VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA PODNIKATELSKÁ

FACULTY OF BUSINESS AND MANAGEMENT

ÚSTAV INFORMATIKY

INSTITUTE OF INFORMATICS

INTERAKTIVNÍ NÁSTROJ PRO VYTVÁŘENÍ SPUSTITELNÝCH BPMN WORKFLOW V PLATFORMĚ GPC

INTERACTIVE DESIGNER OF EXECUTABLE BPMN WORKFLOWS FOR GPC PLATFORM

DIPLOMOVÁ PRÁCE

MASTER'S THESIS

AUTOR PRÁCE

AUTHOR

Bc. Ondřej Havelka

VEDOUCÍ PRÁCE

SUPERVISOR

Ing. Viktor Ondrák, Ph.D.

BRNO 2022

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
FAKULTA PODNIKATELSKÁ

Zadání diplomové práce

Ústav:	Ústav informatiky
Student:	**Bc. Ondřej Havelka**
Vedoucí práce:	**Ing. Viktor Ondrák, Ph.D.**
Akademický rok:	2021/22
Studijní program:	Informační management

Garant studijního programu Vám v souladu se zákonem č. 111/1998 Sb., o vysokých školách ve znění pozdějších předpisů a se Studijním a zkušebním řádem VUT v Brně zadává diplomovou práci s názvem:

Interaktivní nástroj pro vytváření spustitelných BPMN workflow v platformě GPC

Charakteristika problematiky úkolu:

Úvod
Cíle práce, metody a postupy zpracování
Teoretická východiska práce
Analýza současného stavu
Vlastní návrhy řešení
Závěr
Seznam použité literatury
Přílohy

Cíle, kterých má být dosaženo:

Návrhnout interaktivní nástroj pro vytváření spustitelných BPMN workflow v platformě GPC.

Základní literární prameny:

ENSTROM, David. A Simplified Approach to It Architecture with Bpmn: A Coherent Methodology for Modeling Every Level of the Enterprise. iUniverse, 2016. ISBN 978-1491784976.

FLANAGAN, David. JavaScript: The Definitive Guide, 6th Edition. Sebastopol: O'Reilly Media, 2011. ISBN 978-0596805524.

FRIESEN, Jeff. Java XML and JSON: Document Processing for Java SE. 2nd edition. Apress, 2019. ISBN 978-1484243299.

REINHARTZ-BERGER, Iris a Jelena ZDRAVKOVIC. Enterprise, Business-Process andInformation Systems Modeling: 20th International Conference, BPMDS 2019, 24th InternationalConference, EMMSAD 2019... 1st edition. Springer, 2019. ISBN 978-3030206178.

RUSSELL, Nick, Arthur ter HOFSTEDE a Wil van der AALST. Workflow Patterns: The Definitive Guide. MIT Press, 2016. ISBN 978-0262029827.

STIEHL, Volker. Process-Driven Applications with BPMN. Springer, 2014. ISBN 978-3319072173.

Termín odevzdání diplomové práce je stanoven časovým plánem akademického roku 2021/22

V Brně dne 28.2.2022

L. S.

doc. Ing. Miloš Koch, CSc. garant	doc. Ing. Vojtěch Bartoš, Ph.D. děkan

Abstrakt

Tato diplomová práce se zabývá problematikou procesního modelování a její aplikaci při modelování workflow v rámci frameworku GPC. Konkrétně se zabývá návrhem a implementací modulu, který umožní uživatelům jednoduše transformovat procesní diagramy široce rozšířeného standardu BPMN 2.0 na spustitelná proprietární workflow, která se využívají ke konfiguraci procesů frameworku. Řešení umožní uživatelům přizpůsobit platformu pro své účely způsobem, který je známý, vizuální a podstatně jednodušší než dosavadní řešení.

Abstract

This master's thesis deals with the issue of process modeling and its application in workflow modeling within the GPC framework. Specifically, it deals with the design and implementation of a module which will allow users to easily transform process diagrams of the widely used BPMN 2.0 standard into executable proprietary workflows that are used to configure the framework's processes. The solution will allow users to customize the platform for their purposes in a way that is familiar, visual and significantly simpler than existing solutions.

Klíčová slova

podnikové procesy, procesní modelování, GPC framework, XML, BPMN, Javascript, ECMAScript, JSON

Key words

business processes, process modeling, GPC framework, XML, BPMN, Javascript, ECMAScript, JSON

Bibliografická citace

HAVELKA, Ondřej. *Interaktivní nástroj pro vytváření spustitelných BPMN workflow v platformě GPC*. Brno, 2022. Dostupné také z: https://www.vutbr.cz/studenti/zav-prace/detail/139266. Diplomová práce. Vysoké učení technické v Brně, Fakulta podnikatelská, Ústav informatiky. Vedoucí práce Ing. Viktor Ondrák, Ph.D.

Čestné prohlášení

Prohlašuji, že předložená diplomová práce je původní a zpracoval jsem ji samostatně. Prohlašuji, že citace použitých pramenů jsou úplné, že jsem ve své práci neporušil autorská práva (ve smyslu Zákona č. 121/2000 Sb., o autorském právu a právech souvisejících s autorským právem).

V Brně dne 9. května 2022

.....

Ondřej Havelka

Poděkování

Rád bych v první řadě poděkoval vedoucímu práce Ing. Viktoru Ondrákovi Ph.D. za odborné vedení práce. Také bych chtěl poděkovat vedení a zaměstnancům Expect-IT s.r.o. za spolupráci při tvorbě práce. V neposlední řadě děkuji partnerce a rodině za podporu, kterou mi při psaní poskytovali.

Obsah

Úvod

Tato diplomová práce se obecně zabývá problematikou procesního řízení. Jelikož samotné procesy, jejich znalost a efektivita jsou s příchodem a rozvojem nejnovějších technologií, průmyslu 5.0, IoT, smart buildings či autonomního řízení, důležitější než kdy dřív, je nutné, aby měly organizace vhodné nástroje k jejich návrhu, modelování, implementaci a následnému monitoringu a analýze.

Obdobný nástroj je i v rámci platformy GPC. Ta je obecné SaaS softwarové řešení, které umožňuje plánování, monitoring a automatizaci podnikových procesů. Celá platforma je koncipována tak, že umožňuje přizpůsobení zákazníkovi, takže ji může efektivně používat jak HR oddělení velké korporátní firmy, tak průmyslový závod pro správu svých budov či malý e-shop. Procesy jsou v platformě konfigurovány pomocí tzv. workflow.

V současné době však řešení nabízí omezené možnosti tvorby a úpravy těchto workflow. Ty slouží jako bariéry přístupu k tomu, aby si každý zákazník byl schopný platformu nakonfigurovat podle jeho potřeb. V předchozí práci bylo navrhnuto a implementováno řešení, které umožňuje interaktivní modelování podnikových procesů pomocí editoru BPMN diagramů.

Práce má za úkol nastínit návrh a implementaci konkrétního řešení, které rozšíří funkcionalitu tohoto editoru tak, aby pomocí něj bylo možné vytvářet zmíněná GPC workflow pomocí nových atributů, funkcionalit a transformací mezi těmito dvěma modely. Tyto workflow bude možné pomocí nástroje aplikovat v rámci platformy, což umožní její konfiguraci. Takové řešení pak umožní snížení bariér přístupu ke konfiguraci, zajistí lepší intuitivitu a UX a také převezme agendu, která v současné době spotřebovává lidské prostředky firmy.

Vymezení problému a cíle práce

Cílem této práce je navrhnout a implementovat modul GPC frameworku pro umožnění transformace BPMN modelů na platformová workflow. K zajištění nutných teoretických znalostí slouží první část práce.

V té je podrobně popsána problematika procesního managementu a příslušných témat. Hlavní důraz je kladen na standard BPMN z již zmíněného důvodu dosažení cíle práce. Mimo to je také představen programovací jazyk Javascript a některé jeho aktuální rozšíření.

Druhá část práce se zabývá analýzou současného stavu. Ta se dělí na 4 části: analýza platformy GPC, jejich workflow, editoru těchto workflow a zmíněném editoru BPMN. Jsou vyzdviženy poznatky a současné nedostatky řešení a jejich dopady na uživatele platformy.

Poslední část práce již řeší samotný návrh a implementaci softwarového řešení. Na základě poznatků z analýzy bude navržen modul, který umožní oboustrannou transformaci GPC workflow a BPMN modelů. Nejprve zpracuji předběžný návrh dle požadavků platformy, podle kterého pak bude navržena logika vzájemného mapování prvků. Poté navrhnu obecné procesy transformace, které následně implementuji jako konkrétní softwarové řešení, jenž detailně popíši. Dle specifik a potřeb transformačního modulu navrhnu rozšíření uživatelských rozhraní, které budou facilitovat aplikaci transformací. Tento návrh také následně implementuji. Po zpracování návrhu a implementace transformační modul uživatelsky otestuji, abych zjistil, zda je navržené řešení funkční a vhodné pro reálné použití v praxi. Na základě testování také analyzuji případné nedostatky tohoto řešení a navrhnu směry, kterými by se v budoucnu mohl vést dodatečný vývoj nástroje. Závěrem nakonec shrnu poznatky, které byly zjištěny v rámci zpracování této diplomové práce.

1 Teoretická východiska práce

První část této diplomové práce je určena k vymezení teoretického pozadí, které je nutné k vypracování praktické části. Bude se zabývat především problematikou procesního řízení. V rámci něj se budu věnovat pojmům jako procesní management, proces, životní cyklus v procesním řízení, procesní modelování, BPMN, BPEL a workflow. Na základě těchto pojmů bude představena nejen teorie a základní principy, ale také konkrétní metodiky či detailní popisy. Zvláštní pozornost budu věnovat standardu BPMN, jelikož jej budu využívat v praktické části práce.

Kromě procesního řízení v této části také představím programovací jazyk Javascript, pomocí jehož bude předmět této práce navržen, vypracován a implementován. Kromě základního popisu a vysvětlení se zaměřím také na standardizaci ECMAScript, která se v současnosti využívá pro zefektivnění práce v tomto programovacím jazyku. Také zmíním některé populární javascriptové frameworky.

1.1 Procesní řízení

Procesní řízení (Business Process Management - BPM) je v současnosti již několik desetiletí existující a zavedená manažerská disciplína, jejíž rozvoj koresponduje s rozvojem informačních technologií a jejich integrací do většiny odvětví. Při studiu managementu ji pozorujeme a vnímáme především v oblastech průmyslového a podnikového řízení, ale stejně tak se projevuje ve zdravotnictví, veřejných službách, justici, governmentu a dalších odvětvích. Tato disciplína zastřešuje návrh, modelování, vykonávání, automatizaci, měření, analýzu a optimalizaci podnikových procesů tak, aby organizace zajistila maximální efektivitu, na již lze pohlížet a vyhodnocovat dle libovolných metrik. [1]

Je nevhodné procesní řízení z jeho principu definovat striktním výčtem operací, jelikož se v praxi jedná spíše o souhrn best practises a obecných postupů, jimiž lze optimalizovat chod organizace. Základním principem je definování opakovatelných a předvídatelných podnikových procesů. Jejich zmapování zvýší jeho transparentnost a umožňuje získat obecný náhled. Konkrétní procesy lze poté atomizovat na sled činností, jehož se účastní konkrétní objekty a jenž převádí vstupy na výstupy. Ačkoli procesní

řízení zažilo největší rozvoj s rozvojem informačních technologií, nelze na něj pohlížet jako na novodobou disciplínu. [1]

Jisté prvky a počátky procesního řízení lze pozorovat například v industriální revoluci 18. - 20. století a tehdejších manufakturách, které začínaly pozvolna integrovat velké stroje pro zefektivnění výrobních procesů, které do té doby vykonávali převážně lidé. Tímto způsobem se nezvyšovala pouze produkce a její kvalita, ale také podstata lidské práce, která pozvolna přecházela z práce těžké k práci specializované, což sebou neslo výrazně pozitivní sociální dopady (zvýšení životní úrovně, prodloužení střední délky života atd…). Rozvoj posledních několika desetiletí lze částečně použít jako paralelu k tomuto období, s tím rozdílem, že místo těžkých strojů nyní automatizujeme především softwarem, který nyní nemá za úkol eliminovat fyzickou práci, nýbrž mentální práci. Například algoritmy a modely strojového učení jsou schopné nahrazovat práci datových analytiků a manažerů a dělat kvalifikovanější rozhodnutí, která jsou založena na přesnějších a komplexnějších datových strukturách, a tedy dokáží lépe měřit a odhadovat dopady či výstupy. Další náznaky procesů a procesního řízení lze kromě industriální revoluce pozorovat například i ve starověké Číně, středověké Evropě či průmyslové výrobě během a po druhé světové válce. [1]

1.2 Procesy

V této kapitole se budu věnovat definici a popisu procesů. Procesy a jejich vnímání lze pozorovat napříč celou lidskou historií. V nejobecnějším pohledu se jedná o něco, co se opakuje a je předvídatelné, tedy deterministické. Touto nejzákladnější specifikací již lze procesy odlišit od projektů (a tedy i projektového řízení, jakožto opaku procesního řízení), jelikož projekty jsou unikátní, probíhají pouze jednou a mohou být součástí programu. [1]

Proces lze definovat jako organizovaný sled činností a stavů, které se postupně vykonávají. Tento popis je pro nás relevantní vzhledem ke spojitosti s pojmem workflow, jenž bude v pozdějších částech práce stěžejní. Workflow lze etymologicky rozdělit na dvě části, work a flow. Jakožto work si můžeme představit právě konkrétní prováděnou činnost a jako flow zmíněný sled neboli pořadí, v jakém jsou po sobě činnosti vykonávány. Zároveň platí, že k zachování procesu se tento sled nemůže změnit, v opačném případě bychom se již bavili o optimalizaci původního procesu či novém

odlišném procesu. Samotné činnosti pak mají své aktéry (obvykle objekty nebo subjekty, které je vykonávají) a přeměňují vstupy na výstupy. [1]

V případě procesů také platí, že mohou být a obvykle bývají součástí jiného, nadřazeného procesu. V tom případě se o nich lze bavit jako o podprocesu (také se používá pojem subproces). Tento poznatek je důležitý z toho důvodu, že v případě náhledu na proces se při zanořování či vynořování mohou procesy obvykle plynule transformovat na podprocesy či činnosti a nazpátek. Výjimkou bývají opravdu atomické činnosti jako například podpis formuláře pracovníkem, avšak při zvolení dostatečné míry abstrakce se i o takové činnosti dá uvažovat jako o procesu (uchopení psacích potřeb a formuláře a podepsání, sled pohybů rukou, sekvence uvolnění a kontrakcí konkrétních svalů atd…). [1]

Další členění procesů se váže k jejich hodnotě. Ta může být uvažována například z pohledu organizace, stakeholderů, manažerů, koncových zákazníků či pracovníků, kteří je vykonávají. Hodnota procesu tedy nemusí být nutně měřena jen monetárně, ale také podle složitosti, důležitosti v rámci celku a dalších subjektivních i objektivních metrik. Obvyklé členění podle hodnoty je pak následující: [1]

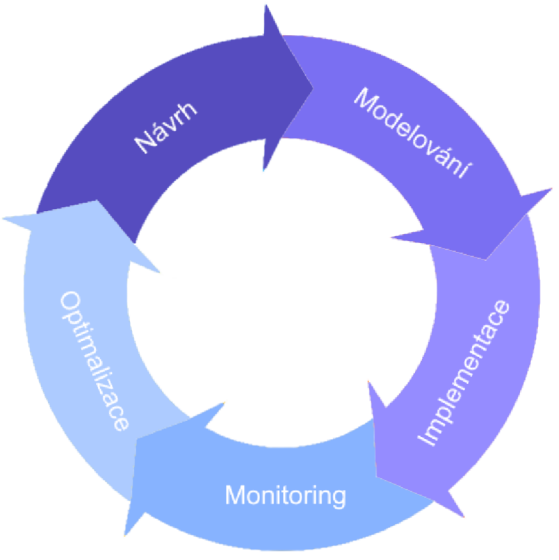
- Hlavní proces, vytvářejí hlavní či nejvyšší hodnotu, jsou stěžejní z pohledu produkce či chodu organizace, bez jejich funkce a existence nemůže subjekt dlouhodobě fungovat, týkají se všech vrstev subjektu. [1]
- Řídící proces, týkají se řízení, organizování, plánování a koordinace celého subjektu a všech jeho částí, mohou v sobě zahrnovat či kontrolovat průběh hlavních procesů. [1]
- Podpůrný proces, veškeré procesy či podprocesy, které zajišťují správný chod a funkci hlavních procesů. [1]

1.3 Cyklus procesního řízení

Pokud jsou v organizaci identifikovány procesy, které již existují, nebo jsou potřebné, je možné započít jejich řízení a optimalizaci. V literatuře bývá životní cyklus procesů nejčastěji uváděný jako pětikrokový. Tyto kroky jsou: [1]

- Návrh
- Modelování
- Implementace
- Monitoring
- Optimalizace

Na následujícím obrázku jsou tyto kroky graficky vizualizovány v kruhovém cyklu. V následujících podkapitolách blíže popíšu jednotlivé součásti cyklu procesního řízení.



Obrázek 1: Cyklus procesního řízení (Zdroj: Vlastní zpracování podle [1])

1.3.1 Návrh

Prvním krokem cyklu řízení je samotný návrh procesu. Návrh zahrnuje jak existující, tak nové procesy. V případě nových procesů se skutečně jedná o jejich návrh takříkajíc "na zelené louce", v případě existujících se jedná o jejich identifikaci a popis. [1]

Primárně se zaměřuje na průběh procesu, tedy zmíněný sled činností a stavů, ze kterých proces sestává. Dále také řeší subjekty, které se procesu účastní, mechanismy kontrolování a měření průběhu procesu. V současné době jde obvykle o upozornění či notifikace, které generuje monitoringový software, SCADA (Supervisory Control And Data Acquisition - stanice pro sběr, agregaci, monitorování a vyhodnocení dat poskytovaných například výrobními stroji v průmyslovém prostředí) a další. Návrh také obsahuje postupy, SLA (Service Level Agreement - smluvní konsensus mezi poskytovatelem a uživatelem služeb), nebo mechanismy pro předávání objektu procesu (například polotovar) mezi jednotlivými činnostmi. Cílem návrhu je získat pochopení toho, jak proces funguje, z jakých částí sestává a posléze zajistit jeho korektnost, proveditelnost a efektivitu. [1]

1.3.2 Modelování

Druhým krokem je modelování procesu, jehož pochopení je stěžejní pro tuto práci. Již korektně navržený proces je zde zmapován jako množina událostí, činností či subjektů, ke kterým se jednotlivé části vážou. V případě potřeby také identifikuje stavy procesu, při kterých může dojít k jeho rozvětvení. V těchto místech se pak musí použít rozhodovací logika, která podle definovaných pravidel (použití logických funkcí) a metrik rozhodne další postup procesem. Tomuto kroku procesního řízení se budu podrobněji věnovat v dalších částech práce. [2]

1.3.3 Implementace

Třetím krokem je konkrétní implementace procesu. Jde o převedení procesu z teoretické fáze do reálného prostředí, kde je produkčně vykonáván. Samotná implementace může být buďto manuální, nebo automatizovaná. Manuální implementace se obvykle týká procesů, ve kterých vystupují převážně lidé, například v případech, kdy se proces soustředí na výměnu kvalitativních informací jakožto hlavních vstupů a výstupů procesu. Automatizace se využívá hlavně u procesů, které jsou řízeny softwarem, který

zpracovává především kvantitativní data a informace, například polohy průmyslových robotů, data ze senzorů a čidel a další informace, které poté předává například SCADA systému. [1]

V případě potřeby může implementace proběhnout i kombinovaně, tedy částečně manuálně a částečně automatizovaně. Tento postup však není doporučený, jelikož procesy implementované obdobným způsobem bývají zpravidla příliš komplikované a komplexní, hůře se optimalizují, mohou být hůře či absolutně lidsky nečitelné a mohou generovat chyby, které způsobí zpomalení, pozastavení či naprosté selhání procesu a tedy logicky negativně ovlivní organizaci. Toto má obzvlášť zásadní dopad například v průmyslové výrobě či kritické infrastruktuře. [1]

1.3.4 Monitoring

Čtvrtým krokem cyklu je monitoring již implementovaného a vykonávaného procesu. V této části se používají metody, které umožňují sledování jednotlivých stavů procesu tak, jak postupně probíhá. Na základě toho lze definovat metriky procesu, které po následné analýze podávají informace o výkonnosti procesu. Tuto analýzu dříve ručně zpracovávali operátoři a datoví analytici, v současnosti se však kvůli prevalenci big data využívají spíše algoritmy strojového učení, které se užívají k detailnímu a komplexnímu data miningu a analýzám a samotní analytici zpracovávají již konkrétnější výstupy. Pomocí získaných informací jednak rozhodujeme, zda je proces funkční a jednak zda je efektivní. Při nalezení nedostatků proces přechází do další fáze. [1]

1.3.5 Optimalizace

Pátým a posledním krokem cyklu je optimalizace procesu. Při té procesní manažer interpretuje informace a poznatky získané během monitoringu. Na základě těch jsou identifikovány problémy, nedostatky či příležitosti procesu, které lze vylepšit. Manažer definuje konkrétní postupy a proces poté přejde opět do fáze návrhu, čímž se spustí nová iterace cyklu. [1]

1.4 Procesní modelování

Tato kapitola se zaměřuje přímo na modelování procesů. Na počátku modelování disponujeme informacemi a znalostmi konkrétního procesu a našim cílem je převést ho do podoby standardizovaného vizualizovaného diagramu. Tímto daný proces zdokumentujeme do podoby, ve které je čitelný, analyzovatelný a dále použitelný v ostatních krocích procesního řízení.

V praxi modelování podléhá jednomu z existujících standardů. Může se jednat jak o obecný standard (BPMN, EPC), tak i proprietární standard (JBPM), ale organizace by se měly vyvarovat vytváření vlastních diagramových standardů, jelikož jimi budou muset nejen školit všechny aktéry, kteří s nimi budou muset pracovat, ale také nebudou čitelné mimo organizaci (například při kooperaci s partnery a dalšími třetími stranami). [2]

Modelování obvykle provádí manažeři a analytici z konkrétního podniku, kteří disponují znalostmi standardů. Kromě toho je ale také zapotřebí hluboká znalost procesů a jejich jednotlivých činností, stavů, účastníků, vstupů a výstupů, aby byl proces korektně modelován a nevznikaly konflikty spojené s jeho nepochopením a neznalostí. Proto je důležité, aby pracovník, který modelování provádí, disponoval těmito znalostmi. V opačném případě se tyto nedostatky řeší tak, že u modelování či před jeho počátkem pracovník konzultuje proces se zaměstnancem, který ho řídí, nebo se jej aktivně účastní, aby byl schopný při modelování korektně identifikovat všechna specifika. [2]

Samotný model je vizualizován podle diagramu podléhajícímu standardu. Obvykle tyto diagramy obsahují počáteční a závěrečné události procesu, činnosti a průběžné události, které mohou sloužit například pro kontrolu, vstupy a výstupy procesu, jeho účastníky, zpracovávaná data, zdroje, rozhodovací logiku a další části. Tyto prvky bývají většinou vizualizovány jako uzly diagramu, či modifikátory existujících uzlů (například pro specifikaci druhu události). Procesní sled událostí a činností je vizualizován spojnicemi mezi těmito uzly, které obvykle bývají orientované, toto bývá vizualizováno například zakončením jednoho konce spojnice šipkou. Na konci modelování pracovník vypracuje diagram, který se v dalších krocích implementuje (ručně či automatizovaně) a stává se součástí organizace. [3]

1.5 Nástroje a jazyky pro modelování

V této kapitole představím několik jazyků a standardů, které částečně či úplně slouží k modelování procesů.

Prvním z těchto jazyků je UML, který vznikl primárně pro účely návrhu a modelování systémů v rámci softwarového inženýrství. Následně se budu věnovat EPC diagramů, jejichž specifikace byla navržena přímo pro modelování podnikových procesů. Jazyk xBML je v současnosti méně známý jazyk, který byl populární během období dot-com bubliny a je opředen kontroverzí, což je jeden z důvodů, proč jej zmíním jako odstrašující případ špatného standardu. [4], [5], [6]

XML a XPDL slouží k serializaci dokumentů. Jazyk XML se používá nejčastěji v kontextu webových stránek, XPDL z něj vychází a používá se pro serializaci BPMN modelů. Samotný standard BPMN je v současnosti jeden z nejrozšířenějších standardů pro účel modelování podnikových procesů se širokou podporou, kompatibilitou a rozšiřitelností. Vzhledem k tomu, že jej budu využívat k dosažení cílů této práce se mu budu věnovat i v samostatné kapitole. [7], [8]

1.5.1 UML

UML je jednotný modelovací jazyk (Unified Modeling Language), který byl původně navržený primárně pro potřebu jednotné platformy pro návrh a grafickou vizualizaci konkrétních systémů, především softwarových. Vznikal na konci 90. let a byl vyvíjen společností Rational Software, která se soustředila na vývoj nástrojů pro softwarové inženýrství. V roce 1997 byl jazyk přijatý jako standard organizací Object Management Group ve verzi UML 1.0. Posléze na jazyku spolupracovaly tehdejší největší americké IT společnosti a díky nim vzniklo několik verzí, které se soustředily na vylepšení jazyku například po stránce kompatibility při transformacích diagramů či modelování vztahů kardinality. [4]

Další důležitá verze standardu byla UML 2.0. Byla vydána v roce 2005 a kromě nových funckionalit, které měly reflektovat změny v IT průmyslu se také celý standard rozdělil do 4 skupin: Superstruktura, Infrastruktura, OCL (Jazyk objektových omezení) a Diagramová směnárna (definovala převádění UML diagramů mezi platformami, typy či formáty). Tato struktura byla však ve verzi 2.5 zjednodušená pouze na UML Specifikaci a OCL. [4]

Samotné diagramy UML mohou spadat do jednoho z mnoha typů, které jsou jazykem zaštitovány. Z tohoto důvodu je dělíme do 2 skupin. První skupinou jsou strukturované diagramy, které reprezentují strukturované informace. Do této skupiny patří následující druhy diagramů: [4]

- Diagramy tříd
- Objektové diagramy
- Diagramy balíčků (package)
- Diagramy profilů
- Diagramy komponent
- Diagramy složené struktury (composite structure)
- Diagramy nasazení (deployment)

Druhá skupina diagramů sjednocuje behaviorální diagramy, které reprezentují informace o chování a interakcích. Jedná se o následující typy: [4]

- Diagramy případů užití (use case)
- Stavové diagramy (state machine)
- Diagramy aktivit
- Diagramy interakcí (dále děleny na časovací, interakční, komunikační a sekvenční)

UML se v současnosti nepoužívá jen k softwarovému inženýrství, ale v určitých situacích například také jako jeden z nástrojů procesního modelování. Obecně je UML silně rozšířeno a relativně čitelné jeho uživateli. Jedná se však o velice rozměrný jazyk, nejspíše i kvůli jeho stáří, a proto se jej složitěji učí noví uživatelé. Také je to důvod toho, že se často používá v případech, pro které není vhodné, a tak vzniká zbytečná komplexita a problémy v rámci organizace a jejich procesů. [4]

1.5.2 EPC

EPC diagramy (Event-driven Process Chain, česky události řízené procesy) jsou jednoduché rozhodovací diagramy pro obecnou vizualizaci procesů. Standard vznikl na počátku 90. let v Německu jako součást frameworku ARIS. V současnosti jsou hojně rozšířené, o jejich vývoj se zasadila mimo jiné německá společnost SAP AG. [5]

Diagramy jsou sledem událostí a činností s podporou rozhodovací logiky. Události v podstatě popisují stavy procesu a samotný přechod mezi nimi je zprostředkován pomocí činnosti. Samotná činnost (také je rozšířený pojem funkce) na sebe může mít navázané další uzly, které reprezentují vstupy a výstupy činnosti, organizační jednotku, která ji provádí, nebo podpůrný systém (v praxi obvykle informační systém nebo databáze). Jak jsem již zmínil, jedná se o diagramy rozhodovací, samotné rozhodování tedy probíhá pomocí logických spojek. Spojka se používá v případě větvení procesu a může jednotlivé větve rozdělovat, sjednocovat, či z nich vybrat další směr sledu procesu. Samotné rozhodování o směru mají na starosti logické operátory AND, OR a XOR. [5]

EPC je velice intuitivní a funkční nástroj pro modelování procesů. Zároveň je relativně snadno převoditelný do strojové podoby, ve které ho lze poskytnout řídícímu systému, který se stará o provoz procesu. Obtíže s EPC přicházejí především v případě příliš dlouhých a komplexních procesů, jejichž vizualizace pomocí diagramu je příliš rozsáhla a uživatel, který ji má analyzovat, se v ní posléze snadno ztratí. [5]

1.5.3 xBML

Metodiku xBML vyvinula společnost BusinessGenetics v čele s Cedricem Tylerem a Stephenem Bakerem na konci devadesátých let minulého století. Zaměřuje se na mapování podniku a jeho procesů s důrazem na informace. Metodika je podporovaná proprietárním softwarovým řešením společnosti BusinessGenetics, který se zaměřuje striktně na xBML, stejně jako samotná metodika, která nepřipouští spojení s jinými standardy a metodikami procesního managementu. [6]

Jádrem této metodiky (a softwaru) je představený koncept pěti W, jedná se o komplexní systém diagramů se zaměřením na: [6]

- Who - Kdo?
- What - Dělá co?
- Which information - S jakými informacemi?
- Where - Kde?
- When - Kdy?

Obdobný koncept (dokonce v širším kontextu) však představil ve svém díle o mapování a modelování podniků John Zachman již v roce 1987, tedy více než 10 let před publikací metodiky xBML. Zároveň jsou dostupné literární prameny k metodice xBML v drtivě většině vydávány právě společností BusinessGenetics a obsahují minimum informací o tom, jak metodiku a software v praxi používat. Spojení nákladných knih, placených seminářů a licencí s tím, že v současnosti již původní web společnosti není funkční (a podle všech dostupných informací společnost prošla rebrandem a vykonává opět stejnou činnost s 5 online lektory) nabízí otázku, nakolik se jednalo o originální metodologii. Všechny důkazy totiž vedou k tomu, že ačkoli se kdysi nejspíš jednalo o použitelnou metodologii, stojí na plagiarismu a spíše připomíná obdobu dnešních podvodů s online finančními guru. [6]

1.5.4 XML a XPDL

XML (Extensible Markup Language) je jazyk vyvinutý v 90. letech, který je dodnes jedním z nejrozšířenějších jazyků pro datovou serializaci. Byl navržený specificky pro potřeby internetu a aplikací, které pomocí něj komunikují. Kromě toho byl při návrhu kladen veliký důraz na to, aby byl jazyk jak strojově, tak lidsky jednoduše čitelný. Sám o sobě není určen pro procesní modelování, ale je rozšiřitelný pro tento účel. [7]

Standard XML Process Definition Language je právě takové rozšíření. Byl vyvinutý k definování procesních modelů a jedná se o nejlepší způsob pro jejich serializaci. Konkrétně se využívá jako XML serializace standardu BPMN. Samotný proces XPDL definuje ve 2 vrstvách. První vrstva uchovává data o logice samotného procesu, návaznosti událostí a činností, definici rolí a tak dále. Druhá vrstva se používá pro data vizualizace procesního modelu, pomocí jejich identifikátorů jsou všechny uzly a hrany diagramu definovány ve dvourozměrném prostoru. Konkrétní uchované informace jsou rozměry a souřadnice umístění (obvykle v pixelech). Díky těmto informacím lze model procesu nejen vykreslit, ale i strojově zpracovat a zapojit do širších systémů. [7]

1.5.5 BPMN

BPMN (Business Process Model Notation) je standard sloužící ke grafické reprezentaci podnikových procesů pomocí diagramů. Tento standard vychází historicky z UML a při jeho návrhu bylo dbáno především na dobrou čitelnost pracovníky v konkrétních profesích, jakožto jeho hlavními uživateli. Do této skupiny patří v první řadě podnikoví manažeři a analytici, v druhé řadě také vývojáři a technici. Díky srozumitelnosti pro obě skupiny usnadňuje jejich komunikaci v rámci procesního managementu a umožňuje kolaboraci napříč celým životním cyklem procesu. [8]

BPMN je v praxi nejrozšířenější standard pro procesní modelování, je snadno přenositelný, čitelný, lze jej transformovat a strojově zpracovávat. Také zajišťuje podporu proprietárních rozšíření standardu, čímž si ho konkrétní organizace mohou lehce přizpůsobit. Stále však platí jako best practise používání jeho originální verze, aby nedocházelo ke zbytečné fragmentaci a problémům s kompatibilitou a čitelností při rozšíření množiny uživatelů. Modely lze v rámci BPMN vytvářet v různých softwarových

nástrojích, patří mezi ně především Camunda Modeler, Bpmn.io, Lucidchart, Draw.io a Yaoqiang BPMN. [9]

1.6 Business Process Model Notation

V této kapitole detailněji popíšu standard BPMN, jenž byl představen v předchozí kapitole. Konkrétně se budu věnovat typům diagramů, prvkům, z nichž se diagramy skládají, strukturu standardu jako takového a nastínění možností rozšiřitelnosti.

1.6.1 Typy procesů

Procesní modely standardu BPMN se dělí do několika typů podle toho, co přesně reprezentují. Obecně je lze rozdělit na soukromé a veřejné, které mají další podskupiny. V této práci se obecně budu věnovat hlavně soukromým procesům, které popisují činnosti v rámci podniku. V této kapitole však představím všechny procesní typy standardu a jejich odlišnosti a určení. [10]

1.6.1.1 Soukromé procesy

Soukromé procesy BPMN představují vnitropodnikové procesy a modelují se vždy v rámci jedné oblasti (prvek diagramů, bude představen v následující kapitole). Také se pro ně používají další názvy jako Workflow a Orchestrace. Soukromé procesy lze identifikovat jako procesy spustitelné a nespustitelné. [10], [11]

Spustitelné (executable) procesy se modelují s důrazem na to, aby byly proveditelné a spustitelné v rámci podniku, obvykle nějakou automatizační platformou, která je zpracuje. V některých případech však můžou postrádat dostatečné informace pro exekuci. V těchto případech je možné buďto poskytnout účastníkovi v dané fázi procesu nějaký rozhodovací interface, nebo používat pro tyto procesy nástavby nad základním standardem BPMN, které exekuci umožní pomocí rozšířených funkcí a proměnných. [10], [11]

Nespustitelné (non-executable) procesy se používají k dokumentaci na úrovni BPMN standardu s dodržením jeho pravidel. Liší se od spustitelných tím, že neobsahují žádná data k exekuci, jedná se tedy o čistou vizualizaci procesu bez jakékoli další funkce. [10], [11]

1.6.1.2 Veřejné procesy

Veřejné procesy se používají pro vyjádření interakcí soukromých vnitropodnikových procesů s jinými procesy nebo účastníky, kteří se přímo neúčastní prvotního soukromého procesu. Přímo obsahuje pouze aktivity (jeden z prvků BPMN, budou popsány v další kapitole), jimiž soukromý proces komunikuje s jeho okolím. Bývají modelovány buďto samostatně, nebo jako součást kolaborace. [10], [11]

1.6.1.3 Kolaborace

Kolaborace jsou množiny několika procesů či podnikových subjektů a modelují jejich vzájemnou interakci v různých stavech a činnostech. Subjekty jsou vizualizovány podle 2 a víc oblastí, které obsahují sled konkrétních aktivit. Vzájemná interakce je modelována komunikačními hranami, obdobně jako u veřejných procesů při komunikaci s jejich okolím. [10], [11]

1.6.1.4 Choreografie

Choreografie slouží k vizualizaci chování a vzájemné interakci dvou subjektů. Modelují se obdobně jako soukromé procesy, ale jejich aktivity (používá se specifický typ) se využívají pouze k zajištění komunikace mezi subjekty. [10], [11]

1.6.2 Prvky procesů

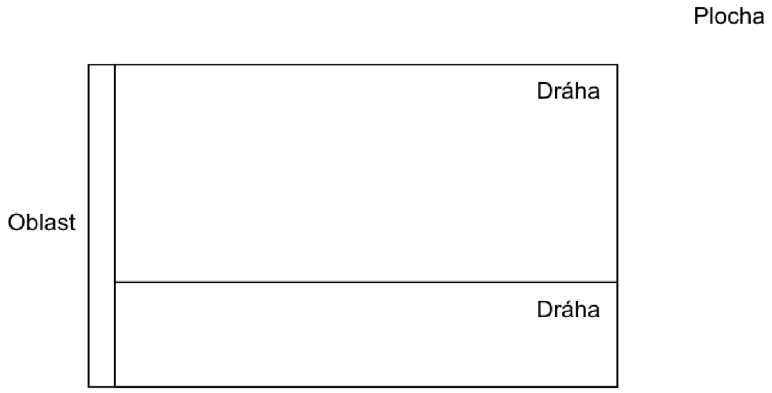
V této kapitole budu popisovat prvky (anglicky elements), které jsou definovány v rámci standardu BPMN. Pro lepší přehlednost jsem je rozdělil do 4 skupin:

- Seskupovací prvky
- Uzlové prvky
- Hranové spojnice
- Dodatečné prvky

Tyto prvky jsou hlavní nástroj manažerů a dalších pracovníků, kteří se standardem pracují a umožňují obecnou tvorbu diagramů. Mnohé z nich mají specifické rozšiřující konfigurace (časovače, zprávy, signály, objekt činnosti atd...), které je mohou blíže upřesňovat, nejdůležitější z těchto konfigurací budou v kapitole také zmíněny. Ačkoli BPMN obsahuje defaultně jen tyto prvky, je možné jej rozšířit vlastními prvky či pravidly dle potřeb konkrétní organizace, ale samozřejmě to poté komplikuje obecnou práci s diagramy. [10]

1.6.2.1 Seskupovací prvky

Jakožto seskupovací prvky uvažujeme součásti diagramu, které slouží pouze k obsažení ostatních prvků. Dle jejich použití lze například definovat typ diagramu (soukromý, veřejný atd…). V širším kontextu pak můžou například přiřazovat ke konkrétním aktivitám jejich aktéry nebo role, které je provádí. V této podkapitole představím jakožto zástupce seskupovacích prvků plochu, oblast a dráhu. [10], [11]



Obrázek 2: Seskupovací prvky (Zdroj: Vlastní zpracování podle [11])

1.6.2.1.1 Plocha

Plocha (anglicky scope, canvas či process) nebývá obvykle definována jakožto konkrétní prvek BPMN. V kontextu této práce považuji plochu za prostředí, v němž modelujeme samotný diagram. V případě konkrétního XPDL zápisu by byla plocha definována jako tag “process”, který slouží jako nadřazený všem ostatním prvkům diagramu. [10], [11]

1.6.2.1.2 Oblasti

Oblast (anglicky pool) v rámci diagramu definuje jeho účastníky. Je vizualizován obdelníkem, který obsahuje název účastníků, může být dále rozdělený na několik dráh a obsahuje konkrétní části procesu. Mimo to se oblasti používají specificky u veřejných procesů a kolaborací. Oblast lze v interaktivních editorech obvykle uzavřít, čímž skryje prvky, jenž obsahuje, aby byl diagram přehlednější. [10], [11]

1.6.2.1.3 Dráhy

Dráhy se modelují v rámci oblastí, které rozdělují do více částí. Používají se například k rozdělení rolí účastníků podle přístupu k aktivitám, událostem či rozhodovacím prvkům. [10], [11]

1.6.2.2 Uzlové prvky

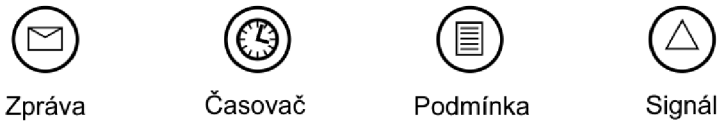
Uzlové prvky jsou naprosto základní komponenty BPMN k modelování procesu. Řadí se mezi ně události, aktivity a brány. Tyto skupiny budou detailně prozkoumány v následujících podkapitolách. [10], [11]

1.6.2.2.1 Události

Události jsou části procesů, které samy od sebe v procesu nastávají. V rámci standardu je dělíme na počáteční, průběžné a závěrečné. V diagramu události vizualizujeme kruhy, jejich konkrétní typ definujeme okrajem těchto kruhů. [10], [11]

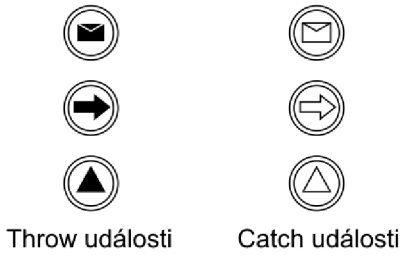
Obrázek 3: Typy událostí (Zdroj: Vlastní zpracování podle [11])

Počáteční události (anglicky start event) identifikujeme podle jejich úzkého jednoduchého okraje. Tyto události definují počátky procesů. Každý BPMN model procesu musí obsahovat alespoň jednu počáteční událost. Počáteční události lze dále modifikovat podle toho, zda událost souvisí se zprávou, časovačem, podmínkou či signálem. Tyto modifikace jsou vizualizovány pomocí příslušných ikon uprostřed prvku události. [10], [11]



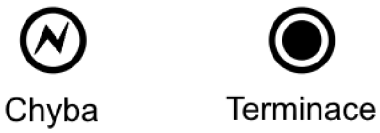
Obrázek 4: Modifikace počátečních událostí (Zdroj: Vlastní zpracování podle [11])

Průběžné události (anglicky intermediate event) identifikujeme podle úzkého zdvojeného okraje. Označují události, ke kterým dochází v průběhu procesu. Také je rozdělujeme na přijímací a odesílací (anglicky catch & throw event). Odesílací průběžné události slouží například k logování hlášek při úspěšném dosažení stavu procesu či dokončení aktivity. Přijímací události existují jako protikus, který například zmíněné zprávy příjme a může dál zpracovat. Modifikátory průběžných událostí jsou obdobné jako u počátečních, kromě výše zmíněných obsahují také eskalace, kompenzace a odkazy. Samotné rozdělení odesílacích a přijímacích závisí na vyplnění ikony modifikátoru - pokud je vyplněná, jedná se o událost odesílací, pokud je jen obrysová, tak se jedná o událost přijímací. [10], [11]



Obrázek 5: Odesílací a příjmací průběžné události (Zdroj: Vlastní zpracování podle [11])

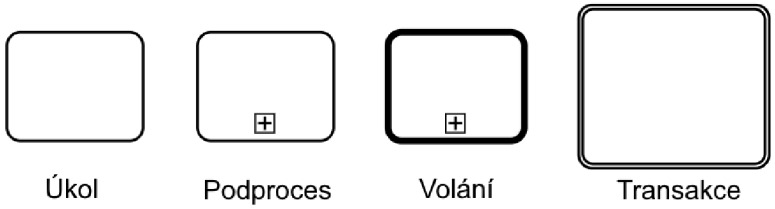
Závěrečné události (anglicky end event) identifikujeme podle širokého jednoduchého okraje. Definují koncové události a stavy procesů a dle pravidel BPMN musí každý proces obsahovat alespoň 1 takovou událost. Mají 2 specifické modifikátory pro reprezentaci neúspěšného dokončení procesu, chybu a terminaci. Při užití terminační závěrečné události tím značíme, že v tomto stavu okamžitě ukončujeme proces. [10], [11]



Obrázek 6: Závěrečné události chybou a terminací (Zdroj: Vlastní zpracování podle [11])

1.6.2.2.2 Aktivity

Aktivity popisují činnost, kterou je potřeba v rámci procesu vykonat a narozdíl od události nemůže nastat samovolně. Jinak řečeno se díky ním modeluje práce, která je v podnikovém procesu prováděná. V diagramu je vizualizujeme pomocí zaoblených obdélníků. Rozdělujeme je na úkoly, podprocesy, volání a transakce. [10], [11]

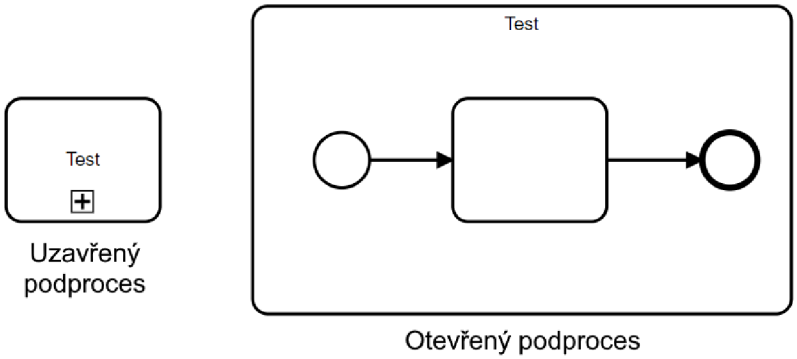


Obrázek 7: Aktivity (Zdroj: Vlastní zpracování podle [11])

Úkoly (anglicky task) jsou atomické činnosti v rámci procesu, které již nelze dále rozdělit či procesně vyjádřit. Pomáhají utvářet jednotlivé podnikové procesy, procedury a postupy. Jejich vizualizace je v BMN vyjádřena pomocí zaobleného obdélníku s jednoduchým úzkým okrajem. Lze je modifikovat obdobně jako události, v jejich případě jsou však ikony modifikátorů v levém horním rohu. Konkrétně jde o příjem a odeslání zprávy (zde funguje obdobné rozlišení ikony jako u událostí), úkol uživatele, podnikového pravidla, služby či programovacího skriptu. [10], [11]

Obrázek 8: Modifikátory úkolů (Zdroj: Vlastní zpracování podle [11])

Podprocesy (anglicky subprocess) jsou složené činnosti v rámci procesu, které lze rozdělit do dalších činností, událostí a dalších artefaktů. Narozdíl od úkolů tedy nemají vlastnost atomičnosti. Mají 2 stavy, otevřený a zavřený. V uzavřeném stavu se vizualizují obdobně jako úkoly s jediným rozdílem. V dolní části obdélníku se nachází tlačítko, které označuje, že podproces je zobrazen ve svém uzavřeném stavu. V tomto stavu s nimi pracujeme jako s černými skříňkami, obdobně jako s úkoly. Pokud je otevřeme, obdélník se rozšíří tak, aby se do něj mohl vykreslit celý proces konkrétní aktivity pomocí klasických prvků. [10], [11]



Obrázek 9: Otevřený a uzavřený podproces (Zdroj: Vlastní zpracování podle [11])

Volání (call activity) jsou zvláštní typ aktivit, který v sobě nenese definici žádné činnosti. Namísto toho funguje obdobně jako volání v programovacích jazycích – na jeho místě v procesu se vykoná úkol nebo podproces, na který odkazuje. Samotný odkaz je zajištěn pomocí shodou názvů s jinou aktivitou. Vizuálně se odlišuje tím, že má široký jednoduchý okraj. [10], [11]

Transakce (anglicky transaction) je speciální druhy podprocesů, které značí operace, při kterých dochází k nějaké platbě. Pravidlem této aktivity je to, že k jejímu dokončení je nutné, aby proběhly všechny aktivity, které se v rámci transakce provádějí. Při selhání některé z těchto aktivit dochází k rollbacku a obnoví se původní stav před započetím transakce. V diagramu se vizualizují pomocí úzkého zdvojeného okraje. [10], [11]

1.6.2.2.3 Brány

Brány se používají k definici rozhodovací logiky v procesu podle vstupů, podmínek, dat a událostí. Dalším jejich užitím může být také větvení sledu událostí v procesu do souběžných podprocesů. Vizualizují se pomocí kosočtverců a jejich typ identifikujeme podle ikony ve středu prvku. V BPMN rozlišujeme 5 druhů bran: paralelní, exkluzivní, inkluzivní, událostní a komplexní. [11]

Obrázek 10: Brány (Zdroj: Vlastní zpracování podle [11])

Paralelní brána slouží k modelování několika souběžných sledů činností v procesu. Může sloužit jak pro rozvětvení (jeden vstup a více výstupů), tak pro spojení (více vstupů a jeden výstup). Narozdíl od ostatních bran paralelní neobsahuje vyhodnocovací logiku a nijak nevybírá konkrétní větev. Je znázorněná kosočtvercem se symbolem plus. [10], [11]

Exkluzivní brána slouží k modelování vyhodnocovacího rozvětvení. Definovaná logika postupně projde výstupy sledů procesu (z pohledu brány se jedná o vstupy) a zvolí první vyhovující sled, čímž vybere větev procesu, která se bude vykonávat. Pokud podmínce nevyhovuje žádná větev, tak proces buďto pokračuje výchozí (musí být definovaná) cestou, nebo skončí s chybou. Exkluzivní brána se vizualizuje pomocí kosočtverce se symbolem X. [10], [11]

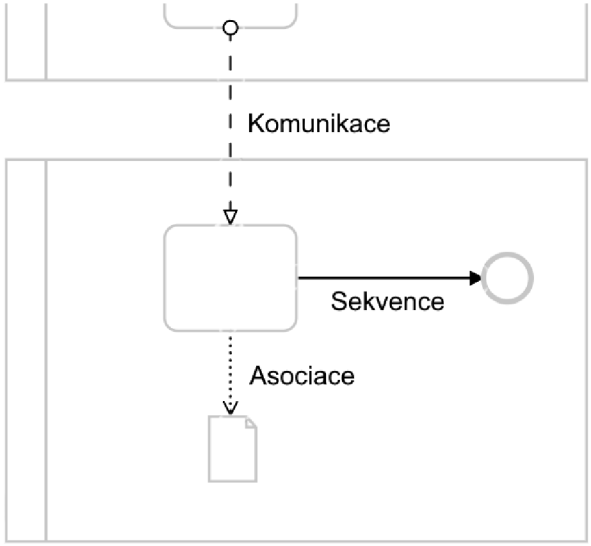
Inkluzivní brána kombinuje rozhodovací logiku exkluzivní brány s možností simultánního průchodu paralelní brány. Pokud se jedná o bránu, která rozvětvuje, tak u všech vstupů vyhodnotí podmínku a spustí ty větve, které jí odpovídají. Pokud brána spojuje několik větví, tak pouze čeká na dokončení všech sledů procesu po bránu a poté v procesu pokračuje dál. Znázorňuje se kosočtvercem se symbolem kruhu. [10], [11]

Událostní brána poskytuje možnost rozhodovací logiky na základě událostí. Používá se ve spojení s několika přijímacími událostmi, u kterých vyhodnotí jejich výsledek a zvolí tu, která odpovídá definované podmínce. Je znázorněna kosočtvercem se symbolem dané (průběžné přijímací) události. [10], [11]

Komplexní brána je vzácně používaný prvek, který se využívá v případě, že by vyhodnocovací logika vyžadovala složitou kombinaci několika bran za sebou, která by byla obtížně čitelná pro uživatele, nebo ani nelze dle výše uvedených bran modelovat. Může být jak rozvětvovací, tak sjednocovací. Graficky se značí jako kosočtverec se symbolem hvězdy. [10], [11]

1.6.2.3 Hranové spojnice

V předchozí kapitole jsem popsal hlavní prvky, které slouží jako uzly BPMN diagramů. Standard nemá pouze jednoduché spojnice, ale dělí hrany, které tyto prvky propojují na několik typů. Konkrétně se jedná o sekvence, komunikace a asociace. Tyto spojnice představím v následujících podkapitolách. [10], [11]



Obrázek 11: Hranové spojnice (Zdroj: Vlastní zpracování podle [11])

1.6.2.3.1 Sekvence

Sekvence určují pořadí provedení sledu prvků v rámci procesu. Jedná se o jednoduché orientované úsečky s počátečním a konečným bodem zakončených šipkou pro určení směru. Počáteční a konečné body slouží jednak pro umístění ve dvourozměrném souřadnicovém systému a také k zajištění referencí propojení prvků procesu. V XPDL kódu BPMN procesu jsou sekvence definované jednak jako samostatný prvek (s referencí na propojené prvky) a také dvakrát jako vstupující a vystupující spojnice v rámci zmíněných prvků. Platí, že spojnice musí vždy spojovat přesně 2 prvky. [10], [11]

1.6.2.3.2 Komunikace

Komunikace (anglicky message flow) slouží pro spojování zpráv v různých typech BPMN procesů. Můžou buďto překračovat hranice oblasti, nebo být v rámci jedné oblasti, dráhy či plochy. Graficky se znázorňují jako přerušovaná orientovaná úsečka zakončená šipkou. [10], [11]

1.6.2.3.3 Asociace

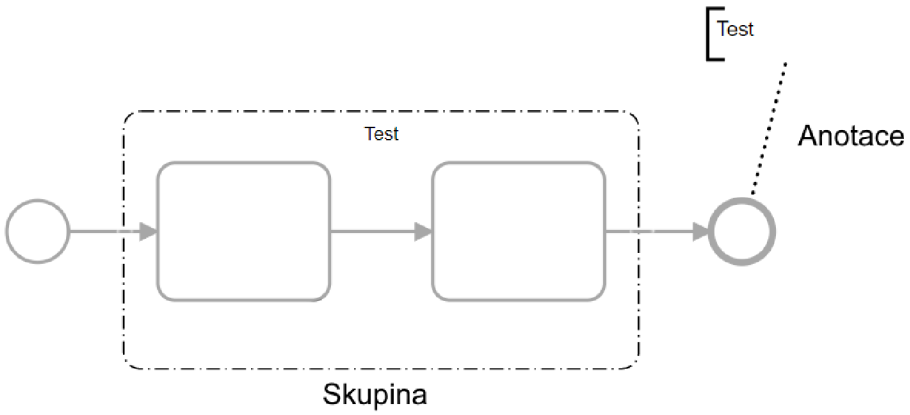
Asociace (anglicky association) se používají k propojení artefaktů nebo textových polí ke konkrétním prvkům. Může být orientovaná, pokud již není směr určen jinou sekvencí. Vizualizuje se jako tečkovaná úsečka, pokud je asociace orientovaná, tak se na jejím konci nachází šipka. [10], [11]

1.6.2.4 Dodatečné prvky

Poslední skupinu prvků, kterou budu zmiňovat, lze obecně označit za dodatečné prvky. Ty se dále rozdělují podle typů užití. Externí vstupy a výstupy se modelují pomocí prvků pro manipulaci dat. Ty jsou ve standardu BPMN datové objekty a datová uložiště. Objekty symbolizují jednoduchá data, uložiště obvykle reprezentují externí databáze. [10], [11]

Obrázek 12: Prvky pro manipulaci dat (Zdroj: Vlastní zpracování podle [11])

Dalším typem dodatečných prvků jsou takzvané artefakty. Ty nemají v procesu žádnou reálnou funkci. Namísto toho slouží uživatelům, kteří pracují s modelem k tomu, aby jim a dalším uživatelům tuto práci usnadnili. Obecně je jejich benefitem především lepší čitelnost a přehlednost modelu. Mezi artefakty patří skupiny, které afunkcionálně seskupují libovolné prvky procesu a anotace, které slouží jako komentáře uživatelů pracujících s modelem procesu. Tyto 4 typy prvků popíši v následujících podkapitolách. [10], [11]



Obrázek 13: Artefakty (Zdroj: Vlastní zpracování podle [11])

1.6.2.4.1 Datové objekty

Datové objekty (anglicky data object) slouží pro vizualizaci informací (dokumenty, e-maily), které se účastní procesu. Rozdělujeme je na 2 druhy. Pokud objekty nejsou součástí procesu, tak se nazývají datovými vstupy a poskytují dodatečné informace užitečné v rámci procesu. V případě že však jsou součástí procesu, jedná se o datové výstupy, které mohou sloužit jako výstupy jednotlivých prvků či celého procesu. Pro jejich vizualizaci se používá symbol přeloženého listu papíru, který může mít levém horním rohu šipku, podle které se identifikuje jako vstup nebo výstup. Dalším modifikátorem jsou 3 svislé čáry v symbolu objektu, které značí, že se jedná o souhrn několika datových objektů (například seznam různých parametrů). [10], [11]

1.6.2.4.2 Datová uložiště

Datová uložiště (anglicky data store) slouží primárně pro interakce s aktivitami, které díky nim mohou znázorňovat zapisování či čtení dat z externího zdroje. Tímto lze v procesu modelovat výstupy, které budou zachovány i po skončení nebo znovuspuštění procesu (jelikož se jedná například o záznam v připojené databázi, proces vytváří v každé své iteraci nový záznam). Graficky se znázorňuje jako trojrozměrný válec. [10], [11]

1.6.2.4.3 Skupiny

Skupiny (anglicky group) nejsou funkcionálním prvkem modelu a slouží pouze uživatelům, kteří pomocí nich mohou organizovat množiny prvků procesu tak, aby byl diagram přehlednější. Skupina je znázorněna zaobleným čtvercem nebo obdelníkem, který má střídavě přerušované a tečkované okraje a svým tvarem a velikostí se přizpůsobuje počtu a uspořádání prvků, které v něm jsou uloženy. Do samotné logiky procesu nijak skupiny nezasahují. [10], [11]

1.6.2.4.4 Anotace

Anotace (anglicky annotation) jsou po skupinách dalším nefunkcionálním prvkem modelu. Jedná se o komentáře uživatelů, které umožňují lepší orientaci nebo kolaboraci nad jedním modelem (v případě, že jej společně upravuje a sdílí například několik pracovníků pomocí mailu) procesu. Mohou být buďto připojeny ke konkrétním prvkům (pomocí neorientované asociace), nebo se nacházet volně na pozadí diagramu. V obou případech pro ně však musí být uloženy souřadnicové informace, právě kvůli slabé či neexistující návaznosti na diagram. Graficky se znázorňují textovým polem, které je z jedné strany úplně a z vedlejších stran částečně ohraničeno. [10], [11]

1.6.3 Rozšiřitelnost

Standard BPMN poskytuje integrovaný mechanismus pro jeho rozšíření. Touto vlastností se odlišuje od většiny konkurenčních jazyků a standardů. Díky proprietárním rozšířením je BPMN vhodnější k použití na modelování procesů ve zdravotnictví, legislativě a dalších odvětvích. [11]

Nejjednodušší způsob rozšíření je pomocí nových artefaktů, tedy dodatečných prvků. K těm se většinou váže jen jednoduchá či téměř žádná funkcionalita, takže přidávání nových artefaktů v teorii nezpůsobuje v procesním modelu konflikty. V praxi je však nutné dodržet jednoduchost artefaktů, jelikož v původním návrhu BPMN se předpokládá, že se jedná o jednoduché prvky a tak se s nimi také nakládá. Jejich přílišná komplexita by přinášela konflikty s vykonáváním procesu a vztahy mezi jeho částmi. [11]

V případě rozšiřování funkčními prvky je v prvé řadě důležité dodržovat základní pravidlo BPMN pro rozšíření. To nám říká, že rozšíření nesmí měnit tvar původně definovaných prvků standardu, nelze tedy v rámci rozšíření například upravit úkoly tak, aby se vykreslovaly jako obdélníky s ostrými hranami. [11]

Nevýhodou rozšíření je však fakt, že BPMN nemá žádný výchozí mechanismus, podle kterého by v diagramu odlišovalo prvky rozšíření od prvků standardu. Proto je v praxi velmi důležité veškerá rozšíření kvalitně dokumentovat (a ideálně stručně představit uživateli, než začne pracovat s modelem) a v nejlepším případě u nich držet jednotný grafický jazyk a vizuálně je odlišit. Toho lze docílit například specifickými identifikačními ikonami u všech prvků rozšíření, či jejich obarvením, jelikož výchozí BPMN prvky nemají barevnou výplň (jedná se o černé vektorové tvary na bílém/transparentním pozadí). [11]

V případě těchto prvků nestačí pouze navrhnout jejich grafickou reprezentaci a implementovat ji do procesního modelu, ale musí se také vytvořit definice rozšíření, která popisuje, jak se rozšíření chová samo o sobě stejně jako v interakcích s ostatními prvky diagramu. Tento proces je velmi složitý a relativně jednoduše v něm může dojít k chybám, které nemusí být na první pohled zjevné, ale mohou se projevit například až po implementaci procesního modelu, což může mít další důsledky v rámci organizace a jejího chodu. Je například nutné definovat prvky standardu, které mohou k rozšíření přistupovat, protože ve výchozí situaci mají tato práva všechny prvky. [11]

Dalším způsobem rozšíření je přidávání atributů prvků. Atributy definujeme pomocí jejich názvu a typu. Konkrétnímu atributu je poté možné přiřadit hodnotu, která musí odpovídat definovanému typu. Pro editování atributů je nutné do modelovacího prostředí přidat rozhraní, které dokáže identifikovat jednotlivé prvky diagramu, z definicí rozšíření rozpoznat atributy, které prvku náleží a poté je upravit pomocí strojového či lidského vstupu. [11]

Při rozšiřování standardu BPMN je důležité nejdříve rozšíření správně naplánovat a analyzovat, zda nemůže docházet ke konfliktům, jestli nejsou příliš komplexní a zda jsou vůbec potřebné. K tomuto je nutné mít bezchybné znalosti procesu, standardu a důvodů pro rozšiřování. V praxi se totiž často setkáváme s tím, že proprietární rozšíření jsou vytvářena ad hoc, aby pokryla aktuální potřebu procesního manažera, ale neuvažuje se již jejich dopad na sémantickou a funkcionální korektnost modelu dle standardu. Tento fakt dále prohlubuje to, že ačkoli BPMN disponuje syntaktickou a logickou flexibilitou pro rozšiřování, neposkytuje k tomuto účelu žádné návody či procesy a postupy, které by měly být dodržovány při návrhu a implementaci rozšíření modelu. Tyto nedostatky standardu musí mít manažer na paměti během definování rozšíření, aby předešel budoucím problémům. [11]

1.7 Business Process Execution Language

Business Process Execution Language (zkráceně BPEL či WS-BPEL jakožto Web Services BPEL) je další standard procesního řízení, který se narozdíl od BPMN zaměřuje již na implementaci a vykonávání procesů. Počátky tohoto standardu leží na počátku 21. století, kdy Microsoft a IBM vyvíjeli 2 obdobné jazyky, které se postupně integrovaly do standardu, jenž se postupem času přetvořil v BPEL. O pár let později obdržel standard další důležité rozšíření v podobě BPEL4People, které do jazyka zaměřeného na procesy, jež pro veškeré vstupy a výstupy dat používají webové služby, přidal podporu implementace pracovníků pomocí definic přístupových rolí. [12]

V současnosti je aktuální verze BPEL 2.0 (vydána v dubnu 2007) jenž přinesla aktualizace ve formě nových programovacích podmínek a cyklů, rozšíření funkcionality proměnných a zlepšila přístup k XML serializaci procesů pomocí XSL transformací a XPath selektorů. [12]

Právě díky podpoře programovacích technik ve spojení s procesními modely je BPEL používaný pro strojovou automatizaci podnikových procesů. Standard však nemá vlastní možnost grafické vizualizace, kvůli čemuž není vhodný přímo pro procesní manažery (pokud tedy nedisponují základními programátorskými vlastnostmi). Kvůli použití syntaxe XML se BPEL hodí spíše pro programátory. V praxi se však obvykle používá pro návrh a modelování procesu již zmíněné BPMN, které se poté ručně či strojově mapuje na BPEL, které lze již implementovat napříč organizací. K tomuto je nutné postupně rozebrat prvky BPMN diagramu a transformovat je na prvky BPEL. Musí však obsahovat všechny potřebné informace a data k exekuci, případně je možné před odsouhlasením BPEL proces validovat a dodat chybějící data. Některé velké proprietární platformy umožňují omezené možnosti této transformace, nelze ale dopředu počítat s tím, že proces transformace lze provést jednou krátkou akcí, obzvlášť když má sloužit v organizaci, která používá rozšíření BPMN nebo nestandardní podnikové procesy. [12]

1.8 Workflow

Pojmem workflow v procesním řízení uvažujeme orientované sledy činností, které za pomocí dalších vstupů uspořádáme do procesů, na jejichž závěru existuje konkrétní výstup (poskytnutí služby či produktu). Workflow jsou specifické tím, že v nich definujeme konkrétní pracovníky organizace či jejich role, které se procesu přímo účastní. Také se liší od obecných procesů tím, že obsahují konkrétně definované stavy procesu, mezi kterými se přechází pomocí činností. Jedná se o jeden ze způsobů, jakými můžeme detailně analyzovat konkrétní úkony a práci zaměstnanců. Díky tomu je možné například identifikovat zbytečné kroky pracovníkovi workflow a patřičně ji upravit, což mu umožní pracovat efektivněji a výkonněji. Jedná se tedy zjednodušeně o konkrétní druhy procesů. Workflow je možné zužitkovat v oblastech procesního managementu, projektového managementu, průmyslové výroby, informačních systémů, IT vývoje, vědě, zdravotnictví a dalších. [13], [14]

Jednou z metodologií, které se používají k návrhu workflow, je Six Sigma. Ta spočívá v minimalizaci ztrát při vykonávání procesů. Řídí se koncepty DMAIC (Define, Measure, Analyze, Improve, Control) a DMADV (Define, Measure, Analyze, Design, Verify), pomocí niž optimalizuje modelované procesy. [15]

Workflow také využívají poznatky z teorie omezení (TOC - Theory Of Constraints). Ta spočívá v identifikaci míst omezení procesu a sledu jeho činností tak, aby pak tato slabá místa mohly být upraveny nebo odstraněny. [16]

Dále se využívá poznatků ze strategie BPR pro kontinuální optimalizaci podnikových procesů či Lean, jenž má za úkol minimalizovat celou strukturu podniku a jeho procesů, aby se zamezilo plýtvání a dosáhlo maximální efektivity. Tyto a mnohé další techniky se v praxi používají ve workflow managementu. [13]

1.8.1 Automatizace workflow

K úspěšné implementaci či úpravě workflow v organizaci je zapotřebí implementace informačního systému, který obsahuje workflow engine a má připojenou databázi s potřebnými daty, které workflow zpracovávají. Engine definujeme jako logické jádro konkrétního softwaru, které pracuje na jeho pozadí a zpracovává funkcionalitu. Workflow enginy zprostředkovávají či zpracovávají jednotlivé činnosti workflow, monitorují jej a zajišťují zachování správné posloupnosti stavů a činností. Také zprostředkovávají výměnu dat mezi uživatelem, informačním systémem či managementovou platformou a připojenou databází a obsahují rozhodovací a vyhodnocovací logiku pro konkrétní podmínky, které můžou být definovány v krocích workflow. Engine musí zaznamenávat všechny existující role, které se účastní workflow a u každého uživatele ověřit, zda má patřičnou roli k vykonání činnosti či přístupu k informacím v konkrétním stavu workflow. Musí taky dohlížet na správný průběh workflow, upozorňovat role v případě, že je potřebná jejich interakce či hlásit, pokud se workflow zpozdí či zastaví v nějakém stavu. V praxi existují již hotové workflow enginy (.NET, Apache, Bonita, Bizagi), obvykle napsané v jazyku Java, které stačí integrovat do informačního systému či platformy. V mnohých případech je ale potřeba navrhovat proprietární workflow a organizace musí vyvinout vlastní workflow engine. Narozdíl od BPMN to však není obdobně závažné, protože workflow nejsou striktně definovány a standardizovány, vzhledem k tomu že se vyskytují v rozličných prostředích a pracují se všemi možnými databázovými strukturami a uložišti dat dle potřeb organizace. [17]

1.9 Javascript

Javascript je dynamický, objektově orientovaný programovací jazyk. Programátoři v něm tedy pracují s objekty, které obsahují data a zároveň je možné po deklaraci typu objektu jeho typ později změnit a přiřadit mu jinou hodnotu. Slouží k interakci softwaru s uživatelem na straně klienta. Jeho nejčastější využití nalezneme na webu, kde slouží k rozšíření funkcionality webových stránek, k chodu webových aplikací a služeb. [18]

Jedná se o velice rozšířený jazyk, dle některých statistik je dokonce nejpoužívanější programovací jazyk (67.8 % vývojářů) a také takřka umožňuje správnou funkci internetu, jelikož jej pro interakce s uživateli a rozšířenou funkcionalitu používá 97,6 % webových stránek. [21], [22]

Ačkoli používá tzv. just-in-time exekuci v prohlížeči (kód je kompilovaný během jeho chodu), existují pro něj některá backendová prostředí, která poskytují kompilaci před spuštěním kódu, společně s integrací javascriptových knihoven a modulů do kódu. Nejpoužívanějším takovým prostředím je node.js. [18]

Pro vývoj webových aplikací se často používají frameworky, které obsahují pozměněnou syntax a nabízí možnosti a funkcionalitu, která je lépe přizpůsobená tomuto účelu. Mezi ně patří například React, Angular, Vue a jQuery. [18], [19]

1.9.1 ECMAScript

ECMAScript (zkráceně se používá výraz ES následovaný celým číslem označujícím konkrétní verzi) je Javascriptový standard. Od roku 1997 se používá pro standardizaci Javascriptu tak, aby bylo možné zajistit jeho kompatibilitu napříč webovými prohlížeči. Historicky totiž jednotlivé prohlížeče implementovaly podporu konkrétních funkcí a chování Javascriptu odlišně, což působilo problémy při vývoji stránek. [20]

Vývojáři museli vytvářet tzv. polyfilly, které emulovaly v problémových prohlížečích nepodporované funkcionality. Tato nutnost výrazně zpomalovala a komplikovala vývoj webových stránek, stejně jako zhoršovala čitelnost kódu. Tento postup se nejčastěji prováděl u starších verzí Safari, Internet Exploreru a občas také Opery a Firefoxu. Díky ES a sjednocování jader browserů (v současnosti projekt Chromium) se podpora JavaScriptu napříč prohlížeči, které mají drtivou většinu market share, víceméně sjednotila. Je však nutné zmínit, že polyfilly úplně nezmizely, jednak je nutné některé zachovávat kvůli uživatelům s neaktuálními prohlížeči, jednak se jedná o obecnou programovací techniku, která není pouze doménou Javascriptu. Kromě toho se dodnes objevuje v css souborech webových stránek. [20]

Alternativou polyfillů pak bylo definovat prohlížeče či jejich verze, které stránka nebo aplikace podporuje a přijmout fakt, že tímto postupem vývojáři odeberou funkcionalitu určité části uživatelů, kteří budou moci web plnohodnotně používat pouze v případě změny či aktualizace prohlížeče. [20]

ECMAScriptu se dostalo asi největší první pozornosti v roce 2015, kdy byla představena verze ES6. Od té doby také zpravidla vychází nová verze ES každý rok. Tato verze přinesla spoustu nových funkcionalit, které se do té doby musely v Javascriptu složitě nahrazovat, nebo úplně chyběly. Především šlo o nové syntaktické změny (šipky pro definici funkcí), nové deklarace lokálních proměnných a konstant (let a const), typovaná pole, mapy či nové cykly a iterátory. Také představila promises, které umožňují asynchronní práci s proměnnými (proměnná při vytvoření nemá hodnotu a očekává, že jí později obdrží a vykoná s ní nějakou operaci). Další velkou změnou byla implementace modulů (import a export) pro lepší organizaci a propojení externích codebases a knihoven. [20]

Ve verzi ES7 byly představeny, mimo jiné, výrazy await a async, které umožnily asynchronní práci s javascriptovými funkcemi. Tato funkcionalita v podstatě nahradila předchozí promises a je nadále rozšiřována. Ve verzi ES9 můžeme zmínit například spread operátor (tři tečky za sebou), který umožnil přenášení vlastností libovolného objektu například při konverzi na pole. ES10 poté přinesl další funkcionality k manipulaci a třídění polí. V současnosti všechny hlavní prohlížeče plně podporují verzi ES7 a poté dle prohlížeče všechny nebo téměř naprostou většinu funkcionalit následujících verzí ECMAScriptu. [20]

2 Analýza současného stavu

V této části práce se zaměřím na současný stav platformy GPC se zaměřením na způsoby, jakými se tvoří workflow sloužící k implementaci podnikových procesů do business logiky frameworku

V první kapitole představím a popíšu platformu GPC, její užití, výhody, technické pozadí a hlavní komponenty platformy, které zajišťují její funkcionalitu.

V druhé kapitole se budu věnovat GPC workflow, které slouží k nakonfigurování platformy dle potřeb konkrétního zákazníka a jeho organizace. Tato workflow následují koncepty zmíněné v první části práce. Blíže se zaměřím především na prvky těchto workflow, jejich syntaktická pravidla, podrobnosti a případy jejich užití.

Ve třetí kapitole se zaměřím na editor těchto GPC workflow, který je součástí platformy takřka od jejího vzniku. Pokusím se zaměřit na postupy práce s editorem, jejich specifik a slabin, které omezují jeho intuitivitu a snižují použitelnost uživateli platformy.

V předposlední kapitole opět představím editor BPMN modelů, který byl navržen a implementován v rámci mé bakalářské práce. Pokusím se analyzovat jeho slabiny a nedostatky, které brání jeho použití k přímé tvorbě GPC workflow.

Veškeré zjištěné poznatky shrnu v poslední kapitole a pokusím se k nim nalézt správná řešení, které by odstranily nedostatky a umožnila zvýšení intuitivity tvorby a editace workflow pro uživatele platformy, což by mělo v důsledku efekt snížení vstupů ze strany společnosti Expect-IT, učinila zákaznickou spolupráci s platformou podstatně soběstačnější a tím zvýšila její potenciál při budoucích soutěžích, kontraktech a vstupu do nových odvětví a trhů.

2.1 Platforma GPC

Platforma GPC (také GPC Framework, zkráceně pouze GPC) je software nabízený jako služba (SaaS), jenž je určený pro popis, standardizaci a automatizaci podnikových procesů. Platforma má compliance se standardy pro veřejnou správu, kromě procesní činnosti také umožňuje určování odpovědností, role management, monitoring procesů či sledování indikátorů výkonnosti. Platforma je proprietární closed-source software vyvíjený společností EXPECT-IT, s.r.o. Umožňuje integraci s většinou standardních nástrojů třetích stran a podporuje použití libovolných datových struktur, uložišť a externích zdrojů, s nimiž je schopná interagovat, používat je k implementaci a monitoringu procesů a ty během celého běhu analyzovat a vyhodnocovat libovolnými metrikami.

Hlavní datové struktury, které GPC vytváří a používá jsou tikety a konfigurační položky. K jejich definici se používají workflow, podle něj se vytváří konfigurační položky, které prochází libovolnými procesy, o jejichž stavu informují uživatele právě tikety. Konfigurační položky lze monitorovat v reálném čase a veškeré jejich změny se uchovávají v záznamech (log) a tiketech, v rámci platformy lze zpětně zobrazit stav položek ke konkrétnímu časovému okamžiku. Celý systém GPC funguje na několika úrovních, které obsahují nástroje a funkcionality oddělené dle potřeby a kvalifikací konkrétního uživatele (dle definovaných a přidělených rolí).

Z pohledu zákazníka je platforma přínosná poté, co obdrží přístup do platformy (případně je implementována v podnikovém prostředí) a projde konfigurací. Během té si zákazník buďto zvolí nějaké z již vytvořených business logik, nakonfiguruje si vlastní či jsou on demand vytvořeny v Expect-IT. Tyto business logiky odráží procesy, pracovní postupy, podnikové know how, a veškeré další potřeby zákazníka (včetně auditových compliance) pro používání platformy. GPC totiž narozdíl od některé konkurence nemá za cíl přizpůsobit chování pracovníka vlastnímu systému, ale modifikovat službu tak, aby fungovala přesně podle jeho přání s minimálními obtížemi, workaroundy a časem stráveným návrhem a testováním před ostrým spuštěním v podnikovém prostředí. Pouze v případě velice specifických, atypických a komplikovaných řešení vzniká nutnost dodatečného vývoje on demand rozšiřujících funkcionalit, které jsou do platformy poté implementovány jako moduly a lze je znovu použít v dalších obdobných případech.

Některé příklady problematik, které jsou vhodné pro užití GPC platformy jsou HR agendy, obecné informační systémy, aukční a tradingové platformy, e-commerce, energetika, municipální management, právní agendy, správa IoT a smart buildings či systémy SCADA a mnohé další.

2.1.1 Součásti platformy

Platforma GPC je tvořena 5 hlavními velkými částmi, jejichž vzájemné interakce slouží k předávání a modifikování dat a společně zajišťují korektní funkcionalitu celého systému. Tyto části definujeme následovně:

- Data & Workflow Management
- Decision Control Logic
- Interoperability Support
- Function Support
- Execution, Modeling & Simulation Support

Ve zbytku této kapitoly se budu zabývat jednotlivými částmi platformy, jejich smyslu a funkcionalitě, aby bylo možné je umístit do logického rozsahu frameworku.

2.1.1.1 Data and Workflow Management

Komponenty pro management dat a workflow poskytují nástroje a funkcionalitu, která uživatelům umožňuje správně definovat procesy, zdroje, třídy a typy dat a vzájemné vazby těchto prvků. Mimo to pomocí nich uživatel může vytvářet a upravovat definice pro vytváření konkrétních typů konfiguračních položek a tiketů, se kterými poté pracují všichni uživatelé platformy dle toho, jaké jim administrátor přiřadí role.

Platforma GPC udržuje všechna interní data (které můžou být generovány z externích zdrojů) ve vlastní temporální databázi a eviduje všechny změny, které jsou prováděny nad definovanými konfiguračními položkami a tikety. Díky tomu je možné monitorovat a analyzovat nejen procesy a data nejen v reálném čase, ale také zpětně v libovolném časovém okamžiku či intervalu.

Tyto komponenty také obsahují nástroje pro definici a analýzu jednotlivých workflow, jejich vzájemných interakcí a jejich implementaci a provoz.

2.1.1.2 Decision Control Logic

Komponenta pro ovládání rozhodovací logiky se skládá z nástrojů a funkcionalit pro automatizaci podmíněných změn stavů konfiguračních položek. K těm dochází obvykle po úspěšném či neúspěšném splnění definovaných podmínek či interakci uživatele. V širším pohledu se jedná o přechody mezi stavy procesů, které jsou modelovány pomocí workflow. Další funkcionalita této komponenty se váže ke korelaci vstupů a událostí implementovaných procesů.

2.1.1.3 Interoperability Support

Komponenta pro podporu interoperability umožňuje platformě interagovat s externími systémy, aplikacemi a datovými uložišti. Díky této komponentě může GPC integrovat většinu řešení používajících standardizované API (rozhraní pro tvorbu a konfiguraci aplikací a systémů).

Mezi některá obvyklá řešení, které se účastní interakce s platformou, patří například MySQL, Microsoft SQL a Oracle SQL databázové systémy, Microsoft Active Directory, Apache Directory Server, cloudová řešení AWS a ServiceNow, nástroje pro virtualizaci VMWare a mnoho dalších.

Platforma interaguje s externími systémy pomocí 2 principů, push a pull. Prvním z nich jsou push metody, mezi které patří například protokoly SNMP, SMTP, Syslog a modul pro detekci změn lokálně uložených souborů. Mezi pull metody patří opět SNMP, TCP a HTTPS protokoly, konzole OS, VMWare VI API, WMI, konkrétní funkce MS Active Dircetory, MS Registrů a LDAP Connectory. Také lze použít SQL pro komunikaci s databázemi, nebo REST a SOAP API ke komunikaci s externími webovými službami.

2.1.1.4 Function Support

Komponenta poskytující podporu funkcí uživatelům umožňuje definovat globální funkce a operátory, které se používají při zpracovávání a vyhodnocování konkrétních datových tříd. Tyto nástroje umožňují filtraci a analýzu konfiguračních položek a tiketů konkrétních typů a jsou užitečné například při zpracovávání big data.

2.1.1.5 Execution, Modelling and Simulation Support

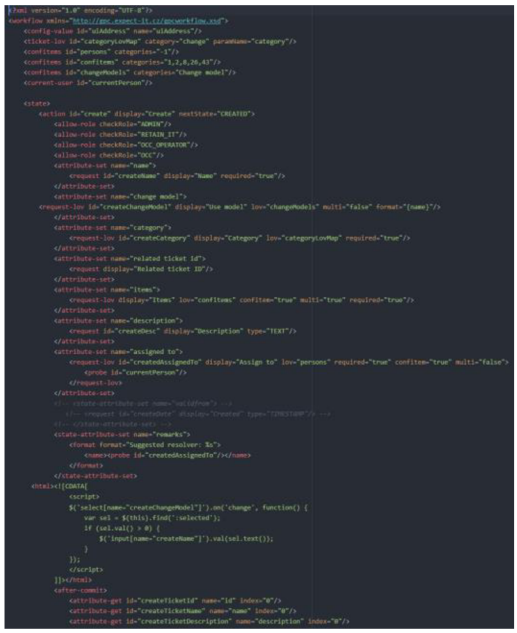
Komponenty pro podporu modelování, spouštění a simulace přinášejí nástroje a funkce, které automatizují procesy a interakce s externími systémy (akce či sběr dat), generování statistik a analýzu těchto procesů a systémů. Mezi konkrétní funkce patří například vyhodnocování dopadů modelovaných událostí, sledování a vyhodnocování trendů či rozhodování dle definovaných pravidel.

2.2 GPC workflow

V této kapitole se budu věnovat GPC workflow. Jedná se o proprietární podniková workflow, které jsou vytvářeny v jazyce XML dle vlastních definicí. Obecně pomocí těchto workflow modelujeme podnikové procesy, které potřebuje uživatel pomocí frameworku zpracovávat či provádět. Workflow definují konfigurační kategorie a tikety poplatné těmto procesů a modelují jejich životní cykly.

První podkapitola se zaměří na hlavní prvky těchto workflow, které fungují jako jejich základní stavební bloky. Poté se budu blíže věnovat syntaxi a formátu GPC workflow. Na závěr kapitoly se zaměřím na jejich používání v praxi a identifikuji specifika a dopady, které s vážou k jejich užití.

Pro ilustraci je na následujícím obrázku náhled GPC workflow pro zpracovávání změny.



Obrázek 14: Náhled GPC workflow pro zpracování změny (Zdroj: Vlastní zpracování)

2.2.1 Prvky

GPC workflow se skládají ze 2 hlavních prvků. Prvním z nich jsou stavy (tzv. state). Ty definují jednotlivé stavy, do kterých se v průběhu procesu může konfigurační položka dostat. Jedná se o primární obecný mechanismus, pomocí kterého lze modelovat proces a zároveň monitorovat jeho průběh.

Stavy mohou být definovány dvojím způsobem. Primární je samotnou definicí ve workflow, kdy je stav vytvořen, pojmenován a obsahuje konkrétní akce, které se během něj mohou ručně či automatizovaně provádět. Sekundární způsob definice stavu je pomocí reference z akce. Každá akce totiž musí mít začátek i konec v nějakém stavu. Akce tak mají vždy nadřazený stav a zároveň atribut "nextState", který určuje, do jakého stavu provedení akce konfigurační položku převede. Pokud je stav definován jen pomocí sekundárního způsobu, pak se jedná o koncový stav procesu (v tomto stavu neexistuje žádná akce, která by ho převedla do jiného stavu), proces přitom může mít pouze jeden nebo i několik rozdílných koncových stavů. Samozřejmě je možné modelovat další proces, který zpracuje konfigurační položky v konkrétním stavu, neznamená tedy že v případě dosažení koncového stavu již konfigurační položka v rámci systému nemůže být změněna. Na následujícím obrázku jsou tyto dva způsoby definicí stavů znázorněny. Zelenou barvou je zvýrazněn primární způsob, modrou barvou sekundární.

Obrázek 15: Způsoby definování stavů ve workflow (Zdroj: Vlastní zpracování)

Již zmíněné akce jsou druhým základním prvkem GPC workflow. Pomocí akcí můžeme definovat samotné aktivity a činnosti, ze kterých se proces skládá. Může se přitom jednat o činnosti vyžadující interakce s uživateli, nebo činnosti, které jsou plně automatizované a probíhají samostatně. Kromě již zmíněného povinného atributu "nextState" musí každá akce obsahovat alespoň identifikátor. Obvykle pak obsahují atribut "display", který slouží pro popis činnosti v procesním modelu, mohou také mít další atributy jako "futureAccesible" pro pozdější vstup do proběhlé akce, či "displayOpened" pro zobrazení aktivních tiketů, které se váží ke konfigurační položce. Tyto ostatní atributy obvykle nabývají binárních hodnot true nebo false.

Samotné atomické činnosti, ze kterých se akce skládá, jsou definovány uvnitř jejího prvku. V první řadě se definují především přístupové specifikace, nejčastěji "allow-param" pro umožnění přístupu podle filtrovaných hodnot či "allow-role" k umožnění přístupu uživatelů s konkrétní přidělenou rolí. Mezi konkrétní atomické činnosti patří práce s daty (získávání vstupů z různých zdrojů a přiřazování konkrétních hodnot atributům či položkám), testování a validace dat, použití cyklů a filtrů a mnoho dalších funkcí.

2.2.2 Syntax

Syntakticky GPC workflow obsahují kromě výše zmíněných prvků v podstatě jen hlavičku. Ta vždy definuje XSD schéma GPC workflow, které obsahuje definice formálního popisu těchto prvků workflow a jejich atributů. V hlavičce mohou pak být další volitelné informace, které již závisí na konkrétním procesu. Můžou zde být předem definovány role, které se mohou účastnit procesu, tyto přístupová privilegia mohou být dále zpřísňovány na konkrétních akcích. Také zde můžeme definovat proměnné a konstanty procesu, či existující konfigurační položky a tikety, se kterými proces pracuje. Ty bývají většinou vybrány pomocí filtrů, které volí například konkrétní typ konfiguračních položek či tikety v určitém stavu (například aktivní, vyřešené apod.).

Zbytek XML souboru workflow poté obsahuje konkrétní definice stavů a jejich příslušných akcí. V rámci akcí se provádějí konkrétní atomické činnosti podle postupu, který byl popsán v předchozí kapitole. Akce v tomto smyslu ovšem nejsou omezeny pouze na jazyk XML, pomocí CDATA (vymezení oblasti textu, která může být interpretována odlišně od zbytku XML) je možné v rámci atomických činností použít například Javascript k programování složitějších cyklů a podmínek při práci s daty.

Formální popis struktury workflow je obsažený v XSD schéma, které je definováno na počátku workflow. Zvýrazňování syntaxe, nabízení nápovědy a automatické doplňování se v editoru tedy řídí XML a definovanému schématu. Na základě těchto definicí je pak při aktualizaci workflow první verifikace prováděná právě na zachování těchto pravidel.

2.2.3 Specifika a problémy

Tato podkapitola se již zaměří na praktickou práci s workflow. Je nutné začít samotným modelováním. To je v současné době prováděno uživatelem, který má znalosti konkrétního procesu (může mít i externí návrh procesu, například pomocí BPMN). Samotné modelování spočívá v "naprogramování" XML kódu a následné testování pomocí ostrých nebo testovacích dat. Pokud je uživatel s výstupy testů, tak může takto vytvořené workflow implementovat ve své (soukromé či podnikové) instanci platformy GPC.

Z tohoto postupu je jasný základní problém procesu modelování. Pro tvorbu workflow je nejen nutné mít rozšířené znalosti pravidel a specifik GPC frameworku a workflow, ale také programovací znalosti a znalost jazyka XML. Proces tvorby je tedy pro většinu potenciálních uživatelů (manažeři, analytici, v případě malých podniků konkrétní OSVČ či jednatelé) příliš komplikovaný, neintuitivní a rigidní. Také kvůli tomuto se v současnosti musí ve společnosti Expect-IT chystat workflow podle specifikací zákazníka či předpřipravovat šablony pro nejčastější podnikové procesy konkrétních odvětví.

Sdílení workflow pro kolaborativní práci na jejich návrhu a modelování je také podstatně omezeno. Workflow editor, který bude blíže představen v následující kapitole, nedává uživateli žádnou funkcionalitu pro sdílení, export (kromě SVG vizualizace diagramu procesu) a kolaboraci. Pro kolektivní modelování tak musí uživatel zkopírovat obsah svého workflow do textového souboru, který poté musí buďto jednoduše sdílet například emailem, nebo za pomocí nějakého řešení třetí strany pro simultánní práci na souborech (Google drive, Git atd…).

2.3 Workflow editor

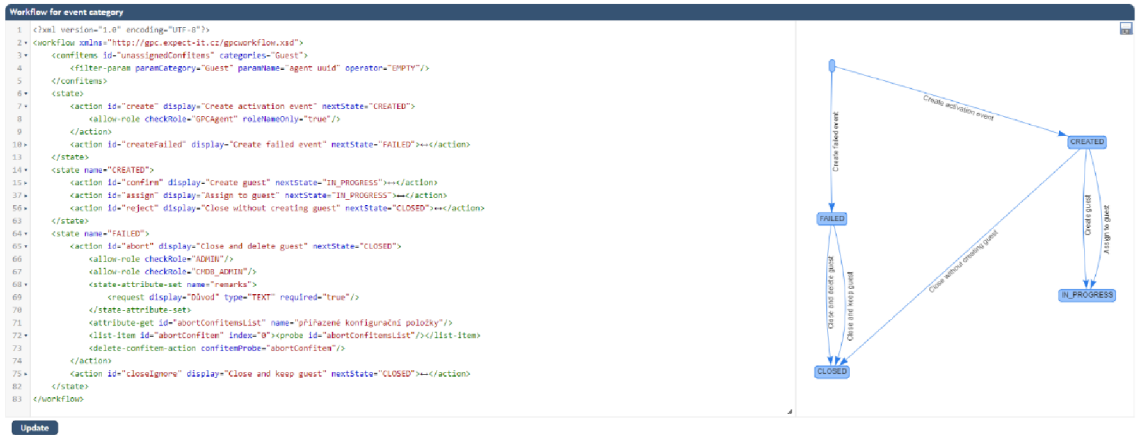
Pro tvorbu a editaci workflow zmíněných v předchozí kapitole slouží v platformě GPC vlastní proprietární editor. Pomocí něj může procesní manažer, analytik či vývojář vytvořit či upravit workflow. Editor také obsahuje validaci syntaktické správnosti a základní funkčnosti. Nachází se na separátní stránce platformy, která se obvykle pro jednoduchost nazývá Workflow Administrator a typicky k ní mají povolený přístup pouze uživatelé s přiřazenou rolí administrátora. Tímto způsobem je zabezpečeno, že neznalý či

nepovolený uživatel pronikne k business logice, kterou se řídí celá platforma a naruší její funkčnost.

Editor se dělí na 3 části. V první části se nachází seznam workflow, které jsou v danou chvíli implementovány v platformě. Podle seznamu si může uživatel zvolit workflow, které chce analyzovat či upravit. Přidávání a mazání workflow je přenecháno v portálu konfigurace administrátora, aby se předešlo nechtěnému smazání workflow.

V druhé části se nachází samotné okno pro editaci workflow. Jedná se o webový textový editor, který obsahuje plnohodnotné vyhledávání (včetně regulárních výrazů), zvýrazňování syntaxe, doplňování výrazů, nápovědu (je definovaná v dokumentaci a může být snadno lokalizovaná pro konkrétní regionální řešení), čísla a seskupování řádků a validaci textového vstupu. Jakmile uživatel dokončí veškeré požadované změny workflow, tak jej jednoduše odešle a implementované workflow se v reálném čase aktualizuje.

Třetí část editoru obsahuje grafickou vizualizaci workflow. Ta je zobrazena v podobě orientovaného diagramu, ve kterém jsou stavy reprezentovány uzly a akce jejich orientovanými hranami. Diagram je vykreslen pomocí grafovací knihovny vis.js a obsahuje tlačítko pro uložení do formátu vektorové grafiky (SVG). Je částečně interaktivní, takže si jej uživatel může přeorganizovat tak, aby měl požadovanou podobu, diagram však vždy dodržuje funkční návaznosti dle workflow. Také se při výběru konkrétního stavu či akce textový editor přesune na řádek, na kterém se tento prvek nachází v kódu workflow. Pro aktualizování diagramu se však musí odeslat a aktualizovat workflow, jelikož jej knihovna musí znovu překreslit.



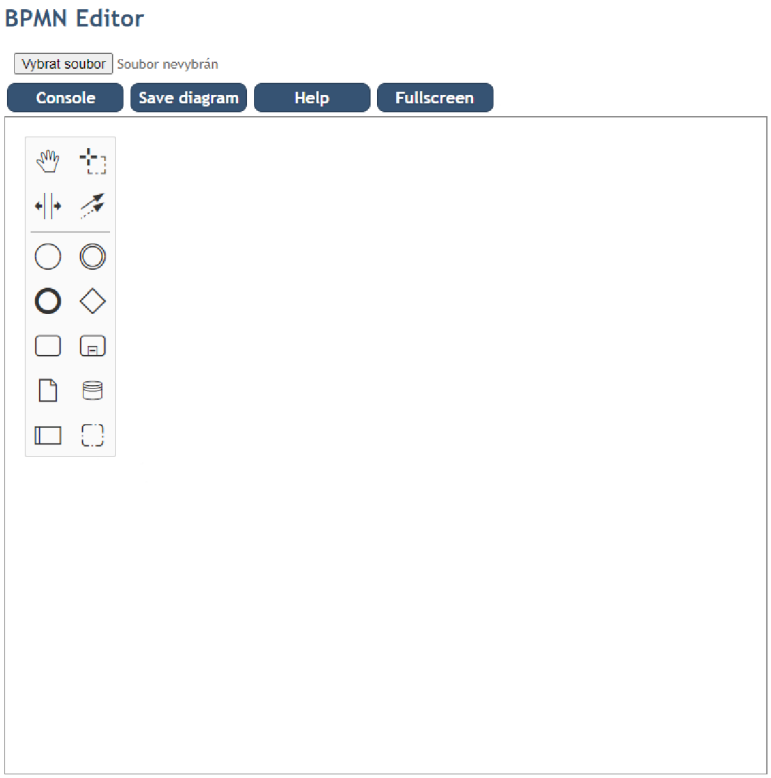
Obrázek 16: Editor workflow s vizualizací procesu (Zdroj: Vlastní zpracování)

2.4 BPMN editor

V rámci předchozí bakalářské práce byl pro platformu navržen a implementován editor BPMN modelů. Jeho účelem bylo přinést uživatelům intuitivní možnost pro návrh a modelování procesů. Standard BPMN byl zvolen pro jeho všestrannost, rozšířenost a relativní pochopitelnost i pro uživatele laika. Jedním ze záměrů bylo přitom implementovat řešení, které by v budoucnu zjednodušilo právě práci s GPC workflow. [24]

Samotný editor se skládá opět ze 2 částí. První z nich slouží pro zprostředkování širší a dodatečné funkcionality. V první řadě se zde nachází tlačítko pro výběr souboru BPMN, který má uživatel lokálně uložený na svém zařízení. Tlačítko otevírá standardní okno operačního systému pro výběr souboru, zároveň obsahuje kontrolu formátu souboru, takže pokud je nevyhovující (podporovány jsou formáty BPMN a XML), soubor neotevře a uživateli poskytne zpětnou vazbu. Další funckionalita se váže k výstupu editoru. Uživatel má možnost buďto vypsat XPDL kód modelu do vývojářské konzole (užitečné pro kontrolu vloženého souboru či průběžný debugging), nebo soubor uložit do svého lokálního uložiště. V druhém případu obdrží okno pro ukládání, ve kterém může zadat název souboru (v případě nevyplnění je poskytnuta výchozí hodnota) a jeho příponu (opět XML nebo BPMN). Soubor je poté stažen dle nastavení prohlížeče buďto do výchozí lokace, nebo do vybraného adresáře. Tlačítko pro pomoc vyvolá modální okno se seznamem klávesových zkratek pro práci s editorem, můžou v něm být také libovolné pokyny pro uživatele či kontakt na podporu, ale to již závisí na konkrétní konfiguraci a preferencích administrátora. Poslední tlačítko umožňuje modelování v režimu celé obrazovky, což je obzvlášť užitečné v případě komplexnějších diagramů. [24]

Druhá část již obsahuje samotný BPMN editor. Ten byl implementovaný pomocí knihovny bpmn-js a vlastních javascriptových funkcí pro nastavení a vlastní funkcionalitu. Okno editoru je plně responzivní a přizpůsobuje se zařízení či velikosti okna, ve kterém je používáno. V levém horním rohu je pak nástrojová lišta s nástroji pro tvorbu BPMN diagramů a jejich manipulaci. Samotné vytvořené prvky pak lze propojovat, modifikovat, či dále upravovat. V editoru se lze pohybovat a také ho přibližovat či oddalovat dle potřeby pro zobrazení kýžené oblasti diagramu. [24]



Obrázek 17: Editor BPMN diagramů (Zdroj: Vlastní zpracování podle [24])

2.5 Shrnutí analýzy

Pro účely globální použitelnosti platformy s minimálními vstupy ze strany společnosti Expect-IT je kritické, aby byly zákazníci a jejich podniky schopné tvořit, upravovat a implementovat vlastní workflow. Současné řešení umožňuje tvorbu workflow pouze se znalostmi programování, jazyka XML a pravidel GPC workflow. Naprosto mu tak chybí intuitivita pro použití manažery či vedoucími malých podniků, kteří mohou disponovat pouze základními znalostmi informačních technologií a elementární chápání standardu BPMN. [24]

V rámci bakalářské práce byl sice navržen a implementován nástroj pro návrh a modelování podnikových procesů ve standardu BPMN, ale dosud chybí rozhraní, které by dokázalo modely procesů převést na workflow. [24]

Z tohoto důvodu platforma vyžaduje možnost transformace modelu BPMN na GPC workflow. Zároveň je nutné, aby transformace vyžadovala co nejméně vstupů od uživatele (po vytvoření samotného modelu), aby byla zajištěna potřeba intuitivity. [24]

Aby bylo řešení kompletní, je vhodné uvažovat transformaci oboustranně, tím pádem budou moci zákazníci převzít již existující workflow, převést ho na srozumitelný procesní model, do toho vložit požadované úpravy a ty poté promítnout zpět do workflow. [24]

Pro tyto účely by bylo vhodné vymyslet správné uživatelské rozhraní tak, aby minimalizovalo potřebu uživatele ukládat redundantní soubory či muset používat textový editor či víc oken prohlížeče najednou. [24]

3 Návrh řešení

V této části diplomové práci se pokusím navrhnout a následně implementovat konkrétní řešení, která budou odpovídat požadavkům a výstupům z předchozí analýzy uživatelských interakcí v rámci editací workflow.

Nejprve se zaměřím na návrh transformačního modulu, jenž je stěžejní pro tuto práci. Tento modul bude obsahovat algoritmickou logiku, která umožní automatizovat procesy oboustranné transformace GPC workflow a BPMN modelů. Nejprve problém rozložím po čemž již budu schopný navrhnout obecný logický popis procesu transformace. Na základě tohoto procesu navrhnu systém vzájemného mapování prvků workflow a modelů a vytvořím základní objektovou strukturu, kterou budou obě strany transformace využívat. Poté již navrhnu a implementuji konkrétní algoritmy a popíšu funkce, ze kterých se budou skládat.

Poté uzpůsobím uživatelské prostředí editorů BPMN a workflow tak, aby umožnili uživatelům využívat transformačního modulu k co nejefektivnějšímu postupu práce při návrhu workflow.

Předposlední kapitola této části se bude zabývat testováním použitelnosti navrženého řešení. Zde zajistím, zda je řešení opravdu funkční a případně analyzuji některé jeho nedostatky.

Těm se budu věnovat závěrem kapitoly, kdy se pokusím navrhnout další dodatečná řešení, které by bylo možné v dané problematice aplikovat do budoucna. Tato řešení však budou pojata pouze stručně, protože se již netýkají náplně této diplomové práce.

3.1 Transformační modul

Tento modul je stěžejní částí řešení. Jeho úkolem je faciliovat transformaci mezi BPMN diagramy a XML workflow platformy GPC. Modul musí být schopen transformaci provést oběma směry, jako jeho vstup může být tudíž buď ve formátu workflow, nebo diagramu.

Vzhledem k odlišnostem mezi těmito formáty (ve zjednodušeném pojetí se v obou případech jedná o XML soubory) bude nutné, abych přistupoval k oboum směrům této transformace jako k separátním problémům. Navzdory tomu se však pokusím procesy transformací co nejvíce sjednotit, aby bylo dosaženo co nejlepší čitelnosti a upravitelnosti. Zároveň tímto přístupem bude možné recyklovat některé části kódu (objektové struktury, funkce, metody), čímž dosáhnu menších nákladů na výpočetní prostředky.

Jednotlivé transformace popíšu v následujících kapitolách. Nejprve se budu věnovat transformaci z GPC workflow na BPMN model, jelikož je komplexnější a pro její zpracování bude nutné navrhnout správný postup mapování prvků, objektové struktury a další postupy a funkcionality. Poté navrhnu transformaci z BPMN modelu na GPC workflow, která by měla být, díky zisku předchozích faktů i díky samotné podstatě této transformace, jednodušší na navržení i implementaci.

U obou transformací nejprve navrhnu abstraktní logické schéma procesu, který povede k úspěšnému provedení úkolu. Následně se budu věnovat k objektovým strukturám, které budou sdíleny oběma transformacemi a budou stěžejním prvkem, který zaručí úspěšnou 1:1 transformaci. Poté popíši již konkrétní navrhovaný a implementovaný algoritmus transformace. Na závěr každé kapitoly tento algoritmus dekomponuji na jednotlivé funkce a vysvětlím jejich funkcionalitu včetně zobrazení implementace. Tyto funkce budu dělit do skupin na hlavní a podpůrné. Hlavní funkce budou ty, které považuji za stěžejní svou rozsáhlostí či komplexností k dosažení výsledku transformace. Podpůrné funkce pak budou plnit úlohy menších procesů, které budou odděleny pro znovupoužitelnost a čitelnost.

3.1.1 Transformace GPC workflow na BPMN model

V této kapitole popíšu proces transformace GPC workflow na BPMN model. Nejprve je nutné určit si podprocesy, které musí během transformace proběhnout. Ty budou záležet především na specifících workflow a BPMN dokumentů, jejich odlišným a společným prvkům či jejich konkrétním požadavkům. Poté vytvořím návrh mapování prvků workflow na prvky BPMN tak, aby bylo možné provést transformaci bez ztráty informací. Zároveň je nutné v této části uvažovat zachování čitelnosti BPMN diagramů v souladu se standardem, aby byla zaručena jednoduchost a přívětivost pro budoucí uživatele. Poté vypracuji obecný návrh algoritmu, který by měl zpracovat transformaci na BPMN model. Podle tohoto návrhu vytvořím funkční implementaci, která bude schopna automatizovaně transformovat GPC workflow na BPMN modely. V posledních podkapitolách pak popíši jednotlivé funkce a metody této implementace.

V rámci transformace bude nutné zpracovat postupně každý stav, jenž je definovaný uvnitř vloženého workflow. Workflow mezi těmito stavy prochází pomocí akcí. Pro úspěšný průběh procesu bude tedy stěžejní transformovat akce a stavy na prvky BPMN, jež je budou reflektovat a zajistit jejich korektní návaznost, aby nebyl narušen tok procesu. Jelikož v rámci každého stavu workflow může být provedena pouze jedna akce (uživatelem či automatizovaně), nebude zde nutná žádná složitá rozhodovací logika.

Po replikaci procesu a jeho toku bude dále nutné zajistit dodatečné informace, jež jsou uložené ve workflow. První z těchto informací jsou role. Role jsou přidělovány uživatelům platformy a umožňují jejich autorizaci pro provádění konkrétních akcí či zobrazení specifických dat. Ve workflow jsou role přiděleny akcím. Jednotlivá akce může mít přidělenou identickou množinu rolí jako další akce, může mít také přidělena jednu roli, ale také nemusí mít definovanou žádnou roli. V posledním případě je pak přístupná všem uživatelům. V transformovaném modelu tedy bude nutné akcím vizuálně přiřadit tyto role.

Další informací, kterou akce nesou, je jejich samotný obsah. Ten se skládá z řady předdefinovaných sub-akcí, které slouží například k nahrávání či ukládání konkrétních dat, manipulaci parametrů tiketů a konfiguračních položek, ale také v sobě mohou mít embedded skripty pro provádění komplexnějších operací. Tyto sub-akce jsou prováděny sekvenčně tak, jak za sebou následují v serializovaném workflow. Tím pádem neobsahují žádnou rozhodovací logiku, musí však být zachována jejich posloupnost. Bude nutné je však převést z podoby XML tagů do vizualizace a zároveň zachovat jejich data. Tato funkcionalita je již v tomto prvotním pohledu reprezentačně složitá a bude pro ni nejspíše nutné rozšířit používanou BPMN implementaci.

3.1.1.1 Mapování prvků

Po výše uvedeném abstraktním vymezení procesu transformace se nyní pokusím navrhnout vhodné mapování prvků workflow na prvky diagramu BPMN.

Pro účely reprezentace stavů jsem původně zamýšlel události (BPMN prvky event). Tento přístup však postupem času ukázal své nedostatky, především přílišným pozměněním modelovacích praktik standardu BPMN. Proto jsem přešel k reprezentaci pomocí bran (BPMN prvky gateway). Jelikož umožňují větvení a sjednocení toku procesu, ideálně se hodily pro potřeby stavů, které obsahují několik akcí. Již v předchozí iteraci pomocí událostí byly brány pro větvení používány a generovány pomocí dodatečné logiky. Bylo tedy ověřené, že zvolený přístup bude fungovat a zároveň se tím zjednodušily jak výsledné BPMN modely, tak samotný kód. Jelikož lze v každé instanci stavu provést pouze jednu akci, stačí využívat pouze exkluzivní brány.

Pro reprezentaci akcí workflow se prvotně nabízely hranové spojnice (BPMN prvek sequence). Ty jsou totiž použity v diagramech znázorňující současná workflow. Jejich užití jsem však v rané fázi projektu zamítl. Diagramy BPMN jsou totiž uzlově definované a hrany v nich pouze reprezentují tok událostí (jejich návaznost) a nenesou žádné data. Uživatelům by tedy nedávalo smysl tyto spojnice používat jako nositele informací, obzvlášť v případech komplexnějších akcí, které obsahují několik desítek sub-akcí. Vizualizace by byla téměř nemožná. Začal jsem tedy pro reprezentaci akcí nejprve využívat činnosti (BPMN prvek task). Ty umožnily postup na složitější logiku uvnitř transformace, a nakonec její funkční dokončení tak, že byla schopná správně zachovat tok procesu a veškeré návaznosti. Později se však ukázalo, že reprezentace obsahu akce,

tedy sub-akcí, není vizualizovatelná v rámci BPMN a externě stejně jako v prvním případě působí příliš uživatelsky složitě. Z tohoto důvodu tedy byly pro účely vizuální reprezentace akcí zvoleny podprocesy (BPMN prvek Sub-process). Ty mají v rámci BPMN dvě odlišné formy:

- složené (collapsed)
- rozložené (expanded)

Podproces ve složené formě vypadá obdobně jako činnost. Bylo tedy možné je pouze nahradit za činnosti, aniž by se tím změnila funckionalita transformace. Při rozkliknutí podprocesu však přejde do své rozložené formy, ve které je pak možné vizualizovat konkrétní sub-akce. Díky tomu lze jednoduše docílit jak zobecněného zobrazení procesu, které je přehledné, tak komplexnímu zobrazení jednotlivých akcí, kde lze již upravovat jejich obsah při zachování dobré míry vizualizace. Je tedy možné akce postupně dekomponovat a řešit jako samostatné problémy, aniž by uživatele rozptyloval zbytek procesu.

Samotné sub-akce jsou vizualizovány pomocí činností v rámci svého podprocesu. Jelikož probíhají sekvenčně za sebou, není nutné uvažovat žádnou pokročilou rozhodovací logiku a lze je tedy při zachování posloupnosti napojit sekvenčně za sebe. Tato sekvence činností je poté ohraničena počáteční a závěrečnou událostí, které reflektují počátek a konec podprocesu, tedy slovy workflow započetí a dokončení akce. Dále však není možné sub-akce vizuálně rozdělit, jelikož mohou zastávat několik desítek různých funkcí. Proto pro konkrétní reprezentaci jejich dat bude nutné využít rozšířené atributy, které bude možné zobrazovat a upravovat pomocí panelu, který byl zmíněný v předchozí kapitole.

Počáteční a konečné události je také nutné použít na obecné úrovni procesu, jelikož standard BPMN je definovaný tak, že každý proces musí mít svůj začátek a konec.

Počáteční událost bude použita pouze jedna, avšak její použití se bude lišit podle definice konkrétního workflow. V případě, že v počátečním stavu je definovaná pouze jedna akce (např. vytvoření tiketu), může být vizualizován pomocí zmíněné události. Pokud však počáteční stav obsahuje několik akcí, nebo je možné se do něj nějakou akcí později v procesu navrátit, musí být vizualizován pomocí brány. V takovém případě se vytvoří výchozí počáteční událost, která bude zastupovat počátek procesu a bude na ní navazovat právě zmíněná brána počátečního stavu.

Použití konečných událostí je po předchozí definici již snadnější. Workflow obsahují vždy alespoň jeden stav, který funguje jako zakončení jejich procesu. Obvykle se jedná např. o stav Uzavření tiketu, může také dojít k jeho smazání či jinak definovaným konečným stavům. Takové stavy lze tedy vizualizovat pomocí konečné události. Toto zjednodušení však lze využít pouze, vede-li do stavu jen jedna akce. V případě více akcí končících ve stavu, který již sám žádné vlastní akce neobsahuje, musí být stále reprezentován pomocí brány. V tomto případě lze pak na výstup brány napojit právě konečnou událost, která bude nazvaná identicky jako konkrétní konečný stav.

Posledním část workflow, kterou je potřeba vizualizovat, jsou role. Jelikož se pro toto užití v rámci BPMN modelů již používají oblasti a dráhy (BPMN prvky pool a lane), můžu toto využití zachovat. Díky tomu bude jejich používání pro uživatele intuitivní. Jelikož akce mají zpravidla definovány několik rolí, je nutné tyto role nějak zkombinovat při vizualizaci. BPMN oblasti neumožňují jejich vrstvení a každý prvek může být obsahem pouze jedné oblasti. Nelze tedy vizualizovat každou roli podle její vlastní oblasti či dráhy. Z tohoto důvodu musím být schopný v procesu transformace získat jak sadu všech vyskytnutých rolí, tak jejich kombinace u konkrétních akcí a podle těchto kombinací vytvářet konkrétní oblasti a dráhy, které budou obsahovat dané akce. Ostatní prvky, tedy brány a události nejsou v rámci workflow nijak vázány na konkrétní role a tak budou oblasti a dráhy obsahovat pouze podprocesy, tedy vizualizované akce.

Pokud prozkoumáme složení serializovaného BPMN souboru, můžeme si všimnout, že se skládá ze dvou hlavních entit:

- proces
- diagram

Proces obsahuje veškeré prvky diagramu, a definuje jejich propojení pomocí hran a jejich dodatečné informace (např. identifikátory a názvy). Diagram využívá takto definované prvky a obsahuje data o jejich vizuální reprezentaci na dvourozměrném plátnu. Každý prvek zde tedy má uvedeny své rozměry (šířka, výška) a své souřadnice na plátně (osy X a Y). Jednotkou těchto dat jsou pixely. Jelikož workflow žádné tyto informace neobsahuje, protože samo o sobě nebylo původně určeno k vizualizaci, je nutné tyto data vygenerovat automatizovaně.

K těmto účelům lze využít open source knihovnu bpmn-auto-layout. Tato knihovna je udržována několika vývojáři týmu, jež odpovídá za projekt bpmn-js, pomocí jehož byl dříve vytvořen používaný editor BPMN a umožňuje automatické vygenerování diagramové části modelu. Jako vstup používá proces BPMN (který získám pomocí vlastní transformace) a řídí se pravidly, která jsou užívána k rozmístění nově vytvářených prvků při editaci diagramu v editoru [23].

Jak jsem již zmínil dříve, v rámci sub-akcí bude jejich obsah reprezentován rozšířenými atributy. Pro editaci těchto atributů pak budu využívat opět zmíněný grafický panel. Pro něj je však nutné vytvořit řídící logiku. Jelikož jsou sub-akce definované v rámci platformy, je možné vytvořit slovník, který bude obsahovat veškeré známé sub-akce. Těm pak mohou být přiděleny konkrétní atributy, které mohou či musí obsahovat. Hodnoty těchto atributů mohou být vybírány z číselníků, či omezeny na konkrétní datový typ, přičemž tyto informace mohou být opět obsaženy ve zmíněném slovníku. Jelikož využívám skriptovací jazyk JavaScript, pro slovník je optimálním formátem JSON.

3.1.1.2 Základní logické schéma

V této podkapitole se pokusím navrhnout obecný algoritmus pro transformaci workflow na BPMN model. Tento proces lze v nejvyšší míře abstrakce de facto připodobnit k ETL procesu v rámci problematiky datových skladů. V mém případě musím extrahovat data z workflow, transformovat jednotlivé části dat tak, aby odpovídaly standardu BPMN a na závěr z nich sestavit BPMN model, který bude nahrán do BPMN editoru, který jej zobrazí uživateli.

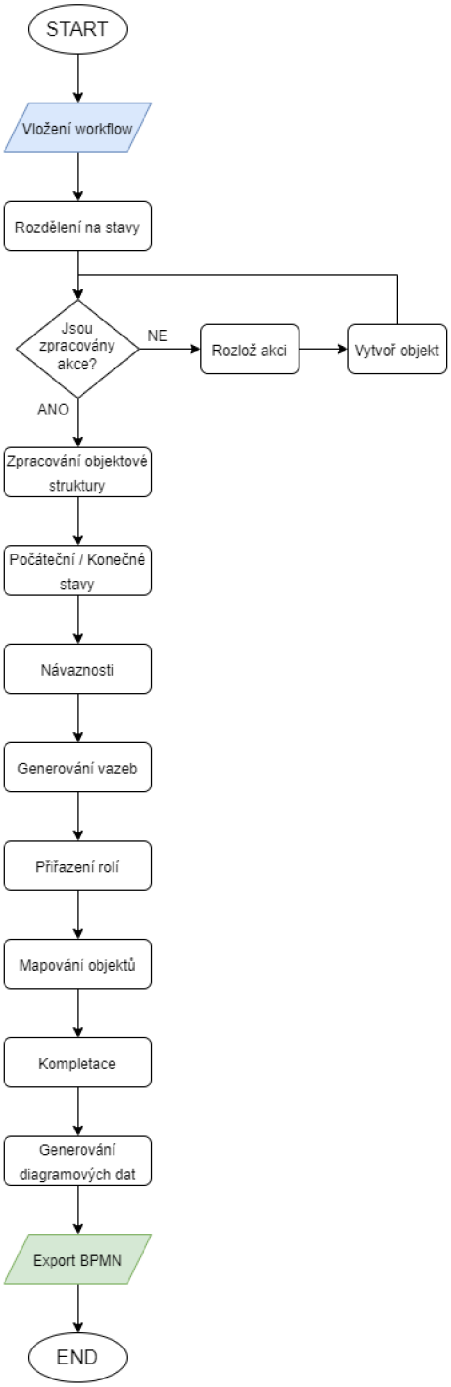
V první fázi tedy algoritmu poskytnu workflow soubor, který pak rozloží na jednotlivé stavy a jejich akce. V rámci každého stavu definuji jeho sadu akcí, pro něž vytvořím objekty, které ponesou informace o akci a stavu, do kterého má přejít. Při tom také pro každý stav vytvořím objekt, na který se bude odkazovat každá ze sady akcí jako její zdroj.

Ve druhé fázi nejprve zajistím správný tok akcí a stavů. Jelikož akce v tuto chvíli obsahují pouze částečné informace o cíli (název cílového stavu), je nutné jim přiřadit konkrétní objekty, které byly vytvořeny v předchozí fázi. Stejně tak musím objektům stavů přiřadit příchozí a odchozí objekty akcí. Nyní již existuje objektová struktura popisující podrobný postup procesu a lze vyhodnotit, které stavy jsou počáteční či závěrečné. Dle počtu jejich příchozích či odchozích akcí lze vytvořit objekty reprezentující počáteční a konečné události podle postupu, který byl definován v předchozí podkapitole. V tomto stavu transformace již existují objekty pro veškeré uzlové prvky budoucího modelu (s výjimkou obsahu akcí, ten bude však vhodnější vyřešit v pozdější fázi). Dále tedy můžu s využitím vzájemných referencí objektů vytvořit hrany BPMN modelu (spojnice), jež obsahují vždy identifikátor zdrojového a cílového objektu (uzlu). V neposlední řadě také v této části mohu vytvořit oblasti a dráhy pro existující kombinace rolí a přidělit jim objekty podprocesů.

Ve třetí fázi již mohu pro každý objekt vytvořit XML prvek dle standardu BPMN. Podle typu objektu se vytvoří konkrétní druh prvku a provedou případné dodatečné operace (například v případě podprocesu jeho vyplnění transformací sub-akcí). Pak už lze z vytvořených prvků vytvořit kompletní BPMN proces, který lze zobrazit jako textový řetězec či objektový model. Pro použití v BPMN editoru k zobrazení či editaci bude nutné proces převést na XML textový řetězec, který se použije jako vstup již zmíněné knihovny

bpmn-auto-layout, která vygeneruje diagramovou část procesu a kompletní model pak bude možné buďto uložit do souboru, nebo přímo zobrazit v BPMN editoru.

Logické schéma transformace je vizualizováno v následujícím vývojovém diagramu:



Obrázek 18: Logické schéma transformace workflow (Zdroj: Vlastní zpracování)

3.1.1.3 Objektová struktura, konstruktory

V předchozí podkapitole jsem zmiňoval v rámci transformace několik druhů objektů. Jazyk Javascript obsahuje metodu constructor. Díky této metody je možné vytvářet instance objektů dle definovaných tříd. V rámci zpřehlednění a zjednodušení implementace jsem se tedy rozhodl, že si vytvořím třídy pro každý typ objektů, čímž si zjednoduším a zpřehledním práci s danými objekty dle jejich typů.

Pro každou vytvořenou třídu si také vytvořím globální počítadlo a pole. Počítadla budou sloužit k vytváření identifikátorů objektů, ale budou moci také dopomoci k lepší orientaci a řádu. Do pole se budou přidávat objekty dané třídy, což později zjednoduší a umožní jejich iterování pro potřeby vyhledávání či editace. Každá nová instance třídy zvýší přiřazené počítadlo a vloží vytvořený objekt do správného pole.

V první řadě vytvořím třídu pro podprocesy, které budou reprezentovat akce workflow. Objekty této třídy budou obsahovat následující vlastnosti:

- identifikátor
- název
- následující stav
- předchozí a navazující uzel
- předchozí a navazující hrana
- množina rolí
- obsah akce

Obdobná bude třída pro činnosti, jenž reprezentují sub-akce workflow. Nebudou však obsahovat vlastnosti o stavu, rolích a obsahu. Namísto obsahu budou nést informace o atributech dané sub-akce.

Nejjednodušší třídou budou sekvence, tedy hrany BPMN diagramu. U těch si v konstruktoru vystačím pouze s vlastnostmi identifikátoru, zdrojového a cílového prvku.

Dále vytvořím třídu pro brány, které budou reprezentovat stavy workflow. Objekty třídy budou obsahovat následující vlastnosti:

- identifikátor
- název stavu
- předchozí a navazující uzly
- předchozí a navazující hrany

Narozdíl od podprocesů budou pro předchozí a navazující prvky vytvořeny pole, jelikož brány mohou mít více než jeden vstupních a výstupních prvků.

Poslední třídou budou události, které budou v této transformaci vždy generovány automaticky (neprobíhá přímé obecné mapování z workflow prvků) a budou obsahovat vlastnosti:

- identifikátor
- název
- předchozí či navazující uzel
- předchozí či navazující hrana

Po vytvoření těchto objektu jim budou přiřazeny ještě vlastnosti start či end, které při transformaci na BPMN prvky využiji na rozpoznání typu události. Jelikož však tato informace není známá při vytvoření objektu, není (stejně jako v případě ostatních tříd a informací) součástí konstruktoru.

3.1.1.4 Algoritmus transformace

V této podkapitole popíši konkrétní implementaci navrhovaného řešení k provedení transformace GPC workflow na BPMN model.

Po vložení XML souboru workflow je proveden převod na textový řetězec. Ten používám jako vstup hlavní funkce createTree(). Ta nejprve převede řetězec na objektový model a vytvoří proměnnou, do které bude nakonec ukládat výsledný BPMN proces. Tuto proměnnou pomocí funkce createBpmnHeader() naplní hlavičkou BPMN modelu. Funkce pak provede cyklus pro všechny stavy workflow. V rámci těchto stavů vytváří pomocí konstruktorů pro každý stav jeho objekt instance brány. Také ve stavu vytvoří pole z jeho akcí. Tímto polem pak iteruje a pro každou akci vytvoří objekty pomocí funkce handleAction(), která má vstupní parametry brány a akce. Pro každou akci se tak

vytvoří její instance, která se naplní požadovanými daty. Reference navazujících objektů se zajistí vyhledáním v poli instancí bran.

Poté je spuštěna funkce assignRoles(), která identifikuje všechny unikátní role procesu, stejně jako všechny kombinace, pro které je nutné vytvořit oblast či dráhu. Ty pak naplní referencemi na instance podprocesů (pomocí identifikátorů). Další funkcí je distributeSubs(), která doplní některé chybějící reference v instancích bran a podprocesů a u vybraných bran zavolá funkci setBorderEvents() pro vytvoření instancí počátečních či konečných událostí.

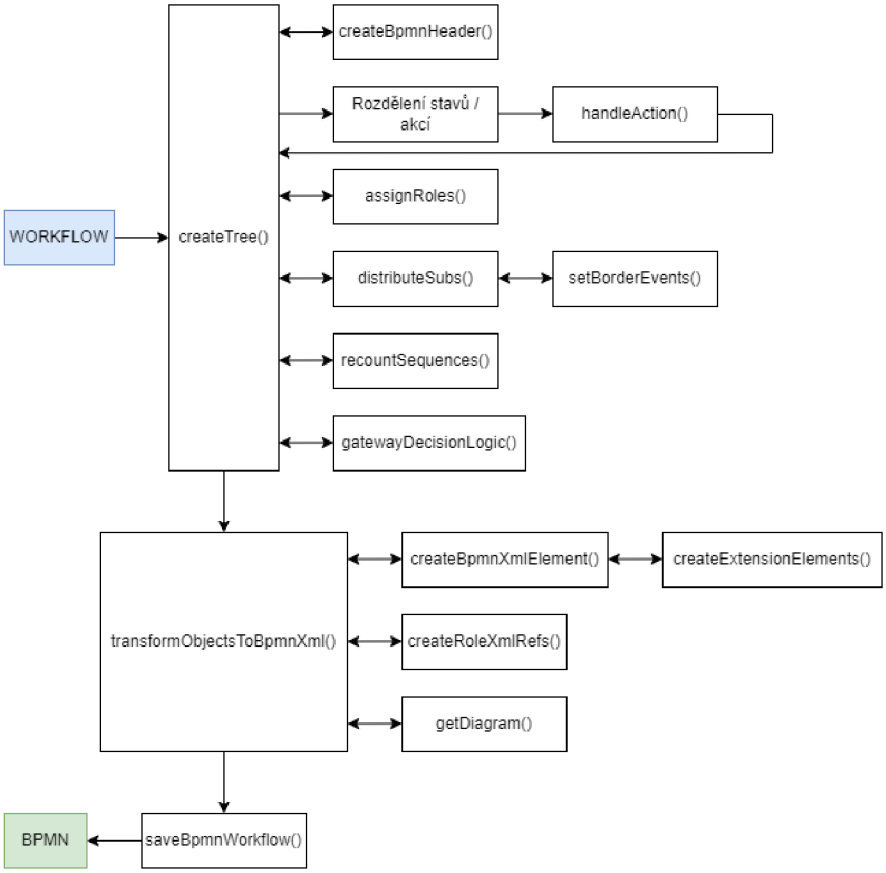
Dále použiji funkci recountSequences(), která vytvoří instance sekvencí, které reprezentují veškeré hrany výsledného diagramu. Prochází přitom všechny instance uzlových prvků a vytváří sekvence jejich referencí. Zároveň aktualizuje reference sekvencí v těchto existujících objektech. Následující funkce gatewayDecisionLogic() projde všechny instance bran a u odchozích sekvencí aktualizuje jejich popis tak, aby reflektoval následující akce. Tato funkce je použita proto, že při modelování BPMN diagramů je zvyklostí pojmenovávat všechny spojnice, které vychází z hran (obvykle slouží k vyhodnocení rozhodování, v případě workflow tato informace není nutná, ale zachová zvyklosti BPMN).

Závěrečnou funkcí je transformObjectsToBpmnXml(), která projde všechny vytvořené objekty (jsou jí předány ve sjednoceném poli) a pomocí funkce createBpmnXmlElement() pro každý z nich vytvoří serializovaný bpmn prvek. Pokud se jedná o podproces, pomocí funkce createExtensionElements() je vytvořen jeho obsah, který reflektuje přiřazené sub-akce. Následně jsou pomocí funkce createRoleXmlRefs() vytvořeny BPMN prvky pro oblasti a dráhy. Nakonec se do textového řetězce BPMN procesu přidá zápatí a samotná transformace procesu je dokončen.

Pro plnohodnotný export se zavolá funkce getDiagram(), která pomocí knihovny bpmn-auto-layout vygeneruje diagramovou část procesu. Výsledný BPMN model lze získat pomocí funkce saveBpmnWorkflow(), která jej uloží do souboru.

Tento soubor lze nahrát do editoru a dále upravovat. Toto může proběhnout buďto ručně, nebo automaticky, kdy se uživateli přepne editor workflow do editoru BPMN, ve kterém je již nahraný BPMN model reprezentující původní workflow.

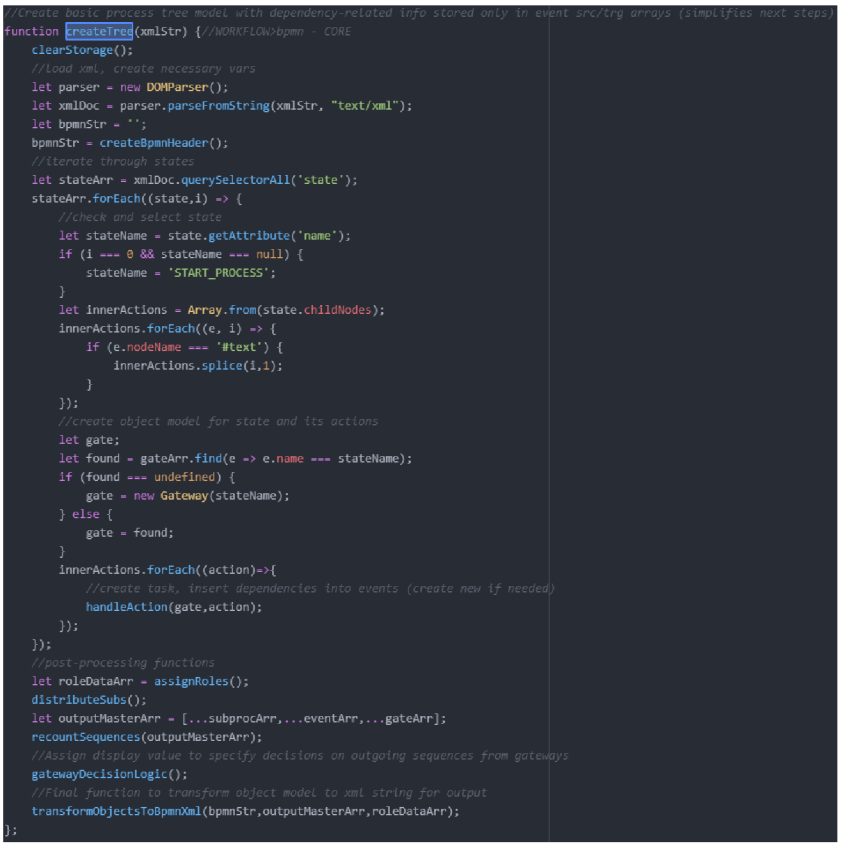
Algoritmus transformace je vizualizován v následujícím blokovém schématu:



Obrázek 19: Algoritmus transformace workflow (Zdroj: Vlastní zpracování)

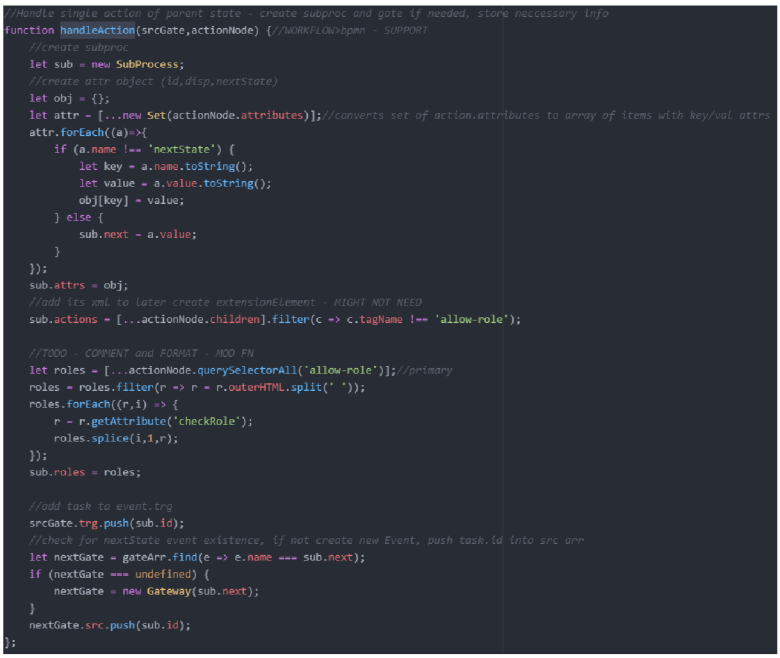
3.1.1.5 Hlavní funkce

Funkce createTree() již byla obecně popsána v předchozí podkapitole. Z tohoto důvodu nyní přeskočím její popis. Samotný kód funkce je na následujícím obrázku:



Obrázek 20: Funkce createTree() (Zdroj: Vlastní zpracování)

Funkce handleAction() je zodpovědná za vytvoření instancí podprocesů. Nejprve vytvoří sadu atributů z workflow akce. Ty se zároveň uloží do objektu vlastnosti attrs pro případ, že by bylo nutné se k nim znovu vracet. Poté vytvoří pole rolí, které mohou akci provádět. Dále pak uloží název podprocesu a předchozí a následující objekt.



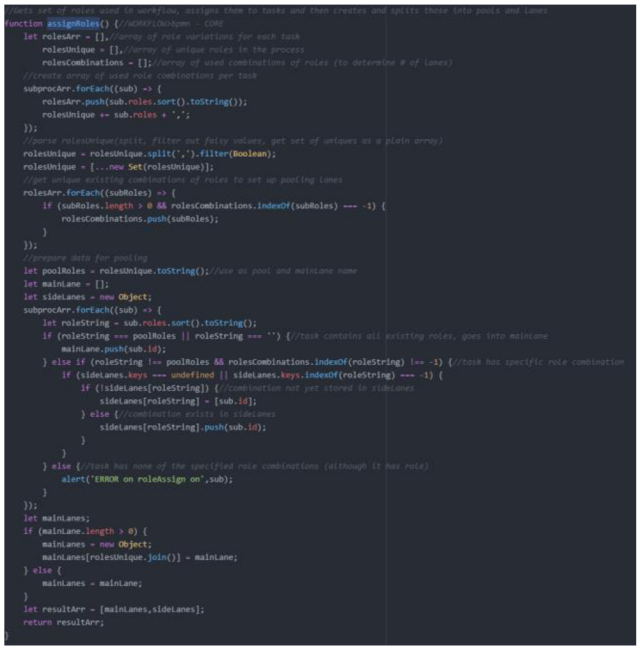
```
//Handle single action of parent state - create subproc and gate if needed, store neccessary info
function handleAction(srcGate,actionNode) {//WORKFLOW>bpmn - SUPPORT
    //create subproc
    let sub = new SubProcess;
    //create attr object (id,disp,nextState)
    let obj = {};
    let attr = [...new Set(actionNode.attributes)];//converts set of action.attributes to array of items with key/val attrs
    attr.forEach((a)=>{
        if (a.name !== 'nextState') {
            let key = a.name.toString();
            let value = a.value.toString();
            obj[key] = value;
        } else {
            sub.next = a.value;
        }
    });
    sub.attrs = obj;
    //add its xml to later create extensionElement - MIGHT NOT NEED
    sub.actions = [...actionNode.children].filter(c => c.tagName !== 'allow-role');

    //TODO - COMMENT and FORMAT - MOD FN
    let roles = [...actionNode.querySelectorAll('allow-role')];//primary
    roles = roles.filter(r => r = r.outerHTML.split(' '));
    roles.forEach((r,i) => {
        r = r.getAttribute('checkRole');
        roles.splice(i,1,r);
    });
    sub.roles = roles;

    //add task to event.trg
    srcGate.trg.push(sub.id);
    //check for nextState event existence, if not create new Event, push task.id into src arr
    let nextGate = gateArr.find(e => e.name === sub.next);
    if (nextGate === undefined) {
        nextGate = new Gateway(sub.next);
    }
    nextGate.src.push(sub.id);
};
```

Obrázek 21: Funkce handleAction() (Zdroj: Vlastní zpracování)

Funkce assignRoles() nejprve vytvoří dvě pole. První z nich obsahuje sadu unikátních rolí v rámci workflow, druhé pak všechny kombinace rolí, které se vyskytují v akcích workflow. Na základě těchto polí pak vytvoří pole pro oblasti a dráhy, které obsahují názvy figurujících rolí a reference na identifikátory podprocesů, které spadají do konkrétní dráhy.



```
//sets set of roles used in workflow, assigns them to tasks and then creates and splits those into pools and lanes
function assignRoles() {//WORKFLOW>bpmn - CORE
    let rolesArr = [],//array of role variations for each task
        rolesUnique = [],//array of unique roles in the process
        rolesCombinations = [];//array of used combinations of roles (to determine # of lanes)
    //create array of used role combinations per task
    subprocArr.forEach((sub) => {
        rolesArr.push(sub.roles.sort().toString());
        rolesUnique += sub.roles + ',';
    });
    //parse rolesUnique(split, filter out falsy values, get set of uniques as a plain array)
    rolesUnique = rolesUnique.split(',').filter(Boolean);
    rolesUnique = [...new Set(rolesUnique)];
    //get unique existing combinations of roles to set up pooling lanes
    rolesArr.forEach((subRoles) => {
        if (subRoles.length > 0 && rolesCombinations.indexOf(subRoles) === -1) {
            rolesCombinations.push(subRoles);
        }
    });
    //prepare data for pooling
    let poolRoles = rolesUnique.toString();//use as pool and mainLane name
    let mainLane = [];
    let sideLanes = new Object;
    subprocArr.forEach((sub) => {
        let roleString = sub.roles.sort().toString();
        if (roleString === poolRoles || roleString === '') {//task contains all existing roles, goes into mainLane
            mainLane.push(sub.id);
        } else if (roleString !== poolRoles && rolesCombinations.indexOf(roleString) !== -1) {//task has specific role combination
            if (sideLanes.keys === undefined || sideLanes.keys.indexOf(roleString) === -1) {
                if (!sideLanes[roleString]) {//combination not yet stored in sideLanes
                    sideLanes[roleString] = [sub.id];
                } else {//combination exists in sideLanes
                    sideLanes[roleString].push(sub.id);
                }
            }
        } else {//task has none of the specified role combinations (although it has role)
            alert('ERROR on roleAssign on',sub);
        }
    });
    let mainLanes;
    if (mainLane.length > 0) {
        mainLanes = new Object;
        mainLanes[rolesUnique.join()] = mainLane;
    } else {
        mainLanes = mainLane;
    }
    let resultArr = [mainLanes,sideLanes];
    return resultArr;
}
```

Obrázek 22: Funkce assignRoles() (Zdroj: Vlastní zpracování)

Funkce distributeSubs() pro každou instanci brány projde a doplní reference na předchozí a následující podprocesy. Pokud je brána na okraji procesu (buďto na začátku, nebo na konci), volá se funkce setBorderEvents(), která vytvoří počáteční či koncovou událost a pokud to situace dovoluje, transformuje na ni bránu.

```
//Goes through eventArr [src/trg], if more sources or targets exist, creates gateway and fills with srcs/trgts
function distributeSubs() {//WORKFLOW>bpmn - CORE
    gateArr.forEach((gate) => {
        //assign src and trg attrs to subprocesses
        if (gate.src.length > 0) {
            gate.src.forEach((sub) => {
                sub = subprocArr.filter(e => e.id === sub)[0];
                if (!sub) {
                    sub = eventArr.filter(e => e.id === sub)[0];
                }
                sub.trg = gate.id;
            });
        } if (gate.trg.length > 0) {
            gate.trg.forEach((sub) => {
                sub = subprocArr.filter(e => e.id === sub)[0];
                if (!sub) {
                    sub = eventArr.filter(e => e.id === sub)[0];
                }
                sub.src = gate.id;
            });
        }
        //set border events for process start and end
        if (gate.src.length === 0) {
            setBorderEvents(gate,'start');
        } else if (gate.trg.length === 0) {
            setBorderEvents(gate,'end');
        }
    });
    //remove gate and map first/last state to start/endEvent when gate acts only as passthrough
    gateArr.forEach((gate) => {
        if (gate.src.length === 1 && gate.trg.length === 1) {

            let firstEvent = [...taskArr,...subprocArr,...eventArr,...gateArr].find(e => e.id === gate.src[0]);
            let onlyAction = [...taskArr,...subprocArr,...eventArr,...gateArr].find(e => e.id === gate.trg[0]);
            firstEvent.trg = onlyAction.id;
            onlyAction.src = firstEvent.id;
            gateArr.splice(gateArr.indexOf(gate),1);
        }
    });
}
```

Obrázek 23: Funkce distributeSubs() (Zdroj: Vlastní zpracování)

Poslední hlavní funkce transformObjectsToBpmnXml() projde všechny instance objektů. V případě událostí uloží informaci o tom, jestli jde o počáteční, či koncovou událost. Pak pro všechny objekty vytvoří BPMN prvky voláním funkce createBpmnXmlElement(). Tyto prvky postupně přidává do řetězce BPMN procesu. Následně pomocnou funkcí createRoleXmlRefs() vytvoří BPMN referenci pro oblasti a dráhy a přidá ji do dokumentu. Poté přidá zápatí procesu a vygeneruje jeho dokumentový model. Nakonec volá funkci getDiagram(), pomocí niž přidá k BPMN procesu také jeho diagramovou část a výsledný BPMN model poté exportuje funkcí saveBpmnWorkflow().

Obrázek 24: Funkce transformObjectsToBpmnXml() (Zdroj: Vlastní zpracování)

3.1.1.6 Pomocné funkce

První pomocnou funkcí je createBpmnHeader(), která pouze vytvoří předdefinovanou hlavičku BPMN dokumentu.

Obrázek 25: Funkce createBpmnHeader() (Zdroj: Vlastní zpracování)

Funkce setBorderEvents() slouží k tvorbě instancí počátečních a koncových událostí. Jejich reference pak aktualizuje v příslušné instanci brány.

Obrázek 26: Funkce setBorderEvents() (Zdroj: Vlastní zpracování)

Funkce recountSequences() slouží k vytvoření výstupní a vstupní sekvence pro všechny uzlové objekty. Skládá se jak ze standardních procesů tvorby, tak i několika specializovaných, které je nutné řešit zvlášť.

```
//create incoming and outgoing sequences for each node connection
function recountSequences(elemsArray) {//WORKFLOW>bpmn - SUPPORT
    //create outgoing sequences
    elemsArray.forEach((elem)=>{
        if (elem.id.includes('gate') === true) {
            elem.trg.forEach((t)=>{
                let seq = new Sequence();
                seq.src = elem.id;
                seq.trg = t;
                elem.outSeq.push(seq.id);
            });
        } else if (elem.trg.length > 0) {
            let seq = new Sequence();
            seq.src = elem.id;
            seq.trg = elem.trg;
            elem.outSeq = seq.id;
        }
    });
    //create incoming sequences
    elemsArray.forEach((elem)=>{
        if (elem.id.includes('gate') === true) {
            elem.src.forEach((s)=>{
                elemsArray.forEach((e)=>{
                    if (s === e.id && typeof e.outSeq === 'object') {
                        sequenceArr.forEach((sequ)=>{
                            if (elem.id === sequ.trg && e.id === sequ.src && elem.inSeq.includes(sequ.id) === false) {
                                elem.inSeq.push(sequ.id);
                            }
                        });
                    } else if (s === e.id) {
                        elem.inSeq.push(e.outSeq);
                    }
                });
            });
        } else {
            elemsArray.forEach((e)=>{
                if (elem.end === true) {//different handling for endevents
                    sequenceArr.forEach((s)=>{
                        if (s.trg === elem.id && s.src === e.id) {
                            elem.inSeq = s.id;
                        }
                    });
                }
                if (elem.src === e.id && elem.src.includes('gate') === true) {//different handling for src=gate
                    sequenceArr.forEach((s)=>{
                        if (s.trg === elem.id && s.src === e.id) {
                            elem.inSeq = s.id;
                        }
                    });
                } else if (elem.src === e.id) {
                    elem.inSeq = e.outSeq;
                }
            });
        }
    });
}
```

Obrázek 27: Funkce recountSequences() (Zdroj: Vlastní zpracování)

Funkce gatewayDecisionLogic() přidává popisky k odchozím sekvencím z objektů bran.

```
//Label outgoing sequences from gateways
function gatewayDecisionLogic() {//WORKFLOW>bpmn - SUPPORT
    gateArr.forEach((gate) => {
        gate.outSeq.forEach((seqId) => {
            let seq = sequenceArr.find(e => e.id === seqId);
            let sub = subprocArr.find(e => e.id === seq.trg);
            if (sub !== undefined) {
                seq.display = 'goto: ' + sub.next;
            }
        });
    });
}
```

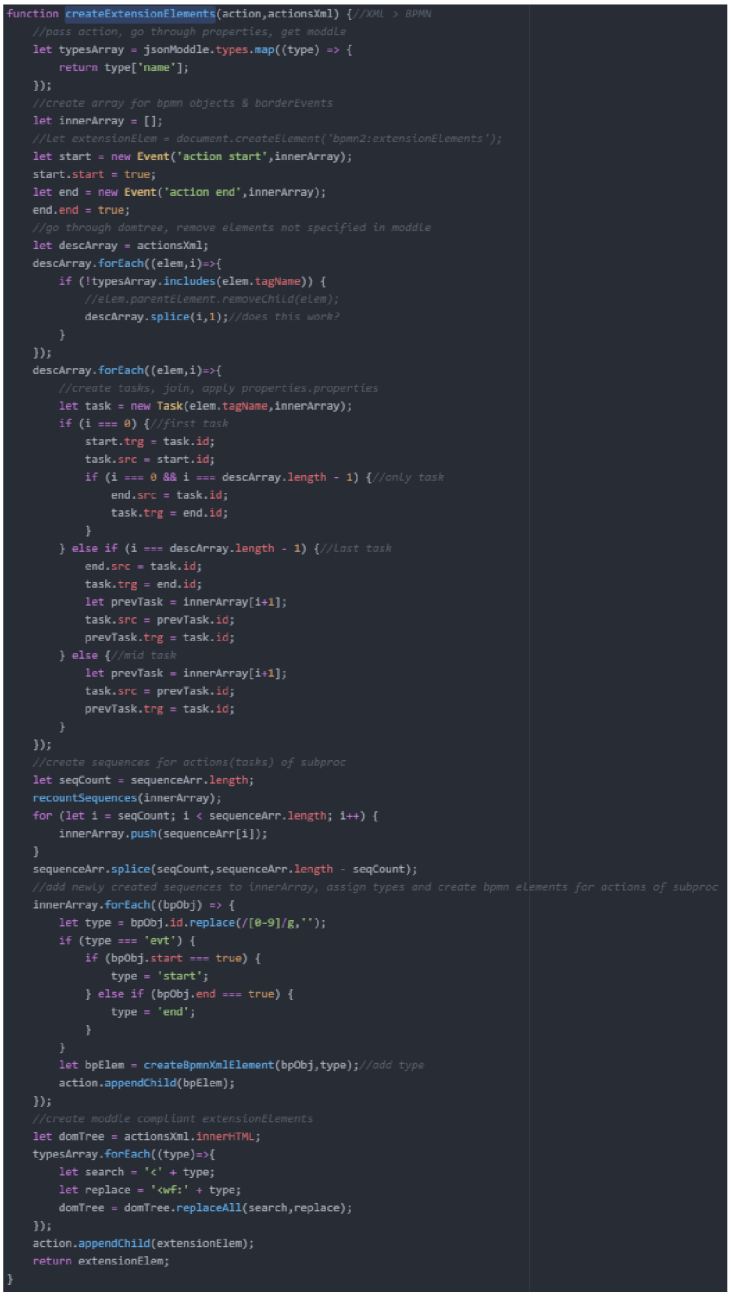
Obrázek 28: Funkce gatewayDecisionLogic() (Zdroj: Vlastní zpracování)

Funkce createBpmnXmlElement() na základě typu objektu vytvoří BPMN prvek, který pak naplní veškerými daty objektu. V případě, že se jedná o podproces, volá další podpůrnou funkci createExtensionElements(), která vytvoří jeho obsah.

```
//Returns model object converted to xml (if extension is needed in future, look to switch and if statements), arg:type = string
function createBpmnXmlElement(obj,type) {//WORKFLOW>bpmn - SUPPORT
    let tagname;
    switch (type)  {
        case 'start':tagname = 'bpmn2:startEvent';break;
        case 'end':tagname = 'bpmn2:endEvent';break;
        case 'evt':tagname = 'bpmn2:intermediateThrowEvent';break;
        case 'task':tagname = 'bpmn2:task';break;
        case 'sub':tagname = 'bpmn2:subProcess';break;
        case 'gate':tagname = 'bpmn2:exclusiveGateway';break;
        case 'sequence':tagname = 'bpmn2:sequenceFlow';break;
        default:console.log('ERR - passed unknown elementType',type,obj);return false;
    }
    //create xml element with id
    let elem = document.createElement(tagname);
    elem.setAttribute('id',obj.id);
    //set xml element attributes/children based on type of elem
    if (type === 'gate') {//gateway
        if (obj.name) {
            elem.setAttribute('name',obj.name);
        } else {
            elem.setAttribute('name',obj.id);
        }
        obj.inSeq.forEach((seq) => {
            let inc = document.createElement('bpmn2:incoming');
            inc.innerHTML = seq;
            elem.appendChild(inc);
        });
        obj.outSeq.forEach((seq) => {
            let out = document.createElement('bpmn2:outgoing');
            out.innerHTML = seq;
            elem.appendChild(out);
        });
    } else if (type === 'sequence') {//sequenceFlow
        elem.setAttribute('sourceRef',obj.src);
        elem.setAttribute('targetRef',obj.trg);
        if (obj.display) {
            elem.setAttribute('display',obj.display);
        }
    } else {//start/endEvent,intermediateEvent,task
        //naming
        if (obj.display) {//name for event
            if (obj.start === true && obj.display !== 'action start') {
                //decide on start event name depending on whether first gate will be remapped (state) or not (start)
                let evtTrg = gateArr.find(e => e.src.includes(obj.id) === true);
                if (evtTrg!== undefined && evtTrg.trg.length > 1) {
                    elem.setAttribute('name','PROCESS_START');
                } else {
                    elem.setAttribute('name',obj.display);
                }
            } else {
                elem.setAttribute('name',obj.display);
            }
        } else {//name for task or subproc
            if (type === 'task' || type === 'sub') {
                if (obj.attrs.display !== undefined) {
                    elem.setAttribute('name',obj.attrs.display);
                } else if (obj.attrs.id !== undefined) {
                    elem.setAttribute('name',obj.attrs.id);
                } else {
                    elem.setAttribute('name',obj.id);
                }
            }
        }
        //in/out
        if (obj.inSeq !== undefined && obj.inSeq.length > 0) {
            let inc = document.createElement('bpmn2:incoming');
            inc.innerHTML = obj.inSeq;
            elem.appendChild(inc);
        }
        if (obj.outSeq !== undefined && obj.outSeq.length > 0) {
            let out = document.createElement('bpmn2:outgoing');
            out.innerHTML = obj.outSeq;
            elem.appendChild(out);
        }
        //extensionElement
        if (obj.actions !== undefined) {
            let extension = createExtensionElements(elem,obj.actions);
            elem.appendChild(extension);
        }
    }
    return elem;
}
```

Obrázek 29: Funkce createBpmnXmlElement() (Zdroj: Vlastní zpracování)

Zmíněná funkce createExtensionElements() vytváří obsah podprocesů. Nejprve vytvoří počáteční a koncovou událost podprocesu, poté eliminuje veškeré sub-akce, které nejsou definované v JSON slovníku. Pro všechny ostatní vytvoří instance činností a naplní jejich rozšíření atributy sub-akce. Poté pomocí funkcí recountSequences() a createBpmnXmlElement() převede celý obsah podprocesu na validní BPMN.



```
function createExtensionElements(action,actionsXml) {//XML > BPMN
    //pass action, go through properties, get moddle
    let typesArray = jsonModdle.types.map((type) => {
        return type['name'];
    });
    //create array for bpmn objects & borderEvents
    let innerArray = [];
    //let extensionElem = document.createElement('bpmn2:extensionElements');
    let start = new Event('action start',innerArray);
    start.start = true;
    let end = new Event('action end',innerArray);
    end.end = true;
    //go through domtree, remove elements not specified in moddle
    let descArray = actionsXml;
    descArray.forEach((elem,i)=>{
        if (!typesArray.includes(elem.tagName)) {
            //elem.parentElement.removeChild(elem);
            descArray.splice(i,1);//does this work?
        }
    });
    descArray.forEach((elem,i)=>{
        //create tasks, join, apply properties.properties
        let task = new Task(elem.tagName,innerArray);
        if (i === 0) {//first task
            start.trg = task.id;
            task.src = start.id;
            if (i === 0 && i === descArray.length - 1) {//only task
                end.src = task.id;
                task.trg = end.id;
            }
        } else if (i === descArray.length - 1) {//last task
            end.src = task.id;
            task.trg = end.id;
            let prevTask = innerArray[i+1];
            task.src = prevTask.id;
            prevTask.trg = task.id;
        } else {//mid task
            let prevTask = innerArray[i+1];
            task.src = prevTask.id;
            prevTask.trg = task.id;
        }
    });
    //create sequences for actions(tasks) of subproc
    let seqCount = sequenceArr.length;
    recountSequences(innerArray);
    for (let i = seqCount; i < sequenceArr.length; i++) {
        innerArray.push(sequenceArr[i]);
    }
    sequenceArr.splice(seqCount,sequenceArr.length - seqCount);
    //add newly created sequences to innerArray, assign types and create bpmn elements for actions of subproc
    innerArray.forEach((bpObj) => {
        let type = bpObj.id.replace(/[0-9]/g,'');
        if (type === 'evt') {
            if (bpObj.start === true) {
                type = 'start';
            } else if (bpObj.end === true) {
                type = 'end';
            }
        }
        let bpElem = createBpmnXmlElement(bpObj,type);//add type
        action.appendChild(bpElem);
    });
    //create moddle compliant extensionElements
    let domTree = actionsXml.innerHTML;
    typesArray.forEach((type)=>{
        let search = '<' + type;
        let replace = '<wf:' + type;
        domTree = domTree.replaceAll(search,replace);
    });
    action.appendChild(extensionElem);
    return extensionElem;
}
```

Obrázek 30: Funkce createExtensionElements() (Zdroj: Vlastní zpracování)

Funkce createRoleXmlRefs() vytvoří dle vygenerovaných polí konkrétní BPMN prvky oblastí a dráh a naplní je validními prvky, jenž referencují podprocesy, které konkrétní role obsahují.

```
function createRoleXmlRefs(roleData) {//WORKFLOW>bpmn - SUPPORT
    //add collab elem
    let refOutputStr;
    let mainLane = {...roleData[0]};
    let sideLanes = {...roleData[1]};
    let rolesElem = document.createElement('bpmn2:roles');
    mainLane = Object.keys(mainLane).map((k) => mainLane[k]);
    if (Object.values(mainLane).length > 0) {
        let mainset = document.createElement('bpmn2:laneSet');
        let lane = document.createElement('bpmn2:lane');
        lane.id = 'Lane_num-1';
        lane.setAttribute('name',Object.keys(roleData[0])[0]);
        Object.values(mainLane).forEach((refArr)=>{
            refArr.forEach((ref)=>{
                let nodeRef = document.createElement('bpmn2:flowNodeRef');
                nodeRef.innerHTML = ref;
                lane.appendChild(nodeRef);
            });
        });
        mainset.appendChild(lane);
        rolesElem.appendChild(mainset);
    }
    let refArr = Object.keys(sideLanes).map((k) => sideLanes[k]);
    if (Object.keys(sideLanes).length > 0) {
        let laneset;
        if (Object.values(mainLane).length > 0) {
            laneset = document.createElement('bpmn2:childLaneSet');
        } else {
            laneset = document.createElement('bpmn2:laneSet');
        }
        Object.keys(sideLanes).forEach((laneName,i) => {
            let lane = document.createElement('bpmn2:lane');
            lane.id = 'Lane_num' + i;
            lane.setAttribute('name',laneName);
            refArr[i].forEach((item) => {
                let nodeRef = document.createElement('bpmn2:flowNodeRef');
                nodeRef.innerHTML = item;
                lane.appendChild(nodeRef);
            });
            laneset.appendChild(lane);
            //add participant elems
        });
        if (Object.values(mainLane).length > 0) {
            rolesElem.querySelector('#Lane_num-1').appendChild(laneset);
            refOutputStr = rolesElem.innerHTML;
        } else {
            refOutputStr = laneset.outerHTML;
        }
    } else {
        refOutputStr = rolesElem.innerHTML;
    }
    return refOutputStr;
}
```

Obrázek 31: Funkce createRoleXmlRefs() (Zdroj: Vlastní zpracování)

Funkce getDiagram() vygeneruje z finálního bpmn procesu pomocí knihovny bpmn-auto-layout diagramová data modelu.

```
744  function getDiagram(bpmnProc) {
745      const AutoLayout = require('bpmn-auto-layout');
746      let autoLayout = new AutoLayout();
747      (async () => {
748          let bpmnComplete = await autoLayout.layoutProcess(bpmnProc);
749          return bpmnComplete;
750      })();
751  }
```

Obrázek 32: Funkce getDiagram (Zdroj: Vlastní zpracování podle [23])

Poslední podpůrná funkce saveBpmnWorkflow() exportuje finální BPMN model do souboru.

```
//Forces download of bpmn xml file
async function saveBpmnWorkflow(xmlstring) {//WORKFLOW>bpmn - SUPPORT
    const result = await exportDiagram();
    let stealthLink = document.createElement('a');
    let linkContainer = document.getElementById('btnContainer');
    linkContainer.appendChild(stealthLink);
    let bob = new Blob([xmlstring], {type: 'text/plain'});
    stealthLink.setAttribute('href', window.URL.createObjectURL(bob));
    stealthLink.setAttribute('download', 'bpmnWorkflow.xml');
    stealthLink.dataset.downloadurl = ['text/plain', stealthLink.download, stealthLink.href].join(':');
    stealthLink.click();
    linkContainer.removeChild(stealthLink);
}
```

Obrázek 33: Funkce saveBpmnWorkflow() (Zdroj: Vlastní zpracování)

3.1.2 Transformace BPMN modelu na GPC workflow

V této kapitole popíšu proces transformace BPMN modelu na GPC workflow. Tento proces je reverzní vůči tomu, který byl popsaný v předchozí kapitole. Vzhledem k jeho odlišnosti k němu musím přistupovat odlišně. Pokusím se však znovu použít co nejvíce praktik, konstruktorů či funkcí z předchozí kapitoly, aby byl výsledný transformační modul co nejlépe optimalizovaný a proces fungoval jednoduše a funkčně. Kapitola neobsahuje podkapitolu o mapování prvků, jelikož to musí být jednotné napříč oběma transformacemi, abych docílil výsledku 1:1. Transformace totiž musí být opakovatelné tak, že výsledek bude vždy odpovídat prvotnímu zdroji.

Po zpracování abstraktního popisu transformace vytvořím základní logické schéma algoritmu. Po jeho zpracování bych měl být schopen definovat, zda bude potřeba nějak upravit konstruktory objektů. Je však možné, že některé úpravy budou odhaleny až během samotné implementace konkrétního algoritmu. Ten bude popsán v navazující podkapitole. Zbylá část kapitoly bude opět věnována popisu hlavních a pomocných funkcí, které budou součástí algoritmu.

V rámci transformace bude potřeba nejprve zpracovat všechny prvky BPMN modelu. To provedu pomocí převedení na již definované objekty instancí tříd. Znovu tak využiji konstruktory, které byly definovány v předchozí kapitole. Je však možné, že některé z nich bude nutné funkčně rozšířit.

Po vytvoření potřebných objektů a zpracování jejich návazností budu moci generovat jednotlivé prvky workflow. Obdobně jako při předchozí transformaci začnu s tvorbou stavů z instancí bran, které naplním obecnými akcemi. Ty pak rozšířím do plnohodnotného stavu včetně rolí a sub-akcí.

Nakonec vytvořím konečný objektový dokument, který bude možné dále exportovat do editoru workflow platformy GPC, která bude schopná pomocí konkrétního workflow vytvořit konfiguraci pro konkrétní kategorii tiketů.

Jelikož se tato transformace týká pouze samotného procesu, nikoli jeho vizualizace, není třeba generovat žádný diagram a nebudu tedy využívat žádné další knihovny, jako tomu bylo v předchozí kapitole.

3.1.2.1 Základní logické schéma

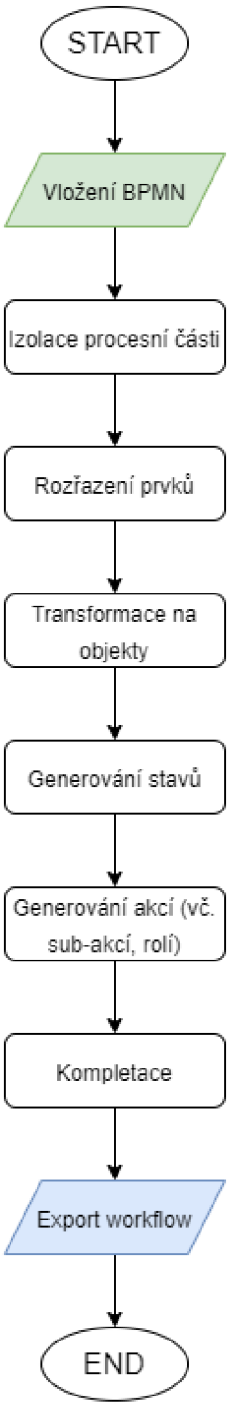
Vzhledem k předchozímu úspěchu se opět pokusím abstraktní algoritmus transformace BPMN modelu na workflow připodobnit k ETL procesu. Tentokrát půjde o extrakci prvků z BPMN modelu, jejich transformaci na sjednocený druh objektů. Ty dále převedu na objekty reprezentující stavy a akce workflow. Výsledkem transformace budou již validní prvky workflow, které budou sjednoceny do dokumentu, jež bude možné nahrát pomocí editoru workflow do platformy GPC, která pomocí jej nakonfiguruje konkrétní proces (například monitoring incidentů a událostí).

V první fázi bude do algoritmu vstupovat BPMN model. Ten izoluji pouze na jeho procesní část, v rámci niž rozřadím prvky do struktur dle jejich druhu. Podle společných konstruktorů z těchto druhů prvků vytvořím standardní instance objektů, které již budu dále transformovat.

V druhé fázi pro každou rozhodovací bránu vytvořím workflow prvek stavu. Z podprocesů, jež budou vycházet této brány, vytvořím prvky akcí workflow. Po vyplnění jejich atributů akce naplním dalšími daty, jež budou reflektovat role a obsah akcí (zmíněné sub-akce).

Ve třetí fázi zkompletuji všechny stavy a jejich akce do jednoho objektového modelu. Tomu případně vygeneruji záhlaví či zápatí. Výsledné workflow pak exportuji buďto jako textový řetězec přímo do editoru workflow, nebo uložím jako samostatný soubor na zařízení uživatele.

Logické schéma transformace je vizualizováno v následujícím vývojovém diagramu:



Obrázek 34: Logické schéma transformace BPMN (Zdroj: Vlastní zpracování)

3.1.2.2 Úpravy konstruktorů

Ze základního logického schématu jsem nezjistil žádné potřeby úpravy konstruktorů. Při zpracovávání implementace jsem však narazil na problémy s hledáním BPMN prvků obsažených v podprocesu.

Z tohoto důvodu jsem upravil konstruktory instancí událostí, činností a sekvencí tak, aby obsahovaly volitelný parametr optionalArray. Pokud jsou objekty vytvořeny s tímto parametrem, prvky se neukládají do globálních polí dle zvyklosti, ale jsou uloženy v dočasných polích, které jsou poskytnuty v tomto parametru. Tímto krokem jsem eliminoval problémy, které vznikaly při zpracovávání událostí a návazností v podprocesech, kdy se prvky duplikovaly i do globálních polí a výsledná transformace trpěla nedostatky v návaznosti prvků. Tento parametr byl zpětně využitý i v transformaci z předchozí kapitoly, jelikož jsem tímto krokem nejen zjednodušil část algoritmu, ale také dosáhl lepší univerzality a čitelnosti kódu.

3.1.2.3 Algoritmus transformace

Nyní se již budu věnovat konkrétní implementaci navrhovaného řešení k transformaci BPMN modelu na GPC workflow.

Po vložení textového řetězce BPMN modelu je proveden převod na objektový model. Tento model je vytvořen pouze z procesní části modelu. Pomocí funkce createWorkflowHeader() je vytvořena hlavička budoucího workflow dokumentu ve formě textového řetězce. Podpůrná funkce createBpmnObjects() poté vytvoří pole dle typů BPMN prvků a naplní je z poskytnutého objektového modelu. Pro každou položku z těchto polí je pak volána funkce createSingleObject(), která slouží k vytvoření instancí objektů pomocí konstruktorů popsaných v předchozí kapitole.

Zmíněná funkce podle poskytnutého typu použije vhodný konstruktor. Ty vytvoří instance a funkce je pak dodatečně doplní daty ze vstupních objektů. V případě podprocesů vytvoří také objektové modely obsahu konkrétních akcí, které připojí k instanci podprocesů, jež akci reprezentují.

Jelikož vytvořené objekty obsahují pouze reference příchozích a odchozích sekvencí, algoritmus využije funkce setSourcesTargets(), která pomocí referencí projde existujícící globální pole objektů a doplní předchozí a následující objekty. V případě

podprocesu také doplní data o stavu, do kterého má konkrétní podproces přejít po svém dokončení.

Následně volám funkci distributeRoles(), která identifikuje existující oblasti a dráhy a přiřadí instancím podprocesů korektní role. Ty jsou nyní již jednotlivě rozdělené, jelikož jsou ve workflow akcích definovány samostatně.

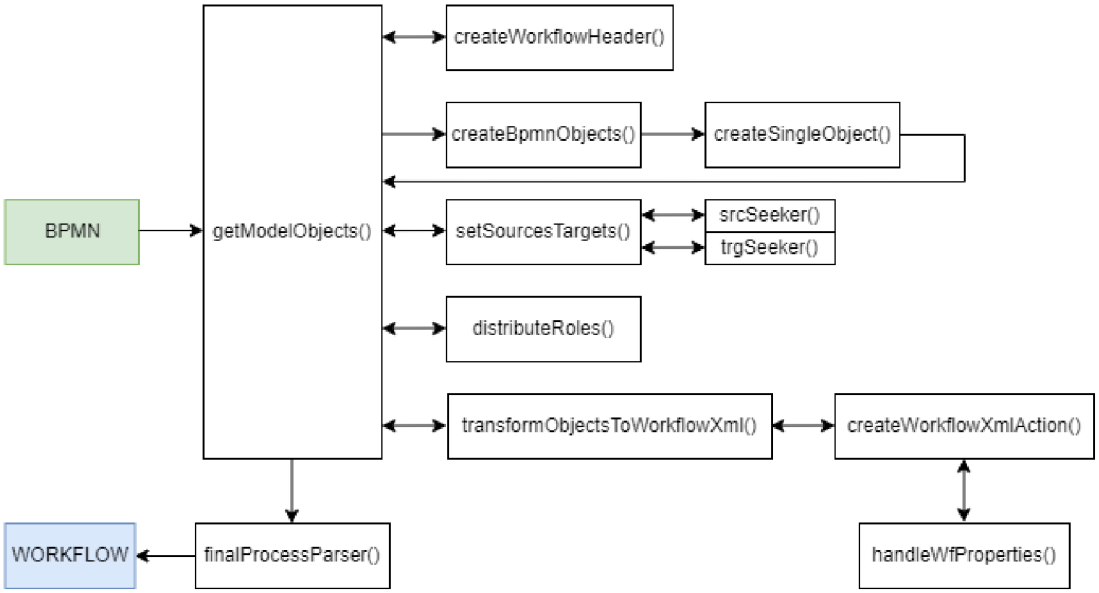
Poté lze již zavolat hlavní funkci transformObjectsToWorkflowXml() pro transformaci instancí tříd na prvky GPC workflow. Tato funkce nejprve vytvoří objektový model workflow. Poté vytvoří pro každou instanci brány prvek stavu, pro který transformuje všechny výstupní podprocesy na akce. Tuto transformaci zprostředkovává funkce createWorkflowXmlAction().

Ta je využita k vytvořením workflow prvků akcí, v jejichž rámci jsou vyplněny data o akci a příslušných rolích. Pro správné vyplnění sub-akcí využívám podpůrné funkce handleWfProperties(). Funkce createWorkflowXmlAction() je poté volána ještě jednou pro podprocesy, které nebyly jednoduše identifikovány jakožto příslušící danému stavu.

Výsledný objektový model workflow je převeden do textové podoby, ve které je k němu připojené záhlaví. Kvůli nedostatkům metod, které v Javascriptu slouží k převodu mezi textem a objektovým modelem jsem byl povinen vytvořit funkci finalProcessParser(), která pomocí doplňujícího JSON slovníku nahrazuje atributy sub-akcí, které byly převedeny na malá písmena.

V této fázi algoritmus poskytuje již kompletní a validní GPC workflow, které může být vloženo do editoru workflow a použito pro konfiguraci platformy.

Algoritmus transformace je vizualizován v následujícím blokovém schématu:



Obrázek 35: Algoritmus transformace BPMN (Zdroj: Vlastní zpracování)

3.1.2.4 Hlavní funkce

Funkce getModelObjects() jsem popsal v rámci předchozí podkapitoly. Jedná se o hlavní funkci transformace BPMN modelu na workflow. Po importu volá funkce createBpmnObjects() a setSourcesTargets(), poté funkci distributeRoles(), která má odlišené parametry podle složení rolí v modelu. Následně uloží do proměnné výstup funkce transformObjectsToWorkflowXml(), která vytváří celkový objektový model workflow. Ten je převeden na textový řetězec, doplněn zápatím a konečně poskytnutý funkci finalProcessParser(), která zajistí zpracování workflow do textové podoby, která je validní a lze použít v editoru workflow.

```
//Main core function for workflow transformation
function getModelObjects(bpmnStr) {//BPMN>wf - CORE
    clearStorage();
    let parser = new DOMParser();
    let entireXml = parser.parseFromString(bpmnStr, "text/xml");
    let bpmnDoc = entireXml.querySelector('process');
    let workflowStr = '';
    workflowStr = createWorkflowHeader();
    lowCaseTags = false;
    //create bpmn object structure
    createBpmnObjects(bpmnDoc);
    //complete task dependencies
    setSourcesTargets(subprocArr);
    //add action role info
    if (entireXml.querySelector('participant') !== null) {
        let allRoles = entireXml.querySelector('participant').getAttribute('name');
        let laneSet = bpmnDoc.querySelectorAll('laneSet lane');
        if (!allRoles) {
            alert('pool is missing role name(s)!');
        }
        //only pool is defined
        if (laneSet.length === 0) {
            laneSet = [entireXml.querySelector('participant')];
            distributeRoles(allRoles,laneSet,bpmnDoc);
        } else {
            //pool has defined lanes with different role sets
            distributeRoles(allRoles,laneSet);
        }
    }
    //create xmlStructure
    let workflow = transformObjectsToWorkflowXml();
    console.log('FINAL WORKFLOW',workflow);
    workflowStr += workflow.outerHTML + '</xml>';
    //output
    //Final parse of attributes
    finalProcessParser(workflowStr);
}
```

Obrázek 36: Funkce getModelObjects() (Zdroj: Vlastní zpracování)

Funkce createSingleObject() slouží k převedení BPMN prvků na instance tříd definovaných konstruktory. Při jejím volání je poskytnutý parametr type, podle kterého funkce určuje konkrétní třídu k transformaci. Pokud se jedná o podproces, funkce ho převede na objektový model propTree, který udržuje pořadí návaznosti jednotlivých činností (budoucích sub-akcí). Funkce závěrem vrací vytvořený objekt.

Obrázek 37: Funkce createSingleObject() (Zdroj: Vlastní zpracování)

Funkci setSourcesTargets() využívám k doplnění chybějících dat u instancí podprocesů. Tato funkce nalezne pomocí funkcí srcSeeker() a trgSeeker() předcházející a následující prvky podprocesu, díky čemuž lze doplnit jak tato chybějící data, tak i následující stav pro budoucí transformaci.

```
//Helper function to add src,trg attributes to tasks and subprocesses via separate functions
function setSourcesTargets(elems) {//BPMN>wf - SUPPORT
    elems.forEach((item) => {
        let previous = srcSeeker(item);
        let next = trgSeeker(item);
        item.src = previous.id;
        item.trg = next.id;
        if (item instanceof SubProcess) {
            item.next = next.name;
        }
    });
}
```

Obrázek 38: Funkce setSourcesTargets() (Zdroj: Vlastní zpracování)

Funkce distributeRoles() nalezne všechny dráhy BPMN modelu. V každé z nich pak podle referencí vyhledá instance podprocesů a přiřadí jim nalezené role.

```
//Distributes roles among pools, lanes and elements
function distributeRoles(allRoles,laneSet,process) {//BPMN>wf - SUPPORT
    let hasLanes = true;
    if (process) {
        hasLanes = false;
    }
    allRoles = allRoles.replace(/ /g,'');
    laneSet.forEach((lane) => {
        let nodes = [...lane.children];
        laneRoles = lane.getAttribute('name');
        if (laneRoles === null) {
            laneRoles = allRoles;
        }
        laneRoles = laneRoles.replace(/ /g,'').split(',');
        if (hasLanes === false) {
            nodes = [...process.children];
        }
        nodes.forEach((node) => {
            let nodeId;
            if (hasLanes === true) {
                nodeId = node.innerHTML;
            } else {
                nodeId = node.id;
            }
            let elem = [...taskArr,...subprocArr,...eventArr,...gateArr].find(e => e.id === nodeId);
            if (elem !== undefined) {
                elem.roles = laneRoles;
            } else {
                console.log('ERR ROLES',node);
            }
        });
    });
}
```

Obrázek 39: Funkce distributeRoles() (Zdroj: Vlastní zpracování)

Poslední hlavní funkce transformObjectsToWorkflowXml() nejprve vytvoří dokumentový model workflow. Následovně projde všechny instance bran a transformuje je na workflow prvky stavů. V každé takové bráně pak všechny navazující podprocesy pomocí funkce createWorkflowXmlAction() transformuje na akce, které přiřadí stavu. Stav poté vloží do objektového modelu workflow.

V této fázi se mohou vyskytovat některé nepřiřazené instance podprocesů (důsledkem napojení na události namísto bran). Pro tento případ provádím korekci jejich zdrojů, po kterém mohu opět provést proces transformace na akce a přiřazení stavům v objektovém modelu. Funkce poté navrací výsledný objektový model workflow.

```
//Transforms finished object model from bpmn to workflow xml structure
function transformObjectsToWorkflowXml() {// BPMN>wf - CORE
    let workflow = document.createElement('workflow');
    workflow.setAttribute('xmlns','http://gpc.expect-it.cz/gpcworkflow.xsd');
    //fill with tasks
    let parsedSubs = [...subprocArr];
    eventArr.forEach((evtState) => {
        if (evtState.start === true && trgSeeker(evtState) instanceof SubProcess) {
            let stateElem = document.createElement('state');
            if (evtState.display) {
                stateElem.setAttribute('name',evtState.display);
            }
            let action = createWorkflowXmlAction(trgSeeker(evtState));
            stateElem.appendChild(action);
            workflow.appendChild(stateElem);
        }
    });
    gateArr.forEach((state) => {
        let stateElem = document.createElement('state');
        if (state.name) {
            stateElem.setAttribute('name',state.name);
        }
        let childSubs = subprocArr.filter(e => e.src === state.id);
        childSubs.forEach((sub) => {
            let action = createWorkflowXmlAction(sub);
            stateElem.appendChild(action);
            //remove from parsedSubs
            parsedSubs.forEach((e, i) => {
                if (e === sub) {
                    parsedSubs.splice(i,1);
                }
            });
        });
        if (stateElem.children.length > 0) {//states without own actions are only referenced by action.next
            workflow.appendChild(stateElem);
        }
    });
    if (parsedSubs.length > 0) {
        parsedSubs.forEach((sub) => {
            let src = [...eventArr,...gateArr].find(e => e.id === sub.src);
            if (src instanceof Gateway === true) {
                let srcState = srcSeeker(sub);
                let stateElem = workflow.querySelectorAll('state').find(e => e.getAttribute('id') === srcState.id);
                let action = createWorkflowXmlAction(sub);
                stateElem.appendChild(action);
                //remove from parsedTasks
                parsedSubs.forEach((e, i) => {
                    if (e === sub) {
                        parsedSubs.splice(i,1);
                    }
                });
            }
        });
    }
    return workflow;
}
```

Obrázek 40: Funkce transformObjectsToWorkflowXml() (Zdroj: Vlastní zpracování)

3.1.2.5 Pomocné funkce

První pomocná funkce createWorkflowHeader() obdobně jako v předchozí kapitole slouží pouze k vytvoření předdefinované hlavičky dokumentu.

```
//Returns workflow template string header
function createWorkflowHeader() {//BPMN>wf - SUPPORT
    let header = '<?xml version="1.0" encoding="UTF-8"?>';
    return header;
}
```

Obrázek 41: Funkce createWorkflowHeader() (Zdroj: vlastní zpracování)

Funkce createBpmnObjects() rozděluje BPMN model na jednotlivé typy prvků, kterými naplňuje několik polí. Obsahuje také ošetření v případě, že byl k vytvoření modelu použitý jiný editor BPMN diagramů, který pojmenovává prvky malými písmeny. Pro každou položku v každém poli poté dle jejího přiděleného typu volá již zmíněnou hlavní funkci createSingleObject().

```
//Get bpmn xml elements and convert them to tree model objects
function createBpmnObjects(xml) {//BPMN>wf - SUPPORT
    let eventsArray = [...xml.querySelectorAll('process > startEvent,process > endEvent,process > intermediateThrowEvent')],
        tasksArray = [...xml.querySelectorAll('subProcess > task')],
        subprocArray = [...xml.querySelectorAll('subProcess')],
        gatewaysArray = [...xml.querySelectorAll('exclusiveGateway')],
        sequencesArray = [...xml.querySelectorAll('sequenceFlow')];//faster and simpler than nodelist.toArray
    //support case insensivity
    if (eventsArray.length === 0) {
        lowCaseTags = true;
        eventsArray = [...xml.querySelectorAll('startevent,endevent,intermediatethrowevent')],
        subprocArray = [...xml.querySelectorAll('subprocess')],
        gatewaysArray = [...xml.querySelectorAll('exclusivegateway')],
        sequencesArray = [...xml.querySelectorAll('sequenceflow')];
    }
    eventsArray.forEach((event) => {createSingleObject(event,'event');});
    tasksArray.forEach((task) => {createSingleObject(task,'task');});
    subprocArray.forEach((subproc) => {createSingleObject(subproc,'subproc');});
    gatewaysArray.forEach((gate) => {createSingleObject(gate,'gate');});
    sequencesArray.forEach((sequence) => {createSingleObject(sequence,'sequence');});
}
```

Obrázek 42: Funkce createBpmnObjects() (Zdroj: Vlastní zpracování)

Funkce srcSeeker() a trgSeeker() uvádím společně, jelikož jejich funckionalita, ačkoli opačná, funguje na obdobném principu. Funkce se volají s parametrem obsahujícím objekt libovolného uzlového prvku BPMN modelu. Pomocí jeho dat o spojnicích provedou vyhledávání v polích dle konkrétní reference a vrací buďto předchozí, nebo následující instanci uzlového objektu parametru.

```
//Checks task precursors to find source event(state), goes through gateways, stops at first match
function srcSeeker(node) {//BPMN>wf - SUPPORT
    let prevSeq = sequenceArr.find(e => e.id === node.inSeq);
    //validation for existing start event
    if (!prevSeq) {
        alert('ERROR - Diagram is missing a start event.');
    }
    let prevNode = prevSeq.src;
    prevNode = [...taskArr,...subprocArr,...eventArr,...gateArr].find(e => e.id === prevNode);
    return prevNode;
}

//Checks task successors to find target event(state), goes through gateways, stops at first match
function trgSeeker(node) {//BPMN>wf - SUPPORT
    let nextSeq = sequenceArr.find(e => e.id === node.outSeq);
    //validation for existing end event
    if (!nextSeq) {
        alert('ERROR - Diagram is missing an end event.');
    }
    let nextNode = nextSeq.trg;
    nextNode = [...taskArr,...subprocArr,...eventArr,...gateArr].find(e => e.id === nextNode);
    return nextNode;
}
```

Obrázek 43: Funkce srcSeeker() a trgSeeker() (Zdroj: Vlastní zpracování)

Funkce createWorkflowXmlAction() vytvoří z instance podprocesu akci workflow. Této akci vyplní veškerá potřebná data a pomocí funkce handleWfProperties() převede sub-akce do formátu, který je validní v rámci GPC workflow.

```
//Creates xml action from task
function createWorkflowXmlAction(sub) {//BPMN>wf - SUPPORT
    let action = document.createElement('action');
    action.setAttribute('id',sub.id);
    action.setAttribute('display',sub.name);
    //check for gateway (multiple tasks -> state)
    let trg = [...eventArr,...subprocArr,...gateArr,...taskArr].find(e => e.id === sub.src);
    //set nextstate via param, otherwise via trg endEvent
    if (sub.next !== undefined) {
        action.setAttribute('nextState',sub.next);
    } else {
        action.setAttribute('nextState',trgSeeker(sub).display);
    }
    if (sub.roles !== undefined) {
        sub.roles.forEach((role) => {
            let roleElem = document.createElement('allow-role');
            roleElem.setAttribute('checkRole',role);
            action.appendChild(roleElem);
        });
    }
    //++++added workflow extension support++++//$ changes due to switch to subproc
    if (sub.actions) {
        let propertiesXml = handleWfProperties(sub.attrs.wfProperties);
        action.innerHTML += propertiesXml;
    }
    //++++ clean propId and parentId ++++
    let propArr = action.querySelectorAll('*');
    propArr.forEach((prop) => {
        prop.removeAttribute('propId');
        prop.removeAttribute('parentId');
    });
    //out
    return action;
}
```

Obrázek 44: Funkce createWorkflowXmlAction() (Zdroj: Vlastní zpracování)

3.2 Uživatelské rozhraní

V této kapitole zpracuji návrh a implementaci změn a úprav, které se budou týkat uživatelského rozhraní platformy. Tyto úpravy budou sloužit k tomu, aby uživatelům umožnily používání nástroje pro transformaci workflow a BPMN modelů.

Změny jsou rozděleny do dvou skupin. První skupina se týká rozšíření BPMN editoru. Tato rozšíření budou sloužit k tomu, aby mohl uživatel připravit BPMN model procesu tak, že jej bude možné transformovat na GPC workflow. Druhá skupina změn se zaměřuje na rozšíření současného editoru workflow. Tyto změny jsou jednodušší a budou sloužit k tomu, aby se oba transformační procesy odehrávaly v jednotném prostředí a byly tak uživatelsky přívětivé.

3.2.1 Rozšíření BPMN editoru

V první části kapitoly se budu věnovat BPMN editoru. Ten jsem navrhl a implementoval pro potřeby uživatelů již v rámci své bakalářské práce. V předchozích kapitolách jsem navrhl a implementoval logiku, která umožňuje samotnou transformaci. V rámci nich jsem také zmínil některé činnosti, které bude nutno zprostředkovat pomocí uživatelského rozhraní.

V rámci BPMN editoru musí být uživatel schopný importovat, upravovat a exportovat BPMN modely. Tato funkcionalita je již zpracována. Nové změny se týkají především rozšíření atributů. Ty je nutné využít v rámci činností jednotlivých podprocesů, které vizuálně reprezentují sub-akce původních workflow. Pro transformaci z BPMN modelu na workflow je nutné, aby měl uživatel možnost úpravy těchto činností nad rámec základní implementace standardu BPMN 2.0. V rámci tohoto standardu je definována část s rozšířením pro libovolné prvky diagramu, nazvaná Extension elements.

Ty slouží k připojení libovolných datových objektů k prvkům diagramu, které jsou pak v BPMN definovány v rámci vlastních XML tagů. Ačkoli lze tyto objekty přidávat čistě programaticky, uživatelé budou vyžadovat UI nástroj, který jim umožní tyto atributy přidávat, zobrazovat či upravovat na každém vybraném prvku činnosti. Nástroj by jim měl nabídnout konkrétní sadu předdefinovaných GPC sub-akcí, které je možné v rámci akcí aplikovat. Pro tyto potřeby navrhnu a implementuji tzv. slovník ve formátu JSON, který bude obsahovat vybrané sub-akce, jejich parametry a povolené datové typy hodnot těchto parametrů.

3.2.1.1 Externí selekce prvků

První rozšiřující funkcionalita BPMN editoru je možnost externí selekce prvků. V současném stavu lze prvky vybírat tradičním způsobem, který vytváří výchozí dialogové okno pro navazování dalších prvků a spojnic či modifikaci typu prvku. Pro mé účely je nutné navrhnout a implementovat funkcionalitu, pomocí niž bude možné zobrazit a modifikovat rozšířená data, jež využiji v případě činností k definování sub-akcí.

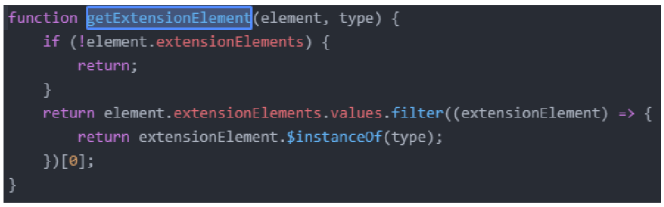
Výsledná funkcionalita by tak měla sestávat z dvou hlavních částí. První část se bude týkat samotné uživatelské interakce. Navrhnu tedy asynchronní rutinu, která bude čekat na specifickou interakci uživatele a na základě ni umožní přístup k požadovanému

prvku a jeho vlastnostem. Druhá část této funkcionality bude mít za úkol zobrazovat a zpracovávat konkrétní data prvku.

První část funkcionality tvoří skript, který vymezuje sledované interakce s editorem. Pro mé účely stačí pouze událost kliku myší. Pokud tato událost nastane na požadovaném prvku (BPMN činnost), skript zavolá funkce, které získají data sub-akcí z prvku a vypíší je do rozšiřovacího panelu v uživatelském rozhraní.

Obrázek 45: Skript pro vyhodnocování interakcí s prvky editoru (Zdroj: Vlastní zpracování)

Druhá část funkcionality sestává primárně ze dvou funkcí. První z nich je funkce getExtensionElement(). Ta nám vrací po selekci činnosti v BPMN diagramu obsah rozšíření extension element, konkrétně typ definovaný v rámci slovníku sub-akcí.



Obrázek 46: Funkce getExtensionElement() (Zdroj: Vlastní zpracování)

Druhou funkcí je taskModal(). Ta slouží k provázání činnosti a možný sub-akcí tak, aby uživatel mohl zvolit požadovanou sub-akci pro vybraný prvek činnosti.

```
//Fills interface forms with data from wf moddle extension on task select
function taskModal(task) {
    let taskObject = task.businessObject;
    let select = document.querySelector('#propSelect select');
    //interface
    let name = '';
    if (taskObject.name) {
        name = ': ' + taskObject.name;
    }
    document.querySelector('#taskTitle').innerHTML = taskObject.id + name;

    if (!document.querySelectorAll('#propSelect select option').length) {
        jsonModdle.types.forEach((type) => {
            let opt = document.createElement('option');
            opt.value = type.name;
            opt.innerHTML = type.name.replaceAll('-',' ');
            select.appendChild(opt);
        });
    }
}
```

Obrázek 47: Funkce taskModal() (Zdroj: Vlastní zpracování)

3.2.1.2 Definice nových atributů

Tato funkcionalita se týká způsobu, jakým v rámci BPMN editoru definuji existující možné sub-akce. Jak jsem již zmínil, pro tyto účely využiji JSON slovníku a rozšíření extension elements.

Slovník sestává z jednoho hlavního objektu. V něm definuji název slovníku, předponu určenou pro tagy a pole samotných typů sub-akcí. Název a předpona jsou vyžadovány bpmn-js rozšířením extension elements k tomu, aby mohl být slovník přidán do definic BPMN moddle a editor byl schopný pomocí jeho definic vytvářet tagy do extension elements objektu jednotlivých prvků.

Pole sub-akcí (nazvané types) pak již obsahuje objekty pro sadu vybraných sub-akcí, které jsou v současné době požadovány v rámci editoru. Díky tomu stačí pro budoucí rozšiřování funkcionality pouze přidávat nové objekty do tohoto pole.

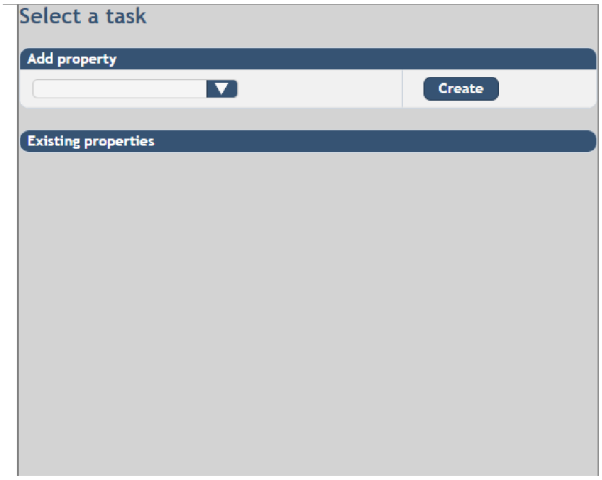
Objekty slovníku obsahují 2 hlavní informace. První z nich je název objektu, který je shodný s názvem sub-akce a využívá se pro název tagu. Druhou je pak pole properties, jenž obsahuje objekty popisující jednotlivé atributy sub-akce. Tento popis obsahuje jak název atributu, tak datový typ jeho hodnoty. Na následujícím obrázku se nachází část výsledného slovníku.

Obrázek 48: Náhled JSON slovníku workflowModdle (Zdroj: Vlastní zpracování)

Rozšíření extension elements je původní funkcionalitou standardu BPMN. V běžném modelování se však nevyužívá a její implementace jsou převážně v podnikovém prostředí v situacích, kdy společnosti musí standard rozšířit pro uspokojení svých potřeb či požadavků zákazníků. Vlastnost extension elements se váže na businessObject jednotlivých prvků v diagramu. Ten popisuje funkcionální součásti prvku (název, vlastnosti, potomky apod…). Ve výchozím stavu businessObject neobsahuje část extension elements, pro její použití je nutné ji přidat v každém takto modifikovaném prvku. Při finální validaci modelu před vykreslením diagramu ji pak libovolný BPMN editor zkontroluje proti definovaným rozšířením v BPMN moddle a pokud jim extension elements odpovídá, je vepsán jako validní rozšíření do BPMN modelu. Tento proces budu aplikovat při editaci atributů činností.

3.2.1.3 Panel pro editaci atributů

Poslední rozšíření BPMN editoru se týká zobrazovacího panelu pro editaci atributů. V předchozích částech jsem popsal funkcionální způsoby, jakými lze vybírat konkrétní prvky a přistupovat k jejich extension elements datům. Nyní je zapotřebí vytvořit panel v uživatelském rozhraní, který dokáže zobrazit data konkrétního prvku a upravovat je. Panel bude sestávat ze statické části, která bude sloužit pro výběr sub-akce a dynamické části, která bude jak zobrazovat atributy pro editaci zvolené sub-akce, tak zobrazovat aplikované vlastnosti na vybraném prvku v editoru. Samotná šablona vytvořeného panelu vypadá následovně.



Obrázek 49: Šablona panelu pro editaci atributů (Zdroj: Vlastní zpracování)

První funkce, která zprostředková část této funkcionality je propertyEditor(). Ta dokáže na základě zvolené sub-akce z nabídky, do které importuji data z typů definovaných v JSON slovníku, dynamicky vytvořit tabulku pro editaci atributů.

```
//Completes interface form from extension wf moddle
function propertyEditor(moddleObj,select,table) {
    let wfProp = moddleObj.types.find(e => e.name === select.value);
    table.innerHTML = '';
    wfProp.properties.forEach((prop) => {
        let row = document.createElement('tr');
        let nameCell = document.createElement('td');
        nameCell.innerHTML = prop.name.split(':')[1];
        let inputCell = document.createElement('td');
        //input types
        let input = document.createElement('input');
        if (prop.type === 'Boolean') {
            input.setAttribute('type','checkbox');
        } else if (wfProp.name === 'code' && prop.name === 'wf:input') {
            input.style.display = 'none';
            let cmInput = document.createElement('textarea');
            cmInput.id = 'codeEditor';
            inputCell.appendChild(cmInput);
            let cmEditor = CodeMirror.fromTextArea(cmInput, {
                lineNumbers: true,
                gutter: true,
                lineWrapping: true,
                value: '\n'
            });
            cmEditor.on('change',function(e){
                let content = e.getValue();
                input.value = content;
            });
        } else {
            if (prop.name === 'wf:id') {
                input.required = true;
            }
            input.placeholder = prop.type;
        }
        inputCell.appendChild(input);
        row.appendChild(nameCell);
        row.appendChild(inputCell);
        table.appendChild(row);
    });
}
```

Obrázek 50: Funkce propertyEditor() (Zdroj: Vlastní zpracování)

Funkce addNewProperty() slouží k přiřazení sub-akce ve formě extension elements konkrétnímu prvku činnosti. Vytvoří rozšiřující prvek dle definice ve slovníku a hodnoty jeho atributů získá z formuláře v panelu, který se dynamicky vytvořil při výběru typu sub-akce. Formulář je samozřejmě ošetřený tak, aby vždy obsahoval všechny povinné hodnoty. Funkce přiřazuje vytvořeným sub-akcím vlastní identifikátor, který usnadňuje jejich budoucí vyhledávání či zajišťování návaznosti.

```
//Adds new property to task within bpmn model on form submit (inserted into xml to be usable in transforms)
function addNewProperty(task,parentProp){
    let moddle = bpmnModeler.get('moddle');
    let modeling = bpmnModeler.get('modeling');
    let taskExtension;
    let extensionElements = task.businessObject.extensionElements;
    let addedProperty = document.querySelector('#addPropForm #propSelect select').value;

    if(!extensionElements) {
        extensionElements = moddle.create('bpmn:ExtensionElements');
    }
    if (!taskExtension) {//condition might be unnecessary
        taskExtension = moddle.create('wf:'+addedProperty);
        if (parentProp) {
            taskExtension.$parent = parentProp;
            parentProp.$children = [taskExtension];
            if (!parentProp.values) {
                parentProp.values = [taskExtension];
            } else {
                parentProp.values.push(taskExtension);
            }
        }
        extensionElements.get('values').push(taskExtension);
    }

    let propRows = document.querySelectorAll('#addPropForm table#addProp tbody tr');

    //assign unique id to property for dependency solver
    let propId = 'prop' + globalPropCounter;
    $.extend(taskExtension.$attrs,{['propId']:propId});
    //add task extension attributes
    propRows.forEach((row) => {
        let pKey = 'wf:'+row.childNodes[0].innerHTML;
        let pVal = row.childNodes[1].querySelector('input').value;
        if (row.childNodes[1].querySelector('input').type === 'checkbox') {
            pVal = row.childNodes[1].querySelector('input').checked;
        }
        let obj = {[pKey]:pVal};
        //extend attributes object of taskExtension with form data
        $.extend(taskExtension.$attrs,obj);
    });
    if (parentProp) {
        $.extend(taskExtension.$attrs,{['parentId']:parentProp.$attrs.propId});
    }
    modeling.updateProperties(task,{extensionElements});
    globalPropCounter++;
};
```

Obrázek 51: Funkce addNewProperty() (Zdroj: Vlastní zpracování)

Funkce displayExistingProperties() slouží k dynamickému zobrazení všech extension elements na vybraném prvku diagramu. Toto zobrazení je zprostředkováno tabulkou. Kromě toho také dynamicky vykresluje rozhraní pro interakci s prvkem, jako je přidávání, úprava či mazání sub-akcí.

```
//Gets tasks existing properties to insert into interface
function displayExistingProperties(task) {
    let extensions = task.businessObject.extensionElements;
    if (extensions) {
        let extArray = extensions.get('values');
        let table = document.querySelector('#existProp tbody');
        table.innerHTML = '';
        extArray.forEach((ext)=>{
            //property title
            let nameRow = document.createElement('tr');
            nameRow.classList.add('propCtlRow');
            let name = document.createElement('td');
            let buttons = document.createElement('td');
            name.classList.add('propertyName');
            name.innerHTML = ext.$type.split('wf:')[1];
            nameRow.appendChild(name);
            //add options for properties
            let edit = document.createElement('button');
            let nest = document.createElement('button');
            let remove = document.createElement('button');

            edit.innerHTML = 'Edit';
            nest.innerHTML = 'Add property';
            remove.innerHTML = 'X';

            edit.addEventListener('click',function(e){editProperty(task,ext,e);});
            nest.addEventListener('click',function(){nestProperty(task,ext);});
            remove.addEventListener('click',function(){removeProperty(task,ext);});

            buttons.appendChild(edit);
            buttons.appendChild(nest);
            buttons.appendChild(remove);
            nameRow.appendChild(buttons);

            table.appendChild(nameRow);
            //property attributes
            Object.entries(ext.$attrs).forEach(([key, value]) => {
                if (key !== 'propId' && key !== 'parentId') {
                    let attrRow = document.createElement('tr');
                    let attrKey = document.createElement('td');
                    attrKey.innerHTML = `${key}`.split('wf:')[1];
                    attrRow.appendChild(attrKey);
                    let attrVal = document.createElement('td');
                    attrVal.innerHTML = `${value}`;
                    attrRow.appendChild(attrVal);
                    table.appendChild(attrRow);
                }
            });
            if (extArray.length > 1 && extArray.indexOf(ext) !== extArray.length-1) {
                let separatRow = document.createElement('tr');
                let separator = document.createElement('td');
                separator.classList.add('separator');
                separator.setAttribute('colspan',2);;
                separatRow.appendChild(separator);
                table.appendChild(separatRow);
            }
        });
    }
}
```

Obrázek 52: Funkce displayExistingProperties() (Zdroj: Vlastní zpracování)

Funkce editProperty() dynamicky vykresluje formulář pro modifikaci vybrané sub-akce na konkrétním prvku. Po potvrzení formuláře upraví tuto sub-akci a opět zavolá funkci displayExistingProperties(), která aktualizuje zobrazení sub-akcí v rozšiřujícím panelu.

```
function editProperty(task,ext,evt) {
    let modeling = bpmnModeler.get('modeling');
    let extensionElements = task.businessObject.extensionElements;
    let propIndex = extensionElements.get('values').indexOf(ext);
    let propAttrCount = Object.keys(ext.$attrs).length;
    let firstRow = evt.srcElement.parentElement.parentElement;
    let nextRow = firstRow.nextSibling;

    //toggleable edit (switches between text and inputs)
    if ((nextRow.children[1].innerHTML.includes('input') === false)) {
        for (let i = 0; i < propAttrCount-1; i++) {
            //switch to input
            let cell = nextRow.childNodes[1];
            let input = document.createElement('input');
            if (cell.innerHTML) {
                input.placeholder = cell.innerHTML;
                cell.innerHTML = input.outerHTML;
            }
            nextRow = nextRow.nextSibling;
        }
        //add updateBtn and fn
        let updateBtn = document.createElement('button');
        updateBtn.innerHTML = 'Update property';
        updateBtn.classList.add('propUpdate');
        updateBtn.addEventListener('click',function(e){
            let rowArr = [];
            let next = firstRow.nextSibling;
            for (let i = 0; i < propAttrCount-1; i++) {
                rowArr.push(next);
                next = next.nextSibling;
            }
            rowArr.forEach((row) => {//adds task extension attributes
                let pKey = 'wf:'+row.childNodes[0].innerHTML;
                let pVal = row.childNodes[1].querySelector('input').value;
                if (!pVal) {
                    pVal = row.childNodes[1].querySelector('input').placeholder;
                    if (!pVal) {
                        pVal = '';
                    }
                }
                if (row.childNodes[1].querySelector('input').type === 'checkbox') {
                    pVal = row.childNodes[1].querySelector('input').checked;
                }
                let obj = {[pKey]:pVal};
                $.extend(ext.$attrs,obj);//extends attributes object of taskExtension with form data
            });
            extensionElements.get('values').splice(propIndex,1,ext);
            modeling.updateProperties(task,{extensionElements});
            displayExistingProperties(task);
        });
        firstRow.children[1].appendChild(updateBtn);
    } else {
        for (let i = 0; i < propAttrCount; i++) {
            //switch to text
            let cell = nextRow.children[1];
            let input = cell.children[0];
            cell.innerHTML = input.placeholder;
            nextRow = nextRow.nextSibling;
        }
        //remove updateBtn and fn
        firstRow.children[1].removeChild(firstRow.querySelector('.propUpdate'));
    }
}
```

Obrázek 53: Funkce editProperty() (Zdroj: Vlastní zpracování)

Funkce removeProperty() slouží k odstranění sub-akce z vybraného prvku. Po odstranění volá funkci displayExistingProperties() pro aktualizaci panelu atributů.

```
//Remove property from element
function removeProperty(task,ext) {
    let modeling = bpmnModeler.get('modeling');
    let extensionElements = task.businessObject.extensionElements;
    let index = extensionElements.get('values').indexOf(ext);

    extensionElements.get('values').splice(index,1);
    modeling.updateProperties(task,{extensionElements});
    displayExistingProperties(task);
}

//Clears interface when an action is deselected
function clearTaskModal() {
    let cell = document.querySelector('#propSubmit');
    let options = document.querySelectorAll('#propSelect option');

    document.querySelector('#taskTitle').innerHTML = 'Select a task';
    document.querySelector('#addProp th').innerHTML = 'Add property';
    document.querySelector('#addProp tbody').innerHTML = '';
    document.querySelector('#existProp tbody').innerHTML = '';

    if (cell.children.length > 1) {
        cell.removeChild(cell.children[1]);
    }
    if (options) {
        options.forEach((opt) => {
            opt.parentElement.removeChild(opt);
        });
    }
}
```

Obrázek 54: Funkce removeProperty() (Zdroj: Vlastní zpracování)

Tato výsledná implementace umožňuje uživateli, aby ve vytvořeném BPMN modelu přiřadil či upravil rozšíření činností. Díky tomu lze plně modelovat veškeré stavy, akce a sub-akce GPC workflow a výsledný model tak transformovat k použití uvnitř platformy.

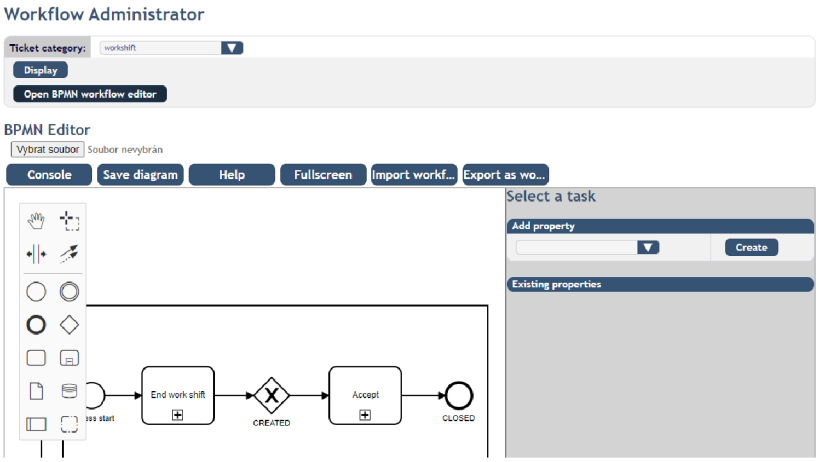
3.2.2 Rozšíření Workflow editoru

V druhé části kapitoly se budu věnovat workflow editoru. Ten je původní součástí platformy GPC a doposud neprocházel žádnými významnějšími změnami. I rozšíření navržená v této práci budou spíše kosmetická a zprostředkují pouze jednodušší práci uživatelů při editaci workflow.

3.2.2.1 Předání workflow do transformace

První funkcionalita bude zodpovědná za zobrazování BPMN editoru při editaci workflow. Měla by umožnit uživateli kdykoli existující workflow transformovat na BPMN model a ten otevřít v editoru pro jednodušší editaci a čitelnost. Tento proces by měl být ideálně na jednom místě, aby se uživatel nemusel navigovat několika stránkami a neustále ukládat a nahrávat soubory.

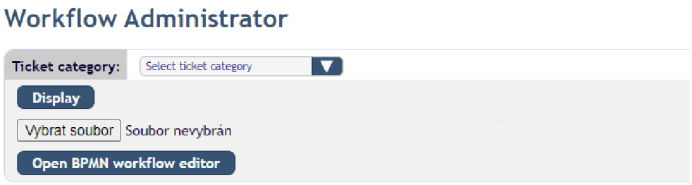
Funkcionalita tedy ve workflow editoru přidá tlačítko, které jednou uživatelskou interakcí inicializuje GPC komponentu BPMN editoru, exportuje a transformuje současné workflow do BPMN modelu, který následně zobrazí v editoru. Stejným tlačítkem pak uživatel bude moci opět komponentu skrýt a mít přístup ke standardnímu editoru workflow. Tento přístup ocení jak noví uživatelé bez znalostí proprietárních workflow, tak stávající vývojáři, kteří jsou zvyklí je upravovat přímo a bez obstrukcí.



Obrázek 55: Komponenta BPMN editoru v rámci workflow (Zdroj: Vlastní zpracování)

3.2.2.2 Import workflow ze souboru

Poslední navrhovaná funkcionalita umožní importovat GPC workflow ve formátu XML do textového editoru workflow. Současné řešení totiž funguje na bázi vkládání textu do textového editoru. Toto nejen přidá další způsob importu workflow, ale také zprostředkovaně umožní nahrát BPMN model po transformaci na workflow pomocí fiktivního souboru.



Obrázek 56: Funkcionalita pro import workflow ze souboru (Zdroj: Vlastní zpracování)

3.3 Ověření výsledného řešení

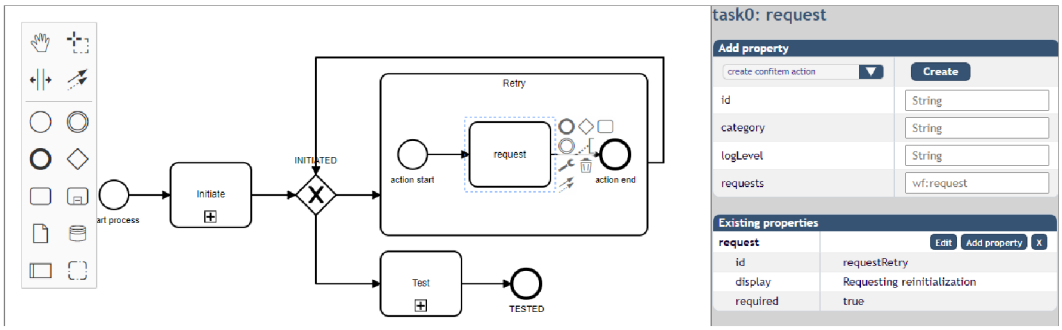
V této kapitole otestuji navržené funkcionality pro zjištění jejich funkčnosti a případných nedostatků.

3.3.1 Transformace workflow na bpmn model

Nejprve otestuji transformaci z workflow na cvičném souboru. Ten má obdobnou strukturu jako klasická workflow, ale obsahuje zjednodušené informace. Tento testovací soubor lze vidět na následujícím obrázku:

Obrázek 57: Cvičný workflow soubor (Zdroj: Vlastní zpracování)

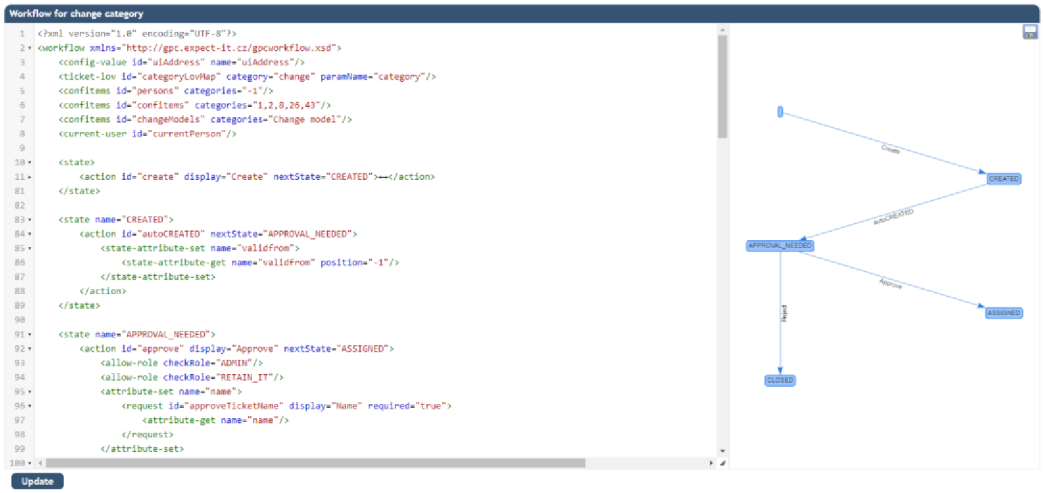
Výsledný transformovaný BPMN model pak vypadá následovně:



Obrázek 58: Cvičné workflow po transformaci na BPMN (Zdroj: Vlastní zpracování)

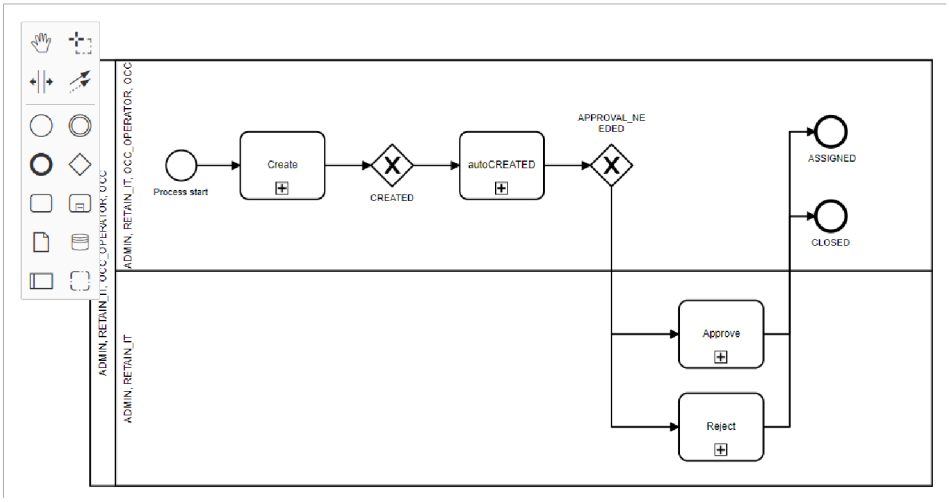
Jak lze z obrázku vidět, modul provedl transformaci úspěšně a dosáhl očekávaného výsledku tak, jak byl popsán v návrhu.

Nyní se pokusím transformovat již existující workflow. Konkrétně se jedná o workflow vyhodnocování procesu změny. Vizuální reprezentaci workflow vyjadřuje následující obrázek:



Obrázek 59: Workflow procesu změny (Zdroj: Vlastní zpracování)

V obrázku jsou stavy procesu reprezentovány pomocí uzlů grafu a akce podle pojmenovaných spojnic. Akce obsahují své sub-akce, ale tato současná vizualizace není schopná jejich zobrazení. Po transformaci modul vygeneroval následující BPMN model:

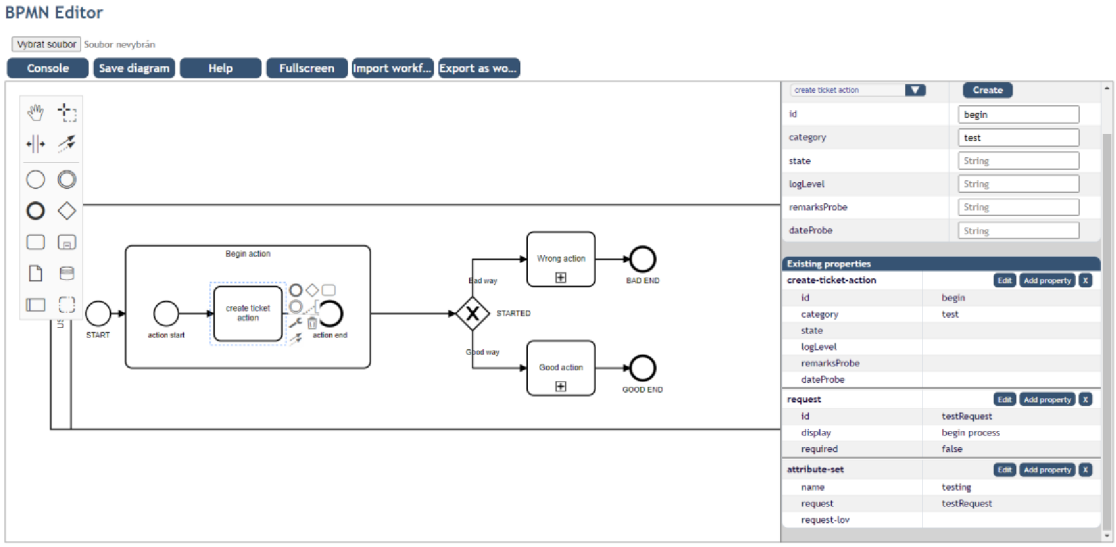


Obrázek 60: Transformovaný BPMN model procesu změny (Zdroj: Vlastní zpracování)

Opět jsem zjistil, že transformace proběhla úspěšně a vygenerovala BPMN model, který reprezentuje workflow tak, jak bylo navrženo.

3.3.2 Úprava a transformace BPMN modelu na workflow

Transformace z BPMN modelu na workflow je komplexnější než předchozí proces. Proto její testování začnu na jednoduchém vytvořeném procesu:



Obrázek 61: Cvičný BPMN model (Zdroj: Vlastní zpracování)

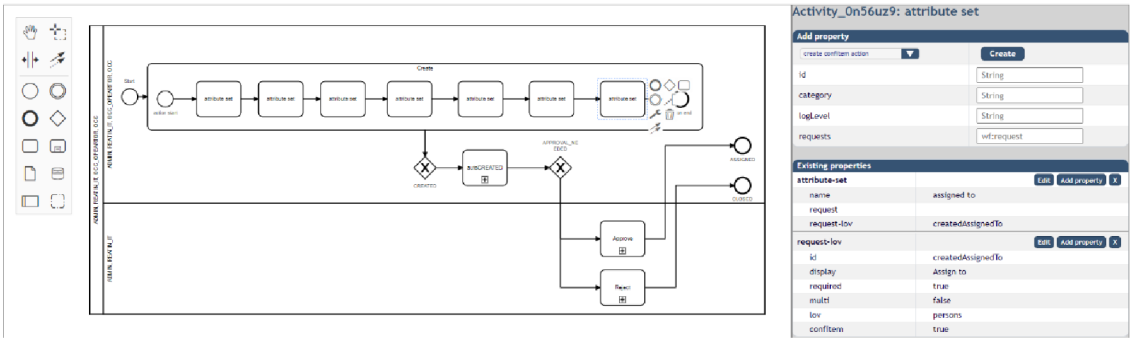
Z procesu bylo transformací vygenerováno následující workflow:

```
<?xml version="1.0" encoding="UTF-8"?>
    <workflow xmlns="http://gpc.expect-it.cz/gpcworkflow.xsd">
        <state name="START">
            <action id="Activity_1vo2kq3" display="Begin action" nextState="STARTED">
                <allow-role checkRole="USER"></allow-role>
                <create-ticket-action id="begin" category="test">
                    <attribute-set name="testing" request="testRequest">
                        <request id="testRequest" display="begin process" required="false"/>
                    </attribute-set>
                </create-ticket-action>
            </action>
        </state>
        <state name="STARTED">
            <action id="Activity_0iaw0zh" display="Good action" nextState="GOOD END">
                <allow-role checkRole="USER"></allow-role>
            </action>
            <action id="Activity_1iik6wf" display="Wrong action" nextState="BAD END">
                <allow-role checkRole="USER">
                </allow-role>
            </action>
        </state>
    </workflow>
</xml>
```

Obrázek 62: Transformovaný cvičný model na workflow (Zdroj: Vlastní zpracování)

Workflow odpovídá navrženému procesu, obsahuje všechny stavy a jejich akce. U jedné akce byla modelována i sub-akce s dalšími vnořenými operacemi. Také ona se korektně promítla ve výsledném workflow. Do všech akcí se také správně aplikovala role uživatele.

Nyní použiji jako vstup transformace již dříve zmíněný model skutečného procesu pro vyhodnocování změny:

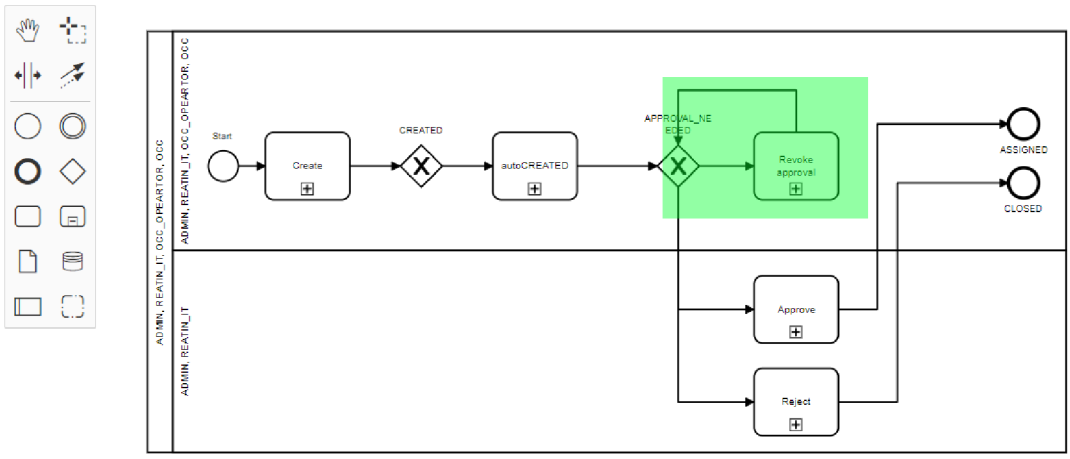


Obrázek 63: Replikovaný BPMN model procesu změny (Zdroj: Vlastní zpracování)

Obrázek 64: Workflow změny vygenerováno transformací z BPMN (Zdroj: Vlastní zpracování)

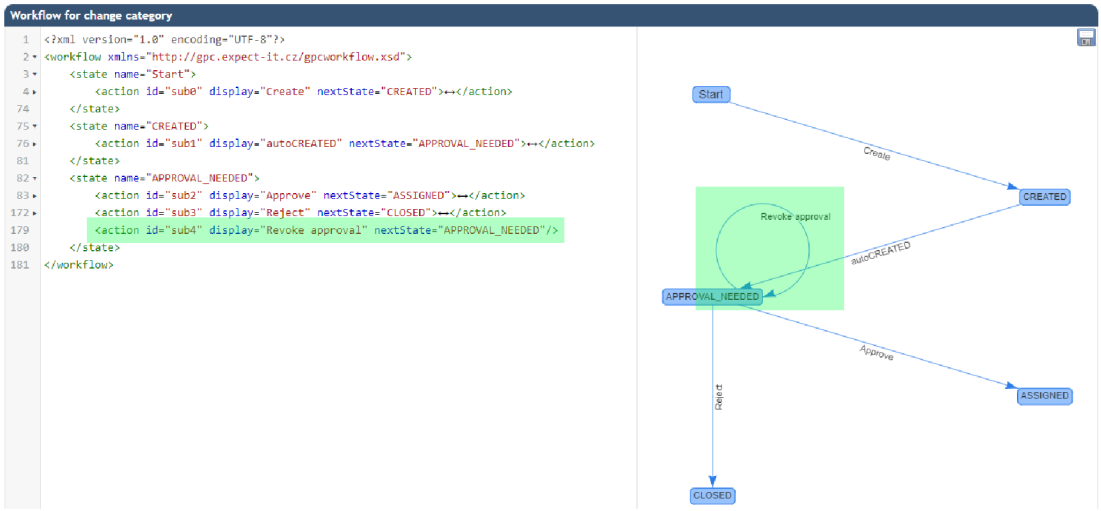
Výsledné workflow je stejné, jako bylo ve svém původním stavu mimo transformace. Transformační modul tedy lze považovat za funkční, jelikož produkuje očekávaná a syntakticky validní workflow, při opačné transformaci pak generuje funkční BPMN modely.

Nyní se pokusím stejný počáteční bpmn model upravit přidáním akce Revoke approval ve stavu Approval needed. Tato akce by měla pokračovat zpět do původního stavu. Po transformaci by tato změna měla být zaznamenána ve workflow.



Obrázek 65: Přidaná akce v BPMN modelu změny (Zdroj: Vlastní zpracování)

Podle výsledku transformace usuzuji, že i modifikace v BPMN modelu jsou korektně přenášeny do workflow. Jako poslední test ověřím, zda toto modifikované workflow lze bez problémů nahrát do platformy a změna bude znázorněna i ve vizualizaci workflow.



Obrázek 66: Vygenerované workflow s přidanou akcí (Zdroj: Vlastní zpracování)

Jak lze pozorovat, workflow bylo úspěšně nahráno i s provedenou změnou. Na základě těchto testů tedy můžu považovat návrh a implementaci řešení, které bylo tématem této práce, za úspěšné.

3.4 Budoucí možné úpravy

Ačkoli transformace proběhly úspěšně, pozoroval jsem několik nedostatků či funkcionalit, které by nástroj více zdokonalily.

Hlavní současný nedostatek nástroje tkví v tom, že při transformaci nemá způsob, jak přenést diagramové rozložení do workflow, aby při opětovné transformaci zpět na BPMN model zůstalo zachováno rozložení diagramu. Tento problém je méně závažný u jednodušších procesů či v případě, že uživatel zachovává při modelování výchozí rozmístění prvků, které generuje editor. Pokud se však jedná o komplexnější model, obsahuje větší počet oblastí či byl modelován tak, že uživatel specificky rozmisťoval jednotlivé prvky, ztráta těchto dat má znatelný dopad na čitelnost modelu. Vhodným řešením tohoto problému by bylo rozšířit možnosti workflow tak, aby u stavů a akcí bylo možné zachovat alespoň souřadnice prvků. Při transformaci by pak přidaná funkcionalita po vygenerování diagramové části vyhledala a aktualizovala prvky, které by obsahovaly již ve workflow souřadnicová data. Tento postup by ale zabral značnou dobu vývoje, především kvůli tomu, že některé prvky modelu nejsou přímo reprezentovány ve workflow, ale jsou výsledkem transformačního modulu.

Nedostatek lze pozorovat také v případě validace BPMN modelů. Ačkoli zde existují některé základní omezení, které předchází tomu, aby bylo možné transformovat chybný proces (a upozorňují uživatele na konkrétní chyby modelu), neexistuje zde podrobnější validace, která by zkoumala například souvislosti v atributech společných sub-akcí či spustitelnost workflow během modelování.

Další úpravy by se mohly týkat panelu pro úpravu činností. Jednotlivé atributy sub-akcí jsou nyní vyplňovány ručně. V budoucnu by však bylo vhodné umožnit systému detekci dle typu sub-akce a případné nabízení seznamů použitelných hodnot, například pokud má vyplňovaný atribut referencovat jednu z kategorií tiketů systému (pro hodnoty by se například vypsal seznam těchto kategorií).

Z obecného pohledu by pak bylo vhodné, aby se vytvořila dokumentace či návody pro obsluhu obou editorů. Ty by umožnily rychlejší zorientování uživatelů a obecně zjednodušily práci s nástroji. Mohly by například obsahovat best practises při návrhu workflow, popis a odůvodnění způsobu, jakým jsou namapovány prvky či příklady obecných procesů, které bývají pomocí workflow nejčastěji modelovány.

V případě, kdy by se platforma začala poskytovat jako OTS produkt, bylo by možné v rámci dokumentací a návodů vytvořit také komunitní fórum, pomocí kterého by mohl vývojářský tým komunikovat s komunitou, která by si vzájemně mohla poskytovat zpětnou vazbu a kolaborovat na řešení případných problémů jednotlivých uživatelů.

Tyto změny již nejsou předmětem diplomové práce, ale zmínil jsem je pro nastínění budoucího možného vývoje platformy.

Závěr

V této diplomové práci jsem se zabýval návrhem a implementací nástroje, který umožní uživatelům platformy GPC vytvářet proprietární konfigurační workflow pomocí procesních modelů standardu BPMN 2.0.

V první řadě jsem prozkoumal teoretické pozadí této problematiky a následně provedl analýzu současné tvorby workflow a jejich nedostatků. Na základě analýzy jsem pak vytvořil prvotní abstraktní návrh modulu, který má za úkol oboustranně zprostředkovat transformace mezi workflow a BPMN modely. Po prozkoumání specifik XML notací obou prostředků jsem zpracoval návrh vzájemného mapování jejich prvků. Tento krok byl stěžejní k tomu, aby bylo umožněno převádění navržených procesů bez ztrát informací či změně procesů. Zároveň jsem kladl důraz na zachování jednotnosti obou směrů transformace, aby byl výsledný software dobře čitelný a efektivní.

Po zmapování prvků jsem již vytvořil konkrétnější logické popisy procesů transformace, které již bylo možné optimalizovat a dále využít k návrhu konkrétních algoritmů v jazyku Javascript. Algoritmy jsem následně implementoval a detailně popsal. V rámci zpracování návrhu a implementace bylo také nutné provést některé změny v uživatelském rozhraní editorů, především z pohledu BPMN.

V této fázi práce již byl nástroj dokončený a funkční. Dále jsem se věnoval ověřování jeho funkčnosti, ošetřování chyb a identifikaci případných nedostatků. Uživatelské testy prokázaly, že navržený nástroj splňuje požadavky na funkčnost a použitelnost a zprostředkovává uživatelům možnost tvorby workflow pomocí BPMN editoru.

V poslední části práce jsem identifikoval několik současných nedostatků nástroje a několik vhodných rozšíření, kterým bych se mohl v budoucnu věnovat pro jeho dodatečné zdokonalení. Tato rozšíření by v budoucnu dále zjednodušily práci s nástrojem a obecně umožnily uživatelsky příjemnější, rychlejší a efektivnější konfiguraci platformy GPC.

Přínosy této práce lze rozdělit do dvou skupin: přímé a nepřímé. Přímým přínosem je samotné zpracování funkcionality, jelikož je jednou z několika specifických funkcionalit, které jsou vyžadovány pro splnění konkrétní poptávky. Další přímý přínos je umožnění návrhu a modelování workflow pomocí diagramů standardu BPMN 2.0.

Mezi nepřímé přínosy patří v první řadě přesunutí možností konfigurace z interních vývojářů na samotné uživatele platformy. Díky tomu budou uvolněny některé firemní zdroje, z pohledu uživatelů pak bude možné docílit většího pochopení platformy, které povede k jejímu efektivnějšímu užívání. Samotný proces konfigurace (tvorba workflow) se díky navržené funkcionalitě podstatně zjednodušil, jelikož umožňuje přechod z programátorského návrhu na vizuální práci zprostředkovanou diagramy. Díky využití standardu BPMN 2.0 se navíc jedná o rozšířený způsob procesního modelování, který má v povědomí většina manažerů či jej lze případně v krátkém čase uživatele doučit. Funkcionalita také zlepšuje výhledy platformy z pohledu OTS distribuce, jelikož nadále nevyžaduje spolupráci interního vývojového týmu se zákazníky k zajištění prvotní konfigurace platformy.

Tyto i další přínosy práce ve finálním hledisku zlepšují využitelnost platformy, zjednodušují a urychlují uživatelské procesy a obecně zvětšují již existující konkurenční výhody platformy.

Seznam použitých zdrojů

[1] VON ROSING, Mark, August-Wilhelm SCHEER a Henrik VON SCHEEL. *The complete business process handbook: body of knowledge from process modeling to BPM*. Waltham, MA: Morgan Kaufmann, 2015. ISBN 9780127999593.

[2] REINHARTZ-BERGER, Iris a ZDRAVKOVIC, Jelena. *Enterprise, Business-Process and Information Systems Modeling: 20th International Conference, BPMDS 2019, 24th International Conference, EMMSAD 2019...* 1st edition. Springer, 2019. ISBN 978-3030206178.

[3] ENSTROM, David. *A Simplified Approach to It Architecture with Bpmn: A Coherent Methodology for Modeling Every Level of the Enterprise*. iUniverse, 2016. ISBN 978-491784976.

[4] PILONE, Dan a Neil PITMAN. *UML 2.0 in a nutshell*. Sebastopol: O'Reilly, 2005. ISBN 9780596007959.

[5] AMJAD, Anam, AZAM, Farooque, ANWAR, Muhammad Waseem, BUTT, Wasi Haider a RASHID, Muhammad. *Event-Driven Process Chain for Modeling and Verification of Business Requirements–A Systematic Literature Review* [online]. IEEE, 2018, s. 9027-9048 [cit. 2021-12-22]. Dostupné z: https://ieeexplore.ieee.org/document/8306984

[6] TYLER, Cedric a BAKER, Stephen. *Business Genetics: Understanding 21st Century Corporations using xBML*. 1st Edition. Hoboken, NJ: Wiley, 2007. ISBN 978-0470066546.

[7] XPDL - Workflow Management Coalition. *Workflow Management Coalition* [online]. [cit.2021-12-22]. Dostupné z: https://www.wfmc.org/standards/xpdl

[8] MENDLING, Jan a Matthias WEIDLICH. *Business Process Model and Notation: 4th International Workshop, BPMN 2012, Vienna, Austria, September 12-13, 2012. Proceedings (Lecture Notes in Business Information Processing).* New York: Springer, 2012. ISBN 978-3642331541.

[9] STIEHL, Volker. *Process-Driven Applications with BPMN*. Springer, 2014. ISBN 978-3319072173.

[10] SILVER, Bruce. *BPMN method and style: with BPMN implementer's guide. 2nd edition.* Aptos: Cody-Cassidy Press, 2011. ISBN 978-0-9823681-1-4.

[11] BPMN 2.0 Implementation Reference | docs.camunda.org. *Camunda BPM Documentation* [online]. [cit. 2021-12-22]. Dostupné z: https://bit.ly/3J9apEh

[12] OUYANG, Chun, Marlon DUMAS, Arthur TER HOFSTEDE a VAN DER AALST, Wil. *From BPMN Process Models to BPEL Web Services* [online]. IEEE International Conference on Web Services (ICWS'06), 2006, s. 285-292 [cit. 2021-12-22]. Dostupné z: https://ieeexplore.ieee.org/document/4032038

[13] CARDA, Antonín a KUNSTOVÁ, Renáta. *Workflow – nástroj manažera pro řízení podnikových procesů*. Grada, 2006. ISBN 978-8024706665.

[14] RUSSELL, Nick, HOFSTEDE, Arthur ter a VAN DER AALST, Wil. *Workflow Patterns: The Definitive Guide*. MIT Press, 2016. ISBN 978-0262029827.

[15] EBY, Kate. *Six Sigma for Beginners. Smartsheet* [online]. 2021 [cit. 2021-12-22]. Dostupné z: https://www.smartsheet.com/all-about-six-sigma

[16] GOLDRATT, Eliyahu. *Theory of constraints*. Great Barrington, MA: The North River Press, 1998. ISBN 978-0884271666.

[17] SKOURADAKI, Marigianna, ROLLER, Dieter, LEYMANN, Frank, FERME, Vincenzo a PAUTASSO, Cesare. *On the Road to Benchmarking BPMN 2.0 Workflow Engines. In Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering* [online]. Association for Computing Machinery, 2015, s. 301-304 [cit. 2021-12-22]. Dostupné z: https://dl.acm.org/doi/10.1145/2668930.2695527

[18] FLANAGAN, David. *JavaScript: The Definitive Guide, Sixth Edition*. Sebastopol: O'Reilly Media, 2011. ISBN 978-0596805524.

[19] FRIESEN, Jeff. *Java XML and JSON: Document Processing for Java SE. 2nd edition.* Apress, 2019. ISBN 978-1484243299.

[20] SUBRAMANIAM, Venkat. *Rediscovering JavaScript: Master ES6, ES7, and ES8. 1st Edition*. USA: Pragmatic Bookshelf, 2018. ISBN 978-1680505467.

[21] *Developer Survey Results: Most Popular Technologies - Programming, Scripting, and Markup Languages*. Stack Overflow [online]. 2019 [cit. 2021-12-22]. Dostupné z: https://bit.ly/3pmrzGw

[22] *Usage statistics of JavaScript as client-side programming language on websites: Historical Trend*. W3Techs [online]. 2021 [cit. 2021-12-22]. Dostupné z: https://bit.ly/3sotblj

[23] *Bpmn-auto-layout: Layout BPMN diagrams, generating missing DI information.* Github [online]. 2022 [cit. 2022-04-14]. Dostupné z: https://bit.ly/37PBCyh

[24] HAVELKA, Ondřej. *Vizuální editor workflow pro platformu GPC* [online]. Brno, 2020 [cit. 2022-05-02]. Dostupné z: https://www.vutbr.cz/studenti/zav-prace/detail/125654. Bakalářská práce. Vysoké učení technické v Brně, Fakulta podnikatelská. Ing. Viktor Ondrák, Ph.D.

Seznam použitých obrázků

Seznam použitých zkratek

BPMN	Business Process Model and Notation
UML	Unified Modeling Language
EPC	Event-Driven Process Chain
xBML	extended Business Modelling Language
XML	Extensible Markup Language
XPDL	XML Process Definition Language
UX	User Experience
UI	User Interface
BPEL	Business Process Execution Language
JS	JavaScript
ES	EcmaScript
JSON	JavaScript Object Notation
DMAIC	Define, Measure, Analyze, Improve, Control
DMADV	Define, Measure, Analyze, Design, Verify
TOC	Theory Of Constraints
XSD	XML Schema Definition
OTS	Off The Shelf

Pozn.: V seznamu nejsou uvedeny všeobecně známé zkratky nebo zkratky, které se v textu vyskytly pouze ojediněle s vysvětlením.