Zbornik radova Fakulteta tehničkih nauka, Novi Sad

UDK: 004.41
DOI: https://doi.org/10.24867/01BE28Mirkovic

STATIČKA ANALIZA KODA ZASNOVANA NA UPOTREBI *ROSLYN* KOMPAJLERA

STATIC CODE ANALYSIS BASED ON USAGE OF *ROSLYN* COMPILER

Željka Aleksić, *Fakultet tehničkih nauka, Novi Sad*

Oblast – Primenjeno softversko inženjerstvo

Kratak sadržaj – *Statička analiza koda predstavlja proces analize izvornog ili binarnog koda softvera. Njen cilj jeste da se otkriju potencijalne slabosti softvera bez potrebe da se on prethodno izvrši, kao i da se proveri kompatibilnost stila sa postojećim preporukama i opštim standardima. Ovaj rad razmatra mogućnosti .NET kompajlerske platforme pod nazivom Roslyn i način na koji se upotrebom Roslyn API-ja može kreirati specijalizovani alat za statičku analizu koda zasnovan na smernicama .NET standarda za programski jezik C#.*

Ključne reči: *Statička analiza koda, Roslyn, kompajler, C# standard kodiranja*

Abstract – *Static code analysis is a process of analyzing software's source or binary code. The aim of analysis is to check the compliance to specific coding rules and discover potential vulnerabilities of software without actually executing the code. This work analyzes the possibilities of .NET compiler platform named Roslyn and describes the implementation of a custom tool for static code analysis based on .NET coding standard for C#.*

Keywords: *Static code analysis, Roslyn, compiler, C# coding standard*

1. UVOD

Sa porastom obima softverskog proizvoda, raste i potreba za automatizovanim alatima koji pomažu u otkrivanju potencijalnih problema u kodu. Statička analiza koda predstavlja prvi korak u analizi koda, pre puštanja u rad samog softvera. Statička analiza koda može da se primenjuje u ranim fazama razvoja softvera i za razliku od testiranja, može da se primeni i na nedovršenim izvornim kodom. Štaviše, ona otkriva koren problema dok dinamičko testiranje ističe samo posledice.

Na tržištu postoji veliki broj alata namenjenih statičkoj analizi koda kao što su *FxCop*, *StyleCop* i *Resharper*. Njihova zajednička karakteristika jeste da moraju da parsiraju izvorni kod pre same analize, što je u stvari zadatak kompajlera. Međutim, tradicionalni kompajleri se mogu opisati kao crne kutije, gde su poznati ulazi, u vidu izvornog koda, i izlazi, u obliku objektnog fajla, dok sam proces razumevanja koda u toku kompajliranja ostaje nepoznat krajnjim korisnicima kompajlera.

.NET kompajler platforma, poznata pod nazivom Roslyn, je grupa open-source kompajlera i API-ja za analizu koda namenjenih C# i Visual Basic programskim jezicima. Glavna motivacija za razvoj Roslyn-a je upravo otvaranje crnih kutija kako bi se krajnjim korisnicima, inženjerima, omogućio pristup bogatstvu informacija o kodu kojima kompajleri raspolažu. Na ovaj način kompajler postaje API. Upotrebom .NET kompajler platforme (*Roslyn*), eliminiše se potreba da se izvorni kod parsira više puta, tako da se alati zasnovani na ovoj platformi mogu koncentrisati isključivo na analizu izvornog koda.

Cilj ovog rada jeste upoznavanje sa principima i mogućnostima .NET kompajler platforme, poznate pod nazivom *Roslyn* kao i kreiranje specijalizovanog alata za statičku analizu koda upotrebom *Roslyn* kompajlera. Implementacija alata je zasnovana na smernicama C# standarda za pisanje koda. To su smernice vezane za imenovanje, formatiranje i struktuiranje izvornog koda.

2. STATIČKA ANALIZA KODA

Statička analiza koda predstavlja proces evaluacije sistema ili komponente na osnovu njene strukture, sadržaja ili dokumentacije. Ona adresira slabosti programskog koda koje mogu dovesti do ranjivosti sistema. Njen zadatak je donošenje zaključaka o mogućem ponašanju programa bez potrebe da se on prethodno izvrši. Cilj analize jeste provera stepena podudaranja izvornog koda sa specificiranim pravilima i identifikacija delova koda koji potencijalno mogu biti štetni.

Statička analiza koda može biti manuelna, kao što je revizija koda ili može biti automatizovana upotrebom softverskih alata. Automatizovani alati uglavnom vrše analizu izvornog koda. Međutim. postoji i manji broj alata koji vrše analizu nad kompajliranim, binarnim kodom [1].

2.1. Analiza izvornog koda

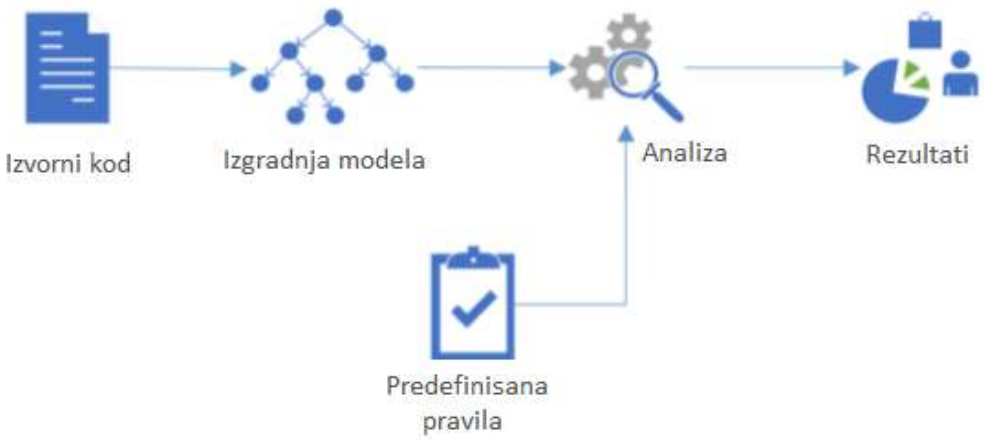
Kako bi alat mogao da analizira izvorni kod, potrebno ga je razumeti. Stoga prvi korak jeste da se kreira strukturni model koji prestavlja izvorni kod. Formiranje modela u statičkoj analizi koda je postupak sličan prvom delu procesa kompajliranja koji obuhvata parsiranje, leksičku i semantičku analizu koji rezultira sintaksnim stablom izvornog koda.

Nakon formiranja modela, sledeći korak je sama analiza. Mogu se koristiti mnogi algoritmi za analizu koda i uobičajeno je da se oni kombinuju u jedno rešenje. Algoritmi su veoma često izvedeni iz tehnika koje i sam kompajler primenjuje.

Predefinisana pravila predstavljaju početnu tačku statičke analize. Ona su podjednako važna, ako ne i važnija, od heuristika i algoritama na kojima se zasniva sama implementacija alata.

NAPOMENA:
Ovaj rad proistekao je iz master rada čiji mentor je bio dr Branislav Atlagić, docent.

Način na koji se vrši izveštavanje rezultata statičke analize koda je od velike važnosti. Ukoliko izveštaj nije razumljiv, sam rezultat statičke analize je neupotrebljiv, jer će nerazumevanje dovesti do toga da se greška ignoriše, ili još gore, označi kao lažno pozitivna. Dobar alat za statičku analizu koda treba da obezbedi grupisanje i sortiranje rezultata, kao i da izbegne izveštavanje nerelevantnih rezultata. Svaki detektovani problem, mora biti praćen detaljnim opisom problema, nivoom kritičnosti, kao i preporukama kako bi problem mogao biti rešen [2]. Struktura procesa statičke analize je prikazana na slici 1.



Slika 1. *Proces statičke analize [2]*

2.2. Problemi kojima se bavi statička analiza koda

Postoje različiti tipovi alata za statičku analizu koda koji se bave različitim oblastima problematike: provera sintakse, stila, razumevanje programa, pronalaženje grešaka.

Provera sintakse je funkcionalnost koja je integrisana u svaki kompajler. Sintaksna pravila su uglavnom izvedena iz samog programskog jezika, te se na njih oslanjaju alati za proveru sintakse.

Alati za proveru stila vrše proveru kompatibilnosti izvornog koda sa definisanim pravilima za imenovanje, komentarisanje, kao i generalnu strukturu programa koja će najviše doprineti čitljivosti i održavanju programa.

Pojedini alati imaju za cilj da obezbede razumevanje programa na visokom nivou. Najefikasniji su kad su integrisani u razvojno okruženje (IDE) gde mogu da podrže *go to implementation* ili *find all references* funkcionalnosti, ili čak automatsku modifikaciju izvornog koda u vidu promene naziva promenljive.

Svrha alata za pronalaženje grešaka jeste da se predstave uobičajene greške u kodu. Oni upozoravaju na delove programa koji su u skladu sa specifikacijama programskog jezika ali ne izražavaju nameru inženjera.

3. NIVOI *ROSYLN* API SPREGE

Roslyn kompajlerski API se sastoji iz dva osnovna nivoa, kompajlerski API i API namenjen radnom prostoru. Pored ova dva nivoa, postoji i funkcionalni API višeg nivoa. Kompajlerski API pruža objektni model koji predstavlja rezultat sintaksne i semantičke analize procesa kompajliranja. Ovaj sloj sadrži presek stanja referenci, opcija vezanih za kompajliranje i izvornog koda koji nije podložan modifikaciji. Ovaj sloj je nezavisan od bilo koje *Visual Studio* komponente i kao takav se može koristiti i kao nezavisna aplikacija.

Sloj radnog prostora je početna tačka statičke analize i refaktorisanja koda. Unutar ovog sloja, API za radni prostor je zadužen za organizovanje informacija o projektima u okviru *solution*-a u jedan objektni model. API za radni prostor nudi direktan pristup objektnim modelima sloja kompajlera kao što su izvorni tekst, sintaksno stablo, semantički model i kompilacija bez potrebe za parsiranjem fajlova ili promenom podešavanja konfiguracije.

Funkcionalni API se oslanja na kompajlerski sloj i sloj namenjen radnom prostoru i dizajniran je tako da nudi API namenjen refaktorisanju i automatskim ispravkama koda [3].

3.1. Dijagnostifikacija problema

Implementacija logike statičke analize koda može da se sastoji od više pravila za detekciju domenski specifičnih grešaka i problema u kodu. Svako pravilo je potrebno definisati kao tip *DiagnosticDescriptor*.

Detektovani problemi se korisniku prijavljuju kao upozorenje. Linija ili deo koda koji nije u skladu sa definisanim pravilom će biti podvučen zelenom linijom. Prebacivanjem fokusa na podvučen deo koda, otvara se dodatni prozor sa porukom obaveštenja o grešci.

Code Fix je brza akcija koja nudi predlog mogućeg rešenja za dijagnostifikovane probleme pronađene u kodu od strane alata za statičku analizu koda. Može se primeniti pritiskom na prečicu Crtl+. koja je sastavni deo *Visual Studio*-a. Pre same primene ponuđenih izmena, programer može da vidi kako bi te izmene izgledale

4. IMPLEMENTACIJA SPECIJALIZOVANOG ALATA ZA STATIČKU ANALIZU KODA

Implementacija specijalizovanog alata za statičku analizu koda zasnovana je na smernicama za pisanje koda po

.NET standardu namenjenom programskom jeziku C# [5]. Ovaj dokument opisuje pravila i preporuke za razvoj aplikacija i biblioteka pisanih C# programskim jezikom. Cilj jeste da se definišu opšte smernice kako bi se obezbedila konzistencija u stilu i formatiranju izvornog koda, kao i da bi se izbegle uobičajene greške softver inženjera. Dokument je podeljen na 4 oblasti: konvencija imenovanja, stil pisanja koda, upotreba jezika i dizajn objektnog modela. Cilj ovog rada jeste implementacija specijalizovanog alata za statičku analizu koda koji vrši proveru prilagođenosti izvornog koda nekim od smernica .NET standarda iz oblasti konvencije imenovanja, stila pisanja koda, kao i upotrebe jezika. Ukoliko je to moguće, alat treba da ponudi i odgovarajuću ispravku dijagnostifikovane greške.

Implementirani alat se može uključiti u *Visual Studio* razvojno okruženje kao ekstenzija. Kreiranjem projekta tipa *.Vsix*, kreira se i *Visual Studio Extension* instalacija koja kada se pokrene, instalira novi alat u razvojno okruženje. Analiza i *code fix* postaju dostupni onog trenutka kada se dodaju u *Visual Studio*. Sama analiza se odvija u vreme pisanja koda. Ukoliko neki deo izvornog koda nije napisan u skladu sa pravilima definisanim analizom, generisaće se upozorenje u vidu zelene podvučene linije spornog dela koda sa porukom obaveštenja o grešci. Kada se prebaci fokus na označeni deo izvornog koda, otvara se padajući meni sa *code fix*-om i prikazom primene ispravke. Klikom na ponuđeni *code fix* izmeniće se sporni deo koda u skladu sa ponuđenom ispravkom, tako da se nakon primene ispravke, upozorenje više neće prikazivati.

Karakteristika alata za statičku analizu koda jeste da se uglavnom sastoje od kratkih provera poštovanja pravila koje se mogu implementirati u svega par linija koda. Pošto se dijagnostike pravila razlikuju po jedinstvenom identifikatoru, implemetirana pravila su definisana identifikatorom *CR00X* gde oznaka *CR* označava *custom rule*, dok *X* predstavlja redni broj pravila. Prilikom ispisa dijagnostike, prikazuje se identifikator pravila i poruka, dok se otvaranjem dodatnog prozora prikazuje i opis dijagnostike.

Stil pisanja koda je najveći uzročnik nekonzistentnosti koda. Većina inženjere tokom godina i radnog okruženja i zahteva kompanije u kojoj radi stiče određeni stil pisanja koji je neretko u suprotnosti od preferenci njegovih kolega. Međutim, konzistentan format i organizacija koda su ključni prilikom kreiranja održivog koda. .NET standard nudi niz smernica kako bi se kreirao čitljiv i konzistentan softver koji se lako razume i lako održava.

Kako bi se povećala čitljivost koda, .NET predlaže da se koriste vitičaste zagrada kada god je to moguće, i uvek u novom redu, što povećava preglednost koda i stvara utisak hijerarhije. Za iskaze uslova koji nisu duži od jedne linije, u C# programskom jeziku nije potrebno navoditi vitičaste zagrade, ali s obzirom na to da njihova upotreba dovodi do lakšeg razumevanja koda, preporučljivo je da se koriste. Pravilo *CR003* je namenjeno proveri iskaza uslova *if*, tačnije proveri da li je neki od njegovih čvorova dece tipa *Block*.

Kako bi se obezbedila modularnost koda, preporuka je da se po jednom fajlu definiše najviše jedan imenski prostor i jedna klasa. Definisanje više klasa u istom fajlu smanjuje čitljivost koda i otežava njegovo razumevanje. Pravilo CR004 proverava da li se unutar jednog *CompilationUnit*-a nalazi više od jednog *NamespaceDeclaration*-a ili *ClassDeclaration*-a. Ukoliko se detektuje narušavanje ove sugestije, generisaće se upozorenje u vidu zelene podvučene linije *CompilationUnit*-a u celosti.

Komentari su elementi izvornog koda koji ne utiču na funkcionalnost izvršavanja programa. *Roslyn* kompajler ih svrstava u sintaksne trivije koje se vezuju za tokene. Njihova uloga može biti u dokumentovanju metode i klase, međutim, prekomerna upotreba komentara može loše da utiče na razumevanje koda i oteža navigaciju i manipulaciju nad izvornim kodom. Za komentare važi preporuka da treba da se izbegava upotreba takozvanih *flowerbox* komentara koji se navode između znakova /* i */. Umesto ovog tipa komentara, savetuje se primena jednostavnijeg oblika u vidu // ili ///. Pravilo CR005 prepoznaje *flowerbox* komentare i generiše upozorenje sa prikladnom porukom koja sugeriše format u kojem treba da se navode komentari.

Opšta praksa za navođenje imenskih prostora jeste da naziv imenskog prostora treba da sadrži i naziv projekta, i nazive svih foldera u kojem se posmatrani .cs fajl nalazi. Prilikom kreiranja novog .cs fajla u projektu, imenski prostor će u nazivu sadržati i naziv projekta i nazive fajlova u kojima se nalazi. Međutim, ukoliko dođe do izmene u organizaciji foldera ili promene lokacije .cs fajla, potrebno je promeniti i naziv imenskog prostora.

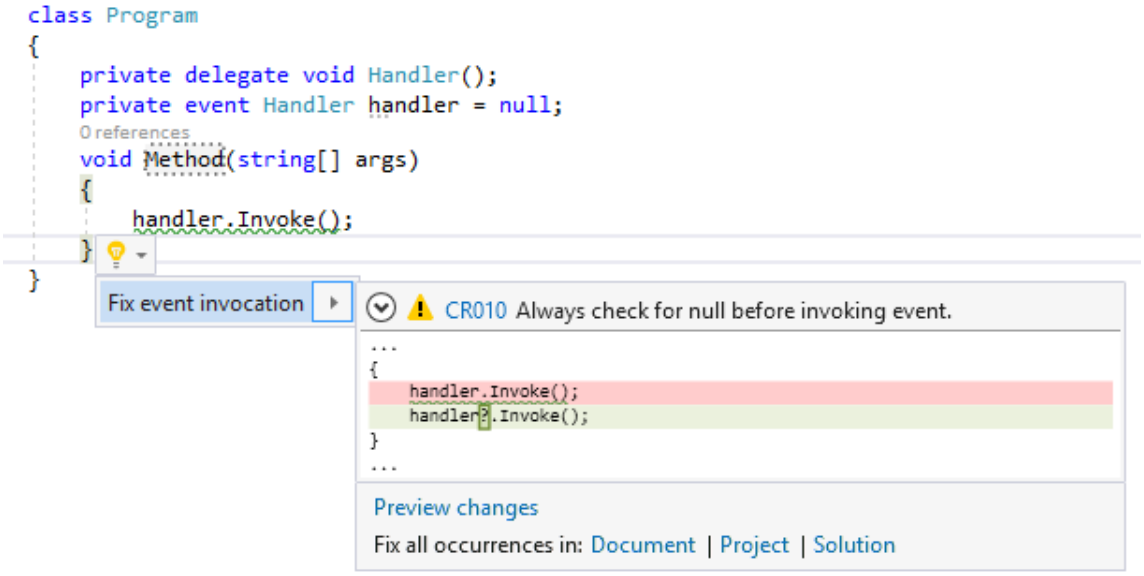
Pored ovog opšteg pravila, mnoge kompanije poseduju specijalizovano pravilo za navođenje imenskih prostora koje glasi da je potrebno da se u nazivu imenskog prostora, pored naziva svih hijerarhijski nadređenih fajlova, nalazi i naziv *assembly*-ja. Kako bi se promena naziva imenskog prostora automatizovala, implementirana je analiza i *code fix* koji vrše proveru i ispravku naziva na poziv prečice *Ctrl+.* .

Ključna reč *base*, programskog jezika C#, koristi se za pristup članovima bazne klase iz izvedene klase, dok ključna reč *this* predstavlja referencu na trenutnu instancu klase. Smernice za pravilno korišćenje konstrukcija ovog programskog jezika nalažu da se ove dve ključne reči koriste samo u okviru konstruktora ili metoda označenih sa *override* modifikatorom. Kako bi se izvršila provera upotrebe pomenutih ključnih reči potrebno je da se pristupi njihovim roditeljskim čvorovima sintaksnog stabla. Potrebno je da neki od čvorova hijerarhijski viših on *BaseKeyword* ili *ThisKeyword* budu tipa *ConstructorDeclaration* ili *MethodDeclaration*, dodatno, u slučaju da je reč o metodi, potrebno je da neki od modifikatora metode ima vrednost *override*. Ukoliko prethodni uslovi nisu zadovoljeni analiza će rezultirati upozorenjem.

Događaji omogućuju da se neka klasa ili objekat obaveste kada se desi nešto od interesa. Publikacija događaja se vrši pozivom metode *Invoke()*, ali pre toga je potrebno proveriti vrednost promenljive tipa *event* koja ne sme da ima vrednost *null* kako bi publikacija mogla da se izvrši (slika 2). Postoje dva načina povere vrednosti promenljive tipa *event*. Jedan način jeste da se poziv *Invoke()* metode navodi unutar uslovnog iskaza *if* koji vrši proveru vrednosti promenljive, dok drugi pristup zahteva upotrebu uslovnog *null* operatora (?). Pravilo *CR010* vrši proveru

poštovanja navedene smernice. Analiza se sastoji iz dva dela, prvog koji pronalazi promenljive tipa *event* kako bi se sačuvao njegov identifikator, i drugog koji vrši proveru poziva metode *Invoke()*.

Ponovo je potrebno pristupiti roditeljskim čvorovima *InvocationExpression*-a, te proveriti da li je neki od njih tipa *ConditionalAccessExpression* (?) ili *IfStatement*. U slučaju kada je poziv *Invoke()* metode enkapsuliran u uslovni iskaz *if*, treba izvršiti i proveru samog uslova, kako bi sa sigurnošću utvrdili da se odnosi tačno na promenljivu tipa *event*.



Slika 2: *Prikaz dijagnostike i preporuke ispravke dela izvornog koda koji nije u skladu sa smernicama C# standarda*

5. ZAKLJUČAK

Glavi cilj rada jeste istraživanje mogućnosti *Roslyn* kompajlera sa aspekta statičke analize koda. Činjenica, da je *Roslyn* relativno nova platforma i da postoji malo alata zasnovanih na njegovoj upotrebi, glavni je razlog ovog rada i implementacije specijalizovanog alata za statičku analizu koda. *Roslyn* omogućuje pisanje alata koji nude opciju analize izvornog koda već pri samom pisanju koda, gde se softver inženjeru ukazuje potencijalni problemi već u ranim fazama razvoja.

Pored toga, upotrebom intuitivnih *Roslyn* API-ja, koji izlažu veliki broj metoda za rukovanje objektima koji reprezentuju delove izvornog koda, može da se kreira alat koji će vršiti proveru kompatibilnosti koda sa pravilima nekog standarda ili kompanijskim pravilima. Pravila mogu biti različitog tipa, od konvencije imenovanja do načina upotrebe programskih iskaza.

Pravac daljeg razvoja jeste proširenje skupa implementiranih pravila. Do sada je akcenat bio na stilskim pravilima, dok je cilj budućeg razvoja unapređenje alata sa dodatnim pravilima namenjenim detektovanju šablona koji se često pojavljuju u izvornom kodu, a čija zamena može da se izvrši na automatizovan način sa ciljem poboljšanja performansi ili refaktorisanja koda.

Reč je o naprednoj analizi koda koja može da detektuje beskonačne petlje, rekurzivne pozive metoda i duboko ugnježdene petlje, a čijim izmenama se mogu poboljšati performanse softvera. Takođe, može da se upotrebi za izračunavanje raznih metrika kvaliteta, kao što su proširivost, kompleksnost, održivost, hijerarhija nasleđivanja klasa i dupliciranje koda.

6. LITERATURA

[1] owasp.org, 2017, *Static Code Analysis*, [online] dostupno na: https://www.owasp.org/index.php/Static_Code_Analysis [posećeno 4 Sep. 2018]

[2] Chess, B and West, J 2007, *Secure programming with Static Analysis*, Addison-Wesley, Boston

[3] github.com, 2018, *.NET Compiler Platform ("Roslyn") Overview*, [online] dostupno na: https://github.com/dotnet/roslyn/wiki/Roslyn%20Overview [posećeno 6 Sep. 2018]

[4] Chess, B and West, J 2007, *Secure programming with Static Analysis*, Addison-Wesley, Boston

[5] Hunt, L 2007, *C# coding standard for .NET*

Kratka biografija:

Željka Aleksić rođena je u Rijeci 1994. god. Diplomski rad na Fakultetu tehničkih nauka iz oblasti Elektrotehnike i računarstva – Sigurnost i bezbednost u sistemu pametnih brojila odbranila je 2017.god.