Szegedi Tudományegyetem
Informatikai Intézet

TypeScript támogatás megvalósítása ipari JavaScript elemző szoftverhez

Szakdolgozat

Készítette:
Baga Csaba
programtervező informatikus BSc szakos hallgató

Témavezető:
Dr. Antal Gábor
egyetemi docens

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:

Szöveg

A megadott feladat megfogalmazása:

Szöveg

A megoldási mód:

Szöveg

Alkalmazott eszközök, módszerek:

Szöveg

Elért eredmények:

Szöveg

Kulcsszavak:

Szöveg

Tartalomjegyzék

1. fejezet

Bevezetés

A szakdolgozatom célja egy forráskód elemző eszközkészlet, névlegesen a SourceMeter bővítése volt, melynek az Analyzer-Javascript alrészébe TypeScript nyelv elemzésének implementálása volt a fő célja. Mivel ez a szoftvercsomag sok, kisebb programból áll össze, melyek egymásra épülnek, az új nyelv teljeskörű támagotásához több alprogramban kellett változtatásokat hozni. 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-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 sozftvercsalád

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 gyengepontokat és kódolási szabálysértéseket mutat 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 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, avagyNesting Level), vagy megjegyzések sűrűsége (CD, avagy Comment Density). Ezek mellett kódolási szabálysértések és duplikált kódsorok detektálására is alkalmas az eszközkészlet adott szoftverjei.

A SourceMetert 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. 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 (fejlesztés során schema), 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. –image?–

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 használva. 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 irányítási folyam-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 végzi, mely segítségével az elemzett nyelvtani elemeket beágyazza 'burkoló csomagpontba', másnéven 'wrapper'-ekbe. A nyelvtani elemeket és a 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 implementálása, 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 feladatok automatikus legenerálását valósítottam meg, melynek segítségével könnyebben karbantarthatóbbá vált az eszköz.

JavaScriptCallGraph, 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 redszeren 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

Változtatások megvalósítása

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.

Az új nyelv támogatásához a typescript-eslint által felvázolt AST 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 alapján készített és az új, TypeScriptet támogató séma között. Emellett naprakész dokumentáció nem állt rendelkezésre sem a meglévő sémához, sem teljesen új készítéséhez.

[esetleg példa volt sémából]

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

Az Analyzer-Javascript 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 hogy mely fájlokban fordulnak elő, sor és karakter pontosságra. Ezeket egy .xml formátumú fájlba helyezi a program futás után.

–image?–

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' 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 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 ESLintRunner 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.

Ahhoz, hogy ezen beállítások futásidő során helyesen be lehessen állítani, elemzés előtt kell átvizsgálni a programmal az elemzendő projektnek a mappaszerkezetét. Mivel formailag kötelező TypeScript projekteknek a gyökérkönyvtárában, 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 adott fájlok szűrését meg lehet határozni az 'exclude' mezőkben.

—példa—

Eddigi 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 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.

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 beépítve volt. Az újonnan használt bővítmények használata miatt a program fordítója kényszerítve van külső útvo-

nalak 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. 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 amodulokat tartalmazó mappát is átmásolásra kerül. 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.

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 legfőbbképp 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. Ez az szám 0 vagy 1 értéket vehet fel, 0 esetén átlépji, 1 esetén pedig detektálja a program. A második érték, mely nem minden szabálynál található meg, bővebb információt tartalmaz arról hogy pontosan milyen fajtáját detektálja a program.

A másik szabály fájl (.rul.md) ugyan ezeket a szabályokat tartalmazza egy átláthatóbb formátumban.

4.2.3. Elért eredmények ESLintRunnerben

4.3. JSAN

4.3.1. JSAN bevezető

Az Analyzer-JavaScript másik alapozó programja a JSAN, avagy a JavaScript Analyzer. 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 van egy nagy terjedelmű switch, melyben az elemzés során talált "csomópontok" (továbbiakban node-ok) 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.

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 vezetünk 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, hogy már korábban használt fájlok generálásához (például szoftvercsomagunk esetén a korábban említett javascriptAddon fájl) már előre meg voltak írva séma bejárására használt funkciók. SchemaGenerátoron 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.

[kódrészlet]

A kettő bejárás során közel azonos módon dolgozza fel a program az adatokat, mivel csak formailag más az adott node-hoz tartozó kód kiiratása. Bár a két funkció összevonható lenne tartalmuk alapján, kód olvashatóság és későbbi továbbfejleszthetőség miatt külön vannak kezelve. Ezeken kívül adott dolgokat statikusan kiiratunk 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 nodeokat veszünk fel. Emellett az AST ellenére, melyben a literálok különböző nodeokat 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) [CITATION HERE].

Ezek a nodeok speciális módon vannak megkülönböztetve az AST-ben és a futásidő közben elemzett nodeok között. AST-ben ezek külön-külön bejegyzéseket kapnak azonos

tulajdonságokkal, viszont egy ilyen tulajdonságuk név alapján kapnak értéket. A nodeok tartalmaznak egy "computed" mezőt, mely "Computed" nodeoknál igaz értéket kapnak, "NonComputed" esetén pedig hamis értéket. Elemzés során viszont ezek a nodeok nincsenek névlegesen megkülönböztetve, de a "computed" mező alapján különböző típusú node-nak számítanak. Emellett vannak absztrakt nodeok is meghatározva az AST-ben, melyek a schemában megjelennek mint nodeok, de elemzés során nem jelennek meg. 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 kiszűrji az absztakt nodeokat, melyeknek a neve nem "LiteralExpression", majd azokat, melyeknek a neve tartalmazza a "NonComputedName" szövegrészletet. Azért szűrünk kifejezetten NonComputedNamere, mert a node párosokból legalább az egyikre szükség van hogy helyesen lehessen legenerálni a switchet. Ezekben az esetekben a node nevének végéről levágjuk a "ComputedName" szövegrészletet és úgy illesztődik be a fájlokba.

Az egyes nodeokhoz tartozó fájlok legenerálása sokkal több szűréssel és feltétellel jár. Bár legtöbb fájl azonos sablon alapján lesz felépítve, szelekt fájlok, mint a Program és a Literal statikusabb módon épülnek fel, emellett a "Computed" és "NonComputed" nodeoknak 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 nodeokat, az előző funkciókhoz hasonló módon. Adott, minden node esetén megjelenő tulajdonság kiiratása után le kell generálnunk a first visit-be tartozó tulajdonságok settereit, és az él metódusok settereit. Ezt két külön funkcióval valósítottam meg, mindkettőnek paraméterként megadva a jelenlegi fájl generálásához felhasznált node. Ezekkel a funkciókkal bejárjuk az adott nodeoknak a first visit attribútumait, és az edge settereit és, mint eddig nevek alapján tettük, tulajdonság nevekre és típusokra szűrve írattatom ki a megfelelő settereket. First visit attribútumok esetén, mivel adott nodeoknál más típusú attribútumok fordulhatnak elő, vagy szituációtól függően máshogy kell beállítani az attribútumokat, ezeket külön külön kiszűrjük futásidő folyamán.

[image?]

Él setterek esetén csak két esetre szükséges figyelnünk. Mivel minden edge setter esetén más nodeokat kapnak a legenerált funkciók, így típusra nem szükséges figyelnünk (mivel ez már schemában le van kezelve), kizárólag a multiplicitást kell ellenőriznünk, azaz egy adott node tulajdonsághoz egy nodeot, vagy egy tömbnek a nodejait rendeljük. [kódrészlet?] [provide example]

4.3.3. Generált fájlok átmásolása

Miután ezeket a lépéseket minden szükséges nodera 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 struktúrálisan már elő van készítve másolásra, rendelkezik a megfelelő mappaszerkezettel. Korábbi fordításpl 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 az addon fájl, és ezzel együtt az újonnan létrehozott fájlok generálása végbe ment.

Ezzel a bővítéssel jelentősen elő lettek segítve a JSAN jövőbeli fejlesztései. 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 erjednek, 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

5. fejezet

Tesztelés

5.1. Változtatások tesztelésének folyamata, új tesztek felvétele

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 Pythonban megírt szkriptek segítségével futtatjuk le, melyek

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 eredmlnyek vizsgálata során a fejlesztőnek kötelessége eldönteni, hogy a kimutatott eltérések ténylegesen hibák vagy hamis pozitívak.

Ezek a tesztek minden implementált változtatás után futtatva vannak, 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.

Említendő, hogy fejlsztés során kötelességem volt az ESLintRunner által újonnan detektálható szabálysértések mindegyikéhez új tesztesetet írni. Ezeket a teszteket fordító dokumentációjában feltüntetett példákra alapoztam.

5.2. Regressziós tesztek módosítása

Ezeknek a statikus teszteknek volt egy olyan hiányossága, mely munkavégzés 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 ESLintRunnert, ahhoz a teljes SourceMeter-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.

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 célpontokhoz a kellő függőségeket.

Ezután 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 a "_all" utótaggal rendelkezők tartalmazták az adott nyelvnek összes tesztét, míg a többi, melyek program nevekkel voltak ellátva, csak az adott program tesztjét futtatta.

Adott helyeken még szükséges volt módosítani ahhoz, hogy akadásmentesen 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. Mivel nem akartam hogy egy hosszú sor, vagy egy nagy, nehezen olvasható szövegblokk legyen, nyelvenként csoportosítva, tagolva adtam meg ezeket.

Ezek után 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 (dictionaryben) 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 'Regtest_all' futtatása volt a feladat, azaz minden, nyelvhez tartozó összetett tesztet töröltünk az 'all_tests'-ből.

5.3. Tesztfolyamatok felgyorsítása grafikus interfésszel

Az előbb említett tesztelési célpontok bővítése és korábbi tesztelések és fejlesztések folyamán sok olyan folyamat volt, melyeket gyakran kellett megismételni. Tesztelések folyamán adott hibák keresése vagy javítása esetén például többször kellett a szükséges tesztelési célpontot lefordítani. Ez adott funkciók fejlesztése esetén többször járt terminálba több parancs írásával is. Ezeknek a lépéseknek folyamatos elvégzése időigényes, emellett az adott szituációkban hajlamos lehet a fejlesztő a használt parancsok félreírására és egyéb hibák elvétésére, mely tovább lassítja a munkafolyamatot. Ennek a problémának enyhítésére egy olyan programot készítettem egy programot, mellyel több lépéses folyamatokat egy futtatható konfigurációba lehet sűríteni.

A cliGlasses egy Python nyelven írt grafikus interfész, melyben futtatási konfigurációkat hozhatunk létre, hozzájuk tartozó részleteket könnyen módosíthatjuk és futtathatjuk.

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 10. .....

aláírás

Köszönetnyílvánítás

Morbi sem. Nulla facilisi. Vestibulum ante ipsum primis in faucibus orci luctus et ultrices posuere cubilia Curae; Nulla facilisi. Morbi sagittis ultrices libero. Praesent eu ligula sed sapien auctor sagittis. Class aptent taciti sociosqu ad litora torquent per conubia nostra, per inceptos hymenaeos. Donec vel nunc. Nunc fermentum, lacus id aliquam porta, dui tortor euismod eros, vel molestie ipsum purus eu lacus. Vivamus pede arcu, euismod ac, tempus id, pretium et, lacus. Curabitur sodales dapibus urna. Nunc eu sapien. Donec eget nunc a pede dictum pretium. Proin mauris. Vivamus luctus libero vel nibh.

Irodalomjegyzék