Západočeská univerzita v Plzni
Fakulta aplikovaných věd
Katedra informatiky a výpočetní techniky

Diplomová práce

Určování nahraditelnosti a kompatibility webových služeb

Plzeň 2020 Zdeněk Valeš

Místo této strany bude zadání práce.

Prohlášení

Prohlašuji, že jsem diplomovou práci vypracoval samostatně a výhradně s použitím citovaných pramenů.

V Plzni dne 23. dubna 2020

Zdeněk Valeš

Abstract

The text of the abstract (in English). It contains the English translation of the thesis title and a short description of the thesis.

Abstrakt

Text abstraktu (česky). Obsahuje krátkou anotaci (cca 10 řádek) v češtině. Budete ji potřebovat i při vyplňování údajů o bakalářské práci ve STAGu. Český i anglický abstrakt by měly být na stejné stránce a měly by si obsahem co možná nejvíce odpovídat (samozřejmě není možný doslovný překlad!).

Obsah

1 Úvod

- k čemu je práce dobrá - co text práce obahuje - use casy

2 Principy webových služeb, techniky

V této kapitole jsou definovány webové služby a související pojmy. Jsou zde představeny strojově čitelné způsoby popisu rozhraní služeb a protokoly sloužící ke komunikaci s webovými službami. Tato kapitola také obsahuje popis REST.

2.1 Webové služby

Pojem 'webová služba' má různé významy pro různé lidi, ale dá se najít několik společných bodů [2]:

- Použití HTML, XML a dalších standardů webové architektury jako stavebních kamenů
- Zaměření na podnikové a vnitropodnikové operace

- pro účely této práce je použita následující definice od W3C [7]: "Webová služba je softwarový systém navržený pro podporu mezistrojové komunikace po síti. Webová služba má rozhraní, které je popsáno ve strojově čitelném formátu (konkrétně WSDL). Ostatní systémy interagují s webovými službami předepsaným způsobem za použití zpráv protokolu SOAP, které jsou typicky zprostředkované protokolem HTTP s využitím serializace XML a dalších webových standardů."

- poskytovatel služby = server

- konzument služby = klient (typicky nějaká aplikace)

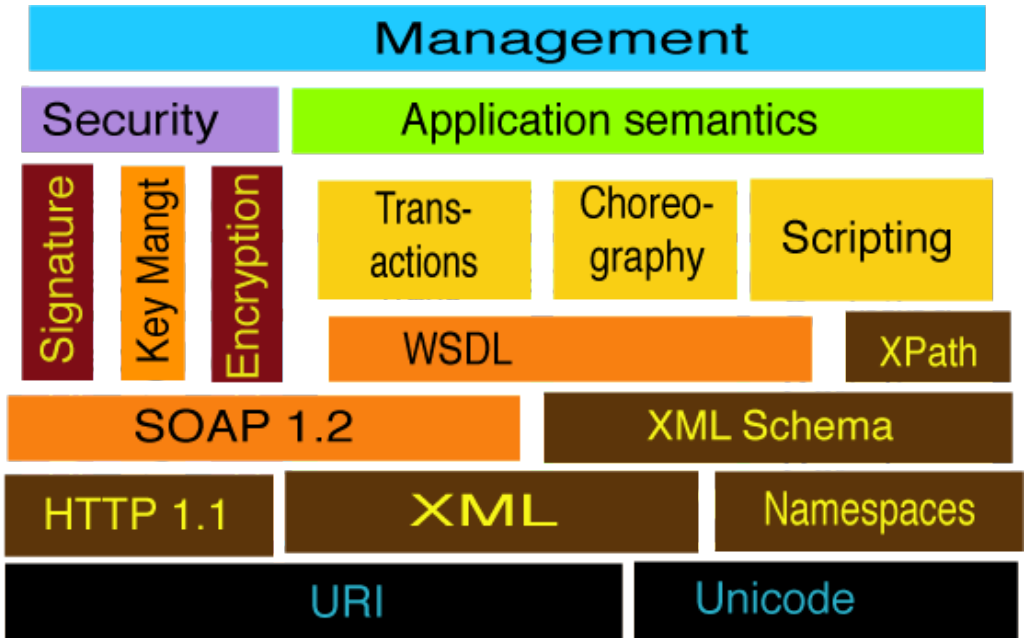
2.2 REST

- mohlo by se hodit: https://www.w3.org/TR/2004/NOTE-ws-arch-20040211/#relwwwrest
- popisuje vztah REST a WWW - REST: založeno na manipulaci s XML reprezentací webových resources skrze stateless operace

- popis architektonického stylu REST: https://www.ics.uci.edu/ fielding/pubs/dissertation/res
- popis elementů: - data, konektory, komponenty - popis view(na modelování)

- RFC na HTTP: https://tools.ietf.org/html/rfc7231#section-4

- odkaz konkrétně na request methods, mohlo by se hodit, protože ty jsou indexovány, tak alespoň na citaci



Obrázek 2.1: Technologie zahrnuté ve webových službách

2.2.1 Technologie použité k realizaci webových služeb

- zmínit: RPC, SOAP

2.3 Protokoly

- relevantní protokoly: RPC, SOAP, HTTP
  - HTTP (na REST a WS obecně)
  - protokol aplikační vrstvy SOAP (web service)
  - XML pro popis datového modelu
  - specifikace SOAP: https://www.w3.org/TR/soap12-part1/#intro

2.4 Formální popis webových služeb

- existují různé, strojově čitelné, formáty pro popis API
  - WSDL 1.1, 2.0, WADL (REST), JSON-WSP
  - Swagger, Raml, OpenApi
  - v případě REST bohužel není nic formálně nutné (oproti třeba SOAP), takže specifikace API nemusí být kompatibilní, nemusí být úplné, nebo můžou být ad-hoc (např. slovní popis ve Word dokumentu) a tím pádem nemusí existovat univerzální způsob strojového čtení těchto specifikací
  - v mé práci se věnuji především formátům WSDL, WADL a JSON-WSP

3 Datové typy a porovnávání

- přednášky z FJP - jak jazyky řeší datové typy - rekurzivní vs. nerekurzivní - primitivní typy (v xsd) - built-in typy (v Jave) - tady budu citovat [1] - subtyping: A <: B <=> A může být použito v kdekoliv kde je očekáváno B - kontravariance: F'(A) <: F(B) <=> B <: A

3.0.1 Porovnávání datových typů

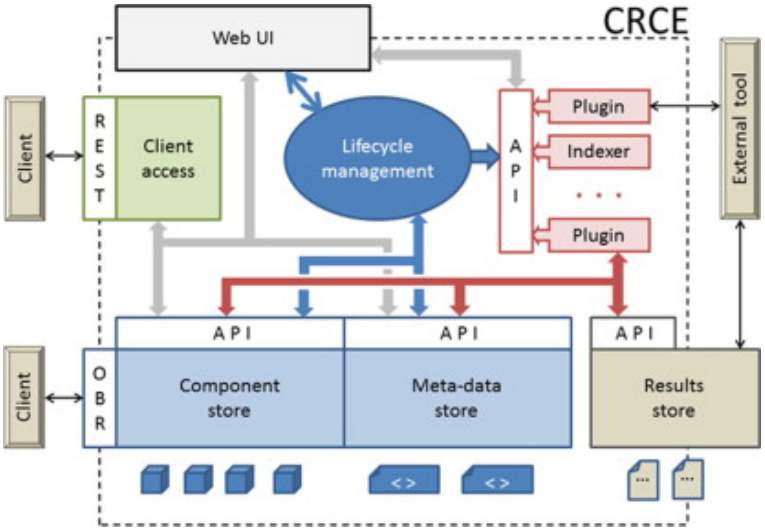
- jak to funguje - problémy při porovnání - subtyping vs. matching ([1])

4 Získávání metadat webových služeb v CRCE

Cílem této práce je vytvořit rozšíření pro úložiště CRCE[1], které bude schopno vyhodnocovat vzájemnou kompatibilitu indexovaných webových služeb. Tato kapitola představuje samotné úložiště, formát metadat a obecný způsob jejich získání. Na konci kapitoly je popsán princip fungování konkrétních rozšíření, která indexují webové služby (tzv. indexery).

4.1 CRCE

CRCE je komponentové úložiště dlouhodobě vyvíjené a spravované výzkumnou skupinou ReliSA na Katedře Informatiky ZČU, jehož primárním účelem je indexace a následná kontrola vzájemné kompatibility komponent. Úložiště je postaveno na modulární architektuře (viz obrázek 4.1) a je tedy možné přidat rozšíření pro indexaci a zpracování vlastních dat.



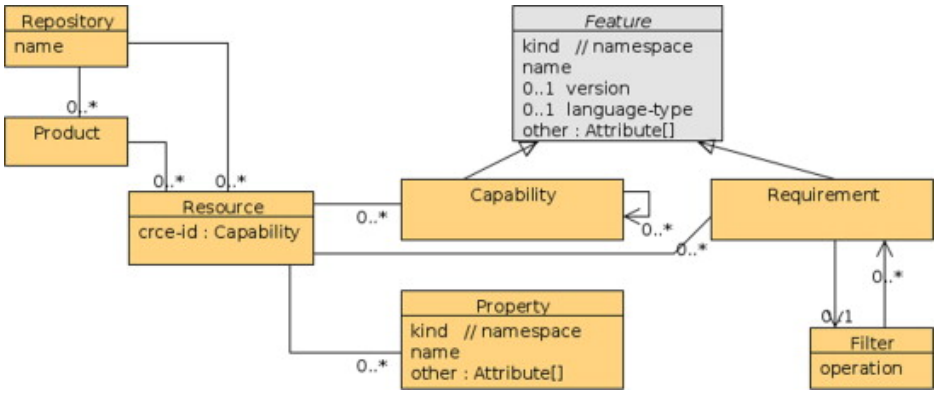
Obrázek 4.1: Architektura CRCE

[1]Component Repository supporting Compatibility Evaluation

4.1.1 Metadata komponent

Data, která vzniknou indexací komponenty a případným dalším zpracováním (např. porovnáním) jsou uložena do souboru metadat a představují klíčový element systému CRCE. Návrh struktury těchto metadat, který je naznačen na obrázku 4.2, vychází z konceptu OBR[2] jehož základními entitami jsou mimo jiné *Resource*, *requirements* a *capabilities*[3].

Entita *Resource* reprezentuje komponentu uloženou v CRCE, *requirements* a *capabilities* jsou množiny vlastností, popisující co komponenta ke své správné funkci vyžaduje, respektive co naopak poskytuje. Model také umožňuje přidání key-value atributů k jednotlivým vlastnostem a jejich detailům.



Obrázek 4.2: Reprezentace metadat v CRCE

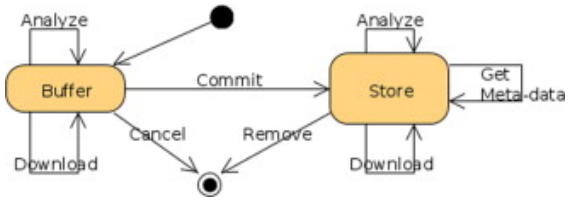
V mé práci jsem pracoval především s poskytovanými vlastnostmi (množina *capabilities*) a proto zde popíši hlavně jejich strukturu. Každá konkrétní vlastnost je reprezentována elementem *Capability* a od ostatních je odlišena identifikátorem *namespace*. Detaily konkrétní vlastnosti jsou popsány elementy *Property* a *Attribute*, kde *Property* reprezentuje logický celek několika atributů. Například parametr endpointu REST služby shlukuje atributy popisující jeho jméno, datový typ atp. *Attribute* pak představuje pár klíč-hodnota, který nese konkrétní informace jako např. jméno endpointu, nebo datový typ parametru.

Z obrázku 4.2 je vidět rekurzivní povaha elementu *Capability*, čehož je využito ke skládání jednodušších vlastností do složitějších celků. Vznikne tím stromová struktura, která je vhodná k modelování hierarchických dat mezi něž patří například popisy webových služeb. V případě takto komplexních vlastností je ke komponentně (*Resource*) přiřazena pouze jedna, tzv. kořenová, *Capability*, která reprezentuje celou vlastnost.

[2]OSGi bundle repository

4.1.2 Životní cyklus komponenty v CRCE

Úložiště bylo původně navrženo pro ukládání OSGi komponent, nicméně indexovat lze jakoukoliv komponentu. Komponenta je v CRCE popsána již zmíněnými metadaty a prochází vlastním životním cyklem naznačeným na obrázku 4.3.



Obrázek 4.3: Životní cyklus komponenty v CRCE

Životní cyklus má dvě hlavní fáze, jimiž jsou *Buffer* a *Store*. Komponenta po nahrání do úložiště nejprve prochází fází *Buffer*, v níž dojde k indexaci obsahu, kontrole vnitřní konzistence a kompatibility komponenty. Pokud touto fází projde bez chyb, je komponenta nahrána do trvalého úložiště (operace *commit*) a přechází do fáze *Store*.

V obou fázích jsou nad komponentou prováděny operace z nichž *analyze* je nejvíce relevantní mé práci, protože právě během této operace dochází ke sběru metadat (fáze *Buffer*) a dalším výpočtům nad nimi (fáze *Store*). Modul s rozšířením, který je předmětem mé práce bude zařazen mezi výpočty prováděné nad metadaty během *analyze*, kde bude vyhodnocovat vzájemnou kompatibilitu webových služeb. Způsoby sběru těchto metadat, ke kterým dochází ve fázi *Store*, jsou popsány v následující sekci.

4.2 Indexování webových služeb

V této podkapitole je krátce popsán obecný způsob indexace komponent v CRCE. Následně je podrobněji rozebráno indexování webových služeb konkrétními moduly a reprezentace popisu API metadaty v CRCE. Na závěr jsou také uvedeny limity indexování.

4.2.1 Obecná indexace komponenty

Indexace komponenty a související sběr metadat je proveden ve fázi *Buffer* k tomu určenými moduly. Ty jsou vzájemně nezávislé a obecně platí, že každý nich je zaměřen na sběr nějaké logicky ucelené části dat jako například informace o OSGi bundlu, maven koordináty, nebo popis webových služeb.

Vlastní data komponenty zůstávají během tohoto procesu nezměněná což v kombinaci s řetězením indexerů zaručuje mimo snadnou rozšiřitelnost také transparentní přístup ke komponentě každému z nich.

4.2.2 Indexace komponenty s webovou službou

Soubor obsahující implementaci, nebo popis API je v CRCE vnímán jako komponenta a prochází tedy zmíněným životním cyklem včetně výše popsané indexace. Pro popis webových služeb existuje mnoho standardních i nestandardních způsobů, jak již bylo zmíněno v kapitole 2. Z tohoto důvodu je není možné všechny analyzovat jedním indexerem a je nutné zaměřit se pouze na část z nich.

V současné době tedy existují dva moduly podporující několik popisných formátů a implementací. Konkrétně se jedná o modul pro indexaci webových služeb založených na architektonickém stylu REST[5] a o modul pro indexaci webových služeb s popisem ve formátu WSDL, WADL, nebo Json-WSP [6]. Oba dva vznikly v rámci diplomových prací a jsou stručně popsány v následujících sekcích.

Obrázek 4.4: Příklad indexované SOAP webové služby pro komix Dilbert

4.2.3 Struktura metadat popisující webovou službu

Jak již bylo zmíněno v části 4.1.1, hierarchickou strukturu popisu API lze vhodně vyjádřit metadaty CRCE. Během indexování komponenty reprezentující API jsou shromážděny různé typy popisných vlastností. Jedním z těchto typů je i samotný popis webové služby, který je reprezentován stromem metadat a ke komponentě je přiřazen skrze kořenovou *Capability*.

I když jsou různé druhy API indexovány rozdílnými moduly, výsledná metadata mají podobnou strukturu. Příklad metadat API je zobrazen na objektovém diagramu 4.4, jedná se o webovou službu, která vrací strip komixu Dilbert pro dnešní den.

Z uvedeného obrázku je vidět, že klíčové elementy API jako web service, nebo endpoint jsou reprezentovány objektem *Capability*. Detaily těchto elementů jsou popsány objekty *Property*. Jedná se zejména o parametry endpointů, těla requestů a response. Objekt *Attribute* pak představuje konkrétní hodnoty, jež jsou na obrázku naznačeny jen jako páry "klíč=hodnota". *Attribute* nemusí být vázaný jen na *Property* a lze jej použít i pro popis *Capability*, jak je tomu např. u objektu *WebService1*.

4.2.4 Indexování REST služeb

Modul pro indexování REST API vznikl v rámci diplomové práce Bc. Gabriely Hessové. Princip sběru dat je založen na binární analýze java archivů (JAR) obsahujících implementaci REST služeb pomocí frameworků splňujících specifikaci JAX-RS a frameworku Spring Web MVC. Modul byl testován na frameworcích Jersey verze 2.26, RESTEasy verze 3.0.16 a Spring Boot verze 1.5.9 [5].

Z implementace REST služby modul rekonstruuje kolekci endpointů s jejich parametry, tělem requestu, response a případnými parametry response. Každý endpoint je reprezentován entitou *Capability*, všechny další jeho vlastnosti pak entitami *Property*. Výčet všech indexovaných elementů rozhraní je uveden v tabulce 4.1.

4.2.5 Indexování webových služeb na základě popisu

Modul pro indexování webových služeb vznikl v rámci práce Bc. Davida Pejřimovského. Oproti předchozímu modulu pro indexaci REST služeb nepracuje tento s její implementací, ale s popisným souborem služby, tak jak bylo uvedeno v kapitole 2. Podporované formáty popisu služeb jsou WSDL (verze 1.1 i 2.0) pro SOAP webové služby, WADL a Json-WSP pro REST služby[6].

Element API	Entita v metadatech	Vstaženo k
Endpoint	Capability	-
Request body	Property	Endpoint
Request parameter	Property	Request
Response	Property Endpoint	Endpoint
Response parameter	Property	Response

Tabulka 4.1: Seznam indexovaných elementů REST služby a jejich reprezentací v metadatech

Struktura dat vytvořených pro REST služby (tedy z popisu WADL, nebo Json-WSP) je podobná struktuře dat vytvořenou předchozím indexerem. Endpoint je tedy reprezentován entitou *Capability*, jeho parametry entitami *Property*. Z popisu Json-WSP je ještě vytvořena reprezentace response pro daný endpoint (entita *Property*). Z popisu WADL se žádné další vlastnosti endpointů nezískávají.

Z WSDL popisu je vytvořena reprezentace služeb, jež jsou popsány xml elementy `<wsdl:service>` a jejich vnořených endpointů. Endpoint je ve WSDL popsán elementem `<wsdl:port>` a má definované operace (elementy `<wsdl:operation>`), nicméně modul tyto nevnořuje a vytváří zjednodušenou reprezentaci. Model endpointu tedy obsahuje metadata získaná z elementů `<wsdl:operation>` a url definovanou v elementu `<wsdl:port>`. Služby i endpointy jsou v metadatech reprezentovány entitami *Capability*. Oproti REST službám, které mají jednu úroveň vnoření *Capability*, zde vznikají úrovně dvě. Výčet elementů API a jejich reprezentace v metadatech je uveden v tabulce 4.2.

Element API	Entita v metadatech	Vstaženo k
Service	Capability	-
Endpoint	Capability	Service
Endpoint parameter	Property	Endpoint
Response	Property	Endpoint
Response parameter	Property	Endpoint

Tabulka 4.2: Seznam indexovaných elementů webové služby a jejich reprezentací v metadatech

Logika parsování WSDL souborů (verze 1.1. i 2.0) obsahovala chybu ve čtení adresy endpointu. Ta byla očekávána v atributu `action` elementu `<wsdl:operation>`, který ale není uveden ve specifikaci WSDL 1.1 [8] ani WSDL 2.0 [9]. Tuto chybu jsem v rámci mé práce opravil.

4.2.6 Limity indexování

Současný proces indexování webových služeb naráží na dva známé problémy týkající se datových typů. Jedná se o indexování rekurzivních datových typů a absenci samotných definic typů.

První problém se týká zejména indexování webových služeb podle popisných souborů, protože ty definice typů obsahují. Způsoby rozvoje a ukládání datových typů jsou popsány v [1], nicméně logika zatím není implementována. Druhý problém se týká binární analýzy REST služeb, protože archiv s implementací služby nemusí nutně obsahovat definice tříd. Ty mohou být například v jiném artefaktu, na který se archiv pouze odkazuje skrze závislost.

Z těchto důvodů je do metadat uložen pouze název datového typu což snižuje možnosti porovnávání služeb.

5 Modul pro porovnávání webových služeb

V této kapitole je detailně popsána funkce porovnávacího algoritmu společně s daty, nad kterými je možné porovnávač použít. Zároveň je zde popsán způsob vyhodnocení výsledků porovnání a formát uložení takto získaných dat.

5.1 Reprezentace rozdílů a kompatibility

Datové struktury použité pro reprezentaci rozdílů dvou entit a jejich vzájemné kompatibility se nazývají *Diff* a *Compatiblity*. Byly navrženy v rámci zkoumání možností určení kompatibility komponent na základě relace subtypingu [4]. Článek na komponentu nahlíží skrze její rozhraní, jako na rozhraní datového typu. Stejně tak se dá nahlížet i na rozhraní webové služby a proto jsem tento systém použil k reprezentaci výsledků v mé práci.

Diff je definován jako rekurzivní typ, který uchovává jak konkrétní informace o rozdílu skrze podřazené *Diff* tak i úroveň odlišnosti zvanou *Difference*. Tyto úrovně jsou popsány tabulkou 5.1 a v citovaném článku tvoří obor hodnot funkce $diff(a, b) : Type \times Type \rightarrow Difference$. Třída *Compatibility* uchovává informace o kompatibilitě dvou komponent, reprezentovány *Resource*, společně s detaily jejich rozdílů, jež jsou reprezentovány stromem *Diff*.

Název úrovně	Zkratka	Popis
None	NON	$a = b$
Insertion	INS	a není definováno, ale b ano
Specialization	SPE	b je subtyp a ($b <: a$)
Deletion	DEL	a je definováno, ale b ne
Generalization	GEN	a je subtyp b ($a <: b$)
Mutation	MUT	kombinace INS/SPE a DEL/GEN
Unknown	UNK	a nelze porovnat s b

Tabulka 5.1: popis úrovní rozdílů

Způsob vyhodnocení rozdílů dvou webových služeb a význam jednotlivých úrovní *Difference* pro klienta je popsán na konci této kapitoly.

5.2 Algoritmus porovnání

Porovnávací algoritmus pracuje s metadaty popsanými v části 4.1.1. Jedná se o stromovou strukturou, jejíž uzly tvoří instance tříd *Capability*, *Property* a *Attribute*, kde objekty *Attributes* jsou listy této struktury . Soubor metadat může obsahovat další vlastnosti komponenty, která představuje webovou službu, ta však zůstanou nedotčena, protože algoritmus pracuje pouze s daty, která byla vytvořena indexery popsanými v části 4.2.

Moduly pro indexování webových služeb popsané v předchozí kapitole používají dvě různé množiny *namespace* identifikátorů po pojmenování entit (*Capability*, *Property*, *Attribute*). Do budoucna je zároveň plánované rozšíření těchto modulů o funkcionalitu pro indexování datových typů a datová struktura metadat je tedy předmětem změny. Z těchto důvodů je algoritmus schopen porovnat pouze metadata z jednoho indexeru. Není tedy možné vzájemně porovnat například metadata REST služby získaná binární analýzou JAR s metadaty získanými čtením JSON-WSP dokumentu i když by se mohlo jednat o jednu službu.

5.2.1 Složitost algoritmu

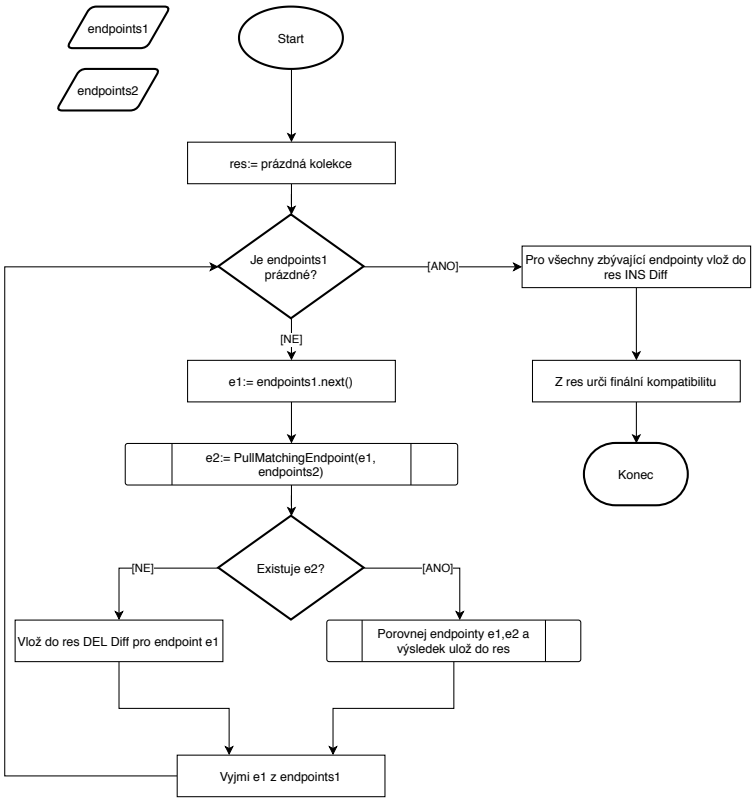
- REST služby: $O(n^2)$ kde n je počet endpointů
- WADL, JSON-WSP: $O(n^2)$ kde n je počet endpointů
- WSDL: $O(m \cdot n^2)$ kde m je počet WS a n je počet endpointů ve WS

5.2.2 Popis algoritmu

Vstup algoritmu: reprezentace obou webových služeb v podobě *Resource*. Výstupem algoritmu: objekt *Compatibility*.

Před porovnáváním datových struktur je zkontrolována jejich kompatibilita: typ popisu, *communication pattern*, *identity capability*. Pokud kontrola proběhne úspěšně, je spuštěn samotný algoritmus jehož průběh je naznačen na vývojovém diagramu 5.1. V případě porovnání SOAP webových služeb je tento postup obalen ještě porovnáním *services*, které WSDL definuje a které jsou v CRCE reprezentovány jako kolekce endpointů.

Mezivýsledky porovnání jsou ukládány v datové struktuře *Diff*, která je detailně popsaná v sekci 5.1. Algoritmus postupuje po stromu metadat od kořene směrem k listům. V listech dojde k porovnání konkrétních hodnot. Pokud jsou porovnány všechny potomci uzlu, je vyhodnocena jeho kompatibilita. Takto se postupuje zpět ke kořeni stromu, který na konci algoritmu



Obrázek 5.1: Vývojový diagram porovnávacího algoritmu

obsahuje finální údaj o kompatibilitě dvou API. Jednotlivé fáze porovnání jsou popsány v následujících odstavcích.

Výběr endpointu vhodného k porovnání Protože nelze předpokládat pořadí endpointů v metadatech, je nutné před samotným porovnáním nejprve vybrat dvojici endpointů k tomu vhodnou. Endpoint z první webové služby (*e1*) je vybrán sekvenčně (jak je naznačeno na 5.1). Druhý endpoint (*e2*) je pak vybrán na základě určité shody s metadaty endpointu *e1*, konkrétně se jedná o počet povinných parametrů, jméno a URL na které jsou endpointy dostupné. URL nemusí být shodné úplně, pokud tak nastane, je nastavena vlajka MOV, která je detailně popsána v sekci 5.3.

O porovnatelnosti dvou *services* (v případě SOAP) je rozhodnuto na základě úplné shody jejich jmen.

Porovnání dvou services Asi jen název.

Porovnání dvou endpointů Po výběru vhodné dvojice endpointů dojde k jejich porovnání. Postupně se porovnají (pokud jsou pro daný typ webové

služby definovány) parametry, response a request.

Porovnání parametrů endpointů Je porovnáváno: jméno, pořadí, datový typ, případně povinnost. V případě GEN/SPE u datového typu parametru je použita kontravariance a výsledná hodnota je opačná.

5.2.3 Porovnání datových typů

V současnosti jsou největším omezením porovnávacího algoritmu datové typy. Jako již bylo zmíněno v kapitole 4.2, jméno datového typu je jedinou dostupnou informací a tedy také jediným kritériem, podle kterého je lze porovnávat. To je dostatečné v případě vestavěných typů jako například třídy z balíku `java.lang`, nebo typy definované v xsd.

Nad těmito lze provádět plnohodnotné porovnání včetně kontroly generalizace. U vestavěných typů Javy je generalizace určena na základě dědičnosti a lze například určit, že typ s názvem `java.lang.Number` je generalizací typu s názvem `java.lang.Long`, protože třída *Number* je rodičem třídy *Long*. Obdobně je tomu i u vestavěných typů definovaných v xsd, kde je generalizace $a <: b$ definována v případě, že typ b je dost velký na pojmutí typu a. Například typ `xsd:long` dokáže reprezentovat číslo typu `xsd:int` a proto platí, že $xsd : int <: xsd : long$.

Porovnání vestavěných typů je založeno na podmínce správně zaindexovaného jména datového typu. V případě analýzy byte kódu je tato podmínka splněna, nicméně u metadat získaných čtením popisných XML souborů tomu tak vždy nemusí být. Problém spočívá v předponě datového typu, která je součástí jeho jména a porovnávací algoritmus očekává její určitou hodnotu (konkrétně `xs`). Hodnota předpony je však určena definicí namespace v XML popisu a ta se může od očekávané lišit. Tato definice není indexerem ukládána a není proto možné ji během porovnání přesně určit.

Pro uživatelsky definované typy je porovnání omezeno pouze na úplnou shodu, protože na základě pouhého jména datového typu nejde s jistotou usoudit nic dalšího.

5.3 Migrace webových služeb

V popisu porovnání metadat endpointů bylo zmíněno použití URL endpointu jako identifikátoru. Tento přístup naráží na problém, pokud poskytovatel webovou službu přesune, nebo provede změny v cestě k danému endpointu. Na obrázku 5.2 je uveden příklad dvou shodných endpointů, jež se liší pouze v URL. Aplikujeme-li výše popsaný algoritmus na tato data,

skončí negativním výsledkem, i přes to, že endpointy mají totožné rozhraní a klient by k nim mohl bez obtíží přistoupit.

Podobným příkladem je i verze API v cestě k endpointu. Klient může mít požadavek na zjištění kompatibility API dvou různých verzí, ale algoritmus vrátí rozdíl *MUT*, protože endpointy z prvního API vyhodnotí jako chybějící v API druhém a naopak, právě kvůli rozdílným cestám. Tím vznikne kombinace rozdílů *DEL* a *INS*, která vede na *MUT*. Takové chování není žádané a je potřeba těmto problémům předcházet, proto je nutné případné změny v URL detekovat a brát je při porovnávání v potaz.

Obrázek 5.2: Příklad shodných endpointů s rozdílnou URL

Jako řešení popsaného problému byl zaveden příznak "MOV", který je ortogonální k úrovni změny (*Difference*) popsané v předchozích sekcích a je součástí objektu *Diff*. Díky ortogonalitě je možné stále určit méně nebezpečné rozdíly jako například vložený endpoint (INS), nebo generalizaci v příznaku parametru (SPE) a zároveň předat klientovi informaci o změně v URL.

Příznak MOV má smysl brát v úvahu jen u případů porovnání jejichž úroveň změny je v podmnožině NON, GEN, SPE, která je v rámci této sekce označována jako bezpečná. Všechny ostatní úrovně představují v kontextu migrace příliš velkou změnu. Je tedy například nesmyslné nastavit příznak MOV u rozdílu dvou endpointů s úrovní změny *DEL*, protože samotná úroveň změny říká, že endpoint nebyl v druhé webové službě nalezen a proto nelze ani určit zda byl přemístěn.

5.3.1 Detekce změn v URL

Součástí URL endpointu je také jeho název, změny v URL se tedy dají detekovat na třech místech. Prvním z nich je doména (včetně protokolu), druhým je cesta k endpointu a třetím jeho jméno. TODO co a jak se detekuje.

Obrázek 5.3: Třída nesoucí výsledek detekce změn v URL

Výsledek detekce změn v URL, jež je zachycen třídou *MovDetectionResult* znázorněnou na obrázku 5.3, nese tři příznaky, kde každý z nich určuje, zda byla v dané části URL detekována změna. Vzhledem k tomu, že algoritmus detekce pracuje pouze s URL a jmény endpointů, dojde k pozitivnímu výsledku i v případě, že jsou porovnávány dvě odlišné web služby. Aby se redukoval počet false-positives, je potřeba určit, které kombinace příznaků

změny mohou vést na MOV a které představují příliš velké odchýlení. Tyto kombinace jsou uvedeny v tabulce 5.3.1, kde proměnné h, p, n označují změnu v doméně, cestě a jménu endpointu.

Kombinace	MOV	Zdůvodnění
$!h \wedge !p \wedge !o$	ne	Nebyla detekována žádná změna.
$h \wedge !p \wedge !o$	ano	Změna v doméně, může se jednat o migraci webové služby.
$!h \wedge p \wedge !o$	ano	Změna v cestě k endpointům, může se jednat o restrukturalizaci služby.
$h \wedge p \wedge !o$	ano	Může se jednat o kombinaci obou předchozích.
všechny ostatní	ne	Změna je příliš velká.

Z tabulky je vidět, že změna ve jménech endpointů vždy vede na záporný výsledek. Důvodem je fakt, že jméno endpointu je druhé kritérium pro výběr vhodného páru endpointů (prvním je URL) a bez něj by algoritmus zdegradoval na porovnání "každý s každým"což by značně snížilo efektivitu.

5.3.2 Výběr entit k porovnání s příznakem MOV

Protože jedním z kritérií pro výběr vhodného páru endpointů je i URL, je potřeba upravit logiku výběru tak, aby byly brány v potaz výsledky detekce popsané v předchozí sekci. Při porovnávání URL dvou endpointů je podle kombinace změn daná část URL ignorována a pracuje se pouze s nezměněnou částí u které je vyžadována striktní rovnost.

Tento způsob může vést na případy, kdy je dvojce endpointů vyhodnocena jako potencionálně vhodná k porovnání (s nastaveným příznakem MOV), nicméně porovnání skončí negativním výsledkem, například *UNK*. Pouhá akceptace takového výsledku a pokračování algoritmu by mohlo zapříčinit negativní vyhodnocení kompatibility webových služeb v případech, kdy skutečný rozdíl není tak silný. TODO: příklad

Algoritmus 'pick best'

Řešením zmíněného problému je algoritmus 'pick best', který postupně porovnává dvojice endpointů (k tomu předem vybrané) a jako výsledek vrátí dvojici s nejlepším rozdílem. Vstupem algoritmu je tedy endpoint z první webové služby *e1* a množina endpointů druhé webové služby *endpoints2*.

Algoritmus postupně vybírá elementy *e2* z *endpoints2* a pokud je pár $e1, e2$ vyhodnocen jako porovnatelný, dojde k detailnímu porovnání. V pří-

padě výsledku spadajícího do bezpečné množiny (NON, GEN, SPE) je pár $e1, e2$ vrácen a porovnání pro endpoint *e1* je ukončeno. V opačném případě je negativní výsledek uložen a z množiny *endpoints2* jsou vybírány další porovnatelné endpointy dokud algoritmus nedojde ke dvojici, jejíž rozdíl spadá do bezpečné množiny, nebo dokud není *endpoints2* vyčerpána. Pokud je množina *endpoint2* vyprázdněna, znamená to (alespoň částečnou) nekompatibilitu obou webových služeb a je vrácen první porovnávaný pár $e1, e2$.

5.3.3 Verze REST API v cestě k endpointu

TODO: citate, že verze API v cestě je common practice

Jednou ze speciálních změn detekovatelných v cestě k endpointu je verze API. Tato praktika je běžná [todo: cite] a k jejímu zpracování lze použít jednodušší přístup, než byl dosud popsán. Část cesty, která obsahuje verzi lze jednoduše vypustit a porovnat URL bez verze, což případě stejného API (s odlišností verze) znamená porovnání dvou identických URL. Příznak MOV je potom nastaven pokud jsou URL s verzemi rozdílné a URL bez verzí stejné.

Algoritmus podporuje standardní formát verze `vmajor.minor.micro`, který je vyjádřen regulárním výrazem: `/[vV][0-9]+(?:[.-][0-9]+){0,2}/`.

5.4 Vyhodnocení výsledků

Mezivýsledky porovnání jednotlivých elementů stromu metadat jsou ukládány do hierarchické struktury *Diff* a vždy po vyhodnocení všech rozdílů potomku uzlu, dojde na základě těchto i vyhodnocení rozdílu uzlu samotného. Každá z úrovní rozdílů *Difference* popsaných v tabulce 5.1 má určitou prioritu a uzel od svých potomků přejímá *Difference* s nejvyšší prioritou. Pokud tedy například dojde k rozdílu *UNK* (maximální priorita) při porovnání dvou atributů parametru endpointu, postupným vyhodnocováním se tato *Difference* dostane až ke kořenovému uzlu a výsledná kompatibilita bude mít hodnotu *UNK*. Hodnoty jsou v tabulce 5.1 seřazeny od nejnižší priority po nejvyšší.

- jak probíhá vyhodnocení (nejdříve se určí hodnoty listů, z nich se pak počítá dál nahoru)

- kontravariance

- není to úplně problém, ale při vyhodnocování finálního Diffu pro endpoint je potřeba brát v potaz

- GEN/SPE může vzniknout jen z datových typů parametrů/response endpointu, takže je to poměrně přímočaré

Difference	Dopad na klienta
None (NON)	bezpečné
Specialization (SPE)	bezpečné
Insertion (INS)	bezpečné
Deletion (DEL)	potenciálně nebezpečné
Generalization (GEN)	potenciálně nebezpečné
Mutation (MUT)	nebezpečné
Unkown (UNK)	nebezpečné

Tabulka 5.2: Dopad jednotlivých úrovní rozdílu na klienta

6 Implementační detaily (jen stručně)

- zmínit, proč třídy pro porovnávání REST API a WS nemají společného předka (krom rozhraní) - důvod: chtěl jsem nechat implementaci obou porovnávačů oddělenou pro případ, že by se změnila funkce indexerů

7 Testování

- nějaká reálná data - STAG (WSDL) - Fuel Economy - i syntetická data - algoritmus testován pomocí unit testů

7.1 Integrační + akceptační testy

- testování skrze REST API - pomocí Postman (Collection Runner) - několik verzí jednoho API -> testování křížem - todo: příklad - popsat verze API (čím se liší), případně zdůvodnit očekávaný výsledek - tabulka vzájemného porovnání s výsledky

7.1.1 Syntetický server v Jave

- postaveno na Jersey
- REST API
- alespoň obrázek/raml api?
- 2 verze otestované křížem

Popis verzí API (rozdíl je vždy popsaný oproti verzi 1):

- V1: základní verze
- V2: typ parametru změněn z Long na Number

Co se testuje:

- GEN/SPEC a kontravariance u parametru endpointu

7.1.2 Příklad JSON-WSP z wiki

- popsáno JSON-WSP souborem
- https://en.wikipedia.org/wiki/JSON-WSP
- 4 verze otestované křížem

Popis verzí API (rozdíl je vždy popsaný oproti verzi 1):

- V1: základní verze

- V2: typ User je jiný
- V3: přidána metoda deleteUser
- v4: metoda listGroups změněna na getUsersInGroup

Co se testuje:

- ne-indexování custom datových typů (takže v2 = v1)
- mutace endpointu
- INS, DEL

7.1.3 FuelEconomy API

- popsáno WADL souborem
- https://www.fueleconomy.gov/ws/rest/application.wadl
- 3 verze otestované křížem

Popis verzí API (rozdíl je vždy popsaný oproti verzi 1):

- V1: základní verze
- V2: odebrán (poslední) resource /labelvehicle
- V3: odebrán (poslední) resource /labelvehicle, přidán resource /somethingDifferent

Co se testuje:

- INS, DEL a MUT

7.1.4 Stag WS

TODO

8 Závěr

Literatura

[1] Abadi, M. – Cardelli, L. On Subtyping and Matching. In *European Conference on Object-Oriented Programming (ECOOP), Lecture Notes in Computer Science*, 952, s. 145–167. ACM Press, January 1995. Dostupné z: https://www.microsoft.com/en-us/research/publication/on-subtyping-and-matching/.

[2] Berners-Lee, T. *Web Services - Program Integration across Application and Organization boundaries* [online]. aug 2009. Dostupné z: https://www.w3.org/DesignIssues/WebServices.html.

[3] Brada, P. – Jezek, K. Repository and Meta-Data Design for Efficient Component Consistency Verification. *Science of Computer Programming*. 2015, 97, part 3, s. 349–365. ISSN 0167-6423. doi: 10.1016/j.scico.2014.06.013. Dostupné z: http://www.sciencedirect.com/science/article/pii/S0167642314002925.

[4] Brada, P. – Valenta, L. Practical Verification of Component Substitutability Using Subtype Relation. s. 38 – 45, 10 2006. doi: 10.1109/EUROMICRO.2006.50.

[5] HESSOVÁ, G. Automatické získání historických údajů z webových zdrojů [online]. Bakalářská práce, Západočeská univerzita v Plzni, Fakulta aplikovaných věd, Plzeň, 2015 [cit. 2020-02-22]. Dostupné z: https://theses.cz/id/pzbgj7/.

[6] PEJŘIMOVSKÝ, D. Vytváření a ukládání popisu webových služeb v úložišti CRCE [online]. Diplomová práce, Západočeská univerzita v Plzni, Fakulta aplikovaných věd, Plzeň, 2015 [cit. 2020-02-22]. Dostupné z: https://theses.cz/id/bb74eq/.

[7] W3C. *Web Services Architecture* [online]. feb 2004. Dostupné z: http://www.w3.org/TR/2004/NOTE-ws-arch-20040211/.

[8] W3C. *Web Services Description Language (WSDL) 1.1* [online]. mar 2001. Dostupné z: https://www.w3.org/TR/2001/NOTE-wsdl-20010315.

[9] W3C. *Web Services Description Language (WSDL) Version 2.0 Part 1: Core Language* [online]. jun 2007. Dostupné z: https://www.w3.org/TR/wsdl/.

Seznam zkratek

CRCE	Component Repository supporting Compatibility Evaluation
API	Application Programming Interface
HTTP	Hypertext Transfer Protocol
JAR	Java Archive
JSON	JavaScript Object Notation
JSON	JSON Web-Service Protocol
OSGi	Open Services Gateway initiative
REST	Representational State Transfer
SOAP	Simple Object Access Protocol
UML	Unified Modeling Language
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
WADL	Web Application Description Language
WSDL	Web Services Description Language
XML	Extensible Markup Language
XSD	XML Schema Definition

Příloha A - WSDL webové služby pro komix Dilbert

Tady bude WSDL