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 JavaScript-nek 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 az eszközcsalád optimalizálása.

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 elemezni TypeScript fájlokat és projekteket. 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 JavaScript elemző (JavaSript Analyzer, továbbiakban: 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. Ebből adódik az, hogy gyökerestül át kellett írni az eszközt, 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 néhány 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, TypeScript nyelvi elemek [6, 5, 9, 4] és a TypeScript séma [13] ismeretét igényelte meg. Emiatt a fejlesztés

előtt tanulmányozni kellett ezeket.

A JSAN2Lim fejlesztése után a JavaScript elemzőnek (JavaScript Analyzer, továbbiakban: JSAN) a binder függvényét kellett optimalizálnom. Először megérteni kellett a kódot, át kellett nézni az Abstract Syntax Tree-t (továbbiakban: AST), 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) [12] nyelvű összetett programok forráskódján. A FrontEndART a Szegedi Tudományegyetem Szoftverfejlesztés Tanszékén kutatott és fejlesztett Columbus technológián [3] 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 [11], 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 [10] 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 [2] eszközben van használva.

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. [8]

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.

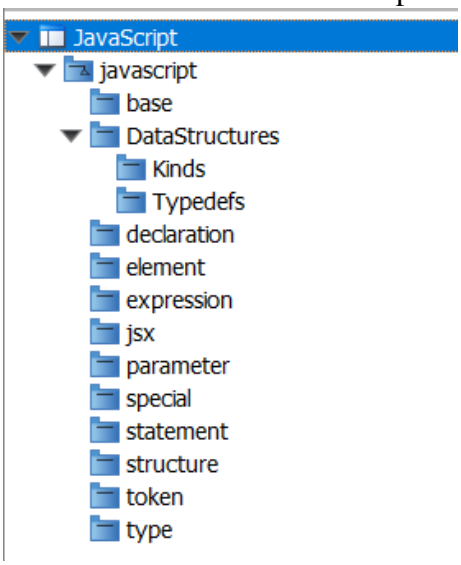
A SourceMeter for JavaScript projekt több alprojektet is magába foglal, amelyek különböző részfeladatokra specializálódnak.

2.2.1. Nyelvi séma bevezetés

A JavaScript nyelvi séma(továbbiakban: séma) [7] 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

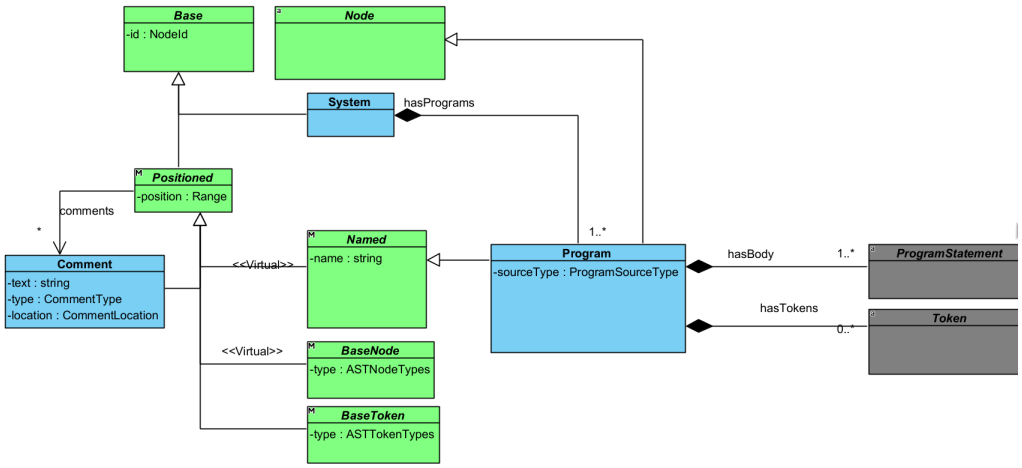
ebből a vpp fájlból létrehozza a C++ alap fájlokat, amiket az elemző programok tudnak használni. A felépítése a következőképpen néz ki:

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 [13] alapján lettek elnevezve. Átláthatóság és könnyebb bővíthetőség szempontjából lett létrehozva több package. Követtük a TypeScript-eslint [13] 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 osztály-

ból 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 Declaration-Statement, 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 Comment-Location 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 hasPrograms-nak a JavaScriptAddon-ban van jelentősége. A ProgramSourceType is a DataStructures package-ben lett definiálva. A ProgramSourceType értéke lehet source, vagy module. A Base package eltér a TypeScript-eslint [13] 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.

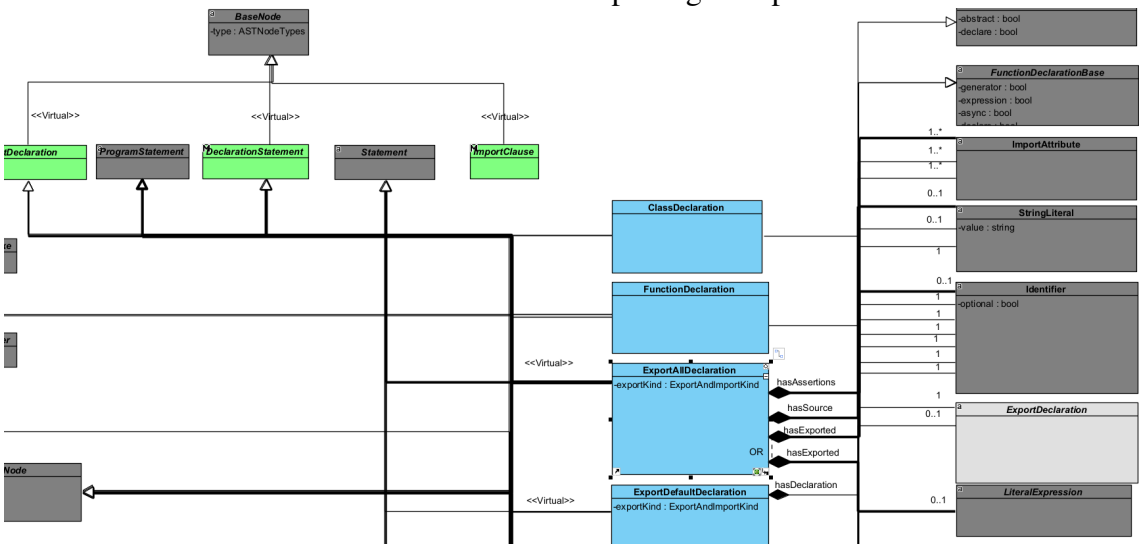
```
1 export interface ExportAllDeclaration extends BaseNode {
2       type: AST_NODE_TYPES.ExportAllDeclaration;
3       assertions: ImportAttribute[];
4       exported: Identifier | null;
5       exportKind: ExportKind;
6       source: StringLiteral;
7 }
```

Kódrészlet 2.1. ExportAllDeclaration TypeScript megvalósítása

A 2.1 kódrészlet megvalósítása a sémában:

Az ExportAllDeclaration osztály öröklődik a DeclarationStatement osztályból, ami öröklődik a BaseNode osztályból. Ezért az ExportAllDeclaration egyaránt öröklődik a

2.3. ábra. A Declaration package felépítése

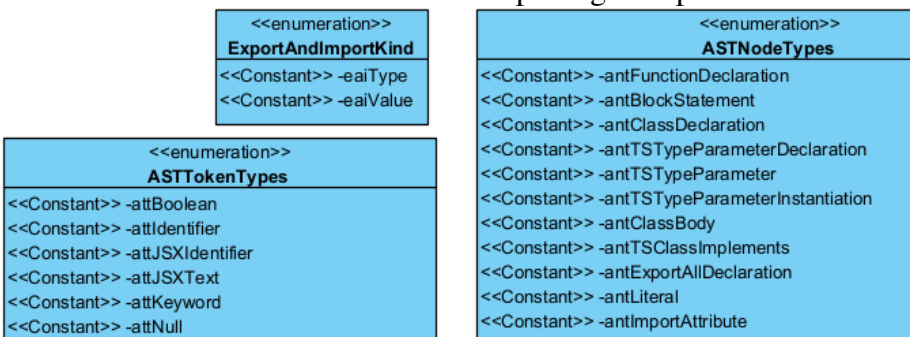


BaseNode osztályból. A 2.1 kódrészletben ez az extends BaseNode-al van jelezve. A főbb osztályok a BaseNode alatt találhatóak az ábrán. Az ábra jobb oldalán lévő osztályok, amik sötét- és világosszürkével vannak jelölve, az attribútumok. Az ExportAllDeclaration osztály öröklődik a Statement, DeclarationStatement, Node és a ProgramStatement osztályokból. Az összes öröklődés a TypeScript-eslint [13] 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 JavaScriptAddon-nál fog mutatkozni.

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

A DataStructures package-ben található 2 package, Kinds és Typedefs. A 2.1 ábrán

2.4. ábra. A DataStructures package felépítése



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:

```
1 KIND ASTNodeTypes (ant) {
2       FunctionDeclaration;
3       BlockStatement;
```

```
4       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.

```
1  SCOPE declaration {
2       NODE DeclarationStatement : virtual base::BaseNode [ABSTRACT] {
3       }
4       NODE ExportAllDeclaration : DeclarationStatement, statement::
           Statement, virtual statement::ProgramStatement, special::Node
            {
5             ATTR ExportAndImportKind exportKind;
6             EDGE TREE 1 hasExported (expression::Identifier |
                 expression::LiteralExpression);
7             EDGE TREE 1 hasSource (structure::StringLiteral);
8             EDGE TREE * hasAssertions (special::ImportAttribute);
9       }
10 }
```

Kódrészlet 2.4. Asg fájl ExportAllDeclaration

Az UmlToAsg minden egyes package-et Scope-nak értelmez. A Scope-on belül az osztályokat gráf csomópontnak (továbbiakban: node) 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

A JavaScriptAddon a SchemaGenerator által generált fájl. A 2.4 kódrészleten látható, hogy node-okat generált az UmlToAsg az osztályokból. Ezeknek a node-oknak a Schema-

Generator generál külön egy WrapperHeader és egy WrapperCC fájlt. A SchemaGenerator továbbá generál Factory.cc és a hozzátartozó header fájlt. A Factory.cc-t bővebben taglalom a 2.11 kódrészletnél. A WrapperCC, Factory.cc és a hozzájuk tartozó header fájlok összeolvasztása eredményezi a JavaScriptAddon fájlt. A JavaScriptAddon fájl generálás menetét a NodeAddonGenerator fájlban írjuk le. A SchemaGenerator main.c fájlba beimportáljuk a NodeAddonGenerator összes metódusát, aztán ezeket a metódusokat használva legeneráljuk a JavaScriptAddon fájlt. A $-genNodeAddon$ kapcsolóval fogja legenerálni a JavaScriptAddon fájl a SchemaGenerator. A SchemaGenerator main.c fájl elején először vizsgál a megadott kapcsolókra az alábbi módon:

```
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 fog semmi se történni.

```
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 először létrehoz egy addon nevű mappát és ch-

dir parancsot használva belelép. Ha ez a mappa már létezik, akkor a létrehozást kihagyja és a chdir paranccsal belelép. Létrehozás és chdir parancs használat után az addon mappába generál package.json fájlt. Ebben a fájlban 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 node-gyp build segítségével fog történni a buildelés. A node-gyp egy eszköz, amely összeállítja a Node.js Addonokat. A Node.js Addonok natív Node.js modulok, amelyeket C vagy C++ nyelven írnak. Miután az ilyen modulokat eszközökkel, például a node-gyppel lefordították, funkcióik elérhetők lesznek a $require()$ segítségével, ahogy bármely más Node.js modul esetében.

A package.json fájl után generálja le a binding.gyp fájlt. A binding.gyp fájlban találhatóak meg az összes node fájl relatív útvonala a package.json fájlhoz képest. A binding.gyp fájl után generálja le az addonCC fájlt.

,

```
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 traversalDescendantBFT metódus egy bejárás, ami az összes node-ra lefut. Ezekre a node-okra meghívja a generateWrapIncludes metódust. A generateWrapIncludes metódus a NodeAddonGenerator 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

Ha a jelenlegi node absztrakt, akkor nem ír semmit a fáljba. Ha nem absztrakt, akkor a 2.9 kódrészlet alapján importálja a node-nak a header fájl relatív útvonalát. A `node->name` több értéket is vehet fel, például FunctionDeclaration, ClassDeclaration, ezek megtalálhatóak az asg fájlban. Ezután ez a bejárás még egy alkalommal végrehajtódik. Ebben az esetben a wrapperIniteket ír az addonCC fájlba.

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

Kódrészlet 2.10. generateWrapInit leimplementálása

Az addonCC fájl generálása után a generateFactoryWrapper függvényhívás történik. Ebben a függvényben 2 metódus függvényhívás található, a generateFactoryWrapperHeader és generateFactoryWrapperSource.

A generateFactoryWrapperHeader a Factory.h fájlt generálja le. Először importálja az összes nodeWrapper header fájlt, 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 metódusai.

A generateFactoryWrapperSource a Factory.cc fájlt generálja le. DECLARE_NAPI_METHOD-okat hoz létre, az összes nodeWrapper fájlnak.

```
1 napi_property_descriptor props [] = {
2       DECLARE_NAPI_METHOD("getRoot", getRoot),
3       DECLARE_NAPI_METHOD("createCommentWrapper", createCommentWrapper)
4       ...}
```

Kódrészlet 2.11. Factory.cc fájl

Legvégül, a SchemaGenerator legenerálja az inc és az src mappákat. Az inc mappában találhatóak a header fájlok minden egyes nodeWrapper-nek, az src mappában maguk a nodeWrapper-ek vannak megvalósítva.

A SchemaGenerator legenerálj az ExportAllDeclarationWrapper.h és az ExportAllDeclaration.cc fájlokat. A header fájlban létrehozza az ExportAllDeclarationWrapper

osztályt, ami öröklődik a BaseWrapperből. A BaseWrapper alap Wrapper osztály. Három publikos metódust generál le, az Init-et, a Destructor-t és a NewInstance-t. Ezen felül több private metódust is.

```
napi_property_descriptor props [] = {
      static napi_value setPosition(...);
      static napi_value addAssertions(...);
      static napi_value setType(...);
...}
```

Kódrészlet 2.12. ExportAllDeclarationWrapper.h fájl

A 2.12 kódrészleten megfigyelhető, hogy létrehozott az ExportAllDeclarationWrapper-nek egy setPosition, addAssertions, setType metódusokat. Ezek az ExportAllDeclaration attribútumai, ami a sémában be lett állítva. A 2.3 ábrán látható, hogy az ExportAllDeclaration hasAssertions attribútumának a multiplicitása 1..*, ezt átkonvertálta addAssertions-re. Ahol 0..* a multiplicitás, azt átkonvertálta setAttribute-ra, például setSource vagy setExported-re. Minden nodeWrapper rendelezik setPath, setPosition, setType és addComments metódusokkal.

Az ExportAllDeclaration.cc fájlt ugyanúgy generálja le, mint a Factory.cc fájlt. DECLARE_NAPI_METHOD-okat generál, a setExported, setSource, setType és a többi metódushoz. Ezután az összes metódust megvalósításra kerül.

2.2.3. JSAN2Lim bevezetés

A JSAN2Lim C++ programozási nyelvben megírt program. A JSAN általi eredményt alakítja át nyelvfüggetlen (továbbiakban: LIM) formátumú fájlokra. Azért LIM formátumú fájlokra alakít át, mivel egy nyelvfüggetlen modellen sokkal könnyebb metrikákat mérni. A JSAN2Lim függ a JSAN programtól, hiszen a JSAN eredményeit alakítja át, tehát, ha a JSAN programot változás éri, akkor a JSAN2Lim programot is változtatni kell, különben nem lesznek pontosak a metrika mérések. A JSAN2Lim használja a LimSchema-t. A LimSchema hasonlóan van felépítve, mint a JavaScript nyelvi sémája. A JSAN2Lim bejárja az összes node-ot ami megtalálható a JSAN általi eredmény fájlban. Minden node-ra külön meg kell írni a bejáró (továbbiakban: visit) függvényt.

A JSAN2Lim működését egy kódrészleten keresztül mutatom be.

```
1  VISIT_BEGIN(clNode, callVirtualBase);
2  lim::asg::logical::Class& classLimNode = dynamic_cast<lim::asg::logical
       ::Class&>(createLimNode(clNode, callVirtualBase));
3  fillData(classLimNode, clNode);//sets the name
4  fillMemberData(classLimNode);
5  demangledNameParts.push_back(classLimNode.getName());
6  classStack.push(ClassInfo());
7  classStack.top().classNodeId = classLimNode.getId();
8  if (clNode.getSuperClass() != NULL) {
9        if (javascript::asg::Common::getIsIdentifier(*clNode.
       getSuperClass())) {
10             javascript::asg::expression::Identifier& superClass =
       dynamic_cast<javascript::asg::expression::Identifier&>(*clNode.
       getSuperClass());
11             classStack.top().isSubclass = superClass.getId();}}
12 if (!packageStack.empty()) {
13       lim::asg::logical::Package& packageLimNode = dynamic_cast<lim::
       asg::logical::Package&>(limFactory.getRef(packageStack.top().
       packageNodeId));
14       SAFE_EDGE(packageLimNode, Member, lim::asg::logical::Member,
       classLimNode.getId());}
15 addIsContainedInEdge(classLimNode, clNode);
```

Kódrészlet 2.13. ClassDeclaration Visitor

A VISIT_BEGIN a ViscitorAbstractNodes programnak a visit metódusa. Ezután létrejön a classLimNode változó, ami `lim::asg::logical::Class&` típusú. Ez a típus a LimSchema-ban található meg, a logical package-ben. A változónak a createLimNode metódus által visszaadott node lesz az értéke. Ezt az értéket konvertáljuk át a erre a `lim::asg::logical::Class&` típusra.

A createLimNode függvény meghívja a createNode függvényt. Ez a függvény vár paraméterbe egy nodeKind-ot. A nodeKind az adott node típúsát jelöli. A nodeKind-ot a getLimKind függvény határozza meg. A getLimKind függvény megvizsgálja, hogy az adott node `getIsClassDeclarationBase` vagy `getIsMethodDefinition` vagy `getIsFunctionDeclarationBase`. A nodeKind értéke típustól függően adódik meg. Ezek az értékek a következők lehetnek: ndkClass, ndkMethod, ndkParame-

ter, ndkMethodCall, ndkAttribute, ndkComment, ndkPackage, ndkAttributeAccess, ndkComponent. Ezek az értékek mind a LimSchema-ban vannak deklarálva. A nodeKind értékadása után a függvény visszaadja ezt a változót és tovább használja a visitor-ban. Ezután, a createLimNode függvény létrehozza a megfelelő node-ot a nodeKind-tól függően. Beállítja a pozícióját a node-nak és visszaadja a *limNode-ot.

Ebben az esetben a nodeKind az ndkClass értéket kapja meg. A classLimNode értékadás után meghívódik a filldata függvény. Kettő paramétert vár, classLimNode-ot, és a node-ot. A filldata a JSAN általi adott eredményet bejárja, visitorok segítségével. A filldata-ban beállítódik a limnode láthatósága, nyelve, az hogy absztrakt-e és a classKindja, ami jelen esetben clkClass értéket fog felvenni.

Ezután beállítódik a limnode neve. Ha a node-nak volt identifier-e, akkor a `getName` függvénnyel beállítódik a limnode neve különben `anonymousClass` érték állítódik be. Ezután megvizsgálja, hogy a methodstack (ami egy stack és metódusokat tartalma) üres-e. Ha nem üres, akkor létezik szülője, ezért a SAFE_EDGE metódussal hozzáadódik egy csúcs a szülő és a jelenlegi limnode között. Ezt uganúgy megvizsgálja a packageStack-re (ami egy stack és csomagokat tartalmaz), ha nem üres, akkor ugyanezt elvégzem csak a szülő más lesz. Legvégül az `addIsContainedInEdge` függvény meghívódik a limnode és a jsNode paraméterekkel.

Ezután a `fillMemberData` függvény hívódik meg. Ez a függvény a limnode pozícióját állítja be, és a nevén változtat. Utána a classStack-hez hozzáadódik az adott osztály tulajdonságai, beállítja a szülő osztályt ha van, a pozícióját is beállítja, és a végén meghívódik az `addIsContainedInEdge` a classLimNode és a clNode paraméterekkel. Minden egyes node-nál ez az eljárás fut le.

2.2.4. Regressziós tesztelés bevezetés

A regressziós tesztelés általában egy olyan szoftvertesztelési típus, amely akkor használatos, amikor megerősítjük, hogy a legfrissebb program- vagy kódmódosítások nem érintették negatív módon a meglévő funkciókat. A regressziós tesztelés során megállapítják, hogy a szoftver vagy alkalmazás megfelelően működik-e az új változások és hibajavítások tekintetében. A regressziós tesztelés főként arról szól, hogy az már végrehajtott teszteseteket újra lefuttatjuk annak érdekében, hogy megerősítsük, az alkalmazás

megfelelően működik-e.

A regressziós tesztelésnek 7 különböző típusa van:

Re-test All (Mindent újratesztelni): Egy módja a regressziós tesztelésnek az, hogy az összes tesztesetet újra futtatjuk. Ez valójában egy költséges folyamat, mivel több időt, erőforrást és erőfeszítést igényel az egész teszteset újbóli végrehajtása. Az ebben a folyamatban észlelt hibákat előre jelentik a javítás érdekében.

Corrective Regression Testing (Javító regressziós tesztelés): A javító regressziós tesztelés a regressziós tesztelés egyszerűbb formái közé tartozik, minimális erőfeszítést igényel. A javító regressziós tesztelés nem igényel változásokat a meglévő kódbázisban és az alkalmazásba új funkciók hozzáadását. Egyszerűen csak a már meglévő funkciókat és azok teszteseteit kell tesztelni, és nem kell újakat létrehozni.

Unit Regression Testing (Egység regressziós tesztelés): Az egység regressziós tesztelés a regressziós tesztelés elengedhetetlen része, amelyben a kódot elszigetelve teszteljük. Az integrációkat és függőségeket kikapcsoljuk az egység regressziós tesztelés során, és az egyes egységekre helyezzük a hangsúlyt. Általában alacsony forgalmú és csúcsidőn kívül végzik ezt a tesztelést.

Selective Regression Testing (Szelektív regressziós tesztelés): A szelektív regressziós tesztelés elemzi a meglévő kódot és mind az új, mind a meglévő kód hatását. A közös elemeket, mint a változókat és funkciókat, az alkalmazásba beillesztik, hogy gyors eredményeket kapjanak, anélkül, hogy befolyásolnák a folyamatot.

Progressive Regression Testing (Progresszív regressziós tesztelés): A progresszív regressziós tesztesetek az elvárások alapján készülnek. Amikor csak kisebb termékfejlesztések vannak, az új teszteseteket úgy tervezik, hogy ne befolyásolják a termék meglévő kódját.

Complete Regression Testing (Teljes regressziós tesztelés): Néhány kisebb vagy jelentős változtatásnak hatalmas hatása lehet a termékre. A teljes regressziós tesztelést ebben az esetben használják, amikor jelentős módosítások vannak a jelenlegi kódban. Segít a tesztelési folyamat során történt bármilyen módosítás javításában.

Partial Regression Testing (Részleges regressziós tesztelés): Amikor új kód kerül hozzáadásra egy meglévő kód bázishoz, akkor a részleges regressziós tesztelést alkalmazzák. Ez segít felfedezni a kritikus hibákat a meglévő kódban, és lehetővé teszi a tesztelést

anélkül, hogy a rendszer működését befolyásolná.

3. fejezet

Oszthatlan közösen készített programok

3.1. A nyelvi séma átírása

Az eddigi JavaScript nyelvi séma (továbbiakban: séma), ami csak javascript nyelvű projekteket és fájlokat elemzett, a javascript hivatalos oldala [1] alapján készült. A 2.2.1 fejezetben ami ábrák láthatóak, azok már az átírt sémából lettek kifotózva.

A séma átírásánál arra a döntésre jutottunk, hogy elölről kezdjük az egészet. Átláthatatlan volt az előző séma, és nem volt hozzá megfelelő dokumentáció, ezért a bővítés nehezebbnek bizonyult, mint az újraírás a kezdetekről. Továbbá, egy séma ami képes TypeScript fájlokat és projekteket elemezni, az képes JavaScript fájlokat és projekteket is. A Base package-et emeltük át az előző sémából, BaseNode-al és BaseToken-el kibővítve. Ez látható a 2.2 ábrán is. A TypeScript-eslint github [13] alapján hoztuk létre a struktúrális felépítést, annyi változtatással, hogy mi hoztunk létre még egy structure nevű package-et, amiben azok az osztályok találhatóak meg, amelyek a TypeScript-eslint github base mappájában voltak. Fejlesztés során dokumentáltuk a lépések nagy részét.

Minden egymásra épül a sémában, emiatt a működést nem tudtuk az elején letesztelni, csak miután több package is készen lett. Megfigyelhető a 2.4 kódrészleten, hogy az ExportAllDeclaration több mindenből öröklődik és sok attribútuma van. Az ExportAllDeclaration egyik attribútumának a típusa az ImportAttribute. Ehhez az ImportAttribute osztályt is le kellett fejleszteni a sémában.

```
1 export interface ImportAttribute extends BaseNode {
2       type: AST_NODE_TYPES.ImportAttribute;
```

```
3       key: Identifier | Literal;
4       value: Literal;
5   }
```

Kódrészlet 3.1. ImportAttribute

Az ImportAttributet lefejlesztéséhez le kell fejleszteni az Identifiert, és a Literalt is. Ezek lefejlesztéséhez is kellett más-más osztályokat lefejleszteni.

3.2. Nehézségek, problémák

A séma újból írása során több nehézségbe ütköztünk. Dokumentáció hiánya miatt nem tudtuk az előző sémának minden részét jól értelmezni, néhány félreértés volt fejlesztés közben. Fejlesztés végén, miután az összes package és osztály le volt fejlesztve, próbáltuk tesztelni. Tesztelés során Segmentation Fault hibát kapott a program, és nem tudtunk rájönni, hogy miért történik ez. Emiatt az egész sémát átnéztük, biztosra mentünk, hogy az osztályok öröklődési jól voltak a sémában lefejlesztve, illetve az összes attribútumot is ellenőriztük. Több kisebb figyelmetlenséget is észrevettünk, és javítottuk. Több javítás után, újra próbáltunk tesztelni. Ebben az esetben egy másik, jobban beazonosítható hibát kaptunk futtatás során. Már nem kaptunk Segmentation Fault hibát, hanem futás közben egy node-nál minden hibaüzenet nélkül leállt a program. Kiderítettük, hogy melyik node-nál áll le a program mindig, ez volt maga a Literal node. Ismét átnéztük a LiteralExpression osztályt, mivel ez maga a Literal, és egy rossz osztályból öröklődött le, ez okozta a program leállását. Javítottuk a problémát, és ezek után hibamentesen és jól letudtuk futtatni.

Továbbá sok nyelvi elemet a TypeScriptben nem ismertünk. Tanulmányozni kellett jobban a TypeScript programozási nyelvet, hogy tesztek átnézése során tudjuk mit elemzett le jól és mit nem.

4. fejezet

Általam változtatott programok

4.1. JavaScriptAddon változások

A séma változtatás után, magát a NodeAddonGenerator fájlt nem módosítottam. Ennek ellenére a JavaScriptAddon mérete lényegesen megnőtt, mivel a séma nagy bővítésen esett át. A JavaScript-es nyelvi elemek mellett a TypeScript-es nyelvi elemek is megtalálhatóak benne. Emiatt sokkal több node, és hozzájuk a wrapper-ek generálódtak le. A JavaScript-es nyelvi elemek megmaradtak, néhány helyen még javítottunk hibákat is.

4.2. JSAN2Lim Bővítések

A JSAN tud már TypeScript fájlokat és projekteket is elemezni, emiatt a JSAN2Lim-et is fejleszteni kellett. A JSAN2Lim használja a JavaScript és a Lim asg fájlokat egyaránt.

Több helyen is át kellett írni a JSAN2Lim-et:

- Több visitor-nál a várt paraméter típusát átírni, például Class-ról ClassDeclarationBase-re, mivel a sémában már nem Class-ként, hanem ClassDeclarationBase-ként szerepel.
- Több visitor-nál a várt paraméternél helyét meg kellett változtatni egyes node-nál. Például a RestElement eddig a Statement package-en belül helyezkedett el, de a bővítés során átkerült a Parameter package-be.

- A Pattern, mint node, Parameter-re lett átnevezve. A helye is megváltozott, Statement package-ből átkerült a Parameter package-be.
- Típus lekérdezésnél a `getIsClass` helyett `getIsClassDeclarationBase` kellett használni.

A bővítések, hogy mikkel bővítettem a JSAN2Lim-et:

- Több kind-ot is létrehoztunk a sémában, ezekkel bővíteni kellett a JSAN2Lim-et.
- Új node visitor-ok írása. Mint például a TSEnumDeclaration, TSImportEqualsDeclaration, TSInterFaceDeclaration, és még TypeScript node amihez kell visitor. A visitor-hoz a fillData függvényeket is megírni.
- JSAN2Lim-ben a kindStrings tömb kiegészítése TypeScript-es node-okkal.
- Hibák kijavítása, a VariableDeclaration-nél nem detektálta az összes változó deklarálást, főleg metódusokon belül.
- A `getLimKind` metódus bővítése `getIsTSTypeAliasDeclaration`, `getIsTSInterfaceDeclaration`, `getIsTSAbstractMethodDefinition` és `getIsTSAbstractPropertyDefinition` esetekkel.
- TSEmptyBodyFunctionExpression osztály hibajavítása, hiszen több helyen is leállt a program emiatt.

4.3. Ast binder optimalizálás

Az AST binder referencia kötésekre (továbbiakban: bindolásra) alkalmas. Ezek a referenciák a JSAN2Lim programhoz szükségesek. Binder néven van definiálva a metódus a JSAN-ban. Kétszer van használva, ezért is kulcsfontosságú, hogy gyors legyen. Egyszer VariableUsages referenciákat bindol és egyszer ACG referenciákat bindol.

A binder 4 argumentumot vár:

TODO: ACG megnézése, hogy mit rövidít

- Egy stringet, ami lehet vagy VariableUsages (továbbiakban: VU) vagy ACG (egy javascript callgraph).

- Abstract Syntax Tree-t (továbbiakban: 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 JavaScriptAddon-ban található meg és onnan hívódik meg. Abban az esetben kap addCalls értéket, ha ACG referenciákat bindol a binder, és akkor setRefersTo, ha VU referenciákat.

A linkeket tartalmó tömb egy JSON objektuma a következőképpen 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 4.1. Binder JSON objektuma

Ilyen JSON elemekből épül fel a tömb. A source és a target objektum felépítése ugyanaz, csak másak az értékek. A kódrészletben a source van kifejtve bővebben. A `label` az egy Identifier vagy PrivateIdentifier node nevét fogja jelölni. A `file` egy abszolút útvonalat kap, ami az Identifiert tartalmazó fájlt jelöli. A `start` az egy objektum, 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 objektum, 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 objektum ami maga az Identifier vagy PrivateIdentifier.

4.3.1. Lassúság okai

Binder-nek a futásideje lényegesen megnő, ha nagyonn projektekre futtatjuk le. Ennek több oka is van:

- Minden node hívásnál meghívja a JavaScriptAddon-ból egy metódust. A JavaScriptAddon már magában nagy terjedelmű fájl. TypeScript támogatás után kétszeresére nőtt a mérete, mint ezelőtt, emiatt lassult is.
- Nagyobb projektekben több függvényhívás és változószám található az elemzett kódban.
- Az AST nagyobb projekteknél nagyon nagy is lehet. Ezt az AST-t bejárjuk többször is bindolás alatt. Emiatt lesz nagyon lassú a binder.
- A JSON objektumokat tartalmazó tömb is nagy méretű lesz, de ez kevésbé lassítja a bindert, mint az AST-s bejárás.

4.3.2. Optimalizálás

Optimalizálás során először a JavaScript Profiler-t használtam memória és futásidő vizsgálatra. Nekem ez nem felelt meg, mivel a JavaScriptAddon-ban rengeteg minden egymásra épül, és 50-100 mélységű függvényhívásnál nem tudta kiírni a futásidejét és a memórihasználatot. Ezután a binder metódust 3 részre bontottam `console.time` és `console.timeEnd` beépített függvénnyek segítségével. Futásidőt írta ki a függvény a `console.time` és a `console.timeEnd` sorok között milliszekundumban. Kiderült, hogy a következő sor felettébb nagy futási idővel rendelkezik:

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

Kódrészlet 4.2. Lassú metódus

A resolveNode metódus első paraméterként vár egy AST-t, utána egy fájlnevet, kezdő- és végPozíciót, illetve egy igaz vagy hamis értéket.

Az AST-t forEach-el bejárja a program. Az AST egyes elemei az AST node-ok. A forEach-en belül az AST node-on walk metódus segítségével bejártuk, addig amíg nem találtuk meg a keresett node-ot. Rosszabb esetben az AST legvégén volt a keresett node, mivel már a végén voltunk a bindolásnak. Kisebb projektek esetén ez nem baj, mivel az AST mérete nem akkora, mint egy nagyobb projekt esetében.

Optimalizálás során, létrehoztam indexAST metódust, ami indexeli az AST-t.

```
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 4.3. indexAST metódus

A 4.3 kódrészletben látható, hogy egy AST-t várunk paraméterben. Ezt a bindertől fogja kapni. A metódusban forEach-el bejárjuk az AST elemeit, amik az AST node-ok. Beállítom a `globals.setActualFile()` függvénnyel az aktuális fájlnevet. Ezután az adott AST node-ot walk függvény segítségével bejárjuk. Csak az enter metódusát kellett szerkeszteni. Ezután a setIndexed metódus meghívódik. A setIndexed függvénynek megkapja a fájlnevet, a node range-nek a kezdő- és a végparaméterét, ahol kezdődik az adott node és hol végződik, karakterpontosan, és magát a node-ot.

```
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 4.4. setIndexed metódus

A setIndexed először vizsgál arra, hogy az adott node be van-e már indexelve. Ezt úgy teszi meg, hogy az indexedAST tömbben keres egy indexre. Ez az index a következőképp néz ki: FájlNév-KezdőPozíció-VégPozíció. Továbbá vizsgál arra is, hogy ha van ilyen indexű elem a tömbben, akkor ezen az indexen található-e már ilyen node. Ha van akkor nem állít be semmit, csak kilép, mivel már be van indexelve az adott node. Ha nincs, akkor beállítja az indexedAST tömbnek az adott indexre az adott node-ot. Az AST-t csak

egyszer járjuk be a foreach-el. A bejárás után minden node be lesz indexelve a tömbbe.

A 4.2 kódrészleten látható, hogy először a resolveNode függvény segítségével megkerestük a keresett node-ot kezdő- és végPozíció alapján. Ezután a megkapott node-ra meghívtuk a `getWrapperOfNode` függvényt, hogy megkapjuk a wrapperjét a JavaScriptAddon-ból. A keresést megváltoztattam, írtam rá egy getIndexed metódust.

TODO: Source vagy target Nodeot fordítsd le

A `getIndexed` függvény megvizsgálja, hogy source vagy target node-ot keresünk. Ha target node-ot, akkor visszaadjuk a `getWrapperOfNode(indexedAST[FájlNév-KezdőPozíció-VégPozíció])`-t. Ha source node-ot keresünk, akkor a walk függvény segítségével bejárjuk a beindexelt tömböt. Ez lényegesen gyorsabb, mint a resolveNode-os megoldás, mert ott minden egyes esetben az egész AST-t bejártuk a walk függvény segítségével, jelen esetben csak a beindexelt tömb elemét járjuk be. Ha megkaptuk a source node-ot akkor visszaadjuk azt a `getWrapperOfNode(result)` hívással.

4.3.3. Eredmények

TODO: Kisebb és nagyobb projekt összehasonlítása

```
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 4.5. 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.

5. fejezet

Regressziós tesztek frissítése

5.1. Tesztek kibővítése

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

- -i:Jelentése input, lehet relatív vagy abszolút útvonal a fáljoz vagy projekthez.
- -o:Jelentése output neve, lehet relatív vagy abszolút útvonal.
- -d:Jelentése dumpjsml, a JSAN eredményét átgenerálja XML stílusú fájlba és ezt egy jsml fájlba kiírja.
- -e:Jelentése ExternalHardFilter, relatív vagy abszolút útvonal egy olyan fájlhoz, ami szövegalapú és olyan szintaxis található benne, ami kell az externalHardFilternek.
- -help:Jelentése help, kiírja minden kapcsolóhoz tartozó leírást.
- -r:Jelentése useRelativePath, eredményben az útvonalakat átírja relatív útvonalra.
- -h:Jelentése html, a JSAN html fájlokra is lefut, bennük keresve JavaScript-es szkripteket és azokat teszteli.
- -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 JavaScript-es tesztek mellé kell majd TypeScript-es teszteket is keresnem. A projekteket python szkriptek 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_path-ban van a mappa is, és a js-externalHardFilter-ben megtalálható az external-HardFilter 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 kapcsolót annak függvényében, hogy igazat kaptunk e vagy sem. Létrehozzuk az externalHardFilter fájlt a tesztelendő fájl mellé vagy a tesztelendő projekt gyökerébe, és attól függően, hogy ki akarjuk hagyni az adott fájlt vagy hozzáadni, írunk egy $+$ vagy egy $-$ jelet a sor elejére és utána relatív útvonal és a fájl neve. Alapértelmezetten minden fájl elemez az adott program, JSAN esetében ezek azok a fájlok amiknek a kiterjesztése js, jsx. (külön kapcsolóval megadhatjuk neki, hogy a html kiterjesztésű fájlokat elemezze-e vagy se.) TypeScript-es bővítés után már a ts és a tsx kiterjesztésű fájlokat is elemzi. Ezért írtam több tesztesetet is. Egy példa, hogy

hogyan néz ki ez a fájl:

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

Kódrészlet 5.3. ExternalHardFilter fájl

Az 5.3 kódrészletben látható, hogy filtered01 02 és 03 at kivettük, hogy azokat ne elemezze a jsan. Ezután visszavettük a filtered01et. A filtered fájlok azok JavaScript-es fájlok, JavaScript-es kóddal. Ezután referenciába csak a filtered01 fájl eredményét raktam be, hiszen a 02 és 03at nem elemzi, ha elemezné, akkor szólna a program, hogy hiányzó referecia fájl. Természetesen lehet regurális kifejezést is használni filterezésnél.

Ezután teszteltem a `-i` kapcsolót, itt a programnak vissza kellene adnia, hogy üres az input ha nincs megadva. Ezután a `-o` kapcsolóra teszteltem, itt ebben az esetben alapértelmezett értékben `out.jssi` fájlt kellene visszaadnia. Végül a `-h` kapcsolót néztem meg, itt ha megvan adva ez a kapcsoló, akkor az adott projektben a html fájlokban a JavaScript-et 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 TypeScript támogatással és a JSAN2Limet átírtam, elkezdtem teszteket keresni, mind a JSAN-nak és mind JSAN2Lim-nek. Kerestem egyszerűbb TypeScript és picit összetettebb TypeScript projekteket is, hogy lássam az esetleges hiányosságokat. Természetesen ami eredményeket adtak a programok, azokat át kellett néznem egyesével, hiszen csak így tudom meg, hogy jól tesztelte-e a megadott fájlt vagy sem. Több hiányosságot is észrevettem, mind JSAN oldalról, mind JSAN2Lim oldalról, ezek kisebb hiányosságok voltak. Például, hogy egy node-nak 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 a JavaScript-es nyelvi séma tartalmazza a TypeScript-es nyelvi elemeket is. Továbbá a JSAN2Lim-et sikerült kibővíteni TypeScript támogatással. Mindeközben a regressziós teszteléseket sikeresen frissítettem és validáltam, amit később mások is jóvá hagytak. Legvégül sikerült az AST binder metódusát optimalizálnom, sokkal gyorsabban fut le, mint ezelőtt.

6.2. Jövőbeli tervek

Rengeteg változtatás volt a TypeScript résznél, de a sémát nem tudtuk még naprakészre hozni, mivel voltak ennél fontosabb feladatok. Mint például az AST binder optimalizálása, vagy a JSAN2Lim továbbfejlesztése. Ebből kifolyólag jelenleg a séma nem naprakész. Mivel a szakdolgozatom elkészítése során megfelelő jártasságra tettem szert, így a későbbi bővítést az eddigiektől gyorsabban el tudnám végezni.

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 10. .....

aláírás

Irodalomjegyzék

[1] The javascript language. `https://javascript.info/js`.

[2] Tibor Bakota, Péter Hegedűs, István Siket, Gergely Ladányi, and Rudolf Ferenc. Qualitygate sourceaudit: A tool for assessing the technical quality of software. In *2014 Software Evolution Week-IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE)*, pages 440–445. IEEE, 2014.

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

[4] Gavin Bierman, Martín Abadi, and Mads Torgersen. Understanding typescript. In Richard Jones, editor, *ECOOP 2014 – Object-Oriented Programming*, pages 257–281, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg.

[5] B. Cherny. *Programming TypeScript: Making Your JavaScript Applications Scale*. O'Reilly Media, 2019.

[6] Steve Fenton, Fenton, and Spearing. *Pro TypeScript*. Springer, 2014.

[7] Rudolf Ferenc, Árpád Beszédes, Mikko Tarkiainen, and Tibor Gyimóthy. Columbus-reverse engineering tool and schema for c++. In *International Conference on Software Maintenance, 2002. Proceedings.*, pages 172–181. IEEE, 2002.

[8] 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.

[9] C. Nance. *TypeScript Essentials*. Community experience distilled. Packt Publishing, 2014.

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

[11] 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.

[12] 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.

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