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

Statikus és dinamikus analízis JavaScript-környezetben

Készítette
Lucz Tamás Soma

Konzulens
Honfi Dávid

2016

TARTALOMJEGYZÉK

Kivonat

Szoftvereink kódját emberek írják. Az emberek természetes tulajdonsága, hogy hibákat követnek el, amik a megfelelő eszköztárak hiányában felfedezetlenek maradhatnak. Ezen fejlesztői hibák fokozott kockázatot jelenthetnek a készülő szoftverre, hiszen a logikailag esetlegesen helytelen működés mellett jelentős biztonsági réseket eredményezhetnek; kiaknázásuk a szoftver nemkívánatos viselkedését idézheti elő. Ez rosszindulatú támadóknak lehetőséget nyújt arra, hogy a szoftvert számukra kedvező, a fejlesztők számára kedvezőtlen módon, de mindenképpen a szándékolttól eltérő módon futtassák.

Feladatom volt a félév során, hogy a fenti szempontokat figyelembe véve egy olyan komplex analízis-eszköztár kifejlesztésének elméleti és gyakorlati lehetőségeit vizsgáljam, amely vállalati JavaScript-kódtárak elemzésével fejlesztői hibák jelenlétére hívja fel a figyelmet, csökkenteni igyekezvén ezzel a készülő szoftverbe kerülő biztonsági kockázatokat előidéző hibák számát.

Ennek első lépéseként megismerkedtem a forráskódanalízis általános fogalmaival, valamint a JavaScript-forráskódok statikus és dinamikus elemzésének lehetőségeivel. A nyelv különféle változatainak, szabványainak mélyebb megismerése után konkrét, kurrens technológiai eszközöket kerestem, amelyek lehetővé teszik egy testreszabható, automatizált munkafolyamat létrehozását a fentebb közölt probléma megoldására.

Beszámolómban a forráskódanalízis általános módszereinek áttekintése után betekintést nyújtok a JavaScript programozási nyelv sajátosságaiba, majd egy rövid történeti kitekintés és a nyelv szabványosításának bemutatása után ismertetem a JavaScript-specifikus kódanalízis módszereinek egy részhalmazát.

Ezek után bemutatom a félév során általam megismert eszközök főbb működésmódjait, jellegzetességeit, a legtöbb eszköztár funkcionális lehetőségeit saját példán illusztrálva.

Összefoglalásként felvázolok egy, a korábban bemutatott eszközökre támaszkodó, azokat összekapcsoltan, moduláris és bővíthető munkafolyamatban használó hibrid analízist, melynek eredménye egy JavaScript-szoftver komplex, testreszabható analitikai áttekintése. A munkafolyamatra építendő IDE-plugin konkrét fejlesztői hibákat lesz képes feltárni, jelentős mértékben lecsökkentve ezzel az éles környezetbe kikerülő szoftver használatának biztonsági kockázatát.

1. Bevezetés

1.1. Forráskódanalízis és motivációi

Szoftvereink kódját emberek írják. Az emberek természetes tulajdonsága, hogy hibákat követnek el, amik a megfelelő eszköztárak hiányában felfedezetlenek maradhatnak. Ezen fejlesztői hibák fokozott kockázatot jelenthetnek a készülő szoftverre, hiszen a logikailag esetlegesen helytelen működés mellett jelentős biztonsági réseket eredményezhetnek; kiaknázásuk a szoftver nemkívánatos viselkedését idézheti elő. Ez rosszindulatú támadóknak lehetőséget nyújt arra, hogy a szoftvert számukra kedvező, a fejlesztők számára kedvezőtlen módon, de mindenképpen a szándékolttól eltérő módon futtassák.

A forráskódanalízis módszertanának kidolgozása mögött elsődleges motivációként áll, hogy fejlesztői hibákat még a futtatási idejű tesztelés folyamatának megkezdése előtt, vagyis a fejlesztési folyamat közben – akár a kód írásának idejében, valós időben – fedezzünk fel, és figyelmeztessük a kód készítőjét a hibák jelenlétére.

Amennyiben a fentiekre lehetőségünk nyílik, szeretnénk az elkövetett hibákat minél teljesebben feltárni, ezáltal minimalizálni a szoftverbe kerülö biztonsági kockázatokat, „bug”-okat. Azonban a forráskódanalízis önmagában csak egy ún. „best effort” tevékenység, vagyis helyesség/teljesség nem feltétlenül követelménye: a formális verifikáció foglalkozik az egyes szoftvertulajdonságok matematikai bizonyításával.

Mivel a hibák hiányát kézi teszteléssel nehéz feltárni, automatizált munkafolyamatra van szükség. Ez a folyamat szükségszerűen determinisztikus: az egyes ugyanolyan paraméterekkel, azonos kódon készített eredmények megegyeznek.

A továbbiakban a forráskódanalízis fogalmat egy automatizált számítógépes eszköz által végzett analízisként használom.[1]

1.2. Statikus analízis

Statikus forráskódanalízis során a kód által reprezentált szoftvert nem futtatjuk. A forráskódot, mint absztrakt entitást értelmezzük, és ennek során próbálunk megadott szabályok alapján következtetéseket levonni.

A forráskódot szinte minden esetben különféle matematikai eszközökkel (pl. fák, gráfok) vizsgáljuk. Ennek követelménye, hogy a forráskód által reprezentált programot matematikailag értelmezhető struktúrákba transzformáljuk. A transzformációnak egyértelműnek és helyesnek kell lennie, hiszen helytelen strukturális reprezentációval az analízis eredménye elméletileg sem lehet helyes.

A következőkben bemutatott három struktúra terjedt el széleskörűen statikus analízisek során.[2]

[1] vö. humán analízis, forráskódértelmezés, code review, software walkthrough

[2] Elsősorban meglévő eszközökre, eszköztárakra hagyatkoztam a félév során, nem volt feladatom saját, a bevált formáktól eltérő forráskód-reprezentációt kidolgozni.

1.2.1. Absztrakt szintaxisfa (Abstract Syntax Tree, AST)

Az absztrakt szintaxisfa (a továbbiakban: AST) a forráskód absztrakt szintaktikai struktúrájának fa-alapú reprezentációja. A fa minden eleme szükségszerűen egy, a forráskódban megjelenő elemet jelöl.

A forráskód a program logikai szerkezetének szempontjából egértelműen megfeleltethető egy AST-nek, és viszont. Tehát a forráskód–AST-transzformáció, illetve az AST–forráskód-transzformáció a reprezentációk programlogikai szempontból egyértelmű egymásba alakítása.

A fa abban az értelemben absztrakt, hogy nem veszi figyelembe a forráskód minden egyes részletét: pl. blokkokat csoportosító kapcsos zárójelek a program logikai struktúrájában nem játszanak szerepet, így nem kell, hogy feltétlenül szerepeljenek a fában.[3]

Az AST többek között a program szintaktikai szempontból történő ellenőrzését teszi lehetővé, ezzel a statikus analízisben kiemelt szerepet játszik.

1.2.2. Absztrakt szemantikus gráf (Abstract Semantic Graph, ASG)

Az absztrakt szemantikus gráf (a továbbiakban: ASG) a forráskód absztrakt szintaktikai struktúrájának az AST-nél egy magasabb absztrakciós szinten történő reprezentációja.

> ***Definíció [term]:*** *Azon szimbólumokat, melyeket konstansokból, változókból, vagy függvényekből állítunk elő,* ***termeknek*** *nevezzük.*

Az ASG a forráskódot egy kifejezésként ábrázolja, csúcsai a kifejezés résztermjei. Általában irányított körmentes gráf (DAG), ha kört tartalmaz, az pl. rekurziót jelenthet.

1.2.3. Vezérlésfolyam-gráf (Control-Flow Graph, CFG)

A vezérlésfolyam-gráf (a továbbiakban: CFG) a forráskód absztrakt reprezentációja gráf formában. Tartalmazza a program összes lefutási útvonalát.

> ***Definíció [vezérlésfolyam-blokk]:*** *Azon kódrészletet, amely nem tartalmaz ugrást vagy elágazást,* ***vezérlésfolyam-blokknak*** *nevezzük.*

A gráf minden csúcsa egy vezérlésfolyam-blokk, a gráf irányított élei a blokkok közötti vezérlésfolyamot reprezentálják. Az ún. belépési blokk a gráfba belépő vezérlésfolyam belépési pontja, az ún. kilépési blokk pedig a gráfot elhagyó vezérlésfolyam helye.

CFG-k használata statikus analízisek terén igen elterjedt. Többek között elérhetőségi problémákra nyújt megoldást: pl. ha egy részgráf belépési pontjának nincs bemenő éle, a részgráf elérhetetlen kódrészletet reprezentál; ha egy kilépési blokk nem elérhető a belépési blokkból, az végtelen ciklust jelenthet.

---

[3] vö. Concrete Syntax Tree (CST): A hagyományosan Parse Tree-nek is nevezett, tipikusan fordítók által készített reprezentáció a forráskód minden egyes elemét – a whitespace-ektől eltekintve – egyértelműen reprezentálja, a forráskód–CST-transzformáció után a CST–forráskód-transzformációval az eredetivel pontosan egyező kódot kapunk vissza.

1.3. Dinamikus kódanalízis

Dinamikus analízis során a programot futtatjuk, és a futtatás során végbemenő viselkedést vizsgáljuk. Ennek követelménye, hogy a program futtatható (tehát fordított nyelvek esetén fordítható) legyen. A dinamikus analízisra tehát úgy tekinthetünk, mint egy statikus analízis után végzett „második lépés”, mellyel más típusú, futtatási idejű teszteredményeket kaphatunk.

Dinamikus tesztelésnél fontos szerepet játszik, hogy milyen bemeneti paraméterekkel, inputokkal futtatjuk a programot, hiszen más inputokkal más viselkedést produkálhat a vizsgált szoftver.

> ***Definíció [instrumentáció]:*** *Azt a tevékenységet, melynek során egy forráskódot vagy programot olyan formába alakítunk, hogy a futtatás során számunkra fontos tulajdonságait vizsgálni tudjuk,* ***instrumentációnak*** *nevezzük.*

Rendkívül fontos továbbá a kód instrumentációja során keletkező mellékhatások minimalizálása. Az instrumentáció semmiképpen nem változtathatja meg a kód logikai működését, de egyéb futtatási tulajdonságok (pl. futási idő, memóriahasználat) szempontjából is legfeljebb elhanyagolható mértékben befolyásolhatja a programot.

Dinamikus analízist széleskörűen alkalmaznak a szoftverfejlesztés különféle szintjein, ezt az 1.1. táblázat ábrázolja.

Analízis szintje	Leírás
Egységtesztelés	A kód logikailag legkisebb logikai egységeinek (tipikusan osztályok) egyedileg végzett tesztelése.
Integrációs tesztelés	Az önálló szoftveregységek (tipikusan osztályok) együttes tesztelése, melynek során az egységek egymás, és a rendszer felé nyújtott interfészeit vizsgálják.
Rendszertesztelés	A kész szoftvertermék tesztelése követelmények, funkcionális specifikáció, rendszerterv szempontjából.

1.1. táblázat. Dinamikus analízis alkalmazása különféle szinteken

1.4. Hibrid analízis

Hibrid analízis során egyszerre statikus és dinamikus eszközökkel is vizsgáljuk a kérdéses szoftvert. Ennek előnye, hogy a két analízistípus eredményeit egymással kölcsönhatásban is tudjuk értelmezni: lehetőségünk nyílik tehát statikusan, illetve dinamikusan önmagában nem elvégezhető analízisek megismerésére is, további rejtőzködő szoftverhibákat tárva fel ezzel.

Egy számomra igen ígéretes hibrid analízis-irány a dinamikus anomáliakeresés statikus predikciók alapján. Egy statikus eszközökkel kinyert vezérlésfolyam-gráf segítségével a program dinamikus analízise során pl. rendellenes lefutási útvonalakat detektálhatunk.

2. A JavaScript-nyelv és kódanalízise

2.1. Sajátosságok

A JavaScript egy magasszintű, dinamikus, dinamikusan és gyengén típusos, interpretált programozási nyelv. Szkriptnyelv.

2.1.1. Dinamikusan és gyengén típusos nyelv

A legtöbb szkriptnyelvhez hasonlóan a JavaScript is dinamikusan típusos. Emellett gyengén típusos is: a típusok nem kifejezésekhez, hanem értékekhez kötöttek. Egy pl. integer típusú változó típusa futásidőben is módosítható pl. string típusúra implicit típuskonverzióval.

Mindez azt jelenti, hogy nem áll rendelkezésünkre fordítási idejű típusellenőrzés, hiszen sem fordítási idő, sem explicit típusellenőrzés nincsen.

2.1.2. Futásidejű kódkiértékelés

A nyelv lehetőséget ad arra, hogy futásidőben „futtassunk” kódot az `eval()` függvény segítségével. A függvény inputja egy string, amely – ha a string értelmezhető JavaScript-kódot tartalmaz – futtatásra kerül.

2.2. Történelem és kitekintés

2.2.1. Kezdetek

A JavaScript nyelvet 1995 tavaszán kb. 10 nap alatt fejlesztette ki a Netscape Communications Corporation egy mérnöke. Sokáig a böngészők kliens-oldali nyelveként tartották számon, de miután a Google publikálta a Chrome böngészőjéhez tartozó V8-motort, robbanásszerű terjedésnek indult böngészőkön kívül is. A V8 azzal alakította át gyökeresen az addigi JavaScript-szcénát, hogy nem csak interpretálja és nem csak bájt-kódra fordítja a forrást, hanem natív gépi kódot képes gyártani a program egyes részeiből.

2.2.2. A futtatókörnyezet kiemelése a böngészőkből

2009-ben a Joyent szoftverfejlesztőcég egy mérnökének ötlete nyomán létrejött egy, a Google-féle V8-motor alapján kifejlesztett natív JavaScript futtatókörnyezet. Ebből nőtte ki magát később a node.js nyílt „platform”, kiegészülve a saját csomagkezelőjével, és példaértékűen aktív közösségi támogatással.

A node.js-alapú JavaScript-technológiák újnak és nagyvállalati környezettel teljesen inkompatibilisnek[4] számítanak. Mindezek ellenére rugalmassága miatt ma már széleskörűen alkalmazzák vállalati környezetben is: az IBM, a General Electric, a Walmart, a PayPal és a LinkedIn is az „nyelv”, platform aktív felhasználói között van. Ez nyilvánvalóan tovább erősítette a JavaScript szabványosítására vonatkozó igényeket.

[4] A nyelvből többek között teljesen hiányzik az interfészek használata.

2.3. Szabványok

A JavaScript szabványosítására irányuló törekvések 1996 novemberében kezdődtek el az Ecma International szabványtestület által. A munka kódszámára – ECMA-262 – azóta is sokan hivatkoznak, mint „a szabványosított JavaScript”. A nyelvet az ISO/IEC is szabványosította ISO/IEC 16262 kódszám alatt.

Alább látható a 2.1. táblázat, amelyben összefoglalom az ECMAScript fejlődéstörténetét.

Verzió	Publikálás éve	A szabvány fontosabb elemei
1	1997	A nyelv első szabványosított kiadása
2	1998	Kisebb módosítások az ISO/IEC-szabvány érdekében
3	1999	Try-catch-típusú kivételkezelés, stringek kényelmesebb kezelése, reguláris kifejezések
4	—	*Nem lett kiadva*
5	2009	Reflection, strict mode, JSON-támogatás
5.1	**2011**	**Teljes egyeztetés az ISO/IEC-szabvánnyal**
6	2015	Új szintaxiselemek (osztályok és modulok), iterátorok, generátorok, kollekciók, teljes reflection
7	—	*Fejlesztés alatt*

2.1. táblázat. Az ECMAScript fejlődéstörténete

Jelenleg a táblázatban félkövéren jelölt 5.1-es a legelterjedtebb JavaScript-verzió, a legtöbb böngésző ezt támogatja. A továbbiakban a *JavaScript/plain JavaScript/JS* kifejezéssel erre fogok hivatkozni.

2.4. Átjárás a szabványok között

Az egyes JavaScript-szabványok közötti átjárás ma leginkább az ECMAScript 6 (a továbbiakban: ES6) és a plain JavaScript közötti átjárásra korlátozódik.

Az ES6 a fejlesztők körében kényelmes, új szintaxisa miatt hamar elterjedt, általános kliens-oldali támogatottsága ma még azonban nem létezik. A probléma megoldására jött létre az ún. transpiling fogalma, melynek során ES6-ot JS-re „fordítanak” vissza.

Definíció [transpiling]: *Olyan kódfordítási folyamat, melynek kimenete forráskód.*

A mai elterjedt ECMAScript- és JavaScript-transpilerek már nem csak egy funkciót látnak el: egész eszköztárak, keretrendszerek épültek rájuk, melyeket saját, általunk fejlesztett plugineken kívül az egész JavaScript-közösség által fejlesztett és elérhetővé tett kiegészítőkkel is bővíthetünk, az ES6–JS-transpiling folyamatot jelentősen kiterjesztve ezzel.

Látható, hogy a funkciók keveredésével egyáltalán nem egyértelmű, hogy melyik eszköztárnak mi a pontos feladata. E dolgozatnak nem felelőssége eligazodni az egyes ECMAScript-verziók közötti fordítók, illetve eszköztárak között, a témakör kódanalízis szempontjából azonban nagy jelentőséggel bír.

2.5. Analízis-eszköztárak támogatottsága a JavaScript-szcénán belül

A hozzáférhető analízis-eszköztárak száma az egyes transpilerek és kiegészítőik számához hasonlóan hatalmas. A JavaScript utóbbi években történt jelentős ívű felfutása miatt rengeteg eszköz jelent meg az interneten; ezek többsége alacsony minőséget és nem kiemelkedő funkciókat biztosít, azonban vannak remekül használható eszköztárak is.

2.5.1. Statikus eszköztárak

A fejezet eddigi olvasatából nyilvánvalóan tükröződik, hogy a JavaScript dinamikus nyelv. A dinamikus és gyenge típusosság, a fordítási idejű típusellenőrzés hiánya, valamint a futási időben történő tetszőleges kód „futtatásának" lehetősége felveti a kérdést, hogy érdemes-e egyáltalán statikus analízist végezni JavaScript-kódon.

A fentiek ellenére a JavaScript jelentős számú statikus analízis eszköztárral rendelkezik. A fejlesztést közvetlenül támogató eszközök – szintaktikai szabályellenőrzők, típusellenőrzők – mellett nagyszámú végletekig optimalizált AST-eszköztár és CFG-transzformátor áll rendelkezésünkre. Ez utóbbiakra saját analízis-keretrendszert építhetünk, melynek segítségével tetszőleges statikus analízisre lehetőségünk nyílik.

2.6. AST-generálás, bejárás és manipuláció

AST generálásával létrejön a programunk egy absztrakt reprezentációja. Álljon itt egy rendkívül egyszerű JavaScript-kód:

```
function z() {
      return 2;
}

z();
```

E fenti, 4 soros kódunk JSON-formátumban megjelenített AST-je 106 sor, ez látható az **A) függelék**ben. Az AST kinyeréséhez az Esprima nevű külső eszköztárat használtam, ennek használatára később részletesen ki fogok térni.

Az AST feldolgozása után a legtöbb eszköztár egyszerűen biztosítja, hogy a fa különböző, analízisünk szempontjából nem jelentős tulajdonságaira szűrőfeltételeket állítsunk fel, eliminálva az így keletkező AST-ből a felesleges adatokat.

A szűrés utáni validáció biztosítja azt, hogy a kinyert AST egy érvényes ECMAScript- vagy JavaScript-programot reprezentál, tehát egyértelműen megfeleltethető egy olyan programnak, amit le lehet futtatni a verziójának megfelelő futtatókörnyezetben. Ez az analízisünk minden fázisában jelentős lépés, hiszen így bizonyosodhatunk meg arról, hogy a manipulációs műveleteink során nem sérült az eredeti reprezentációnk.

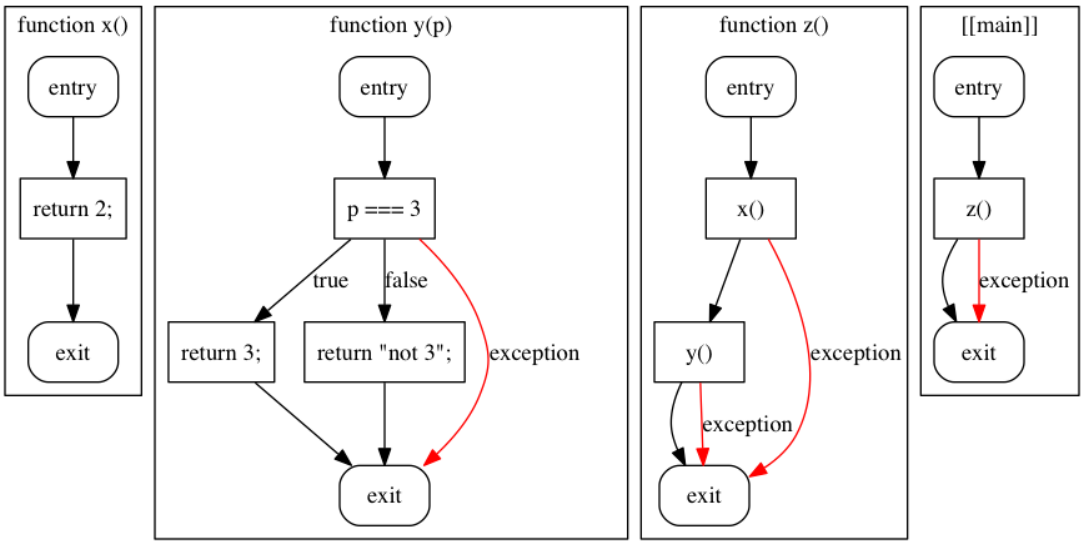
Ezek után a fa bejárásával végezhetünk effektív analízist a forráskódunkon. Egy AST bejárása során nyerhetünk többek között: információt arról, hogy az egyes kifejezések hol kezdődnek, és hol érnek véget, valamint milyen logika szerint vannak csoportosítva; átfogó képet a kód általános felépítéséről; egy olyan struktúrát, amelynek segítségével a programunk kódja a kód konkrét string-reprezentációja nélkül transzformálható.

2.7. CFG-generálás

A legtöbb elérhető eszköztár AST-transzformáció segítségével készít CFG-t. Tekintsük az alábbi, szintén nagyon egyszerű JavaScript-kódot:

```
function x() {
	return 2;
}

function y(p) {
	if (p === 3) {
		return 3;
	}

	return "not 3";
}

function z() {
	x();
	y();
}

z();
```

A fenti kód külső eszköztár által kinyert CFG-jének egy ábrázolása a 2.1 ábrán látható. A CFG amellett, hogy ad egy szemléletes áttekintést a program lehetséges lefutási útvonalairól, absztrakt formában lehetőséget ad többet között olyan, korábban már tárgyalt kódtulajdonságok detektálására, mint az elérhetetlen kód, vagy bizonyos esetekben a végtelen ciklus.



2.1. ábra. Általam generált Control-Flow Graph a fenti példakód alapján.

2.8. Kódinstrumentáció

Dinamikus analízis során a kódot fel kell készítenünk arra, hogy futás közben vizsgáljuk: el kell látnunk olyan kódtulajdonságokkal, amely lehetővé teszi az analízis számára, hogy a számunkra érdekes kimenetet tudja produkálni. JavaScriptben ez tipikusan callbackek segítségével szokott történni.

Tekintsük az alábbi, korábban már vizsgált egyszerű kódunkat:

```
function x() {
	return 2;
}

function y(p) {
	if (p === 3) {
		return 3;
	}

	return "not 3";
}

function z() {
	x();
	y();
}

z();
```

Egy később részletesen ismertetett keretrendszer, a Jalangi2 a **B) függelék**ben látható instrumentált kódot produkálja.

A kód vizsgálata során látható, hogy a keretrendszer ún. labelekkel, illetve azonosítókkal írja tele a kódunkat az instrumentáció során: ezen labelek és azonosítók teszik lehetővé az egyes hívási helyek beazonosítását. A labeleken kívül becsomagolt, „wrapelt” függvényhívásokat látunk: ez alapján történik a különféle viselkedéstípusok (pl. függvényhívás, elágazás) identifikációja.

3. Összegyűjtött eszközök

3.1. Fejlesztést közvetlenül támogató eszközök

3.1.1. TAJS: Type Analysis for JavaScript

A TAJS egy, a dán Aarhus University-n kifejlesztett statikus analízist használó típusellenőrző eszköztár, amely részletes és helyes típuskövetkeztetéseket képes végezni tetszőleges, ECMAScript-sztenderdnek megfelelő programon.

Az eszköztár a változók kezdeti értékei, valamint egy vezérlésfolyam-gráf alapján követi a típusokat, és implicit típuskonverziók vagy típus szerint helytelen változóérték-ellenőrzés esetén figyelmeztet.

3.1.2. Flow

E facebook által fejlesztett eszköz működési elve hasonló a TAJS-hez, azonban fejlesztői annotációkkal explicit típuskövetelések kikényszerítése is lehetséges.

Az alábbi egy kipróbált példa az eszköz weboldaláról.

```
// @flow
function bar(x: string, y: number): string {
      return x.length * y;
}
bar('Hello', 42);
```

A kódot keresztülfuttatva az eszközön, a következő üzenetet kapjuk:

```
$> flow
  3:   return x.length * y;
              ^^^^^^^^^^^^ number. This type is incompatible with
  2:   function bar(x: string, y: number): string {
                                           ^^^^^^ string
```

A hibaüzenet érthető, hiszen mi explicit módon stringet követelünk visszatérési értékként, azonban a függvény számmal tér vissza a példánkban.

3.1.3. Jest

Ezen szintén facebook által fejlesztett eszköz egy egységtesztelési keretrendszer. Moduláris és bővíthető, tetszőleges assertion library-kkel lehet használni.

Legfontosabb tulajdonsága az ún. automock-funkciója: a keretrendszer automatikusan felfedezi és – egy saját, a Java Reflection API-jához hasonlatos módszer implementációjával[5] – kimockolja az éppen tesztelt egység/osztály/unit összes függőségét, így fejlesztőként nem szükséges azzal foglalkoznunk, hogy manuálisan leválasszuk a függőségeket a tesztelt egységről. Az automatikus mockoláson kívül lehetőségünk van arra is, hogy manuálisan adjuk meg az egyes függőségek mockjait.

[5] Valójában a háttérben a modulok importálásához használt `require()` függvény felüldefiniálásáról van szó: e során történik az importált modul vizsgálata és mockja.

3.1.4. Istanbul

A San Francisco öbölnegyedéből érkező eszköztár lefedettségi tesztek végzésére alkalmas. A kód instrumentálása után futtat, majd a saját maga által definiált outputokat figyelve képes megmondani azt, hogy a kódunk mekkora része futott le a tesztesetek során. Szöveg-/HTML-riportot is képes exportálni, de CLI-környezetben is használható.

3.2. Statikus analízis eszközök

A megismert eszközök második csoportja az AST-manipulációs eszközöket foglalja magába. Alapvetően a következő műveletekre van szükségünk egy program AST-alapú statikus analízise során: a fa felépítése; szűrés; validálás; részfák lekérdezése; bejárás.

A JavaScript-szcénában kétféle AST-család terjedt el. A korábban Mozilla kezei alatt lévő **EStree** (régen: SpiderMonkey AST) szélesebb körben használt és régebb óta van jelen, de lehetőségei korlátozottabbak, mint a Shape Security által fejlesztett **Shift AST**-éi, amely bővebb eszköztárával minimalizálni igyekszik azon AST-k előfordulásának lehetőségét, amelyek nem érvényes ECMAScript-programot reprezentálnak.

Mindkét család kiterjedt eszköztárral rendelkezik.

3.2.1. EStools (EStree-formátumú AST-khez)

A következő eszközök mindegyikét ES6-nyelven implementálták.

Az **Esprima** nagy teljesítményű, ES6-kompatibilis kódértelmező eszköztár. Bemenete tetszőleges JavaScript-kód, kimenete egy EStree-formátumú AST.

Az **espurify** AST-szűrő, amellyel tetszőleges AST-ben szereplő tulajonság elhagyását teszi lehetővé, így csak a számunkra érdekes tulajdonságok maradnak meg az analízis során. Testreszabható, a szűrendő feltételek blacklist- és whitelist-formában is megadhatóak.

Az **esvalid** biztosít minket arról, hogy az inputként beadott faelem által reprezentált részfa valódi, érvényes ECMAScript-programot/-programrészletet reprezentál.

Az **estraverse** segítségével bejárhatunk tetszőleges EStree-formátumú AST-t. Íme egy rövid példa, amely szemlélteti a használatát. AST-node-ba belépés esetén logolunk:

```
estraverse.traverse(ast, {
      enter: function (node, parent) {
            if (node.type == 'FunctionDeclaration')
                  console.log('Function declaration: ' + node.id.name);
      }
});
```

Az **escope** scope-vizsgáló eszköztár, amellyel a fabejárás során a változók, illetve függvények scope-jait tudjuk kinyerni, illetve állítani.

Az **esquery** segítségével CSS-szerű lekérdezéseket írhatunk AST-fákra, illetve részfákra, rendkívül kényelmessé téve ezzel közel tetszőleges tulajdonságú részfa lekérdezését.

Az **esdispatch** lehetővé teszi, hogy az AST bejárása során eseményvezérelt viselkedést határozzunk meg az analízisünkben.

3.2.2. Shift (Shift-formátumú AST-khez)

A **Shift Parser** kódértelmező eszköztár, amelynek bemenete ECMAScript-forráskód, kimenete pedig Shift-formátumú AST. Automatikusan kiszűri azon AST-ket, amelyek nem feleltethetőek meg érvényes JavaScript-programnak.

A **Shift Scope Analyser** dinamikus scope-analízis eszköztár, amely a vizsgált program összes scope-információját képes egyszerre kinyerni a scope típusával, AST-node-jával, és scope-on belül deklarált változóival együtt.

A **Shift Validator** validál egy korábban kinyert AST-t.

3.2.3. Átjárás a két AST-család között

Létezik konverzió, azonban mivel a Shift-család információtartalomban jóval bővebb AST-t képes értelmezni, így csak Shift-AST-ből tudunk EStree-re konvertálni, visszafele nem lehetséges az átalakítás.

3.3. Egy dinamikus analízis eszköztár: Jalangi2

A Samsung által forkolt, majd továbbfejlesztett Jalangi2 egy JavaScript-framework dinamikus analízisek írásához. A példaként előre definiált analízisek között találunk többet között NaN-ellenőrzést, valamint undefined-string konkatenáció-ellenőrzést.

3.3.1. A keretrendszer működése fejlesztői szemszögből

Amikor Jalangi2-analízist írunk, akkor a framework által meghatározott események callbackjeire írunk le valamilyen viselkedést. A Jalangi2 a kódunk futtatása során minden esemény bekövetkezésekor meghívja az eseményhez tartozó callbacket, így az általunk definiált viselkedés érvényre jut. A callbackekben átadott paraméterek segítségével változatos adatokat tudunk megszerezni az adott eseményről: pl. függvénybe belépés esetén a függvény adatait, változódeklaráció esetén a változó adatait, stb.

A Jalangi2 analysisCallbackTemplate.js fájlja tartalmazza az összes, keretrendszer által biztosított esemény-callbacket. Többek között:

- **declare:** függvény, illetve változódeklarálás esetén,
- **invokeFunPre:** függvény meghívása esetén,
- **functionEnter:** függvénytörzsbe belépéskor, a törzs futtatása előtt,
- **unary:** egyoperandusos művelet végrehajtása esetén,
- **conditional:** feltétel ellenőrzése esetén, még az elágazás előtt,
- **forinObject:** for-in ciklus objektumtulajdonságokon történő iterálása esetén,
- **literal:** literál létrehozása esetén,
- **read:** változó olvasása esetén,
- **getField:** tagváltozó olvasása esetén,
- **getFieldPre:** tagváltozó olvasása előtt,
- **write:** változó írása esetén,
- **_throw:** érték throw-val történő eldobása előtt,
- stb.

3.3.2. Egy példaanalízis

A következőben meghatározott kódra lefuttattam egy analízist Jalangi2 segítségével.

```
function foo(){
	console.log("foo");
}

function bar(){
	console.log("bar");
}

for (var i = 0; i < 10; i++){
	if (i%2 === 0) {
		foo();
	} else {
		bar();
	}
}
console.log("done");
```

A konkrét analízis kódja a **C) függelék**ben látható. Az analízis kimenete a **D) függelék**ben látható. Az, hogy logoljuk az egyes hívási eseményeket, természetesen messze nem meríti ki a rendszer lehetőségeit.

3.3.3. Anaílzis futtatásának módjai

A Jalangi2 keretrendszerben többféle lehetőségünk van analízisek futtatására.

Amennyiben instrumentálunk és analizálunk, a Jalangi2 csak a számunkra releváns outputot adja ki, a belső működést elfedve előlünk.

Tudunk explicit instrumentálni, majd explicit futtatni. Ekkor a Jalangi2 felinstrumentálja a kódunkat a keretrendszeren belüli callbackekkel, és mi hozzáférünk az instrumentált kódhoz még az analízis futtatása előtt. Célszerű lehet így tenni, ha további manipulációkat, instrumentációt szeretnénk végrehajtani a kódunkon, még a Jalangi2-nek történő analízisvezérlés átadása előtt.

Böngészőn keresztüli analízis esetén a programunk böngészőben történő futtatása során egy meghatározott billentyűkombinációra a böngésző JavaScript-konzolára íródik ki az eredmény. Hasznos, ha sok felhasználói interakciót igénylő programot vizsgálunk.

Proxyn keresztüli analízis esetén lehetőségünk nyílik arra is, hogy on-the-fly instrumentáljunk fel JavaScript-kódokat, mielőtt átadnánk őket a folyamatosan futó Jalangi2-analízisnek. Felhő-alapú kódellenőrző-szolgáltatásoknál megfontolandó lehet, hiszen nagyon sok terhet levesz a vállunkról az, hogy van kész, megbízható, proxyn keresztül működő hálózati implementációja a funkciónak.

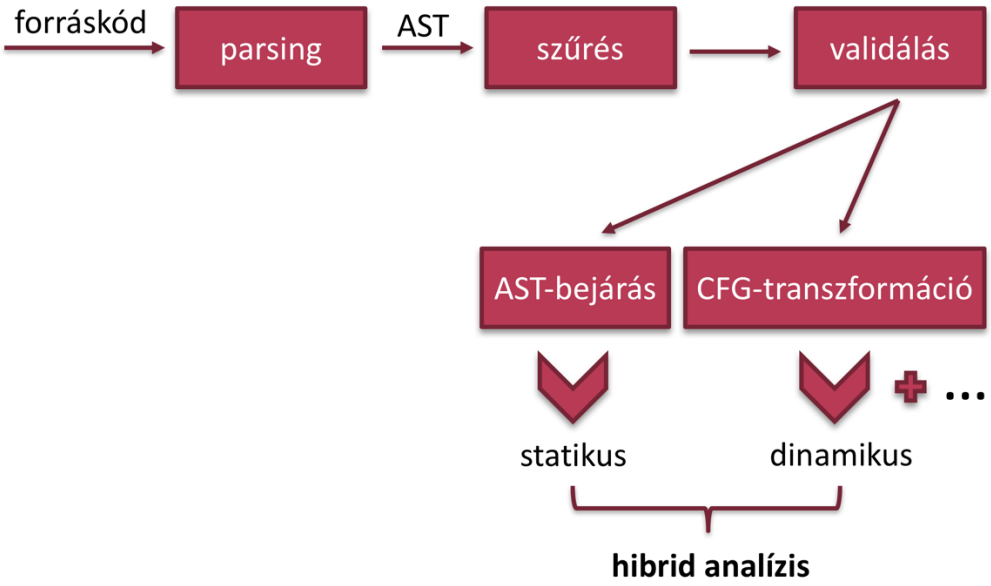
3.3.4. Támogatottság

A Jalangi2 jelenleg csak az ECMAScript 5.1-es változatával működőképes. Amennyiben mindenképp szeretnénk ES6-kóddal működésre bírni, tehetünk kísérletet arra, hogy egy transpilert integrálunk az analízis folyamatába. Én ezen az úton még nem jártam sikerrel, de a jövőben biztosan szükséges lesz ebben az irányban is eredményt elérni.

4. Összefoglalás és további lehetőségek

4.1. Egy lehetséges hibrid munkafolyamat

A megismert eszközök segítségével a következő hibrid analízis-munkafolyamat lehetőségét vázolom fel:



4.1. ábra. Egy lehetséges hibrid analízis-munkafolyamat

A forráskód értelmezése, a kapott AST szűrése és validálása után bejárjuk az AST-t, statikus analízis-eszközökkel vizsgáljuk a kódunkat, és ezzel szimultán a Jalangi2 segítségével dinamikus analízist végzünk, az AST-ből transzformált CFG-t, mint statikus predikciót felhasználva anomáliakereséshez.

Jelenleg e munkafolyamat minden eleme működik külön-külön, a funkciók együttes működését lehetővé tévő implementáció pedig folyamatban van.

4.2. Együttműködési lehetőségek

4.2.1. Stein Dániel, BME-MIT

Stein Dániel BME-MIT-en diplomázó MSc-s hallgató robosztus, jól skálázódó workflow-t dolgozott ki JavaScript-kódok AST-jének kinyerésére. Munkafolyamatának outputja egyszerűen kompatibilissá tehető az enyémmel. A segítségével hatékony AST–CFG-konverziót valósíthatunk meg, amely nagymértékben előremozdítja hibrid JavaScript-analízisünk sikerét.

4.2.2. Tresorit

A Tresorittal együttműködve középtávú célunk egy IDE-plugin implementálása, ami az itt definiált eszközök segítségével fejlesztői hibákat képes feltárni valós időben.

Irodalomjegyzék

[1] Simon Holm Jensen, Anders Møller, Peter Thiemann2., *Type Analysis for JavaScript (SAS 2009)*
http://cs.au.dk/~amoeller/papers/tajs/paper.pdf

[2] ECMA International, *ECMAScript 2015 Language Specification*
http://www.ecma-international.org/ecma-262/6.0/

[3] Dr. Axel Rauschmayer, *A JavaScript glossary: ECMAScript, TC39*
http://www.2ality.com/2011/06/ecmascript.html

[4] Marc Andreesen, Netscape Communications Corporation, *Innovators of the Net: Brendan Eich and JavaScript*
https://web.archive.org/web/20080208124612/http://wp.netscape.com/comprod/columns/techvision/innovators_be.html

[5] Facebook, *Flow documentation*
http://flowtype.org

[6] Facebook, *Jest documentation*
https://facebook.github.io/jest/

[7] Liang Gong, Electric Engineering & Computer Science, University of California, Berkeley, *Jalangi2 in a nutshell*
http://people.eecs.berkeley.edu/~gongliang13/jalangi_ff/

[8] Samsung, *Jalangi2 documentation*
https://github.com/Samsung/jalangi2

[9] Shape Security, *Shift-AST family documentation*
http://shift-ast.org

[10] *ESTools documentation*
https://github.com/estools/

Függelék

A) függelék


```
{
    "type": "Program",
    "body": [
        {
            "type": "FunctionDeclaration",
            "id": {
                "type": "Identifier",
                "name": "z"
            },
            "params": [],
            "defaults": [],
            "body": {
                "type": "BlockStatement",
                "body": [
                    {
                        "type": "ReturnStatement",
                        "argument": {
                            "type": "Literal",
                            "value": 2,
                            "raw": "2"
                        }
                    }
                ]
            },
            "generator": false,
            "expression": false
        },
        {
            "type": "ExpressionStatement",
            "expression": {
                "type": "CallExpression",
                "callee": {
                    "type": "Identifier",
                    "name": "z"
                },
                "arguments": []
            }
        }
    ],
    "sourceType": "script"
}
```

B) függelék

```
J$.iids =
{"8":[6,7,6,14],"9":[2,10,2,11],"10":[6,7,6,14],"17":[2,10,2,11],"25":[2,3,2,
12],"33":[1,1,3,2],"41":[1,1,3,2],"49":[6,7,6,8],"57":[6,13,6,14],"65":[7,12,
7,13],"73":[7,12,7,13],"81":[7,5,7,14],"89":[10,10,10,17],"97":[10,10,10,17],
"105":[10,3,10,18],"113":[5,1,11,2],"121":[5,1,11,2],"129":[5,1,11,2],"137":[
14,3,14,4],"145":[14,3,14,6],"153":[14,3,14,7],"161":[15,3,15,4],"169":[15,3,
15,6],"177":[15,3,15,7],"185":[13,1,16,2],"193":[13,1,16,2],"201":[18,1,18,2]
,"209":[18,1,18,4],"217":[18,1,18,5],"225":[1,1,18,5],"233":[1,1,3,2],"241":[
1,1,18,5],"249":[5,1,11,2],"257":[1,1,18,5],"265":[13,1,16,2],"273":[1,1,18,5
],"281":[1,1,3,2],"289":[1,1,3,2],"297":[6,3,8,4],"305":[5,1,11,2],"313":[5,1
,11,2],"321":[13,1,16,2],"329":[13,1,16,2],"337":[1,1,18,5],"345":[1,1,18,5],
"nBranches":4,"originalCodeFileName":"/Users/luczsoma/projects/tresorit/modul
es/jalangi2/tests/octane/somi.js","instrumentedCodeFileName":"/Users/luczsoma
/projects/tresorit/modules/jalangi2/tests/octane/somi_jalangi_.js","code":"fu
nction x() {\n  return 2;\n}\n\nfunction y(p) {\n  if (p === 3) {\n    return
3;\n  }\n\n  return \"not 3\";\n}\n\nfunction z() {\n  x();\n
y();\n}\n\nz();"};
jalangiLabel3:
    while (true) {
        try {
            J$.Se(225,
'/Users/luczsoma/projects/tresorit/modules/jalangi2/tests/octane/somi_jalangi
_.js',
'/Users/luczsoma/projects/tresorit/modules/jalangi2/tests/octane/somi.js');
            function x() {
                jalangiLabel0:
                    while (true) {
                        try {
                            J$.Fe(33, arguments.callee, this, arguments);
                            arguments = J$.N(41, 'arguments', arguments, 4);
                            return J$.X1(25, J$.Rt(17, J$.T(9, 2, 22,
false)));
                        } catch (J$e) {
                            J$.Ex(281, J$e);
                        } finally {
                            if (J$.Fr(289))
                                continue jalangiLabel0;
                            else
                                return J$.Ra();
                        }
                    }
            }
            function y(p) {
                jalangiLabel1:
                    while (true) {
                        try {
                            J$.Fe(113, arguments.callee, this, arguments);
                            arguments = J$.N(121, 'arguments', arguments, 4);
                            p = J$.N(129, 'p', p, 4);
                            if (J$.X1(297, J$.C(8, J$.B(10, '===', J$.R(49,
'p', p, 0), J$.T(57, 3, 22, false), 0)))) {
                                return J$.X1(81, J$.Rt(73, J$.T(65, 3, 22,
false)));
                            }
                            return J$.X1(105, J$.Rt(97, J$.T(89, 'not 3', 21,
false)));
```

```
                        } catch (J$e) {
                            J$.Ex(305, J$e);
                        } finally {
                            if (J$.Fr(313))
                                continue jalangiLabel1;
                            else
                                return J$.Ra();
                        }
                    }
            }
            function z() {
                jalangiLabel2:
                    while (true) {
                        try {
                            J$.Fe(185, arguments.callee, this, arguments);
                            arguments = J$.N(193, 'arguments', arguments, 4);
                            J$.X1(153, J$.F(145, J$.R(137, 'x', x, 1), 0)());
                            J$.X1(177, J$.F(169, J$.R(161, 'y', y, 1), 0)());
                        } catch (J$e) {
                            J$.Ex(321, J$e);
                        } finally {
                            if (J$.Fr(329))
                                continue jalangiLabel2;
                            else
                                return J$.Ra();
                        }
                    }
            }
            x = J$.N(241, 'x', J$.T(233, x, 12, false, 33), 0);
            y = J$.N(257, 'y', J$.T(249, y, 12, false, 113), 0);
            z = J$.N(273, 'z', J$.T(265, z, 12, false, 185), 0);
            J$.X1(217, J$.F(209, J$.R(201, 'z', z, 1), 0)());
        } catch (J$e) {
            J$.Ex(337, J$e);
        } finally {
            if (J$.Sr(345)) {
                J$.L();
                continue jalangiLabel3;
            } else {
                J$.L();
                break jalangiLabel3;
            }
        }
    }
// JALANGI DO NOT INSTRUMENT
```

C) függelék

```
(function(){
  var branches = {};
  J$.analysis = {

    /**
     * This callback is called after a condition check before branching.
     * Branching can happen in various statements
     * including if-then-else, switch-case, while, for, ||, &&, ?:.
     *
     * @param {number} iid - Static unique instruction identifier of this
callback
     * @param {*} result - The value of the conditional expression
     * @returns {{result: *}|undefined} - If an object is returned, the
result of
     * the conditional expression is replaced with the value stored in the
     * <tt>result</tt> property of the object.
     */
    conditional : function (iid, result) {
            var id = J$.getGlobalIID(iid);
            var branchInfo = branches[id];
            if (!branchInfo) {
                branchInfo = branches[id] = {trueCount: 0, falseCount: 0};
            }
            if (result) {
                branchInfo.trueCount++;
            } else {
                branchInfo.falseCount++;
            }
        },

    invokeFunPre : function (iid, f, base, args, isConstructor, isMethod,
functionIid) {
            console.log('Called before function');
        },

    invokeFun : function (iid, f, base, args, result, isConstructor,
isMethod, functionIid) {
            console.log('Called after function');
        },


    functionEnter : function (iid, f, dis, args) {
            console.log('Called before function body starts to execute');
        },

    functionExit : function (iid, returnVal, wrappedExceptionVal) {
            console.log('Called after function body completes, before
return');
            return {returnVal: returnVal, wrappedExceptionVal:
wrappedExceptionVal, isBacktrack: false};
        },
```

```
    /**
     * This callback is called when an execution terminates in node.js.  In
a browser
     * environment, the callback is called if ChainedAnalyses.js or
ChainedAnalysesNoCheck.js
     * is used and Alt-Shift-T is pressed.
     *
     * @returns {undefined} - Any return value is ignored
     */
    endExecution : function () {
           for (var id in branches) {
               if (branches.hasOwnProperty(id)) {
                   var branchInfo = branches[id];
                   var location = J$.iidToLocation(id);
                   console.log("At location " + location +
                   " 'true' branch was taken " + branchInfo.trueCount +
                   " time(s) and 'false' branch was taken " +
branchInfo.falseCount + " time(s).");
               }
           }
       }
};

}());
```

D) függelék

```
Called before function: foo
Called before function body starts to execute: foo
Called before function:
foo
Called after function:
Called after function body completes, before return
Called after function: foo
Called before function: bar
Called before function body starts to execute: bar
Called before function:
bar
Called after function:
Called after function body completes, before return
Called after function: bar
Called before function: foo
Called before function body starts to execute: foo
Called before function:
foo
Called after function:
Called after function body completes, before return
Called after function: foo
Called before function: bar
Called before function body starts to execute: bar
Called before function:
bar
Called after function:
Called after function body completes, before return
Called after function: bar
Called before function: foo
Called before function body starts to execute: foo
Called before function:
foo
Called after function:
Called after function body completes, before return
Called after function: foo
Called before function: bar
Called before function body starts to execute: bar
Called before function:
bar
Called after function:
Called after function body completes, before return
Called after function: bar
Called before function: foo
Called before function body starts to execute: foo
Called before function:
foo
Called after function:
Called after function body completes, before return
Called after function: foo
Called before function: bar
Called before function body starts to execute: bar
Called before function:
bar
```

```
Called after function:
Called after function body completes, before return
Called after function: bar
Called before function: foo
Called before function body starts to execute: foo
Called before function:
foo
Called after function:
Called after function body completes, before return
Called after function: foo
Called before function: bar
Called before function body starts to execute: bar
Called before function:
bar
Called after function:
Called after function body completes, before return
Called after function: bar
Called before function:
done
Called after function:
At location
(/Users/luczsoma/projects/tresorit/modules/jalangi2/experiments/example.js:9:
17:9:23) 'true' branch was taken 10 time(s) and 'false' branch was taken 1
time(s).
At location
(/Users/luczsoma/projects/tresorit/modules/jalangi2/experiments/example.js:10
:7:10:16) 'true' branch was taken 5 time(s) and 'false' branch was taken 5
time(s).
```