Sem vložte zadání Vaší práce.

České vysoké učení technické v Praze
Fakulta informačních technologií
Katedra softwarového inženýrství

Diplomová práce

Data-Flow programování podnikových pravidel nad databázemi

Bc. Aliaksandr Maksimau

Vedoucí práce: Ing. Robert Pergl, Ph.D.

25. června 2015

Poděkování

Rád bych poděkoval panu Ing. Robertu Perglovi, Ph.D. za jeho cenné rady, připomínky a odborné vedení při zpracování této diplomové práce.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval(a) samostatně a že jsem uvedl(a) veškeré použité informační zdroje v souladu s Metodickým pokynem o etické přípravě vysokoškolských závěrečných prací.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona, ve znění pozdějších předpisů. V souladu s ust. § 2373 občanského zákonníku tímto uděluji nevýhradní oprávnění (licenci) k užití této mojí práce, a to včetně všech počítačových programů, jež jsou její součástí či přílohou a veškeré jejich dokumentace (dále souhrnně jen „Dílo“), a to všem osobám, které si přejí Dílo užít. Tyto osoby jsou oprávněny Dílo užít jakýmkoli způsobem, který nesnižuje hodnotu Díla a za jakýmkoli účelem (včetně užití k výdělečným účelům), vč. možnosti Dílo upravit či měnit, spojit jej s jiným dílem a/nebo zařadit jej do díla souborného. Toto oprávnění je časově, teritoriálně i množstevně neomezené.

V Praze dne 25. června 2015 .....

Odkaz na tuto práci

Maksimau, Aliaksandr. *Data-Flow programování podnikových pravidel nad databázemi.* Diplomová práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2015.

Abstrakt

Tato diplomová práce by měla přispět ke zlepšení situace v databázovém světě v rámci projektu Manta Checker, který řeší kontrolu kvality za pomoci aplikace podnikových pravidel nad databázovými skripty. Cílem práce je implementace modulu vyhodnocení podnikových pravidel a integrace do systému Manta Checker. Aplikace najde praktické uplatnění v průmyslu, protože pomůže snížit náklady při vývoji a údržbě datových skladů.

Klíčová slova Manta Checker, DFP, AST, JavaScript, Rhino, RequireJS, AMD, Manta

Abstract

This thesis should contribute to improving the situation in the database world within the Manta Checker Project, which addresses the quality control with the help of the application of business rules over the database scripts. The aim of this thesis is the implementation of the evaluating module for business rules and its integration to the Manta Checker System. This application could be practically used in industry, as it will help to reduce the cost of developing and maintaining data warehouses.

Keywords Manta Checker, DFP, AST, JavaScript, Rhino, RequireJS, AMD, Manta

Obsah

Seznam obrázků

Úvod

Zajištění řádné kvality softwarového produktu je v dnešní době velice důležité. Pro vývoj a údržbu aplikací, napsaných v moderních programovacích jazycích, existuje spousta pomocných nástrojů v podobě chytrého IDE, pomocných pluginů a nebo samostatných aplikací, které tento problém řeší. Situace je zcela jiná u kontroly kvality databázových skriptů, kde podobné možnosti nejsou dostupné. Tato diplomová práce by měla přispět ke změně situace v databázovém světě v rámci projektu Manta Checker, který řeší kontrolu kvality za pomoci aplikace podnikových pravidel nad databázovými skripty. Implementace modulu vyhodnocení podnikových pravidel je jeden z hlavních cílů této práce.

Toto téma jsem si zvolil, protože je velice důležité a zajímavé vytvořit něco, co najde praktické uplatnění v průmyslu, řeší konkretní problémy a zlepšuje současný stav věcí. Bezpochyby se jedná o práci, jejíž výsledek by měl být přínosný pro podniky. Práce se rovněž zaobírá novými technologiemi, čímž je přínosná i pro autora samotného.

Cíle a požadavky na implementovaný produkt

Cílem práce je implementace DFP (Data Flow Programming - programování pomocí datových toků) interpretu na základě návrhu, vzniklého v rámci projektu Manta Checker. Výsledkem řešení má být modul, který bude zapojen do stávajícího systému Manta Checker, a v rámci práce je potřeba provést integraci. Hotové řešení musí být otestované formou funkčních a integračních testů.

Požadavky na implementovaný modul

Vzhledem k tomu, že modul bude součástí rozsáhlejšího řešení, jsou na něj kladeny následující požadavky:

- Kompatibilita a bezproblémová integrace s systémem Manta Checker
- Flexibilita úprav a rozšiřování, protože v budoucnu mohou vzniknout nové požadavky na stávající modul
- Snadná testovatelnost a řádné předchozí otestování, protože podmínkou pro správné fungování systému je správné fungování jeho součástí

Popis struktury diplomové práce

Tato implementační diplomová práce má pět částí:

- Úvod – seznámení se s problematikou a stanovení cílů
- Analýza – analýza požadavků a prvotního návrhu
- Návrh – návrh jednotlivých komponent modulu
- Realizace – implementace modulu a popis architektury
- Testování – testování výsledné aplikace

Kapitola 1

Analýza

Nejprve popíšu kontext práce a souvislost s projektem Manta Checker. Větší část analýzy interpretu vnikla v rámci tohoto projektu, a proto v této kapitole je stručné shrnutí výsledků analýzy. Jsou zde také popsáné části, které nebyly pokryty původní analýzou, anebo ty, které bylo potřeba upřesnit. Dále popíšu použité technologie a důvody, proč byly zvolené.

1.1 Manta Checker

Projekt Manta Checker je součásti rodiny Manta. Jedná se o automatizovaný nástroj pro syntaktickou a sémantickou analýzu skriptů a dalších zdrojových souborů. Projekt vznikl ze sady nástrojů pro analýzu skriptu, které byly zaměřené na specifické SQL dialekty a byly kastomizované pro potřeby zákazníka. První nástroje zvládaly pokročilé funkce, ale práce s podnikovými pravidly vyžadovala dobré technické znalostí.

S vzrůstajícím počtem použití datových skladů v oblasti Business Inteligence (BI) vzniká v aplikacích větší potřeba pro kontrolu kvality, jelikož údržba takových systémů je velíce nákladná a vyžaduje velký počet lidských zdrojů. Často je BI řešení ve firmách také postaveno na technologiích různých dodavatelů, což znamená, že je potřebné v nástrojích, které by mohly provádět analýzu v celém kontextu řešení.

V reakci na potřeby trhu byly naplánovány změny existujících nástrojů, provedení jejich konsolidace a zobecnění pro možnost aplikace na širokou škálu skriptu. Změny se plánovaly na všech úrovních fungování nástroje od vstupu přes zadávání pravidel do vyhodnocování.

První krok byl unifikace vstupu, t.j. bude možno provádět analýzu jedním nástrojem bez ohledu na konkretní dialekt, ve kterým byl zdrojový soubor napsán. Aby to bylo možné zajistit, bylo potřeba navrhnout a implementovat obecnou strukturu, na kterou bude převeden konkretní vstup.

Dále potřeba změnit způsob programování pravidel. Bylo potřeba zjednodušit zadávání, úpravu a vyhodnocování podnikových pravidel. Proto byl

analyzován jazyk pro implementaci pravidel, který by byl dostatečně jednoduchý, aby mohl být použit i business uživateli, bez hluboké technické znalosti, a zároveň dostatečně silný, aby splňoval veškeré potřeby, kladené na nástroj. Byla provedena analýza a design vizualizačního nástroje pro manipulací s pravidly a jejich tvorbu, který by měl umožnit uživatelsky přívětivý a efektivní způsob práce s pravidly. Nakonec byla vypracována analýza a návrh modulu pro vyhodnocování pravidel, který je dovede rychle zpracovávat.

Tato práce navazuje na provedené aktivity a zabývá se implementací modulu pro vyhodnocení podnikových pravidel.

Čerpáno z [1].

1.2 Ontologická analýza problematiky

Jelikož v rámci celé aplikaci Manta Chacker byla na začátku provedená unifikace a zobecnění vstupu, je třeba popsat strukturu a pojmy se kterými pak budou pracovat její jednotlivé části.

Jako univerzální struktura, na kterou lze převést jakýkoliv soubor se skriptem nebo zdrojovým kódem je abstraktní syntaktický strom (AST). Dále je možné s takovouto strukturou pohodlně pracovat, optimalizovat ji, a používat veškeré existující algoritmy pro práci se stromy.

1.2.1 Strukturální analýza

Interpret bude operovat nad strukturou syntaktického stromu. Můžeme tedy definovat následující základní prvky se kterými budeme pracovat:

- strom (tree) - graf bez cyklů
- uzel stromu (node) - speciální případ kořen (root)
- označení uzlu (node tag)
- obsah uzlu (node contents)
- atribut uzlu (node attribute)
- klíč atributu (attribute key)
- hodnota atributu (attribute value)

Strom je rekurzivní struktura, tudíž každý uzel stromu definuje vlastní strom, který nazýváme podstrom.

Čerpáno z [2].

1.2.2 Behaviorální analýza

Z hlediska klíčových funkcí a chování jsou zvoleny následující operace:

- vytvoření datového elementu (create)
- změna hodnoty datového elementu (update)
- odstranění datového elementu (delete)
- vyhledání datového elementu ve stromě (search)
- transformace datového elementu (transform)
- ověření podmínky (check)

Čerpáno z [2].

1.3 Analýza vhodných paradigmat

Jako hlavní paradigma vývoje softwaru je pro řešení tohoto zadání zvolené programování pomocí datových toků. V této kapitole popíšu důvod této volby, a také stručně uvedu i další paradigmata, která se používají pro řešení v menší míře.

1.3.1 Programování pomocí datových toků

Programování pomocí datových toků (DFP) je přístup k programování, kdy program se moduluje v podobě orientovaného grafu datových toku mezi operacemi, podobně diagramu datových toků.

DFP se používá jako základní paradigma pro řešení problémů, především protože poskytuje přirozenou vizuální reprezentaci problému a umožňuje jednoduchou paralelizaci. Navíc toto paradigma je jednoduché na pochopení.

Tento princip je velice používán při modelování různých procesů. Většina databázových nástrojů, jako například Microsoft SQL Server Integration Services nebo Informatica PowerCenter jsou také postavené na DFP principu.

Čerpáno z [2, 3].

1.3.1.1 Funkcionální programování

Funkcionální programování je programovací paradigma, ve kterém se výpočet rozumí jako vyhodnocení matematických funkcí. Tento typ programování je založen na lambda-kalkulu, a většina funkcionálních jazyků může být považováno za jeho nadstavbu.

Funkcionální programování přináší několik zajímavých konceptů, mezi které patří následující:

- First-class functions
- Pure functions
- Rekurze

Funkce jakožto first-class objekt znamená, že funkci lze použít jako argument a vracet jako výsledek jiné funkce.

Pure functions jsou funkce, které nemají žádné vedlejší efekty. Jsou závislé jenom na vstupních parametrech a jenom vracejí výsledek výpočtu. Pure functions mají užitečné vlastnosti, jako například pokud není žádná závislost mezi dvěma funkci, můžeme prohodit pořadí jejích vyhodnocování a nebo paralelizovat.

Rekurzí je řešeno opakování v funkcionálním programování, kde funkce vyvolává sama sebe, čímž v podstatě nahrazuje cyklus.

Celkově použití funkcionálního paradigma dělá kód jednodušší na pochopení, a taky na programování, podporuje znovupoužitelnost kódu a modularitu.

Čerpáno z [4].

1.3.1.2 Logické programování

Logické programovací paradigma je založeno na použití matematické logiky. Matematická logika je velíce rozsáhlá, a pro tuto práci je zajímavá především predikátová logika. Pro ověřování podmínek je důležité formulování logických výroků. Nejdřív se formuluje logický výraz, které je potřeba ověřit pro množinu objektu, a dále při výpočtu probíhá jeho vyhodnocení, pro určité ohodnocení vstupních proměnných.

1.4 Základy doménově-specifického jazyka

Pokud již máme definované konkretní struktury, se kterými bude potřeba pracovat, a jsou specifikované operace, které budou prováděny, je třeba zamyslet se nad možností nadefinovat a použít doménově-specifický jazyk.

1.4.1 Doménově specifický jazyk

Doménově specifický jazyk (DSL) je programovací jazyk, který je zaměřen na vymezenou, konkrétní problémovou doménu. Na rozdíl od běžného programovacího jazyka DSL nepotřebuje řešit všechny existující problémy, ale zaměřuje se na jejich určitou podmnožinu, což je pravě potřeba při realizaci tohoto projektu. Navíc DSL přináší řadu výhod, mezi které patří:

- oddělení realizace od řešení problému – uživatel nemusí přemýšlet jak provést realizaci, ale zaměřuje se rovnou na řešení konkrétního problému

- lepší modifikovatelnost – je snazší porozumět programovému kódu, najít chybu a kód modifikovat
- zvýšení opětovného použití kódu – pro řešení podobných úloh mohou být použitý stejné funkce
- otevřenost pro uživatele – jednodušší způsob práce s DSL umožňuje podílet se na rozvoji aplikací i lidem bez hlubokých znalostí programování
- zjednodušení komunikace – dostupnost jazyka podporuje porozumění mezi doménovými experty a vývojáři

Čerpáno z [5].

1.4.2 Datové typy

Pro problematiku této práce jsou použity následující datové typy:

- Tree – typ reprezentující strom,
- Node – uzel stromu. Má jméno a hodnotu, obsahuje kolekci atributů, může mít potomky,
- Attribute – atribut. Má jméno a obsahuje páry klíč-hodnota,
- AttributeKey – klíč atributu,
- AttributeValue – hodnota atributu,
- Symbol – klíčové slovo (označení, jméno, uzlu, atributu, apod.),
- String – řetězec znaků,
- Integer – celé číslo,
- Any – libovolný typ,
- Function – speciální typ, který představuje soubor operací, jiný název lambda funkce.

Pro definici seznamu hodnot je vhodné použit kolekce:

- Col – seznám objektů, jednoduchá kolekce.
- Set – množina. Tento typ kolekci neobsahuje duplicity.
- Map – asociativní pole, kde klíče jsou unikátní a hodnoty mohou mít duplicity

1.4.3 Základní komponenty

Doménově specifický jazyk je postaven na principu programování pomocí datových toků, tudíž bude obsahovat všechny jeho prvky:

- uzly
- porty
- hrany

Uzly jsou základním elementem, jejich účelem je zpracování dat ze vstupu a předání výsledků na výstupu z uzlu. Výpočet v uzlu by měl proběhnout jako atomická operace. Uzly získávají a posílají data přes porty.

Porty představují rozhraní uzlů a jejích spojení s okolím. Porty můžeme rozdělit podle směru toku dat na vstupní a výstupní. Nakonec jsou mezi sebou porty propojeny pomocí hran.

Hrany propojují uzly přes jejich porty. Veškerá data v diagramu protékají přes hrany. Vznik dat v hraně může způsobit aktivaci uzlu (provedení výpočtu). Výpočet je proveden poté, co do uzlu dorazí data ze všech vstupních hran.

1.5 Jazyk pro vyhledávání ve stromě

Jednou ze základních operací, která se používá v podnikových pravidlech je vyhledávání. Vyhledávání se bude provádět nad obecnou strukturou zdrojových souboru, která je představena syntaktickým stromem.

Pokaždé, když je potřeba najít určitou hodnotu nebo kolekci hodnot ve stromové struktuře, potřebujeme postupně aplikovat jednu z funkci, a to buď konverzi, a nebo filtraci. Z toho vyplývá potřeba mít minimálně dva druhy uzlů: filtrovací a konverzní. Zřetězením filtrovacích a konverzních uzlů (aplikací nové funkce na výsledek předchozí) je možné docílit požadovaného výsledku.

Pro operování nad kolekcí hodnot se použije složený uzel, který bude obsahovat uzel iterátor a uzel s lambda funkcí pro provedení operace.

1.5.1 Filtrace

Filtrace je operace, která vytváří podmnožinu prvků na vstupu. Jeden vstupní datový tok je vždy lambda funkce vracející pravdivostní hodnotu a druhý jsou filtrované prvky. Tyto mohou být:

- kolekce, výstupní datový tok je potom typu kolekce, která je podmnožinou kolekce na vstupu
- strom, výstupní datový tok je kolekce podstromů stromu

Sémantika operace filtrace je následující: vstup je prvek po prvku filtrován přes logickou formuli: prvky, pro které je formule vyhodnocena jako true, jsou zahrnuty do výsledku, prvky, pro které má hodnotu false, nejsou do výsledku zahrnuty.

Čerpáno z [2].

1.5.2 Konverze

Konverze je operace, kdy jeden či více vstupních datových toků typu atomická hodnota či kolekce je převeden na jinou atomickou hodnotu či kolekci.

Konverze typu „Tree → Node“

- (Tree → Node) – převod stromu na kořen

Konverze typu „Node →“

- (Node → Node) – převod uzlu na potomky
- (Node → String) – převod uzlu na označení uzlu
- (Node → String) – převod uzlu na jeho obsah
- (Node → Col of Attributes) – převod uzlu na kolekce atributů

Konverze typu „Attribute →“

- (Attribute → AttributeKey) – konverze atributu na jeho klíč
- (Attribute → AttributeValue) – konverze atributu na jeho hodnotu

Čerpáno z [2].

1.6 Jazyk pro ověřování podmínek

Po nalezení potřebných hodnot můžeme provést kontrolu, zda hodnoty odpovídají očekávaným hodnotám, definovaným v podnikovém pravidle.

Ověřování předpokládá dvě složky: podmínku nebo výraz, který je potřeba ověřit, a samotné objekty, na kterých se bude ověření provádět. Podmínky jsou založeny na výrazu, který je aplikován na vyhledané prvky. Výraz je formulován ve zjednodušené predikátové logice. Objekty pro ověřování jsou většinou výsledkem vyhledávání.

Obecnou variantou ověření podmínky je ujištění, zda konkrétní výraz je platný pro každý (některý a nebo žádný) prvek z kolekce. V takovém případě ověřování připomíná filtraci, ale s jedním rozdílem: výsledkem by měla být odpověď „ano“ nebo „ne“, místo podmnožiny vstupní kolekce. Odpověď ale může být odvozena od výsledku srovnání vstupní kolekce a výstupní kolekce po filtraci. Záleží, zda jsou stejné nebo se liší.

1.7 Jazyk pro formulace úprav stromu

Kromě vyhledávání ve stromu a ověřování podmínek existuje třetí případ využití podnikových pravidel. Jedná se o optimalizaci, zjednodušeni nebo úpravu obecné struktury, se kterou interpret pracuje, t.j. jsou to úpravy syntaktického stromu.

1.7.1 Úpravy v kontextu DFP

Možnost úpravy struktury objektů v rámci programování pomocí datových toků je logický požadavek, který se na první pohled příliš neliší od vyhledávání nebo ověřování podmínek, a přináší navíc jen uzly s jinými funkcemi. Částečně je toto tvrzení pravdivé, ale existuje mnoho případů, které je potřeba brát v potaz v případě úprav obecně, a obzvlášť u úprav stromové struktury.

1.7.1.1 Úpravy a funkcionální přístup

Koncept DFP je těsně spjat s konceptem funkcionálního programování, jedním z aspektu kterého jsou tzv. pure functions. Pokud chceme využít jednu z výhod DFP - jednoduchou a přirozenou paralelizaci problému - je potřeba zachovat funkcionální přistup. Z toho vyplývají dva problémy:

- využití funkconálních uzlů místo stavových (stateful)
- přípustnost jenom neměnných (immutable) datových toků

Tyto problémy dříve nebyly zohledněny hlavně kvůli tomu, že v případě vyhledávání nebo ověřování podmínek se datové toky neměnily, probíhala pouze transformace dat ve smyslu převodu (konverze) bez úprav.

Využití funkconálních uzlů místo stavových (stateful)
Požadavek na použití pouze bezstavových uzlů je splnitelný jen do určité úrovně. Pro běžné úpravy, vytvoření a odstranění uzlů můžeme navrhnout strukturu, která bude představovat funkci s jedním až několika vstupy, a vyhodnocení bude provedeno až v momentě, kdy jsou data vložena do všech portů. Situace se mění, pokud chceme provádět složitější úpravy.

Uvádím příklad, kdy by bylo obtížné se vyhnout použití vnitřního kontextu uzlu. Je potřeba vytvořit uzel, který bude mít následující charakteristiky:

- vstup: kolekce řetězců, přičemž mnohé z nich jsou stejné
- výstup: kolekce řetězců bez duplicit. Duplicitní záznamy jsou přejmenovány na `jménoN`, kde jméno je předchozí řetězec a `N` je pořadové číslo od dvou do nekonečná

V takovémto případě za použití přímočarého návrhu bude uzel mít netriviální výpočetní funkci a bude nezbytné uchování interního kontextu. Byl by to jeden uzel s jedním vstupem a výstupem. Uzel by musel mít interní mapu, kam by se ukládaly řetězce a jejich počet. Pak pro každý řetězec by bylo potřeba navýšit počet výskytů v mapě a použít tuto hodnotu pro přejmenování. V případě pokud takový řetězec v mapě ještě není obsažen, bylo by potřeba jej uložit a vrátit původní hodnotu.

Řešení tohoto problému je možné vymyslet i bez změn použití kontextu, ale řešení bude složitější na pochopení, jelikož bude obsahovat sofistikovaný design a bude časově a výkonnostně náročnější. Uzel by mohl být představen složenou komponentou, která obsahuje iterační a složený funkční uzel. Složený funkční uzel by pak obsahoval výpočetní a modifikační uzly. Pro každý prvek v kolekci ve funkčním uzlu by bylo potřeba spočítat, kolik je před ním stejných řetězců.

Modifikační uzel by měl vytvořit novou hodnotu přidáním počtu opakování k původní hodnotě. Složitost v tomto případě bude kvadratická oproti případu přímočarého postupu, kde je lineární.

Nejspíše bude počet případů s podobnými problémy relativně nízký, a pokud takový případ nastane, bude potřeba obětovat možnost paralelizace při výpočtu a nebo zkomplikovat návrh řešení.

Přípustnost pouze neměnných (immutable) datových toků
Čistě funkcionální přístup předpokládá existenci pouze neměnných objektů a v případě úprav také vytvoření nového objektu s novou strukturou. V extrémním případě, pokud tento princip nebude splněn, problém nastane ihned u druhého ze dvou uzlů, do kterých tečou data ze stejného zdroje. Jednoduše se může stát, že po úspěšné úpravě prvního uzlu, uzel stromu již nebude mít obsah (potomka, atribut a nebo nebude existovat sám o sobě), kvůli čemuž úprava druhého uzlu skončí chybou. Daný problém nastane již v sekvenčním zpracování bez paralelizace.

Nejlepším řešením je v tomto případě držení se čistě funkionálního přístupu bez úprav datových toků, což je u úprav stromové struktury velmi problematické.

1.7.1.2 Úpravy stromové struktury

V závislostí na implementaci stromová struktura může obsahovat více nebo méně vazeb, ale vždy zde existuje vazba na předka a potomky, pokud jsou přítomny. V natolik provázané struktuře je poměrně obtížné provádět změny bez porušení funkcionálního přístupu. Jednodušší cestou je vytvoření kopie s novou, již změněnou strukturou, což je u velkých stromů náročné na výkon a také vždy paměťové náročné.

Při tvorbě nové kopie po každé změně je vždy potřeba vytvářet nový podstrom od kořene do úrovně uzlu, ve kterém došlo ke změně, uzly s větší hloub-

kou stačí zkopírovat. Tím pádem po každé změně by vznikaly velké struktury v paměti, tudíž by prostředí muselo mít dostatečné kapacity, aby umožnilo průběh takového algoritmu.

Alternativním řešením je možnost použití již hotové knihovny, která umožní pracovat s neměnnými (immutable) stromy efektivněji. Jenže veškerá taková řešení jsou založená na uchování interní proměnlivé struktury a tudíž zmenší potřebu paměti a výkonu jen ve velmi malé míře.

Copy-on-write stromy
Copy-on-write struktura je určena pro zachování neměnnosti objektů s menší režií. V takovém stromě jsou odkazy uspořádány pouze směrem dolů, tedy každý předek ví o svých potomcích. Pak při potřebě vytvořit kopii tohoto stromu stačí jen zkopírovat cestu od kořene do změněného uzlu, ostatní uzly jsou napojeny pomocí odkazů na původní strom. Při takovém způsobu kopírování cest se výrazně snižuje paměťová spotřeba oproti způsobu kopírování celého stromu. Toto řešení se často využívá při implementaci knihoven neměnných objektů. Čerpáno z [6].

Multiversion concurrency control
Multiversion concurrency control (MCC nebo MVCC) je další způsob zachování imutability, který se primárně používá v databázích. Jedná se o možnost verzování dat, kdy každá změna vytváří novou verzi, čímž zachovává staré verze beze změn. Tento princip se používá při implementaci Software Transactional Memory (STM) v jazyce Closure.[7] Rozdíl oproti databázím je v tom, že verzování neprobíhá v databázových tabulkách, ale v objektech paměti. Tento princip ale musí byt zaveden na úrovní jazyka a je obtížně použitelný v rámci jedné jednotky nebo knihovny.

1.7.2 Základní operace

Pokud se vrátíme k jednoduššímu pohledu, jazyk pro úpravy stromu jen rozšiřuje doménově-specifický jazyk o nové uzly. Tyto uzly mají stejnou strukturu, obsahuji minimálně jeden vstupní a výstupní port a aktivují se až po přítoku dat do všech vstupních portů. Základní uzly provádí operace nad stromem dle principu CRUD:

- create (vytvoř)
- update (změň)
- delete (odstraň)

Operace můžeme provádět v jakékoliv častí stromu. Můžeme měnit, vytvářet a odstraňovat uzly, atributy nebo jejích hodnoty.

Pro uzly jsou dostupné následující operace:

- `create-node(String): Node` – vytvoření nového uzlu, parametr je označení (tag)
- `update-node(Node, String): Node` – změna obsahu uzlu
- `add-attribute(Node, Attribute): Node` – přidání nového atributu k uzlu
- `replace-node(Node, Node): Node` – náhrada jednoho uzlu jiným
- `remove-node(Node): Node` – odstranění uzlu, výsledkem je předek uzlu

S atributy umíme provést tyto operace:

- `create-attribute(String, String): Attribute` – vytvoření atributu s klíčem a hodnotou
- `update-attribute(Attribute, String): Attribute` – nastavení nové hodnoty atributu
- `delete-attribute(Attribute)` – odstranění atributu

1.7.3 Aplikace úprav stromu

Aplikace úprav bude probíhat stejně jako konverze při vyhledáváni. Rozdíl ale bude ve výsledku, který bude představovat buď úplně nový uzel a nebo změněný starý.

V případě potřeby úpravy kolekce hodnot se analogicky použije iterační uzel a lambda funkce, obsahem které bude konkrétní uzel pro úpravu. Konečný výsledek úpravy kolekce je pak tvořen postupnou úpravou každého z jejích elementů.

Každý uzel po úpravě bude vracet hodnotu, a to buď nově vytvořený, upravený, a nebo rodičovský uzel, v případě odstranění uzlu ze stromu. Tím se zachová možnost řetězení komponent a pohodlnějšího návrhu pravidel.

1.7.4 Úrovně složitosti úprav

Ne všechny modifikační operace nad stromem jsou stejně složité. Dá se je rozřadit do určitých tříd složitostí podle počtu a složitostí jednotlivých operací úprav například takto:

1. úroveň – nahrazování konstantou
    - náhrada klíčového slova `cv` za klíčové slova `create view`
2. úroveň – nahrazování hodnotami v daném uzlu
    - náhrada spojení pomocí `||` dvou řetězců za spojený řetězec

3. úroveň – nahrazování hodnotami nacházející se jinde ve stromě
    - náhrada číselných referencí v `order by` a `group by` za odkazované aliasy/sloupce
4. úroveň – dynamická tvorba výsledku a držení vnitřního kontextu
    - náhrada rekurzivní struktury `decode` za `case`
    - unifikace aliasů přes celý strom

Definováno v [8]

Jak je vidět, čtvrtá úroveň složitostí úprav již vyžaduje použití stateful uzly a uchování vnitřního kontextu, což je nežádoucí stav v případě paralelizaci. Takové úpravy nebudou řešený v rámci této práce, protože se jedná o speciální případy. Během realizace interpretu budou navržený a implementované jen obecné, univerzální komponenty pro modifikace, kterými lze pokryt první tři úrovní úprav. Specifický uzly budou implementované v budoucnu jen pro konkretní požadavek.

1.8 Analýza vhodných technologií

V této kapitole se zaměřím na analýzu technologií, které by byly vhodné pro implementaci. V současnosti je k dispozici velká spousta technologií, a pro jednodušší rozhodnutí je potřeba si stanovit určitá kritéria, aby se co-nejvíce zmenšil počet vhodných kandidátu.

1.8.1 Kritéria pro vyhodnocování

Pro vyhodnocení byla stanovená řada nejdůležitějších kriterii, která by mohla omezit veškeré technologii na relativně malou množinu, vhodnou pro porovnání. Mezi tato kritéria patří:

- paradigma
- zralost
- podpora/komunita
- dostupnost knihoven
- použitelnost

Čerpáno z [2].

1.8.1.1 Paradigma

Zvoléné paradigma by mělo umožnit rychlý a efektivní vývoj projektu, tvořit modulární strukturu a podpořit znovupoužitelnost kódu.

1.8.1.2 Zralost

Nové ne vždy znamená dobré, proto je lepší zvolit vyspělou a vyzkoušenou technologii. Navíc cílový segment aplikaci (bankovnictví, pojišťovnictví.telekomunikace, velké korporace obecně) jsou konzervativní a často přijímají jen ověřené technologie.

1.8.1.3 Podpora/komunita

Néma cenu používat technologii, která brzo zanikne. A porto technologie by se měla podporovat a rozvíjet se zastřešující společností a komunitou. Je velká pravděpodobnost, že v rámci projektu bude třeba řešit různé speciální situace, což je další důvod potřeby existence aktivní a velké komunity pro danou technologii.

1.8.1.4 Dostupnost knihoven

Použití již existující knihovny s potřebnou funkcionalitou je vždy nejlepší řešení. Hotová řešení je již ověřeno, otestováno a často i funguje mnohem efektivnější, než to, který by bylo vytvořeno vlastními silami.

1.8.1.5 Použitelnost

Pro úspěšný vývoj a budoucí údržbu aplikaci je velíce důležitá snadná testovatelnost kódu, možnost rychlého a efektivního ladění, dostupnost pokročilých vývojových nástrojů. To všechno děla programovácí jazyk použitelným.

1.8.2 Volba programovacího jazyka

Kromě uvedených výše kriterií, volba technologií pro DFP interpret byla omezená technologií celé aplikaci Manta Checker. Jelikož pro celou aplikaci byl zvolen jazyk Java, interprete by také měl implementován v jednem z jazyku, které mohou byt spuštěný na Java Virtual Machine (JVM). Mezi tyto jazyky patrí:

- Clojure
- Groovy
- Java
- JavaScript
- Scala

Na začátku, v rámci Proof of Concept (PoC), pro implementaci systému pro kontrolu podnikových pravidel byl zvolen jazyk Clojure. Ale jak uvádí autor

PoC: *„Po praktické zkušenosti bylo nutno tuto volbu přehodnotit: řešení nevyhovovalo v kategorii „Použitelnost“. Problémy nastávaly s laděním kódu v důsledku kompilace Clojure na JVM – drobné chyby v kódu měly za následek kryptické hlášky překladače a hledání a odstraňování chyb bylo značně problematické a pracné.“*[8].

Byla hledaná alternativa, která by vyhověla v tomto i ostatních kritériích. Takovou alternativou se ukázal jazyk JavaScript. Hlavní výhodou jazyka, která odlišuje jej od zbytku ze seznamu, je přímá interpretace zdrojového kódu prohlížečem, kvůli čemu při vývoji odpadá krok komplikace.

1.8.2.1 JavaScript

JavaScript (JS) je multiplatformní, objektově orientovaný skriptovací jazyk. Jazyk vznikl v roce 1995 a nejvíce se používá jako interpretovaný programovací jazyk pro webové stránky. Název jazyka připomíná známý programovácí jazyk Java, ale ve skutečností byl to jenom marketingový tah, a tyto jazyky spojuje jen podobná syntaxe.

JS je dynamický typovaný, což přináší určitou flexibilitu. Bez ohledu na to že jazyk byl tvořen jako objektově orientovaný, obsahuje funkcionální prvky, takové jak first-class functions, anonymní funkce a uzávěry (closures). Jazyk je velíce výstižný a umožňuje vytvářet malé moduly a používat je dále v kódu.

JS je vyspěly jazyk, které se používá po celém světě v každé webové aplikaci. JavaScript je možné použít i na straně serveru, například pomoci platformy Node.js. Je možné jej spouštět i v libovolné aplikaci běžící na JVM pomocí JavaScriptového enginu Rhino[1] nebo Nashorn.

Kvůli velké popularitě jazyka existuje obrovská komunita uživatelů. Vznikají zajímavé odvětví jazyka jako TypeScript[2], který děla JavaScript statický typovaným, a nebo CoffeeScript[3], který přináší „syntaktický cukr“ a zpříjemňuje použití jazyka. Samozřejmě pro natolik používaný jazyk vznikla velká spousta knihoven a existuje velíce mnoho připravených řešení.

Diky přímé interpretaci jazyka prohlížečem, v JavaScriptu se velíce dobře vyvějí, a ladění kódu je příjemné a rychlé. Přítomnost spousta frameworku pro testování zdrojového kódu ještě více zvyšuje použitelnost jazyka.

1.9 Jazyk pro definici pravidel

Pravidla, který bude vyhodnocovat interpret musí mít nějakou podobu, ve které budou uložené a předané pro zpracování. Samozřejmě to musí byt nějaký strojově zpracovatelný formát, který může byt přečten pomoci algoritmu. Existuje řada formátu, které lze použit pro uchování a přenos dát:

[1] https://www.mozilla.org/rhino/
[2] http://www.typescriptlang.org/
[3] http://coffeescript.org/

- Fixed-width
- CSV
- XML
- JSON

1.9.1 Fixed-width file formát

Fixed-width file formát (FWF) představuje formát, kdy v rámci jednoho souboru je definováno několik sloupců, a je daná předem šířka a index začátku každého sloupce. Je to jeden z nejjednodušších formátů, ale má řadu nevýhod:

- je potřeba vědět na začátku velikosti dat
- nejde dynamický měnit počet a velikost sloupců
- jakmile se objeví hodnota větší než sloupec dojde k chybě a potřebě měnit strukturu
- pro definici různých vlastností je potřeba mít několik souborů

1.9.2 CSV formát

CSV (Comma-separated values, hodnoty oddělené čárkami) je další jednoduchý formát pro výměnu dat, kde každý hodnota v řádku je oddělená nějakým oddělovačem, typický čárkou. Pokud je potřeba mít hodnoty, které obsahují oddělovač, tyto hodnotu mohou být uzavřeny do dalšího speciálního symbolu, typický uvozovek. Je možné také mít v rámci jedné hodnoty dynamické seznamy, kde elementy jsou oddělené například středníkem. Tento formát na rozdíl od FWF již může fungovat dynamický, ale pořad jeden soubor dává jeden pohled na objekt a pro definici různých typu objektů je potřeba použit více souborů.

1.9.3 XML

XML (Extensible Markup Language) je obecný značkovací jazyk, ve kterým lze nadefinovat vlastní značkovací jazyk pro své účely. XML je velíce flexibilní, má pohodlnou pro zpracování stromovou strukturu a vysoký informační obsah. Pro hodně technologie XML je standardní formát pro výměnu informací. Na rozdíl FWF a CSV formátu, všechny objekty a závislosti lze popsat v jednom souboru.

1.9.4 JSON

JSON (JavaScript Object Notation) je další formát nezávislý na počítačové platformě, určený pro přenos dat. Data mohou byt organizována v polích nebo agregována v objektech. JSON lze považovat za jednodušší alternativu k XML. Formát je dobře čitelný člověkem a strojově zpracovatelný. Navíc je velíce používaný v webových aplikacích, protože může byt zpracován JavaScriptem bez dalších konverzí.

1.9.5 Definice pravidel v interpretu

Pro zápis pravidel byl zvolen formát JSON, který zcela odpovídá řadě požadavků: jednoduše čitelný i zapisovatelný člověkem a snadno analyzovatelný i generovatelný strojově. Možnosti formátu jsou postačující a použití silnějšího formátu XML by nepřineslo nic navíc. Další výhoda použití JSON v tomto případě je že obě strany (interpret na jedné straně a webová aplikace pro tvorbu a úpravu pravidel na druhé), které budou operovat s daty jsou aplikace napsané v jazyce JavaScript, což výrazně ulehčí zpracování pravidel.

Kapitola 2

Návrh

V teto kapitole bude popsán návrh jednotlivých části modulu a jejích propojení. Zaprvé se budeme věnovat základním prvkům a následně i složitějším elementům.

2.1 Základní stavební bloky

Návrh základních stavebních bloků vyplývá z výsledků analýzy, zvolených technologií a paradigmatu. Při programování pomoci datových toků pracujeme s DFP diagramem neboli grafem, na kterém jsou různé uzly, spojené mezi sebou hranami přes porty.

2.1.1 Uzly

DFP je graf tvořený uzly a orientovanými hranami. Uzly mohou mít žádný až mnoho vstupních a výstupních portů. Uzly jsou dělené na

- atomické uzly
  - literální uzly (literály) – Mají pouze výstupy, pouze vrací hodnotu. Výstup je typicky jeden, ale obecně jich může být i více.
  - výpočetní uzly – Provádí určitou transformaci vstupů na výstupy.
  - modifikační uzly – Je druh výpočetních uzlů, které navíc mají vedlejší efekt (např. aktualizaci hodnoty atributu).
  - uzly s vedlejším efektem – Mají pouze vstupy, žádné výstupy a provádějí pouze určitý vedlejší efekt (např. zápis do reportu).
- složené uzly (komponenty) – uzly, které zapouzdřují jeden nebo více provazaných atomických uzlů

Každý uzel má svůj prototyp. Prototyp uzlu má tuto strukturu:

- `ID` – identifikátor, musí být unikátní v rámci balíčku.
- `description` – nepovinný popis
- `input_ports` – názvy a typy vstupních portů
- `output_port` – názvy a typy výstupních portů

Uzel má tuto strukturu:

- `ID` – identifikátor
- `nprototype` – prototyp uzlu
- `parent` – rodičovský uzel v případě, že je uzel obsažen v komponentě.
- `ports` – seznám portů

Komponenta má navíc:

- `children` – uzly obsažené v komponentě.

Komponenta navíc představuje jmenný prostor (namespace) pro své uzly. Čerpáno z [2].

2.1.2 Porty

Porty jsou určený pro příjem/odeslání pro uzly. Každý port má tyto složky:

- `id` – identifikátor, který musí být unikátní v rámci uzlu, ke kterému patří.
- `description` – nepovinný popis
- `role` – role portu pro uzel, ke kterému patří (vstupní nebo výstupní)
- `type` – typ hodnoty portu
- `value` – aktuální hodnota v portu

2.1.3 Hrany

Hrana představuje sebou objekt, který obsahuje jenom prototyp a provazuje dva uzly mezi sebou pomocí jejích portů.

2.2 Wrapper pro práci s AST

Během návrhu a vývoje DFP interpretu, paralelně probíhá vývoj a implementace rozhraní pro práci s AST. Jelikož jeho výsledná podobá není známa a je velká pravděpodobnost budoucích změn, bylo rozhodnuto vytvořit mezivrstvu (wrapper) pro práci s AST.

Zavedení teto třídy umožní práci s různými implementacemi a typy uzlů. Při přechodu od jedné implementaci uzlu do druhé, mění se jen vnitřní implementace wrapperu nebo přidává se nový, ale zbytek interpretu se zůstává neměnný, jelikož používá stejné rozhraní wrapperu.

Wrapper třída obsahuje metody pro základní dotazovací operace nad AST:

- Získání předka daného uzlu - `getParent(Node): Node`
- Získání potomků daného uzlu - `getChildren(Node): Node[]`
- Zjištění označení uzlu - `getTag(Node): String`
- Získání obsahu uzlu - `getContents(Node): String`
- Zjištění atributu uzlu - `getAttributes(Node): Attribute[]`
- Získání kořene stromu - `getRoot(Node): Node`

Metody pro konverzní operace:

- Převod uzlu na atributy - `nodeToAttributes(Node): Attribute[]`
- Převod atributu na klíč - `attributeToKey(Attribute): AttributeKey`
- Převod atributu na hodnotu - `attributeToValue(Attribute): AttributeValue`

Metody pro modifikaci uzly:

- Vytvoření uzlu - `createNode(String): Node`
- Nastavení hodnoty uzlu - `setNodeValue(String): Node`
- Výměna uzlu - `replaceNode(Node, Node): Node`
- Odstranění uzlu - `deleteNode(Node): Node`
- Přidání atributu - `addNodeAttribute(Attribute): Node`
- Aktualizace atributu - `updateNodeAttribute(Node, Attribute): Node`
- Odstranění atributu - `deleteNodeAttribute(Node, Attribute): Node`

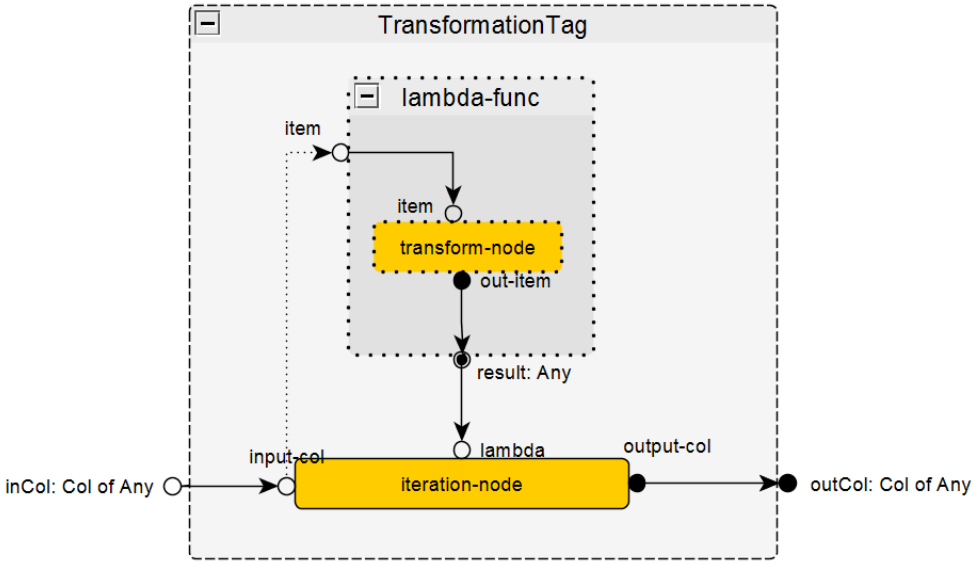
Metody pro hromadné operace:

- Filtrování ve stromě - `filterOnTree(Node, Function): Node`

- Filtrování seznamu - `filterOnCollection(Any[], Function): Any[]`
- Transformace seznamu - `transformOnCollection(Any[], Function): Any[]`

2.3 Vyhledávání ve stromě

Při programování pomoci datových toků se velíce těžko zobrazují cykly, které by při funkcionálním programováni ani neměli používat. Proto při provedení operací nad kolekcí se budou používat složené uzly, uvnitř kterých bude iterační uzel a funkční blok (neboli lambda komponenta).



Obrázek 2.1: Příklad složené konverzní komponenty

2.3.1 Iterační uzel

Iterační uzel tvořen dvěma vstupními porty a jedním výstupním, přičemž jeden ze vstupním portu očekává funkci.

Vyhodnocovací funkce iteračního uzlu volá jednu z funkci wrapperu pro hromadné operace (filtrování nebo konverzí), kam jako argumenty dosazuje vstupní kolekci a funkci. Výsledek hromadné operace pak předává do výstupního portu.

2.3.2 Funkční blok

Lambda komponenta neboli funkční blok je složená komponenta, která umožňuje provádět výpočty, konverze a modifikace, a zapouzdřit celý diagram do

jedné funkce.

Komponenta má minimálně jeden až nekonečně vstupních portů, jeden z kterých je lambda port. Lambda port je určen jako vstup do vnitřního obsahu komponenty, propaguje vstupní argument výsledné funkce do zapouzdřeného řetězce. Po nastavení hodnoty do lambda portu, začíná se vyhodnocování a výpočet vnitřního diagramu.

Jeden výstupní port na obrázku má duální charakter a fyzický reprezentuje dva porty. První výstupní port je ten, kam se nastavuje funkce. Hodnota tohoto portu se nastavuje když všechny vstupní porty, kromě lambda portu, budou připravené, t.j. do portu přijde nenulová hodnota. Druhý port je lambda výstupní port, kam se nastavuje výsledek vyhodnocení vnitřního obsahu komponenty. Výstupní hodnota lambda portu je pak návratovou hodnotou celé lambda funkce, která je výsledkem vyhodnocení pro konkretní vstup do vstupního lambda portu.

2.3.3 Zřetězení komponent

Pro nalezení určitého uzlu nebo atributu uzlu ve stromě může nestačit použití jedné složené komponenty, proto se předpokládá, že se komponenty budou zřetězovat. Na nejvyšší úrovně pohledu v diagramu bude sada složených komponent, propojených pro postupnou transformaci dat od kořene stromu na prvním vstupu do kolekce hodnot potřebného typu na výstupu.

2.4 Ověřování podmínek

Mechanizmus ověřování podmínek je postaven na stejném principu jak vyhledávání ve stromu. Za účelem ověření podmínky přidáme složenou komponentu, která nám dovede vyhodnotit výraz nad všemi elementy kolekce a vrátit výsledek.

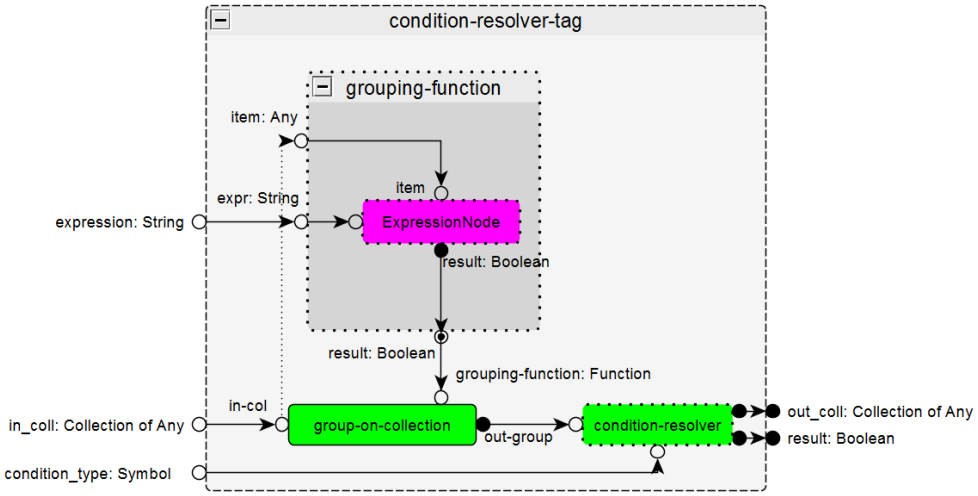
Vstupy komponenty (obrázek 2.2):

- `in_coll` – kolekce objektů které budeme ověřovat
- `expression` – výraz který chceme prověřit a aplikujeme na každý objekt kolekci pro ověření
- `condition_type` – podmínka, na základě které budeme vracet výsledek

Podmínka je hodnota z následujícího číselníku:

- `all-satisfy` – všechny elementy kolekci musí splňovat daný výraz
- `non-satisfy` – žádný element nesmí splňovat výraz
- `any-satisfy` – alespoň jeden element splňuje výraz

Výstupy komponenty (obrázek 2.2):



Obrázek 2.2: Komponenta pro ověřování podmínek

- `result` – Boolean hodnota výsledku vyhodnocení (true v případě že podmínka je splněná, jinak false).
- `out_coll` – kolekce objektů které nesplňují podmínku

Algoritmus výpočtu je následný. Pro každý element kolekce, iterační uzel `group-on-collection` spouští lambda funkci, která vrací pravdivostní hodnotu, na základě které uzel zařazuje element do příslušné skupiny. Tyto skupiny se předávají do rozhodovacího uzlu `condition-resolver`, které podle vstupní podmínky kontroluje skupiny elementů a generuje výsledky.

Grupovací funkce, či vnitřek tvoří jenom jeden výpočetní uzel, je jenom jedná z možností implementace. Obsah grupovací funkcí může byt mnohem složitější a lišit se od představeného návrhu. Je důležitě jenom zachovat návratový typ funkce.

2.5 Úpravy stromu

Modifikace neboli úpravy stromu navazují na předchozí sekci. Princip fungování modifikačních uzlů je stejný jako výpočetních, ale má jeden zásadní rozdíl – vedlejší efekt, t.j. tento uzel mění stav nebo strukturu stromu, se kterým pracuje.

2.5.1 Problémy a omezení

Při prvotním návrhu úprav se objevila sada problémů, které mohly výrazně ovlivnit návrh těchto uzlů. Mezi tyto problémy patří:

- merge & split

- kopírování stromu
- uchování výchozí struktury stromu

2.5.1.1 Merge & split

Sloučení a rozdělení různých částí stromu je největší z problémů.

Sloučení jedntolivých uzlů je relativně jednoduchá operace, ale změny nastávají když chceme sloučit celé stromy, neboli uzly s potomky. Při sloučení uzlů s potomky je potřeba zajistit aby výsledný strom měl správnou strukturu a obsahoval všechny potřebné podstromy. Jak se má zachovat interpret pokud je potřeba sloučit uzly se změnami v potomcích? Kterého potomka máme přenést do výsledného stromu? Jak zvolit správného potomka pokud množiny nebudou disjunktní?

Rozdělení stromu na podstromy není na první pohled problematické a používá se bez omezení například ve vyhledávání, když paralelně pracujeme z různými části stromu. Je to tak, protože víme, že se strom nebude měnit, tím pádem nenastane situace, že by strom byl v nekonzistentním stavu. Ale pokud víme, že dále nad těmi části budou probíhat úpravy, nastává situace, kdy musíme rozhodnout co budeme předávat – ten samy strom nebo jeho kopii?

2.5.1.2 Kopírování stromu

Problém kopírování navazuje na problém rozdělení stromu. Pokud se chceme vyhnout kolizím, je potřeba udělat hlubokou kopii, a předávat ji pro další zpracování. Mnohokrát se může stát, že by stačila mělká kopie, místo hluboké. Použití hluboké kopie pak způsobuje paměťovou neefektivitu a zbytečnou režii za běhu interpretu.

Druhé místo kde může byt potřeba kopírování jsou modifikační uzly. Je potřeba definovat, zda výsledek úpravy bude kopie stromu, nebo můžeme přímo upravovat strukturu, která přišla na vstupu.

2.5.1.3 Uchování struktury stromu

Otázka uchování struktury stromu vyplývá z předchozích problémů. V závislostí na tom, zda budeme kopírovat stromy nebo ne je potřeba uchovávat strukturu stromu někde zvlášť. Pokud navrhneme implementaci, kde velíce dráha operace kopírování bude nahrazená nastavením určitého příznaku uzlu, uchováním změn vedle nebo jiným způsobem, musíme zajistit uchování dat a přístup k nim. Musí vzniknout kontext, který obsahuje buď originální strukturu, nebo změny, a nebo oboje.

2.5.1.4 Vymezení funkčnosti interpretu

Každá z výše popsáných problémů byla posuzovaná v kontextu fungování celé aplikaci Manta Chceker a možných případů využití. Plyne z toho řada vymezení funkčnosti interpretu.

Základním předpokladem který vychází z byznys analýzy je že jeden DFP diagram bude obsahovat jenom jednu modifikační funkci, t.j. se bude provádět jenom jeden druh změn, například náhrada klíčového slova. Výsledkem tohoto předpokladu je jednoznačná odpověď na otázky k problémům merge & split, kopírování a uchování struktury stromu:

- Sloučení paralelně upravených stejných částí stromu se dělat nebude. Není potřeba provádět natolik složité operace pro jeden druh úprav.
- Rozdělení stromu se může provádět stejným způsobem jako při vyhledávání, protože dále nedojde k úpravě stejných uzlů v různých větvích diagramu.
- Nutnost kopírování zmizí.
- Uchování celé struktury stromu někde v kontextu není potřeba.

Tyto vymezení však nezabraňuji mít několik zřetězených modifikačních uzlů a postupně upravovat strom.

2.5.2 Modifikační uzel

Modifikační uzel je výpočetní uzel, který obsahuje minimálně jeden vstupní a jeden výstupní port. Po nastavení hodnot na všech vstupních portech můžeme začít vyhodnocovat tělo funkci. Zpravidla vnitřní funkce bude volat odpovídající funkci v objektu wrapper, který provede úpravu nad aktuální implementací AST.

Pro uzly, které mají více než jeden vstup je potřeba zajistit, aby se vyhodnocení začalo až po nastavení aktuálních hodnot do všech vstupních portů, což je velíce důležité v případě volání úprav více uzlů v cyklu. Proto je vhodné vynulovat hodnoty na vstupních portech po každé provedené úpravě.

2.6 Reporting

Jeden z požadavků aplikaci Manta Checker do které se zapojí DFP interpret je reporting. Je potřeba zapisovat informaci o výsledcích ověření podmínek do vnějšího rozhraní, které pak generuje report pro uživatele. Pro tyto účely je navržená reportovací komponenta.

Jelikož uživatele budou zajímat uzly, na kterých podmínka nebyla splněná, reportovací komponenta bude napojená na výstupní port složené komponenty, který vrací kolekci objektů, které nesplňují podmínku. Žádný další vstup ani

vystup pro tuto komponentu není potřeba. Pro každý prvek kolekce reportovací uzel vygeneruje zprávu do vnější rozhraní, kde uvede textovou chybovou hlášku a místo (konkretní uzel) na kterém ověření selhalo.

2.7 Jazyk pro definici pravidel

Pro zápis pravidel byl zvolen JSON formát, jako nejvhodnější v porovnání s ostatními (viz analýza 1.9). Dále je potřeba definovat způsob zápisu pravidel v tomto formátu. Bylo rozhodnuto použit přímočarý a nejjednodušší způsob definici pravidel, kdy definice elementů bude odpovídat jejích struktuře v interpretu.

Pravidla se zapisují ve formátu JSON s pevnou strukturou:

```
{
    components: [...],
    nodes: [...],
    edges: [...]
}
```

Kde `components` je pole definic komponent, `nodes` je pole definic použitých uzlů a `edges` je pole definic hran, které propojují uzly.

2.7.1 Definice komponent

Pokud chceme použit nějakou novou komponentu, která neexistuje v interpretu, musíme ji nadefinovat. Definice jedné komponenty je následující:

```
{
    component: "Component",
    name: "NewComponent1",
    ID: "new-component-1",
    nprototype: { ... },
    structure: {
        nodes: [...],
        edges: [...]
    }
}
```

Položky objektu znamenají:

- `component` – typ komponenty (obyčejná složená nebo lambda komponenta)
- `name` – název komponenty, který se následně plouživá při definicí uzlů
- `ID` – identifikátor komponenty
- `nprototype` – objekt s informací o rozhraní komponenty
- `structure` – objekt s definicí vnitřní struktury komponenty:

- `nodes` – pole uzlů použitých v komponentě,
- `edges` – pole hran, které propojují nadefinované uzly

Komponenta může mít v sobě jiné komponenty, t.j. vnoření komponent je přípustné. Pro použití odkazu na jinou komponentu je potřeba, aby tato komponenta byla již předem definovaná. Pro následné použití stačí jen odkaz na její jméno v definicí uzlů.

2.7.2 Definice uzlů

Definice uzlů se používá jak v definici pravidel, tak i v definici komponent. Každý uzel je nadefinován pomoci následujícího objektu:

```
{
    component: "KnownComponent",
    ID: "component-1",
    nprototype: { ... }
}
```

Kde `component` je název již existující nebo nově nadefinované komponenty, `ID` je identifikátor komponenty a `nprototype` je objekt s informací o portech, případně `ID` komponenty u literálů.

2.7.3 Definice hran

Definice hran se používá jak v definici pravidel tak i v definici komponent. Každá hrana je nadefinován pomoci následujícího objektu:

```
{
    component: "Edge",
    nprototype: {
        from: {node: "from-node-id", port: "from-node-port-id"},
        to: {node: "to-node-id", port: "to-node-port-id"}
    }
}
```

Kde `component` říká o tom, že se jedná o hranu (`Edge`), objekt `nprototype` obsahuje informaci o uzlech (`node`) a identifikátorech portů (`port`) ze kterého (`from`) a do kterého (`to`) jde hrana.

Kapitola 3

Realizace

V kapitole Realizace bych se chtěl věnovat implementaci samotné aplikace. Při implementaci jsem se držel návrhu a poznatku z analýzy. Abych neunavil čtenáře podrobným popisem každé třídy a metody, zaměřím se jen na nestandardní řešení a nejproblematičtější části implementace.

3.1 Jmenné prostory v JavaScriptu

Jmenné prostory (napespaces) lze považovat za logické seskupení jednotek kódu pod jedinečným identifikátorem. Na identifikátor lze odkazovat z jiných jmenných prostorů, a každý identifikátor může sám o sobě obsahovat hierarchii vnořených jmenných prostorů, neboli podprostorů.

Jmenné prostory jsou potřebné a vhodné k použití protože:

- zabraňuji kolizi s jinými objekty a proměnnými v globálním jmenném prostoru
- organizuji funkční bloky v aplikaci do skupin
- zjednoduší práci s kódem, zajistí lepší udržovatelnost

V JavaScriptu, na rozdíl od většiny jiných jazyků, není zabudovaná podpora jmenných prostorů, ale jazyk má objekty a uzávěry (closures), které mohou být použity k dosažení stejného efektu.

3.1.1 Základy jmenných prostorů

V větších aplikacích se nedá vyhnout rozdělení kódu do funkčních bloků. Pokud se to bude řešit pomocí jmenných prostorů, pomohou tyto návrhové vzory:

- Jediná globální proměnná
- Jmenný prostor tvořený předponou (Prefix namespacing)

- Zápis objektovým literálem (Object literal notation)
- Zanořené jmenné prostory (Nested namespacing)
- Bezprostředně vyvolané funkční výrazy (Immediately-invoked Function Expressions)
- Injektáž jmenných prostorů (Namespace injection)

Čerpáno z [9, 10, 11].

3.1.1.1 Jediná globální proměnná

Populární návrhový vzor, kde pro vytvoření jmenného prostor je zvolená jediná globální proměnná, jako primární objekt, který zapouzdřuje veškeré funkce a proměnné. Tento přístup ale nefunguje vždy, protože stejná proměnná může být již nadefinovaná jinde a tím dojde ke kolizi.

3.1.1.2 Jmenný prostor tvořený předponou

Další návrhový vzor, který částečně řeší předchozí problém je použití předpony pro veškeré objekty. Toto řešení snižuje pravděpodobnost výskytu stejných objektů, ale není to moc efektivní přístup. Zaprvé pořád je šance, že někdo použije stejné jméno. Zadruhé s růstem aplikaci se výrazně zvýší počet globálních objektů s nepřehlednými názvy, a tím se ztratí většina výhod použití jmenných prostorů.

3.1.1.3 Zápis objektovým literálem

Zápis objektovým literálem ve zjednodušené verzi je objekt, složený z párů klíč:hodnota, které jsou oddělené čárkami. Takový objekt můžeme rozšiřovat a přidávat další funkce. Tento vzor nezahlcuje globální prostor a vytváří uspořádaný kód. Na rozdíl od jediné globální proměnné na začátku se provádí kontrola, zda proměnná již není použita, a pokud ano, použije se pro další rozšíření.

3.1.1.4 Zanořené jmenné prostory

Tento vzor rozšiřuje předchozí o definici většího vnoření při použití. Proto ke všem vlastnostem předchozího vzoru se přidá ještě lepší uspořádání a menší pravděpodobnost kolizí.

3.1.1.5 Bezprostředně vyvolané funkční výrazy

Immediately-invoked Function Expressions (IIFE) je návrhový vzor, jehož základ je funkce, která je vyvolaná hned po její definici. Jelikož v JavaScriptu je interní kontext funkci viditelný jenom zevnitř, vnitřní funkce a proměnné se

můžou pak bezpečně používat. Navíc se dá říct, že takto vzniká samostatný modul. Standardně se do funkcí můžou dávat parametry, které se používají interně jako závislosti pro vnitřní implementaci. T.j. je to způsob jak se dá tvořit závislost mezi moduly. Existuje varianta IIFE, kde se jako parametr přidává objekt, do kterého je potřeba přidat požadované vlastnosti. Pomoci tohoto návrhového vzoru můžeme rozšiřovat jmenné prostory obalením včetně přidání nových metod do další takové funkce.

3.1.1.6 Injektáž jmenných prostorů

Injektáž jmenných prostorů je rozšíření IIFE, kde jako jmenný prostor se používá this objekt. Tímto způsobem se dá rozšiřovat i již nadefinovaný namespace a přidávat k němu podprostory. Nevýhodou ale je větší obtížnost než existující alternativy.

3.1.1.7 Použití jmenných prostorů v interpretu

Pro implementaci jmenných prostorů v rámci interpretu byl jako nejlepší varianta nejdřív zvolen návrhový vzor IIFE, který nejlépe odpovídal potřebám pro tvorbu modulární struktury, a v dnešní době se považuje za nejlepší praxe (best practice). Pak byl ale nalezen ještě lepší způsob zajištění modularity v podobě, který byl použit pro interní implementaci. Nicméně knihovny třetích stran jsou v podobě IIFE, tím pádem tento návrhový vzor je pořád použit v implementaci interpretu.

3.1.2 Modularita v JavaScriptu

Modularita aplikace je to nejdůležitější, co by mělo být výsledkem použití jmenných prostorů. Pod modulární aplikací rozumíme, že je to sada volně provázaných funkčních bloků, důsledkem čehož je snadná udržovatelnost a minimální závislost v kódu.

Pomoci návrhových vzorů jako zápis objektovým literálem nebo IIFE se dá definovat moduly, ale nejedná se o úplně přirozený způsob. Není jednoduchý způsob jak naimportovat moduly včetně závislostí plně automaticky, a nebo to vyžaduje ruční konfiguraci. Další věc je, že jejích definice probíhá v globálním kontextu, a je potřeba hlídat správnost jmen aby nedošlo ke kolizi.

Řešení které by umožnilo snadnou práci s moduly na úrovní jazyka se jenom vyvíjí, ale již jsou koncepty a knihovny, které se o řešení pokoušejí, a mezi ně patří Asynchronous Module Definition (AMD).

3.1.2.1 Asynchronous Module Definition

AMD je JavaScript specifikace, která definuje API pro definici modulů a jejich závislosti, a také možnost asynchronního načítaní v případě potřeby.

Koncept AMD předpokládá definici modulu určitým způsobem a použití některé z implementací knihovny Script Loader, který moduly načte a poskládá dohromady.

Základní metody pro práci s moduly jsou `define`, pro definici modulů a `require`, pro import závislostí.

`define` se používá pro definici pojmenovaného nebo nepojmenovaného modulu, a má následující signaturu:

```
define(
    module_id /*optional*/,
    [dependencies] /*optional*/,
    function /*definition function for instantiating the module
        or object*/
);
```

module_id je nepovinný atribut, který umožňuje vytvořit pojmenovaný modul. Pokud nebude vyplněn, bude vytvořen anonymní modul, který má větší přenositelnost. Je potřeba jej zadávat, pokud moduly budou zpracované non-AMD nástrojem.

Kolekce dependencies je odkaz na jiné moduly, na kterých tento modul závisí.

Třetí argument function je definice samotného modulu, která se používá pro vytvoření jeho instanci. Metoda require má stejnou signaturu, ale bez prvního parametru, a používá se v top-level třídách a nebo v modulu, o kterým se ví, že se na něj nebude odkazovat.

Čerpáno z [11].

3.1.2.2 RequireJS

Jedná z implementaci Script Loader, knihovny pro načítaní a spouštění modulu je RequireJS[4]. Tato knihovna je nejrozšiřenější z podobných implementací. Součástí knihovny je nástroj pro optimalizace r.js, který umožní:

- sjednotit řadu modulu do jednoho js souboru
- provést minifikaci výsledného skriptu
- spustit AMD aplikaci v Java

3.1.2.3 Almond

Jedná se o odlehčenou verzí RequireJS. Knihovna almond[5] má určitá omezení, ale je užitečná v případech kdy hraje roli objem, a je potřeba jen základní funkčnost RequireJS.

[4] http://requirejs.org/
[5] https://github.com/jrburke/almond

3.1.2.4 Realizace modularity v DFP interpretu

Pro realizaci modularity v DFP interpretu bylo rozhodnuto použít nejlepší způsob z existujících, což je řešení pomoci AMD. Hlavní výhody AMD oproti jiným návrhovým vzorům jmenných prostorů:

- poskytuje jasný způsob, jak definovat moduly
- čistý způsob, jak deklarovat moduly a závislostí mezi nimi
- nedochází k znečištění globálního jmenného prostoru
- umožňuje sjednotit více moduly do jednoho skriptu

Čerpáno z [11].

Při vývoji jako Script Loader byla použita knihovna RequireJS, protože je jednoduchá v použití a je dobře zdokumentovaná. Při generování zdrojového kódu interpretu byla místo ní použita odlehčená AlmondJS pro úsporu místa a paměti.

3.2 Proces vývoje

Byla škoda nevyužít možnost přímé interpretace JavaScriptu v prostředí prohlížeče, proto vývoj DFP interpretu probíhal duálním způsobem:

1. Vývoj a ladění v interaktivním prostředí prohlížeče
2. Generování dodávky modulu pro prostředí JVM a finální testování

3.2.1 Interaktivní JavaScript vývojové prostředí

Interaktivní vývojové prostředí prohlížeče umožňuje pohodlný vývoj, ladění a testování. Součástí prostředí jsou:

- textový editor nebo IDE pro úpravu zdrojového kódu
- testovací framework Jasmine
- webový prohlížeč s ladicími nástroji

Zdrojový kód jazyka JavaScript se ukládá do běžného textového souboru s rozšířením js. Proto se je dá editovat v běžném textovém editoru. Lepší ale je použit pokročilé IDE, které navíc umožňuje formátování, našeptávaní a analýzu kódu.

Jasmine je testovací JavaScript framework, který umožňuje jednoduché a pohodlné testování. Framework má přehlednou syntaxi, umožňuje automatické spouštění testu a výpisu stavu zásobníku v případě selhání testu.

Webový prohlížeč poskytuje prostředí, které přímo interpretuje JavaScript kód. Většina moderních prohlížečů má integrované ladicí nástroje a nebo podporuji pluginy jako populární Firebug. Tyto nástroje poskytují:

- Step-by-step interaktivní ladění
- konzoli pro logování a ladicí výpisy
- inspektor objektů

Vývoj JavaScript aplikací tím pádem je velíce pohodlný a splňuje veškeré požadavky kladené na vývoj softwarového produktu.

3.2.2 Generování dodávky pro prostředí JVM

Jelikož modul DFP interpretu se integruje do aplikací Manta Checker, napsané v jazyce Java a běžící na platformě JVM, modul se musí překompilovat z JavaScript do Java. Je nezbytné zajistit použití správných skriptů a implementací modulem rozhraní dle specifikaci Manta Checker.

3.2.2.1 AMD-feature

Kvůli duálnímu vývoji modulu vznikly dvě verze wrapper objektů pro práci s AST. Jeden pracuje s HTML DOM modelem a je určen pro vývoj a testování. Druhý pracuje s interní reprezentací AST v aplikaci Manta Checker. Vždy se musí použít ten správný, je potřeba se vyhnout ručnímu přepisovaní a vyřešit problém chytře. Řešením se stalo použití knihovny AMD-feature[6].

AMD-feature je AMD Loader plugin, který usnadňuje konfigurační řízení a podporuje cross-target development. Plugin přidává konfigurační mapu, ve které můžeme nadefinovat mapování mezi identifikátorem modulu a jeho implementací. Takové mapy se dají vytvořit pro každé prostředí. Pak stačí používat jako závislost potřebný modul ve formatu `feature!module_id` a modul bude načten dle aktuálně používané konfigurace.

Výhoda řešení je očividní. Pokud by se v budoucnu objevila další implementace wrapperu, stačí jenom změnit konfigurační soubor na jednom místě.

3.2.2.2 Implementace rozhraní Manta Checker

Aby byla zajištěna kompatibilita DFP modulu s celou aplikací Manta Checker, modul musí implmentovat rozhraní definované v aplikaci. Aplikace očekává potomka třídy `AbstractValidationTask`, který bude implementovat jeho metody, ze kterých je nejdůležitější:

```
protected void doExecute(IMantaUniversalAst input,
    ValidationReport output);
```

[6] https://github.com/jensarps/AMD-feature

Metoda, která se volá z metody `doExecute`, kde první parametr je stejný, a druhý je objekt `ValidationReport` obalený do třídy `InterpretTaskReportWriter` pro pohodlnější zápis výsledků:

```
protected abstract void doExecuteInternal(IMantaUniversalAst
    input, InterpretTaskReportWriter output)
```

`InterpretTaskReportWriter` obsahuje jenom jednu metodu pro reportování o chybě:

```
public void report(IMantaUniversalAst node, String error)
```

Parametry jsou uzel, který nesplňuje podmínku a textová chybová hláška.

V JavaScriptu pak byl vytvořen soubor, který definuje v globálním kontextu příslušné metody. Metoda `setJson` nastavuje pravidlo do lokální proměnný, a metoda `doExecuteInternal` spouští proces vyhodnocení pravidla.

Dále bylo potřeba doplnit globální prostor o řadu objektů, které jsou definované jenom v prostředí prohlížeče:

- `console` – používá se pro logování
- `setTimeout(fn, timeout)` – funkce pro odložené spouštění jiné funkce o timeout

3.2.2.3 Balíček pro kompilace kódu a integrace do Javy

Pro účely kompilace JavaScript kódu do Java byly použité knihovny a moduly:

- Rhino – JavaScript engine, napsaný v programovacím jazyce Java. Hlavní knihovna, která umožňuje integrovat JavaScriptové aplikace do Java programu.
- r.js – command line nastroj pro spouštění AMD aplikaci v Rhino, který také poskytuje možnost optimalizovat kód a sjednotit moduly do jednoho souboru.
- almond – Script Loader knihovna, která poskytuje obdobnou funkcionalitu jako RequireJS, t.j. umožňuje spouštění AMD aplikaci. Má omezenou funkcionalitu, proto je mnohem kompaktnější, ale obsahuje všechno co potřebujeme pro spouštění interpretu.
- AMD-feature – knihovna pro AMD aplikace, která umožňuje dynamické načítáni potřebného module (cross-target development) a usnadňuje správu kódu. Používá se pro automatickou volbu správného wrapper objektu pro práci s AST: jeden pro lokální vývoj a testování v Jasmine, jiný pro sestavení a použití v cílové aplikaci.

3.2.2.4 Generování dodávky

Po připravě potřebných knihoven, je možné vytvořit dodávku. Výsledkem by měl byt jar soubor, který se použije jako modul do aplikaci Manta Checker.

Pro generování dodávky je potřeba projít následující kroky:

- Sjednocení modulů do jednoho skriptu: Nástroj r.js sjednotí všechny moduly, včetně Almond jako AMD Loader
- Kompilace JavaScript souboru do Java tříd: pomoci nástroje Rhino se vygenerují class soubory
- Zabalení výsledků kompilaci do jar

Pro sestavení modulu se používá nástroj Maven[7]. Všechny kroky se konfigurují, což umožňuje generovat dodávku plně automaticky.

Nad dodávku modulu spolu s aplikaci je možné provádět integrační a systémové testy.

3.3 Algoritmus vyhodnocování pravidel

Základní část, která odpovídá za sestavení a vyhodnocení pravidel odpovídá procesor pravidel (`jsonProcessor`).

3.3.1 Procesor pravidel v formátu JSON

Na vstupu `jsonProcessor` dostává jako parametr kompletní definici pravidla ve formátu JSON a začíná je zpracovávat v následujícím pořadí:

1. Vytvoření komponent definovaných v pravidle
   - Vytvoření uzlů pro tuto komponentu
   - Vytvoření hran pro tuto komponentu
   - Inicializace lambda komponent definovaných v pravidle
2. Vytvoření uzlů definovaných v pravidle
   - Provádí se kontrola, zda uzel je nově definována komponenta: pokud ano, vytvoří se její instance, jinak se vytvoří uzel ze standardní komponenty
3. Vytvoření hran definovaných v pravidle
   - Vytvoří se hrana, která spojí porty dvou již vytvořených uzlů
4. spuštění vyhodnocení literálních uzlů

[7] https://maven.apache.org/

Jakmile literály budou vyhodnocené, t.j. jejich hodnoty budou předány do výstupních portů, spustí se mechanizmus propagací hodnot do dalších uzlů a následující jejich vyhodnocení, což zas dál propaguje hodnoty do výstupních portů. Tento proces se bude opakovat dokud celý řetězec pravidla nebude vyhodnocen.

3.3.2 Mechanizmus propagací hodnot

Mechanizmus propagací hodnot, zmíněný výše, hraje důležitou roli při provedení vyhodnocení v interpretu. Tento mechanizmus se též jmenuje Callback mechanizmus, a používá se ve dvou částech interpretu.

První použití je při vytvoření nové hraný. Na výstupní port se registruje Callback vstupního portu dalšího uzlu – funkce, která se má zavolat po aktualizaci hodnoty v příslušném portu. Tělem teto funkce většinou je změna hodnoty na novou v následujícím vstupním portu, kam směřuje hrana. Tak se po nastavení nové hodnoty v portu volají všechny na něm zaregistrované funkce, což propaguje novou hodnotu dál.

Druhé použití je v uzlech. Skoro na každý vstupní port se nastavuje Callback pro aktivaci vyhodnocení funkce uzlu. Tělem této funkce je volání `evaluate` metody uzlu s konkretní implementací. Po změně hodnoty na vstupním portu se vyhodnotí uzel a pošle nově vypočtenou hodnotu na výstupní port. Dále zafunguje již známý Callback na výstupním portu, které zase aktivují další uzly.

Tím pádem mechanizmus propagací hodnot vždy automaticky aktualizuje řetězec a umožňuje okamžité vyhodnocení po nastavení vstupních hodnot pravidla.

3.4 Lambda komponenta

Nejsložitější komponenta v interpretu ze všech pohledu je Lambda komponenta. To je kvůli roli této komponenty – můstek mezi JavaScript funkci a podřetězcem pravidla.

Nejdřív mi implementace nepřipadala těžká a stačilo jen nastavit funkci do výstupního portu po přípravě všech vstupních portů. Taková implementace lambda komponenty dobře fungovala, dokud se nezačalo testovat více do sobe vnořených lambda komponent a složitějších komponent pro úpravy. Byly udělané úpravy při vyhodnocení, které jsou potřeba pro správné fungování komponenty:

- na začátku vyhodnocení funkce je potřeba aktualizovat fixní (ne lambda) vstupy
- po vyhodnocení funkce je potřeba vynulovat lambda vstup

- v případě pokud se změní jeden z fixních vstupu na nulový, je potřeba vynulovat výstupní funkční port

Všechny výše uvedené body mezi sebou úzce souvisí a je potřeba je aplikovat najednou. První bod je důležitý pro úpravy stromu. Jelikož hodnoty na vstupních portech modifikačního uzlu jsou odstraněny, po jeho vyhodnocení je potřeba je tam znovu poslat. Pokud uzel pro úpravu bude mít více než jeden vstup, může dojít k tomu, že hodnota se nenastaví a neproběhne vyhodnocení.

Bez provedení druhého bodu dojde k problému, pokud máme dvě vnořené lambda komponenty: první s fixním vstupem A a lambda vstupem B, a druhá bude mít obraceně fixní vstup B a lambda vstup A. Pokud nevynulujeme lambda vstup B, a při novém vstupu budeme provádět aktualizaci fixních vstupů, ve vnořené lambda komponentě začne vyhodnocování pro starý neaktuální fixní vstup, což povede k špatnému výsledku.

Poslední bod je potřeba dodržovat, protože v případě vynulování jednoho z fixních vstupů u vnořené komponenty dojde k tomu, že funkce pořád bude nastavená po předchozím vyhodnocení, ale jeden z fixních vstupů je nulový, a proto funkce bude vracet špatné výsledky vyhodnocení, nebo dokonce může vyvolat chybu.

Po aplikaci těchto pravidel lambda funkce již měla korektní chování a použití vnoření komponent spolu s kombinací jiných uzlů nedělalo problémy.

3.5 Architektura DFP interpretu

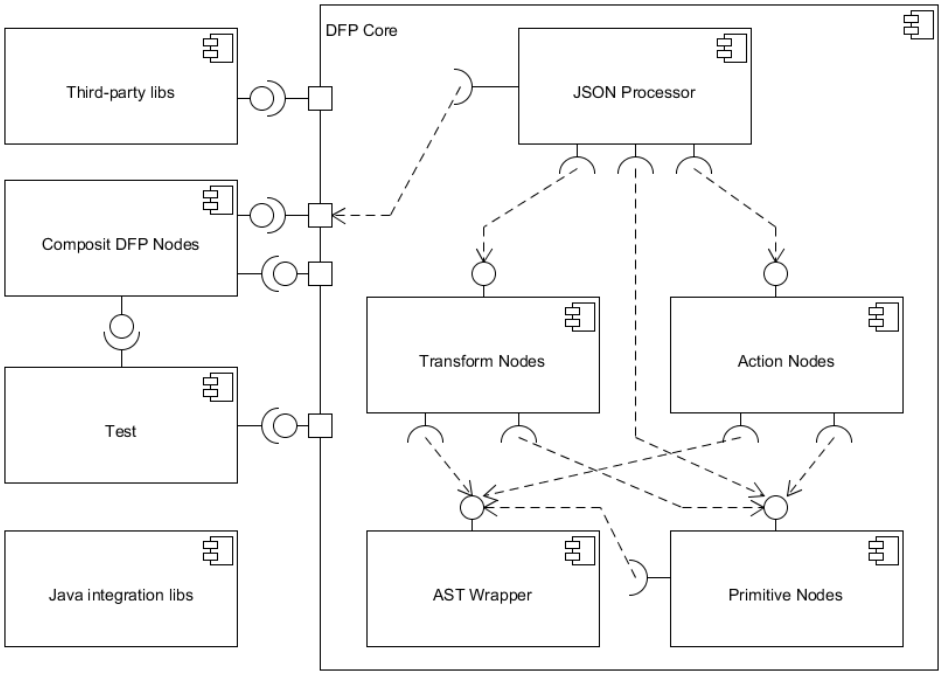
V této kapitole se zaměřím na architekturu DFP interpretu. Nejdřív představím celkovou architekturu na úrovní komponent, a pak podrobněji popíšu doménový model aplikace.

3.5.1 Komponentový model

DFP interpretu se skládá z následujících komponent:

- jádro DFP interpretu
  - knihovna primitivních uzlů
  - knihovna výpočetních uzlů
  - knihovna modifikačních uzlů
  - wrapper pro práci s AST
  - procesor pravidel v formátu JSON
- pomocné knihovny třetích stran (lodash.js)
- knihovna složených uzlů DFP
- knihovny pro integrace do Java (Rhino, r.js, Almond, AMD-feature).

- sada testů aplikace



Obrázek 3.1: Diagram komponent

Pro znázornění závislostí je představen diagram komponent. Téměř všechny komponenty z jádra interpretu konzumuji služby pomocné knihovny třetích stran. Naopak jádro DFP poskytuje svoje komponenty pro sestavení složených uzlů, které dohromady jsou pokryty testovacími scénáři (použity odpovídající komponentou).

Uvnitř jádra všechny uzly konzumují služby AST wrapper. JSON Procecsor používá veškeré knihovny uzlů včetně knihovna složených uzlů která není součástí jádra.

3.5.2 Doménový model aplikace

Pro lepší představu, je vhodné popsat aplikaci v jemnější granularitě. V této sekci bude doménový model každé z výše uvedených komponent.

3.5.2.1 Jádro DFP interpretu

Knihovna primitivních výpočetních a modifikačních uzlů, wrapper pro práci s AST a procesor datového formátu JSON – to všechno jsou součásti jádra

interpretu. Dále je potřeba zmínit důležité třídy, které se používají v rámci celé aplikace: uzel a atribut.

Třída uzel má standardní metody uzlu syntaktického stromu, které odpovídají rozhraní wrapper objektu. Atribut je objekt obsahující klíč a hodnotu.

3.5.2.2 Knihovna primitivních uzlů DFP

Primitivní uzly jsou základní komponenty, bez kterých by interpret nedovedl fungovat. Primitivní komponenty tvoří základní primitivní objekty a primitivní komponenty, které jsou definované pomocí základních objektů. Základní primitivní objekty jsou:

- `Port` – port je základní komponenta, která umožňuje tok dat. Porty obohacují uzly o možnost příjetí a odesíláni informaci.
- `Edge` – hrana je základní komponenta, jejím hlavním účelem je propojení uzlů (portů) mezi sebou.
- `DfpNode` – uzel je základní stavební kamen komponenty, který obsahuje její funkcionalitu. Bez ohledu na to, zda uzel transformuje, třídí nebo vypočítavá, všechny operace se provádí uvnitř uzlu. Tento objekt je základní a všechny jiné `Node` komponenty jej rozšiřují.

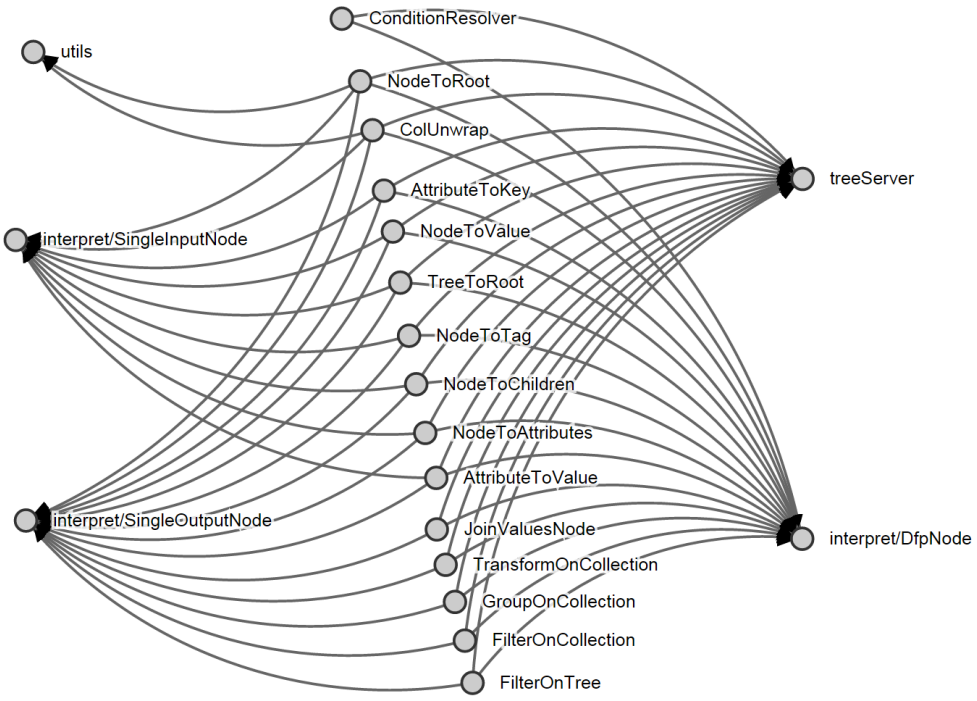
Další primitivní komponenty:

- `InputPort` – rozšiřuje Port o roli vstupního portu
- `OutputPort` – rozšiřuje Port o roli výstupního portu
- `LambdaInputPort` – rozšiřuje Port o pod-roli lambda vstupního portu, určen převážně pro LambdaComponent.
- `LambdaOutputPort` – rozšiřuje Port o pod-roli lambda výstupního portu, určen převážně pro LambdaComponent.
- `SingleInputNode` – omezuje DfpNode na jeden vstupní port
- `SingleOutputNode` – omezuje DfpNode na jeden výstupní port
- `LiteralNode` – rozšiřuje DfpNode. Jedná se o uzel s již nastavenou hodnotou, která po aktivaci se hned posílá dál, t.j. tento uzel je vstupem pro nějakou hodnotu pravidla.
- `SelfEvaluatingNode` – rozšiřuje DfpNode. Účelem tohoto uzlu je provádět okamžité vyhodnocení po změně na vstupních portech
- `ExpressionNode` – rozšiřuje DfpNode. Uzel obsahující nějaký vyraz, který musí byt vyhodnocen dynamicky za běhu. Používá se ve složených komponentech například pro porovnání hodnot z dvou vstupních portů.

- `ComponentNode` – rozšiřuje DfpNode. Říká, že uzel je komponentou a může mít vnořené komponenty.
- `LambdaComponent` – rozšiřuje DfpNode. Nejsložitější komponenta, která má za účel vyhodnocovat interní funkci a předávat výsledek dál.

Primitivní komponenty mohou byt vnořené a obalovat jedna druhou. Tak například můžeme vytvořit komponentu `ExpressionNode`, která ale bude obalena `SingleOutputNode`, čím zaručíme, že bude obsahovat jen jeden výstupní port.

3.5.2.3 Knihovna výpočetních uzlů DFP



Obrázek 3.2: Výpočetní uzly DFP

Výpočetní uzly se skládají z komponent, které provádí tříděni, konverzi nebo iterují uzly syntaktického stromu. Všechny takové uzly rozšiřují `DfpNode`, některé navíc omezuji komponentu na jeden vstupní nebo výstupní port (viz obrázek 3.2[8]). Konverzní uzly jsou:

- `NodeToAttributes` – provádí převod uzlu na seznam jeho atributů
- `NodeToChildren` – provádí převod uzlu na seznam jeho potomků

[8]Generováno pomoci dependo (https://github.com/auchenberg/dependo)

- `NodeToRoot` – provádí převod uzlu na kořen, t.j. posledního předka
- `NodeToTag` – provádí převod uzlu na jeho označení
- `NodeToValue` – provádí převod uzlu na jeho hodnotu
- `TreeToRoot` – provádí převod objektu typu Tree na objekt typu Node
- `AttributeToKey` – provádí převod uzlu na hodnotu jeho klíče
- `AttributeToValue` – provádí převod uzlu na jeho hodnotu

Iterační uzly představují sadu uzlů, které mají více vstupů. Jedním z nich je funkce, pomocí které provádí třídění, grupování a nebo konverzi nad kolekcí:

- `FilterOnCollection` – uzel pro filtrování kolekcí na základě filtrovací funkce
- `FilterOnTree` – uzel pro filtrování nad stromem na základě filtrovací funkce. Výsledkem je kolekce uzlů stromu, které splňují kritérium.
- `GroupOnCollection` – uzel pro grupování kolekcí dle kritéria grupovací funkce
- `TransformOnCollection` – uzel, který prochází celou kolekci a spouští nad každým prvkem transformační funkci. Výsledkem je kolekce objektů vrácených transformační funkcí.

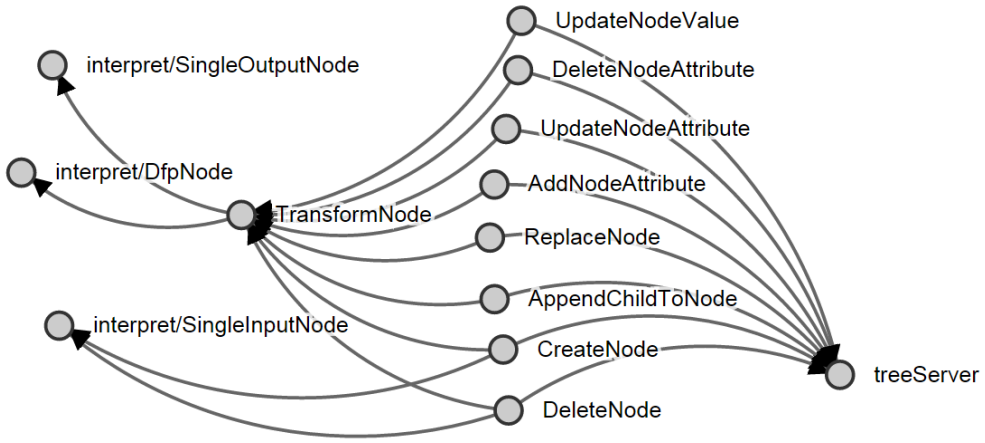
Poslední uzel z knihovny je `ConditionResolver`. Komponenta je určená pro vyhodnocení pravidel, má na vstupu grupu, kterou vytvoří uzel `GroupOnCollection`, a podmínku typu `all-satisfy` (`any-satisfy` nebo `non-satisfy`). Podle vstupu rozhodne, zda podmínka platí a do výstupních portů vrátí kolekci objektů, které nesplňuji podmínku, a booleovskou hodnotu, zda podmínka je splněna nebo není.

3.5.2.4 Knihovna modifikačních uzlů

Knihovna modikačních uzlů obsahuje veškeré uzly potřebné pro provádění úprav nad stromem. V rámci teto knihovny se používá návrhový vzor šablonová metoda (Template method pattern).[12]

Komponenta obsahuje společného předka `TransformatinNode`, uzel rozšiřuje `DfpNode` a obsahuje hlavní logiku:

- spouští vyhodnocení jen po naplnění daty všemi vstupní porty
- po vyhodnocení resetuje vstupní porty, aby nedošlo k nekonzistenci a opětovné úpravě již upraveného uzlu



Obrázek 3.3: Modifikační uzly DFP

Ostatní uzly z knihovny pak rozšiřují `TransformatinNode` a implementují jenom vyhodnocovací funkci (viz obrázek 3.3). Jsou to uzly:

- `AppendChildToNode` – provádí přidání potomka k uzlu
- `AddNodeAttribute` – provádí přidání atributu k uzlu
- `CreateNode` – vytváří nový uzel s konkretním označením
- `DeleteNode` – odstraňuje uzel
- `DeleteNodeAttribute` – odstraňuje atribut v uzlu
- `ReplaceNode` – nahrazuje jeden uzel druhým
- `UpdateNodeAttribute` – nastavuje novou hodnotu atributu uzlu s konkretním klíčem
- `UpdateNodeValue` – nastavuje novou hodnotu (obsah) uzlu

Výsledkem každého uzlu je ten uzel, který byl upravován. V případě odstranění je výsledkem jeho rodič.

3.5.2.5 Wrapper pro práci s AST

Komponenta představuje velkou knihovní funkci pro jednoduchou a pohodlnou práci s uzly. V daném případě se jedná o návrhový vzor strategie (Strategy pattern). Interpret obsahuje několik implementací a tím umožňuje práci s různými typy uzlů. Při přechodu od jedné implementace uzlu do druhé, se používá jen jiná implementace wrapper objektu a zbytek interpretu zůstává neměnný, jelikož používá stejné rozhraní wrapper objektu.[13]

Wrapper obsahuje metody dle návrhu:

- metody pro základní dotazovací operace nad AST
- metody pro konverzí mezi objekty AST
- metody pro modifikace uzlů
- metody pro hromadné operace nad uzly

3.5.2.6 Procesor pravidel v formátu JSON

Procesor pravidel se také skládá z jedné funkce, ale přesto je jedním z hlavních modulů interpretu, který umožňuje sestrojit zadané pravidlo a vyhodnotit jej.

Procesor je závislý na všech komponentách, které jsou nadefinováné v interpretu, aby je mohl použít jako stavební bloky. Interně má jednu metodu: `process(json)`. Na vstupu procesor čeká JSON reprezentaci pravidla a následně sestavuje podle něj celý diagram komponent. Vyhodnocení pravidla se provádí dle algoritmu.

3.5.2.7 Pomocné knihovny

Komponenty interpretu zatím používá jen jednu pomocnou knihovnu: lodash.js[9]. Jedná se o sadu funkcí, které zjednodušují práci a zpřehledňují kód. Knihovna poskytuje funkcionální varianty mnoha již existujících funkci a zbavuje potřeby použití iterativní konstrukce. Mezi nejvíce používané funkce patří: Funkce pro provedení určité operace nad každým elementem z kolekcí

```
_.forEach(collection, [iteratee])
```

Funkce pro filtrování elementů dle kritéria

```
_.filter(collection, [iteratee])
```

Funkce pro rozdělení kolekci na části dle kritéria

```
_.groupBy(collection, [iteratee])
```

Funkce pro transformaci kolekce na jiné objekty pomocí transformační funkce

```
_.map(collection, [iteratee])
```

Všude jako argumenty jsou:

- `collection` – kolekce, přes kterou se iteruje
- `iteratee` – funkce, která se volá pro každý element

[9] https://lodash.com/

3.5.2.8 Knihovna složených uzlů DFP

Složeným uzlem se rozumí jakákoliv komponenta definovaná za pomoci sady propojených primitivních uzlů.

Interpret může obsahovat předdefinované složené komponenty, aby pak ulehčil práci uživatelům. Má smysl předdefinovat a nabídnout uživateli komponenty, které evidentně budou používány. Do takových komponent patří:

- `TreeFilterOnNodeTag` – komponenta pro výběr uzlů ve stromě podle kriteria
- `NodesToAttributes` – komponenta pro konverzi uzlu na jejích atributy
- `FilterOnAttributeKey` – komponenta pro výběr atributu ze seznamu dle kriteria

3.5.2.9 Knihovny pro integrace do Java

Veškeré knihovny, které jsou potřeba pro tvorbu build modulu do Java jsou součástí této komponenty. Podrobný popis knihoven je v sekci 3.2.2.3.

3.5.2.10 Sada testů aplikaci

Sada testů spolu s testováním framework tvoří tuto komponentu. Testovací scénáře byly připravené pro každou z komponent interpretu. Podrobný popis testování je k dispozici v odpovídající kapitole 4.

Kapitola 4

Testování

Pro úspěšný vývoj interpretu je nezbytně nutné testovat aplikaci v každé fázi vývoje. Aby bylo možné pro aplikaci zajistit dostatečnou kvalitu, byly vytvořeny a provedeny jednotkové, integrační a systémové testy. V této kapitole popíši použité technologie, druhy testů a průběh testování.

4.1 Testovací framework

Před začátkem samotného testování bylo potřeba rozhodnout, jaký framework je vhodné použít. Na testovací frameworky byly kladeny následující požadavky:

- jednoduchá syntaxe
- možnost spouštění testu v prohlížeči
- přehledný výsledek testu

Tato kritéria splňovala řada frameworků, mezí kterými byly vybrány tři nejpopulárnější:

- Jasmine[10]
- Mocha[11]
- QUnit[12]

[10] http://jasmine.github.io/
[11] http://mochajs.org/
[12] https://qunitjs.com/

4.1.1 Jasmine

Jedná se o nejpopulárnější testovací framework. Má velíce jednoduchou a přehlednou syntaxi. Výsledky testů jsou velmi přehledné. Dobrá a výstižná dokumentace s příklady byly také velkým plusem. Příklad testu:

```
describe("A suite", function() {
    it("contains spec with an expectation", function() {
        expect(true).toBe(true);
    });
});
```

4.1.2 Mocha

Framework Mocha má skoro stejnou syntaxi jako Jasmine. Nevýhodou je, že framework nemá zabudovanou knihovnu assert a je potřeba použit knihovnu třetích stran. Dokumentace je postačující.

4.1.3 QUnit

QUnit se liší od dvou předchozích frameworků. Syntaxe je jiná a více podrobná. Struktura testovacích scénářů je méně přehledná, protože není vnořená, ale lineární.

4.1.4 Volba frameworku

Mezí frameworky, které odpovídaly požadovaným kritériím, byl jednoznačný vítěz framework Jasmine. Hlavní důvody výběru byly přehledná syntaxe, jednoduchost a dobrá dokumentace. Právě proto byl použit pro testování celého projektu.

4.2 Průběh testování

Protože se jedná o aplikaci bez uživatelského rozhraní a počet komponent v aplikaci je poměrně velký, bylo potřeba testovat velmi důkladně. Testování začalo hned po vytvoření první funkce.

Testy se spouštěly v prostředí prohlížeče Chrome, protože je moderní a má pokročilé zabudované ladicí nástroje.

Nejvíce se používala konzola prohlížeče, kam se dá pomocí volání metod na objektu console vypisovat ladicí hlášky. Konzola v Chrome umožňuje seskupovat hlášky, čímž usnadňuje dohledávání potřebných informací.

Další užitečný nástroj je inspektor objektů. Inspektor dovede podrobně ukázat strukturu objektů a jejich reference. Inspektor objektů funguje jak během ladění tak i na objektech vypsaných do konzoli.

I nakonec neméně užitečná je možnost step-by-step interaktivního ladění. Jedná se o možnost zastavit provedení instrukci v jakémkoliv místě pomoci

breakpoint, provádět instrukci po krocích, sledovat tok algoritmu a přeskakovat bloky kódu.

Dohromady tyto nástroje umožnily vyvinout aplikaci v dostatečné kvalitě za rozumný čas.

4.3 Unit testy

Unit testy jsou určené pro testování větších nebo menších celků nebo jednotek aplikací. Jelikož aplikace obsahuje několik úrovní, testování probíhalo postupně na každé z nich:

- wrapper objektu pro práci s AST
- objektů knihovny primitivních uzlů
- objektů knihovny konverzních a modifikačních uzlů
- knihovny složených uzlů

Při přidání nové funkce do wrapper objektu pro práci s AST byl hned napsán test pro tuto funkci. Poté byla tato funkce použita v konverzním nebo modifikačním uzlu a okamžitě byl přidán test této komponenty. Dále tato komponenta byla použita s řadou dalších v rámci složené komponenty, na kterou byl také napsán unit test. Tím pádem bylo zajištěno správné fungování všech jednotek aplikace, a většina chyb byla odhalená okamžitě po jejích vzniku.

4.4 Integrační testy

V rámci integračních testů interpret byl testován jako modul do aplikace Manta Checker. Cílem bylo ověřit správné chování v jiném prostředí. Nejvíce bylo potřeba otestovat funkčnost wrapper objektu pro práci s AST, protože ten měl jinou implementaci. Objekty na vyšších nejsou závislé na konkretní implementaci AST a používají společné rozhraní wrapper objektu.

Proto pro integrační testy byla připravena sada testů, každý z nich představoval jednoduché pravidlo pro jednu komponentu. Takovým způsobem byla odladěná funkčnost wrapperu a zajištěna kvalita modulu.

Bohužel kvůli nedostatku času a nepřipravenosti společného rozhraní Manta Checker pro úpravy, nebylo možné otestovat knihovnu modifikačních uzlů, což se ale udělá v nejbližší době.

4.5 Systémové testy

V rámci systémových testů byla testována funkčnost celého interpretu. Aby bylo možné otestovat DFP interpret jako hotový produkt, byla naimplemento-

vaná sada komplexních podnikových pravidel řetězců ze základních a složených komponent.

V prostředí prohlížeče byly otestována pravidla, která byla složena z komponent z knihovny primitivních, konverzních, modifikačních a složených uzlů. V prostředí Manta Checker byly otestovaná pravidla postavená ze všech uzlů kromě modifikačních.

Testy byly napsáné v podobě co nejvíce přiblížené produkčnímu použití interpretu, t.j. zápis pravidel používal jazyk pro definice pravidel, a s interpretem se pracovalo jako s černou skříňkou.

Výsledkem systémových testu bylo doladění aplikace a ověření plné funkčnosti všech komponent.

Závěr

Cílem práce bylo implementovat DFP interpret jako modul do aplikaci Manta Checker, což se mi podařilo splnit. Modul byl úspěšně vytvořen a dobře otestován. Interpret byl implementován s dostatečnou mírou obecnosti, proto budoucí rozvoj by neměl být problematický. Byla provedena integrace se systémem Manta Checker a funkcionalita byla ověřena během integračních testů. Kvůli nedostatku času a nepřipravenostím společného rozhraní pro úpravy nebyla otestována integrace těchto komponent, což bude učiněno v nejbližší době.

Vývoj této aplikace byl pro mě dobrou zkušeností a velkým přínosem. Rozšířil jsem svoje znalosti o funkcionálním programování a programování pomoci datových doků.

Interpret se bude dál rozvíjet ve směru přidání nových komponent. Podle vznikajících požadavků na úpravy, budou přibývat specifické komponenty pro složité typy úprav, pokud nebudou stačit existující komponenty. Dále lze uvažovat o přechodu na použití neměnných datových struktur, ale zatím taková potřeba není.

Věřím, že DFP interpret v rámci aplikaci Manta Checker bude mít praktické uplatnění v průmyslu a bude velkým přínosem pro podniky.

Literatura

[1] Přihláška projektu: Nástroje pro automatizaci Quality Assurance rozsáhlých Business Intelligence systémů a datových skladů. 2012.

[2] Závěrečná zpráva: Jazyk pro zadávání podnikových metodik a algoritmy vyhodnocování pravidel. 2014.

[3] Carkci, M.: *Dataflow and Reactive Programming Systems: A Practical Guide*. CreateSpace, 2014, ISBN 978-1497422445.

[4] Hudak, P.: Conception, evolution, and application of functional programming languages. *ACM Computing Surveys*, 1989.

[5] Ward, M.: Language Oriented Programming. 1994. Dostupné z: `http://www.cse.dmu.ac.uk/~mward/martin/papers/middle-out-t.pdf`

[6] Okasaki, C.: *Purely functional data structures*. Cambridge, U.K. New York: Cambridge University Press, 1998, ISBN 978-0521663502.

[7] Clojure. [cit. 2015-06-07]. Dostupné z: `http://clojure.org/refs`

[8] Závěrečná zpráva: Jazyk pro definici pravidel v XML. 2015.

[9] Osmani, A.: Essential JavaScript Namespacing Patterns [online]. [cit. 2015-06-05]. Dostupné z: `http://addyosmani.com/blog/essential-js-namespacing/`

[10] Malý, M.: Základní vzory pro vytváření jmenných prostorů v JavaScriptu [online]. [cit. 2015-06-05]. Dostupné z: `http://www.zdrojak.cz/clanky/zakladni-vzory-pro-vytvareni-jmennych-prostoru-v-javascriptu/`

[11] Osmani, A.: Learning JavaScript design patterns [online]. 2015, [cit. 2015-06-05]. Dostupné z: `http://addyosmani.com/resources/essentialjsdesignpatterns/book/`

[12] Gamma, E.: *Design patterns : elements of reusable object-oriented software*. Reading, Mass: Addison-Wesley, 1995, ISBN 0201633612.

[13] Freeman, E.: *Head First design patterns*. Sebastopol, CA: O'Reilly, 2004, ISBN 9780596007126.

Příloha **A**

Seznam použitých zkratek

DFP Data Flow Programming

IDE Integrated Development Environment

SQL Structured Query Language

BI Business Inteligence

DSL Domain-Specific Language

MCC Multiversion concurrency control

MVCC Multiversion concurrency control

STM Software Transactional Memory

CRUD Create, Read, Update, Delete

JVM Java Virtual Machine

JS JavaScript

FWF Fixed Width Format

CSV Comma-Separated Values

XML Extensible markup language

JSON JavaScript Object Notation

AST Abstract Syntax Tree

IIFE Immediately-invoked Function Expressions

API Application Programming Interface

AMD Asynchronous Module Definition

HTML HyperText Markup Language

DOM Document Object Model

PoC Proof of Concept

Příloha B

Instalační příručka

Pro kompilaci modulu pro systém Manta Checker je potřeba spustit příkaz `mvn package` ve složce s projektem. Pro instalaci modulu do lokálního Maven repozitáře je určen příkaz `mvn install`.

Aby bylo možno modul zkompilovat je potřeba mít nástroj Maven a závislosti z projektu Manta Checker, které jsou definovány v souboru `pom.xml`.

Ověřit funkčnost testu lze spouštěním frameworku Jasmine. Soubor pro spuštění je na cestě: manta-connector-rules-interpret/src/main/scripts/SpecRunner.html

Příloha C

Obsah přiloženého CD

- `readme.txt` ..... stručný popis obsahu CD
- `src`
  - `impl` ..... zdrojové kódy implementace
  - `thesis` ..... zdrojová forma práce ve formátu LaTeX
- `text` ..... text práce
  - `DP_Maksimau_Aliaksandr_2015.pdf` ..... text práce ve formátu PDF