Budapesti Műszaki és Gazdaságtudományi Egyetem
Villamosmérnöki és Informatikai Kar
Méréstechnika és Információs Rendszerek Tanszék

Beöthy Bence

INTEGRÁLT SZOFTVER ÉLETCIKLUS MENEDZSMENT

GRÁFADATBÁZIS TECHNOLÓGIÁKKAL

KONZULENSEK

Dr. Ráth István Zoltán

(BME MIT)

Dr. Hegedüs Ábel

(IncQuery Labs Kft.)

BUDAPEST, 2018

Tartalomjegyzék

Kivonat 4
Abstract 5
1 Bevezetés 6
1.1 Problémafelvetés 6
1.2 Célkitűzés és megoldási javaslat 6
1.3 Dolgozat felépítése 7
2 Előzmények 8
2.1 Irodalomkutatás 8
2.2 VIATRA 11
2.3 IncQuery Server 12
2.4 Kapcsolódó technológiák 13
2.4.1 Application/Project Lifecycle Management 14
2.4.2 Implementációs technológiák 15
3 Integrációs szoftver tervezése 18
3.1 Követelmények 18
3.2 Általános rendszer architektúra 18
3.2.1 IQS modulok 19
3.2.2 Adapter modul 20
3.2.3 Ütemező 20
3.3 ALM specifikus rendszer 20
3.3.1 Architektúra 21
3.3.2 Metamodellek 22
4 Megvalósított rendszer 29
4.1 Esettanulmány 29
4.1.1 Massif GitHub Repository 29
4.1.2 Massif Jenkins Projekt 29
4.1.3 Massif SonarQube projekt 30
4.2 Adatok bejárása 30
4.2.1 GitHub Repository bejárása 30
4.2.2 Jenkins projekt bejárása 39
4.2.3 SonarQube projekt bejárása 41

4.2.4 Kereszthivatkozások 47
4.3 Adatok indexelése 48
4.4 Adatok kiértékelése 49
5 Értékelés 52
5.1 Felmerülő nehézségek 52
5.1.1 Web API inkonzisztenciák 52
5.1.2 GitHub API limitációk 53
5.2 Teljesítmény 54
6 Összefoglalás 62
6.1 Elvégzett munka 62
6.2 Továbbfejlesztési lehetőségek 62
6.3 Köszönetnyilvánítás 63
Irodalomjegyzék 64

Kivonat

Napjainkban egyre összetettebb informatikai rendszerek fejlesztését végezzük, melynek során kritikus kérdés a költségek és határidők becslése, valamint egy kívánt minőség betartása. Ezek jelentős kihívást jelentő feladatok, megoldásukban Application Lifecycle Management (ALM) és Product Lifecycle Management (PLM) eszközök lehetnek segítségünkre, melyek egy rendszer életciklusát az ötlettől kísérik végig a megvalósításon, a tesztelésen és a telepítésen át egészen az üzemeltetésig.

Egy ilyen menedzsment környezetben számos adatforrást találhatunk, mint például követelménykezelő eszközök, tervező és szimulációs programok, valamint kód- és adatbázis kezelő rendszerek. Ezek az eszközök csak egy-egy részét fedik le a problématérnek, így egy fejlesztés folyamán tipikusan több ilyen eszköz egyidejű és integrált használata szükséges. A gyakorlatban azonban az integráció általában nem teljeskörű, annak ellenére, hogy a különböző eszközök által kezelt információk között erős és fontos összefüggések vannak, melyek felismerése és kiaknázása kritikus fontosságú minőségbiztosítási és gazdasági szempontokból.

A dolgozat témája egy olyan integrációs platform megtervezése és kifejlesztése, amely hatékony, modell alapú megoldást kínál különböző modern ALM/PLM rendszerek nyílt interfészein keresztül elérhető adatok összekötésére. Az adatok tárolása az IncQuery rendszer segítségével gráf alapon történik, mely megközelítés alkalmas a különböző forrásból származó információk integrált kezelésére és hatékony kiértékelésére. A megoldás továbbá erős eszközt ad gráfminta alapú VQL (VIATRA Query Language) lekérdezések futtatására is, az adatok összességén értelemezett komplex összefüggések kinyerése céljából.

A rendszer működését egy esettanulmányon keresztül mutatom be. Munkám során több adatforrás-integrációs modult is megvalósítottam, melyek feladata forráskód projektekhez tartozó GitHub kódtárak és hibajegykezelők, Jenkins szerveren megtalálható fordítással és csomagolással kapcsolatos információk, valamint SonarQube szerveren végrehajtott statikus forráskód analízis eredmények feldolgozása, majd az általam definiált, adatintegrációs célú gráf lekérdezések kiértékelése. Az esettanulmány kiértékelése kiterjed hatékonysági mérésekre is, melyekkel a módszer gyakorlati alkalmazhatóságát igazoltam.

Abstract

Nowadays, we are developing more and more complex IT systems, in wich, it is a critical question to estimate costs and deadlines and to reach the desired quality. These are challenging tasks that can be easier with Application Life Management and Product Lifecycle Management tools. ALM and PLM tools cover the entire lifecycle from the idea, through the development, testing, deployment, support, and operation of a system.

In such a management environment, there are many data sources, such as requirements management tools, design and simulation programs, and code and database management systems. These tools cover only a part of the problem space, so a development typically involves the simultaneous and integrated use of many devices. In practice, however, integration is generally not complete, despite the fact that the connection between the information handled by the different tools is strong and important. Recognizing and exploiting such connections are critical for quality assurance and economical aspects.

The subject of the essay is the design and development of an integration platform that provides an effective, model-based solution for linking data accessible through open interfaces of various modern ALM/PLM systems. Data storing relies on a graph-based approach using the IncQuery system, which approach is suitable for the integrated management and efficient evaluation of information from different sources. The solution also provides a powerful tool to run graph pattern-based VQL (VIATRA Query Language) queries, which can be used to extract complex data interdependencies.

I introduce the system through a case study. During my work I implemented several data integration modules, which are designed to crawl the Github repository and issue tracker, the build information that can be found on the Jenkins server, and the SonarQube static source code analyses of a certain project, then process the accessible data, and evaluate the graph queries developed by myself. The evaluation of the case study extends to the efficiency analytics too, which were used to verify the practical usability of the method.

1 Bevezetés

Napjainkban egy fejlesztési projekt kapcsán egyre fontosabb kérdés a hatékonyság növelése, a költségek és a szükséges idő csökkentése, valamint ezek minél pontosabb becslése. Ezen követelmények megvalósításának elősegítését számos fejlesztési metodika és eszköz tűzte ki célul, melyek használata általános körben elterjedt. Ilyen eszközök például a kollaboratív fejlesztéseket lehetővé tevő forráskódtárak, hibakövető és projekt menedzsment rendszerek. Ezeket az eszközöket az Application lifecycle management (ALM), vagyis az alkalmazás életciklus menedzsment módszertana fogja össze.

1.1 Problémafelvetés

Egy szoftver fejlesztésekor egy arra a projektre specifikus toolchain-t, azaz eszközök egy láncolatát használjuk. Ezek olyan fejlesztői eszközök, amelyek valamilyen módon összeköttetésben állnak egymással és a fejlesztés egy-egy szakaszát segítik. A megfelelő eszközök segítségével a szoftver életciklusának egészét lefedhetjük, ami az ALM célkitűzése. Az eszközök integrációja azonban nem teljes. Számos olyan információt tárolnak külön-külön, melyek összeköttetésben állnak egymással, viszont ezek a származtatott információk csak nehezen vagy egyáltalán nem nyerhetők ki az adott eszközökből. Azonban ezek az adatösszefüggések fontos információkat hordozhatnak minőség, idő és költség szempontjából. Ilyen fontos összefüggés lehet például, hogy egy változtatás milyen hatással volt a fejlesztés későbbi fázisaira, mely változtatás okozta a legtöbb hibát és ezzel a legtöbb plusz munkát, illetve, hogy ezek a jelenségek mely felhasználókhoz és feladatokhoz köthetők.

1.2 Célkitűzés és megoldási javaslat

A felvázolt probléma megoldásához a különböző eszközök által szolgáltatott adatok együttes kezelésére van szükség. A dolgozat célja egy olyan rendszer megtervezése és megvalósítása, majd pedig egy esettanulmányon keresztüli bemutatása, mely lehetőséget nyújt eltérő forrásból származó adatok és a köztük lévő kapcsolat strukturált tárolására, azok együttes kezelésére, valamint tetszőleges lekérdezések futtatására. Ennek segítségével az eszköz olyan információk kinyerésére is alkalmas lesz, amelyek kinyerése az integrált ezközökből technikai vagy egyéb limitációk miatt nem

lehetséges. Az így összegyűjtött adatok ezen felül a későbbiekben további célokra is felhasználhatók, mint például jelentések vagy dokumentáció generálása.

Ennek az integrált ALM eszköznek egyik kulcskritériuma a könnyű bővíthetőség, aminek segítségével tetszőleges eszközök adatainak integrációja valósítható meg. Kutatásom során a GitHub (2.4.1.1), Jenkins (2.4.1.2) valamint SonarQube (2.4.1.3) eszközök adatintegrációját tűztem ki célul, mivel számos projekt fejlesztése folyik e három eszköz segítségével.

Az általam fejlesztett rendszer gráfadatbázis technológiákon alapszik, melynek oka, hogy a gráfadatbázisok különböző struktúrájú adatok együttes tárolását és a már meglévő struktúrák könnyű kiterjesztését is támogatják. Ezen felül gráfban reprezentált adatokon kifejező deklaratív gráfminta-alapú lekérdezéseket is futtathatunk, melyek komplex összefüggések kinyerésére is alkalmasak csupán az adatok közötti összefüggések megadásával.

1.3 Dolgozat felépítése

A dolgozatot további öt fő fejezetre bontom. A következő, vagyis 2. fejezetben ismertetem kutatásom technológiai hátterét valamint a hozzá kapcsolódó szakirodalmat. A 3. fejezetben ismertetem az integrációs rendszer felé támasztott követelményeket, valamint bemutatom és kifejtem az elkészült általános és specifikus architekturális terveket. A 4. fejezetben a megvalósított rendszer működését mutatom be egy esettanulmányon keresztül. Az 5. fejezetben értékelem az elkészült ezközt, valamint bemutatom a fejlesztés során felmerülő főbb akadályok megoldásait. Végül pedig a 6. fejezetben összefoglalom munkám eredményeit, és ismertetem a továbbfejlesztési lehetőségeket.

2 Előzmények

Ebben a fejezetben a projekt előzményeit mutatom be. Irodalomkutatással kezdem, mellyel elhelyezem a dolgozatot kapcsolódó kutatások között, majd pedig ismertetem azokat a technológiákat melyeket a célom megvalósításához választottam.

2.1 Irodalomkutatás

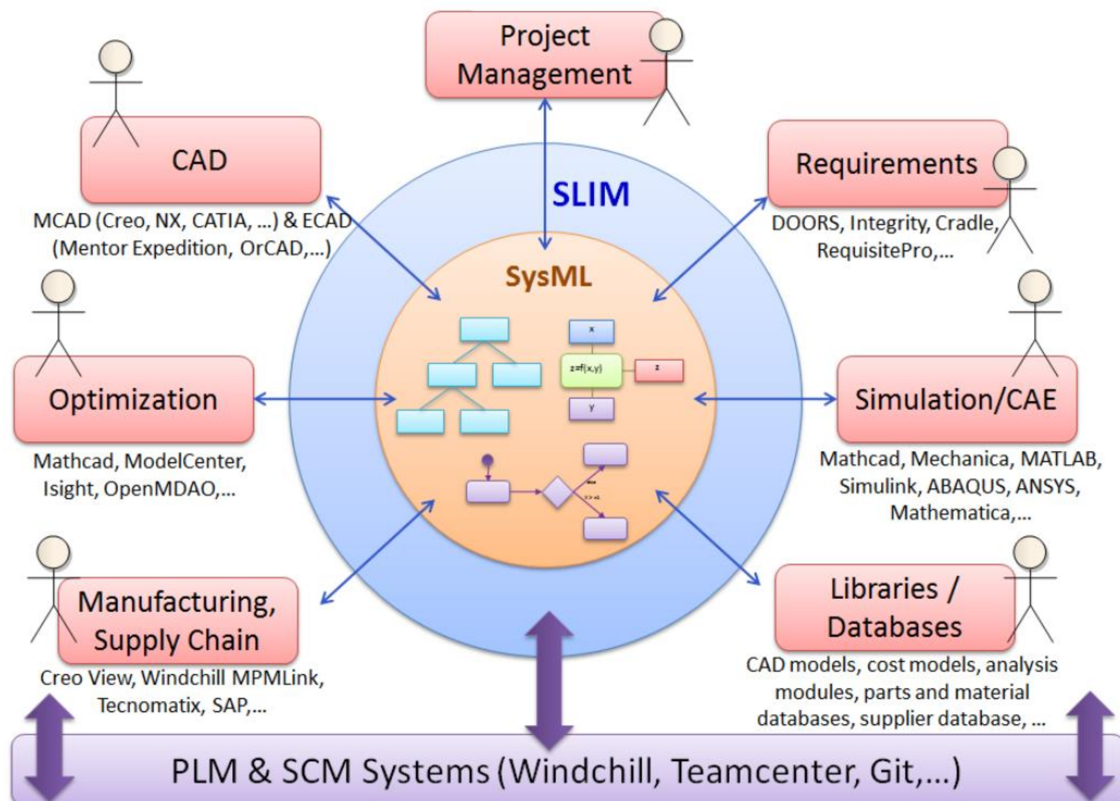
Az Application Lifecycle Management, azaz alkalmazás életciklus menedzsment magába foglalja egy szoftveralkalmazás specifikálásának, tervezésének, fejlesztésének és tesztelésének folyamatát. A szoftver életciklusát lefedi egészen az ötlettől a megvalósításon, a tesztelésen és a telepítésen át egészen a visszavonásig. A PLM, azaz termék életciklus menedzsment pedig nem csupán szoftverek, hanem más mérnöki termékek életciklus menedzsmentjével foglalkozik. Az ALM és a PLM egyre szélesebb körben ismert technológiák, melyek fontosságáról és hasznáról számos cikk született.

Christof Ebert például „Improving engineering efficiency with PLM/ALM"[3] című munkájában arról ír, hogy ALM/PLM módszerek használatában látja az MDE, vagyis a modell vezérelt tervezés hatékonyságának növelését, mely a rövidebb fejlesztési időnek és a megnövekedett minőségnek köszönhető. Ebert fontosnak tartja kiemelni, hogy az ALM és PLM nem csupán eszközök, hanem módszertanok, folyamatok összessége is, a siker érdekében pedig metodikaváltásra van szükség. Komplex projektek esetén sokszor elosztott fejlesztés jellemző, mely fragmentált folyamatokhoz és eszközláncokhoz, ez pedig heterogén, redundáns és inkonzisztens adatkezeléshez vezet. Ezen jelenség kiküszöbölése kulcskérdés, hiszen

A nyomon követhetőség és az integráció a fejlesztési fázisok és a használt módszerek, eszközök között rendkívül fontos. A „Developing the Requirements of a PLM/ALM Integration: An Industrial Case Study"[2] című cikk az ALM és a PLM integrációjának kérdésével foglalkozik. Az integráció nem könnyű, interfészekre van szükség. A kidolgozott megközelítés lényege egy olyan modell definiálása az ALM és PLM között, mely nem kapcsolódik az adott informatikai megoldásokhoz. Így az integráció költsége alacsonyabb, hiszen nem szükséges interfészeket megvalósítani a használt IT rendszerek között.

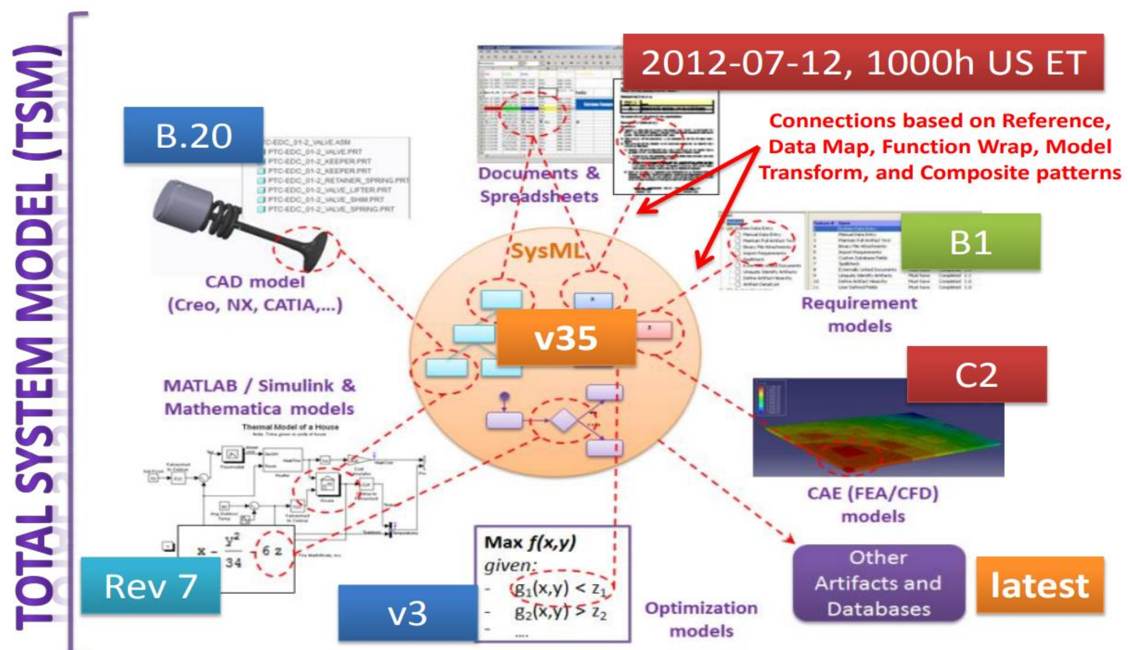
Az „MBSE++ — Foundations for Extended Model-Based Systems Engineering Across System Lifecycle”[18] című tanulmány egy következő generációs (MBSE) modell alapú megközelítést mutat be, mely az MBSE++ nevet kapta. Ez a cikk is a decentralizált, heterogén adatok és azok között fellépő redundanciát látja a fő problémának. Számos különböző forrásból származó modell, adat és kód kapcsolódik egy fejlesztéshez, ezeket a Syndeia nevű keretrendszerbe integráltak, mely lehetőséget nyújt modell-alapú és projekt menedzsment eszközök összekötésére. A teljes követhetőséghez új kapcsolatok felvételére is szükség lehet. Az adatok általában modern web standardokon alapuló interfészeken, mint REST és JSON érhetőek el, de egyre nagyobb teret kap egy új megoldás a GraphQL is.

Fejlesztésekhez kapcsolódó modellek és adatok integrálását természetesen már kész eszközök is lehetővé teszik. Az Intercax cég által fejlesztett Syndeia nevű eszköz például egy integrált modell-alapú mérnöki platform, amely lehetőséget nyújt különböző külső eszközök által szolgáltatott adatok és modellek integrálására.[14] A támogatott eszközök az alábbi ábrán láthatóak.



1. ábra: System Lifecycle Management (SLIM)[17]

A Syndeia a rendszertervezés során létrehozott SysML modelleket képes összekötni különböző feladatot ellátó eszközök adataival. Ezek az eszközök lehetnek PLM platformok, mint például a Git vagy a Teamcenter, lehetnek követelménykezelő rendszerek, mint a Doors, vagy akár olyan szimulációs keretrendszerek, mint a MATLAB Simulink vagy a Mathematica. A Syndeia emellett CAD szoftverek, modell tárak és adatbáziskezelő rendszerek integrálását is támogatja. Ezzel lehetőséget ad egy projekt teljes rendszermodeljének, az úgynevezett Total System Model-nek, azaz TSM-nek a menedzselésére és analizálására. Ennek eléréséhez új kapcsolatokat hoz létre a már létező adatok között, melyek lehetnek egyszerű adatkötések, vagy akár bonyolult adatleképezések az elemek attribútumai között. A teljes rendszermodell felépítése az alábbi ábrán látható.



2. ábra: Teljes rendszer modell (Total System Model - TSM)[17]

A teljes rendszermodell a rendszer egy tervrajza, amely végigkíséri azt az egész életciklusán. A Syndeia a rendszer SysML terveit összeköti a megfelelő CAD vagy egyébb modellekkel, kapcsolatot hoz létre modell elemek, forráskód és a hozzájuk kapcsolódó dokumentációk között, továbbá kétirányú leképezést készít a SysML követelmények és a használt követelménykezelő eszközben fellelhető követelmények között. Ezeket a kapcsolatokat tárolja és tartja karban a Syndeia, valamint képes az összekapcsolt adatokat manipulálni és követni a különböző eszközök által végzett módosításokat.

2.2 VIATRA

Az általam javasolt megoldás gráf és modell alapokra épít, egyik legfontosabb alkotóeleme a VIATRA[8] modelltranszformációs és lekérdezés kiértékelő keretrendszer, amely számos komplex feladatra hatékony megoldást adhat.

A VIATRA keretrendszer modelltranszformációk, különösképpen esemény vezérelt, reaktív transzformációk fejlesztését támogatja. Saját szakterületspecifikus nyelvével olyan transzformációk is leírhatók, melyek az adott modellek változásának hatására futnak le. A VIATRA keretrendszer magja egy inkrementális lekérdezés kiértékelő rendszer, amely gráfminta alapú lekérdezések futtatását teszi lehetővé a gráfszerű modellek példányai felett. A lekérdezések kiértékelése mintaillesztéssel, gyorsan, jól skálázódó módon történik.

Csikós Donát „Incremental dependency analysis of a large software infrastructure”[4] című TDK dolgozatában például inkrementális dependencia analízist végez Java projekteken EMF és VIATRA technológiák segítségével. Célja annak elősegítése, hogy egy szoftverkomponens frissítése ne legyen hatással a többi szoftverkomponens helyes működésére.

Ezzel szemben Stein Dániel „Incremental Static Analysis of Large Source Code Repositories”[21] című TDK munkájával az inkrementális statikus kódanalízist fektette modell alapokra EMF és gráfalapú technológiák segítségével. Megoldásának lényege, hogy először Java kódot alakít át EMF reprezentációba, majd pedig azon VIATRA segítségével mintaillesztéssel keres hibákat. Mintaillesztéssel végzett statikus analízisről szól a „Anti-pattern detection with model queries: A comparison of approaches”[22] című tanulmány is, melyben Java anti-pattern keresést végeznek modell lekérdezések segítségével. A tanulmány nagy hangsúlyt fektet annak lemérésére, hogy miként viszonyulnak egymáshoz időbeli lefutás és memóriahasználat szempontjából a hagyományos ASG, a tisztán EMF valamint az VIATRA különböző algoritmusain alapuló megoldások. A mérés eredményei azt mutatják, hogy a tisztán EMF implementáció lekérdezései lassúak, de a betöltési idő nagyon gyors, így egyszeri futtatásra alkalmas, addig az inkrementális megoldás rendkívül gyors lekérdezései viszont lassú inicializálása egyszeri betöltésre, majd ismétlődő lekérdezésekre használható jól. A lokális keresésen alapú megközelítés gyorsaságban a két előző közé tehető, mellyel Commit-idejű analízisre lehet alkalmas.

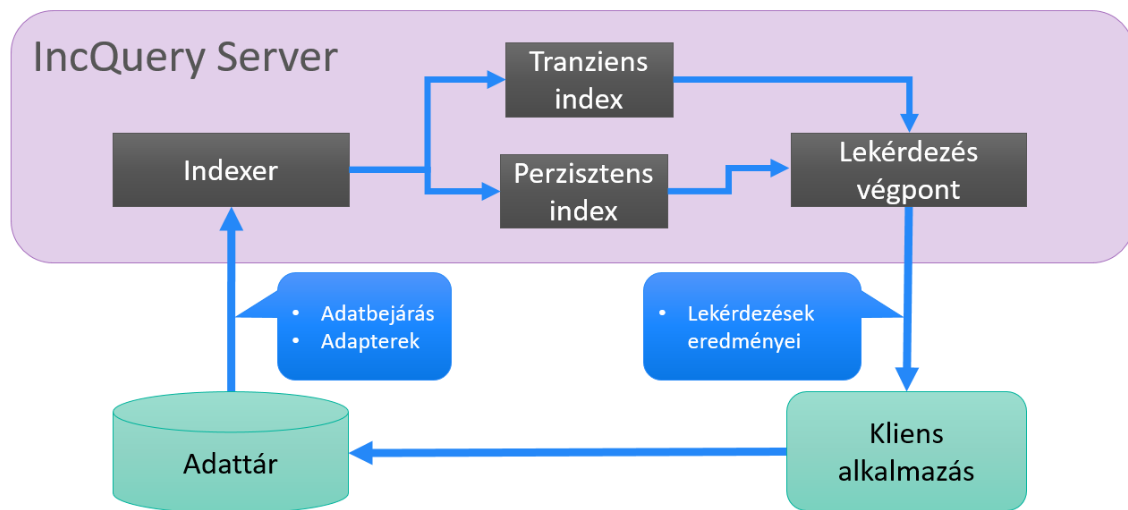
Ugyan az én rendszerem már meglévő eszközök adatintegrációját végzi, ezek a tanulmányok megmutatták, hogy egyfelől van igény az elosztott adatok ALM szintű kezelésére, valamint, hogy a VIATRA lokális keresésen alapú implementációja megfelelő tulajdonságokkal bír célom megvalósításához, amely egy olyan adatintegrációs szoftver fejlesztése, mely alkalmazások fejlesztése során használt eszközök adatait integrálja, s ezzel az Application Lifecycle Management, azaz alkalmazás életciklus menedzsment nézetet valósítja meg.

A VIATRA Query Language, (VQL) egy deklaratív, gráfminta-alapú lekérdező nyelv, melyet összetett strukturális modell lekérdezések leírására terveztek.[7] A VQL a VIATRA nevű modelltranszformációs és inkrementális lekérdező keretrendszer része, melyen az IncQuery Server is alapszik.[8] Előnyei többek között a jó kifejezőerő, olyan lehetőségekkel, mint az újrahasznosítható és kombinálható gráfminták, a minták negálásának lehetősége, továbbá az illeszkedések számolása és a paraméterek tetszőleges lekötése. Segítségével összetett lekérdezések írhatók, csupán az adatok közötti kapcsolatok megadásával.

2.3 IncQuery Server

Az IncQuery Server[13], vagyis IQS egy, az IncQuery Labs által fejlesztett skálázható lekérdezés kiértékelő rendszer, mely kimondottan nagy és összetett modellek kezeléséhez lett kifejlesztve.[1] Segítségével validációs és analitikai megoldások fejleszthetők. Az IQS ad-hoc lekérdezések futtatására alkalmas, melyek VIATRA Query Language (VQL) nevű gráfminta-alapú lekérdezőnyelvben írhatók.

Az IncQuery Server leegyszerűsített architektúrája és integrációs lehetőségei az alábbi ábrán láthatók.



3. ábra: IncQuery Server architektúra

Az IncQuery Server bemenetét az indexer modul szolgáltatja, melybe tetszőleges tárból, tetszőleges adatok (modellek) tölthetőek be gráfként reprezentálva. Ez az úgynevezett indexelés, amely a permanens illetve a tranziens indexekbe történhet, amelyeken lehetőségünk van lekérdezések futtatására. A lekérdezések eredményeihez a lekérdezés végpont nevű modul nyújt interfész.

Tanulmányom célja egy olyan, a Syndeiához hasonló adatintegrációs rendszer megtervezése, amely lehetőséget ad egy alkalmazás életciklusa során használt eszközök elosztott adatainak integrálására.

A megvalósítandó adatintegrációs ALM rendszer technológiai alapjául az IncQuery Server-t választottam. A megoldás lényege, hogy a különböző forrásból érkezett adatokat az eszköz egységesen az IQS indexében tárolja és kezeli. Lényegi különbség a Syndeia megközelítéséhez képest, hogy míg a Syndeia csupán az adatok és eszközök közötti kapcsolatokat tárolja, addig az általam tervezett rendszer belső tárjában az egész rendszermodellt, azaz TSM-et felépíti. Ezen megközelítés előnye, hogy az adatforrásokkal való kommunikáció csupán időközönként szükséges, nincs szükség valós idejű kapcsolatra, az adatok kiértékelése pedig rendkívül hatékony lehet.

2.4 Kapcsolódó technológiák

Ebben a fejezetben a projekt során felmerülő technológiákat fogom bemutatni. Először az integrálandó ALM eszközöket, majd az integrációs szoftver megvalósításához szükséges technológiákat ismertetem.

2.4.1 Application/Project Lifecycle Management

Habár léteznek ALM eszközök, azok általában csak bizonyos részeit fedik le az említett fázisoknak, emiatt egy fejlesztés során jellemzően több eszköz együttes használata jellemző. Kutatásom során a GitHub, Jenkins valamint SonarQube eszközök adatintegrációját végeztem el, ezeket az eszközöket a következő fejezetekben fogom bemutatni.

2.4.1.1 GitHub

„GitHub is how people build software."[12] A GitHub egy Git alapú webes verziókezelő szolgáltatás, melyet főleg számítógépes szoftverek tárolására használnak. Természetesen a Git minden eszközével rendelkezik, de túlmutat annál. A forráskód menedzsment mellett hozzáférés kezelést, hibakövetést, dokumentációkezelést valamint projekt menedzsment feladatokat is támogat.

Egy fejlesztés során keletkező fájlok az úgynevezett Repositoryban foglalnak helyet, azok változásait pedig úgynevezett commitok reprezentálnak. Az elvégzendő feladatok, valamint kijavítandó hibák úgynevezett issue-kkal írhatóak le, melyek milestone-okba, azaz mérföldkövekbe, illetve projektekbe szervezhetők, projekt menedzsment céljából. A kódbázis bármely állapotából kiadás készíthető. Ezen eszközökkel a GitHub alkalmas egy alkalmazás vagy termék specifikációs, fejlesztési és kiadási fázisainak ellátására. A szoftveréletciklus további állomásainak, mint például a folyamatos integráció, telepítés vagy a tesztelés elvégzéséhez külső szolgáltatásokat kell igénybe vennünk, ezek integrációja azonban megoldott.

2.4.1.2 Jenkins

A Jenkins egy széles körben használt nyílt forráskódú, elosztott és kiterjeszthető automatizációs szerver, mely olyan posztfejlesztési feladatok automatizálásában segít, mint folyamatos integráció, build feladatok, karbantartás, telepítés és tesztelés.[15] Különböző verziókezelő, futtató és dependenciakezelő eszközöket támogat, valamint pluginek segítségével még jobban testre szabható. Jellemzően a forráskód fordítását, telepítését és futtatását, illetve a projekthez tartozó testesetek kiértékelését végzi.

Egy build lefutását egy másik automatizált lépés vége, valamint külső szolgáltatások is kezdeményezhetik. Githubbal integrálva elérhető például, hogy a repository-hoz adott összes új commit hatására lefusson a folyamat. A megfelelő

beállítások mellett például a GitHub nem engedi olyan változtatások véglegesítését, amelyhez tartozó Jenkins ellenőrzés hibás eredményt mutat.

2.4.1.3 SonarQube

A SonarQube egy nyílt forráskódú statikus forráskód analízis eszköz, mely kódbázisok minőségének és biztonságának emelését tűzte ki célul. Ennek érdekében automatizáltan keres hibákat és sebezhetőségeket. [20]

Egy analízis eredményeként issue-k és mérőszámok egy-egy listája áll elő. Egy issue jelölhet kritikus problémát vagy figyelmeztetést, melynek tárgya potenciálisan hibákat, problémákat okozhat. Továbbá jelölhet úgynevezett „Code Smell”-eket is, melyek nem okoznak funkcionális problémát, viszont például rontják a kód olvashatóságát és teljesítményét. Ezen felül az eszköz képes megbecsülni a hibák kiküszöbölésének várható idejét, illetve kiszámolni a fellelhető tesztek kódlefedettségét is.

2.4.2 Implementációs technológiák

Ebben a részben az implementáció során használt technológiákat ismertetem.

2.4.2.1 Kotlin

A fejlesztést a Kotlin nevű objektum-orientált programozási nyelvvel végeztem az IntelliJ Idea fejlesztői keretrendszerben, melyeket a JetBrains fejleszti.[16]

A hivatalos weboldal szerint a Kotlin egy „Statikusan típusos programozási nyelv modern multiplatform alkalmazások fejlesztéséhez”. Kotlinban írt kód JVM-en, azaz Java Virtual Machine felett futtatható, valamint JavaScript forráskóddá fordítható. A szintaxis nem kompatibilis a Java-val, de teljes mértékben együttműködik azzal, bármilyen Java könyvtárt használható Kotlin kódban.

Számos erős tulajdonsága közül az alábbiakat emelném ki: típus ellenőrzés, automatikus kasztolás, lambda kifejezések, függvény kiterjesztés, genericitás, jobbrekurzivitás nyelvszintű támogatása, String template-ek. A funkcionális programozásból kölcsönzött elemeknek, a null biztosságnak, illetve a kompakt szintaxisnak köszönhetően összetett rendszerek kényelmes fejlesztésére is alkalmas.

2.4.2.2 Eclipse Vert.x

A reaktív, eseményvezérelt, többszálú és aszinkron programozás nem egyszerű feladat. a megfelelő API nélkül könnyen hibát véthetünk. A rendszer eseményvezérelt komponenseinek elkészítéséhez a Vert.x nevezetű API-t használtam Kotlin nyelv felett.[6]

Az „Eclipse Vert.x egy tool-kit JVM-en futó reaktív alkalmazások készítéséhez”, áll a hivatalos oldalon. A Vert.x eseményvezérelt és non-blocking, vagyis kisszámú szálon is konkurens működésre képes, valamint jól skálázódik. Számos nyelvre elérhető úgy, mint Java, JavaScript, Groovy, Ruby, Scala és Kotlin.

A Vert.x általános célú és rendkívül flexibilis. Több alapvetően különálló, de bonyolultabb feladatoknál egymást segítő API komponenssel rendelkezik. Ezek közül a projekt során jellemzően a Core és a Webclient alkotó elemeket használtam.

Vert.x Core

A Vert.x API magja, melyben rengeteg alapszolgáltatást megtalálunk, úgy mint blokkoló és nem blokkoló végrehajtások, Event loop, Eventbus, streamek, parserek és szálkezelés, valamint a Vert.x alapjául szolgáló verticle kezelés is itt kapott helyet.

Vert.x-ben úgynevezett verticle-öket hozhatunk létre, melyek különálló feldolgozó egységként működnek. Egy-egy verticle futhat több szálon, több példányban, valamint akár más nyelveken megírt csomópontok is képesek együtt dolgozni a Vert.x engine felett. Ezek az egységek aszinkron kommunikálnak egymással.

Web Client

A Vert.x API kimondottan webkliensek implementálását segítő része. A kiterjedt és intuitív API-nak köszönhetően egyszerűen írhatók HTTP és HTTPS kérések, valamint a válaszok feldolgozása is könnyedén elvégezhető.

2.4.2.3 Eclipse Modelling Framework (EMF)

Az Eclipse Modeling Framework egy modellező keretrendszer és kódgeneráló eszköz modellalapú alkalmazások fejlesztéséhez.[5] Az EMF lelke az úgynevezett Ecore önleíró metamodell, amely más modellek leírását teszi lehetővé konstrukció segítségével. Az általunk elkészített metamodellekből az EMF képes Java reprezentációt generálni. A metamodellt leíró osztályokon kívül példányosítást végző, összehasonlító és egyéb segédosztályok is generálódnak, melyek lehetővé teszik az adott metamodellnek eleget

tevő példánymodellek futás-idejű felépítését és manipulálását. Az EMF továbbá eszközöket nyújt modellben történő változások követésére, a modellek szerializációjára és modellszerkesztők készítésére

Az EMF három alapvető részből áll. Az EMF Core a keretrendszer magja, mely tartalmazza az Ecore metamodellt, az XMI alapú szerializációt és az API-t az EMF objektumok generikus módosításához. Az EMF.Edit generikus újrahasznosítható osztályokat nyújt az EMF modellszerkesztők létrehozásához. Az EMF.Codegen pedig egy kódgenerátor, amely képes legenerálni minden olyan eszkört, mely egy komplett szerkesztő létrehozásához szükséges.

EMF alapú fejlesztés mindig egy metamodell elkészítésével kezdődik. A keretrendszer grafikus és táblázatos szerkesztőt is nyújt metamodellek meghatározásához. A fejlesztés során az indexálni kívánt adatokat reprezentáló belső metamodelleket készítettem EMF segítségével, melyeket 3.3.2. fejezetben fogok részletesen bemutatni.

3 Integrációs szoftver tervezése

Célul tehát egy különböző forrásból érkező adatok integrációjára és kiértékelésére képes rendszer fejlesztését tűztem ki. Ebben a fejezetben a rendszerrel szemben támasztott követelményeket, majd pedig megtervezett architektúráját részletezem.

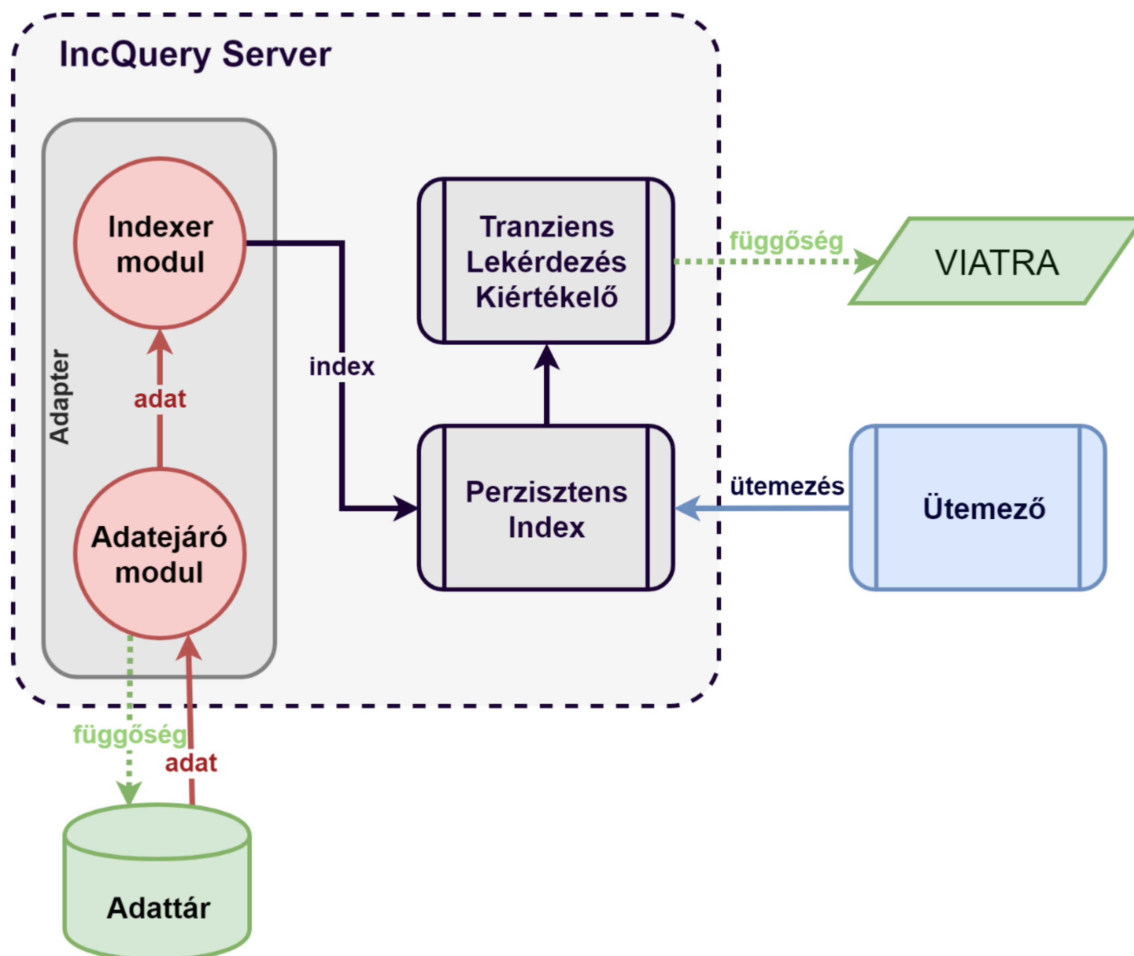
3.1 Követelmények

Az általam tervezett adatintegrációs szoftvernek funkcionális, minőségbeli és teljesítménybeli követelményeknek kell megfelelnie. Funkcionálisan képesnek kell lennie különböző külső forrásból származó adatok begyűjtésére és a kapott adatok tárolására, indexelésére az IncQuery Serverbe. Egy ilyen rendszer segítségével képesek lennénk vezetői összesítő oldalak készítésére, amelyek az adatok összességén kiértékelt fontos összefüggéseket és statisztikákat mutatják, valamint riport- és dokumentációgenerátor megvalósítására.

A tervezés során a megfelelő modularitás elérését tartottam szem előtt, igyekeztem a komponenseket jól elkülönülő felelősségi körök alapján megválasztani. Ennek előnye, hogy az elkészült kód könnyen módosítható és bővíthető, illetve egy esetleges hiba könnyebben lokalizálható. Jelen esetben ezek a tulajdonságok rendkívül fontosak, hiszen az elkészült rendszernek alkalmasnak kell lennie arra, hogy igény esetén újabb és újabb adatforrásokkal lehessen bővíteni. A moduláris felépítés továbbá a megfelelő teljesítmény elérésének szempontjából is fontos, mert segítségével aszinkron, skálázható működés valósítható meg.

3.2 Általános rendszer architektúra

Az integrációs rendszert gyakorlatilag az IncQuery Server kiterjesztése külső adatforrásokból adatokat feldolgozó egységekkel, adapterekkel. A rendszer architektúrája az alábbi ábrán látható.



4. ábra: Általános rendszer architektúra

Az ábra alapvetően két részre osztható. Az IQS belsejében lila kerettel jelölt modulok az IQS-ben már létező modulok, míg az Adapter és az Orchestrator elkészítése az én feladatom volt, ezeket külön fogom ismertetni.

3.2.1 IQS modulok

Az IncQuery Server moduljai közül a projekt a „Tranziens Lekérdezés Kiértékelő" és a „Perzisztens Index" modulokat használja közvetlenül. Az IQS egy hibrid adatbáziskezelő megoldást alkalmaz. A perzisztens index az indexált adatok perzisztens tárolásáért felelős, a tranziens kiértékelő pedig képes a perzisztált adatok egy metszetét betölteni a memóriába és azokon rendkívül gyors lekérdezéseket futtatni a VIATRA keretrendszert segítségével. A perzisztens adatokon is van lehetőség lekérdezéseket futtatni, melynek előnye a kis memória használat, viszont ennek kiértékelése lassú folyamat. A megoldás során az úgynevezett in-memory, vagyis memóriában futó, tranziens lekérdezések lehetőségét használom.

3.2.2 Adapter modul

Az IQS kiterjesztése tehát úgynevezett adapterek elkészítésével történik. Ezeknek az adaptereknek a feladata, hogy képesek legyenek valamilyen külső adatforrás adatainak beolvasására, valamint az adatok indexálására. Egy adaptert két alapvető almodulra osztottam, adatbejáró és indexer modulokra, melyek megvalósítása lehetővé teszi azt, hogy egymástól függetlenül, akár több szálon, több példányban végezzék feladatukat aszinkron módon, a teljesítmény növelésének érdekében.

3.2.2.1 Adatbejáró modul

Az adatbejáró modul a külső adatszolgáltatók adatainak bejárásáért felel. Publikus interfészeken keresztül gyűjti be az adatokat, melyeket folyamatosan küld feldolgozásra az indexer modulnak.

3.2.2.2 Indexer modul

Az indexer komponens az adatbejáró modultól kapott adatok értelmezését, azok feldolgozását végzi megfelelő struktúra mentén. Az adatokat ezután az IQS-be indexálja.

3.2.3 Ütemező

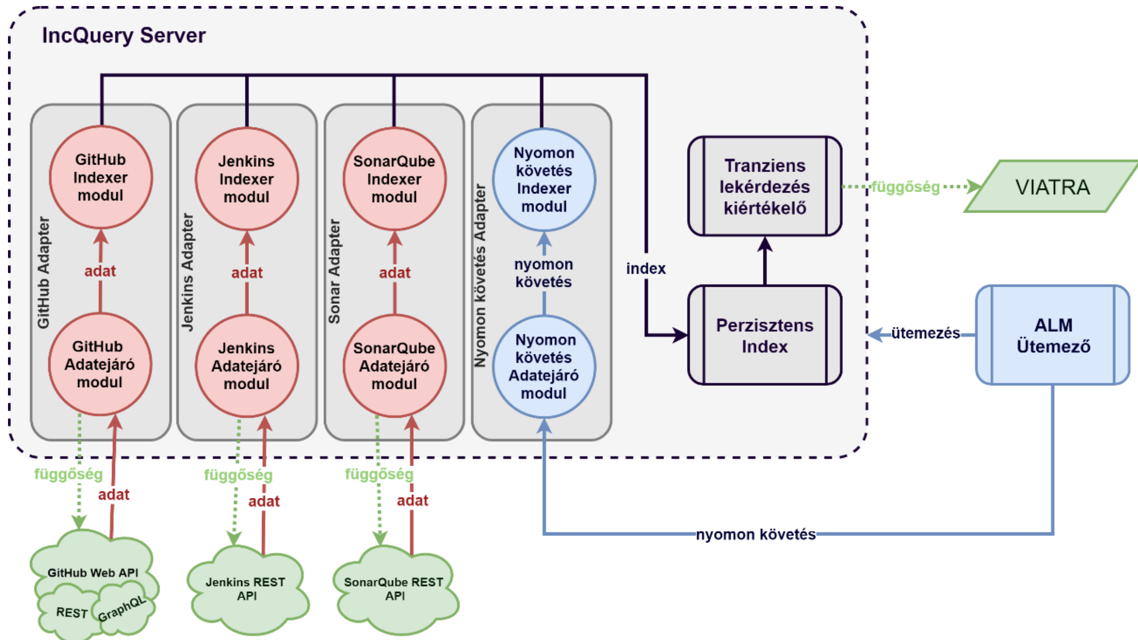
Az ütemező logikailag nem az IQS kiterjesztése, hanem egy különálló modul. Az Adapterek és így az adatok begyűjtésének és feldolgozásának ütemezését végzi, továbbá összekapcsolja az IncQuery Serveren futó folyamatokat a külső rendszerekkel. Ha több adapterünk van, melyek által kezelt adatok függőségben állnak egymással, akkor e függések kezelése is az ütemező dolga.

3.3 ALM specifikus rendszer

Az imént ismertetett általános adatintegrációs rendszer egy alkalmazás életciklu s menedzsmentet támogató verzióját készítettem el. Az eszköz a már bemutatott GitHub, Jenkins és SonarQube eszközök által szolgáltatott adatokat hivatott begyűjteni, kezelni, majd kiértékelni. Ezekkel az eszközökkel együttesen jó formán egy projekt életciklusának egészét lefedhetjük, így az integrált adathalmaz átfogó képet adhat egy adott fejlesztés minőségével kapcsolatban.

3.3.1 Architektúra

A rendszer architektúráját tehát GitHub, Jenkins és SonarQube specifikus modulokkal kell kiegészíteni. Az ALM specifikus rendszer architektúra az alábbi ábrán látható.



5. ábra: Az ALM specifikus rendszer architektúrája

Az feldolgozni kívánt adatok a GitHub Web API-ról valamint a Jenkis és SonarQube REST interfészekről származnak. Ennek a három forrásnak az IQS-hez kapcsolásához egy-egy adapterre van szükség, amelyek az ábrán a a GitHub, Jenkins és SonarQube Adapter néven jelennek meg. Adatbejáró almoduljuk a már tárgyalt módon bejárja az adatokat a megfelelő interfészen, az indexer modul pedig az adatok indexeléséért felelős.

Az ábrán látható még egy adapter. A GitHub, Jenkins és SonarQube által szolgáltatott adatok között van kapcsolat, az adaptereknek viszont egymástól függetlenül kell működniük. Ennek okán létrehoztam egy különálló úgynevezett „Nyomon követés Adaptert”, melynek feladata a különböző forrásból származó adatok közötti kapcsolatok nyomon követése és indexelése. Az ütemező modul ebben a speciális esetben nem csak az ütemezésért, hanem az imént említett relációk definiálásáért is felel.

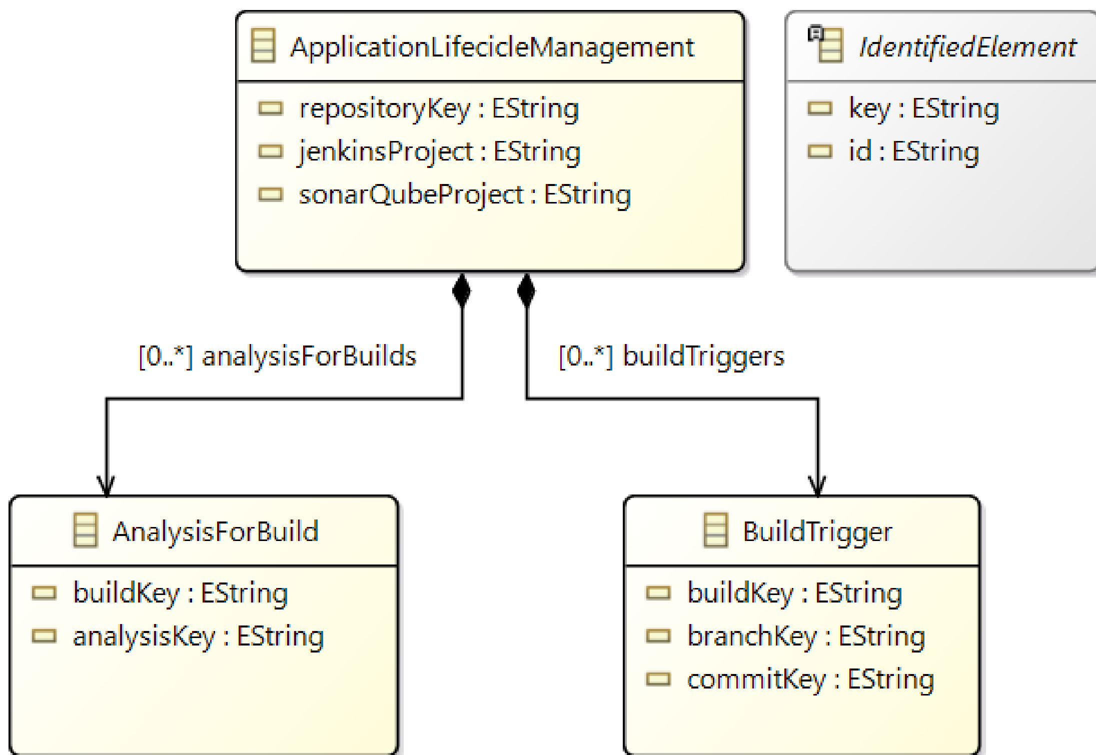
3.3.2 Metamodellek

Ahhoz, hogy az adatokat az IncQuery Server fel tudja dolgozni, illetve, hogy szakterületspecifikus lekérdezéseket írhassunk, az adatoknak szükségük van valamilyen struktúrára. Az IQS és VQL technlógiák EMF metamodellek használatát támogatják. Az IncQuery Server az EMF metamodelleket csupán az adatok sémájának definiálására használja, az EMF modellek ebben az esetben sosem példányosodnak. Az adatok tárolása és kiértékelése egy hatékony belső objektumstruktúra felett történik.

A tervezés során külön metamodellt készítettem a GitHubtól és a Jenkinstől, illetve a SonarQube-tól lekért adatok reprezentálásához. Ezeken kívül készítettem egy összefogó metamodellt is, mely az előző három adathalmazt köti össze.

3.3.2.1 ALM metamodell

Elsőnek a másik három metamodellt összefogó metamodellt, az Application Lifecycle Management metamodellt mutatom be. Ez az adatszerkezet határozza meg a különböző forrásokból származó adatok közti kapcsolatot, vagyis az integráció megvalósításában játszik szerepet. Az AML metamodell az alábbi ábrán látható.



6. ábra: Application Lifecycle Management metamodell

ApplicationLifecycleManagement

Az ApplicationLifecycleManagement nevű modell elem az ALM metamodell gyökéreleme. Attribútumaiban egy adott projekthez tartozó GitHub Repository, Jenkins projekt és SonarQube projekt azonosítóját tárolja, kollekciói pedig AnalysisForBuild és BuildTrigger modell elemeket tartalmaz.

AnalysisForBuild

Ez az elem köti össze a SonarQube által elvégzett analíziseket a Jenkins Buildjeivel. Az összekötéshez tárolja az analízis és a Build kulcsait.

BuildTrigger

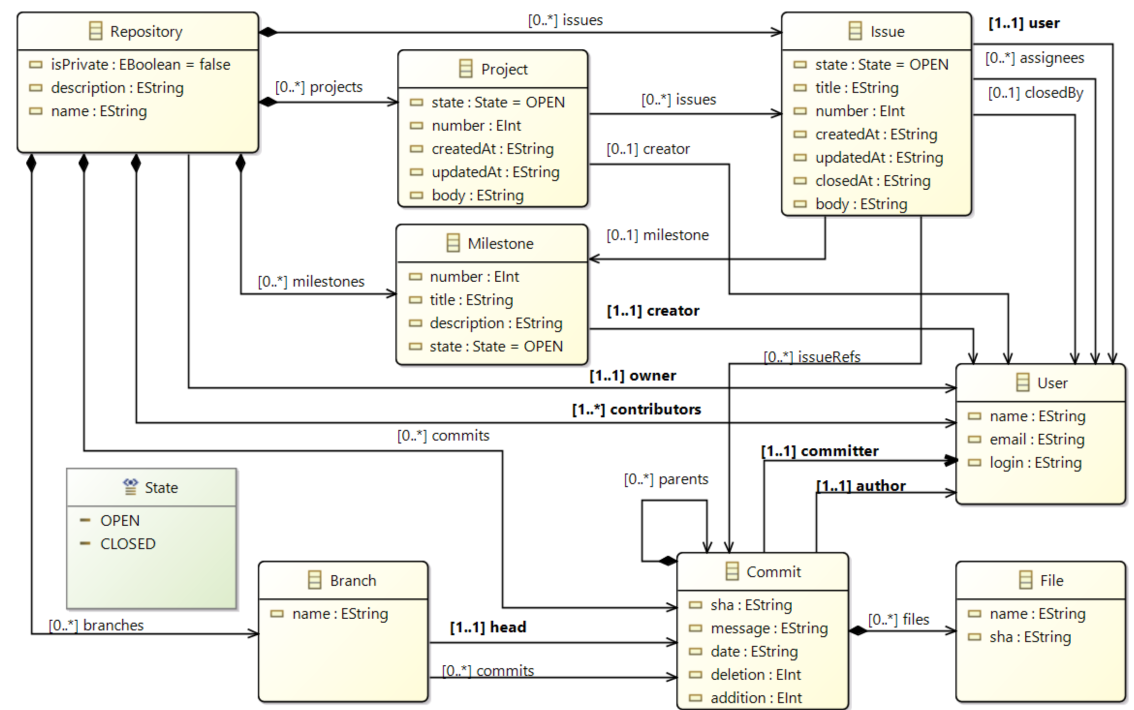
A BuildTrigger elem egy Jenkins Buildet köt össze az azt kiváltó Committal és tárolja a hozzá tartozó GitHub Branch-et is. Mezői a megfelelő Build, Branch és Commit kulcsokat tartalmazza.

IdentifiedElement

Absztrakt osztály, amelynek egy azonosító és egy kulcs mezője van. Az id attribútum egy külső forrásból kapott azonosítót tárol, míg a key attribútum egy az egész adathalmazra egyedi generált értéket. Minden további külső forrás által szolgáltatott modell elem ebből az osztályból származik.

3.3.2.2 GitHub metamodell

Az alábbi EMF metamodell a GitHub által szolgáltatott Repository adatok struktúráját írja le.



7. ábra: GitHub Repository metamodell

Repository

A metamodell gyökéreleme, amelyből az összes további modellelem elérhető tartalmazási hierarchia mentén. Ezen felül tárolja, hogy privát-e, ki a tulajdonosa, illetve leírás attribútummal rendelkezik. Közvetlenül a Branch, Commit, Project, Milestone és Issue modell elemeket, valamint a résztvevő felhasználókat tárolja.

Branch

A fejlesztés egy ágát, elkülönülő változatait tároló egység. Név mezőjén kívül a hozzá tartozó Commitok, illetve a kezdő Commitja érhető el.

Commit

A fejlesztés során létrejövő változásokat egységbe foglaló strukturális elem. Egyedi azonosítóval, üzenet, valamint dátum attribútummal rendelkezik, valamint tárolja a hozzá kapcsolódó fájlokat, a létrehozó és a végrehajtó felhasználót. Minden Commit továbbiakra referálhat, így épül fel a Repository Commit hierarchiája.

File

A File modellelem az egyes változásokhoz tartozó fájlokat írják le, egy egyedi azonosító, valamint név segítségével.

Project

Egy fejlesztés során elkülönülő projekteket leíró modellelem, mely a projektek állapotát, sorszámát, keletkezési és módosítási időpontjait, valamint a hozzá tartozó leírást tartalmazza. Egy projekt ismeri a létrehozóját, valamint a hozzá tartozó feladatokat, Issue-kat tartalmazza.

Issue

Az Issue elem a projekt során felmerülő feladatokat és problémákat írja le. Sorszám és cím attribútumai vannak, body mezőjében pedig az Issue leírása található. Ezen felül tárolja a nyitásának, módosításának és bezárásának idejét, valamint az állapotát, amely a State nevű enumeráció alapján OPEN, tehát nyitott és CLOSED, tehát zárt értékeket vehet fel. Az Issue továbbá ismeri az őt létrehozó, a hozzárendelt valamint az őt bezáró felhasználót, valamint tárolja, hogy melyik Milestone-ba tartozik. Az issueRefs referencia mentén érhetőek el azok a Commitok, amelyeket az adott Issue-hoz kapcsoltak.

Milestone

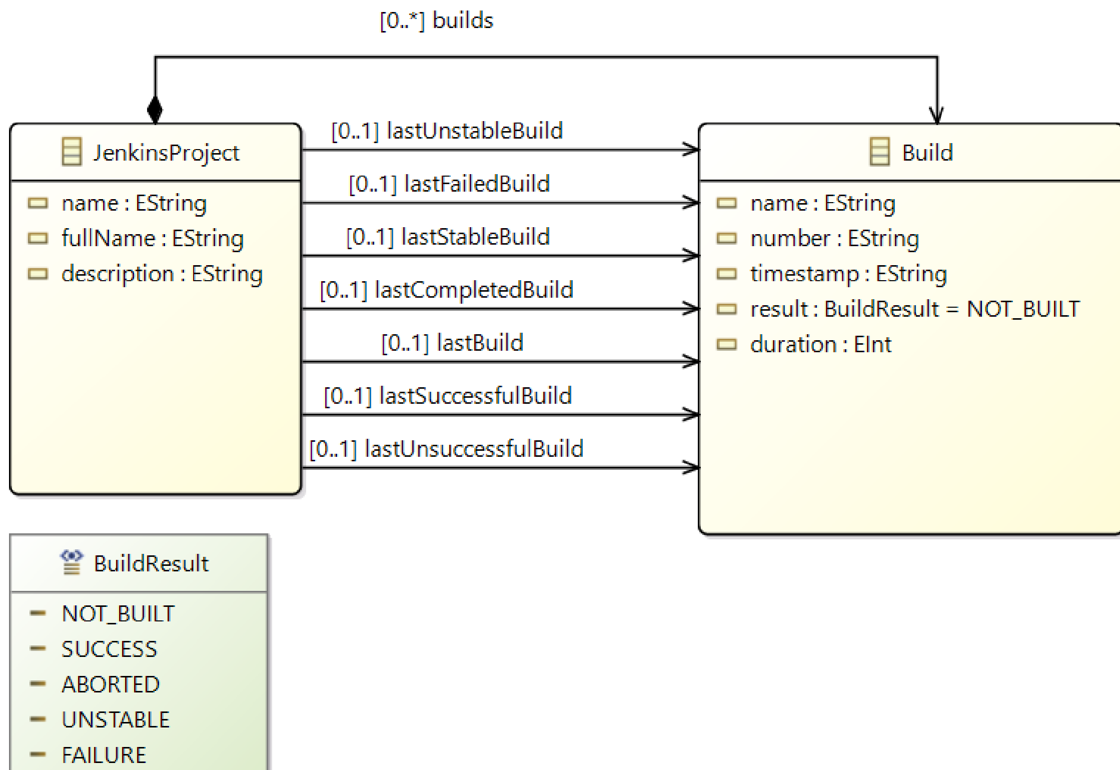
A Milestone, azaz mérföldkő a fejlesztés egy elérni kívánt pontját jelöli. Ennek reprezentálásaként a Milestone elemnek sorszám, cím, leírás és állapot értékei vannak, amely itt is nyitott vagy zárt értékű lehet. A Milestone továbbá mutat az őt létrehozó felhasználóra.

User

A projektben résztvevő GitHub felhasználókat reprezentálja. Név, azonosító, belépési azonosító és emailcím mezők jellemzik.

3.3.2.3 Jenkins metamodell

A következő ábrán a Jenkis szerver által szolgáltatott adatok struktúráját leíró metamodell látható.



8. ábra: Jenkins Build metamodell

JenkinsProject

A metamodell gyökéreleme, mely egy Jenkins projektet reprezentál. Név, teljes név és leírás mezői vannak, valamint tárolja a hozzá tartozó Buildeket. Az ábrán látható további referenciák különböző korábbi kitüntetett Buildekre mutatnak.

Build

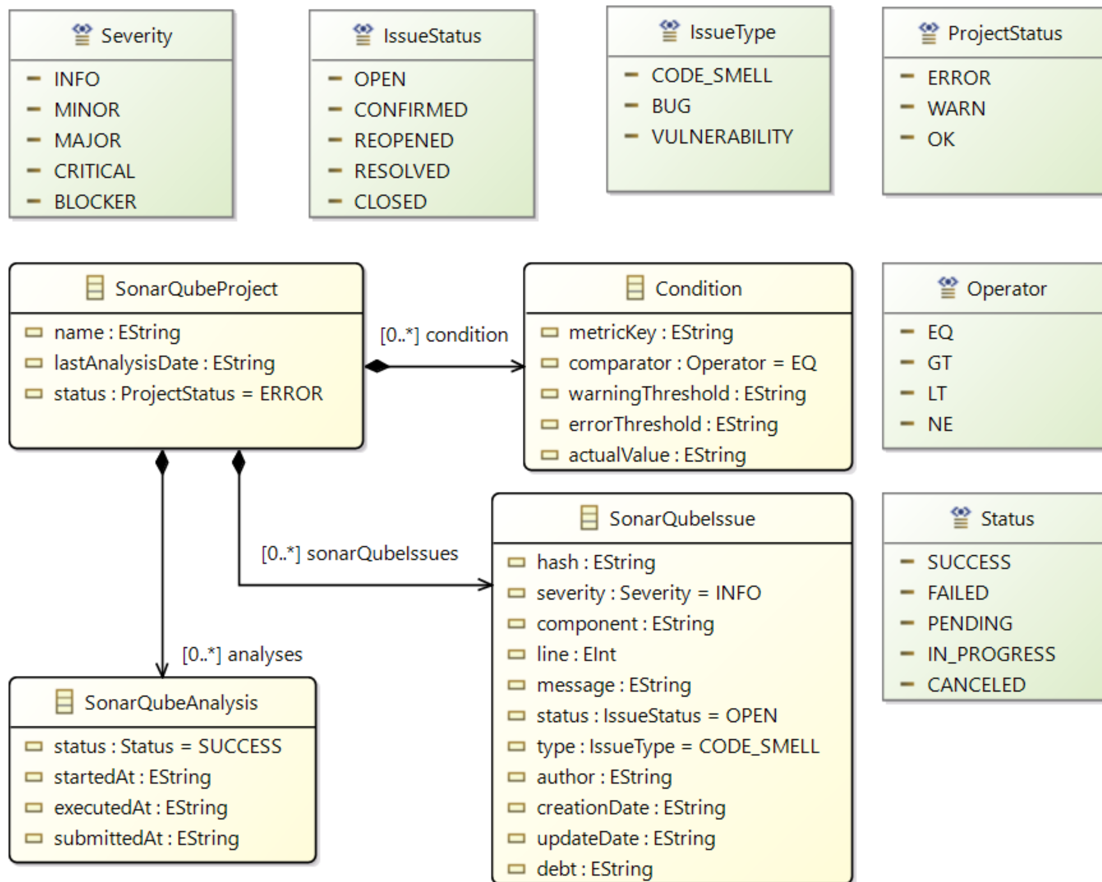
A Build elem egy Jenkins buildet ír le, tárolja annak nevét és számát, időbélyegét, hosszát valamint eredményét.

BuildResult

A Build eredményét leíró enumeráció. Lehetséges értékei nem buildelt, sikeres, megszakított, instabil és hibás.

3.3.2.4 SonarQube metamodell

A SonarQube eszközből kinyert adatok közötti kapcsolatot az alábbi ábrán látható EMF metamodell írja le.



9. ábra: SonarQube metamodell

SonarQubeProject

A SonarQubeProject elem a metamodel gyökéreleme, mely analízis projektet ábrázol, annak nevével és utolsó analízisének idejével. A projekt állapotát tekintve lehet rendben és lehet hibás, valamint tartozhat hozzá figyelmeztetés. A gyökérelemből elérjük továbbá a projekthez tartozó analíziseket, issue-kat és feltételeket.

SonarQubeAnalysis

Egy lefuttatott statikus analízist reprezentál. Tárolja az analízis állapotát, amely a Status enumeráció értékeit veheti fel, valamint az analízis kezdeményezésének, indításának és végrehajtásának időpontját.

SonarQubeIssue

A SonarQubeIssue hasonlóan a GitHub esetén megismert Issue-hoz lehet hiba, figyelmeztetés vagy egyszerű információ is. Az Issue-nak van azonosítój és üzenete, tárolja ki készítette, hol helyezkedik el a projektben és a kódban, valamint, hogy mikor hozták létre és mikor módosították. Egy Issue állapotát tekintve lehet nyitott, újranyitott, elfogadott, megoldott és zárt. Ezen felül a típusa pedig hiba, sebezhetőség vagy code smell lehet.

Condition

Ez a metamodell elem olyan feltételeket képvisel, melyeknek teljesülniük kell, ahhoz, hogy az analízis eredménye sikeres lehessen. Egy feltételnek van metrika azonosítója, aktuális értéke és küszöbértékei, valamint egy összehasonlító logikai függvénye. Ha az aktuális érték az adott logikai függvénnyel megsérti a figyelmeztetési küszöbértéket, akkor a státusz WARN, azaz figyelmeztetés értéket mutat, ha pedig a hiba küszöbértéket sérti meg, akkor a státusz ERROR, azaz hiba értéket vesz fel.

4 Megvalósított rendszer

Ebben a fejezetben a tervek alapján megvalósított rendszert mutatom be. Példákon keresztül részletezem az adatintegrációs folyamatot. Először az adatok bejárásól majd pedig az adatok indexeléséről ejtek szót.

4.1 Esettanulmány

Az adatok bejárásának, indexelésének és kiértékelésének folyamatát egy esettanulmány segítségével fogom bemutatni. Az esettanulmányom tárgyául egy közepes méretű publikus projektet, a VIATRA projekt Massif keretrendszerét választottam. A Massif egy MATLAB Simulink integrációs keretrendszer az EMF-et használó modellezőeszközök számára. Segítségével Simulink modellek egyszerű kezelésére van lehetőség, melyet kétirányú EMF-Simulink transzformációval valósít meg.

4.1.1 Massif GitHub Repository

A Massif projekt fejlesztése a GitHub verziókezelő és projektmenedzsment felületen történik. A fejlesztéshez tartozozó repository „viatra/massif"[1] néven érhető el, melynek 7 elágazásán az összcommitszám, azaz az összes branch összes commit-jának a száma 2000 commit köré tehető, melyekhez összesen 105 issue és 43 pull request tartozik.

4.1.2 Massif Jenkins Projekt

A Massif projekthez Jenkins projekt[2] is tartozik, amely az IncQuery Labs build szerverén érhető el. A Jenkins a projekt folyamatos integrációjának részeként összeállítja, futtatja és teszteli a projektet, annak változásai nyomán. A Massif-hoz a Jenkins szerveren jelenleg 51 build tartozik.

[1] https://github.com/viatra/massif

[2] https://build.incquerylabs.com/jenkins/job/Massif/

4.1.3 Massif SonarQube projekt

A Massif Jenkins projektjéhez hozzávan kapcsolva egy SonarQube projekt[3], amelyhez a SonarQube szerver, szintén az IncQuery Labs build szerverén található. Bizonyos Jenkins build-ek SonarQube analíziseket indítanak, melyek a projekt forráskódját ellenőrzik, hibák után kutatva. A Massifhoz 9 analízis, valamint összesen 594 SonarQube issue tartozik.

4.2 Adatok bejárása

Az integrációs folyamat első részfeladata az adatok bejárása az integrálni kívánt eszközök által nyújtott nyilvános interfészeken keresztül. Ezt a feladatot a 3.2.2. fejezetben bemutatott Adapter modulok adatbejáró moduljai végzik. Az integrálni kívánt eszközök webes interfészeket nyújtanak. A Jenkins és SonarQube esetében REST, a GitHub esetán pedig REST és GraphQL API-kat használtam. A folyamatot a Viatra/Massif nyílt projekten mutatom be.

4.2.1 GitHub Repository bejárása

Megoldásom során tehát a GitHub webes interfészei közül a REST, illetve a GraphQL interfészeket is használtam. A továbbiakban a Massif GitHub Repository bejárást mutatom be.

4.2.1.1 GitHub REST API

A GitHub V3 nevű API-ja egy REST API, amelyhez teljes dokumentáció tartozik példákkal kiegészítve.[11]

Az első lekérdezés magának a repo adatainak a lekérdezése, ehhez az alábbi elkérdezést kell elküldeni.

```
GET https://api.github.com/repos/viatra/massif
```

[3] https://build.incquerylabs.com/sonar/dashboard?id=hu.bme.mit.massif%3Ahu.bme.mit.massif.parent

A lekérdezésre az alábbi JSON válasz érkezik. Tartalmazza a Repository információit, mint annak neve, tulajdonosa és például létrehozásának dátuma.


```
{
	"id": 25926236,
	"name": "massif",
	"full_name": "viatra/massif",
	"private": false,
	"owner": {
		"login": "viatra",
		"id": 18396431,
		"type": "Organization",
		"site_admin": false
	},
	"created_at": "2014-10-29T15:04:03Z",
	"updated_at": "2018-10-11T09:35:11Z",
	"pushed_at": "2018-10-11T09:37:45Z"
	//...
}
```

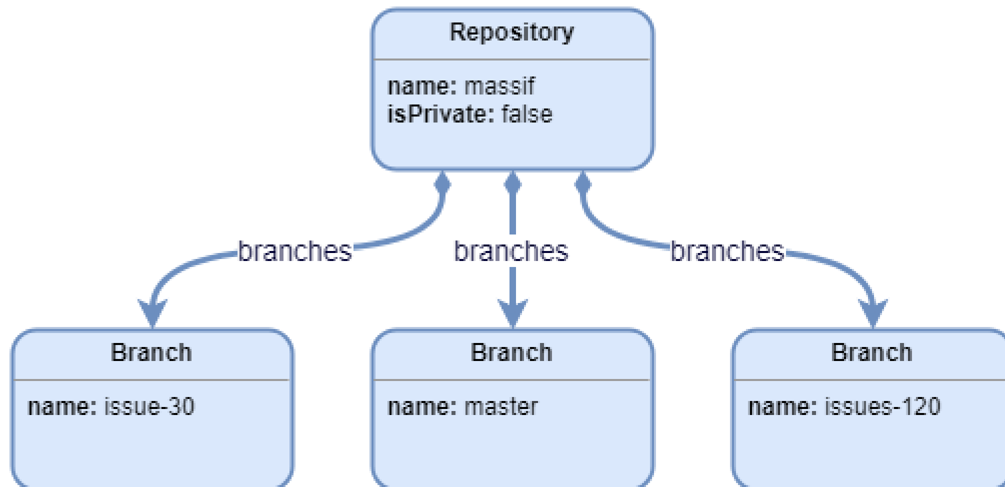

Ezzel párhuzamosan lekérdezzük a repository-hoz tartozó branch-eket, issue-kat és a résztvevő felhasználókat. A branch-ek lekérdezéséhez az alaábbi GET kérést kell elküldenünk.

```
GET https://api.github.com/repos/viatra/massif/branches
```

A lekérdezés eredménye az alábbi JSON dokumentum, amely egy listában repo-hoz tartozó branch-ek nevét és legfrissebb commit-juk hash értékét tartalmazza.


```
[
	{
		"name": "issue-30",
		"commit": {
			"sha": "6e69af8deb21b166f621f7155fd866a49de59af7"
		}
	},
	{
		"name": "issue-120",
		"commit": {
			"sha": "23363446d8488ac21d6cebed2f515e31d685ff46"
		}
	},
	{
		"name": "master",
		"commit": {
			"sha": "41bdde9eaad5ce7b9cf289efbc522d933ba17b5a"
		}
	}
]
```

Ezekkel az információkkal elkezdhetjük a branch-ekhez tartozó commit-ok lekérdezését, melyet a következő, 4.2.1.2. fejezetben mutatok be, mivel ahhoz GraphQL lekérdezést használtam. A bejárt branch-ek adatait feldolgozva az alábbi példányok jönnek létre a belső modellben.



10. ábra: Massif repository és branch példányok

Az repo issue-inak lekérdezése „/issues” címen tehető meg, amihez a teljes lekérdezés az alábbi.

```
GET https://api.github.com/repos/viatra/massif/issues
```

A válasz az issue-k listája, melyben nem csak olyan adatok szerepelnek, mint az issue száma, címe és státusza, hanem a hozzá kötött felhasználó is.


```
[
	{
		"id": 346874317,
		"number": 117,
		"title": "Update provided example .simulink models to be in...",
		"state": "closed",
		"assignees": [
			{
				"login": "imbur",
				"id": 2046210,
				"type": "User"
			}
		]
		"created_at": "2018-08-02T06:18:57Z",
		"updated_at": "2018-08-15T10:08:24Z",
		"closed_at": "2018-08-15T10:08:24Z",
		"body": "Changes to the metamodel recently has been made as..."
	},
	{
		"id": 64722203,
		"number": 30,
		"title": "Properties of port need to be preserved",
		"user": {
			"login": "phanthanhliem",
			"id": 10292378,
			"type": "User"
		},
		"state": "open",
		"assignees": [],
		"created_at": "2015-03-27T08:53:13Z",
		"updated_at": "2018-07-11T13:33:16Z",
		"closed_at": null,
		"author_association": "NONE",
		"body": "All properties of port..."
	}
]
```

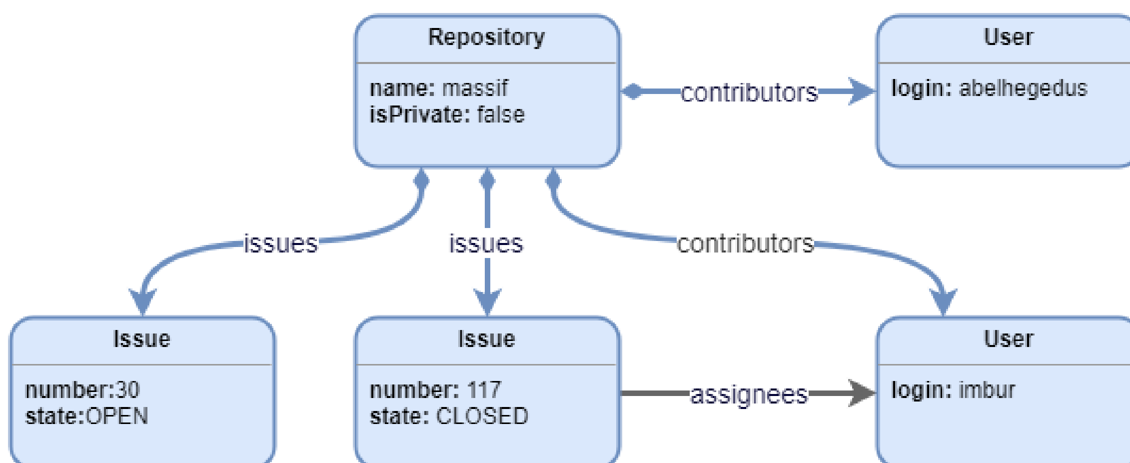

A következő kéréssel lekérdezhető a repository-hoz tartozó összes közreműködő listája.

```
GET https://api.github.com/repos/viatra/massif/contributors
```

A válasz üzenet az alábbi.

```
[
	{
		"login": "imbur",
		"id": 2046210,
		"type": "User",
		"site_admin": false,
		"contributions": 211
	},
	{
		{
		"login": "abelhegedus",
		"id": 716108,
		"type": "User",
		"site_admin": false,
		"contributions": 167
	}
]
```

Az issue-k és a közreműködők adatainak példánymodell szintű reprezentációját az alábbi ábra mutatja.



11. ábra: Issue-k és közreműködők példányai

A példában megjelenő egyik issue a 30-as, mely nyitott állapotban van, míg a 117-es issue egy zárt issue, amelynek felelős fejlesztője az "imbur" felhasználónevű fejlesztő.

4.2.1.2 GitHub GraphQL API

A GitHub a REST interfészen kívül egy GraphQL API-t is nyújt v4 néven. A v4-hez is tartozik dokumentáció, bár ez kevésbé részletes, mint a v3-é, inkább a sémáról ír, valamint online kipróbálási lehetőséget nyújt.[10]

A GraphQL egy típusos lekérdező nyelv webes API-khoz. A GraphQL lekérdezések mindig egy az adott adatokon értelmezett sémán alapszik, mely lehetővé

teszi a kliensek számára, hogy a sémát használva a lekérdezésben leírják pontosan milyen adatokat várnak a szervertől. A szerver csak a lekérdezett adatokat küldi vissza a válaszban, szemben egy REST lekérdezéssel, melyben csak elérési utak adhatók meg, és az ahhoz tartozó összes adat része a válaszüzenetnek. A GraphQL nagy előnye tehát a testreszabhatóság.

Mivel egy GraphQL lekérdezésben az adathierarchia több szintjét is bejárhatjuk így sok adatot tartalmazó listák lekérdezése célravezetőbb, mint a REST API. Listákból egyszerre száz elemet kérdezhetünk le egyszerre, de ezzel esetenként akár 100 REST kérést helyettesíthetünk. Emiatt a GitHub repository bejárása során egy adott branch commit-jainak és a commit-okhoz tartozó fájlok bejárásához a GitHub GrapQL interfészét használtam.

A Massif projekt master branch-ének első száz commit-ja, annak attribútumai, és a hozzá tartozó fájlok az alábbi GrapQL mintával kérdezhetőek le.

```
{ repository(name: "massif", owner: "viatra") {
    ref(qualifiedName: "master") {
      target {... on Commit {
          history(first: 100) {
            edges { node {
                oid
                message
                committer{user{login}}
                author{user{login}}
                parents(first: 100){
                     edges{node{id}}
                }
                committedDate
                additions,deletions
                tree{entries{oid,name,type}}
            }}
          }
        }}
    }
}}
```

A válasz a vártaknak megfelelően pontosan azokat az adatokat és olyan struktúrában tartalmazza, ahogyan a GraphQl mintában le lett írva.


```
{"data": {"repository": {
     "ref": {"target": {
          "history": {"edges": [
               {"node": {
                    "oid": "00a799370f6da8fc9c42487e74d47a13cb48aca2",
                    "message": "Merge pull request #138 from
                               viatra/issue 117 Updating .simulink
                               models #117",
                    "committer": {"user": null},
                    "author": { "user": { "login": "imbur"}
                    },
                    "parents": { "edges": [
                         {"node": {"id": "...N2FkNQ=="}},
                         {"node": {"id": "...g2NA=="}}
                    ]},
                    "committedDate": "2018-08-15T10:08:23Z",
                    "additions": 47560,
                    "deletions": 7607,
                    "tree": {"entries": [
                         {"name": "bus_goto.simulink"},
                         {"name": "bus_simple.simulink"},
                         {"name": "bus_unnamed.simulink"}
                    ]}
               }}
          ]}
     }}
}}}
```

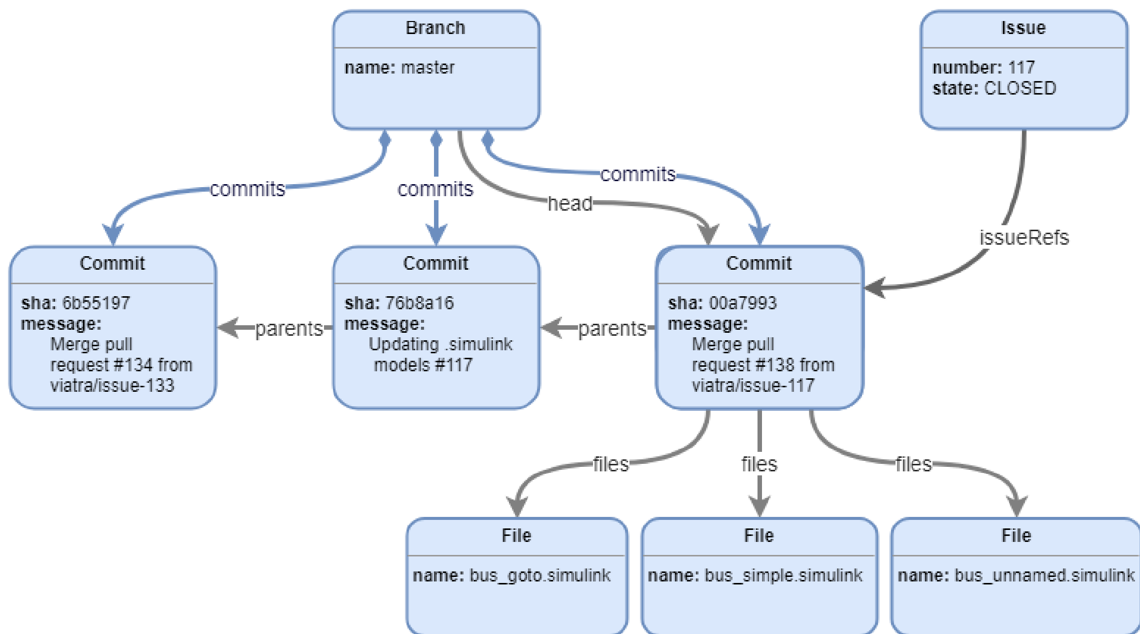

A GitHub repository bejárása során továbbá az issue-k és commit-ok közötti referenciák bejárásához használtam GraphQL lekérdezést. Ehhez egy olyan mintát írtam, mely lekérdezi az issue-kat, de azoknak csak az azonosítójára, a számára és az úgynevezett idővonalára kíváncsi. Egy issue idővonala tartalmazza az issue-n végzett módosítások listáját, melyek közül ez a minta a hozzárendelt commit-okra fókuszál.

```
{ repository(name: "github-indexer", owner: "incquerylabs") {
    issues(first: 100){
          edges{node{
               id
               number
               timeline(first: 100){
                    edges{node{
                         ... on ReferencedEvent{
                              commit{oid}
                         }
                    }}
               }
          }}
     }
}}
```

A válasz üzenet a repository első 100 issue-ját és az azokhoz referált commitok hash értkét tartalmazza a megadott struktúrában.

```
[
    {"node": {
        "id": "MDU6SXNzdWU2NDcyMjIwMw==",
        "number": 30,
        "timeline":{"edges": [
            {"node":{"commit": {"oid": "bb029ce"}}},
            {"node":{"commit": {"oid": "de30af0"}}}
        ]}
    }},
    {"node": {
        "id": "MDU6SXNzdWUzNDY4NzQzMTc=",
        "number": 117,
        "timeline":{"edges": [
            {"node":{"commit": {"oid": "76b8a16"}}},
            {"node":{"commit": {"oid": "00a7993"}}}
        ]}
    }}
]
```

A példában szereplő bejárt commit-ok és a hozzájuk tartozó issue referenciák a rendszermodellben a következő gráfot alkotják.

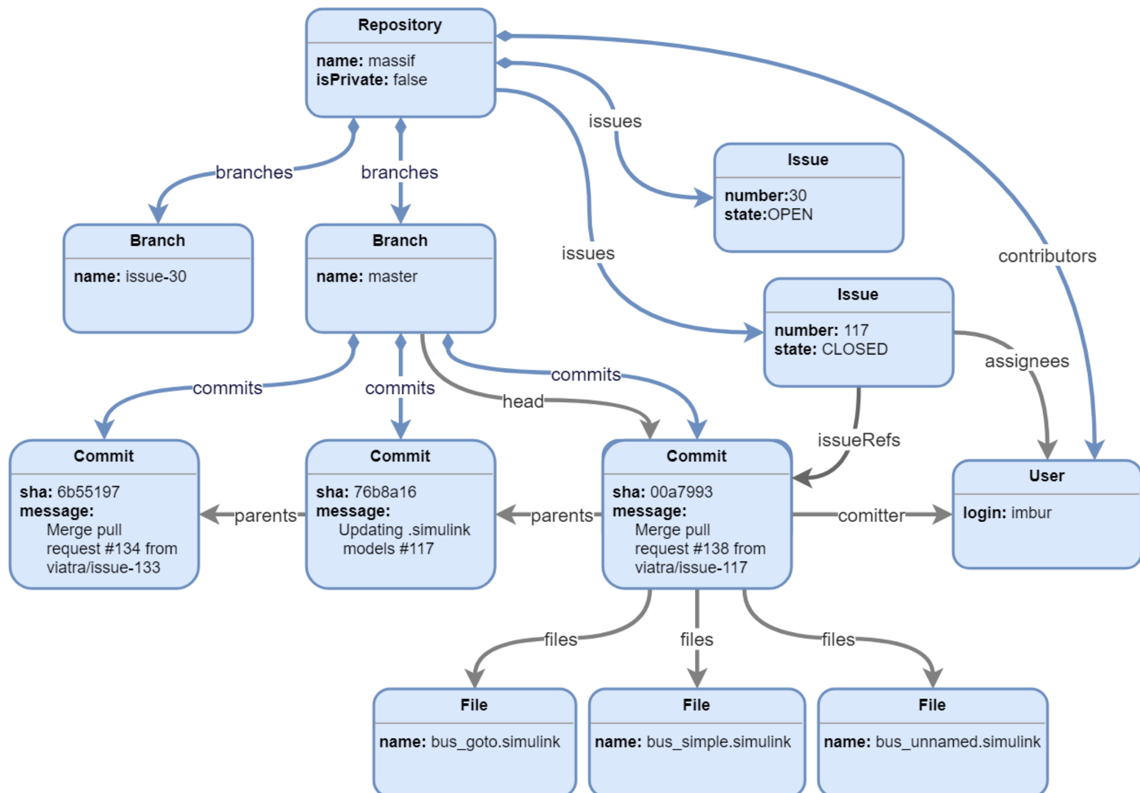


12. ábra: Massif commit és issue példányai

A példában a „master" branch-nek három commit-ját látjuk, melyek egymásra épülnek. A branch jelenlegi fő commit-jához három fájl tartozik, továbbá a 117-es issue-t kötötték hozzá.

4.2.1.3 GitHub példánymodell

Az alábbi ábra a Massif GitHub repository egy részfáját, a GitHub metamodell egy példánymodelljét ábrázolja, mely az összes bemutatott lekérdés adatait ábrázolja.



13. ábra: Massif GitHub példánymodell

A modell felépítése a következő. A példánymodell gyökéreleme a „massif" nevű Repository, melynek vannak Branch-ei, mint a „master" és az „issue-30", vannak Issue-i, mint a 30-as és a 117-es Issue, illetve közreműködői is vannak, melyek közül a jelenlegi modell egyet tartalmaz „imbur" felhasználói névvel. Ez a felhasznó a 117-es Issue-hoz van kötve, mint megbízott. A „master" Branch-nek három Commitja van ábrázolva, melyek közül az egyik az utolsó Commit, a „master"-en. Mint látható a Commithoz fájlok is tartoznak, „imbur" felhasználó hozta létre és a 117-es Issue-hoz van rendelve.

4.2.2 Jenkins projekt bejárása

Az eszköz a GitHub Repository-hoz kapcsolt Jenkins projekt bejárását is elvégzi. Ehhez a Jenkins szerver által nyújtott JSON REST API-t használja, melyhez külön dokumentáció nem tartozik, viszont minden a Jenkins által nyújtott oldal alján megtalálható az adott információ eléréséhez szükséges REST hívás linkje.

A Massif projekt master nevű Jenkins projektje az alábbi hívással kérdezhető le.

```
GET https://build.incquerylabs.com/jenkins/job/Massif/job/master/api/json
```

A válasz egy JSON objektum, mely a projekt információit tartalmazza, úgy mint név, leírás, a build-ek listája és a különböző kitüntetett build-ek.

```
{
      "description": null,
      "fullName": "Massif/master",
      "name": "master",
      "builds": [
            {"number": 86},
            {"number": 85},
            {"number": 71}
      ]
      "firstBuild": {"number": 67,}
      "lastBuild": {"number": 86},
      "lastCompletedBuild": {"number": 86},
      "lastFailedBuild": null,
      "lastStableBuild": {"number": 86},
      "lastSuccessfulBuild": {"number": 86},
      "lastUnstableBuild": null,
      "lastUnsuccessfulBuild": null,
}
```

A projekt build-jeinek listáját megkapva bejárhatóak az egyes build-ek is. A 71-es build lekérdezéséhez az alábbi GET hívás elküldése szükséges.

```
GET
https://build.incquerylabs.com/jenkins/job/Massif/job/master/71/api/json
```

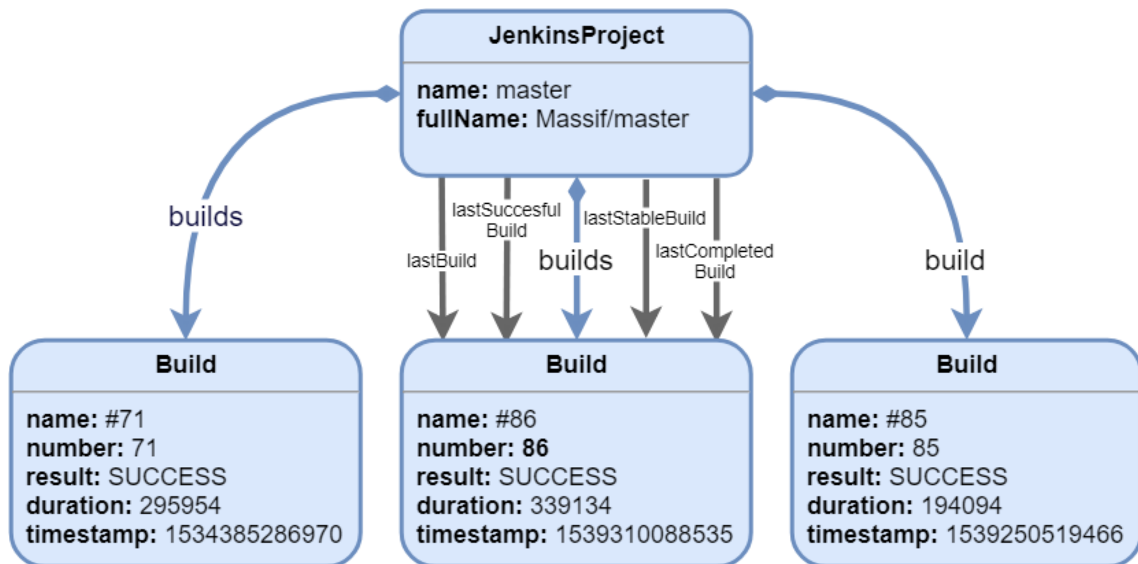
A hívásra érkezett válasz az alábbi. Ebben olyan információk lelhetőek fel, mint a build neve, száma, hossza és eredménye, valamint a build-hez tartozó GitHub branch neve és a build-et elindító commit azonosítója.


```
{
  "actions" : [
    {
      "lastBuiltRevision" : {
        "branch" : [
          {
            "SHA1" : "00a799370f6da8fc9c42487e74d47a13cb48aca2",
            "name" : "master"
          }
        ]
      }
    },
    {
      "_class" : "hudson.plugins.sonar.action.SonarAnalysisAction",
      "ceTaskId" : "AWVAgbZ9ZWS5sd5jLC8z",
      "serverUrl" : "https://build.incquerylabs.com/sonar",
    }
  ],
  "description" : null,
  "displayName" : "#71",
  "duration" : 295954,
  "fullDisplayName" : "Massif » master #71",
  "id" : "71",
  "number" : 71,
  "queueId" : 291587,
  "result" : "SUCCESS",
  "timestamp" : 1534385286970,
}
```


Amennyiben a build-hez tartozik SonarQube analízis, akkor az action-ök között megtaláljuk az analízis azonosítóját is.

4.2.2.1 Jenkins példánymodell

A következő ábra a Jenkins metamodell egy példánymodelljét, a Massif projekthez tartozó Jenkins projekt egy részletét ábrázolja. Az ábrán szereplő példánymodell az imént bemutatott lekérdezések válaszban adott adatainak belső reprezentációja.



14. ábra: Massif Jenkins példánymodell

A Jenkins példánymodell gyökéreleme a „Massif/master” nevű JenkisnProject. A projekthez az ábrán három Build tartozik, a 71, a 86 és a 85-ös számú. Mindhárom Build sikers volt és például az is látható, hogy a 71 és a 86 közel azonos ideig futott, a 85-ös jóval kevesebb ideig. A modell által mutatott állapot szerint, a 86-os Build a projekt utolsó, utolsó sikeres, utolsó stabil és utolsó befejezett Buildje is.

4.2.3 SonarQube projekt bejárása

Ahogy arról már esett szó, a Jenkins build-ekhez és a GitHub repository-hoz tartozhatnak SonarQube analízisek. Az integrációs eszköz ezeknek a bejárását a SonarQube szerver által nyújtott REST API-n végzi. A SonarQube szerverre bejelntkezve elérhető az API dokumentációja példa lekérdezésekkel és válaszokkal együtt.

Az Massif projekt azonosítója „hu.bme.mit.massif:hu.bme.mit.massif.parent”. A hozzá tartozó SonarQube projekt az alábbi GET kéréssel kérdezhető le.

```
GET
https://build.incquerylabs.com/sonar/api/project_analyses/search?project=h
u.bme.mit.massif:hu.bme.mit.massif.parent
```

A válasz JSON üzenet a projekthez tartozó analízisek listáját tartalmazza. A lista megadja az analízisek azonosítóját, indításuk dátumát, valamint a hozzájuk tartozó eseményeket.


```
{
     "analyses": [
          {
               "key": "AWWNwaBux456XUji1tKf",
               "date": "2018-08-31T04:12:48+0200",
               "events": []
          },
          {
               "key": "AWVAgbgax456XUji1qTA",
               "date": "2018-08-16T04:12:20+0200",
               "events": []
          }
     ]
}
```

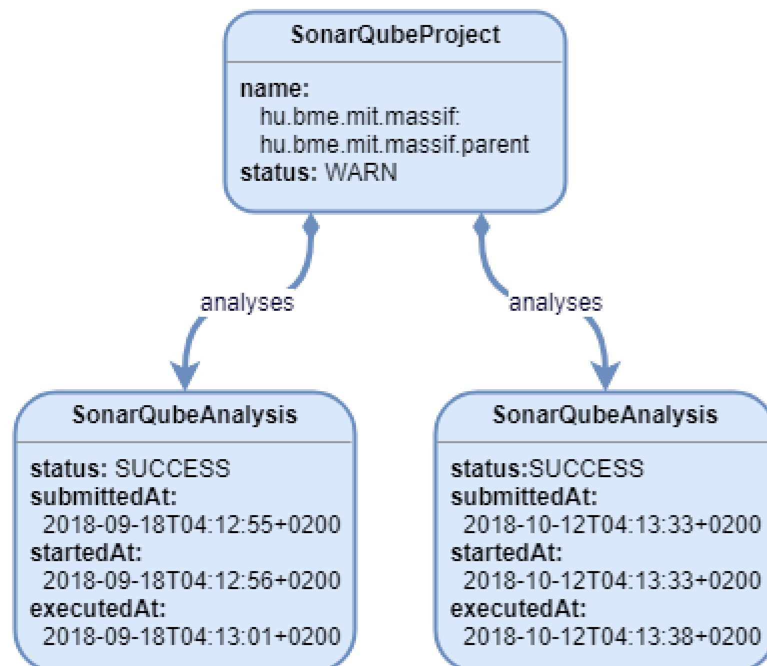

Az analízisek részletesebb lekérdezése az alábbi lekérdezéssel a Jenkins build által szolgáltatott task azonosítóval lehetséges.

```
GET
https://build.incquerylabs.com/sonar/api/ce/task?id=AWVAgbZ9ZWS5sd5jLC8z
```

A válaszban az analízis típusa, eredménye és időbeli attribútumai találhatók.


```
{
     "task": {
          "id": "AWVAgbZ9ZWS5sd5jLC8z",
          "type": "REPORT",
          "componentKey": "hu.bme.mit.massif:hu.bme.mit.massif.parent",
          "componentName": "Massif",
          "componentQualifier": "TRK",
          "analysisId": "AWVAgbgax456XUji1qTA",
          "status": "SUCCESS",
          "submittedAt": "2018-08-16T04:12:47+0200",
          "startedAt": "2018-08-16T04:12:47+0200",
          "executedAt": "2018-08-16T04:12:52+0200",
          "executionTimeMs": 5165,
     }
}
```

Az bejárt analízisek adatai az példánymodellben az alábbiak szerint épülnek fel. A projekthez kettő sikeres analízis tartozik.



15. ábra: SonarQube analízisek példányai

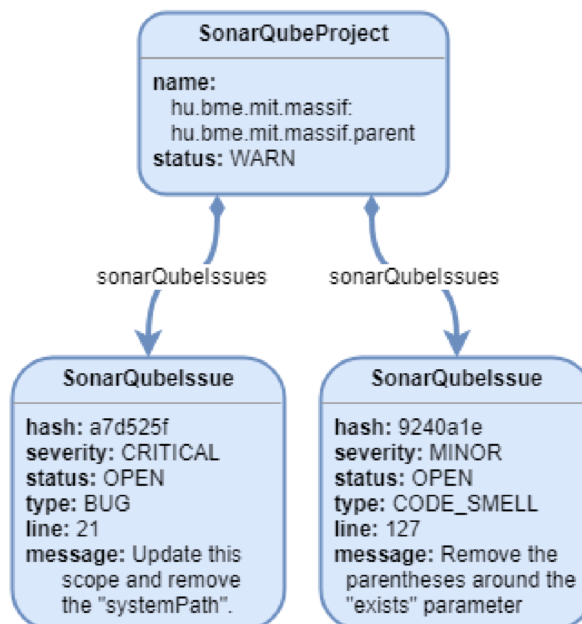
Egy SonarQube projekthez úgynevezett Issue-k tartoznak, melyek hibákat, figyelmeztetéseket vagy úgynevezett „code-smell”-eket jelölhetnek a kódban. Ezeket az alábbi GET hívássla kérdezhetjük le.

```
GET https://build.incquerylabs.com/sonar/api/issues/search
```

A válasz az issue-kat tartalmazó JSON lista, mely az adott issue 2.4.1.3. fejezetben ismertetett attribútumait tartalmazza.


```
{
     "issues": [
          {
               "key": "AWZg7YRax456XUji10yR",
               "severity": "CRITICAL",
               "component":"hu.bme.mit.massif:massifmatlabengine:pom.xml",
               "project": "hu.bme.mit.massif:hu.bme.mit.massif.parent",
               "line": 21,
               "hash": "a7d525f8eddadfdc2ec47ce13afafbf4",
               "status": "OPEN",
               "message":"Update this scope... "
               "debt": "5min",
               "creationDate": "2018-10-11T04:20:40+0200",
               "updateDate": "2018-10-11T04:20:40+0200",
               "type": "BUG",
          },
          {
               "key": "AWZrNZEgx456XUji102I",
               "severity": "MINOR",
               "component": "TransformationType.java",
               "project": "org.eclipse.viatra.examples.cps.parent",
               "line": 127,
               "hash": "9240a1ea92bd54ba42425a672fb10775",
               "status": "OPEN",
               "message": "Remove the parentheses...",
               "debt": "2min",
               "creationDate": "2018-10-13T04:15:03+0200",
               "updateDate": "2018-10-13T04:15:03+0200",
               "type": "CODE_SMELL",
          }
     ]
}
```


A megkapott adatok a rendszer belső modelljének példányaiként az alábbi módon épülnek fel. A példánymodellben jelenleg kettő issue tartozik a Massif SonarQube projektjéhez.



16. ábra: SonarQube issue-k példányai

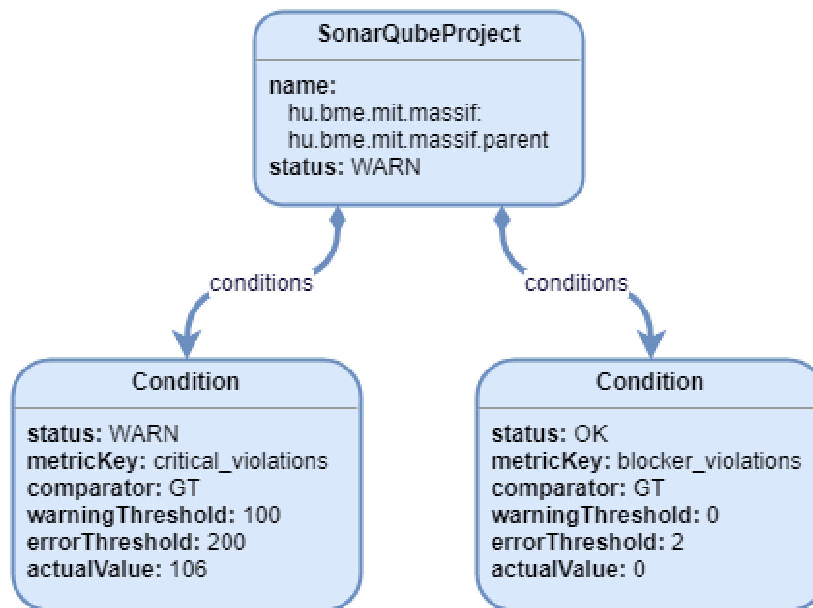
Az analíziseken és issue-kon kívül a projekt státuszát is lekérdezzük az alábbi kéréssel.

```
GET
https://build.incquerylabs.com/sonar/api/qualitygates/project_status?proje
ctKey=hu.bme.mit.massif:hu.bme.mit.massif.parent
```

A válasz megadja a Massif-hoz tartozó SonarQube projekt státuszát, mely jelen esetben egy figyelmeztetés.

```
{
	"projectStatus": {
		"status": "WARN",
		"conditions": [
			{
				"status": "OK",
				"metricKey": "blocker_violations",
				"comparator": "GT",
				"warningThreshold": "0",
				"errorThreshold": "2",
				"actualValue": "0"
			},
			{
				"status": "WARN",
				"metricKey": "critical_violations",
				"comparator": "GT",
				"warningThreshold": "100",
				"errorThreshold": "200",
				"actualValue": "106"
			}
		]
	}
}
```

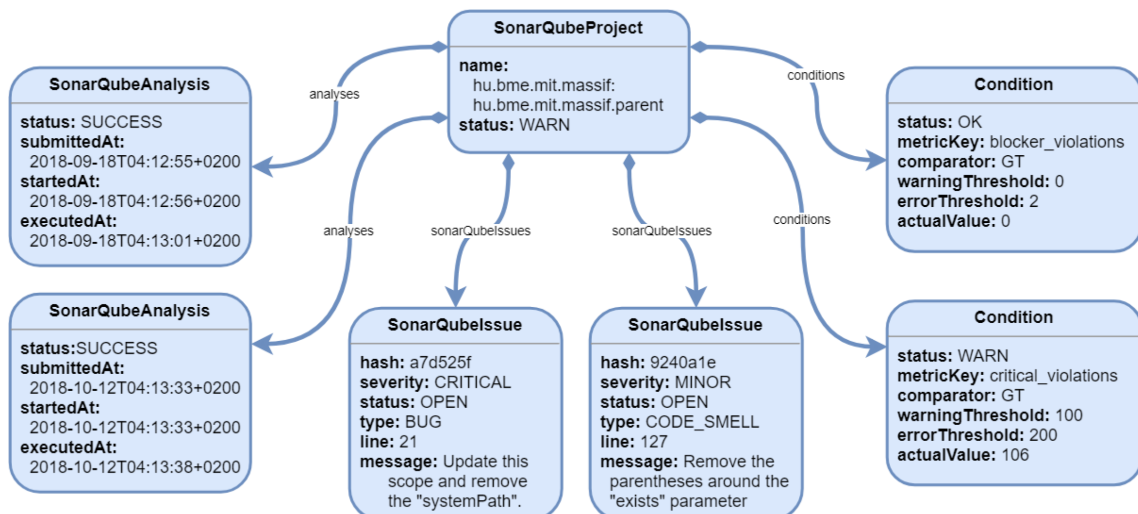
A válasz továbbá tartalmazza a projekthez tartozó feltételek adatait. A feltételek példánymodell szinten objektumokként jelennek meg az alábbi módon.



17. ábra: SonarQube feltételek példányai

4.2.3.1 SonarQube példánymodell

Az imént bemutatott példa SonarQube projekt bejárás adatainak rendszerszintű példánymodeljét az alábbi összefogó ábra mutatja.



18. ábra: Massif SonarQube példánymodell

A SonarQube metamodellt bemutató példánymodell gyökéreleme a „hu.bme.mit.massif:hu.bme.mit.massif.parent” nevű SonarQube projekt, melynek aktuális státusza WARN, azaz figyelmeztetés. A projekthez kettő analízis tartozik az ábrán, melyek lefutása között majdnem egy hónap telt el és mindkettő sikeresen zárult. A

projekt tartalmaz két Issue-t, melyek közül az egyik egy kritikus hiba, a másik pedig egy kisebb fontosságú „Code Smell”. Mind a kettő Issue nyitott állapotban van, vagyis jelenleg is igazak a megvizsgált kódhalmazra. A projekt ezen felül tartamaz kettő feltételt is. A „blocker_violations” feltétel értékének 0 alatt kell lennie, hogy ne legyen figyelmeztetés és 2 alatt, hogy ne legyen hiba, ezek teljesülnek is. A „critical_violations” nevű feltétel értéke 106, amely átlépi a 100-as figyelmeztetési küszöbértéket, ami miatt, ahogy már láthattuk a projekt WARN állapotú.

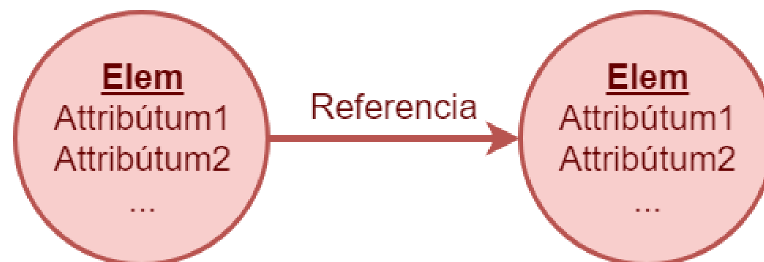
4.2.4 Kereszthivatkozások

A GitHub, Jenkins és SonarQube szerverek által szolgáltatott adatok között van kapcsolat. Ezeket a metmaodelleken átívelő referenciákat a 3.3.2.1. fejezetben bemutatott metamodell írja le. Amennyiben bármelyik adatbejáró modul ilyen kapcsolathoz ér, elküldi ezt az információt feldolgozásra a Trace Adapter modulnak, amely a külső kapcsolatok indexelését is elvégzi.

4.3 Adatok indexelése

Az adatbejáró modulok a bejárás során folyamatosan küldik a bejárt adatokat a hozzájuk tartozó indexer modulnak. Az indexer modulok feladata az IncQuery Server által nyújtott interfészen keresztül az adatok indexbe táplálása a 3.3.2. fejezetben bemutatott metamodellek szerint.

Az IQS indexe az adatok tárolására gráfreprezentációt használ. Adatok gráf alapú reprezentációjának a módja az alábbi ábrán látható.



19. ábra: Gráf adatok felépítése

A gráf csúcsai elemek, melyeknek lehet típusa és tetszőleges számú attribútuma. Az elemek közötti referenciák a gráf élei. Ebben a formában bármilyen adat és modell reprezentálható, EMF modellek leképzése pedig triviális feladat. Egy EMF példány elem egy gráfcsúcsnak felel meg a saját EMF osztályának megfelelő típussal, attribútumai a csúcs attribútumaival egyeznek meg, referenciái pedig más gráfcsúcsokhoz vezető élek.

Ennek megfelelően az IQS által nyújtott interfész elemek, attribútumok és referenciák írására ad lehetőséget. Egy branch és a hozzá tartozó legújabb commit-ra mutató referencia indexelése az alábbiak szerint történik.

```
val branchEClass = epckgContext.nameToEClass["Branch"]

writeElement(
        UpdateDirection.INSERT,
        EClassifierDescriptorProxy(branchEClass.name,GITHUB_REPO_NSURI),
        masterBranchId
)

writeAttribute(
        UpdateDirection.INSERT,
        indexerUtil.getEAttributeDescriptor(branchEClass,"name"),
        masterBranchId,
        "master"
)
```

```
writeReference(
        UpdateDirection.INSERT,
        indexerUtil.getEReferenceDescriptor(branchEClass,"head"),
        masterBranchId,
        "00a799370f6da8fc9c42487e74d47a13cb48aca2"
)
```

Először beillesztünk egy Branch típusú elemet az indexbe a master branch azonosítójának segítségével. Ezután az azonosítóval megadott elemhez hozzárendelünk egy név attribútumot „master" értékkel, majd pedig létrehozzuk a „head" nevű referencia élt a master branch és a commit között.

4.4 Adatok kiértékelése

Ha az adatok bejárása és indexelése megtörtént, vagyis van adat az IQS indexében, akkor lehetőségünk nyílik VQL lekérdezések futtatására az In-memory Query Engine modul segítségével. A VQL segítségével gráfmintákat, úgynevezett pattern-eket írhatunk, melyek megadják milyen illeszkedéseket keresünk az indexben lévő adathalmazon.

Először nézzünk két egyszerű mintát, melyeket egy bonyolultabb minta megírásához fogunk felhasználni. A GitHub, Jenkins és SonarQube metamodelleket az ALM metamodell köti össze. Az alábbi mintákkal ezeket a külső hivatkozásokat fogjuk újrahasználható mintákba csomagolni.

```
pattern analysisForBuild(
     build: Build,
     analysis: SonarQubeAnalysis
){
     AnalysisForBuild.buildKey(analysis4Build, build);
     AnalysisForBuild.analysisdKey(analysis4Build, analysis);
}

pattern buildTrigger(
     build: Build,
     branch: Branch,
     commit: Commit
){
     BuildTrigger.buildKey(buildTrigger, build);
     BuildTrigger.branchKey(buildTrigger, branch);
     BuildTrigger.commitKey(buildTrigger, commit);
}
```

Az analysisForBuild nevű minta két paramétere egy Build és egy SonarQubeAnalysis típusú objektum, melyeket a minta belsejében egy AnalysisForBuild elemmel köt össze. Az analysis4Build nevű elem build-je a build nevű parameter, analízise pedig az analysis nevű. A buildTrigger minta ezzel szemben Build-eket köt

össze az azokat kiváltó Commit-okkal és Branch-ekkel. A minta akkor illeszkedik, amennyiben a paraméterként kapott build, branch és commit ugyanazon BuildTrigger példányhoz tartoznak.

Az imént bemutatott segédmintákkal könnyedén köthetjük össze a különböző metamodellek adatait, így lehetőségünk van metamodelleken átívelő, összetett lekérdezések futtatására. Az alábbi minta olyan GitHub Issue-kat és azokhoz tartozó Commit-okat keres, melyek indítottak Jenkins Build-et, amelyhez SonarQube Analysis is tartozik.

```
pattern analysisAndBuildForIssueAndCommit(
	issueNumber: EInt, commitSHA: EString,
	commitMessage: EString, userLogin: EString
	buildNumber: EInt, buildResult: BuildResult,
	analysisStatus: Status,

){
	Build.id(build, buildNumber);
	Build.result(build, buildResult);

	find analysisForBuild(build, analysis);
	SonarQubeAnalysis.status(analysis, analysisStatus);

	find buildTrigger(build,_, commit);
	Commit.sha(commit, commitSHA);
	Commit.message(commit, commitMessage);
	Commit.author(commit, user);
	User.login(user, userLogin);

	Issue.issueRefs(issue, commit);
	Issue.number(issue, issueNumber);
}
```

A minta visszaadja az erre illeszkedő Issue-k sorszámát, az Issue-hoz tartozó és Build-et elindító Commit hash azonosítóját és szerzőjét, valamint a Build sorszámát, eredményét és a hozzá tartozó analízis státuszát. Látható, hogy a segédminták illeszkedése a find kulcsszóval ellenőrizhető. A buildTrigger branch paraméterét nem szükséges nevesíteni, mert ebben a mintában csak a commit-ra vagyunk kíváncsiak.

Az analysisAndBuildForIssueAndCommit nevű mintára illeszkedő adatokra az alábbi táblázatok mutatnak példát.

issueNumber	commitSHA	userLogin	buildNumber	buildResult	analysisStatus
147	41bdde9	ujhelyiz	**86**	SUCCESS	SUCCESS
148	41bdde9	ujhelyiz	**86**	SUCCESS	SUCCESS
117	00a7993	imbur	**71**	SUCCESS	SUCCESS
133	6b55197	imbur	**69**	SUCCESS	SUCCESS

commitSHA	commitMessage
41bdde9	Downgrades version to 0.7.0-SNAPSHOT #147 (#148)
00a7993	Merge pull request #138 from viatra/issue-117 Updating .simulink models #117
6b55197	Merge pull request #134 from viatra/issue-133 Quick fix for #133

Az eredmény egy példailleszkedése az következő: A 117-es GitHub Issue-hoz tartozik a 00a7993-as Commit, melyet imbur felhasználó commitolt. A Commit a 71-es Jenkins Build-et indította, melynek eredménye sikeres, ahogy a hozzá tartozó SonarQube analízis is.

5 Értékelés

Ebben a fejezetben az elkészült eszköz értékelését tűztem ki célul. Szót fogok ejteni az integrációs szoftver teljesítményéről, valamint a fejlesztés során felmerülő nehézségekről.

5.1 Felmerülő nehézségek

Az alábbi részben a fejlesztés során felmerülő kihívásokat és esetleges megoldásukat fogom részletezni. Egy reaktív, esemény vezérelt alkalmazás fejlesztése önmagában is számos kihívást hordozhat magában, azonban a körültekintő tervezésnek és a felhasznált technológiák, mint a Vert.x és az IncQuery Server megfelelő ismeretének köszönhetően az Adaper modulok, különösen az Indexer modul fejlesztése nem ütközött komolyabb nehézségekbe. Az adatbejáró modulokban az adatok bejárása során az integrálni kívánt külső eszközök által nyújtott interfészek tulajdonságai miatt viszont annál több munkát kellett befektetni a megfelelő megoldás megtalálásának érdekében.

5.1.1 Web API inkonzisztenciák

A fejlesztett szoftver feladata különböző külső forrásoktól származó adatok bejárása. Ez számos különböző interfész használatát jelentheti, azonban egy szolgáltatás APi-jában is lehetnek hiányosságok, vagy akár inkonzisztenciák. Az adatok bejárása során a legtöbb adat a GitHub szerveréről lett lekérdezve, így a GitHub web interfészek megfelelő használata volt a legfontosabb. Eltekintve attól, hogy a Jenkins szerver válaszaiban rendre üres JSON mezőkkel és számos redundáns értékkel operál, illetve a SonaQube esetében előfordul, hogy ugyanazt az elemet két lekérdezésben két különböző azonosító azonosít, ezen szolgáltatások használata nem okozott különösebb nehézségeket.

A GitHub esetén az adatok lekérdezésére a REST és a GraphQL API-t is használtam. A GraphQL előnye, hogy egy modern megoldás, mellyel megadható, hogy pontosan milyen adat szerepeljen a válaszban, így egy lekérdezés számos REST kérést helyettesíthet. Ez a tulajdonság nagyban hozzájárulhat a megfelelő teljesítmény eléréséhez. Felmerülhet, hogy érdemes-e az összes adat elérésére GraphQL mintákat alkalmazni. Tapasztalataim szerint teljesítményben a REST-el szembeni előnye nagy adathalmazok együttes lekérdezése során észlelhető. Ilyen lekérdezés például a commit-

ok és a hozzájuk tartozó fájlok lekérdezése. Olyan esetben, mint például egy repository kisszámú statikus adatainak lekérdezése a REST egyszerűbb és gyorsabb megoldás lehet. Meg kell említenem továbbá, hogy a GitHub nem minden adatot, vagy relációt vezetett még ki a GraphQL API-ra. Például a repository-hoz tartozó közreműködők listája, vagyis a „contributors" kapcsolat csak a REST interfészen érhető el, így annak használata elkerülhetetlen, amennyiben az összes adat elérése a cél.

5.1.2 GitHub API limitációk

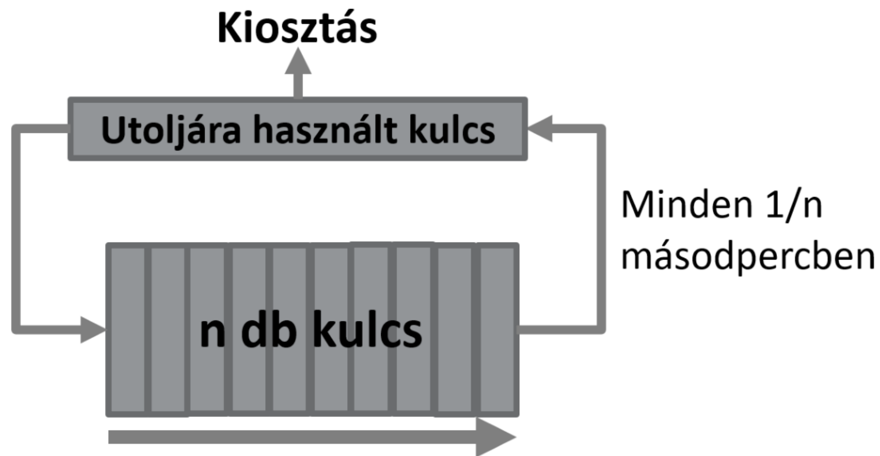
A Jenkins és a SonarQube szerverek általában saját üzemeltetésüek, így azok rendelkezésre állásának biztosítása is saját feladat. Ezzel szemben a GitHub-on található adatok elérését a GitHub szerverei biztosítják. Annak érdekében, hogy a kiszolgálás számos konkurens felhasználó esetén is folyamatos legyen a GitHub kliens szintű API korlátozásokat vezetett be.[9] Amennyiben egy kliens megszegi, vagy túllépi ezeket a szabályokat, a GitHub szerverei megtagadják a kiszolgálást. A korlátozásokról részletesen a GitHub „Rate Limit" oldalán olvashatunk, azok dióhélyban a következők.

- Egy kliensről egyidőben egy lekérdezés küldhető
- Egy kliensről másodpercenként egy lekérdezés küldhető
- Rest API limit: 5000 lekérdezés/óra
- GraphQl API limit: 5000 pont/óra

Míg a REST API esetében 5000 lekérdezést kezdeményezhetünk egy klienskulccsal óránként, addig a GraphQL esetén 5000 pont használható fel, ahol 1 pont 100 elem lekérdezését jelenti. Így óránként akár 500000 elemet is lekérdezhetünk a GraphQl segítségével. Mivel már egy közepes repository esetén is könnyedén áthághatjuk a REST API limitációját, a nagy elemszámú lekérdezésekhez, mint például a commit-ok és az issue referenciák lekérdezése érdemes GrapQL mintákat használni.

A további kettő szabály, melyek szerint egy kliensről egyidőben és másodpercenként is csak egy lekérdezés küldhető, komolyan visszavethetik az adatok elérésének hatékonyságát. Munkám során nagy hangsúlyt fektettem, hogy az Adapter modulok minél hatékonyabban, jól skálázható módon működjenek. A különböző adatbejáró és indexer modulok egymástól függetlenül, akár több szálon és példányban is

üzemelhetnek aszinkron módon. Annak érdekében, hogy az adatbejárás felgyorsítható legyen több klienskulcsot regisztráltam a GitHub felületén, ugyanis erre nincsen semmilyen korlátozás. Az integrációs szoftver tehát több autentikált klienskulcsot használ az adatbejárás során. A szabályok betartását egy időosztásos ütemező megoldás biztosítja, melynek logikája az alábbi ábrán látható.



20.ábra: Kulcsosztás folyamata

A kulcsosztás mechanikájának lényege, hogy az n darab beregisztrált klienskulcsot az ütemező tárolja egy visszacsatolt sorban. Az adatbejáró modulok minden lekérdezés előtt kulcsot kérvényeznek az ütemezőtől, amely 1/n másodpercenként kiszolgál egy ilyen kérdést, és az elhasznált kulcsot visszarakja a sor végére. Ezzel a megoldással másodpercenként n darab lekérdezés küldhető szankció nélkül.

5.2 Teljesítmény

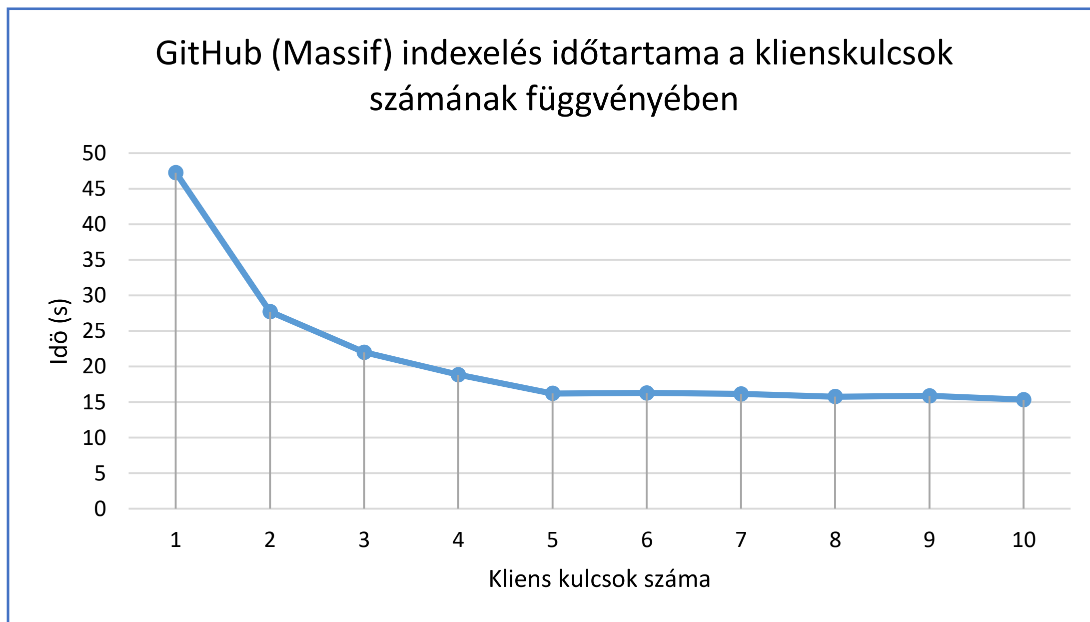
A rendszer tervezése és megvalósítása során hozott döntéseket egytől egyig a minél jobb teljesítmény elérése határozta meg. Az elért teljesítmény megvizsgálásának érdekében méréseket végeztem, melyeket az alábbiakban ismertetek.

Az előző fejezetben bemutatott GitHub klienskulcs használat marginális kérdés teljesítmény szempontjából, ugyanis egy projekt adatainak döntő többsége a hozzá tartozó GitHub repository-ból származik. Az általam végzett első mérés arra keresi a választ, hogy a klienskulcsok számának növelése milyen hatással van az indexelési

sebességre. A GitHub indexelés időtartamának változását a klienskulcsok számának függvényében a Viatra/Massif[4] publikus repository-n vizsgáltam.

GitHub (Massif) indexelés időtartama a klienskulcsok számának függvényében (s)									
1	2	3	4	5	6	7	8	9	10
47.223	27.473	21.768	18.775	15.144	16.57	17.079	16.113	15.982	15.26
47.307	27.072	21.951	19.293	16.561	15.905	15.879	15.439	16.035	14.953
47.267	28.621	22.312	18.461	16.978	16.401	15.527	15.727	15.628	15.802

A táblázat egytől-tízig terjedő klienskulcsszámhoz tartozó indexelési időket tartalmaz másodpercben mérve. Az adatokból az látszik, hogy a kulcsszám növelése kezdetben nagymértékben csökkenti a szükséges időt, majd a csökkenés lelassul. Ugyan tíz kulcs használatával érhetünk el további sebesség növekedést öt kulcshoz képest, a sebesség növekedése öt kulcstól összemérhető a GitHub szerverek válaszidejének ingadozásával. Az eredmények átlagait az alábbi diagramon ábrázoltam.



21. ábra: GitHub (Massif) indexelés időtartama a klienskulcsok számának függvényében

A diagramon is jól látszik, hogy az indexelési idő csökkenése öt kulcsig jelentős, utána az érték a 15 másodperces határt közelíti. Több kulcsot nagyságrendekkel nagyobb projektek bejárásakor lehet érdemes használni, amennyiben a hálózati körülmények is megfelelőek.

[4] https://github.com/viatra/massif

A következő mérések az egyes adapterek teljesítményét vizsgálják. A GitHub, Jenkins és SonarQube modulok indexelési idejét külön viszgáltam. Az adatbejárás folyamatának felépítése alapján a GitHub indexelés komplexitását az összes commit száma, a Jenkins adatainak bejárásáét a buildszám, a Sonar-ét pedig az issue-k és analízisek számának összege határozza meg. Az így meghatározott mérőszámokat feltüntettem a táblázatokban a mért modul neve alatt zárójelben. A mérések során 5 GitHub klienskulcsot használtam és mindegyik projekt esetén modulonként három mérést végeztem. A mérések során összesen négy projekt indexelési sebességét vizsgáltam. A legkisebb projekt a pár elágazással és commit-al rendelkező „api-example”[5] projekt, melyet kimondottan tesztelés céljából készítettem. Párezer commit-jával méretben a második projekt a „massif”, melyet az esettanulmány kapcsán már bemutattam. A „massif”-nál egy nagyságrenddel összetettebb projekt a „viatra”[6] nevű szintén publikus projekt, melynek összcommitszáma tízezres nagyságrendű. A méréseket végül pedig egy még a „viatra” projektnél is nagyobb elemszámú fejlesztésen, az „incquery-server”-en is lefuttattam. A mérések eredményei az alábbi táblázatokban láthatók.

api-example projekt indexelési időtartama (5 kulcs) (s)		
GitHub (37)	Jenkins (18)	SonarQube (7)
6.62	3.53	4.697
5.686	4.15	4.344
5.798	3.129	4.018

Massif projekt indexelési időtartama (5 kulcs) (s)		
GitHub (2112)	Jenkins (51)	SonarQube (603)
24.816	2.803	5.537
18.929	2.819	5.779
19.499	2.623	6.671

[5] https://github.com/beothy/api-example

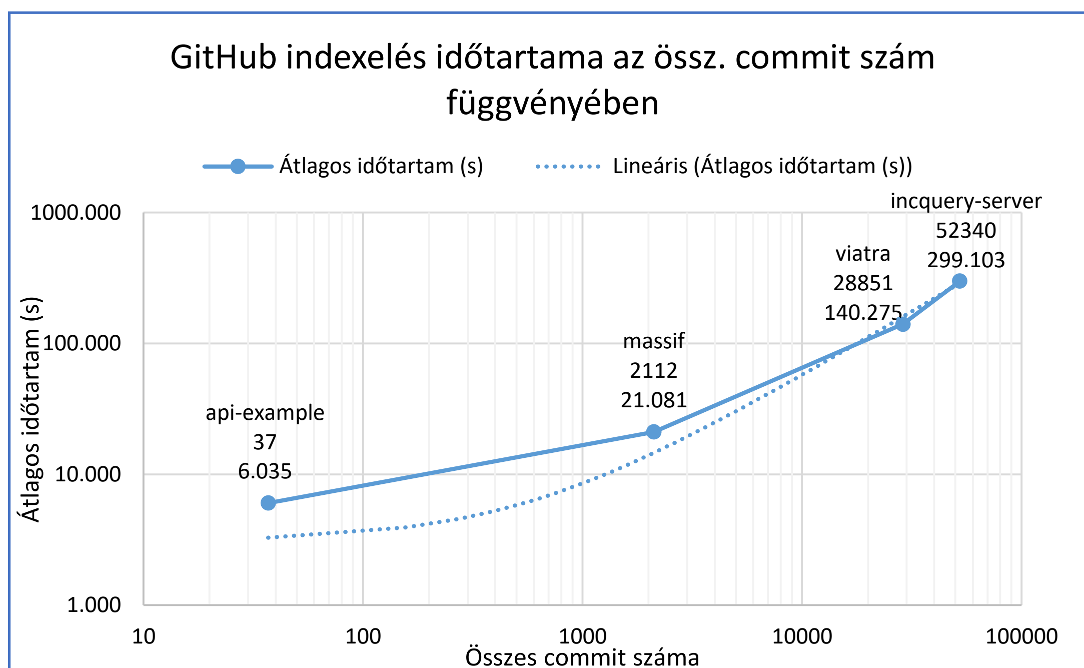
[6] https://github.com/viatra/org.eclipse.viatra

Viatra projekt indexelési időtartama (5 kulcs) (s)		
GitHub (28851)	Jenkins (5)	SonarQube (2028)
134.533	2.622	10.243
143.207	2.817	10.257
143.086	3.093	9.997

incquery-server projekt indexelési időtartama (5 kulcs) (s)		
GitHub (52340)	Jenkins (130)	SonarQube (2340)
281.042	34.501	10.718
305.582	34.426	10.928
310.685	33.552	10.661

Ami elsőre jól látható, hogy a GitHub indexelési folyamata tart minden esetben a legtovább, illetve annál tapasztalhatók a legnagyobb eltérések is, mely abból adódik, hogy a GitHub szervereinek terheltsége kiszámíthatatlan és változó. A Jenkins és Sonar szerverek ezzel szemben saját üzemeltetésűek, így hálózati terheltségük általában kicsi és közel azonos mértékű.

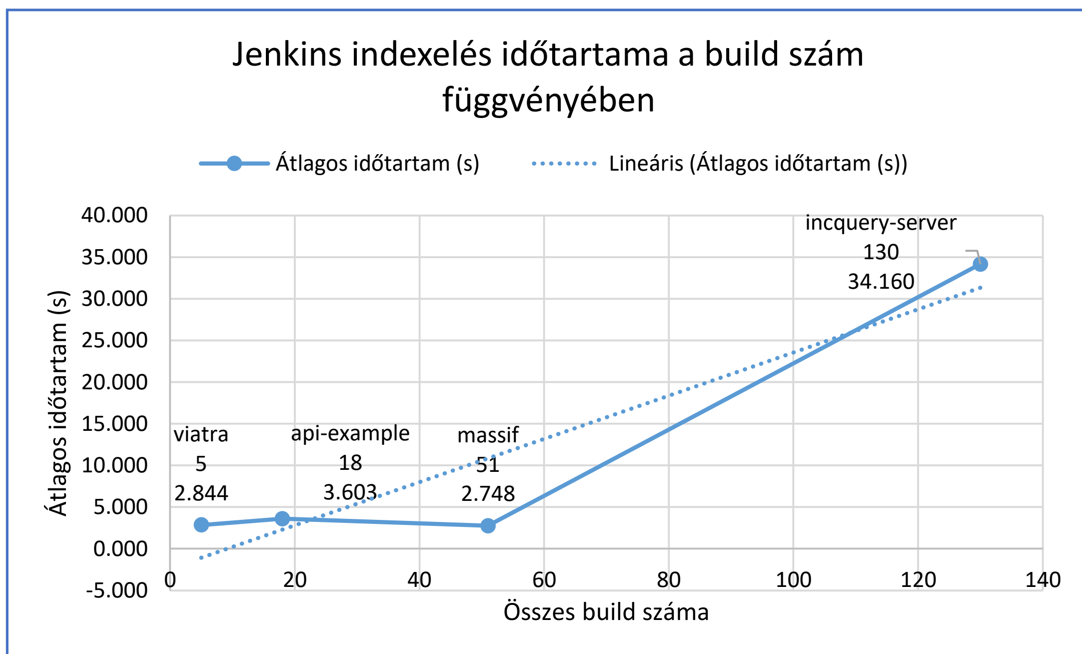
A mérések célja jelen esetben az indexelési sebességek változásának megfigyelése az indexelt projekt komplexitásának függvényében. A mért adatok átlagait egy diagramon ábrázoltam. Az Y tengely minden esetben az átlagos időtartam másodpercekben nézve, az X tengely pedig a projekt komplexitását mutatja.



22. ábra: GitHub indexelés időtartama az össz. Commit szám függvényében

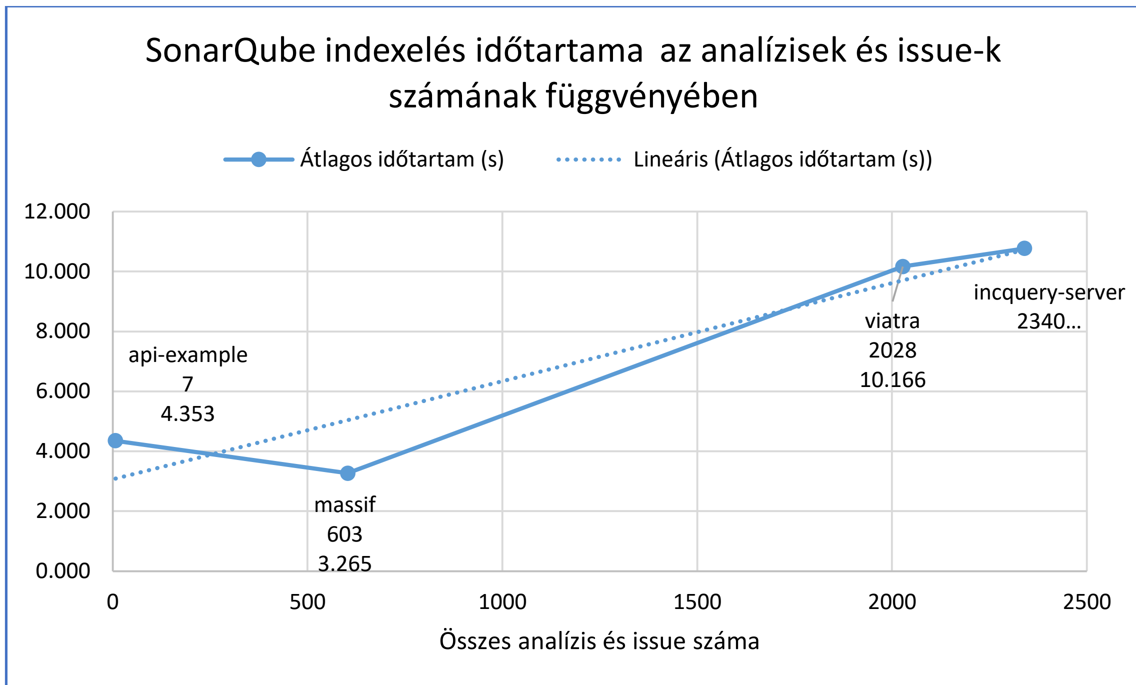
A GitHub esetén az össz. commit szám jelenik meg az X tengelyen. A tengelyek felosztását logaritmikusra választottam amiatt, hogy az „api-example” 37-es és az „incquery-server” 52340-es értéke is jól ábrázolható legyen. A diagramról az olvasható le, hogy a GitHub modul indexelési ideje kisebb commit szám esetén elmarad a lineáristól, mely különbséget okozhatja az, hogy ezeknél a projekteknél a commit-ok számának aránya a többi adathoz képest nem annyira magas, így ezekben az esetekben a commit szám alapú komplexitás becslés nem olyan pontos. Tízezres nagyságrendnél azonban az indexelés idejének változása már közel lineáris változást mutat a mért tartományban.

A Jenkins és SonarQube modulokhoz tartozó diagrammok az alábbiak.



23. ábra: Jenkins modul indexelésének időtartama a build szám függvényében

A Jenkins-hez tartozó diagramon az látható, hogy a build-ek számának növekedésével kezdetben az indexelés időtartama nem növekszik, amely annak tudható be, hogy a Jenkins szervernek 5 helyett 50 kérés kiszolgálása nem okoz számottevő terhelést. Nagyobb elemszám esetén azonban sejtésem szerint az időtartam növekedése közel lineáris.



24. ábra: SonarQube modul indexelésének időtartama az analízisek és issue-k számának függvényében

A SonarQube átlagos indexelési idejét ábrázoló diagram azt mutatja, hogy ebben az esetben is kezdetben az időtartam nem növekszik jelentősen, sőt az aktuális mérésben valószínűleg a hálózati forgalom kiszámíthatatlansága miatt például csökken. Magasabb elemszám esetén azonban a trend a lineárisat közelíti.

A modulok együttes futtatása során a három adatforrás bejárása egymással párhuzamosan történik, így az egész adathalmaz bejárásának idejét a GitHub, vagyis a legnagyobb adatforrás bejárásának ideje teszi ki. A mért értékek alapján az eszköz 150-200 elem/s indexelési sebességet képes elérni.

A különböző forrásból származó adatokat azért töltöttük be egy indexbe, hogy azok összességén összetett gráfminta-alapú lekérdezéseket futtathassunk. Teljesítményértékelés szempontjából lényeges kérdés, hogy a lekérdezések mennyi idő alatt futnak le. Ennek lemérése három VQL mintát írtam, melyek az alábbiak.

```
pattern branches(repo: Repository, branch: Branch){
     Repository.branches(repo,branch);
}
```

```
pattern commits(
	repo: Repository, branch: Branch,
	commit: Commit, user:User
){
	Repository.branches(repo, branch);
	Branch.commits(branch,commit);
	Commit.comitter(commit, user);
} or {
	Repository.branches(repo, branch);
	Branch.commits(branch,commit);
	Commit.author(commit, user);
}

pattern analysisAndBuildForIssueAndCommit(
	buildNumber: EInt, buildResult: BuildResult,
	analysisStatus: Status, issueNumber: EInt,
	commitMessage: EString, userLogin: EString
){
	Build.id(build, buildNumber);
	Build.result(build, buildResult);
	find analysisForBuild(build, analysis);
	SonarQubeAnalysis.status(analysis, analysisStatus);

	find buildTrigger(build, _, commit);
	Commit.message(commit, commitMessage);
	Commit.author(commit, user);
	User.login(user, userLogin);

	Issue.issueRefs(issue, commit);
	Issue.number(issue, issueNumber);
}
```

A „branches" nevű minta a legegyszerűbb, egy darab sorból áll, lekérdezi az összes Repository-Branch párost. A „commits" nevű minta bonyolultabb az előzőnél, de ugyanúgy egy metamodell, a GitHub mtamodell elemeit éri el. Összetartozó Branch, Commit, User hármasokra illeszkedik. A legösszetettebb minta az „analysisAndBuildForIssueAndCommit" minta, amely GitHub Issue-kat köt össze azok Commit-jai által indított Jenkins Build-ekkel és az azokhoz tartozó SonarQube analízisekkel. A 4.4. fejezetben ezt a mintát részletesen is bemutatom. Ez a minta mind a három (GitHub, Jenkins, SonarQube) metamodellt használja, a köztes ALM metamodell összekötve azokat.

A bemutatott három mintát, három eltérő méretű projekten futtattam le. Az „api-example" projekt egy általam kimondottan erre a célra készített egy fájlból és pár commit-ból álló projekt, a „massif" egy közepes méretű, ezres nagyságrendű összcommitszámú

projekt, az „opencv”[7] pedig egy nagy méretű publikus projekt, melynek összes commit-jának száma tízezres nagyságrend. Méréseim eredményei az alábbi táblázatban láthatók.

VQL minta futtatások ideje (s)			
	branches	commits	analysisAndBuild…
api-example	0.152	0.16	0.583
	0.146	0.144	0.611
	0.156	0.232	0.468
massif	0.165	0.106	0.537
	0.21	0.153	0.642
	0.148	0.228	0.518
opencv	0.178	0.404	0.531
	0.165	0.509	0.517
	0.245	0.301	0.613

Az eredményekből az látszik, hogy a legegyszerűbb „branches” minta illesztése megközelítőleg mindhárom projekt esetén hasonlóan gyors volt. A „commits” nevű minta futási idje az „api-example” projekt esetén 57, a „massif” esetén pedig 32666 illeszkedésre közel azonos. Az „opencv” által adott 61327 illeszkedésre azonban már valamivel lassabban futott le a lekérdezés. Az „analysisAndBuildForIssueAndCommit” nevű mint a legbonyolultabb, ez mindhárom metamodellt eléri, így ennek kiértékelése tartott mindhárom projekt esetén a legtovább. A lekérdezések sebessége tehát kis mértékben függ az illesztett minta bonyolultságától és az adatok mennyiségétől.

[7] https://github.com/opencv/opencv

6 Összefoglalás

A dolgozat zárásaként ebben a fejezetben összefoglalom a kutatás eredményeit, valamint felvázolom a jövőbeni továbbfejlesztési lehetőségeket.

6.1 Elvégzett munka

Kutatásom során egy olyan könnyen bővíthető adatintegrációs szoftver fejlesztését tűztem ki célul, amely különböző fejlesztést támogató eszközök adatainak integrálását, tárolását valamint kiértékelését teszi lehetővé, ezzel elősegítve a fejlesztett alkalmazás életciklusának menedzselését.

Először megterveztem a rendszer általános, majd ALM specifikus architektúráját és definiáltam az adatok belső reprezentációját megadó EMF metamodelleket. Az adatintegrációs szoftverhez GitHub, Jenkins és SonarQube adaptereket terveztem és valósítottam meg, melyek az adott eszközök által tárolt adatok bejárását és indexelését végzik. A rendszer működését és az adatintegráció jellemzőit egy valós szoftverprojekteket használó esettanulmányon keresztül vizsgáltam meg. Végezetül pedig mérésekkel vizsgáltam a rendszert teljesítmény szempontjából.

Az elkészült szoftver összesen 7 modulból áll. A projekt 14 Kotlin forrásfájlt tartalmaz, melyekben 7 modul osztály, 3 kiegészítő osztály és 18 adatreprezentációs osztály található. A kódbázist összesen mintegy 2300 sornyi Kotlin forráskód alkotja.

A munkám eredménye egy olyan integrált ALM eszköz, mely egy fejlesztés GitHub repository-ja, Jenkins és SonarQube szervere által kezelt adatok összekötését és kezelését végzi, mellyel lehetőséget nyújt nagy teljesítményű, összetett VQL gráfminta-alapú lekérdezések futtatására az adatok összességén. Méréseim alapján az eszköz által végzett integrációs folyamat futási ideje a projektek build idejével mérhető össze, így alkalmas lehet commit idejű futtatásra.

6.2 Továbbfejlesztési lehetőségek

A dolgozat során bemutatott eszköz a jövőben több irányban is tovább fejleszthető. Az egyik lehetséges irány újabb adapterek fejlesztése és ezzel egyre több eszköz integrációjának támogatása, azzal a céllal, hogy egy teljes ALM platform épüljön ki. A másik fejlesztési irány pedig a rendszer felkészítése az inkrementális működésre.

Jelen esetben minden futtatásnál megtörténik a teljes integrációs folyamat. Az inkrementális megközelítéssel azonban csak a projekt egy kezdeti fázisában volna szükség teljes adatbejárásra. A projekt további változásainak szinkronizációjához elegendő lenne csupán az aktuális változások feldolgozása. Ez a megoldás további teljesítménynövekedéshez, vagyis a futási idő csökkenéséhez vezetne.

6.3 Köszönetnyilvánítás

Ezúton szeretnék köszönetet mondani azoknak, akik kitartó támogatásukkal és szakmai segítségükkel hozzájárultak ennek a dolgozatnak az elkészüléséhez. Szeretném megköszönni konzulenseimnek, Dr. Ráth István Zoltánnak és Dr. Hegedüs Ábelnek, hogy szakértelmükkel és tanácsaikkal segítették munkámat mind a tervezés és a megvalósítás, mind a dolgozat megírása során. Köszönettel tartozok az IncQuery Labs Kft. munkatársainak, akik szakmai kérdésekben mindig rendelkezésemre álltak. Külön köszönet illeti kollégámat, Papp Istvánt, akinek az IncQuery Server technológiával kapcsolatos segítsége nagyban elősegítette munkám kezdeti nehézségeinek áthidalását.

Irodalomjegyzék

[1] Ábel Hegedüs, Gábor Bergmann, Csaba Debreceni, Ákos Horváth, Ákos Menyhért, Péter Lunk, . . . István Ráth . (2018). Incquery server for teamwork cloud: scalable query evaluation over collaborative model repositories. Forrás: https://www.researchgate.net/publication/328114215_Incquery_server_for_teamwork_cloud_scalable_query_evaluation_over_collaborative_model_repositories

[2] Andreas Deuter, Andreas Otte, Marcel Ebert, & Frank Possel-Dölken. (2018). Developing the Requirements of a PLM/ALM Integration: An Industrial Case Study. Forrás: https://www.sciencedirect.com/science/article/pii/S2351978918305377

[3] Christof Ebert. (2013). Improving engineering efficiency with PLM/ALM. Forrás: https://link.springer.com/article/10.1007/s10270-013-0347-3

[4] Csikós Donát . (2012). Incremental dependency analysis of a large software infrastructure. Forrás: http://tdk.bme.hu/VIK/SW/Nagy-szoftverinfrastruktura-feletti

[5] Eclipse Foundation. (2018). *Eclipse Modeling Framework (EMF)*. Forrás: https://www.eclipse.org/modeling/emf/

[6] Eclipse Foundation. (2018). *Eclipse Vert.x*. Forrás: https://vertx.io/

[7] Eclipse Foundation. (2018). *The Viatra Query Language (VQL)*. Forrás: https://www.eclipse.org/viatra/documentation/query-language.html

[8] Eclipse Foundation. (2018). *Viatra - Scalable reactive model transformations*. Forrás: https://www.eclipse.org/viatra/

[9] GitHub Inc. (2018). *GitHub Developer Guide - API Rate Limit*. Forrás: https://developer.github.com/v3/#rate-limiting

[10] GitHub Inc. (2018). *GitHub Developer Guide - GraphQL API v4*. Forrás: https://developer.github.com/v4/guides/intro-to-graphql/

[11] GitHub Inc. (2018). *GitHub Developer Guide - REST API v3*. Forrás: https://developer.github.com/v3/

[12] GitHub Inc. (2018). *The world's leading software development platform, GitHub*. Forrás: https://github.com/

[13] IncQuery Labs Ltd. (2018). *IncQuery - Powerful queries at the fingertips of your engineers*. Forrás: https://incquerylabs.com/incquery/

[14] Intercax. (2018). *Syndeia*. Forrás: http://intercax.com/products/syndeia/

[15] *Jenkins*. (2018). Forrás: https://jenkins.io/

[16] JetBrains. (2018). *Kotlin Programming Language*. Forrás: https://kotlinlang.org/

[17] Manas Bajaj PhD. (2015). System Lifecycle Management Syndeia for MBSE. Forrás: http://blog.nomagic.com/wp-content/uploads/2015/06/Syndeia-for-MBSE-No-Magic-Symposium-2015-06-09.pdf

[18] Manas Bajaj PhD, Dirk Zwemer PhD, Rose Yntema, Alex Phung, Amit Kumar, Anshu Dwivedi, & Manoj Waikar. (2016). MBSE++ — Foundations for Extended Model-Based Systems Engineering Across System Lifecycle. Forrás: https://onlinelibrary.wiley.com/doi/pdf/10.1002/j.2334-5837.2016.00304.x

[19] Roberto Rodriguez-Echeverria, Javier Luis Cánovas Izquierdo, & Jordi Cabot. (2018). Towards a UML and IFML Mapping to GraphQL. Forrás: https://link.springer.com/chapter/10.1007/978-3-319-74433-9_13

[20] *SonarQube*. (dátum nélk.). Forrás: https://www.sonarqube.org/

[21] Stein Dániel . (2014). Incremental Static Analysis of Large Source Code Repositories. Forrás: https://tdk.bme.hu/VIK/Informacios/Nagymeretu-forraskod-tarak-inkrementalis2

[22] Zoltán Ujhelyi, Ákos Horváth, Dániel Varró, Norbert István Csiszár, Gábor Szőke, László Vidács, & Rudolf Ferenc. (2014). Anti-pattern detection with model queries: A comparison of approaches. Forrás: https://ieeexplore.ieee.org/abstract/document/6747181