnog modela prilagodio odabranoj metodi za formalnu provjeru.



Slika 57: Primjer oglednog PML procesa i alata za njegovo uređivanje

Četvrta faza modela za fazni razvoj procesa vezana je za ispravljanje procesa ukoliko su u njemu dogodila neka pogreška temeljem procesa formalne provjere. Budući da se u trećoj fazi proces simbolično opiše prije nego se prevede u formalni model i budući da formalna verifikacija daje konkretnu informaciju u kojem dijelu programa je pronađena pogreška, lako

je pogrešku locirati u simboličkom modelu procesa, a temeljem njega i u izvršnom BPEL modelu te se BPEL model može ispraviti ukoliko postoji potreba za tim.

Zadnja peta faza modela za fazni razvoj procesa vezana je za statičku provjeru pravila koje je BPEL specifikacija definirala kao pravila koja se statičkom analizom procesa moraju provjeriti [3]. Budući da su to pravila koja se provjeravaju nad BPEL aktivnostima (XML elementima) i spadaju u domenu rada s XML sadržajem, provjera odabranih pravila se također implementirala korištenjem LINQ (XML) [52] tehnologije, dakle istovjetnom tehnologijom koja se koristila za algoritam preslikavanja BPMN-BPEL-BPMN.

8. Testiranje pseudokodova modela

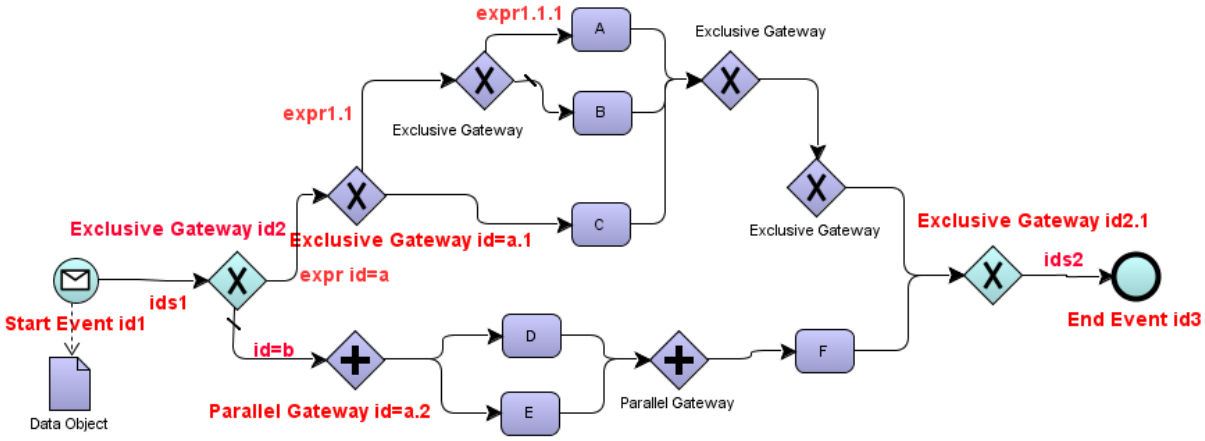
Ovo poglavlje opisuje testiranje algoritama korištenih u modelu za fazni razvoj procesa i pripadnom radnom okviru. Budući da je implementacija faza modela prikazana u radnom okviru, u ovom poglavlju se neće jedan proces provlačiti kroz sve faze modela nego će se demonstracije pojedinih faza modela, odnosno pripadnih pseudokodova, izvršiti na hipotetskim poslovnim procesima bez naglašene semantike ali s dovoljno reprezentativnim izvršavanjima kojima se verificiraju doprinosi. Prvi dio poglavlja demonstrira primjenu algoritma preslikavanja BPMN-BPEL nad hipotetskim procesima i pripadao bi prvoj fazi predloženog faznog modela, dok drugi dio poglavlja demonstrira postupak formalne verifikacije također na zamišljenom hipotetskom procesu i pripadao bi drugoj fazi predloženog faznog modela.

8.1. Testiranje algoritma preslikavanja BPMN-BPEL

Testiranje algoritma preslikavanja BPMN-BPEL provest će se kroz ogledni proces prikazan slikom 59.

Proces 1: slika 59 prikazuje BPMN proces na kojemu će testirati pseudokod algoritma, a koji sadržava sljedeće specifične uzorke ponašanja:

- početni čvor `startEvent`,
- glavna vizualna aktivnost `if`,
- aktivnost `if` ugniježđena unutar glavne aktivnosti `if`,
- aktivnost `if` ugniježđena unutar ugniježđene aktivnosti `if`,
- slijed `sequence` unutar glavne aktivnosti `if` koji sadržava vizualnu aktivnost `flow` i apstraktnu aktivnost `task`.



Slika 59: Primjer BPMN procesa[11]

Preslikavanje BPMN procesa sa slike 59 u pripadajući BPEL proces kreće pozivanjem funkcije `toBPELprocessDraw` koja kreira BPEL `process` sa sadržanim elementom `sequence` i koja potom popunjava generirani element `sequence`.

Prvi koraci funkcije `toBPELprocessDraw` prije popunjavanja glavne BPEL aktivnosti `sequence` jeste generiranje BPEL varijabli i aktivnosti `*eventHandler` na temelju pripadnih BPMN izvršavanja. U primjeru sa slike 59 naveden je samo jedan BPMN element `dataObject`, dok nije naveden niti jedan element `inline event subPorcess`. Element `dataObject` se preslikava u BPEL varijablu[12], dok bi se pripadne aktivnosti `*eventHandler`, da su navedene u primjeru, kreirale na temelju postojećih elemenata `inline event subProcess`[13].

Prvi korak popunjavanja glavne BPEL `sequence` aktivnosti, koji funkcija `toBPELprocessDraw` izvodi na oglednom procesu, jeste pronalazak početnih BPEL aktivnosti, a to su one aktivnosti bez dolaznih slijednih poveznica koje po vrsti odgovaraju dogovrenim početnim BPEL čvorovima (`startEvent`, `receive`, vizualna `pick` aktivnost). Dio koda funkcije `toBPELprocessDraw`, koji to obavlja, je sljedeći:

[11] Aktivnosti na glavnom slijedu su označene plavom bojom, dok su ugniježđene aktivnosti označene ljubičastom bojom radi njihovog razlikovanja.
[12] Na temelju preslikavanja `dataObject` - `variable` u poglavlju 4.1.
[13] Analogija sa slikom 21 iz poglavlja 4.1.

```
forEvery{x € C | E(x).isStartCandidate} :
            nextSequenceNodeId = seqFlowIdFromVisualActivity(x,..);
            toStartBPELactivityDraw(x,..,A);
            //pretvaranje: BPMN kod-početne BPEL aktivnosti
              i dodavanje BPEL sequence aktivnosti (A)
```

U izdvojenom dijelu koda pronalazak početnog čvora na temelju prethodno navedenih svojstava simbolično je obuhvaćen izrazom `isStartCandidate`. U primjeru sa slike 59 to je čvor `startEvent` (`id=1`) koji se funkcijom `toStartBPELactivityDraw`[14] preslikava u BPEL aktivnost `receive` koja se smješta unutar BPEL aktivnosti `sequence`. Pamti se `nextSequenceNodeId=s1` koji izlazi iz čvora `startEvent`.

Slijedni BPMN čvor u koji ulazi slijed izvršavanja `sequenceFlow s1`, a koji će funkcija `toBPELprocessDraw` pronaći, jeste čvor `exclusiveGateway` (`id=2`). Na temelju definiranih uzoraka preslikavanja iz Koraci preslikavanja otkriva se da je riječ o BPEL aktivnosti `if`

te se poziva funkcija `seqFlowIdFromIf` koja skenira sva BPMN izvršavanja na granama predmetne aktivnosti `if` i kao rezultat postavlja `nextSequenceNodeId=s2`. Funkcija `toBPELifDraw` preslikava pronađenu vizualnu aktivnost `if` s njenim izvršavanjima na granama u pripadnu BPEL aktivnost `if` koja se smješta unutar BPEL aktivnosti `sequence`. Dio koda funkcije `toBPELprocessDraw` koji to obavlja je sljedeći:

```
forEvery{x € C | E(x)} :
      if E(x).E(incoming) == nextSequenceNode
      nextSequenceNode = seqFlowIdFromIf(id2,..);
      toBPELifDraw(id2,..,A);
      //pretvaranje: BPMN kod - pripadne BPEL aktivnosti
        i dodavanje  BPEL aktivnosti sequence (A)}
```

Budući da funkcija `toBPELprocessDraw(..)` pronalazi slijedne vizualne aktivnosti i preslikava ih u pripadne vizualne BPEL aktivnosti, dok se ne pronađe čvor koji nema izlaznih vizualnih poveznica (`while nextSequenceNodeId!="none"`), sljedeći čvor primjera sa slike 59 u koji ulazi zadnji pronađeni slijed izvršavanja `sequenceFlow s2` jeste čvor `endEvent` (`id=3`). On nema izlaznih slijednih poveznica i općenito čvor `endEvent`

[14] Temeljem preslikavanja sa slike 16 iz poglavlja 4.1.

predstavlja završetak izvršavanja čime se završava popunjavanje glavne BPEL aktivnosti `sequence` i rad funkcije `toBPELprocessDraw` smatra završenim. Izlaz, koji je dosadašnji kod generirao, je sljedeći:

```
<process>
     <sequence>
          <receive../>
          <if>..generiranje sadržaja na temelju
          izvršavanja funkcije toBPELifDraw..</if>
     </sequence>
</process>
```

Slijedi opis koraka funkcije `toBPELifDraw` koja glavnu vizualnu aktivnost `if` preslikava u BPEL aktivnost `if`. Funkciji se prosljeđuju parametri: početni `exclusiveGateway` (`id=2`) i krajnji `exclusiveGateway` (`id=2.1`) kao i BPEL aktivnost u koju se blokovski dodaje izlazna generirana BPEL aktivnost `if`, što je u ovom slučaju BPEL aktivnost `sequence`.

Prvi korak funkcije `toBPELifDraw` vrši pronalaske izlaznih slijednih poveznica iz početnog `exclusiveGateway` elementa i smješta ih u skup D `sequenceFlows`:

```
D.create("sequenceFlows");
forEvery {a € A | E(a)} :
     M.search(m € M | E(m).R(id) == X.E(outgoing), D);
```

Sadržaj skupa D procesa sa slike 59 će biti elementi `sequenceFlow` (`id=a,b`). Slijedi prolaz kroz svaki element `sequenceFlow` skupa D (u ovom slučaju samo uvjetni `sequenceFlow` elementi koji se preslikavaju u BPEL uvjetne grane, dok je analogija i s jednim neuvjetnim elementom `sequenceFlow` koji se preslikava u granu `else` BPEL aktivnosti `if`) i otkrivanje prvog BPMN čvora na toj grani koji se smještaju u skup E `firstNodeOnFirstBranch` što obavlja sljedeći dio koda funkcije `toBPELifDraw`:

```
forEvery {d € D| E(d)} :
     if d.E(conditionExpression).exists ->****
          if counter==0//prva uvjetna aktivnost
             C.add(E(condition, d.E(conditionExpression));
          elseif counter>0//naredne uvjetne aktivnosti na granama
```

```
            C1.create(„elseif“);
            C1.add(E(condition, d.E(conditionExpression));
        E.create(“firstNodeOnFirstBranch”);
        M.search(m € M | E(m).E(incoming) == d.R(id), E);
        //M - prvi čvorovi na granama vizualne aktivnosti if
```

Sadržaj skupa E procesa sa slike 59 za element `sequenceFlow` (`id=a`) će biti BPMN čvor `exclusiveGateway` (`id=a.1`). Za čvor E vrši se otkrivanje da li je početak jedne vizualne aktivnosti ili je dio slijeda na grani što obavlja sljedeći dio koda funkcije `toBPELifDraw` (G=zadnji `exclusiveGateway` (`id=2.1`)). U pseudokodu je simbolično uvedena funkcija `nextElementAfterVisualActivity`, a u realnoj implementaciji koda se mora proći kroz sve elemente vizualne aktivnosti koja počinje BPMN čvorom E i otkriti koji je njezin slijedni BPMN čvor.

```
if(nextElementAfterVisualActivity(E) == G) ->***
      //jedna aktivnost na grani –
        generiranje sadržane jedne BPEL aktivnosti
else -> //na grani je niz aktivnosti koji se smješta u sequence
```

Za slučaj elementa `sequenceFlow` (`id=a`) i njezin prvi čvor `exclusiveGateway` (`id=a.1`), slijedni čvor jeste zadnji čvor `exclusiveGateway` (`id=2.1`) čime se zaključuje da je na ovoj grani sadržana jedna vizualna aktivnost, ugniježđena BPEL aktivnost `if`. Da bi se ugniježđena vizualna aktivnost `if` preslikala u pripadnu BPEL aktivnost `if`, izvršava se sljedeći dio koda funkcije `toBPELifDraw`:

```
if E == "exclusiveGateway" ->
      //ugniježđena aktivnosti if na grani
      //M.search(m € M | E(m).R(id) == seqFlowIdFromIf(E), TEMP);
      G1=ifBPELactivityFollowing(TEMP);
      toBPELifDraw(E,G1,C/C1);
      //C-prva uvjeta grana; C1-slijedne uvjetne grane
```

Funkcija `toBPELifDraw` poziva dakle funkcije `seqFlowIdFromIf` i `toBPELifDraw` radi dohvaćanja izlaznog elementa `sequenceFlow` i generiranja ugniježđene BPEL aktivnosti `if`. Budući da ugniježđena BPEL aktivnosti `if` sadržava unutar sebe novu

igniježđenu aktivnosti `if`, pozvana funkcija `toBPELifDraw` rekurzivno će odraditi identične korake.

Sadržaj skupa E procesa sa slike 59 za element `sequenceFlow` (`id= b`) će biti BPMN čvor `parallelGateway` (`id=a.2`). Za čvor E vrši se otkrivanje da li je početak jedne vizualne aktivnosti ili je dio slijeda na grani i budući da `nextElementAfterVisualActivity(parallelGateway(id=b))` jeste BPMN čvor `taskF`, zaključuje se da se radi o slijedu u kojem su smještene BPEL vizualne aktivnosti `pick` (koja počinje čvorom `parallelGateway (id=a.2)`[15]) i simbolična BPEL aktivnost predstavljena čvorom `taskF`.

Izlaz koji se dobija u konačnici je sljedeći BPEL proces:

```
<process><sequence><receive .../>
<if><condition>expr1</condition>[16]
        <if><condition>expr1.1</condition>[17]
            <if><condition>expr1.1.1</condition>A[18]
              <else>B</else></if>
        <else>C</else></if>
     <elseif><condition>expr2</condition>
        <sequence><flow>D E</flow>F</sequence>
     </elseif>
</if></sequence></process>
```

8.2. Testiranje algoritma formalne verifikacije

Testiranje *Promela* modela kreiranih temeljem pravila iz Pretvaranje u formalni model s ciljem otkrivanja da li se njima može otkriti paralelna manipulacija nad varijablama izvršit će se na hipotetskom BPEL poslovnom procesu prezentiranom u Pretvaranje u formalni model (Tablica 12: BPEL proces s paralelnim izvršavanjima). U ovom poglavlju ponovno je prikazan XML kod pripadnog BPEL procesa i opisan je njegov simbolički model koji sadržava konstrukcije bitne za formalnu provjeru, a to su sve konstrukcije koje se paralelno izvršavaju ili konstrukcije koje dotiču varijable na neki način, bilo da se radi o čitanju ili o ažuriranju varijabli.

[15] Temeljem preslikavanja sa slike iz poglavlja.
[16] Glavna aktivnost `if`.
[17] Prva ugniježđena aktivnost `if`.
[18] Ugniježđena aktivnost `if` unutar prve ugniježđene aktivnosti `if`.

Slijedi BPEL XML kod procesa s označenim elementima bitnim za formalnu provjeru:

```
<process>
<variables>
<variable name="A" ../><variable name="B" ../>
</variables>
<sequence>
   <invoke1 portType=".."    operation="Op1"
   inputVariable="A" outputVariable="B"/>
    <flow2>
     <if3>
         <condition3.1>expr reading A</condition>
                  <assign4>writing B</assign4>
         <else>Activity2</else>
     </if3>
     <assign5>writing A</assign5>
    </flow2>
</sequence>
      <eventHandlers>
            <onEvent6>
            <forEach parallel="yes">
                  <startCounterValue6.1>expr reading A
                  </startCounterValue6.1>
                  <finalCounterValue>10</finalCounterValue>
                  <scope7>
                  <flow8>
                 <assign9>writing A</assign9>
                  <assign10>writing B</assign10>
                  </flow8>
                 </scope7>
            </forEach>
            </onEvent6>
      </eventHandlers>
</process>
```

Slijedi simbolički opis BPEL procesa:

- **process:** variables(A, B), eventHandlers(onEvent6),
- **activities**(invoke1, flow2)
- **flow:** flow2(name), activities(if3, assign5)

- **if**: if3(name), expressions(expression 3.1)
- **expression**: expression3.1(name), activities(assign4)
- **assign**: assign5(name), A(write,toVariableName)
- **assign**: assign4(name), B(write,toVariableName)
- **eventHandlers**: onEvent6(name), activities(forEach)
- **forEach**: forEach(name), boolean(parallelism,true), expression(startCounterValue6.1), activities(scope7)
- **scope**: scope7(name), activities(flow8)
- **flow**: flow8(name), activities(assign9, assign10)
- **assign**: assign9(name), A(write,toVariableName)
- **assign**: assign10(name), B(write,toVariableName).

Slijedi prikaz *Promela* formalnog modela BPEL procesa definiranog na temelju prethodnog simboličkog modela.

Za svaku varijablu temeljem definicija iz poglavlja 6.2.3 (u ovom slučaju globalne varijable definirane na razini procesa) definiraju se sljedeće *Promela* konstrukcije:

```
int typeA=1;/*varijabla A*/
bool valueA=0;
int accessWriteA=0;
int accessReadA=0;

int typeB=2; /*varijabla B*/
bool valueB=0;
int accessWriteB=0;
int accessReadB=0;
```

Glavni BPEL proces temeljem definicija iz Pretvaranje u formalni model predstavlja se glavnim *Promela* procesom *init* unutar kojega se nalaze pripadna *Promela* izvršavanja sadržanih aktivnosti `invoke1` nakon čijeg završetka se pokreću procesi za sadržane paralelne aktivnosti `if3` i `assign5` unutar aktivnosti `flow2`:

```
init { /*BPEL PROCESS*/
atomic { /*početak aktivnosti invoke1*/
accessReadA++; accessWriteB++; /*rezervacija varijabli*/}
atomic { /*kraj aktivnosti invoke1*/
```

```
accessReadA--; accessWriteB--;
/*otpuštanje rezervacija varijabli*/}
run flow2if3(); run flow2assign5();
```

Aktivnost `if3` unutar aktivnosti `flow2`, a koja sadržava izraz `expression3.1` temeljem definicija za predstavljanje izraza iz Pretvaranje u formalni model, predstavljena je donjim *Promela* procesom:

```
proctype flow2if3(){
if
:: (true)->
accessReadA++; accessReadA--;
/*pocetak i kraj izraza expression3.1*/
/*pretpostavka da je uvjet expression3.1 true->assign5 se izvršava*/
accessWriteB++; accessWriteB--;
/*početak i kraj aktivnosti assign5*/
fi}
```

Aktivnost `assign5` unutar aktivnosti `flow2` temeljem definicija za predstavljanje izraza iz Pretvaranje u formalni model predstavljena je donjim *Promela* procesom:

```
proctype flow2assign5(){accessWriteA++; accessWriteA--;}
/*rezervacija i otpuštanje varijabli*/
```

Aktivnost `onEvent6` prikačena za glavnu BPEL aktivnost `process` izvršava se paralelno s glavnim procesom *init* kojim je predstavljena BPEL aktivnost `process` stoga njen pripadajući *Promela* proces ima status `active` te su sadržane aktivnosti unutar BPEL aktivnosti `onEvent` temeljem definicija iz Pretvaranje u formalni model predstavljene na donji način:

```
active proctype onEvent6{
/*pretpostavka je da se desio događaj onEvent6 pa je proces aktivan*/
/*izvršavanje izraza startCounterValue 6.1
unutar aktivnosti forEach*/
accessReadA++; accessReadA--;
run flow8assign9(); run flow8assign10();}
```

Aktivnosti `assign9` i `assign10` unutar aktivnosti `flow8` (koja je pak sadržana unutar aktivnosti `scope7` događaja `onEvent6`) temeljem definicija za predstavljanje izraza iz Pretvaranje u formalni model predstavljene su donjim *Promela* procesima:

```
proctype flow7assign8(){accessWriteA++; accessWriteA--;}
proctype flow7assign9(){accessWriteB++; accessWriteB--;}
```

Provjeravano svojstvo izraženo LTL sintaksom, kojim se provjerava nepostojanje pojave paralelnog ažuriranja i čitanja varijable, glasi:

```
ltl {[] (accessWriteA>0 -> accessReadA==0)}
```

Gore navedeni *Promela* kod predstavlja ulaz u program *spin* kojim se vrši formalna provjera s ciljem otkrivanja navedenog neželjenog ponašanja. Slijede koraci formalne provjere i rezultati njihovog izvršavanja:

- rezultat prve iteracije *spin* provjere generiralo je putanju izvršavanja na kojoj je otkrivena pojava paralelnog ažuriranja i čitanja varijable A. Slijedom izvršavanja generirane putanje koja prikazuje sinkronizirana izvršavanja *Promela* ponašanja BPEL procesa i negiranog željenog ponašanja (*promela never claim* [44]), dobiveni su dole navedeni rezultati. Tijekom njihovog sinkroniziranog izvršavanja, prvo se izvršava korak *Promela* sintakse neželjenog ponašanja, nakon čega slijedi korak *Promela* procesa pri čemu se u svakom koraku prikazuje upravo izvršena linija koda, stanje procesa u kojem se proces našao, i izvršena akcija koja se provela[19].

```
1:    proc  - (ltl_0:1) _spin_nvr.tmp:4 (state 4)     [(1)]
Never claim moves to line 4   [(1)]
2:    proc  0 (:init::1) BPELProces.pml:13 (state 1)
[accessReadA = (accessReadA+1)]
2:    proc  0 (:init::1) BPELProces.pml:13 (state 2)
[accessWriteB = (accessWriteB+1)]

..međuizvršavanja skraćena radi uštede prostora..

11:   proc  - (ltl_0:1) _spin_nvr.tmp:4 (state 4)     [(1)]
```

[19] `BPELProces.pml:13 (state 1) [accessReadA = (accessReadA+1)]` – izvršio se red 13 koda, proces je završio u stanju `state 1`, izvršila se akcija `accessReadA = (accessReadA+1)`.

```
12:   proc  2 (flow2assign4:1) BPELProces.pml:29 (state 1)
[accessWriteA = (accessWriteA+1)]
13:   proc  - (ltl_0:1) _spin_nvr.tmp:4 (state 4)     [(1)]
14:   proc  1 (flow2if3:1) BPELProces.pml:23 (state 2)
[accessReadA = (accessReadA+1)]
MSC: ~G line 3
15:   proc  - (ltl_0:1) _spin_nvr.tmp:3 (state 1)
[(!((!((accessWriteA>0))||(accessReadA==0))))]

_______________________________
[variable values, step  15]
accessReadA  =  1
accessReadB  =  0
accessWriteA  =  1
accessWriteB  =  0
```

- Prva iteracija formalne provjere pokazala je paralelno ažuriranje i čitanje varijable A o kojima svjedoče zadnja dva koraka:

```
12:   proc  2 (flow2assign5:1) BPELProces.pml:29 (state 1)
[accessWriteA = (accessWriteA+1)]
14:   proc  1 (flow2if3:1) BPELProces.pml:23 (state 2)
[accessReadA = (accessReadA+1)]
```

- Paralelno ažuriranje i čitanje varijable se vrši od strane aktivnosti `assign5` unutar `flow2` i `if3` unutar `flow2` stoga slijedi eliminacija aktivnosti `if3` unutar `flow2`.

- Druga iteracija formalne provjere pokazala je paralelno ažuriranje i čitanje varijable A o kojima svjedoče zadnja dva koraka:

```
16: proc  2 (flow8assign9:1) BPELProces.pml:43 (state 1)
[accessWriteA = (accessWriteA+1)]
18: proc  0 (:init::1) BPELProces.pml:15 (state 1)
[accessReadA = (accessReadA+1)]
```

- Paralelno ažuriranje i čitanje varijable se vrši od strane aktivnosti `assign9` unutar `flow8` i *init* proces – BPEL `process` stoga slijedi eliminacija aktivnosti `assign9` unutar `flow8`.

- Treća iteracija formalne provjere pokazala je paralelno ažuriranje i čitanje varijable A o kojima svjedoče zadnja dva koraka:

```
2: proc  1 (onEvent6:1) BPELProces.pml:36 (state 1)
[accessReadA = (accessReadA+1)]
10: proc  2 (flow2assign5:1) BPELProces.pml:29 (state 1)
[accessWriteA = (accessWriteA+1)]

(spin: text of failed assertion:
assert(!(!((!((accessWriteA>0))||(accessReadA==0)))))
#processes: 3
11:   proc  2 (flow2assign5:1) BPELProces.pml:30 (state 2)
11:   proc  1 (onEven6:1) BPELProces.pml:37 (state 2))
```

- Paralelno ažuriranje i čitanje varijable se vrši od strane aktivnosti `assign5` unutar `flow2` i `onEvent6` stoga slijedi eliminacija aktivnosti `assign5` unutar `flow2`.

- Četvrta iteracija formalne provjere napokon nije pokazala paralelno ažuriranje i čitanje varijable A.

9. Zaključak

S razvojem usluga, koje nerijetko predstavljaju izložene funkcionalnosti dijela poslovnih procesa ili pak samo obavljaju pojedinačne funkcionalnosti, javlja se potreba za njihovim ponovnim iskorištavanjem na način da se one integriraju u složenije usluge. BPEL je standard koji je odgovorio na tu potrebu i ponuđenim XML konstrukcijama omogućio integraciju usluga izloženih standardiziranim WSDL sučeljima u složene poslovne procese. Fokus doktorskog rada bio je na identifikaciji poteškoća koje se pojavljuju tijekom izgradnje BPEL poslovnog procesa. Identificiran je problem otežane integracije usluga u poslovni proces kada ne postoji vizualni model procesa. Osim toga, BPEL standard omogućuje konstrukcije paralelnog izvršavanja pa je moguće pojavljivanje potencijalno neodredivih izvršavanja procesa zbog nepredvidivosti brzine izvršavanja paralelnih komponenti. Kao rezultat uočenih problema proizašla je ideja o razvoju algoritama za vizualno modeliranje i formalnu verifikaciju BPEL procesa i njihovoj integraciji u model za fazni razvoj BPEL procesa.

Definiran je model za fazni razvoj BPEL procesa zasnovan na pretpostavci postojanja gotovih usluga koje će se integrirati u poslovni proces. Model integrira faze vizualnog modeliranja, formalne verifikacije i statičke provjere procesa u jednu homogenu cjelinu. Vizualno modeliranje procesa provodi se prema standardu BPMN. Kao premosnica između različitih BPEL i BPML načina strukturiranja procesa definirane su funkcije preslikavanja njihovog XML koda.

Predloženim algoritmima dvosmjernog preslikavanja između BPEL koda i odabranih formalnih BPMN specifikacija za vizualno modeliranje uz trivijalne riješeni su problemi preslikavanja ugniježđenih aktivnosti i otkrivanja kraja slijeda od nekoliko aktivnosti na granama strukturnih aktivnosti. Zbog složenosti problema, vizualno preslikavanje za sinkronizirana izvršavanja unutar BPEL `flow` aktivnosti nije automatizirano, već je napravljen model oslikavanja za sve moguće tipove sinkroniziranog izvršavanja koji se u ovoj fazi prevodi ručno u BPEL.

Za implementaciju algoritma statičke provjere pravila nad BPEL aktivnostima definiranih BPEL specifikacijom, nakon analize ponuđenih pravila i uzevši u obzir blokovsku strukturu BPEL standarda s ugnježđivanjima do proizvoljne dubine, kao glavni zadatak prepoznata je implementacija navigacije kroz svaki element BPEL procesa kako bi se bilo koje pravilo

automatizirano provjerilo nad svim dijelovima procesa naspram ručnog provođenja provjere u čemu leži korisnost ovoga algoritma.

Za predstavljanje formalnog modela procesa odabrani su konačni automati predstavljeni *Promela* sintaksom. Formalna provjera zbog paralelnog asinkronog ponašanja BPEL procesa usredotočena je na provjeru pojave paralelnog pristupa varijablama i na pokušaje čitanja neinicijaliziranih varijabli. Identificirane su sve komponente procesa koje se mogu međusobno paralelno izvršavati i koje na neki način vrše manipulaciju nad varijablama bilo da je riječ o čitanju ili pisanju. Predložen je model označavanja prethodno navedenih komponenti bitnih za formalnu provjeru unutar BPEL procesa te na temelju označenog BPEL procesa kreiranje simboličkog modela procesa popisivanjem svih označenih komponenti. Simbolički model definiran je kao prijelaz prema formalnom modelu na način da olakša njegovo kreiranje. Model označavanja BPEL procesa je uveden s ciljem lociranja pogreške unutar BPEL procesa nakon provedene formalne provjere. Paralelni pristup varijablama otkriva se definiranjem znački čitanja i pisanja nad varijablama prilikom rezerviranja ili otpuštanja varijabli.

Radni okvir modela definiran je s ciljem dokazivanja mogućnosti implementacije predloženog faznog modela. Kroz odabrane alate i tehnologije za vizualno modeliranje, izvršavanje BPEL procesa i formalnu verifikaciju predstavljene u doktorskom radu prikazan je način praktičnog izvršavanja koraka definiranih faznim modelom procesa te način kako se oni mogu povezati u jednu cjelinu. Na taj način je praktično verificirana izvodivost predloženog faznog modela. Provedeno je i testiranje algoritama predloženog faznog modela s ciljem verifikacije izvodivosti na imaginarnim poslovnim procesima s reprezentativnim izvršavanjima.

Ovim doktorskim radom predstavljen je zaokruženi model za razvoj BPEL procesa. Predloženi fazni model je identificirao ključne korake razvoja procesa i može biti polazište za njegovu daljnju razradu na način da se uključi i faza prikupljanja korisničkih zahtjeva, identifikacije i izgradnje usluga koje učestvuju u integraciji ili faza ponovnog iskorištavanja poslovnog procesa kao nove kompozitne usluge za neke buduće integracije.

Predloženi algoritam preslikavanja BPEL-BPMN-BPEL predstavlja polazište za daljnje optimizacije na aktivnostima koje su se mogle predstaviti jednostavnije, kao što je to primjerice sinkronizirano izvršavanje unutar BPEL aktivnosti `flow`, da se napravi više mogućnosti predstavljanja jedne BPEL aktivnosti s ciljem omogućavanja veće fleksibilnosti izbora vizualnih BPMN konstrukcija tijekom procesa modeliranja ili da se na temelju predloženih

uparenih uzoraka ponašanja analizira mogućnost njihovog prepoznavanja i kreiranja korištenjem XSLT transformacija kako bi cijeli proces vizualnog modeliranja ostao u domeni XML-a, neovisan o bilo kojoj tehnologiji. Formalni model predložen u doktorskom radu može poslužiti kao osnova proširenja s ciljem provjere drugih svojstava ili da se u obzir uzmu i ponašanja procesa u slučajevima pogreške što trenutno nije slučaj. U formalnoj provjeri predloženoj u doktoratu u obzir je uzeto regularno ponašanje procesa.

Popis literature

[1] "Service-Oriented Architecture – What Is SOA? ", dostupno na: http://www.opengroup.org/soa/source-book/soa/p1.htm (07. rujna 2019.)

[2] Mohr, F., "Software Reuse for Dynamic Systems in the Cloud and Beyond", Miami, FL, 2015., str. 298-313.

[3] "WS-BPEL 2.0", dostupno na https://slideplayer.com/slide/2483942/ (01. rujna 2019.)

[4] OASIS group, "Reference Model for Service Oriented Architecture 1.0", August 2006

[5] OASIS group, "Reference Architecture for Service Oriented Architecture Version 0.3", March 2008

[6] "About Us", dostupno na: https://www.oasis-open.org/org (10. kolovoza 2019.)

[7] W3C, "XML Schema", October 2004., dostupno na: https://www.w3.org/TR/xmlschema-0/

[8] "ActiveVOS Overview", dostupno na: http://www.activevos.com/products/activevos/overview (05. kolovoza 2019.)

[9] "What is jBPM?", dostupno na: https://www.jbpm.org/ (05. kolovoza 2019.)

[10] "Oracle BPEL Process Manager", dostupno na: https://www.oracle.com/technetwork/middleware/bpel/overview/index.html (25. srpnja 2019.)

[11] OMG Object Management Group, "Business Process Model And Notation", January 2011.

[12] Holzmann, Gearld, J, "Software Model Checking with SPIN", dostupno na: http://spinroot.com/gerard/pdf/Advances2005.pdf html (25. srpnja 2019.)

[13] Papazoglou, M., van den Heuvel, W., "Service oriented architectures: approaches, technologies and research issues", The VLDB Journal, Vol. 16, No. 3, July 2007, str. 389–415.

[14] W3C, "Web Services Description Language (WSDL) 1.1", March 2001.

[15] SOAP, "https://www.w3.org/TR/soap/", April 2007.

[16] Menge, F., "Enterprise Service Bus", dostupno na: https://pdfs.semanticscholar.org/7539/3c46ab62d8c25d13f79d68ef42e232474b53.pdf (27. lipnja 2019.)

[17] W3C "Web Services Addressing (WS-Addressing)", August 2004.

[18] Frey, F., Hentrich, C., Zdun, U., "Pattern-based Process for a Legacy to SOA Modernization Roadmap", 18th European Conference on Pattern Languages of Programs (EuroPLoP), Irsee, Germany, 2014.

[19] Azevedo, L. G., et al., "A Method for Service Identification from Business Process Models in a SOA Approach", International Conference on Exploring Modeling Methods for Systems Analysis and Design, Amsterdam, Netherlands, 2009., str. 99-112.

[20] Fareghzadeh, N., "Web Service Security Method To SOA Development", International Journal of Computer and Information Engineering, Vol. 3, No. 1, January, 2009, str. 815-819.

[21] Zimmermann, O., "Analysis and Design Techniques for Service-Oriented Development and Integration", GI Jahrestagung, January 2005, str. 606-611.

[22] "Open Source Business Automation", dostupno na: https://www.activiti.org/ (25. lipnja 2019.)

[23] "Yaoqiang BPMN Editor", dostupno na: http://bpmn.sourceforge.net (10. srpnja 2019.)

[24] "Microsoft Visio" dostupno na: https://products.office.com/en/visio/flowchart-software (25. srpnja 2019.)

[25] Lapadula, A., Pugliese, R., Tiezzi, F., "Using formal methods to develop WS-BPEL applications", Science of Computer Programming, Vol. 77, No. 3, March 2012, str. 189-213.

[26] Vemulapalli, A., "From Business Process Models to Web Services Orchestration: The Case of UML 2.0 Activity Diagram to BPEL", International Conference on Service-Oriented Computing, Sydney, Australia, 2008., str. 505-510.

[27] Zhang, M., Duan, Z., "Transforming Functional Requirements from UML into BPEL to Efficiently Develop SOA-Based Systems", OTM Confederated International Conferences "On the Move to Meaningful Internet Systems", Portugal, 2009., str. 337-349.

[28] Markoska, E., Ristov, S., Gusev, M., "Translating BPMN to WS-BPEL", Conference of Informatics and Information Technology CiiT2015, Bitola, Macedonia, 2015., str. 59-66.

[29] Jurišić, M., "Transition between process models (BPMN) and service models (WS-BPEL and other standards): A systematic review", Journal of Information and Organizational Sciences, Vol. 35, No. 2, 2011, str. 163-171.

[30] Wohed, P., et al., "On the Suitability of BPMN for Business Process Modelling", Business Process Management: 4th International Conference, Vienna, Austria, 2006.

[31] Ouyang, C., Dumas, M., ter Hofstede, A., van der Aalst, W. M. P., "From BPMN Process Models to BPEL Web Services", Proceedings of the IEEE International Conference on Computer Vision, New York, USA, 2006., str. 285-292.

[32] "Using BPMN to Model a BPEL Process", dostupno na: https://www.omg.org/bpmn/Documents/Mapping_BPMN_to_BPEL_Example.pdf (10. srpnja 2019.)

[33] Ouyang, C., et al., "Formal semantics and analysis of control flow in WS-BPEL", Science of Computer Programming, Vol. 67, No. 2-3, July 2007, str. 162-198.

[34] Kongburan, W., Pradubsuwun, D., "Formal Verification of WS-BPEL Using Timed Trace Theory", Advanced Materials Research, Vols. 931-932, 2014, str. 1452-1456.

[35] Wang, Y., "A Survey on Formal Methods for Web Service Composition", dostupno na: https://pdfs.semanticscholar.org/2f5b/b0856d351cd40036e486c29abc1c814c3f90.pdf (28. lipnja 2019.)

[36] Ait-Sadoune I., Ait-Ameur Y. (2013) Stepwise Development of Formal Models for Web Services Compositions: Modelling and Property Verification. In: Hameurlain A., Küng J., Wagner R., Liddle S.W., Schewe KD., Zhou X. (eds) Transactions on Large-Scale Data- and Knowledge-Centered Systems X. Lecture Notes in Computer Science, vol 8220. Springer, Berlin, Heidelberg

[37] Dragoni, N., Mazzara, N., "A Formal Semantics for the WS-BPEL Recovery Framework", International Workshop on Web Services and Formal Methods, Italy, 2009., str. 92-109.

[38] Darbari, A., "Doc Formal: The evolution of formal verification – Part One", dostupno na: http://www.techdesignforums.com/practice/technique/the-ongoing-evolution-of-formal-verification/ (10. rujna 2019.)

[39] Darbari, A., "http://www.techdesignforums.com/practice/technique/doc-formal-the-evolution-of-formal-verification-part-two/", dostupno na: http://www.techdesignforums.com/practice/technique/the-ongoing-evolution-of-formal-verification/ (15. rujna 2019.)

[40] Darbari, A., " Doc Formal: The crisis of confidence facing verification", dostupno na: http://www.techdesignforums.com/practice/technique/doc-formal-the-crisis-of-confidence-facing-verification/ (18. srpnja 2019.)

[41] Ben-Ari, M., " Principles of the Spin Model Checker", Springer-Verlag, London, 1985.

[42] Mukund, M., "Finite-state Automata on Infinite Inputs", dostupno na: https://www.cmi.ac.in/~madhavan/papers/pdf/tcs-96-2.pdf (28. lipnja 2019.)

[43] Holzmann, G., Peled, D., Yannakakis, M., "On Nested Depth First Search", dostupno na: http://spinroot.com/gerard/pdf/inprint/spin96.pdf (28. kolovoza 2019.)

[44] "Declaration of a temporal claim", dostupno na: http://spinroot.com/spin/Man/never.html (28. srpnja 2019.)

[45] "Linear time temporal logic formulae for specifying correctness requirements", dostupno na: http://spinroot.com/spin/Man/ltl.html (28. lipnja 2019.)

[46] "Concise Promela Reference", dostupno na: http://spinroot.com/spin/Man/ltl.html (28. lipnja 2019.)

[47] Holzmann, G., "The Model Checker SPIN", dostupno na: http://spinroot.com/spin/Doc/ieee97.pdf (15. srpnja 2019.)

[48] "Graphviz - Graph Visualization Software", dostupno na: https://www.graphviz.org/ (15. srpnja 2019.)

[49] "Basic Spin Manual", dostupno na: http://spinroot.com/spin/Man/Manual.html (28. lipnja 2019.)

[50] "Visual Studio Express", dostupno na: https://visualstudio.microsoft.com/vs/express/ (28. lipnja 2019.)

[51] "C# Guide", dostupno na: https://docs.microsoft.com/en-us/dotnet/csharp/ (28. srpnja 2019.)

[52] "LINQ to XML Overview (C#)", dostupno na: https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/linq/linq-to-xml-overview (28. lipnja 2019.)

[53] "XML Document Object Model (DOM)", dostupno na: https://docs.microsoft.com/en-us/dotnet/standard/data/xml/xml-document-object-model-dom (28. lipnja 2019.)

[54] "LINQ to XML vs. DOM (C#)", dostupno na: https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/linq/linq-to-xml-vs-dom (28. srpnja 2019.)

[55] "LINQ in Visual Basic", dostupno na: https://docs.microsoft.com/en-us/dotnet/visual-basic/programming-guide/language-features/linq/ (28. srpnja 2019.)

[56] W3C, "XSL Transformations (XSLT) Version 3.0", June 2017.

[57] "Oracle JDeveloper", dostupno na: https://www.oracle.com/technetwork/developer-tools/jdev/overview/index.html (28. srpnja 2019.)

[58] "Using iSPIN", dostupno na: http://spinroot.com/spin/Man/3_SpinGUI.html (10. srpnja 2019.)

[59] "Tcl Developer Xchange", dostupno na: https://www.tcl.tk/ (28. lipnja 2019.)

[60] Holzmann, Gerard: “The SPIN Model Checker“, Addison-Wesley, Canada, 2003

[61] "Bell Labs", dostupno na: https://ethw.org/Bell_Labs (28. kolovoza 2019.)

Popis kratica

WS-BPEL	Web Service Business Process Execution Language
SOA	Service Oriented Architectures
WSDL	Web Service Description Language
SOAP	Simple Object Access Protocol
ESB	Enterprise Service Bus
Promela	Promela Meta Level
LTL	Linear Temporal Logic
DOM	Document Object Model
WS-ADDRESSING	Web Service Addressing Standard

Životopis

Jelena Matković rođena je 02. travnja 1983. godine, u Mostaru, Bosna i Hercegovina. Nakon završene Gimnazije fra Dominika Mandića, opći smjer, u Mostaru 2002. godine se upisuje na Fakultet strojarstva i računarstva, studij računarstva, Sveučilišta u Mostaru na kojem je diplomirala u siječnju 2007. godine s izvrsnim uspjehom. Tri akademske godine bila je dobitnica Rektorove nagrade i jednu godinu dobitnica Dekanove nagrade.

U svibnju 2007. godine zapošljava se u JP Elektroprivreda HZ-HB d.d. Mostar u Službi za sistemsku administraciju sustava, u kojoj radi i danas. Aktivno je zaposlena na poslovima administracije sistemske infrastrukture, na poslovima održavanja poslužitelja elektroničke pošte, izrade i održavanja sistemskih kopija poslužitelja, održavanja *Cluster* sustava, domenskog poslužitelja i na svim drugim sistemskim poslovima.

Njen osobni interes leži u matematičkim vještinama te dugi niz godina ima otvorenu privatnu praksu za instrukcije iz svih predmeta Matematičke analize, Diskretne matematike i Linearne algebre za tehničke fakultete.

U rujnu 2007. godine upisuje Poslijediplomski doktorski studij na Fakultetu elektrotehnike i računarstva, Sveučilišta u Zagrebu na kojem je do 2010. godine položila 13 predmeta s ocjenama vrlo dobar ili izvrstan. 21. siječnja 2014. na sjednici Senata Sveučilišta u Zagrebu joj je odobrena tema doktorskog rada.

Popis objavljenih radova

[1] Matković, J., Fertalj, K., Kalpić, D., "Referential SOA Architectures and Models", TTEM Technics Technologies Education Management, Vol. 4, No. 2, 2009, str. 128-137.

[2] Matković, J., Fertalj, K., "Comparative Analysis of Web Services and Web Service Development Technologies", 33. međunarodni skup MIPRO, Opatija, 2010., str. 274-279.

[3] Matković, J., Fertalj, K., "Inside of Composite Web Service Development", 34. međunarodni skup MIPRO, Opatija, 2011., str. 200-207.

[4] Matković, J., Fertalj, K., "Models for the development of Web service orchestrations", 35. međunarodni skup MIPRO, Opatija, 2012.

[5] Matković, J., Fertalj, K., "A development methodology for Web service based systems and vendor specific development tools“, TTEM Technics Technologies Education Management, Vol. 8, No. 2, 2013, str. 865-874.

[6] Matković, J., Fertalj, K., "Handling Web Service Interfaces", 36. međunarodni skup MIPRO, Opatija, 2013.

[7] Matković, J., Fertalj, K., "Formal checking of WS-BPEL orchestrations", 37. međunarodni skup MIPRO, Opatija, 2014.

[8] Matković, J., Fertalj, K., "Visual modelling of WS-BPEL processes using BPMN standard", 42. međunarodni skup MIPRO, Opatija, 2019.

Biography

Jelena Matković was born in April, the 2nd, 1983. in Mostar, Bosnia and Herzegovina. Upon graduating from Gymnasium fra Dominik Mandić (common direction) in Mostar, in 2002., she has enrolled Faculty of Mechanical Science and Computing, Department of Computing, University of Mostar, where she graduated in January 2007. with outstanding scholarship. She was awarded Rector's award for three times and Dean's award for one time.

She has been employed at JP Elektroprivreda HZ-HB d.d. Mostar at the System administration department since May 2007. She works on jobs of system administration: e-mail server administration, system backup administration, cluster system administration, domain server administration and on other system administration jobs.

Her personal interest lies in mathematical skills and she has been tutoring students in all mathematics courses: Mathematical Analysis, Linear Algebra and Discrete Mathematics for many years.

In September 2007. she has enrolled Postgraduate study at the Faculty of Electrical Engineering and Computing, University of Zagreb, where she took all the exams with great success by 2010.

In Janury, the 21st, 2014. she has been approved a doctoral thesis by the Senate of the. University of Zagreb.