Szegedi Tudományegyetem
Informatikai Intézet

Ipari JavaScript elemző kiegészítése TypeScript támogatással

Szakdolgozat

Készítette:

Pozsgai Alex

programtervező informatikus BSc szakos hallgató

Témavezető:

Dr. Antal Gábor

Tudományos munkatárs

Szeged
2023

Feladatkiírás

A jelenlegi projektet Sourcemeter JavaScriptnek hívják. A cél az, hogy ez a projekt többet tudjon, mint a piacon a többi kódelemező. Ebből adódóan bővíteni kell TypeScript támogatással.

A hallgató feladata ennek a programnak a fejlesztése úgy, hogy tudjon TypeScript forráskódot, fájlokat és projekteket is kellő pontossággal elemezni. Ezután a meglévő programot optimalizálni úgy, hogy kevesebb erőforrást igényeljen, és hogy gyorsabban fusson le.

Tartalmi összefoglaló

A téma megnevezése:

SourceMeter JavaScript kiegészítése TypeScript támogatással, az eszközcsalád egyes eszközeinek optimalizálása.

A megadott feladat megfogalmazása:

A feladat során el kell érni azt, hogy a SourceMeter for JavaScript elemző eszközkészlet képes legyen jól lefutni és elemezni typescript fájlokra. Továbbá, a JavaScript elemző optimalizálása.

A megoldási mód:

A megoldás során változtatni kell a nyelvi sémán, amire az eszközök épülnek. A létrejövő változtatásokat minden eszközön elvégezni, tesztekkel bővíteni.

Alkalmazott eszközök, módszerek:

A megoldáshoz Visual Studio Code-t, a nyelvi séma szerkesztéséhez Visual Paradigmt használtam. A projekt lebuildeléséhez és ellenőrzéséhez Windowson Visual Studio 2017, Linuxon make programot használtam.

Elért eredmények:

A SourceMeter JavaScript képes TypeScript forráskódokat, fájlokat nagy pontossággal elemezni, nagyobb projektre lényegesen gyorsabban fut le, kevesebb erőforrást igényel. A JSAN2Lim sikeresen alakítja át a JSAN eredményét nyelvfüggetlen modellre, TypeScript fájlok elemzése során is, amin dolgozik több eszközcsalád.

Kulcsszavak:

JavaScript, TypeScript, Optimalizálás, Visual Paradigm, C++, Statikus kódelemzés, AST

Tartalomjegyzék

1. fejezet

Bevezetés

Szakdolgozatomban a SourceMeter for JavaScript továbbfejlesztése volt a cél. A JavaScript elemzése mellett TypeScript nyelvű fájlokat és projekteket is elemezni kellett. Ez azt eredményezte, hogy gyökerestül át kellett írni, annak érdekében, hogy JavaScriptet és TypeScriptet is tudjon egyaránt elemezni.

A SourceMeter for JavaScript projekten többen is fejlesztettek egyszerre, ezáltal voltak átfedések, oszthatatlan részek. A projekt több része is egymásra épül, ezért kiemelten fontos volt a csapatmunka egyes részeknél, hogy a projekt eredményesen és hatékonyan haladjon. Az eredményes csapatmunka magában foglalja az eredmények és az előrehaladás folyamatos kommunikálását, ami azt jelenti, hogy mindketten megértjük, hogy milyen célkitűzések vannak a projektben, és rendszeresen jelentést adunk egymásnak az elvégzett munkáról és az elért eredményekről.

Megtalálható számtalan JavaScript és TypeScript fájl elemző eszköz amiknek nyílt a forráskódja. Ezek között megtalálható a Codehawk CLI, Codelyzer vagy a CodeClimmate-Duplication. Mind a három eszköz jól tud elemezni JavaScript és TypeScript fájlokat, viszont csak egy-egy specifikus esetre jók.

Fejlesztés során a csoportos munka elkerülhetetlen volt, hiszen volt olyan rész, amire számtalan eszköz épült. Ez a JavaScript nyelvi séma (továbbiakban:séma) szerkesztése volt. Ezt a 2.2.1 alfejezetben taglalom bővebben.

A séma szerkesztése után a JavaScript Analyzer To Lim (továbbiakban JSAN2Lim) továbbfejlesztése volt a feladatom. A továbbfejlesztés C++, C nyelv és TypeScript séma [6] ismeretét igényelte meg. Emiatt a fejlesztés előtt tanulmányoznom kellett ezeket.

A JSAN2Lim fejlesztése után a JavaScript elemzőnek (JavaScript Analyzer, továbbiakban: JSAN) a binder függvényét kellett optimalizálnom. Először értelmeznem kellett a kódot, át kellett néznem az AST-t, illetve több adatszerkezetet is, gyorsasági szempontból.

Legvégül bővítettem a regressziós teszteket új projektek behozatalával. Ezt az 5 fejezetben taglalom.

2. fejezet

SourceMeter

2.1. SourceMeter bevezetés

A SourceMeter egy forráskód-elemző eszköz, amely képes mély statikus programelemzést végezni a C, C++, Java, Python, C#, JavaScript, TypeScript és RPG (AS/400) [5] nyelvű összetett programok forráskódján. A FrontEndART a Szegedi Tudományegyetem Szoftverfejlesztés Tanszékén kutatott és fejlesztett Columbus technológián [1] alapuló SourceMeter eszközt fejlesztette ki. A statikus kódelemzés egy olyan módszer, amely során a program forráskódját elemezzük, anélkül hogy azt ténylegesen futtatnánk. Az elemzés során különböző eszközök segítségével ellenőrizhetjük a kód helyességét, hatékonyságát, biztonságosságát és karbantarthatóságát. Az ilyen típusú elemzés során gyakran felhasználnak különböző szabályokat és előírásokat, amelyek segítenek az azonosításban és a hibák javításában.

A statikus elemzés során absztrakt szemantikus gráf (ASG) készül a forráskód nyelvi elemeiből. Ezután az ASG-t különböző eszközökkel dolgozzák fel a csomagban annak érdekében, hogy kiszámítsák a metrikákat (LLOC [4], NLE vagy NOA), azonosítsák az ismételt kódrészleteket (másolás-beszúrás; klónok), a kódolási szabályszegéseket, stb. A SourceMeter képes elemzést végezni olyan forráskódon,amely megfelel a Java 8 és korábbi verzióinak, a C/C++, az RPG III és az RPG IV verzióinak (beleértve a szabadon formázottakat), a C# 6.0 és korábbi verzióinak, valamint a Python 2.7.8 és korábbi verzióinak. A C/C++ esetében a SourceMeter támogatja az ISO/IEC 14882:2011 [3] nemzetközi szabványt, amelyet kiegészítettek az ISO/IEC 14882:2014 új funkcióival, és a C

nyelvet az ANSI/ISO 9899:1990, az ISO/IEC 9899:1999 és az ISO/IEC 9899:2011 szabványok határozzák meg. Az alapértelmezett funkciókon túl, a GCC és a Microsoft által meghatározott kiterjesztések is támogatottak.

A SourceMeter a QualityGate eszközben van használva.

TODO: Képeket betenni

A SourceMeternek található egy plug-in a SonarQubehoz. A SourceMeter plug-in a SonarQube platformhoz egy kiterjesztése az nyílt forráskódú SonarQube platformnak, amelyet a kód minőségének kezelésére használnak A plug-in a SourceMeter-t futtatja a SonarQube platformról, és feltölti a forráskód elemzésének eredményeit a SourceMetertől a SonarQube adatbázisába. A plug-in nyílt forráskódú, és az összes szokásos SonarQube kódelemzési eredményt biztosítja, kiegészítve sok további metrikával és problémakeresővel, amelyeket a SourceMeter eszköz biztosít. A plug-in támogatja a C/C++, a Java, a C#, a Python és a RPG nyelveket. [2]

2.2. SourceMeter for JavaScript

A SourceMeter for JavaScript a SourceMeternek egy nagyobb alprojektje. A SourceMeter for JavaScript egy olyan eszköz, amely lehetővé teszi a mély statikus forráskód elemzést a bonyolult JavaScript és TypeScript rendszerekben. Képes felismerni a kód hibáit, mint például a nem definiált változók vagy függvények használata, a nem biztonságos kódrészletek, a nem hatékony kódrészletek, valamint a redundáns és ismétlődő kódok. Ezenkívül az eszköz képes összehasonlítani a kódot az általános gyakorlatokkal és a meghatározott szabályokkal, és jelezni az eltéréseket. Az ilyen eszközök használata segíthet az észlelt hibák javításában és a kód minőségének javításában, ami végső soron javíthatja a rendszer biztonságát és hatékonyságát.

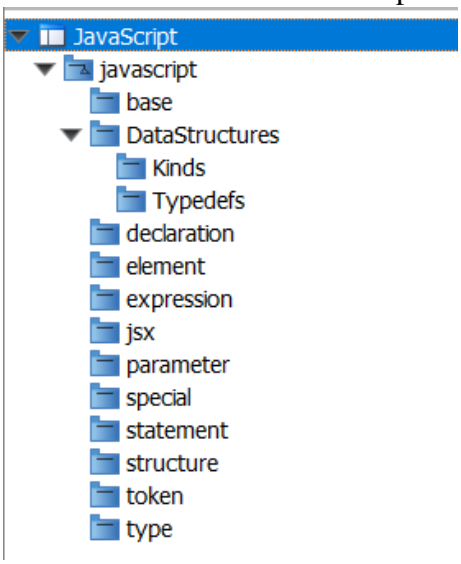
A SourceMeter for JavaScript projekt több alprojektet is magába foglal, amelyek különböző részfeladatokra specializálódnak.

2.2.1. Nyelvi séma bevezetés

A JavaScript nyelvi séma(továbbiakban: séma) egy UML Diagramhoz hasonlító séma. A Visual Paradigm(továbbiakban: vpp) alkalmazással szerkeszthető. Azért szerkesztjük a

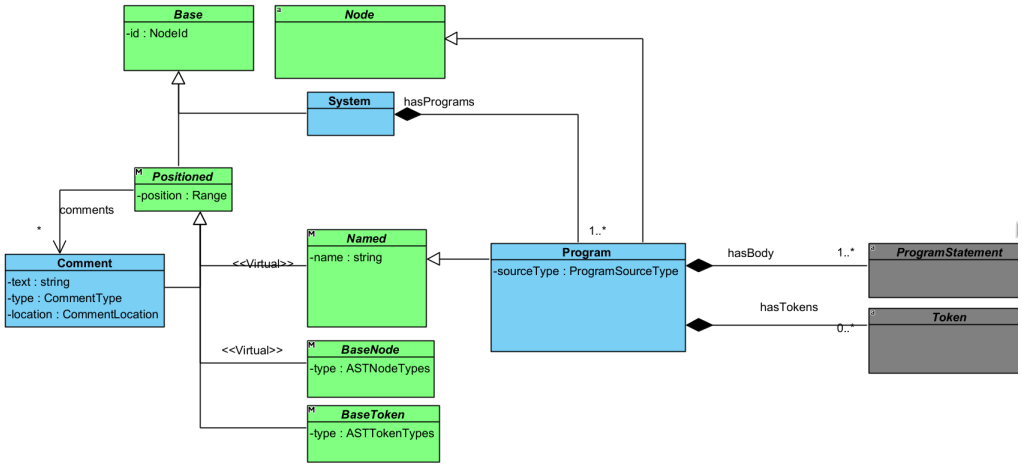
sémát vpp-ben, mivel a többi nyelvi változat esetében is így volt, és van koncerter, ami ebből a vpp fájlból létrehozza a C++ alap fájlokat, amiket az elemző programok tudnak használni. A felépítése a következőképpen néz ki:

2.1. ábra. Séma struktúrális felépítése



A 2.1 ábrán a ProjektNlv/Model/Package-ek struktúra figyelhető meg. A projekt neve JavaScript, ezen belül található egy model, amit javascript-nek hívnak. A modellen belül találhatóak meg a package-ek. A Package-ek a TypeScript séma [6] alapján lettek elnevezve. Átláthatóság és könnyebb bővíthetőség szempontjából lett létrehozva több package. Követtük a TypeScript-eslint [6] projekt struktúrális felépítését. A package-ek a következőképpen épülnek fel:

2.2. ábra. A base package felépítése



A 2.2 ábrán a Base, Node, Positioned, Comment, System, Named, BaseNode, BaseToken, Program, ProgramStatement és a Token osztályok találhatóak. A Base osztályból öröklődik minden osztály. A Base osztály rendelkezik id attribútummal, ami NodeId típusú. Az absztrakciót zöld háttérszínnel jelöltük. A séma könnyebb bővíthetősége és olvashatósága miatt jeleztük így, illetve más nyelvi sémákban is ezt a logikát követjük. Más package-ekben lévő definiálást szürke háttérszínnel, ugyanabban a package-ben lévő definiálást világosszürke háttérszínnel jelöltük. Az alapértelmezett osztályt kék háttérszínnel jelöltük. A Positioned osztályból öröklődnek a Named, BaseNode és a BaseToken osztályok. A BaseNode osztályból több minden öröklődik, mint például a DeclarationStatement, ImportDeclaration, Expression és több minden is. Ez látható a 2.3 ábrán. A Positioned osztály rendelkezik position attribútummal, ami Range típusú. A Positioned osztályhoz tartozhatnak kommentek is. A Comment osztály öröklődik a Positioned osztályból, ezzel biztosítva azt, hogy minden kommentnek lesz pozíciója. A Comment osztály rendelkezik text, type és location attribútummal. A CommentType és a CommentLocation a DataStructures package-ben lett definiálva. Végül, Program osztály öröklődik a Named osztályból, ezáltal rendelkezik name attribútummal, ami string típusú. Minden Program osztályhoz tartozik legalább 1 System, ezt az 1..*-al jelöltük. A hasProgramsnak a JavaScriptAddonban lesz jelentősége, amit a 2.2.2 alfejezetben taglalok bővebben. A ProgramSourceType is a DataStructures package-ben lett definiálva. A ProgramSourceType értéke lehet source, vagy module. A Base package eltér a TypeScript-eslint [6] projektben lévő Base package-től. Régebbi sémának a Base package-ét használjuk, BaseNode és BaseToken osztályok kibővítésével.

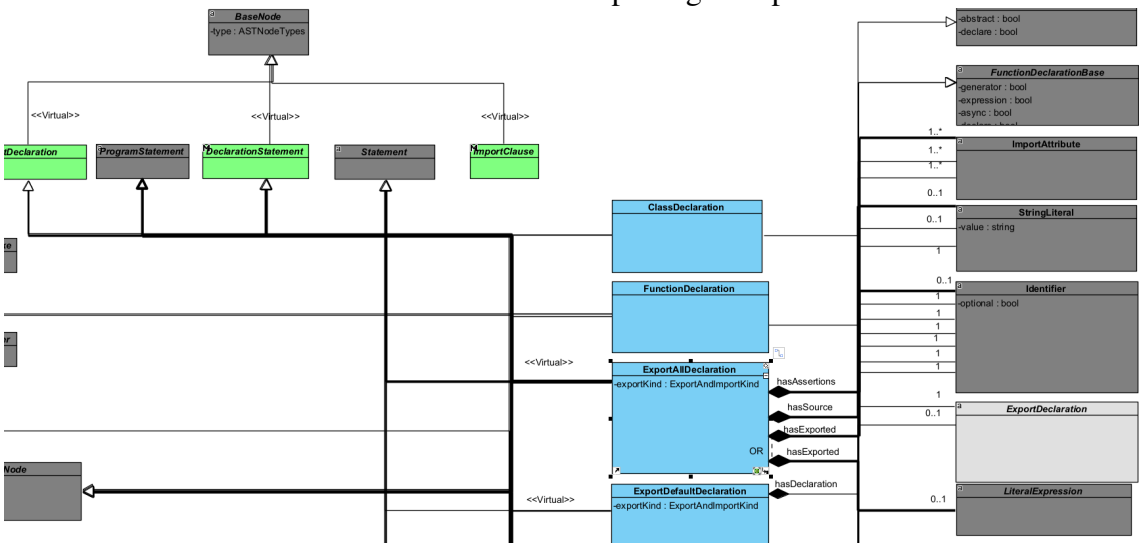
Mivel a Base package egyedi a mi esetünkben, ezért kitérnék a Declaration package-re is. A Declaration package-n belül az ExportAllDeclaration osztályt mutatom be.

```
1 export interface ExportAllDeclaration extends BaseNode {
2       type: AST_NODE_TYPES.ExportAllDeclaration;
3       assertions: ImportAttribute[];
4       exported: Identifier | null;
5       exportKind: ExportKind;
6       source: StringLiteral;
7 }
```

Kódrészlet 2.1. ExportAllDeclaration TypeScript megvalósítása

A 2.1 kódrészlet megvalósítása a sémában:

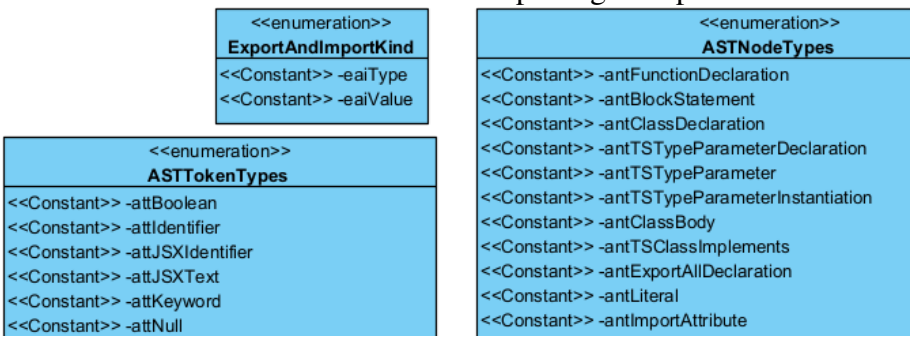
2.3. ábra. A Declaration package felépítése



Az ExportAllDeclaration osztály öröklődik a DeclarationStatement osztályból, ami öröklődik a BaseNode osztályból. Ezért az ExportAllDeclaration egyaránt öröklődik a BaseNode osztályból. A 2.1 kódrészletben ez az extends BaseNode-al van jelezve. A főbb osztályok a BaseNode alatt találhatóak az ábrán. Az ábra jobb oldalán lévő osztályok, amik sötét- és világosszürkével vannak jelölve, az attribútumok. Az ExportAllDeclaration osztály öröklődik a Statement, DeclarationStatement, Node és a ProgramStatement osztályokból. Az összes öröklődés a TypeScript-eslint [6] projekt unions mappájában találhatóak meg. Az ExportAllDeclaration osztály rendelkezik pozíció, Comment, type és NodeId attribútummal, mivel öröklődik a BaseNode osztályból. Emellett még rendelkezik Assertions, Source és Exported attribútummal is. Az Assertions attribútum típusa ImportAttribute, amihez legalább egy ImportAttribute tartozik. Az Exported attribútumnál típusa lehet vagy Identifier vagy LiteralExpression. A vagyolást egy OR-ral jeleztük a sémában. Az Exported attribútum opcionális, ezért az értéke lehet null is. Ezt a sémában a 0..1-el jelöltük. A Source attribútum is opcionális, típusa StringLiteral. Végül, ExportKind attribútummal is rendelkezik, típusa ExportAndImportKind. Az ExportAndImportKind az a DataStructures package-ben lett definiálva. Így lett minden package és osztály felépítve. A DeclarationStatement, Statement, ProgramStatement és több osztály, ami a unions mappában található, gyűjtő osztály, aminek a jelentősége a javaScriptAddonnál fog mutatkozni.

A DataStructures package többször volt említve, hadd mutassam be a következő ábrán a felépítését:

2.4. ábra. A DataStructures package felépítése



A DataStructures package-ben található 2 package, Kinds és Typedefs. A 2.1 ábrán látható. Ebből a Kinds package-et mutatom be. A 2.4 ábrán a Kinds package-nek egy kisebb része látható. Enumok vannak deklarálva ebben a package-ben. Az enumokban konstansok találhatóak. Minden konstans előtt található 3 karakter, ezek a karakterek ASG konvertáláskor el fognak tűnni. Ha ExportAndImportKind-ot adunk meg egy attribútum típusának, akkor az attribútum típusa lehet Type vagy Value.

Ahhoz, hogy tudjuk használni a sémát, először ki kell exportálni. Ezt a vpp segítségével tehetjük meg, egy xml formátumú fájlba exportáljuk az egész projektet. Ezután ezt a fájlt átkonvertáljuk egy asg kiterjesztésű fájlra. A SourceMeter alprojektei közé tartozik egy úgynevezett UmlToAsg. Ez a projekt egy xml fájlból egy asg fájlt generál. Az UmlToAsg projektet a SchemaGenerator segítségével tudjuk legenerálni. A SchemaGenerator C++ nyelven megírt eszköz. Több mindent is generál C, C++, vagy Java nyelven. A SchemaGenerator is a SourceMeter alprojektei közé tartozik. A legenerált asg fájl elején a következő sorok láthatóak:

```
1 NAME = javascript;
2 APIVERSION = 0.3.1;
3 BINARYVERSION = 0.3.1;
4 CSIHEADERTEXT = JavaScriptLanguage;
```

Kódrészlet 2.2. Asg fájl első sorai

A verziókat kézzel tudjuk átírni abban a fáljban ami generálja ezt, ez egy C++ fájl, a SchemaGeneratorban található meg. Ezután a Kinds package tartalmát írja bele a következőképpen:

```
KIND ASTNodeTypes (ant) {
      FunctionDeclaration;
      BlockStatement;
      ClassDeclaration;}
```

Kódrészlet 2.3. Asg fájl kind

A 3 karaktert, amit minden konstans elé írtunk, kikerült paraméterbe és csak az utáni stringet írta át. Ugyanígy jár el a többi enumnál is. Ha az összes enumot beleírta, akkor a többi package-et kezdi el bele írni a fájlba. A 2.1 alapján megy sorba. Példának a Declaration package-et mutatom be, azon belül is az ExportAllDeclaration osztályt.

```
SCOPE declaration {
      NODE DeclarationStatement : virtual base::BaseNode [ABSTRACT] {
      }
      NODE ExportAllDeclaration : DeclarationStatement, statement::
          Statement, virtual statement::ProgramStatement, special::Node
           {
            ATTR ExportAndImportKind exportKind;
            EDGE TREE 1 hasExported (expression::Identifier |
                expression::LiteralExpression);
            EDGE TREE 1 hasSource (structure::StringLiteral);
            EDGE TREE * hasAssertions (special::ImportAttribute);
      }
}
```

Kódrészlet 2.4. Asg fájl ExportAllDeclaration

Az UmlToAsg minden egyes package-et Scope-nak értelmez. A Scope-on belül az osztályokat node-nak értelmezi. A node-nak kettő különféle attribútuma lehet, ATTR és EDGE TREE. Akkor konvertálja ATTR-ra az attribútumot, ha az a DataStructures package-ben deklarálva lett. Egyéb esetben EDGE TREE-vé konvertál. Az EDGE TREE után álló szám vagy csillag, az adott attribútum multiplicitását jelenti. A vagyolást egy | jellel jelzi. Az öröklődést kettősponttal jelzi. Felsorolásszerűen írta át, ha valami másból származik, akkor packageNev::osztalyNev szerint. A 2.3as ábrán látható, hogy mit hogyan írt át.

2.2.2. JavaScriptAddon bevezetés

TODO: nodeWrapperHeader és Wrapper fájl magyarázása

A JavaScriptAddon az egy node kiterjesztésű fájl, amit a SchemaGenerator generál. Minden egyes node-nak külön generál nodeWrapperheader és egy nodeWrapperCC fájlt. A SchemaGenerator által generált Factory.cc, Factory.h, az összes nodeWrapper.cc és a hozzátartozó nodeWrapper.h fájlok összeolvasztása eredményezi a javascriptAddon.node fájlt. A SchemaGeneratornak több generálása is definiálva van, hiszen több funkciója is van. A JavaScriptAddon generálást a NodeAddonGenerator fájlban hajtjuk végre. A main.c fájlba importáljuk a NodeAddonGenerator metódusait, és ezeket használva legeneráljuk a JavaScriptAddon fájlt. A következő oldalakon bemutatom, hogy a generálás hogyan zajlik. A SchemaGeneratornak több kapcsolója is van, köztük a genNodeAddon. Ha a genNodeAddon kapcsoló meg van adva, akkor legenerálja a JavaScriptAddon fájlt. Minden kapcsolóra vizsgál a SchemaGenerator.

```
1 if(!strcmp(argv[i], "-genNodeAddon")){
2       options.generateNodeAddon = true;
3 }
```

Kódrészlet 2.5. SchemaGenerator kapcsoló vizsgálás

Ha az argumentumban megtalálható a genNodeAddon, akkor az options.generateNodeAddon-t igazra állítja. Az alapértelmezett értéke az options.generateNodeAddon-nak hamis. Több fájl generálás után, ami kell több program működéséhez, megvizsgálja az options.generateNodeAddon-t. Ha az options.generateNodeAddon hamis, akkor nem történik semmi, tovább megy.

```
1  if (createAndEnterDirectory(SOURCE_NODE_ADDON_DIR_NAME)) {
2        generatePackageJson();
3        generateBindingGyp();
4        generateAddonCC();
5        generateFactoryWrapper();
6        if (createAndEnterDirectory("inc")) {
7              generateWrapperHeaders();
8              leaveDirectory();}
9        if(createAndEnterDirectory("src")){
10             generateWrapperSources();
```

```
11              leaveDirectory();}
12 leaveDirectory();}
```

Kódrészlet 2.6. SchemaGenerator JavaScriptAddon generálás

A createAndEnterDirectory metódus létrehoz addon nevű mappát és chdir parancsot használva belelép. Ha az addon mappa már létezik, akkor a létrehozást kihagyja és a chdir paranccsal belelép. Létrehozás és chdir parancs használat után az addon mappában generál egy a package.json fájlt. Ebben a fájljban beállítja a projekt nevét, verziószámát, függőségeket, szkripteket és a gypfájl kapcsolónak igaz értéket beállítja.

```
1 fprintf(f, "    \"rebuild\": \"node-gyp configure && node-gyp rebuild -
    j 8\",\n");
2 fprintf(f, "    \"install\": \"node-gyp configure && node-gyp build -j
    8\"\n");
```

Kódrészlet 2.7. NodeAddonGenerator package.json szkriptek

A 2.7 kódrészleten látható, hogy a gyp segítségével fog történni a generálás.

Ezután legenerálja a binding.gyp fájlt.

TODO: Wrapper és headerek lefordítása

A binding.gyp fájl fog felelni azért, hogy a JavascriptAddonhoz sikeresen generálódjanak le a wrapperek és azok headerjei. Ezután fogja legenerálni az addonCC-t. Az addon.cc fájlba importálja a Factory.h fájlt, amiben több metódus is megtalálható.

,

```
1 if (!traversalDescendantBFT(rootNode, generateWrapIncludes, false)) {
2       debugMessage(0, " failed\n");
3       fclose(f);
4       return false;
5 }
```

Kódrészlet 2.8. Addon.cc Wrapperek importolása

A 2.8 kódrészletben a traversalDescendantBFT metódust egy másik projektből importáltuk. Ez egy bejárás, ami az összes node-ra lefut, és az összes node-ra meghív egy metódust, jelen esetben a generateWrapIncludes metódust. Ez a generateWrapIncludes a nodeAddonGenerator.c fájlban van leimplementálva a következőképp:

,

```
1 if(node->type.abstract){
2       return true;
3 }
4 fprintf(f, "#include \"");
5 fprintf(f, "inc/%sWrapper.h\"\n", node->name );
6 return true;
```

Kódrészlet 2.9. generateWrapIncludes leimplementálása

Először megvizsgálja, hogy az adott node absztrakt-e, ha nem akkor tovább megy, és az addon.cc-be importálja az adott nodeWrappernek a headerjét. A node.name több értéket is vehet fel, például FunctionDeclaration, ClassDeclaration, amik megtalálhatóak az asg fájlban. Ezután ezt a bejárást mégegyszer végrehajtja, de most a wrapperIniteket generálja az addon.cc fájlba. Annyi változtatással, hogy picit mást ír az addon.cc fájlba.

```
1 fprintf(f, "  columbus::%s::asg::addon::%sWrapper::Init(env, exports);\
    n", schemaName, node->name);
```

Kódrészlet 2.10. generateWrapInit leimplementálása

Ezek után az addon.cc fájlt sikeresen legeneráltuk. Benne találhatóak a wrapperek headerjének importolása és a wrapperek Initjei.

Ezután következik egy generateFactoryWrapper függvényhívás. A generateFactoryWrapperben 2 metódus függvényhívás található, a generateFactoryWrapperHeader és generateFactoryWrapperSource.

A generateFactoryWrapperHeader a Factory.h fájlt generálja le a következőképp: Először importálja az összes nodeWrappernek a headerjét, és utána létrehoz egy Factory osztályt, a publikus metódusait, ami az Init és Destructor, illetve a private metódusait, ami a destructor, New, SaveAST, LoadAST, Clear, getRoot és az összes nodeWrapper create metódusait. Erre azért van szükség, mivel majd a JSAN-ban ezeket a wrappereket fogjuk létrehozni és szerkeszteni.

A generateFactoryWrapperSource pedig a Factory.cc fájlt generálja le. Ebben a fájlban megvan az összes metódus valósítva, ami a headerjében található. A Factory Initjének a props változója a következőképpen néz ki:

```
1 napi_property_descriptor props [] = {
2       DECLARE_NAPI_METHOD("getRoot", getRoot),
```

```
3       DECLARE_NAPI_METHOD("createCommentWrapper", createCommentWrapper)
4       ...}
```

Kódrészlet 2.11. Factory.cc fájl

DeclareNapiMethodokat hoz létre, az összes nodeWrappernek, ezeket majd a JSAN-ban fogjuk használni.

Utolsó lépésként, az inc mappát majd az src mappát generálja le a SchemaGenerator. Az inc mappában találhatóak a header fájlok egyes wrappereknek, az src mappában maguk a wrapperek vannak megvalósítva. Minden node-nak hoz létre wrapper fájlt, és ezekben valósít meg több metódust. Létrehoz egy ExportAllDeclarationWrapper.h és egy ExportAllDeclaration.cc fájlt. A header fájlban létrehoz egy ExportAllDeclarationWrapper osztályt, ami öröklődik a BaseWrapperből. A BaseWrapper alap Wrapper osztály, amit bővíteni fogunk. Három publikos metódust generál, Init, Destructor és a NewInstancet. Ezen felül több private metódust is.

```
1 napi_property_descriptor props [] = {
2       static napi_value setPosition(...);
3       static napi_value addAssertions(...);
4       static napi_value setType(...);
5 ...}
```

Kódrészlet 2.12. ExportAllDeclaration.h fájl

Látható a 2.12 kódrészleten, hogy létrehozott az ExportAllDeclarationWrappernek egy setPosition, addAssertions, setType és még több metódust. Ezek az ExportAllDeclaration attribútumai, amit a sémában beállítottunk. A 2.3 ábrán látható, hogy az ExportAllDeclaration hasAssertions attribútumának a multiplicitása 1..*, ezt átkonvertálta addAssertionsre. Ahol 0..* a multiplicitás, azt átkonvertálta setAttribute-ra, pl setSource vagy setExportedre. Minden nodeWrappernek van olyan metódusa, hogy setPath, setPosition, setType, addComments.

Az ExportAllDeclaration.cc fájlban ugyanaz történik, mint ami volt a Factory.cc fájlban. DECLARE_NAPI_METHOD-okat generál, a setExported, setSource, setType és a többi metódushoz. Ezután az összes metódust megvalósítja.

2.2.3. JSAN2Lim bevezetés

2.2.4. Regressziós tesztelés bevezetés

3. fejezet

Oszthatlan közösen készített programok

3.1. A nyelvi séma átírása

A JSAN a JavaScriptSchemára épül. Ha valamit meg akarunk változtatni gyökerestül a JSANba, akkor a schémát is változtatni kell. Azt elérni, hogy javascript mellett még typescriptes kódokat is elemezzen a JSAN, ahhoz gyökerestül meg kellett változtatni a JavaScript Analyzert. Az előző JavaScriptSchema (ami csak javascriptet elemzett) az a javascript hivatalos oldala alapján készült. Ami ábrák fentebb megtalálhatóak, azok már az átírt schémából vannak. Több opció is volt, hogy most vagy legyen átírva a schéma, vagy legyen egy külön typescriptre is írva. Végül rájöttünk, hogy ha van egy typescriptes schémánk, az képes javascriptet ugyanúgy elemezni ha megfelelően van lefejlesztve. Így egy schémát fejlesztettünk le. Az előző schéma átláthatatlan volt, ezért úgy döntöttünk, hogy majdnem a nulláról újraírjuk. Egyedül a base packaget emeltük át a régiből, BaseNode-al és BaseTokennel kibővítve (A 2.2 ábrán látható). Emellett el kellett dönteni, hogy milyen struktúrát kövessünk, ami jól átlatható és később könnyebben bővíthető. Végül a typescript-eslint official github alapján haladtunk. Annyi változtatással, hogy ami nekik a base mappában volt, mi arra létrehoztunk egy külön structure packaget. Mindenhez készítettünk dokumentációt, jól érthetően leírtuk, hogy mit miért kellett csinálni. Mivel minden is egymásra épül a typescriptes schémában, ezért nem tudtuk tesztelni minden egyes package után, hogy működik-e vagy sem. Mint például, látható a 2.4 kódrészleten, hogy az ExportAllDeclaration mennyi mindenből származik le vagy mennyire sok attribútuma van. Az ExportAllDeclaration egyik attribútuma, az assertions típusa az Im-

portAttribute. Ahhoz, hogy jól észrevegye az ExportAllDeclarationt az Analyzer, ahhoz jól le kellett fejleszteni az ImportAttributet.

```
1 export interface ImportAttribute extends BaseNode {
2       type: AST_NODE_TYPES.ImportAttribute;
3       key: Identifier | Literal;
4       value: Literal;
5 }
```

Kódrészlet 3.1. ImportAttribute

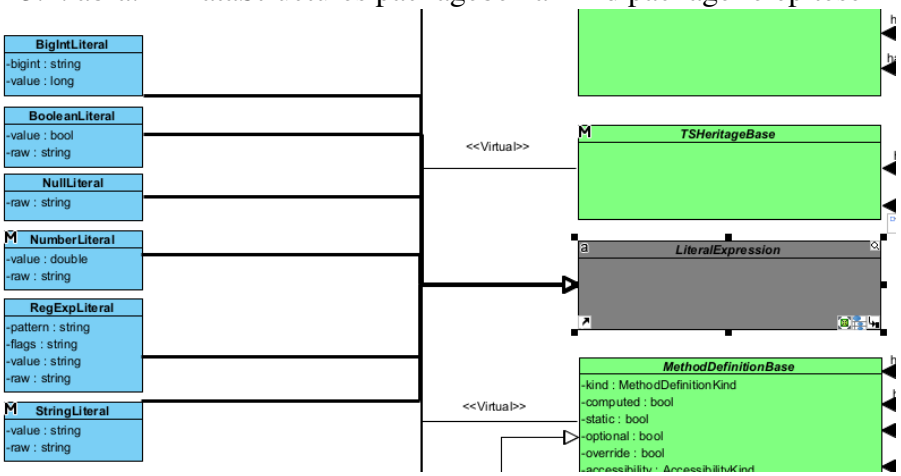
Ahhoz, hogy az ImportAttributet letudjuk fejleszteni a schémában, ahhoz le kell fejleszteni az Identifiert, és a Literalt. Kicsit összetett, hogy mi minden épül egymásra. Ebből adódik a következő alfejezet mondandója is.

3.2. Nehézségek, problémák

A schéma átírása során több nehézségbe is ütköztünk. A legelső nehézség az a Visual Paradigm program korrekt használata volt. Egy kisebb időbe tellett rájönni arra, hogy egy adott classnak hogyan kell megváltoztatni a háttérszínét, packagek közötti mozgást, hogyan kell egy adott classra hivatkozni ami másik packageben volt. Ezután ami szerintem a legnagyobb nehézség volt, az az, hogy nem volt dokumentáció az előző schémához. Mindent nekünk kellett kitalálni, hogy mit miért csináltak az előző fejlesztők. Akik ezt a schémát fejlesztették le, ők már nem foglalkoztak ezzel, és más nem nagyon mélyedt bele ebbe az egészbe azóta. A következő nagyobb nehézség, az az volt, hogy összehozzunk egy olyan schémát ami működőképes, és az analyzer ezt tudja is használni. Előző alfejezetben említettem, hogy egy adott classt (mint például ExportAllDeclaration) lefejlesszünk, ahhoz sok minden mást is le kell fejleszteni, amihez meg még több minden kellett. Ebből adódóan tesztelni nagyon nem tudtunk, mivel az egésznek működnie kellett ahhoz, hogy az analyzer egyáltalán lefusson. Mivel mi majdnem nulláról írtuk újra, ez volt a hátrány. Amikor elérkeztünk ahhoz az állapothoz, hogy mindenre rámondjuk azt, hogy működik, akkor jött egy nagyobb probléma. Valahol Segmentation Fault-ot kapott a program, és nem nagyon tudtuk ezt debugolni. Több sejtésünk is volt, hogy mi lehet a baj. Emiatt az egész schémát át kellett nézni alaposan, hogy hol vétettünk hibákat. Sok

helyen voltak pontatlanságok, rossz osztályból származtattunk le, rossz típus volt megadva attribútumnak. Ezeket mind kijavítottuk, de most már más hibát kaptunk. A JavaScript Analyzerben kiderítettük, hogy pontosan hol szállt el a program. A Literal volt a hiba, mivel ezt nem a github alapján írtuk meg, hanem egyedi ötlettel. Előző schémában is máshogy volt megoldva.

3.1. ábra. A DataStructures packageben a Kind package felépítése



A 3.1 ábrán látható, hogy a LiteralExpressionből (Ami a Literal) több literal is öröklődik.

```
1 export interface LiteralBase extends BaseNode {
2   type: AST_NODE_TYPES.Literal;
3   raw: string;
4   value: RegExp | bigint | boolean | number | string | null;
5 }
```

Kódrészlet 3.2. Literal

Próbáltuk úgy megoldani, hogy 4 vagyolással 4 különböző literál tartalmazza, de az analyzerben máshogy kezeltük le a literalt, mint nodeot. Ezért a tartalmazás helyett inkább örököltettünk a literalból, így megoldva ezt a problémát. Még az is probléma volt, hogy volt egy LiteralBaseünk, amiből származott ez az 5 literal. A LiteralBase származott a LiteralExpressionből, de valamiért Segmentation fault lett a vége ha így próbáltuk megoldani. Ezért a LiteralBase-t kivettük, és LiteralExpressionből származik minden literal. Végül még az volt a nehézség, hogy kódrészletet keressünk az analyzernek, amin le tudjuk tesztelni, hogy az analyzer jó outputot ad-e.

Mivel nem nagyon volt typescriptes háttértudásunk, először a typescriptet kellett átnézni, hogy mit hogyan tudunk megvalósítani. Emellett még a javascriptes outputokat is át kellett nézni, mivel a mi célunk a fejlesztés volt. A jelenlegi javascriptes projektekre ugyanolyan eredményt adott, sőt néhány helyen jobbat is, mert az előző schémában is voltak pontatlanságok. A typescriptes projektek nagy részét tudja elemezni az analyzer, de közel sem tökéletes, mivel egyfolytában kellene fejleszteni a schémát, mivel a typescript nem annyira régi.

4. fejezet

Általam változtatott programok

4.1. JavaScriptAddon változások

A JavaScriptSchema változtatások után, a NodeAddonGeneratort nem módosítottuk. Ennek ellenére a javascriptAddon.node mérete megduplázódott, csupán azért, mert a schéma ekkora bővítésen esett át. Sokkal több nodeWrapper lett, mivel bejöttek a typescript általi dolgok is a javascript mellett. Természetesen a javascriptes dolgok megmaradtak, így javascriptet ugyanolyan jól tud elemezni, sőt néhány esetben jobban is, mert voltak figyelmetlenségek előző verzióban.

Mivel a javascriptAddon mérete nőtt, ezáltal a sebesség csökkent. Ezt majd egy másik fejezetben jobban kifejtem, célunk az volt, hogy először tudjon typescriptes fileokat és projekteket elemezni a jsan.

4.2. JSAN2Lim

A JSAN2Lim egy c++-ban megírt program. A JSAN általi outputot átalakítja lim formátumú fileokra. Azért lim formátumú fileokra, mivel a lim az egy környezetfüggetlen nyelv, és sokkal könnyebb ezen metrikákat mérni. A JSAN2Lim is egy alprojektje az Analyzer JavaScriptnek. Fontos, hogyha a JSAN programot fejlesztik, akkor a JSAN2Lim-et is fejleszteni kell, mivel ha nem, akkor a metrikák nem fognak jól számolódni. A JSAN2Lim használja a LimSchemat, ez is ugyanúgy egy $.vpp$ kiterjesztésű file. Más logikával van felépítve, mint a JavaScriptSchema, ezért nem is nagyon mennék bele

az elemzésébe, csak csupán néhány dolgot fogok használni belőle. A JSAN2Lim visiteli az összes nodeot amit a JSAN outputba kirak, ezért JSAN2LimVisitor a fő file neve. Minden egyes nodera külön meg kell írni a visit függvényt, hogy tudja a c++ program, hogy mit csináljon ha olyan nodeot kap.

A JSAN2Lim működését egy kódrészleten keresztül mutatom be.

```
1  void JSAN2LimVisitor::visit(const javascript::asg::structure::
       ClassDeclarationBase& clNode, bool callVirtualBase)
2  {
3        VISIT_BEGIN(clNode, callVirtualBase);
4        lim::asg::logical::Class& classLimNode = dynamic_cast<lim::asg::
       logical::Class&>(createLimNode(clNode, callVirtualBase));
5        fillData(classLimNode, clNode);//sets the name
6        fillMemberData(classLimNode);
7        demangledNameParts.push_back(classLimNode.getName());
8        classStack.push(ClassInfo());
9        classStack.top().classNodeId = classLimNode.getId();
10       if (clNode.getSuperClass() != NULL) {
11       if (javascript::asg::Common::getIsIdentifier(*clNode.
       getSuperClass())) {
12       javascript::asg::expression::Identifier& superClass =
       dynamic_cast<javascript::asg::expression::Identifier&>(*clNode.
       getSuperClass());
13       classStack.top().isSubclass = superClass.getId();
14       }
15       }
16       classStack.top().TLOC = clNode.getPosition().getEndLine() -
       clNode.getPosition().getLine() + 1;
17       classStack.top().LOC = clNode.getPosition().getEndLine() - clNode
       .getPosition().getLine() + 1;
18       if (!packageStack.empty()) {
19       lim::asg::logical::Package& packageLimNode = dynamic_cast<lim::
       asg::logical::Package&>(limFactory.getRef(packageStack.top().
       packageNodeId));
20       SAFE_EDGE(packageLimNode, Member, lim::asg::logical::Member,
       classLimNode.getId());
21       }
```

```
22        else {
23        common::WriteMsg::write(common::WriteMsg::mlDebug, "Empty
      packageStack while visiting node %d.", clNode.getId());
24        }
25        addIsContainedInEdge(classLimNode, clNode);
26  }
```

Kódrészlet 4.1. ClassDeclaration Visitor

A VISIT_BEGIN definiálva van a file elején, ami a ViscitorAbstractNodesnak a visit metódusa. Ezután létrehozok egy classLimNode nevű változót, ami $lim :: asg :: logical :: Class$ típusú, ez a limFactoryba található meg. Ennek a változónak adunk egy értéket, ami a createLimNode által visszaadott node lesz, és ezt castoljuk erre a típusra.

A createLimNode meghívja a limFactoryból a createNode-ot, ami vár paraméterbe egy nodeKind-ot. Ezt a nodeKind-ot mi adjuk meg neki, az alapján, hogy az adott node milyen típusú. A nodeKindot a getLimKind függvény határozza meg a következők szerint. Megvizsgáljuk, hogy az adott node $getIsClassDeclarationBase()$ vagy $getIsMethodDefinition$ vagy $getIsFunctionDeclarationBase$ vagy sorolhatnám még. Ebből van 10-15db. Típustól szerint adjuk meg a nodeKindnak az értékeket, amik lehetnek a következők: ndkClass, ndkMethod, ndkParameter, ndkMethodCall, ndkAttribute, ndkComment, ndkPackage, ndkAttributeAccess, ndkComponent. Ezek mind a lim-nek az asgben való típusok és mindet tudja kezelni. Miután beállítottam a nodeKind értékét, szimplán returnolom. Ha megkaptuk a nodeKind értékét, akkor ezt létrehozza a createLimNode függvény. Állít be ennek pozíciót és returnoli a *limNode-ot.

Ezáltal most megkaptuk a classLimNode-ot, hiszen a nodeKind az ndkClass lett és ennek megfelelően hozta létre a lim. Ezután meghívjuk a filldata függvényt, első paraméternek magát a létrehozott classLimNodeot, másodiknak magát a nodeot adjuk meg. A JSAN által kiadott outputot $.jssi$ kiterjesztésű fileon végig megy. Visitorok segítségével oldja meg ezt. Ebben a filldatában beállítom a limnode láthatóságát, nyelvét (ami konstans javascript), azt, hogy abstract-e. Lekérem a nodeIdját és a lastLimMemberNodeId illetve a lastLimScopeNodeIdba pusholom. Ezután beállítom a classKindját, ami clkClass lesz (ezt is a factoryból). Aztán a nevét szeretném beállítani, a node-nak le-

kérem azt,hogy identifier-e, vagy sem. Ha nem null, akkor lekérem az identifiert és $identifier.getName()$el lekérem a nevét és be is állítom a limNodenak a $setName()$ függvénnyel. Ha az identifier null, akkor a $limNode.setIsAnonymous()$-t beállítom true értékre és adok neki egy "$anonymousClass$" + $ss.str()$ ahol az ss az egy stringstream, szimplán egy counter, hogy eddig hány anonymous class volt. Ezután megvizsgálom, hogy e methodstack empty-e, ha nem, akkor van szülője, ezért a SAFE_EDGE function hozzáadok egy eegy edget a parent és a jelenlegi limnode-hoz. Ezt ugyanúgy megvizsgálom packageStackre, ha nem empty, akkor ugyanezt elvégzem csak a parent most más lesz. Végezetül az $addIsContainedInEdge$ függvényt meghívom a limnode és a jsNode paraméterekkel.

Ezután a $fillMemberData()$ függvényt hívjuk meg, ami igazából csak pozíciót állít be, illetve a neven picit változtat. Utána a classStack-hez hozzáadja az adott osztály tulajdonságait, beállítja a parent classt ha van, pozíciót állít be, és a végén ugyanúgy meghívjuk a $addIsContainedInEdge$ a classLimNode és a clNodeal. Ez a procedúra történik minden egyes nodenál a JSAN2Limben.

4.2.1. Bővítések

Mivel a JSAN mostmár tud typescriptet is elemezni, emiatt a JSAN2Lim-et is fel kell erre készíteni, hogyha typescriptes nodeokat kap, akkor tudja, hogy mit kell azzal csinálni. A javascript asg-t is használja a JSAN2Lim, ezért sok helyen először változtatni kellett néhány dolgon.

- Néhány visitornál a várt paraméter típusát átírni, például Classról ClassDeclarationBase-re, hiszen a schémában változtatás volt.
- Néhány visitornál a várt paraméternél a helyét az adott nodenak. Például a RestElement már nem a statement packagen belül van, hanem a parameteren belül.
- A Pattern mint node megszűnt, Parameterre lett átnevezve, és a helye is megváltozott, statementből átkerült parameter packagebe
- Típus lekérdezésnél a $getIsClass()$ helyett például

$getIsClassDeclarationBase()$ lett, változott ez is a JavaScriptSchema miatt.

A bővítések, hogy miket adtam hozzá a JSAN2LImhez:

- Az új Kindokat amiket felvettünk a JavaScriptSchemaban. A következőképp: $std :: stringgetEnumText(javascript :: asg :: ASTNodeTypes)$;
- Új visitorok írása. TSEnumDeclaration, TSImportEqualsDeclaration, TSInterFaceDeclaration, és még sok másik, ezekhez a filldatákat is megírni
- kindStrings tömb kiegészítése a typescriptes nodeokkal
- Bugok kijavítása, a VariableDeclarationnél nem vette észre az összes declarationt, főleg metóduson belül.
- A $getLimKind()$ function bővítése $getIsTSTypeAliasDeclaration()$, $getIsTSInterfaceDeclaration()$, $getIsTSAbstractMethodDefinition()$ és $getIsTSAbstractPropertyDefinition()$ else if ágakkal.
- TSEmptyBodyFunctionExpression debugolása, sok helyen lehalt a program emiatt.

4.3. Ast binder optimalizálás

Az AST binder referenciákat bindol. Ezek a referenciák a JSAN2Lim programhoz kellenek. Binder néven van definiálva az astTransformer.js fileban. Kétszer van használva, ezért is kulcsfontosságú, hogy gyors legyen. Egyszer VariableUsages referenciákat bindol és egyszer ACG referenciákat bindol.

A binder 4 argumentumot vár:

- Egy stringet, ami lehet vagy VU (ami VariableUsages-t rövidíti) vagy ACG (egy javascript callgraph).
- Egy Abstract Syntax Tree-t (AST), ami egyedien van felépítve.
- Egy tömböt, amiben JSON objektumok találhatóak, amik a linkeket tartalmazzák.

- Végül még egy stringet, ami lehet addCalls, vagy setRefersTo. Az AddCalls és a SetRefersTo a javascriptAddonban található meg és onnan hívódik meg. Akkor addCalls ha ACG referenciákat bindolunk és akkor setRefersTo ha VU referenciákat.

A linkeket tartalmó tömb egyik JSON objektuma a következőképp néz ki:

```
source: {
      label: IdentifierNeve,
      file: AbsPath,
      start: { row: <int>, column: <int>},
      end: { row: <int>, column: <int>},
      range: { start: <int>, end: <int>},
      node: [Object]
},
target: {...}
```

Kódrészlet 4.2. Binder JSON argumentuma

A 4.2 kódrészletben egy JSON elemét láthatjuk a tömbnek. Ilyen JSON elemekből épül fel a tömb. A source és a target felépítése ugyanaz, csupán másak az értékek bennük. A kódrészletben csak a sourcet fejtettem ki kicsit bővebben. A label az egy Identifier vagy PrivateIdentifier nevét fogja jelölni. A file egy abszolút útvonalat, ami az Identifiert tartalmazó filet jelöli. A start az egy object, amiben külön van sor és oszlop meghatározva. Ez az Identifier első karakter pozíciójának a sor és oszlop értékei. Az end ugyanaz, mint a start, csak itt az utolsó karakter pozíciójának a sor és oszlop értékei. A range is egy object, a start ennél a kezdő karakter hanyadik karakter volt a kódban, és az end pedig az identifier utolsó karakterének a karakterszáma. A node is egy object ami maga az Identifier vagy PrivateIdentifier.

4.3.1. Lassúság okai

A binder nagyobb projektekre lassan fut le. Ennek több oka is van:

- Minden hívásnál meghívja a javascriptAddon.nodeot, ami már magában nagy file. TypeScript támogatás után kétszeres lett a mérete, ezáltal sokkal lassabb lett.

- Mivel nagyobb a projekt, ezért sokkal több a függvényhívás is az elemzett kódban, emellett sok a változószám is.
- Az AST nagyon nagy, és ezt bejárjuk többször is bindolás alatt. Emiatt nagyon lassú a program nagy projektek esetén.
- A JSONokat tartalmazó tömb is nagyon nagy lesz, de annyira ez nem lassítja le a programot.

4.3.2. Optimalizálás

Először ki kellett derítenem, pontosan melyik funkció mennyi memóriát vesz igénybe és milyen gyorsan fut le. Erre találtam egy javascript profilert, ami kiírta minden függvényhívásnál a lefoglalt memóriát és a memória használatot. Ez nekem nem volt a legjobb, mivel nagyon egybe van ágyazva minden, és eléggé mélyre mentek a functionok. Ezután próbáltam picit egyszerűbbet, a binder kódját 3 részre osztani, $console.time()$ és $console.timeEnd()$ használatával. Ez a $console.time()$ parancs kiírja nekem ms-ben, hogy mennyi idő telt el amíg eljutott a $console.timeEnd()$ig. Így megkaptam, hogy melyik rész nagyjából mennyi idő alatt fut le, könnyebb volt szűkíteni, hogy melyik function lesz a bajos. Végül kiderült, hogy a következő sor lassítja be nagyon a programot:

```
1 globals.getWrapperOfNode(resolveNode(astSet, sourceFile, element.source
    .range.start, element.source.range.end, true));
```

Kódrészlet 4.3. Problémás function

Ezen belül is a resolveNode function volt a probléma. A resolveNode kapott egy ast-t (amit a binder kapott meg paraméterbe), egy filenevet, kezdő- és végPozíciót, illetve egy true vagy false értéket, ami attól függ, hogy sourcenodeot vagy targetnodeot keresünk. Az ast-n forEach-el végigmentünk, amin belül az astNodeon walkoltunk addig amíg nem találtuk meg a számunkra megfelelő nodeot. Rosszabb esetben a legvégén volt a node, mivel már a végén voltunk a bindolásnak, és ebből is látható volt, hogy ez kellően sok időbe is kerülhet, ha rengeteg nodeunk van. Kisebb projekt esetében ez nem baj, mivel az ast abban az esetben nem annyira nagy. Ahol már van 100 vagy 200 ezer változó és vagy függvényhívás, ott már nagyobb a baj, mivel ezt rengetegszer meg kell ismételni. Ezután

ötleteltem, hogy hogyan tudnám jobbra átírni a resolveNodeot functiont, vagy egy másik logika alapján megközelíteni a problémát. Végül eszembe jutott egy sokkal könnyebb megoldás erre, ami nem igényli a sokszori bejárását az astnek.

Először is, létrehoztam egy indexAST nevű functiont, ami a következőket hajtja végre:

```
let indexAST = function (ast) {
      ast.forEach(astNode =>{
            globals.setActualFile(astNode.filename)
            walk(astNode, {
                  enter: function(node){
                  globals.setIndexed(astNode.filename, node.range[0],
                     node.range[1], node)}})})
      return globals.indexedAST}
```

Kódrészlet 4.4. indexAST function

A 4.4 kódrészletben látható, hogy egy ast-t várunk paraméterben. Ezt a bindertől fogja kapni, ez az egyénilég létrehozott ast. A functionben forEach-el végig megyünk az ast elemein, amik az astNodeok. Itt beállítjuk a $globals.setActualFile()$ függvénnyel az aktuális filenevet. Ezután az adott astNode-on elkezdünk walkolni. Csak az enter metódusát írtam meg. Meghívunk egy függvényt ami a globalsban lett definiálva, setIndexed a neve. Ennek a függvénynek megadjuk a filevenet, a node rangenek a kezdő- és a végparaméterét (tehát ahol kezdődik az adott node és hol végződik, karakterpontosan), és magát a nodeot.

```
const setIndexed = function(filename, range_start, range_end, node){
      let actualfilename = getFilePathAlt(filename)
      if (indexedAST[actualfilename + "-" + range_start + "-" +
         range_end] !== undefined && indexedAST[actualfilename + "-" +
          range_start + "-" + range_end] !== node){
            return
      }
      indexedAST[actualfilename + "-" + range_start + "-" + range_end]
         = node
}
```

Kódrészlet 4.5. setIndexed function

A setIndexed először végez egy vizsgálatot arra, hogy az adott node be van már indexelve. Ezt úgy teszi meg, hogy az indexedAST tömbben keres egy indexre (Ez az index

a következőképp néz ki: fileNev-StartPosition-Endposition), és még vizsgál is arra, hogy ha van ilyen index, akkor ezen van emár olyan node. Ha van akkor nem állít be semmit, csak returnol, mivel már be van indexelve az adott node. Ha nincs, akkor beállítja az indexedAST tömbnek az adott indexre az adott nodeot.

Ezáltal az adott astn csupán csak egyszer megyünk végig foreach-el és egyszer walkolunk, ekkor egy tömbbe beindelexünk minden egyes létező nodeot ami kellhet nekünk. Ha ezzel megvoltunk akkor utána kezdődhet a binder része. A 4.3 látható, hogy először megkerestük a nodeot index alapján és aztán hívtuk meg rá a $getWrapperOfNode()$ függvényt. Ezt is megváltoztattam, írtam rá egy getIndexed függvényt. A $getIndexed()$ függvény megvizsgálja, hogy source vagy targetNodeot keresünk. Ha targetNodeot, akkor returnoljuk a $getWrapperOfNode(indexedAST[filename - StartPosition - Endposition])$-t. Ha targetNodeot keresünk, akkor walkolunk ismét, de nem az ast-ben, hanem már a beindexelt tömbben. Ez lényegesen gyorsabb, mint a resolveNodeos megoldás, mert ott minden egyes esetben az ast-n walkoltunk, itt meg csak egy beindexelt elemén a tömbnek. Ha megkaptuk a sourceNodeot akkor returnoljuk a $getWrapperOfNode(result)$t.

5. fejezet

Regteszt frissítés

5.1. A regressziós tesztelésről

A Sourcemeter Javascript projektben regressziós tesztelés folyik, mint tesztelési folyamat. A regressziós teszt segítségével hamar tudunk hibákat kiszűrni fejlesztés során. Ez a következőképpen zajlik a projekten:

- Cmake segítségével először legeneráltatjuk a megfelelő fileokat. Vcxproj illetve make fileokat, használt operációs rendszertől függően.
- Visual Studio 2017 vagy make segítségével lebuildeljük a Regtest_javascript targetet.
- A Regtest_javascript buildelése során ellenőrizzük, hogy az adott projekt le van-e már buildelve. Ha nem, akkor lebuildeljük tesztelés előtt.
- Tesztelés közben, programonként írja ki a konzolra, hogy sikeres vagy sem az adott teszt.
- Tesztelés végén egy regtest.xml fileba írja az összesített eredményeket fileokra lebontva.
- Ha valami differencia van a referenciához képest, akkor azt egy külön diff kiterjesztésű fileban jelzi a rendszer nekünk, az elvárt és a kapott eredményt beleírva.

5.2. Tesztekről

A Regtest_javascript target több programot foglal magában, nem csak a JSAN-t. Egészen pontosan a következőket: JSAN, JSAN2Lim, LIM2Metrics, LIM2Patterns, ESLintrunner, ESLint2Graph, ChangeTracker, DuplicatedCodefinder és SourceMeter projekteknek tartalmazza a tesztjeit javascriptes (és mostmár typescriptes) fileokra. Én a JSAN, JSAN2Lim, ESLintrunner és a SourceMeter projekteknek változtattam a tesztelésén. Eddig a tesztelés úgy zajlott, hogy beadtunk inputnak pár nagyobb filet, és lefuttattuk rá a programot amit tesztelni szerettünk volna, és megnéztük az outputot. Azt kaptam feladatnak, hogy a tesztelési logikát írjam át arra, hogy kapcsolókra is teszteljünk. Ez azt jelentette, hogy minden kisebb projektnek vannak külön kapcsolói. Eddig mindig csak egy adott sorral futtattuk le a programot, és nem volt az tesztelve, hogy pl useRelativePath működik-e az elvártak szerint vagy sem.

5.3. Tesztek kibővítése

Először kezdtem a JSAN tesztek átírásával. A JSAN-nak a következő kapcsolói voltak:

- -i:Jelentése input, lehet relatív vagy abszolút útvonal a filehoz vagy projekthez.
- -o:Jelentése output neve, lehet relatív vagy abszolút útvonal.
- -d:Jelentése dumpjsml, a JSAN outputját átgenerálja XML stílusú fileba és ezt egy jsml fileba kiírja.
- -e:Jelentése ExternalHardFilter, relatív vagy abszolút útvonal egy olyan filehoz, ami szövegalapú és olyan syntax található benne, ami kell az externalHardFilternek
- -help:Jelentése help, kiírja minden kapcsolóhoz tartozó descriptiont.
- -r:Jelentése useRelativePath, outputban az útvonalakat átírja relatív útvonalra.
- -h:Jelentése html, a JSAN html fileokra is lefut, bennük keresve javascriptes scripteket és azokat tesztelni.

- -stat:Jelentése statistics, Kiírja a memóriahasználatot és a futásidőt amit a JSAN vett igénybe.

Az input, output, dumpjsml, ExternalHardFilter, és html kapcsolókra tudtam tesztelést írni. A logika az volt, hogy mappanév alapján tesztelek egyes kapcsolókra. ProgramozásiNyelv-kapcsolónév logikát követtem, mivel javascriptes tesztek mellé kell majd typescriptes teszteket is keresnem. A projekteket python scriptek segítségével futtattam le.

```
1 if "externalHardFilter" in input_path:
2       ret_val = self._execute_one_test(input_path, external_hard_filter
            =True) and ret_val
3       return ret_val
```

Kódrészlet 5.1. JSAN kapcsoló vizsgálat pythonban

Az 5.1 kódrészleten látható, hogy hogyan keresek egy adott kapcsolóra. Az input_pathban van a mappa is, és ugye a js-externalHardFilter-ben megtalálható az externalHardFilter szó. Az execute_one_test függvényemben beállítom a teszteléshez az adott dolgokat.

```
1 if external_hard_filter:
2       input_dir = os.path.dirname(input_path)
3       external_hard_filter_path = os.path.join(input_dir, "
            externalHardFilter.txt")
4       external_hard_filter_switch = "-e"
5 else:
6       external_hard_filter_path = ""
7       external_hard_filter_switch = ""
```

Kódrészlet 5.2. JSAN kapcsoló beállítása pythonban

Az 5.2 kódrészletben az execute_one_test függvénynek egy részét láthatjuk, ahol beállítjuk az external_hard_filter_path-t és a switchet annak függvényében, hogy igazat kaptunk e vagy sem. Utána kellett néznem, hogy egy ExternalHardFilter file hogy is néz ki, hogyan kell használni. A használata a következő: létrehozzuk az externalHardFilter filet a tesztelendő file mellé vagy a tesztelendő projekt gyökerébe, és attól függően, hogy ki akarjuk hagyni az adott filet vagy hozzáadni, írunk egy $+$ vagy egy $-$ jelet a sor elejére és

utána relatív útvonal és a file neve. Alapértelmezetten minden filet leelemez az adott program, JSAN esetében ezek azok a fileok amiknek a kiterjesztése js,jsx. (külön kapcsolóval megadhatjuk neki, hogy a html kiterjesztésű fileokat elemezze-e vagy se.) Typescriptes bővítés után már a ts és a tsx kiterjesztésű fileokat is elemzi. Ezért írtam több tesztesetet is. Egy példa, hogy hogyan néz ki ez a file:

```
1 -filtered01
2 -filtered02
3 -filtered03
4 +filtered01
```

Kódrészlet 5.3. ExternalHardFilter file

Az 5.3 kódrészletben látható, hogy filtered01 02 és 03 at kivettük, hogy azokat ne elemezze a jsan. Ezután visszavettük a filtered01et. A filtered fileok azok javascriptes fileok, javascriptes kóddal. Ezután referenciába csak a filtered01 file outputját raktam be, hiszen a 02 és 03at nem elemzi, ha elemezné, akkor szólna a program, hogy missing reference file. Természetesen lehet regexpet is használni filterezésnél.

Ezután teszteltem a $-i$ kapcsolót, itt ugye a programnak vissza kellene adnia, hogy üres az input ha nincs megadva. Ezután a $-o$ kapcsolóra teszteltem, itt ebben az esetben default értékben $out.jssi$ filet kellene visszaadnia. Végül a $-h$ kapcsolót néztem meg, itt ha megvan adva ez a kapcsoló, akkor az adott projektben a html fileokban a javascriptet kellene tesztelni. A $-d$, $-help$, $-stat$ kapcsolókra nem tudtam tesztelni, még a $-useRelativePath$ kapcsolóra sem, hiszen ha abszolút utat kérek, akkor a referenciákban másnál rossz lesz az elvárt eredmény.

Ezután a JSAN2Lim és az ESlintrunner programoknak írtam át a tesztelési menetetét, mivel nekik volt még olyan kapcsoló, amit lehetett tesztelni. A többi projektnek 1 vagy 2 volt, amik kellettek a működésükhöz, nem voltak opcionális kapcsolók.

Miután a JSAN ki lett egészítve typescriptes supportal és a JSAN2Limet átírtam, hogy a JSAN általi typescript elemzéseket jól olvassa be, ideje volt teszteket keresni, mind JSAN-nak és mind JSAN2Limnek. Kerestem egyszerűbb typescript és picit összetettebb typescriptes projekteket is, hogy lássam a hiányosságokat. Természetesen ami outputokat adtak a programok, azokat át kellett néznem egyesével, hiszen csak így tudom meg, hogy jól tesztelte-e a megadott filet vagy sem. Több hiányosságot is észrevettem, mind JSAN

oldalról, mind JSAN2Lim oldalról, ezek kisebb hiányosságok voltak. Például, hogy egy nodenak nem volt beállítva pozíció, vagy nem volt jól beállítva a paramétere.

6. fejezet

Összefoglaló

6.1. Program javulása

A szakdolgozatom során sikerült elérnem azt, hogy az Ast binder közel tízszer gyorsabban fusson le nagyobb projektre, mint ezelőtt. Leteszteltem ugyanarra a nagy projektre az eredeti JSAN lefutását és az én általam átírt lefutását. Ezt A 6.1 kódrészleten láthatjuk. Kisebb projektekre körülbelül ötszörös gyorsulás van. Lefuttattam egy nagyobb projektre a jsan-t, az optimalizált és az optimalizálatlan verzióval, hogy jobban lássuk a különbséget. A következő eredmények születtek:

```
1 Optimalizalt_verzio:
2 Binding VU: 71599/71599
3 VU binding: 1:39.140 (m:ss.mmm)
4
5 Optimalizalatlan_verzio:
6 Binding VU: 71599/71599
7 VU binding: 42:31.211 (m:ss.mmm)
```

Kódrészlet 6.1. JSAN lefutási idő előtte és utána

Ugyanazt az outputot adja mind a kettő program, szóval csak gyorsaságban és memóriahasználatban változott sokat.

Emellett a JavaScriptSchema újraírása is sikeres volt, emiatt a JSAN tud typescriptes kódokat elemezni. JSAN2Limet sikeresen módosítottam, hogy a JSAN által kiadott outputot sikeresen limmé alakítsa.

6.2. Jövőbeli tervek

JavaScriptSchema bővítése sok időbe telt, mivel se dokumentáció, se tapasztalat nem volt. Ezután még a JSAN2Lim átírás is sok időbe került. Eközben a typescript-eslint github repo amit használtunk a bővítésre, frissült, sok új funkciót hoztak be. Rengeteg változtatás volt a typescript résznél, de a schémát nem tudtuk még naprakészre hozni, mivel voltak ennél fontosabb feladatok. Egyik jövőbeli terv az, hogy a schémát up to datere hozzam, mivel már van hozzá dokumentáció is, meg nagyjából én is írtam, ezért nem lesz ez annyi idő, mint volt az elején az újraírása.

Végül a JSAN programra még bőven ráfér az optimalizálás, mivel csak az ast bindert optimalizáltuk, rengeteg helyen feleslegesen van bejárva az ast. Ez a másik jövőbeli terv.

Nyilatkozat

Alulírott Pozsgai Alex programtervező informatikus BSc szakos hallgató, kijelentem, hogy a dolgozatomat a Szegedi Tudományegyetem, Informatikai Intézet Szoftverfejlesztés Tanszékén készítettem, programtervező informatikus BSc diploma megszerzése érdekében.

Kijelentem, hogy a dolgozatot más szakon korábban nem védtem meg, saját munkám eredménye, és csak a hivatkozott forrásokat (szakirodalom, eszközök, stb.) használtam fel.

Tudomásul veszem, hogy szakdolgozatomat a Szegedi Tudományegyetem Informatikai Intézet könyvtárában, a helyben olvasható könyvek között helyezik el.

Szeged, 2023. május 9. .....
aláírás

Irodalomjegyzék

[1] Arpád Beszédes, Rudolf Ferenc, and Tibor Gyimóthy. Columbus: A reverse engineering approach. *STEP 2005*, page 93, 2005.

[2] Rudolf Ferenc, László Langó, István Siket, Tibor Gyimóthy, and Tibor Bakota. Source meter sonar qube plug-in. In *2014 IEEE 14th International Working Conference on Source Code Analysis and Manipulation*, pages 77–82. IEEE, 2014.

[3] Ferenc Rudolf. Sourcemeter for c/c++ with open-source sonarqube plugin released. 2015.

[4] István Siket, Árpád Beszédes, and John Taylor. Differences in the definition and calculation of the loc metric in free tools. *Dept. Softw. Eng., Univ. Szeged, Szeged, Hungary, Tech. Rep. TR-2014-001*, 2014.

[5] Gábor Szőke, Csaba Nagy, Rudolf Ferenc, and Tibor Gyimóthy. A case study of refactoring large-scale industrial systems to efficiently improve source code quality. In *Computational Science and Its Applications–ICCSA 2014: 14th International Conference, Guimarães, Portugal, June 30–July 3, 2014, Proceedings, Part V 14*, pages 524–540. Springer, 2014.

[6] Typescript-eslint github repository. `https://github.com/typescript-eslint/typescript-eslint/tree/main/packages/ast-spec/src`.