Zbornik radova Fakulteta tehničkih nauka, Novi Sad

UDK: 004.41
DOI: https://doi.org/10.24867/01BE45Maric

AUTOMATIZACIJA INSTRUMENTALIZACIJE KODA POMOĆU *ROSLYN* GENERATORA KODA

AUTOMATION OF INSTRUMETALIZATION OF CODE USING *ROSLYN* CODE GENERATOR

Luka Marić, *Fakultet tehničkih nauka, Novi Sad*

Oblast – ELEKTROTEHNIKA I RAČUNARSTVO

Kratak sadržaj – *Automatizovana instrumentalizacija koda predstavlja proces modifikacije koda kako bi se omogućilo vođenje evidencije o događajima koji su se desili u toku izvršavanja softvera od interesa. Motivacija za implementaciju predstavlja ušteda vremena koja može da se dobije ukoliko se čitav proces automatizuje. Za implementaciju rešenja korišćena je Roslyn platforma za pisanje korisnički definisanih kompajlera. Razlog upotrebe Roslyn platforme leži u objektnom modelu koji pruža i intuitivnom načinu za generisanje novog koda što predstavlja centralni deo rada koji će biti opisan.*

Ključne reči: *Roslyn, kod generacija, C#, kompajler*

Abstract – *Automated instrumentation of code is a process of code modification to allow it to keep records of events that occurred during the execution of software of interest. The motivation for implementation is the time savings that can be gained if the whole process is automated. To implement the solution, Roslyn platform for writing user-defined compilers is used. The reason for using the Roslyn platform lies in an object model that it provides and an intuitive way of code generation, which is a central part of the work that will be described.*

Keywords: *Roslyn, code generation, C#, compiler*

1. UVOD

Još od samog početka informatičke revolucije, pa do danas prodor tehnologije u sve sfere ljudskog života nezaustavljivo raste. Tako danas, retko koja grana industrije nije zavisna od raznovrsnih tehnoloških proizvoda. Kako bi ti proizvodi mogli da funkcionišu na predviđen način, uz što manje uplitanja ljudi, neophodno je da njegov rad bude kontrolisan od strane softvera koji je pisan za njega.

Sa napretkom tehnologije, cena potrebnih uređaja je drastično padala, što je prouzrokovalo dalji rast industrija zavisnih od istih. Napredak tehnologije nije samo uticao na cenu uređaja, nego i na njihovu raznolikost, u zavisnosti od njihove namene.

Naravno, kao što je ranije rešeno, kako bi se što optimalnije koristili uređaji, morao biti napisan i softver za njihovo upravljanje.

Pošto je gotovo nemoguće da se u prvom pokušaju, ili uopšte napiše "savršen" softver, od presudnog značaja za profit je vreme provedeno u procesu otkrivanja izvora problema kao i načina za njegovo rešavanje. Nekad je, u slučaju obimnih programskih sistema, veoma teško pronaći izvor problema dok njegovo rešavanje može da bude trivijalno. Pored toga, postoji šansa da se napiše takav softver čije će performanse biti narušene zbog malog broja funkcionalnosti koje nisu dovoljno optimizovane. U tom slučaju, procesor bi gubio dragoceno vreme na izvršavanje tih funkcionalnosti, umesto da to isto vreme posveti nečemu drugom. Zbog te činjenice, proces instrumentalizacija izvornog koda softvera predstavlja proces od izuzetne važnosti.

U kontekstu razvoja softvera, termin instrumentalizacija se odnosi na mogućnost nadzora ili merenja nivoa performansi proizvoda kao i dijagnostifikovanja različitih problema i čuvanja informacija kako bi se problem što lakše mogao lokalizovati i odstraniti. Instrumentalizacija može biti ostvarena različitim tehnikama vođenja evidencije u toku izvršavanja softvera [3]. Takođe, veoma korisno može da bude i zapisivanje u kom trenutku se događaji od važnosti dešavaju.

2. ROSLYN

Roslyn predstavlja jednu od novijih platformi za razvoj kompajlera. Suštinski *Roslyn* je grupa *open-source* kompajlera i API-ja namenjenih za vršenje analize izvornog koda napisanog u C# i *Visual Basic* programskim jezicima.

Motivacija za njegov razvoj leži u tome što je pre njega proces kompajliranja predstavljao crnu kutiju. Termin crna kutija se odnosi na veoma malo informacija o tome šta se zapravo događa u toku kompajliranja, i da su poznati samo ulazi, u vidu izvornog koda, i izlazi u obliku objektnog fajla. Način na koji kompajler u toku prevođenja programa stiče znanja o samom izvornom kodu je dugo vremena bio nepoznat. Nije bilo moguće da se bilo koji način prekine proces kompajliranja kako bi se mogli proučiti međukoraci. Kako kompleksnost razvijanog softvera raste, raste i potreba za razumevanjem načina na koji kompajler stiče potrebne informacije kako bi što bolje preveo izvorni kod. To je omogućeno *Roslyn* platformom.

2.1. Sintaksa

Najosnovnija struktura podataka izložena od strane kompajlerskih API-ja je sintaksno stablo. Svrha ovih

---

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

sintaksnih stabala je da grafički predstave leksičku I sintaksnu strukturu izvornog koda. Razlozi za to su :

1. Omogućavanje različitim alatima (razvojno okruženje, alati za analizu koda itd.) lakše kretanje kroz posmatrani kod, njegovo bolje razumevanje kao i lakše procesuiranje izvornog koda.
2. Omogućavanje alatima, kao što su alati za refaktoring ili razvojna okruženja, da modifikuju ili kreiraju novi izvorni kod bez potrebe da se koristi tekstualni editor.

2.2. Sintaksno stablo

Sintaksno stablo ima tri ključne osobine. Prva osobina je ta da svako sintaksno stablo sadrži sve informacije o izvornom kodu: svaku gramatičku konstrukciju, leksički token, čak i delove koda koji predstavljaju beline, komentare i preprocesorske direktive.
Mogućnost da se kreirano stablo ponovo transformiše u tekstualni format predstavlja drugu od tri ključne osobine. Od svakog sintaksnog čvora, moguće je dobiti tekstualnu reprezentaciju podstabla koje kao koren ima izabrani sintaksni čvor. Zahvaljujući ovoj osobini, moguće je vršiti potrebne modifikacije koje će se kasnije odraziti na izvorni kod.
Treću osobinu sintaksnog stabla predstavlja činjenica da je to *immutable* struktura podataka, a to znači da njeno eventualno menjanje neće uticati na posmatrani objekat, već će biti kreiran potpuno novi objekat koji će se od originala razlikovati u načinjenoj izmeni. Takođe, ove strukture mogu da se koriste od strane više različitih niti, koje se izvršavaju u sklopu programa, bez potrebe za proverom da li je neka druga nit vršila modifikacije na određenim elementom.

2.3. Sintaksni čvor

Sintaksni čvor predstavlja osnovni činilac sintaksnog stabla. Reprezentuje sintaksnu konstrukciju kao što su deklaracije, iskazi i izrazi. Osnovna klasa u objektnom modelu koja predstavlja sintaksni čvor je *SyntaxNode* klasa, iz koje se dalje izvode klase koje predstavljaju čvorove specifične kategorije.

2.4. Sintaksni token

Sintaksni token predstavlja terminalni element gramatike jezika i najmanji sintaksni deo izvornog koda. U sklopu stabla se nikada ne nalaze u poziciji da predstavljaju roditeljski čvor nekom drugom čvoru. Sintaksni token može biti član stabla koji predstavlja identifikator, ključnu reč, broj i interpunkcijski znak. Za razliku od prethodno opisanih sintaksnih čvorova, u sklopu modela podataka Roslyn platforme jedna klasa je zadužena za predstavljanje svih tipova tokena, gde vrednosti svojstava klase variraju i zavise od tipa tokena kojeg modeluju.

2.5. Sintaksna trivija

Sintaksne trivije predstavljaju delove stabla koji nisu od značaja za dobro razumevanje samog koda. Tako sintaksne trivije mogu reprezentovati beline, komentare ili preprocesorske direktive. Sintaksne trivije se ne smatraju kao deca čvorovi drugih čvorova u stablu iz razloga što ne predstavljaju deo sintakse odabranog jezika i mogu da se nađu između bilo koja dva tokena u stablu. Uprkos tome, one su našle mesto u stablu zbog njihove važnosti u procesu refaktoringa i kako bi se osiguralo potpuno poklapanje sintaksnog stabla i izvornog koda na osnovu kog je ono kreirano.
Postoje dve vrste trivija koje se mogu naći u sintaksnom stablu: trivije koje prethode tokenu (Leading Trivia) i trivije koje slede nakon tokena (Trailing Trivia).

2.6. Greške

Važno je istaći da će sintaksno stablo biti kreirano i u slučaju da se u izvornom kodu detektuje greška. Pod greškom se smatra bilo koje neslaganje izvornog koda sa definisanom sintaksom izabranog programskog jezika. U tom slučaju parser može da konstruiše stablo na dva načina:

- Ukoliko parser očekuje token na određenom mestu u stablu i to nije slučaj, parser će ubaciti poseban token koji reprezentuje token koji nedostaje. Token će biti ubačen na tačno ono mesto gde je očekivan originalni token. Ubačeni token se po tipu ni na koji način ne razlikuje od očekivanog, jedino mu je svojstvo *IsMissing* postavljeno na vrednost *true*.
- Parser će preskočiti sve dok ne naiđe na token od kog može da nastavi proces parsiranja. Svi preskočeni tokeni se smeštaju u kolekciju koja predstavlja triviju tipa *SkippedTokens*.

2.7. Generisanje koda pomoću *Roslyn* platforme

Ranije su strategije vezane za generisanje koda bile bazirane na strukturiranju metapodataka po šablonu T4 ili u objekte klasa sadržanih u bibliotekama poput *CodeDom*. Nešto više o pomenutim strategijama biće rečeno u nastavku poglavlja. Još pre upotrebe šablona T4 i biblioteke *CodeDom*, inženjeri su bili primorani da direktno koriste *StringBuilder* kako bi generisali kod.
T4 šabloni predstavljaju mešavinu tekstualnih blokova i kontrolne logike za generisanje tekstualnih datoteka. Kontrolna logika može biti pisana u programskom jeziku C# ili *Visual Basic*. Generisani tekst može biti internet stranica, resursna datoteka ili izvorni kod pisan u željenom programskom jeziku [2].
Biblioteka *CodeDom* pruža tipove podataka koji u najvećem broju slučajeva pokrivaju opšte tipove koji se javljaju u izvornom kodu. Generisanje koda se vrši kreiranjem objektnog grafa koji sadrži elemente biblioteke *CodeDom*.
Roslyn pruža bolju kontrolu nad kodom u poređenju sa *CodeDom* bibliotekom. Generisanje izvornog koda u bilo kom programskom jeziku je takođe podržano. Kako *Roslyn* radi sa objektnim modelom, moguće je da se prati stanje i tok rada napisanog alata kako bi moglo da se vidi na koji način *Roslyn* rukuje generisanim kodom [1].

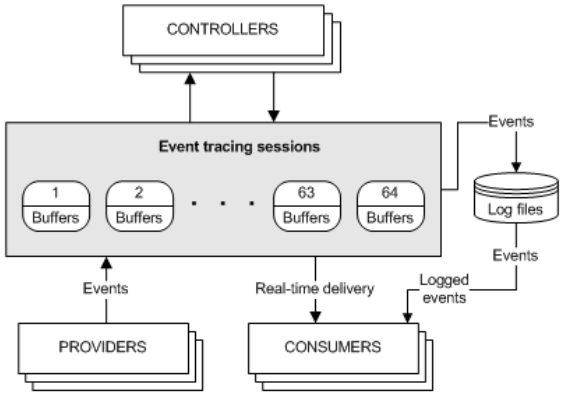
3. EVENT TRACE FOR WINDOWS

Event tracing for windows predstavlja efikasan mehanizam za vođenje evidencije na nivou kernela, koji korisniku omogućava da zapisuje željene informacije o događajima koji su se dogodili u toku rada *Windows* operativnog sistema ili korisnički definisanog programa koji podržava ovakav način vođenja evidencije. Zapisane

informacije se kasnije mogu koristiti kako bi se otkrili problemi u kodu kao i da bi se izmerile performanse izvršavanja pisanog koda.
Event Tracing API podaljen je u tri odvojene komponente:

- kontroleri – zaduženi za startovanje i stopiranje procesa evidentiranja
- provajderi – zaduženi za objavljivanje događaja
- korisnici – alati koji koriste zapisane podatke

Način na koji radi *ETW* prikazan je na slici 3.1.



Slika 3.1 *Evetn Tracing for Windows* mehanizam

3.1. Kontroleri

Kontroleri su programi koje:

- definišu veličinu i lokaciju datoteka u kojoj će se vršiti perzistencija evidentiranih informacija,
- započinju i prekidaju sesije vođenja evidencije,
- omogućavaju provajderima da vrše evidenciju,
- rukuju veličinom *buffer*-a i
- vode statistiku koja uključuje broj korišćenih *buffer*-a, kao i izgubljenih događaja i *buffer*-a

3.2. Provajderi

Provajderi su programi koje su instrumentalisani. Kako bi provajder mogao da vodi evidenciju događaja, prvo je potrebno da se registruje na operativni sistem kako bi kontroler mogao da mu dozvoli vođenje evidencije. Od provajdera zavisi kako će se ponašati u slučaju da mu je dozvoljeno vođenje evidencije i u slučaju da nije.

3.3. Korisnici

Korisnici predstavljaju aplikacije koje se pretplaćuju na željene događaje iz odgovarajućih sesija. Ukoliko je neki od provajdera aktivan, korisnik informacije o njegovim događajima može dobiti iz predefinisanih datoteka ili direktno iz sesije u realnom vremenu.

4. OPIS REŠENJA PROBLEMA

Rešenje problema je implementirano kao dodatak Visual Studio integrisanom razvojnom okruženju koje je moguće u projekat uključiti kao *NuGet* paket.
Pomoćne klase koje su implementirane radi lakšeg implementiranja potrebne logike su:

- *MethodInfo* – klasa koja modeluje pojedinačnu metodu u klasi
- *ClassInfo* – klasa koja modeluje pojedinačnu klasu u dokumentu
- *DocumentInfo* – klasa koja modeluje pojedinčni dokument u sklopu odabranog projekta
- *ProjecInfoHelper* – klasa koja modeluje odabrani projekat

4.1. *MethodInfo*

Klasa *MethodInfo* se nalazi na najnižoj poziciji u hijerarhiji pomoćnih klasa i ona služi za perzistenciju informacija o detektovanim metodama u sklopu izvornog koda, koje su od značaja.

4.2. *ClassInfo*

Klasa *ClassInfo* služi za enkapsuliranje informacije, o detektovanim klasama u sklopu odabranog projekta za instrumentalizaciju, koje će kasnije biti od značaja.

4.3. *DocumentInfo*

U hijerarhiji, klasa *DocumentInfo* se nalazi odmah ispod *ProjectInfoHelper* klase. Ova klasa modeluje čitave .cs datoteke koje su uključene u posmatrani projekat nezavisno od toga da li u sklopu projekta postoji neka hijerarhija foldera.

4.4. *ProjectInfoHelper*

ProjectInfoHelper klasa se nalazi na vrhu opisane hijerarhije, pored zaduženja za skladištenje korisnih informacija o projektu od interesa, zadužena je i za čuvanje napravljenih izmena na originalnim projektom.

4.5. Klase koje vrše generaciju koda

U okviru implementacije rešenja problema, generisanje koda je podeljeno na tri klase. Takva podela je napravljena pošto je bilo potrebno na tri različita mesta vršiti generisanje. Tako klase koje su zadužene za generisanje koda su:

- *InstrumentationGenerator* – klasa zadužena za modifikovanje postojećeg koda,
- *EventSourceEventGenerator* – klasa zadužema za generisanje potrebnih događaja i
- *EventSourceWriterGenerator* – klasa zadužena za generisanje metoda koje upisuju događaje

4.5.1. *InstrumentationGenerator*

InstrumentationGenerator klasa, kao što je rečeno, ima ulogu da modifikuje postojeći kod. Rešenje problema je zamišljeno tako da su za svako metodu od interesa bitna četiri događaja: početak i kraj izvršavanja metode, slučaj kada je došlo do problema prilikom izvršavanja kao i beleženje povratne vrednosti metode. Kako bi se jasno mogli razdvojiti događaji koji obeležavaju kraj metode, događaji koji vraćaju informaciju o povratnoj vrednosti metode kao i događaji u slučaju neuspešnog izvršavanja

metode, čitava metoda je podeljena u tri posebne celine. Tako postoje *try*, *catch* i *finally* segmenti svake instrumentalizovane metode.

4.5.2. *EventSourceEventGenerator*

U skladu sa predefinisanim šablonom koji određuje kako generisana klasa treba da bude formatirana, njeno generisanje u okviru *EventSourceEventGenerator* klase podeljeno je u četiri dela:

1. Generisanje polja potrebnih za konfigurisanje događaja
2. Generisanje samih događaja
3. Generisanje jedinstvenih identifikatora događaja
4. Generisanje jedinstvenih identifikatora metoda na osnovu kojih će događaji biti generisani

Kako je bilo potrebno podržati mogućnost pokretanja procesa instrumentalizacije i u slučaju da je određeni deo koda već instrumentalisan, bilo je potrebno implementirati funkcionalnost kojom će se izvršiti modifikacija postojeće klase koja sadrži generisane događaje. Modifikacije koje je potrebno izvršiti su dodavanje skupa događaja zasnovanih na obrađenim metodama, kao i njihovih identifikatora.

Pošto je produkt rada *EventSourceEventGenerator* klase novi dokument koji u tom trenutku nije ni na koji način vezan za izabrani projekat, bilo je potrebno implementirati logiku za njegovo čuvanje i uključivanje u projekat.

4.5.3 *EventSourceWriterGenerator*

Pošto postoji gotovo neograničeni broj kombinacija skupova parametara, metode koje su zadužene za zapisivanje događaja koje se nalaze u okviru klase *EventSource*, koju nasleđuje generisana klasa zadužena za zapis događaja, nisu dovoljne kako bi se pokrio svaki pojedinačan slučaj. To je uslovilo potrebu za generisanjem posebne klase koja sadrži korisnički definisane metode za zapis događaja na osnovu metode iz izvornog koda nad kojima se vrši proces instrumentalizacije.

Upravo za to služi *EventSourceWriterGenerator* klasa.

Šablon koji klasa treba da poštuje manje je kompleksan u odnosu na šablon *EventSourceEventGenerator* klase tako da se njeno generisanje može podeliti u dva dela. Prvi deo je zadužen za generisanje logike koja odlučuje u kom trenutku će se zapisivati tačno vreme dešavanja događaja, dok se u drugom delu generišu potrebne metode koje će vršiti zapisivanje događaja.

Kako nije redak slučaj da različite metode mogu imati jednake skupove parametara po pitanju broja i tipova, potrebno je obezbediti samo jednu generisanu metodu za zapisivanje događaja. To je neophodno kako se ne bi generisao problematičan kod koji će u daljem radu onemogućiti proces kompajliranja.

5. ZAKLJUČAK

Kada je reč o prednostima i manama implementiranog rešenja, potrebno je adresirati je jedno i drugo. Jedna od najvećih prednosti i glavnih razloga automatizacije postupka instrumentalizacije jeste vreme koje je potrebno da bi se postupak izvršio. Za ručnu instrumentalizaciju projekta prosečne kompleksnosti i veličine vreme koje je potrebno izdvojiti može biti reda veličine nekoliko dana ili čak nedelja. Kada je automatizovani način u pitanju, uz dobro definisani šablon koje bi generisane klase poštovale, postupak instrumentalizacije bi mogao da se izvrši za vreme reda veličine nekoliko minuta. Međutim, uprkos velikoj brzini generisanja koda, problem predstavlja njegova količina.

Za projekte prosečne veličine broj generisanih linija lako može dostići i nekoliko hiljada, što može biti veoma teško za održavanje i proveru validnosti. Osim količine koda, problem predstavljaju različite navike pisanja koda koje inženjeri imaju. Kako bi se pokrili svi mogući slučajevi, potrebno je da se rešenje testira ne velikom broju projekata, ali ni tada se ne bi moglo reći sa sigurnošću da su svi slučajevi pokriveni. Dodatno, implementirano rešenje predstavlja alat koji je zadužen samo za generisanje koda, ali ne i za njegovo validiranje, tako da ukoliko sam izvorni kod nije napisan dobro, instrumentalizacije neće biti generisana na onako kako je predviđeno.

Kako opisano rešenje predstavlja početak istraživanja na polju vođenja evidencije pomoću *ETW* mehanizma i generisanja koda pomoću *Roslyn* platforme za pisanje kompajlera, ima dosta mesta za optimizaciju i napredak u razvoju alata. Jedan od budućih zadataka će biti da se pokriju sve situacije koje nisu pokrivene inicijalnim rešenjem. Primer takvih situacija su metode koje su pitane u programskom jeziku *C++* i *C* (*unsafe*), kao i privatne metode. Zatim bi od velike koristi bilo implementirati korisnički definisan alat za obradu i prikazivanje događaja koji su se desili, kako bi se mogle dobiti funkcionalnosti koje su potrebne korisniku a nisu implementirane u sklopu već postojećih alata.

6. LITERATURA

[1] drdobbs.com, 2011, Commenting, Testing, and Instrumenting Code, [online] dostupno na: http://www.drdobbs.com/architecture-and-design/commenting-testing-and-instrumenting-cod/229300224 [posećeno 07.09.2018]

[2] docs.microsoft.com, *Code Generation and T4 Text Templates*, [online] dostupno na: https://docs.microsoft.com/en-us/visualstudio/modeling/code-generation-and-t4-text-templates?view=vs-2017 [posećeno 10.09.2018]

[3] Alfred, V. Aho 2007, Compilers: Principles, Techniques, and Tools, 2nd edn, AddisonWesley, Boston

Kratka biografija:

Luka Marić rođen je u Novom Sadu 1994. god. Master rad na Fakultetu tehničkih nauka iz oblasti Elektrotehnike i računarstva – Automatizacija instrumentalizacije koda pomoću *Roslyn* kod generatora odbranio je 2018.god.