Szegedi Tudományegyetem
Informatikai Intézet

TypeScript támogatás megvalósítása ipari JavaScript elemző szoftverhez

Szakdolgozat

Készítette:

Baga Csaba

programtervező informatikus BSc szakos hallgató

Témavezető:

Dr. Antal Gábor

Tudományos munkatárs

Szeged
2023

Feladatkiírás

A hallgató feladata az SourceMeter statikus forráskód elemző eszközkészlet bővítése TypeScript nyelv támogatással, és egyéb, általános bővítések fejlesztése, melyekkel javít a program karbantarthatóságán. Mivel a bővintendő eszközkészlet programjai közös sémára épülnek, több program módosítása lesz szükséges különböző programozási nyelvekben. A megoldás opcionálisan bővíthető egyéb, témához illő vagy megvalósítást segítő ötletekkel.

Tartalmi összefoglaló

A téma megnevezése:

SourceMeter for Javascript forráskód elemző szoftvercsomag kiegészítése TypeScript nyelv támogatásával, egyéb tesztelési bővítésekkel.

A megadott feladat megfogalmazása:

A feladat folyamán implementálni kell a SourceMeter adott eszközeibe TypeScript elemzésre vonatkozó különböző bővítéseket. Emellett a tesztelési folyamat kiegészítése új tesztesetekkel, célpontokkal (targetekkel).

A megoldási mód:

Az eszközök által használt nyelvi séma bővítése és az érintett szoftverekben új funkciók, módosítások implementálása szükséges a feladat végrehajtására.

Alkalmazott eszközök, módszerek:

Fejlesztés során forráskód módosításra Visual Studio Code, forráskód fordításhoz Windows rendszereken Visual Studio 2017, Linux rendszereken Unix Makefiles volt használva. Séma módosítására Visual Paradigm volt szükséges. Tesztelések végbevitele lokális rendszeren a fordításhoz használt eszközök, központi szervergépen pedig Jenkins volt használva.

Elért eredmények:

A szoftvercsomag sikeresen tud elemezni TypeScript nyelven írt kódbázisokat. Tesztelés folyamata a SourceMeter különböző szoftvercsaládjainak egyes eszközeire lettek megszabva, tesztelési folyamat időigénye csökkent.

Kulcsszavak:

JavaScript, TypeScript, Statikus kódelemzés, AST, ASG, C, CMake, Python

Tartalomjegyzék

1. fejezet

Bevezetés

A szakdolgozatom célja egy forráskód elemző eszközkészlet, névlegesen a SourceMeter bővítése volt, melynek az SourceMeter for JavaScript alrészébe TypeScript [8][6][3][2] nyelv elemzésének implementálása volt a fő célja. Mivel ez a szoftvercsomag sok, kisebb programból áll össze, melyek egymásra épülnek, az új nyelv teljeskörű támogatásához több alprogramban is kellett változtatásokat létrehozni. Emellett szükséges volt arra is figyelni hogy a meglévő JavaScript támogatás is megmaradjon.

A SourceMeter eszközkészleten belül több szoftvercsomag található, melyek segítségével különböző nyelvek statikus elemzését lehet végezni. Ezek közül a csomagok közül a SourceMeter for JavaScript csomagnak adott programjain végeztem módosításokat, többek között a JavaScript elemzőn (JavaScript Analyzer, továbbiakban JSAN) és az ESLintRunner nevű programon.

Ezen bővítések mellett egyéb, működést, karbantarthatóságot és használhatóságot befolyásoló változtatást valósítottam meg, melyeknek köszönhetően az említett programok karbantarthatósága javult, csökkent az eszközkészlet teszteléséhez szükséges idő.

Az eszközkészlet bővítése egyszerre több ember által volt végezve. Mivel a módosított szoftvercsomag alprogramjai egymásra épülnek, fontos volt a csapatmunka és folyamatos, effektív kommunikálás hogy a fejlesztés sikeresen és hatékonyan haladhasson. Néhány helyen a munkafolyamatok során átfedések, és feloszthatatlan feladatokat volt szükséges elvégezni.

2. fejezet

SourceMeter szoftvercsalád

A SourceMeter eszközkészlet statikus forráskód elemző szoftvercsomag, mely különböző nyelvek elemzéséhez tartalmaz elemző és detektáló szoftvereket. Ezen eszközök segítségével a támogatott nyelvekben írt projektekben többek között kódolási szabálysértéseket mutat ki és különböző metrikákat számol ki egy egységes, nyelvfüggetlen módon.

A SourceMeter statikusan elemzi a bemenetként kapott projekteket. Statikus elemzés során a vizsgált kódbázis futtatás nélkül van vizsgálva szintaktikai és szemantikai szempontból. A szintaxis elemzés az alkalmazás forráskódjának felépítését és szintaxisát vizsgálja, hogy érvényes-e a programozási nyelv szabályai szerint. A szemantikus elemzés azonban a kód mögötti logikát vizsgálja, hogy az megfelel-e az adott feladatnak és az elvárásoknak.

Az eszközkészlet eredetileg C++ nyelv elemzésére volt alkalmas [7], de manapság számos más nyelv elemzésére is alkalmas. A különböző szoftvercsomagok többek között a C, C++, Java, RPG, és a JavaScript nyelvekben írt projektek kódbázisának mély elemzését támogatja. Kódbázis elemzés elvégzéséhez előre összeállított sémák alapján a bemenetként kapott projekt elemeiből többek között Absztrakt Szintaxis Fát (AST) állít elő, melynek feldolgozása során különböző metrikákat számol ki, például logikai kódsorok száma (LLOC, avagy Logical Lines Of Code), kód beágyazások mélysége (NL, avagy Nesting Level), vagy megjegyzések sűrűsége (CD, avagy Comment Density) [9]. Ezek mellett kódolási szabálysértések és duplikált kódsorok detektálására is alkalmas az eszközkészlet adott szoftverjei.

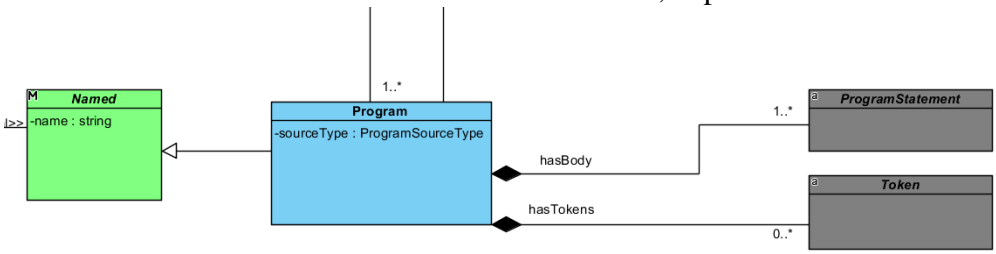
A SourceMeter eszközkészletet egy másik forráskód elemző eszközben, a Sonar

Qube-on belül is használhatjuk bővítményként. A SourceMeter és a SonarQube közötti integráció lehetővé teszi a fejlesztők számára, hogy a SourceMeter eredményeit a SonarQube-ba exportálják, melynek köszönhetően könnyen összehasonlíthatóak az elemzési eredmények. Az integráció lehetővé teszi a fejlesztők számára, hogy a SonarQube felületén keresztül megjelenítse és kezelje a SourceMeter eredményeit, és hogy integrálják az elemző eszközöket a fejlesztési folyamatukba.

A SonarQube-on kívül a QualityGate nevű szoftver is felhasználja a SourceMeter szoftvercsomagjait [1]. A QualityGate a SourceMeter által készített metrikákat és jelentéseket használja az alkalmazás minőségének és biztonságának értékeléséhez, és automatikusan jelzi a fejlesztőknek, ha az alkalmazás nem felel meg a meghatározott küszöbértékeknek. Ez segíti a fejlesztőket abban, hogy gyorsan és hatékonyan javítsák az alkalmazásukat, és biztosítják, hogy az megfelelő minőségű és biztonságos legyen.

Mindegyik szoftvercsomaghoz tartozik az elemzett nyelv alapján egy előre elkészített séma, mely egy Absztrakt Szemantikus Gráfot, másnéven ASG-t tartalmaz. Az elemzés lépései során ezek alapján végzik feladataikat a csomag programjai. Ezek lehetnek Absztrakt Szintaxis Fa, avagy AST felépítése, metrikák számolása, kódolási szabálysértések keresésére és ezek mindegyikének felhasználása duplikált kódszegmensek detektálására. Ezen sémák az elemzett nyelvnek nyelvtanának és szemantikájának megfelelő szerkezetét tartalmazzák, melyekben a csomópontok adott nyelvtani elemeket, az élek pedig az elemek közötti kapcsolatot reprezentálják. Egy példa erre a 2.1-es ábrán látható:

2.1. ábra. Sémában található elemek, kapcsolataik



A SourceMeter szoftvercsomagjai a mély elemzés különböző lépéseit több program használatával végzi el, melyek a folyamat során speciális feladatokat látnak el. A SourceMeter for JavaScript esetén az elemzés fő lépéseit 14 program végzi el, melyek mellett a szoftvercsomag előkészítéséhez fordítás során egyéb segédprogramok is fel vannak hasz-

nálva. Ezek közül szakdolgozatom feladata során az alább említett programokban lesznek bővítések és módosítások megvalósítva.

Az elemzés első lépésében a JSAN futtatásával az elemzendő projekt kódbázisából egy egyedi absztrakt szintaxisfát készít, mely többek között tartalmazza a projekt nyelvtani struktúráját és egyéb, függvényhívásokhoz tartozó információkat. Ezt későbbi lépések során metrikák számolására és duplikált kódszegmensek detektálására lehet felhasználni.

Az ESLintRunner program futtatásával a bemeneti kódbázisban különböző kódolási szabálysértéseket keres, melyek pontos helyét egy `.xml` formátumú fájlban összegzi. A detektált szabálysértéseket különálló szabályfájlok (későbbiekben `.rul.md` fájlok) használatával lehet megadni, melyek Markdown jelölőnyelven vannak megírva. Ezen fájl hiányában a program egy alapértelmezett, `.json` formátumban megadott szabályfájl használatával fogja az elemzést végrehajtani.

A JavaScript CallGraph (továbbiakban JSCG) az elemzett kódbázis alapján egy hívásgráfot (`callgraph`) állít elő. A hívásgráf (amelyet hívásmultigráfnak is neveznek) egy irányított gráf, amely megjeleníti a részprogramok közötti hívási kapcsolatokat egy számítógépes programban.

Minden csomópont egy eljárást képvisel, és minden él (f -> g) jelzi, hogy az f eljárás hívja a g eljárást. Ez a gráf a JSAN által generált kimenetbe van beágyazva.

A SchemaGenerator egy segédprogram, mely a szoftvercsalád fordítása során az előre meghatározott sémákból előállított ASG fájlok alapján generál futtatáshoz szükséges állományokat. A SourceMeter for JavaScript fordítása esetében a JSAN futtatásához szükséges `javascriptAddon.node` fájl generálását is végzi, mely segítségével az elemzett nyelvtani elemeket beágyazza „burkoló csomagpontba”, másnéven „wrapper”-ekbe. A nyelvtani elemeket és hozzájuk tartozó információkat a szoftvercsalád programjai ezeknek a wrappereknek segítségével tudják elérni és felhasználni. Ez a feladat során kibővül további fájlok generálásához szükséges funkciókkal.

3. fejezet

Előre tervezett változtatások

A TypeScript nyelv támogatásának megvalósításához a szoftvercsomag programjai közötti függőségek miatt több programban szükséges volt változtatásokat létrehozni. Működésbeli szempontok miatt külön séma létrehozása szükséges volt az új nyelvhez, mely tartalmazza visszamenőleg a JavaScript nyelv elemzéséhez szükséges elemeket. Az én feladatom része az ESLintRunner nevű programban az új nyelv támogatásának teljeskörű integrálása, JSAN és JSCG nevű programokban pedig meglévő támogatás bővítése.

Az ESLintRunner programon belül korábban csak JavaScript támogatás volt implementálva. Ennek bővítéséhez szükséges volt új „parser”, avagy fordító bevezetése és a hozzá tartozó új szabályok implementálása volt szükséges.

JSAN-on belül a nyelv elemzéséhez használt, különálló fájlok automatikus legenerálását valósítottam meg, melynek segítségével könnyebben karbantarthatóbbá vált az eszköz.

JavaScript-CallGraph, avagy JSCG-n belül korábban nem támogatott nyelvi elemek elemzéséhez szükséges funkciókat implementáltam, mely a JSAN elemzési képességén javított. Ezek mellett egyéb, teszteléssel kapcsolatos változtatásokat volt szükséges implementálni.

Az eddigi feladatok mellett az ESLintRunner új funkcionalitásának tesztelésére új regressziós tesztek írása is szükséges volt.

A feladat elvégzéséhez Windows rendszeren volt a fejlesztési folyamat végezve Visual Studio Code használatával. A programokban elvégzett változtatások helyes működésének tesztelése Windows és Linux rendszereken volt végezve helyi és központi szervergépen.

A fordítási folyamatok Windowson Visual Studio 2017 Build Tools használatával voltak végezve, míg Linuxon Unix Makefiles használatával.

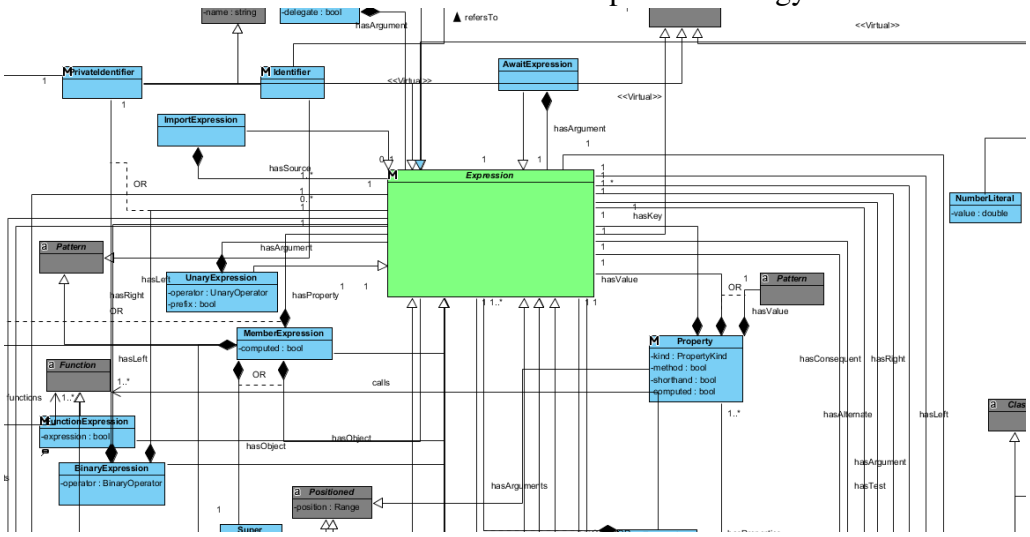
4. fejezet

Szoftvercsomag bővítése

4.1. Megoszthatatlan Feladatok

Mivel a SourceMeter eszközkészlet több nagyobb szoftvercsomagból áll, melyeknek csomagon belül, és néhány specifikus esetben csomagon kívül is adódnak függőségek a használt szoftverek között, néhány munkafolyamat oszthatatlannak minősült. Ezek a bővítési folyamatnak kezdeténél voltak jelentősek, mivel a SourceMeter által használt sémák nélkül nem lehet az elemzési folyamatot helyesen lefuttatni. A korábbi fejezetben említett sémák tartalmazzák az elemzéshez szükséges ASG felépítését, ennek módosítása létfontosságú volt a munkafolyamat folytatásához.

4.1. ábra. Korábban használt JavaScript Sémából egy részlet



Az új nyelv támogatásához a typescript-eslint által felvázolt AST [11] alapján hoztunk

létre egy új, szervezett formájú sémát. Bár a projektben szerepelt JavaScripthez tartozó séma, a kettő nyelv kompatibilitása ellenére számos eltérés volt jelen az eredeti, ESTree specifikáció [4] alapján készített és az új, TypeScriptet támogató séma között. Emellett naprakész dokumentáció nem állt rendelkezésre sem a meglévő sémához, sem teljesen új készítéséhez.

Annak érdekében hogy teljeskörű, lehetőleg hibamentes támogatást tudjunk létrehozni az új nyelvnek, egy teljesen új sémát hoztunk létre az új nyelv által használt AST alapján, miközben folyamatosan ellenőriztük a JavaScript kompatibilitást.

4.2. ESLintRunner bővítése TypeScript támogatáshoz

4.2.1. ESLintRunner bevezető

A SourceMeter for Javascript szoftvercsaládban a forráskód elemzés egyik első lépéseként az ESLintRunner nevű programot futtatja le az elemzendő projekten. Ez a program kimutatja, hogy milyen, előre meghatározott kódolási szabálysértéseket ejtettek a fejlesztés során, és listázza ezen hibák pontos helyét, mely fájlokban fordulnak elő, sor és karakter pontosságra. Ezeket egy `.xml` formátumú fájlba helyezi a program lefutás után, melyre a 4.1-es ábrában szerepel példa.

```
1 <?xml version='1.0' encoding='utf-8'?>
2 <ESLintToGraph version='4.3'>
3     <file name='C:\\path\\to\\file\\test.ts'>
4         <error line='4' column='9' severity='warning' ruleId='no-console' ↩
              nodeType='MemberExpression' message='Unexpected console statement↩
              .' />
5         <error line='4' column='21' severity='warning' ruleId='@typescript-↩
              eslint/restrict-plus-operands' nodeType='BinaryExpression' ↩
              message='Operands of  &apos; + &apos;  operation must either be ↩
              both strings or both numbers. Consider using a template literal.'↩
               />
6     </file>
7 </ESLintToGraph>
```

Kódrészlet 4.1. ESLintRunner eredmény fájl példa

Ez a program egy úgynevezett „parser”, avagy fordító segítségével tudja ezt a feladatát ellátni.

4.2.2. Megtett lépések az új nyelv támogatásához

Ahhoz, hogy TypeScript nyelv elemzésének a támogatását megvalósíthassam, ezen beállításokban megadtam a fordító opciónak (`parser` mező) az újonnan importált `typescript-eslint` [10] csomagnak a fordító modulját, majd a bővítmények közé (`plugin` mező) a csomag `eslint-plugin` modulját. A bővítmény megadása engedélyezi a programnak, hogy TypeScript nyelvhez kötődő, speciális szabálysértéseket detektálhasson és korábbi, JavaScriptes szabálysértéseket bővítsen ki új, opcionális információkkal.

Emellett helyes működés eléréséhez szükséges volt megadni az elemezhető fájlok kiterjesztését, melyet a felülbírálásoknál (`overrides` objektum) külön a `files` és a beágyazott `parser` mezőkben definiáltam. A mezők egyszerre tartalmazzák a JavaScript (`.js` és `.jsx`) és a TypeScript (`.ts` és `.tsx`) nyelvekhez tartozó kiterjesztéseket kompatibilitás megőrzése érdekében. Hiányában az alapértelmezett értékek töltődnek be futásidő során a fordítóhoz, mely nem kívánt működésben és eredményekben járulna. Az új fordító esetében azt jelentené, hogy csak TypeScript fájlok elemzése menne végbe. A 4.2-es ábrán láthatóak ezek a változtatások.

```
1  const typeScriptOptions = {
2  ...
3  overrides: [
4      settings: {
5          ...
6          import: {
7              parsers: {
8                  "@typescript-eslint/parser": ["*.ts", "*.tsx", "*.js", "*.jsx↩
                       "],
9      }}}
10 ],
11 plugins: [
12     "@typescript-eslint/eslint-plugin",
13     ...
14 ],
```

```
15 parser: "@typescript-eslint/parser",
16 ...
17 }
```

Kódrészlet 4.2. Elemző és bővítmény megadása

A beállításokhoz szükséges volt új mezők megadása is. A `parserOptions` mezőn belül szükséges futásidő során a `project` és a `tsConfigRootDir` beállítások megadása. A `project` a TypeScript nyelvben írt projektek konfigurációs fájljainak (névlegesen `tsconfig.json`) tartalmazza az útvonalait. Ezek alapján tudja az ESLint-Runner észlelni, hogy mely fájlokat szükséges elemezni a kódbázisból futásidő során. A `tsConfigRootDir` mező segítségével megadhatjuk, hogy az előbb beállított konfigurációs fájlok közül melyik tartozik az elemzett projekt gyökérkönyvtárába, ami szükséges adott speciális kódolási szabálysértések helyes működéséhez.

```
1 parserOptions: {
2     ecmaFeatures: {
3         jsx: false
4     },
5     tsconfigRootDir: path.dirname(globals.getOption("inputList")[0]),
6     project: [],
7     rules: ""
8 },
```

Kódrészlet 4.3. Szabálysértések megadása .json állományban

Ahhoz, hogy ezen beállítások futásidő során helyesen be lehessen állítani, elemzés előtt kell átvizsgálni a programmal az elemzendő projektnek a mappaszerkezetét. Mivel formailag kötelező, hofy TypeScript projekteknek a gyökérkönyvtárában szerepeljen egy `tscongig.json` fájl, így a `tsConfigRootDir` mező beállítása megegyezik az elemzett projekt útvonalával. Konfigurációs fájlokból viszont egyszerre több is megadható, emiatt a mappaszerkezetben található összes ilyen fájlt meg kell adnunk a `project` mezőbe. A fordító ahhoz, hogy a felhasználók számára megfelelő elemzést biztosítson, csak azokat a fájlokat elemzi alapesetben, melyek a projekt konfigurációs fájljainak az `include` mezőiben meg vannak adva. Emellett több kisebb hiba javítása is szükséges volt:

- Adott fájlok szűrését meg lehet határozni az `exclude` mezőkben. Továbbá figyelni kellett arra, hogy a konfigurációs fájlokat bővített formátumként kellett kezelni, azaz JSON5-ként. Ez konfiguráció olvasás folyamán megjelenő szintaktikai hibák javítása miatt volt szükséges.

- Mivel Linux és Windows operációs rendszereken is futtatható a SourceMeter, ezért adott útvonalakban az elválasztó jeleket általánosítani kellett. Linux rendszereken az útvonalakban a „\" jelek karakterfelszabadítást jelentenek, melyek miatt nem létező útvonalakat generálna futás közben a program.

- Ritka esetekben akadnak olyan hibák, hogy meglévő konfigurációs fájlok olyan további fájlokra mutatnak, melyek nem léteznek a mappaszerkezetben. Mivel az ilyen esetek hibás elemzési eredményeket generálnak, egy helyettesítő fájlt készítünk. Ez a fájl az elemzés futtatása után törlésre kerül.

4.2. ábra. Projekt konfiguráció beolvasás

```
for (let index in configPathsArray) {
    try {
        const options = JSON5.parse(fs.readFileSync(configPathsArray[index]).toString());
        //Checks if tsconfig has exclude parameter.
        if (options["exclude"] !== undefined) {
            for (let i = 0; i < options["exclude"].length; i++) {
                typeScriptOptions["baseConfig"]["ignorePatterns"].push(options["exclude"][i]);
            }
        }
        //In case the tsconfig wants to extend from a non-existent file, we create an empty one.
        if (options["extends"] !== undefined) {
            let extendedFile = path.join(path.dirname(configPathsArray[index]), options["extends"]);
            if (!fs.existsSync(extendedFile)) {
                fs.writeFileSync(extendedFile, "");
                filesToBeDeleted.push(extendedFile);
            }
        }
        typeScriptOptions["baseConfig"]["parserOptions"]["project"].push(configPathsArray[index].replace(/\\/g, '/'));
    } catch (e) {
        console.log(e);
    }
}
return;
}
```

A 4.2-es ábrán látható megvalósítás TypeScript nyelvű projektek elemzésére alkalmas, viszont JavaScript projektek és önálló fájlok elemzésére jelen helyzetben nem alkalmas. Ez abból adódik, hogy ezekben az esetekben nem tartozik az elemzendő kódbázishoz vagy fájlhoz konfigurációs fájl. A probléma megoldására ideiglenes fájlt generál a program (ez nem összetévesztendő a korábban említett ideiglenes konfigurációval, mely egy másik hiba javítására szolgál). Ez a fájl, melynek `include` mezője tartalmazza projekt esetén a gyökérkonyvtártól számítva a mappaszerkezet összes, kiterjesztésnek megfelelő fájlt, melyek megegyeznek az eddig megadott kiterjesztésekkel a fordító beállításaival. Önálló

fájl esetén az `include` mező csak az elemzett fájlt tartalmazza. Ilyen elemzések során az ideiglenes konfigurációs fájl van egyedül megadva a `tsConfigRootDir` mezőben. A fájl az elemzés lefutása után törlésre kerül. Ezzel a módszerrel a fordító alkalmas egyszerre TypeScript és JavaScript nyelven írt fájlok és projektek elemzésére.

4.3. ábra. Projekt ideiglenes konfiguráció készítése

```
const tempTsConfigPath = path.resolve(tempDir, "tsconfig.json");
globals.setTempTSConfigPath(tempTsConfigPath);

typeScriptOptions["baseConfig"]["parserOptions"]["project"] = tempTsConfigPath;

if (globals.getOption(constants.MODULE_BASED_ANALYSIS)
    && globals.getOption('inputList').length === 1
    && fs.existsSync(globals.getOption('inputList')[0])) {
    if (path.extname(globals.getOption("inputList")[0]) === '.ts' || path.extname(globals.getOption("inputList")[0]) === '.js'
    || path.extname(globals.getOption("inputList")[0]) === '.tsx') { //in case a single file is being analyzed
        tsConfigContent["include"].push(resolvePathForTSConfig(globals.getOption("inputList")[0]));
    } else { //directory
        tsConfigContent["include"].push(resolvePathForTSConfig(globals.getOption("inputList")[0]) + '/**/**.ts');
        tsConfigContent["include"].push(resolvePathForTSConfig(globals.getOption("inputList")[0]) + '/**/**.tsx');
        tsConfigContent["include"].push(resolvePathForTSConfig(globals.getOption("inputList")[0]) + '/**/**.jsx');
        tsConfigContent["include"].push(resolvePathForTSConfig(globals.getOption("inputList")[0]) + '/**/**.js');
    }
} else {
    for (let index in globals.getOption("inputList")) { //mba is not enabled, analyzing lots of files
        tsConfigContent["include"].push(resolvePathForTSConfig(globals.getOption("inputList")[index]));
    }
}

// Create a temporary directory which contains the newly created tsconfig
```

Az ESLintRunner futtatásához szükséges használat előtt a programot és a hozzá tartozó modulokat egy önálló fájlba „összetömöríteni” a webpack nevű szoftvercsomaggal. A program működése TypeScript nyelv támogatás implementálása előtt nem függött külső bővítményektől. Összetömörítés után futás közben az alapértelmezett, „esprima” nevű fordítóval elemzett, mely az eddig használt modulokba volt beépítve. Az újonnan használt bővítmények használata miatt a program fordítója kényszerítve van külső útvonalak vizsgálatára. A program önmagára nem tud hivatkozni, mivel ezzel az alapértelmezett fordítót használná eddigi implementáció miatt, a modulokat tartalmazó mappát (`node_modules` nevű mappa) tömörítés során pedig nem másoljuk át a kimeneti útvonalra. Erre a probléma megoldására többek között a webpack beállításainak módosításával és alternatív kötegelő programok alkalmazásával próbálkoztam. A webpack alternatívák, melyek közé tartozik a `Parcel`, a `Browserify` és a `Rollup.js`, eggyike sem volt alkalmas megoldás, mivel mindegyik hasonló, már webpack használata közben észlelt hibákat állított elő. Észrevehető, egyéb különbségként a használatuk változó nehézsége és gyengébb teljesítményüknek köszönhetően a webpack beállítások módosítása maradt.

Webpack konfigurációban a modul importálások rezolválására használt „álnevek”

(alias) beállítása szolgálhatott volna megoldásul [12]. Ezzel meg lehet adott moduloknak határozni, hogy milyen néven lehessen a programon belül importálni. Ismételt módon a fordító egyedi működésének köszönhetően ez nem használható megoldás, mert a felhasznált pluginokat csak név szerint adjuk meg konfigurációval, nem pedig import utasításokkal.

A végső megoldása ennek a hibának bár egyszerű, később változtatást igényel, amint a fordító program bővítmény kezelésén frissítenek. A program „kötegelése” során a generált fájl mellé a modulokat tartalmazó mappa is átmásolásra kerül. Ez a szoftvercsomag fordítása során van elvégezve CMake parancs segítségével, mely a 4.4-es ábrán látható. Ennek köszönhetően futásidő folyamán a keresett modult megtalálja és hibamentesen tudja feladatát végezni. Ennek legfőbb hátránya, hogy a program által foglalt hely többszörösére nőtt. A „kötegelési módszer” változtatása eddigi tesztelések alapján futásidőn nem rontott.

```
1  add_custom_command (
2      OUTPUT ${EXECUTABLE_OUTPUT_PATH}/node_modules/${PROGRAM_NAME}
3      COMMAND ${CALL} npm install > ${CMAKE_CURRENT_BINARY_DIR}/${↩
           PROGRAM_NAME}-npm-install.log 2>&1
4      COMMAND ${CALL} npm run build -- --no-color -o ${↩
           EXECUTABLE_OUTPUT_PATH}/node_modules/${PROGRAM_NAME} > ${↩
           CMAKE_CURRENT_BINARY_DIR}/${PROGRAM_NAME}-npm-build.log ↩
           2>&1
5      COMMAND ${CMAKE_COMMAND} -E copy_directory ${↩
           EXECUTABLE_OUTPUT_PATH}/tmp_${PROGRAM_NAME}/node_modules/↩
            ${EXECUTABLE_OUTPUT_PATH}/node_modules/${PROGRAM_NAME}/↩
           node_modules/
6      ...
7  )
```

Kódrészlet 4.4. CMake parancs modul mappa másolására

Végül a TypeScript támogatás megvalósításához szükséges volt az új fordító által használt szabálysértések bevezetése a programba. Ehhez kettő különálló fájlt kellett megvalósítanom. A korábban említett `.json` kiterjesztésű szabály fájl és a `.rul.md` fájl között legfőképp formai eltérések vannak és különböző módon kell őket elkészíteni.

Az alapértelmezett szabály fájl (`.json` fájl) egy JavaScript objektumot tartalmazó fájl, melyben fel van sorolva a fordítóhoz tartozó összes szabály egy, vagy több hozzárendelt értékkel. A szabályokhoz minden esetben tartozik szám, mely meghatározza, hogy elemzés közben adott szabálysértést detektálni akarjuk-e, és milyen súlyosságot kívánunk hozzá rendelni az elemzés eredményében. Ez a szám 0, 1 vagy 2 értéket vehet fel, 0 esetén átlépi, 1 esetén figyelmeztetésként (warningként), 2 esetén pedig hibaként (errorként) észleli a program. A második érték, mely nem minden szabálynál található meg, bővebb információt tartalmaz arról hogy pontosan milyen fajtáját detektálja a program. A 4.15-es ábrán ennek a fájlnak egy részlete található.

```
{
  "for-direction": 0,
  "getter-return": [ 0, { "allowImplicit": true } ],
  "no-await-in-loop": 0,
  "no-compare-neg-zero": 1,
  "no-cond-assign": [ 1, "except-parens" ],
  "no-console": [ 1, { "allow": [ "warn", "error" ] } ],
  "no-constant-condition": [ 1, { "checkLoops": true } ],
  "no-control-regex": 1,
  "no-debugger": 1,
  ...
}
```

Kódrészlet 4.5. Szabálysértések megadása `.json` állományban

A másik szabály fájl (`.rul.md`) ugyan ezeket a szabályokat tartalmazza egy átláthatóbb formátumban. Ebben a fájlban a szabálysértésekhez a súlyosságukon kívül kulcsszavak és leírás is tartozik, melyek alapján a szoftvercsomag felhasználói személyre szabhatják az elemzést. Ennek a szabályfájlnak módosításához korábbi eszközkészlet verziókban egy több lépésből álló folyamat volt. Eredetileg `.rul` típusú állományokat használtak fel a szabálysértéseket vizsgáló programok. Ezeknek a módosítása során először a módosítandó szabályfájlt `.md` állománnyá kellett konvertálni és a kellő módosításokat Markdown formátumban megírni a fájlban jelen lévő formai követelmények alapján. Ezután a módosított `.md` fájlt `.html` állománnyá (HyperText Markup Language), majd vissza `.rul` állománnyá kellett visszaaállítani külső segédprogramok használatával. Ez a fo-

lyamat a jelenlegi eszközkészlet verzión belül jelentősen rövidebb lett. Az új szabályokat a `.rul.md` állományban formai követelményeknek megfelelően, hosszas konvertálások nélkül lehet implementálni. A szabályokat h2-es fejlécekkel tagolva adjuk meg. Ezen fejléceknél adjuk meg a belső azonosítóját az adott szabálynak, majd alatta h4-es fejlécekkel felsorolva adjuk meg a szabály információit.

- Az `OriginalId` mező tartalmazza az elemző által felismert szabály nevét.
- Az `Enabled` és a `Warning` mezők határozzák meg hogy észleljük-e az adott szabálysértéseket, hibaként vagy figyelmeztetésként.
- A `HelpText` mező rövid leírást tartalmaz a szabálysértésről.
- A `Tags` mező alatt címkéket lehet meghatározni, melyek alapján a szabálysértések kategorizálhatóak.
- A `Settings` mező alatt néhány szabálysértésnek további információkat is lehet megadni.

Mivel nyelvre és eszközre szabottak a `.rul.md` fájlok, így csak a `/general/` címkénél szükséges szabályhoz illő értéket megadni, a `/language/` és a `/tool/` címkék megegyeznek. Emellett jelenleg minden szabálysértés, melyet a SourceMeter for JavaScript észlel, `Settings` mező alatt `Major`, azaz fontos prioritású. Alább a 4.4-es ábrán található egy szabály beállításához példa.

4.4. ábra. Szabálysértés megadása `.md.rul` fájlban

```
## ESLINT_TSKS
### Default
#### OriginalId=@typescript-eslint/keyword-spacing
#### Enabled=false
#### Warning=true
#### DisplayName=keyword-spacing
#### HelpText
  This rule extends the base eslint/keyword-spacing rule. This version adds support for generic type parameters on function calls.


#### Tags
- /language/TypeScript
- /tool/ESLINT
- /general/TS Stylistic Issues

#### Settings
- Priority=Major
- __eslint_param__={ "before": true, "after": true, "overrides": {} }
```

A szabálysértések manuálisan lettek beillesztve, melyek megfelelnek az ESLintRunnerben használt új parser dokumentációjában szereplő információknak. Az, hogy mely szabálysértések vannak engedélyezve elemzés során és szükség esetén ezeknek további részletei megbeszélés alapján voltak meghatározva.

4.3. JSAN

4.3.1. JSAN bevezető

A SourceMeter for JavaScript másik alapvető programja a JavaScript elemző, (továbbiakban JSAN). Ez a program, mint korábbi fejezetben említve, a bemenetként kapott projekt kódját átalakítja egy egyedi absztrakt szintaxisfává, melyet az elemzés későbbi lépéseiben használunk fel duplikált kódszegmensek keresésére és metrikák kiszámolására. Ennek a programnak a bővítése jelentősen egyszerűbb ESLintRunnerhez hasonlítva. Az eredeti kódban található egy nagy terjedelmű switch, melyben az elemzés során talált „csomópontok" (továbbiakban node) típusai vannak felsorolva. Minden felsorolt típushoz tartozik egy külön fájl, mely tartalmazza exportált funkció formájában, hogy a talált node esetén milyen eljárásokat kell elvégezni és milyen adatokat kell beállítani, például egy funkció hívása esetén többek között mi a hozzátartozó név, mik a paraméterei és mi a hozzátartozó kódrészlet.

4.5. ábra. Részlet a JSAN node elemzőjéből

```
let convertNodeToFunction = function (node) {
    switch (node.type){
        case "System":
            return System;
        case "Comment":
            return Comment;
        case "Program":
            return Program;
        case "JSXIdentifier":
            return JSXIdentifier;
        case "ImportDefaultSpecifier":
            return ImportDefaultSpecifier;
        case "ImportNamespaceSpecifier":
            return ImportNamespaceSpecifier;
        case "ImportSpecifier":
            return ImportSpecifier;
```

A program bővítéséhez ezt a switch-et szükséges bővíteni a TypeScript nyelvben található node-ok típusaival, és létre kell hozni ezen típusokhoz tartozó fájlokat. Ennek a feladatnak a manuális megvalósításával több probléma is fellép. A program karbantarthatósága és olvashatósága jelentősen csökken azzal, hogy több mint 100 új típust vezetnék be az elemzéshez használt switch-be, emellett az említett fájlok manuális megírása vagy módosítása során több hibát is ejthet a fejlesztő, mely miatt a fejlesztési folyamat több időt

igényelhet. Manuális fejlesztésre alternatív megoldásként a SourceMeter eszközkészletnek egy segédprogramját, a SchemaGenerator-t olyan funkcióval bővítettem, mely segítségével a szoftvercsomag fordításánál automatikusan legenerálódik a szükséges switch és a hozzá tartozó fájlok.

4.3.2. SchemaGenerator bővítése sablonkód generálással

A korábban említett segédprogrammal, a SchemaGenerator-ral valósítottam meg az automatikus fájl generálást. A segédprogram bővítésének célja, hogy sikeresen lehessen generálni több száz node típushoz tartozó fájlt előre meghatározott sablon alapján és a node típusra vizsgáló switchet tartalmazó fájlt, importálásokkal együtt. A fejlesztés folyamatát fokozottan felgyorsította az, hogy már korábban használt fájlok generálásához (például a szoftvercsomag esetén a korábban említett javascriptAddon fájl) már előre meg voltak írva séma bejárására használt funkciók. SchemaGenerator-on belül található a `traversalDescendantBFT()`, `traversalAllEdges()` és a `traversalAllAttributes()` nevű funkciók. Ezeknek a funkcióknak segítségével lehet bejárni az adott sémákban definiált node típusokat, az ezekhez tartozó attribútumokat és éleket.

A switch generálása kettő lépésből áll. Először a generált fájlok importálását, majd a típusokat felsoroló switch-et kell legenerálni. Mivel ehhez csak a node-ok nevei szükségesek, a `traversalDescendantBFT()` funkció kétszeri hívásával ez a feladat egyszerű megoldással rendelkezik.

```
bool generateSwitchCase() {
    debugMessage(0, "Generating AST switch...\n");
    file = fopen("ASTSwitchCase.js", "w");
    if (file == NULL) {
        printf("Error opening file!\n");
        return false;
    }
    getFileNamesForImports();
    fprintf(file, "import {default as Literal} from \"./nodes/Literal.js\";\n");
    fprintf(file, "\nlet convertNodeToFunction = function (node) {\n");
    fprintf(file, "    switch (node.type){\n");
    getFileNamesForCases();
    ...
}
```

Kódrészlet 4.6. Fájl genetálásra használt funkció

A kettő bejárás során közel azonos módon dolgozza fel a program az adatokat, mivel csak formailag más az adott node-hoz tartozó kód kiíratása. Bár a két funkció összevonható lenne tartalmuk alapján, kód olvashatóság és későbbi továbbfejleszthetőség miatt külön vannak kezelve. Ezeken kívül adott dolgokat statikusan kiíratunk a generált fájlba, többek között a switch deklarálását, exportálását, és adott node típusokat a switch-en belül. Sémában definiált, egyedi működés miatt literálokat egy node típusként kezelünk ahelyett, hogy literál fajtánként külön node-okat veszünk fel. Emellett az AST ellenére, melyben a literálok különböző node-okat kaptak, elemzés során minden literál node `LiteralExpression`-ként van észlelve, és ezt manuálisan szükséges lekezelni.

Node típusok között az AST-n belül találhatóak node párosok, melyek `Computed` és `NonComputed` végződésűek. Ezek a node típusok olyan kódolási elemeket foglalnak magukba, melyeknek a hivatkozásaik megadhatóak előre, fordítás előtt statikusan (`NonComputed`), vagy fordítás során, futásidő közben (`Computed`).

```
if (node->type.abstract && strcmp(node->name, "LiteralExpression") !=↩
    0) {
    return true;
}

if (strstr(node->name, "NonComputedName") != NULL) {
    return true;
}
...
```

Kódrészlet 4.7. Absztrakt és speciális node szűrés

Ezek a node-ok speciális módon vannak megkülönböztetve az AST-ben és a futásidő közben elemzett node-ok között. AST-ben ezek különböző bejegyzéseket kapnak azonos tulajdonságokkal, viszont egy ilyen tulajdonságuk név alapján kapnak értéket. A node-ok tartalmaznak egy `computed` mezőt, mely `Computed` node-oknál igaz értéket kapnak, `NonComputed` esetén pedig hamis értéket. Elemzés során viszont ezek a node-ok nincsenek névlegesen megkülönböztetve, de a `computed` mező alapján különböző típusú node-nak számítanak. Emellett vannak absztrakt node-ok is meghatározva az AST-ben, melyek a sémában megjelennek mint node-ok, de elemzés során nem. Ezek a fájlgenerálás folyamán kihagyhatóak, kivéve a LiteralExpression, ami absztrakt létének ellenére szükséges, mint összefoglaló node a literál típusokra.

Ezek lekezelése könnyen elvégezhető azzal, hogy névre és adott tulajdonságokra szűrve kihagyunk adott node-ok generálását. Ebben az esetben kiszűri az absztrakt node-okat, melyeknek a neve nem `LiteralExpression`, majd azokat, melyeknek a neve tartalmazza a `NonComputedName` szövegrészletet. Azért szűrünk kifejezetten `NonComputedName`-re, mert a node párosokból legalább az egyikre szükség van hogy helyesen lehessen legenerálni a switchet.

```
if (strstr(fileNameWithExtension, "ComputedName"))
    subStrCopyFunc(fileNameWithExtension, fileNameWithExtension, ↩
    strlen(fileNameWithExtension) - strlen("ComputedName"));
    strcat(fileNameWithExtension, ".js");
```

Kódrészlet 4.8. Computed és NonComputed node-ok kezelése

Az egyes node-okhoz tartozó fájlok legenerálása sokkal több szűréssel és feltétellel jár. Bár legtöbb fájl azonos sablon alapján lesz felépítve, vannak speciális fájlok, mint a Program és a Literal, melyek statikusabb módon épülnek fel. Emellett a `ComputedName` és `NonComputedName` node-oknak másképp kell megadni a wrapperjeiket, mivel a két fajta node, bár egyeznek (egy tulajdonság kivételével), mégis külön wrappert kapnak.

Switch generáláshoz hasonlóan itt is le kell szűrni az absztrakt és a `NonComputed` node-okat, az előző funkciókhoz hasonló módon. Adott, minden node esetén megjelenő tulajdonság kiíratása után le kell generálnunk a first visit-be tartozó tulajdonságok szettereit, és az él metódusok szettereit. Ezt két külön funkcióval valósítottam meg, mindkettőnek paraméterként megadva a jelenlegi fájl generálásához felhasznált node. Ezekkel a funkciókkal bejárjuk az adott node-oknak a FirstVisit attribútumait, és az edge szettereit és, mint eddig nevek alapján tettük, tulajdonság nevekre és típusokra szűrve írattatom ki a megfelelő szettereket. FirstVisit attribútumok esetén, mivel adott node-oknál más típusú attribútumok fordulhatnak elő, vagy szituációtól függően máshogy kell beállítani az attribútumokat, ezeket külön külön kiszűrjük futásidő folyamán.

```
if (
  (strcmp(currentNode->name, "TSAbstractMethodDefinitionComputedName") == 0
  || strcmp(currentNode->name, "TSMethodSignatureComputedName") == 0
  || strcmp(currentNode->name, "MethodDefinitionComputedName") == 0)
  && strcmp(lowerName, "kind") == 0) {
  fprintf(f, "if (node.%s != null) %s.set%s(
    conversions.convertMethodDefinitionKind(node.%s));\n",
    lowerName, currentNode->name, upperName, lowerName);}
else if (strcmp(lowerName,"kind") == 0
  || strcmp(lowerName, "importKind") == 0
  || strcmp(lowerName, "exportKind") == 0
  || strcmp(lowerName, "accessibility") == 0) {
  fprintf(f, "if (node.%s != null) %s.set%s(conversions.convertKinds(node.%s)↩
    );\n", lowerName, currentNode->name, upperName, lowerName);}
```

Kódrészlet 4.9. Attribútum szűrésről példa

Él szetterek esetén csak két esetre szükséges figyelnünk. Mivel minden él szetter esetén más node-okat kapnak a legenerált funkciók, így típusra nem szükséges figyelnünk (mivel ez már sémában le van kezelve), kizárólag a multiplicitást kell ellenőriznünk, azaz

egy adott node tulajdonsághoz egy node-ot, vagy egy tömbnek a node-jait rendeljük.

```
if (edge->mult == 1) {
    ...
} else {
    ...
}
```

Kódrészlet 4.10. Él szetterek fájlba írása

4.3.3. Generált fájlok átmásolása

Miután ezeket a lépéseket minden szükséges node-ra elvégeztük, a program áttérhet a fájlok átmásolására megfelelő célmappába. A generált fájlok átmásolása a projekt fordítása folyamán történik meg. A fájlokat a korábban említett javascriptAddon mellé generáljuk egy mappában, mely strukurálisan már elő van készítve másolásra, rendelkezik a megfelelő mappaszerkezettel. Korábbi fordítások során ha a JSAN-t akartuk lefordítani, a SchemaGenerátor automatikusan meg volt hívva javascriptAddon generálására és megfelelő helyre másolására, így az új fájlokat is ehhez a folyamathoz rendeltem. A programhoz tartozó CMakeList-ben adtam meg utasításként a másolási parancsot. Emellett figyeltem arra, hogy a fájlok generálásához szükséges parancsok azután vannak meghívva, amikor a javascriptAddon fájl, és ezzel együtt az újonnan létrehozott fájlok generálása végbe ment. Az alább látható ábrán `-copy_directory` CMake parancs használatával volt ez megvalósítva.

```
add_custom_command (
    OUTPUT ${EXECUTABLE_OUTPUT_PATH}/node_modules/${PROGRAM_NAME}
    COMMAND ${CMAKE_COMMAND} -E copy ${EXECUTABLE_OUTPUT_PATH}/↩
        javascriptAddon.node ${EXECUTABLE_OUTPUT_PATH}/tmp_${PROGRAM_NAME}/
    COMMAND ${CMAKE_COMMAND} -E copy_directory ${EXECUTABLE_OUTPUT_PATH}/../↩
        lib/javascript/addon/ast ${EXECUTABLE_OUTPUT_PATH}/tmp_${PROGRAM_NAME↩
        }/src/ast/
    COMMAND ${CALL} npm install ${MSVS_VERSION} > ${CMAKE_CURRENT_BINARY_DIR↩
        }/${PROGRAM_NAME}-npm-install.log 2>&1
    COMMAND ${CALL} npm run build -- --no-color -o ${EXECUTABLE_OUTPUT_PATH}/↩
        node_modules/${PROGRAM_NAME} > ${CMAKE_CURRENT_BINARY_DIR}/${↩
        PROGRAM_NAME}-npm-build.log 2>&1
```

```
7       WORKING_DIRECTORY ${EXECUTABLE_OUTPUT_PATH}/tmp_${PROGRAM_NAME}
8       DEPENDS javascriptAddon ${EXECUTABLE_OUTPUT_PATH}/tmp_${PROGRAM_NAME}
9       COMMENT "Rebuilding ${PROGRAM_NAME}"
10  )
```

Kódrészlet 4.11. JSAN Cmake utasítások

Az AST-hez tartozó fájlok automatikus legenerálásának köszönhetően a JSAN karbantartása egyszerűbb lett. Jelentősebb AST változtatások esetén nem lesz szükséges az összes node manuális átnézése és ellenőrzése, hanem csak adott sablonokon kell néhány sorral bővíteni vagy módosítani, mely drasztikusan felgyorsítja a folyamatot. Automatikus generálás esetén olyan problémákat is elkerülhetünk, melyek emberi figyelmetlenségből erednek, például hibás változó deklarálások, melyek könnyen előfordulhatnak többszáz fájl módosítása esetén. Emellett fejlesztés során pár kisebb hibát is észrevettem az AST-ben, melyek javításra kerültek.

4.4. JSCG bővítése

4.4.1. JSCG bevezetés, hiba felvázolása

A JSCG, mint korábban említve volt, JSAN által van használva hívásgráf előállítására, melyet a szoftvercsomag későbbi eszközei használnak fel például különböző metrikák számolására. A program futtatása JSAN-on belül történik meg, közvetlen miután az elemzett projekthez tartozó kódbázisnak a fájljai alapján elkészült az AST. Ezeket paraméterként átadja a JSAN a JSCG-nek és meghatározza, hogy milyen stratégiával készítse el a hívásgráfot. Jelenleg `NONE`, `ONESHOT` és `DEMAND` stratégiák használata áll rendelkezésre, melyek közül a `ONESHOT` van felhasználva. A stratégiák közötti különbség az interprocedúrális áramlat követésében különböznek. Nevének megfelelően a `NONE` stratégia ezt nem követi, míg a `ONESHOT` csak az azonnal meghívott egyszeri lezárások esetében teszi meg. JavaScript esetében szükséges volt a hívásgráfok legenerálása a nyelv dinamukissága miatt[5]. Ez TypeScript projekteknél is szükséges, mivel JavaScript nyelv bővítésére szolgál.

4.4.2. Bővítés, hibajavítás lépései

A program már rendelkezett kezdetleges TypeScript támogatással, viszont jelentős mennyiségű node lekezelésére nem volt meghatározva működés. Ezen kívül specifikus TypeScript nyelvtani tényezők észlelése esetén hibába futott a program, mivel eddigi implementáció alapján érvénytelen adatot olvasott be. Ezek a változtatások, JSAN-hoz hasonló módon az új `typescript-eslint` elemző által használt AST specifikáció alapján lett megvalósítva.

Az előző programok bővítése és folyamatos tesztelések során a JSAN futtatása közben a JSCG által kimutatott hibák alapján voltak meghatározva hogy mely node típusoknál volt szükség változtatásra vagy új működés definiálására. Három fő változtatást kellett megvalósítanom:

- `PrivateIdentifier` észlelés `Identifier` helyén, melyek osztályokban metódus és változó definiálásoknál találhatóak
- `Computed` mezők helyes detektálása osztályokon belül
- `TSEmptyBodyFunctionExpression` helyes bekötése

A három közül a `TSEmptyBodyFunctionExpression` helyes bekötése volt az első amit megvalósítottam. A program több helyen node típusra szűrve végez el adott utasításokat. Ezen szűrések között voltak olyanok, melyek adott funkció típusokra volt érvényes, ezeket kiszerveztem külön funkcióba és hozzáillesztettem az új típust. A funkciótípusok és azok kezelése azonos módon történik AST specifikáció alapján, így ezeknél elegendő volt az új típus ilyen módú beillesztése.

```
1 function isFunction(nd) {
2     return nd.type === 'FunctionDeclaration' ||
3         nd.type === 'FunctionExpression' ||
4         nd.type === 'ArrowFunctionExpression'
5         nd.type === 'ArrowFunctionExpression' ||
6         nd.type === 'TSEmptyBodyFunctionExpression'
7 }
```

Kódrészlet 4.12. Funkció típus szűrő

`PrivateIdentifier` esetén hasonló módosításokat hoztam létre, ahol `Identifier` node típust vizsgált a program, helyettesítettem funkcióval, mely `PrivateIdentifier` és `Identifier` típusokra szűrt.

```
1 function isIdentifier(ndType) {
2     return ndType === 'Identifier' ||
3     ndType === 'PrivateIdentifier'
4 }
```

Kódrészlet 4.13. Identifier típus szűrő

`Computed` mezők teljes körű támogatásának bevezetése működésükből adódóan nem teljesen volt implementálható. Ezek a mezők TypeScriptben osztályokban, interfészekben és objektumokban találhatóak, és olyan célt szolgálnak, mellyel futásidő során az említett objektumokhoz dinamikusan nevekkel adhatunk meg metódusokat. Kapcsos zárójeleken belül primitív típusok (szöveg, szám) megadása és ezek között elvégezhető műveletek mellett változók megadhatóak, melyeknek értékei lesznek felhasználva a metódus nevének meghatározására.

```
1 module.exports = {
2     ['foobar3']() {
3       return 'fizz';
4     },
5 };
```

Kódrészlet 4.14. Computed mező példa

Ebből adódóan pontos bekötéseket a hívásgráfba nem lehet létrehozni, mivel a projekt futtatása nélkül pontos név megadása ezeknek a mezőknek nem lehetséges. `FunctionExpression` típusú node-ok esetén, melyek `Property` típusú node-ból származnak (azaz a korábban említett objektumok valamelyikének egyik metódusaként van deklarálva), adott esetekre definiáltam ideiglenes működést.

Három gyakori esetre szűrtem le a működést. Amikor primitív adattípusokkal, vagy azok között valamilyen művelettel volt megadva érték, akkor az említett node-oknak az értékeit állítottam be mint azonosító. Emellett olyan esetben, amikor változó volt értékként megadva, a változó nevét adtam meg azonosítóként.

```
1  if (parent.key) {
2      if(parent.key.left && parent.key.right) {
3          nd.id = {
4              type: 'Identifier',
5              name: parent.key.left.value + parent.key.right.value,
6              range: parent.key.range,
7              loc: parent.key.loc
8          }
9  }
10 ...
```

Kódrészlet 4.15. Azonosító beállítás példa

Változtatások után a kódon belül érintett dokumentációkat felfrissítettem megfelelő külső hivatkozásokkal és AST referenciákkal. Emellett új teszteseteket is létrehoztam helyes működés ellenőrzésére.

5. fejezet

Tesztelés

5.1. Változtatások tesztelésének folyamata

A SourceMeter eszközkészlet szoftverjeinek a működését regressziós teszteléssel (röviden regtest) ellenőrizzük. Ez egy olyan tesztelési módszer, amely a változásokat követően újból elvégzi az alkalmazás korábbi tesztjeit annak érdekében, hogy megbizonyosodjon arról, hogy a változtatások nem okoztak olyan hibákat, amelyek hatással lehetnek az alkalmazás teljesítményére. A célja az, hogy könnyen ellenőrizhető legyen a korábban működő szoftverek új funkciók implementálása után is, és hogy az új funkciók, amelyeket hozzáadtak az alkalmazáshoz, nem okoznak olyan hibákat, amelyek negatívan befolyásolják az alkalmazás teljesítményét. Ezenkívül az alkalmazás megbízhatóságának növeléséhez és a hibák korai észleléséhez. A regressziós teszteket Python szkriptek segítségével futtatjuk le, melyek korábbi verzióban a SourceMeter szoftvercsomagjainak eszközeinek mindegyikéhez nyelv alapján rendelt egy-egy összefoglaló tesztet. Ezen összefoglaló tesztek a különböző eszközök számára tartalmaz nagy mennyiségben teszteseteket, melyek mindegyikének sikeresen kell lefutnia, hogy a programokban végbevitt fejlesztések el legyenek fogadva.

A tesztek eredményeit korábban, helyesen lefutott tesztek eredményeihez hasonlítjuk futtatás után, és a fennakadó különbségeket feltüntetjük `.diff` kiterjesztésű fájlokban. Ennek előnye, és egyben hátránya, hogy minden eltérésről értesül a fejlesztő tesztelés során, súlyosságtól függetlenül. Az eredmények vizsgálata során a fejlesztőnek kötelessége eldönteni, hogy a kimutatott eltérések ténylegesen hibák vagy hamis pozitívak.

5.1. ábra. Példa .diff fájl

```
+++ output\snowflakes\javascript\result\snowflakes.xml
@@ -6227,9 +6227,11 @@
       <attribute type = "int" name = "LLDC" context = "metric" value = "0"/>
       <attribute type = "int" name = "LDC" context = "metric" value = "0"/>
       <attribute type = "string" name = "CLC" context = "metric" value = "__INVALID__"/>
+      <attribute type = "int" name = "ESLINT_NUN" context = "metric" value = "5"/>
+      <attribute type = "int" name = "Variables" context = "metricgroup" value = "5"/>
       <attribute type = "int" name = "WarningBlocker" context = "metric" value = "0"/>
       <attribute type = "int" name = "WarningCritical" context = "metric" value = "0"/>
-      <attribute type = "int" name = "WarningMajor" context = "metric" value = "0"/>
+      <attribute type = "int" name = "WarningMajor" context = "metric" value = "5"/>
       <attribute type = "int" name = "WarningMinor" context = "metric" value = "0"/>
       <attribute type = "int" name = "WarningInfo" context = "metric" value = "72"/>
       <attribute type = "int" name = "CD_warning_Function" context = "metric" value = "22"/>
@@ -6283,6 +6285,8 @@
```

Ezek a tesztek minden implementált változtatás után lefutásra kerülnek, lokális számítógépen és Jenkins használatával központi szerveren. A Jenkins egy nyílt forráskódú, Java alapú, folytonos integrációs (Continous Integration, avagy CI) és folytonos szállítási (Continous Delivery, avagy CD) eszköz, amely lehetővé teszi a szoftvertervezők számára, hogy hatékonyan és automatizáltan integrálják, teszteljék és kézbesítsék az alkalmazásaikat. Ez alapvetően egy központi szerveren van futtatva, amely képes összekapcsolni és koordinálni a különböző forrásokat és eszközöket, például verziókezelő rendszereket, építőrendszereket és tesztelő keretrendszereket.

5.2. Regressziós tesztek módosítása

Ezeknek a statikus teszteknek volt egy olyan hiányossága, mely fejlesztés során nagyon sok időbe telt és jelentős frusztrációt okozott. A program elemzői adott nyelvekre csak egy fordítási célponttal (build targettel) rendelkeztek. Ez azt jelentette, hogy ha például egy adott részét akarnánk ellenőrizni a szoftvercsomagunknak, például csak az ESLint-Runnert, ahhoz a teljes SourceMeter for JavaScriptet kellett tesztelnünk, mert célpont nem volt definiálva. Ezeknek során azon kívül hogy „feleslegesen” teszteltük a programot fejlesztés során, jelentősen sok időbe telt az egyes futtatások végbemenetele, alkalmanként akár több órába is. Emellett olyan esetekben, amikor saját eszközön teszteltem, a build folyamat erőforrásigénye modern rendszereket is le tud terhelni annyira, hogy használhatatlan legyen a teszt buildelése és futtatása során, megállítva a fejlesztési folyamatot. Megoldásként az SourceMeter összes támogatott nyelvének összes programjához létrehoztam külön-külön tesztelési célpontokat, melyek során csak az adott teszthez szükséges

alprogramokat és függőségek fordultak le.

```
create_run_regtest_target(
    javascript_all AN-JavaScript GenealogyTool)
create_run_regtest_target(
    javascript_jsan jsan_virtual_target)
create_run_regtest_target(
    javascript_jsan2lim jsan_virtual_target JSAN2Lim)
create_run_regtest_target(
    javascript_eslintrunner eslintrunner_virtual_target ESLint2Graph)
create_run_regtest_target(
    javascript_duplicatedcodefinder AN-JavaScript GenealogyTool)
...
```

Kódrészlet 5.1. CMake célpontok JavaScript teszteléshez

Első lépésként a teszteléshez tartozó CMakeList fájlban kellett módosításokat hozni. Új célpontok létrehozása önmagában egyszerű, mert már definiált funkciók segítségével lehet létrehozni. A nehezebb része ennek a lépésnek az adott programokhoz tartozó függőségek kikeresése és hozzáadása az adott célpontokhoz. Legtöbb esetben ez egyszerű volt, mivel adott programokhoz név alapján volt rendelve CMake target, míg másoknál meg némi eltéréssel, de követhető módon. JSAN esetén például `jsan_virtual_target` volt megadva, azt jelezve, hogy nem kizárólag a JSAN volt csak fordítva ennek meghívásával, hanem egyszerre több minden, melyek szükségesek voltak helyes működéshez. Ezek kikutatása a meglévő tesztek alapján könnyen elvégezhető volt, tesztek során használt programok alapján rendeltem hozzá az adott függőségeket a célpontokhoz. Emellett adott nyelvek esetén, mint Java vagy C++, egyéb függőségek megadása is szükséges volt. Mivel a programokhoz nem volt kellő dokumentáció, mely részletezné a kellő függőségeket sikeres fordításhoz, ezek kikutatásához a fordítás során megjelenő hibák alapján adtam meg a különböző célpontokhoz a szükséges modulokat.

Ezután az új teszteseteket a teszteket futtató Python szkripten belül is definiálni kellett. Ennek a lépésnek nagy része tömbök létrehozásából és argumentumokhoz új választható lehetőségek felvételéből állt. A tömbök a különböző, futtatandó teszteket tartalmazta, melyek közül az `_all` utótaggal rendelkezők tartalmazták az adott nyelvnek összes tesztét, míg a többi, melyek program nevekkel voltak ellátva, csak az adott program tesztjét

futtatta.

5.2. ábra. JavaScript regressziós tesztjeit tartalmazó objektumok

```
],
'javascript_jsan':                  [JSANTest],
'javascript_jsan2lim':              [JSAN2LimTest],
'javascript_eslintrunner':          [ESLintRunnerTest],
'javascript_eslint2graph':          [ESLint2GraphTest],
'javascript_duplicatedcodefinder':  [DuplicatedCodeFinderJavaScriptTest],
'javascript_lim2metrics':           [LIM2MetricsJSTest],
'javascript_lim2patterns':          [LIM2PatternsJavaScriptTest],
'javascript_sourcemeter':           [SourceMeterJavaScriptTest],
'javascript_changetracker':         [ChangeTrackerJavaScriptTest],
'javascript_all': [
    JSANTest,
    JSAN2LimTest,
    ESLintRunnerTest,
```

Adott helyeken még szükséges volt módosítani ahhoz, hogy akadálymentesen futhassanak a tesztek. Java tesztek esetén ellenőrizni kellett a `JAVA_HOME` környezeti változóra, hiszen hiányában ezek a tesztek nem futnak le. Emellett Python teszteknél futásidőben ellenőrzi a program dinamikusan, hogy megfelelő verzióval rendelkezünk-e, mivel futásidő közben telepíti a teszt az elemzéshez szükséges `pylint` szoftvert. Ezek ellenőrzéséhez létrehoztam külön tömböket, melyek tartalmazzák a nyelvekhez tartozó teszteket szöveg formátumban, mivel a paraméterként kapott teszteket is ilyen formátumban kezeljük. A teszteket, amiket megadhatunk paraméterként, hozzáadtam a megfelelő argumentumhoz. Ezeket nyelvenként csoportosítva, tagolva adtam meg ezeket.

A módosítások után még fennállt egy probléma. Minden teszt futtatása esetén (`Regtest_all` futtatása) minden teszt kétszer futott le. A teszteknek a tömbjei egy `all_tests` nevű szótárban (dictionary) vannak megadva. Tesztek futtatása során ellenőrizzük hogy adott nyelvhez tartozik-e az éppen iterált teszt, ha nem, akkor kihagyjuk annak futtatását. Mivel a tesztek külön-külön meg voltak adva a programokra és nyelvekre, így alapesetben minden teszt kétszer futott volna le. Ennek megoldására legegyszerűbb módszer a duplikált tesztek törlése volt abban az esetben, ha a `Regtest_all` futtatása volt a feladat, azaz minden, nyelvhez tartozó összetett tesztet töröltünk az `all_tests`-ből.

```
1 if LANG == "all":
2     del all_tests['cpp_all']
3     del all_tests['csharp_all']
4     del all_tests['java_all']
5     del all_tests['javascript_all']
6     del all_tests['rpg_all']
7     del all_tests['python_all']
```

Kódrészlet 5.2. Teszt törlési lépések

6. fejezet

Elért eredmények

6.1. Eddigi sikeres bővítések

Szakdolgozatom során sikeresen bővült a SourceMeter for Javascript szoftvercsomag TypeScript nyelvi támogatással.

Az ESLintRunner program az új fordító használatával alkalmas JavaScript és TypeScript projekteken belül eddig használt és újonnan bevezetett szabálysértések észlelésére is.

Karbantarthatóság szempontjából javítottam JSAN-on, jelentősebb séma felépítés változtatások esetén is többek között kisebb kódszegmensek módosítása lesz csak szükséges.

A tesztelési folyamatok szoftvercsomagok és önálló szoftverek tesztelésére is alkalmasak lettek.

6.2. Jövőbeli tervek

A szakdolgozatomban érintett nyelvek és azoknak elemzésére használt eszközök folyamatosan változnak, melyből kifolyólag nem naprakész a projekt. Fejlesztés során a használt elemző [10] is több verziófrissítésben részesült, melyeknek változtatásait a bővítés folyamán próbáltam implementáni. Feladataim során jelentős tudásra tettem szert a szoftvercsomag működéséről, melynek köszönhetően jövőbeli bővítési feladatokat sokkal gyorsabban el tudom végezni.

Nyilatkozat

Alulírott Baga Csaba 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 13. .....
aláírás

Irodalomjegyzék

[1] Tibor Bakota, Petér 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, 2014.

[2] 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.

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

[4] The ESTree Spec. `https://github.com/estree/estree` Utolsó megtekintés: 2023.05.13.

[5] Asger Feldthaus, Max Schäfer, Manu Sridharan, Julian Dolby, and Frank Tip. Efficient construction of approximate call graphs for javascript ide services. In *2013 35th International Conference on Software Engineering (ICSE)*, pages 752–761. IEEE, 2013.

[6] S. Fenton. *Pro TypeScript: Application-Scale JavaScript Development*. Apress, 2017.

[7] R. Ferenc, A. Beszedes, M. Tarkiainen, and T. Gyimothy. Columbus - reverse engineering tool and schema for c++. In *International Conference on Software Maintenance, 2002. Proceedings.*, pages 172–181, 2002.

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

[9] 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.

[10] typescript-eslint - The tooling that enables ESLint and Prettier to support TypeScript. `https://typescript-eslint.io/` Utolsó megtekintés: 2023.05.13.

[11] typescript-eslint AST specification. `https://github.com/typescript-eslint/typescript-eslint/tree/main/packages/ast-spec` Utolsó megtekintés: 2023.05.13.

[12] Module resolution using webpack. `https://webpack.js.org/configuration/resolve/` Utolsó megtekintés: 2023.05.13.