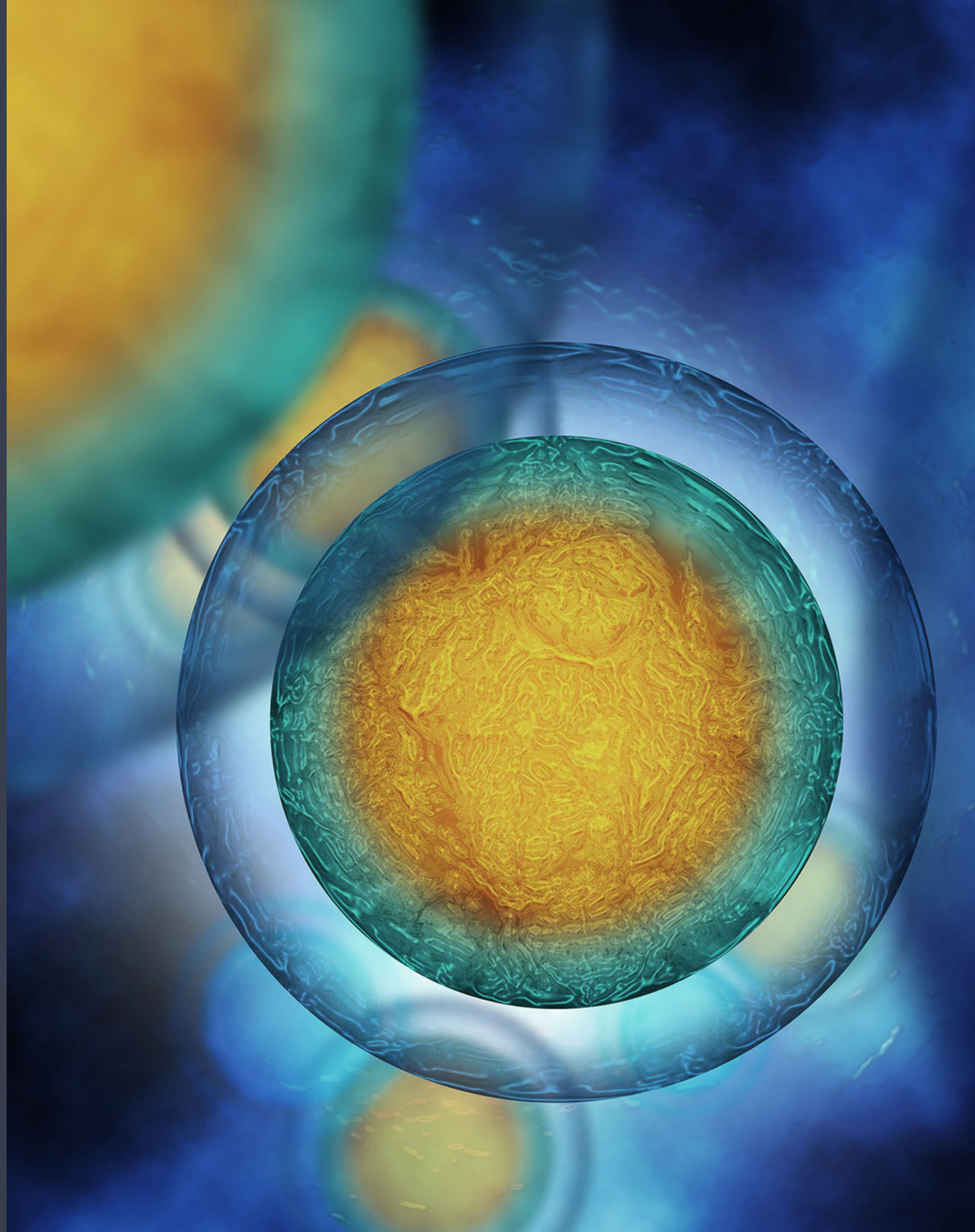
FE3CWS

MATERIJAL ZA ZIMSKU ŠKOLU 3CO



Intelekualni doprinos O1 iz ERASMUS+ projekta 2017-1-SK01-KA203-035402

Nekoliko riječi o

SADRŽAJU

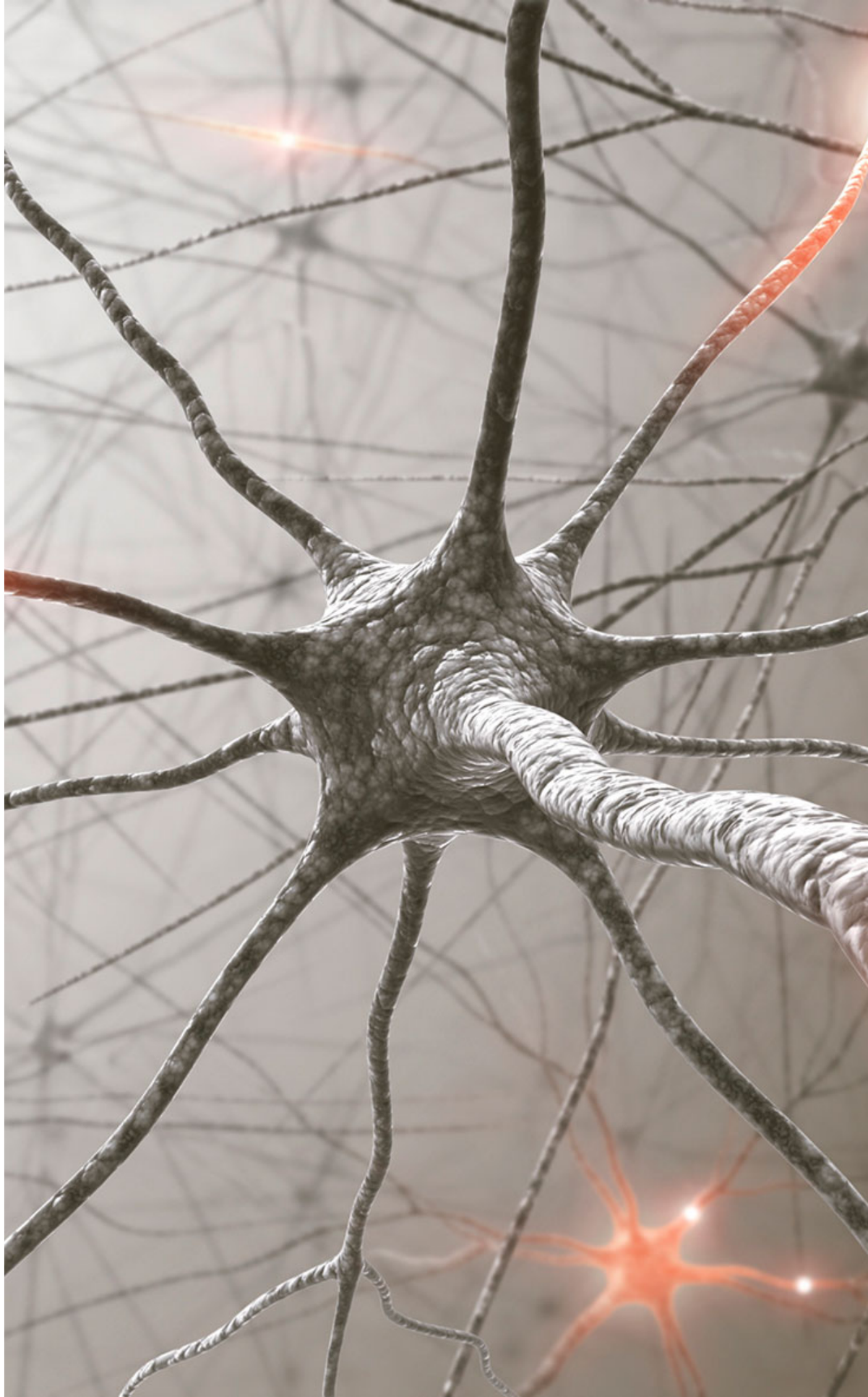
- **9 tema o kompoziciji, razumljivosti i ispravnosti softvera**
- **10 autora sa 7 europskih sveučilišta iz Hrvatske, Mađarske, Nizozemske, Portugala i Slovačke**
- **Dostupan na 7 jezika: engleski, mađarski, slovački, hrvatski, rumunjski, bugarski i portugalski**

Co-funded by the Erasmus+ Programme of the European Union

Zimska škola 3CO o kompatibilnost, razumljivost i ispravnost (3COWS: Composability, Comprehensibility and Correctness) je prvi intenzivan program za učenike i nastavno osoblje visokog obrazovanja koji šire zajednicu ljetne škole Centralno-europskog funkcionalnog programiranja (eng. *Central European Functional Programming*, CEFP) u okviru ERASMUS + projekta Br. 2017-1-SK01-KA203-035402 pod naslovom "Fokusiranje obrazovanja na kompatibilnost, razumljivost i ispravnost radnog softvera" koji se održava od 22. do 26. siječnja 2018. godine.

Uključeni materijal izrađen je i predstavljen u okviru navedenog projekta. Ova je publikacija tiskana verzija intelektualnog izlaza O1 iz ovoga projekta.

SADRŽAJ



AKVIZICIJA PRIRODNOG JEZIKA UPOTREBOM STROJEVA

Izgradnja stroja za razmišljanje je veliki izazov. Odnos jezika i uma zanimljiv je za znanstvenike računarstva, a u širem kontekstu o tome raspravljaju čak i povjesničari, psiholozi, lingvisti i filozofi.

Na primjer, Renfrew [31] karakterizira čovjeka kao simboličku bit obzirom da je razvoj prirodnog jezika rezultat razmjene simbola u komunikaciji. Gardenfors [14] navodi da jezični pojmovi imaju hijerarhijsku strukturu. Gramatička evolucija [24] i genetska evolucija jezika [15] utječu na problem strukturne ekspanzije, stoga se primjenjuju samo za jednostavne jezike na deterministički način. Ispada da genetski nizovi u ulozi vrijednosti semantičkih zaključaka predstavljaju obećavajući pristup evoluciji jezika od izravne primjene genetske strukture ljudskog mozga analogijom. Također smo prepoznali da se proces evolucije odvija na različitim meta-razinama jezika.

Chomskysova hipoteza o postojanju univerzalne gramatike za sve prirodne jezike [7] vrlo je zanimljiva i motivirajuća. No, univerzalnu gramatiku možda ne treba shvaćati kao fiksnu i parametriziranu gramatičku strukturu, već kao univerzalni algoritam, tj. deterministički napredak na razini meta jezika određenim parametrima koji predstavljaju smislene, tj. semantičke vrijednosti. Edelman [12] kaže da strojni um ima jezičnu tvar koja je istovremeno i simbolička i računska, iz čega slijedi da um nije samo simboličan nego i dinamičan proces koji izražava neprekinutu promjenu jezika. Konačno, u Shaumyanovoj aplikacijskoj univerzalnoj gramatici [34], jezik se također izražava u smislu dinamičkih, čak i aplikativnih procesa.

Prednost teorije semiotičkog jezika je da se sintaksa i semantika međusobno vezuju u svakom trenutku i da nisu razdvojive. Naravno,to upućuje na potrebu da se uzmu u obzir semantičke kategorije za izražavanje promjena jezika.

Po našem mišljenju, semiotička teorija jezika pruža priliku za deterministički razvoj strojnog uma za slučaj podskupina prirodnih jezika i formalnih jezika u različitim oblicima koji se koriste u komunikaciji. U tom smislu komunikacija čovjek-računalo i računalo-računalo utemeljena na jezicima može biti jedinstvena. Neka polazišta ove ideje, ograničena na redovite jezike koje predstavljena su u [17]. Trenutno poznajemo algoritam za preobrazbu jezičnih pojmova na unutarnji jezik strojnog uma i također možemo izvesti koncepte iz tog unutarnjeg jezika. Taj je unutarnji jezik analogija unutarnjem jeziku ljudskog razmišljanja - to je mirni jezik iza glasnog prirodnog jezika. Trenutačno ne poznajemo ni pravila konceptualizacije niti pravila o razmišljanju o konceptima predstavljenima u strojnom umu. Međutim, imamo jasnu metodologiju evolucije strojnog uma, predstavljenog primjenljivim dinamičkim procesima s visokim stupnjem paralelizma koji ne-redundantno predstavljaju informacije jezične supstance. Štoviše, povećanje apstrakcije jezičnih koncepata smanjuje broj meta operacija i povećava broj aplikacijskih veza.

Diskusijom o učinkovitom algoritmu dječjeg razmišljanja procjenjujemo i brzinu protoka informacija u ljudskom mozgu na 300 trilijuna signala u sekundi. Potom smo uveli slovačku tekstualnu i fonetsku gramatiku, kao i način prevođenja teksta u glas. Nadalje, ilustriramo proces akvizicije jezičnih elemenata različitih apstrakcija koji čitaju kratki uzorak teksta pomoću modela temeljenog na hijerarhijskim hash tablicama. Također procjenjujemo proces akvizicije za veliki tekst odabrane knjige i zaključujemo da akvizicija jezika s hijerarhijskom apstrakcijom donosi redukcijski grafikon s istim brojem sekvencijalnih i paralelnih veza bez potrebe za fizičkom pohranjivanjem podataka.

Zatim ćemo koristiti gramatički pristup priopćavanju prepoznavanja vizualnih objekata. Prvo, moramo shvatiti kako opisati objekte, a zatim možemo primijeniti metodu apstrakcije na te podatke. Prvenstveno se usredotočimo na opisivanje 3D objekta pomoću gramatika. U teoriji gramatike, ovaj se korak naziva simbolizacija.

Simbolizacija osigurava opis objekta i daje temeljni sloj apstrakcije podataka. Kao što možemo vidjeti, apstrakcija podataka pomoću funkcionalnog jezika omogućuje nam laku apstrahiraju i obradu objekata.

Prijavni pristup može se koristiti u jezičnoj obradi čak i kada koristimo gramatike oslobođene konteksta. Prikazujemo algoritam koji je u stanju transformirati bilo koju gramatiku oslobođenu konteksta u formu superkombinacije. Rezultirajuća forma ovisi o obliku gramatike unosa, što dovodi do novog problema: pronalaženje odgovarajuće gramatike za promatrani zadatak.

Ukratko ćemo prikazati dobivene oblike superkombinacijskih tipova različitih gramatičkih tipova i usporediti njihova svojstva. Usporediti ćemo učinkovitost algoritma u prezentiranim slučajevima gramatike i pokazali da se naš algoritam može poboljšati upotrebom gramatike bez ciklusa. Također, raspraviti ćemo o rezultirajućim oblicima superkombinacija, kompresiji gramatike i ponovljivosti uporabe elemenata koji su krajnji rezultat obrade većih tekstova.

Literatura

[1] Renfrew, C.: Prehistory: The Making of the Human Mind. Weidenfeld & Nicholson, (2009)

[2] Gardenfors, P.: Symbolic, conceptual and subconceptual representations, Human and Machine Perception, pp. 255-270, (1997)

[3] ONeill, M., and Ryan, C.: Grammatical evolution. IEEE Transactions on Evolutionary Computation, Vol. 5, No. 4, pp. 349–358, (2001)

[4] Hugosson J., Hemberg E., Brabazon A., ONeill M.: Genotype Representations in Grammatical Evolution. Applied Soft Computing, Vol.10, No.1, pp.36-43, (2010)

[5] Chomsky, N.: Syntactic Structures, Walter De Gruyter: Mouton classic, (1957)

[6] Edelman, Sh.: Computing the Mind: How the Mind Really Works, (2008)

[7] Shaumyan, S: A Semiotic Theory of Language. Bloomington: Indiana University Press, (1987)

[8] Kollar, J.: Formal Processing of Informal Meaning by Abstract Interpretation, Smart Digital Futures 2014, June 18-20, Chania, Greece, pp. 122–131, (2014)

ANALIZA STATIČKOG KODA S CODECHECKEROM

PREGLED

Simbolično izvršenje [4] popularna je statička analiza koja se koristi za provjeru programa i alate za otkrivanje neispravnosti. Ona funkcionira tumačenjem koda, uvođenjem simbola za svaku vrijednost koja nije poznata prilikom kompilacije (npr. korisnički unosi) i provedbom simboličnog izračuna. Analizator nastoji istodobno istražiti više puteva izvršenja, iako je provjera svih puteva predstavlja težak problem, zbog prevelikog broja mogućnosti.

Iako postoji bogata literatura o alatima za provjeru programa, alati za pronalaženje pogrešaka obično se trebaju podmiriti za pregledne radove o pojedinačnim tehnikama [1]. U ovom radu ne samo da raspravljamo o pojedinačnim metodama, već i o tome kako te odluke međusobno djeluju i uzajamno se pojačavaju, stvarajući sustav koji je veći od zbroja njegovih dijelova. Usredotočiti ćemo se na alat za pronalaženje pogrešaka nazvan Clang Static Analyzer [2] (u daljnjem tekstu "Analizator") i infrastrukture izgrađene oko njega pod nazivom CodeChecker [3]. Naglasak je na postizanju krajnje krajnje skalabilnosti.

To uključuje potrošnju vremena i memorije prilikom analize, predstavljanje neispravnosti korisnicima, automatska supresija lažno pozitivnih pronalazaka, inkrementalna analiza, otkrivanje uzoraka u rezultatima i korištenje u kontinuiranim integracijskim petljama. Također iznijeti ćemo buduće smjernice i otvoriti probleme vezane uz ove alate. Iako Analizator može raditi nositi samo s C / C ++ / Objektnim-C kodom, tehnike uvedene u ovom radu su neovisne o jeziku i primjenjive su na druge slične alate za statičku analizu.

CLANG STATIC ANALYZER

Sažeti ćemo radni mehanizam simboličkog izvršavanja i njegove implementacije u Analizatoru. Raspravljamo o njegovom prikazu memorije [6], njegovom manipuliranju vezama između vrijednosti i memorijskih lokacija te njegovom prikazu stanja specifičnih za provjeru (pri čemu pod provjerom podrazumijevamo jedan modul Analizatora napisanog kako bi pronašli jednu specifičnu vrstu neispravnosti). Također uvodimo i pojam simboličkih izračuna. Odabir prikaza koje koristi Analyzer igraju ključnu ulogu u izvedivosti kod analize velikih softvera.

Budući da nije moguće provjeriti sve moguće putove izvršenja u razumnom vremenskom razdoblju, moramo uvesti koncept budžeta analize: procjena vremenskog raspona možemo si priuštiti za analizu određenog dijela koda. Cilj je pronaći što je moguće više neispravnosti uz nisku stopu lažnih pozitivnih procjena. Pokazujemo kako Analizator daje prioritet zanimljivijim putevima za analizu i kako na učinkovit način eliminira neizvedive puteve pomoću rasterećenja u nizu [5].

Analizator također koristi brojne heuristike za automatsko suzbijanje izvješća koja će vjerojatno biti lažna.

Kada se pronađe neispravnost, odgovarajući put i skup ograničenja korisni su za razumijevanje problema. Međutim, nepraktično je predstaviti sve ove informacije korisnicima. Pokazujemo kako Analizator nastoji korisniku prezentirati sažeto, ali uvjerljivo izvješće o neispravnosti kojim se minimizira vrijeme za ispravljanje navedene pogreške.

CODECHECKER

Određujemo skalabilnost statičke analize ne samo u smislu učinkovite upotrebe računalnih resursa, već i u smislu učinkovite upotrebe ljudskih resursa kao što je vrijeme programera. CodeChecker je alat dizajniran za olakšavanje integracije Analyzera i drugih sličnih alata za statičku analizu u sustave izgradnje i kontinuirane integracije petlje. To je također puni sustav upravljanja neispravnostima koji prati pogreške koje su pronašli ovi alati.

S obzirom na konačni budžet vremena razvojnog programera i tisuće izvješća o velikim softverima, važno je prvo ocijeniti izvješća s najboljim povratom ulaganja.

CodeChecker također podržava diferencijalnu analizu koja sprečava programere da uvedu nove neispravnosti bez potrebe da prethodno isprave sva naslijeđena izvješća.

SAŽETAK

U ovom smo radu sažeti svoja iskustva prikupljena pridonoseći najsuvremenijim Static Analyzer Clang i CodeChecker proizvodima. Naša je nada da će se pokazati korisnim resursom za svakoga tko odluči raditi na alatima za statičku analizu.

Literatura

[1] Baldoni, R., Coppa, E., Delia, D.C., Demetrescu, C. and Finocchi, I., 2018. A survey of symbolic execution techniques. ACM Computing Surveys (CSUR), 51(3), p.50.

[2] Clang Static Analyzer, https://clang-analyzer.llvm.org/. Last accessed 4 Nov 2018

[3] CodeChecker, https://github.com/Ericsson/codechecker. Last accessed 4 Nov 2018

[4] King, J.C., 1976. Symbolic execution and program testing. Communications of the ACM, 19(7), pp.385-394.

[5] Kovacs, R., Horvath, G., 2018. An Initial Prototype of Tiered Constraint Solving in the Clang Static Analyzer. Studia Universitatis Babes-Bolyai: Series Informatica, 63:(2) pp. 88-101.

[6] Xu, Z., Kremenek, T. and Zhang, J., 2010, October. A memory model for static analysis of C programs. In International Symposium On Leveraging Applications of Formal Methods, Verification and Validation (pp. 535-548). Springer, Berlin, Heidelberg.

PROGRAMIRANJE U MENADŽMENT I ORKESTRACIJI VIRTUALIZIRANIH MREŽNIH RESURSA

Virtualizacija mrežnih funkcija (eng. Nwwork functions virtualization, NFV) je nova paradigma za promjenu načina izgradnje i rada mreža. Odvajanje softverske implementacije iz mrežnih resursa kroz virtualizacijski sloj uvodi potrebu za razvijanjem seta funkcija za menadžment i orkestraciju NFV-a (MANO). Usredotočiti ćemo se na koordinaciju funkcija menadžmenta implementiranih u različitim funkcionalnim blokovima kako bismo ostvarili pouzdani rad za MANO funkcije koje djeluju u distribuiranim okruženjima. Izazovi su predstavljeni praktičnim primjerom virtualne tehnologije Open Stack i problemima inspiriranim telekomunikacijskom industrijom.

UVOD

Svrha ovih bilješki je uvesti nove koncepte, kao što je Virtualizacija mrežnih funkcija (NFV), koji se trenutno provode u složenim softverskim sustavima i mrežama, objasniti nove izazove i pokazati studentima kako se njima nositi pomoću programskih tehnika za koordinaciju funkcija menadžmenta i orkestracije virtualiziranih mrežnih resursa koji djeluju u distribuiranim okruženjima.

SLOŽENI SUSTAVI

Podloga ovih predavanja nalazi se u teoriji složenih sustava, posebice složenih softverskih sustava i složenih mreža. Predavanja stoga počinju s uvodom u teoriju, pažljivo pozicioniranje razmatranih problema i izazova unutar trenutne evolucije mreža i softverskih sustava. Virtualizacija je paradigma koja se često koristi u upravljanju složenim softverskim sustavima. To podrazumijeva uvođenje novog sloja apstrakcije, virtualnog uređivanja sloja sustava i njegovih funkcija, čime se izbjegava uvođenje ovisnosti između slojeva sustava.

FUNKCIJE MENADŽMENTA I ORKESTRACIJE

U telekomunikacijskim mrežama uvedena je nova paradigma, nazvana virtualizacijom mrežnih funkcija (NFV), koja razdvaja mrežnu funkciju iz fizičkih mrežnih resursa kroz novi virtualizacijski sloj [2]. Međutim, time je uvedena potreba za razvojem skupova funkcija za menadžment i orkestraciju NFV-a (MANO). U tu svrhu određena je posebna radna skupina unutar Europskog instituta za telekomunikacijske norme (eng. European Telecommunications Standards Institute, ETSI). Arhitektonski okvir za menadžment i orkestraciju NFV-a definiran je u [1]. U ovim bilješkama, usredotočiti ćemo se na funkcije menadžmenta i orkestracije koje se provode u različitim funkcionalnim blokovima, kako bi se postigla njihova pouzdana operacija u distribuiranim okruženjima.

PRIMJERI

Ove bilješke pružaju uvod u navedenu temu s ciljem objašnjavanja problema i načela, metoda i tehnika koje se koriste za njihovo rješavanje. Radni primjeri i vježbe služe studentima kao nastavni materijal, od kojih mogu naučiti kako koristiti funkcionalno programiranje za učinkovito i djelotvorno koordinirale funkcije menadžmenta i orkestracije u distribuiranim složenim sustavima pomoću NFV.

Metode i tehnike objašnjene u ovim bilješkama i primijenjene na probleme menadžmenta i orkestriranja mrežne virtualizacije već postoje te mi u ovom smislu ne podrazumijevamo originalnost. Njihova svrha je služiti kao nastavni materijal za navedene metode.

Problemi i izazovi koordiniranja funkcija menadžmenta i orkestracije rješavaju se pomoću platforme OpenStack [3]. To je operacijski sustav oblaka otvorenog koda koji integrira zbirku softverskih modula koji su neophodni za pružanje slojevitih modela oblak računanja. Takva je tehnologija nužna u rješavanju problema koji proizlaze iz paradigme virtualizacije u postojećim mrežama, a studenti koji razumiju rješenja u OpenStacku moći će prenijeti svoje znanje na druge postojeće tehnologije s istom ili sličnom svrhom.

Izazovi koji proizlaze iz novih mrežnih paradigmi, kao i njihova rješenja, prikazani su praktičnim primjenama pomoću virtualne tehnologije OpenStack i potaknute problemima telekomunikacijske industrije. Svi primjeri i vježbe razrađeni su u virtualnoj tehnologiji OpenStack.

Literatura

[1] ETSI Industry Specification Group (ISG) NFV: ETSI GS NFV- MAN 001 v1.1.1: Network Functions Virtualisation (NFV); Management and Orchestration European Telecommunications Standards Institute (ETSI), 2014, https://www.etsi.org/deliver/etsi_gs/NFV- MAN/001_099/001/01.01.01 60/gs_NFV-MAN001v010101p.pdf, accessed July 1, 2018

[2] Han, B., Gopalakrishnan, V., Ji, L., Lee, S.: Network function virtualization: Challenges and opportunities for innovations. IEEE Communications Magazine 53(2), 90–97 (2015)

[3] OpenStack Cloud Software. OpenStack Foundation (2018), www.openstack.org, accessed July 1, 2018

RAČUNARSTVO U OBLAKU I FUNKCIONALNO PROGRAMIRANJE U OBRAZOVANJU

Računarstvo u oblaku je postalo ključna tehnologija i zato dio mnogih kurikuluma računalnih znanosti. U dizajnu aplikacije, mikroservisi su ključni koncept. Funkcionalna dekompozicija koja je intrinzična mikro-uslugama može dobro poslužiti za platforme bez poslužitelja, kao što je AWS Lambda.

MIKRO SERVISI

Sve više i više praktičara preporučuje usvajanje arhitekture mikro servisa pri projektiranju aplikacija u oblaku [1,2]. Nejasno definiran, arhitektonski stil mikro servisa stoji u zajedničkom skupu karakteristika: automatsko postavljanje, pametne krajnje točke nijeme cijevi, decentralizirana kontrola podataka [2]. Nedavni napori za migraciju tradicionalnih aplikacija u oblaku na arhitekturu mikro servisa upozoravaju na povećanu složenost prilikom upravljanja brojnim, iako malim, skladnim uslugama [3]. Ovdje je važno razjasniti: usredotočit ćemo se na orkestraciju kao složenu poslovnu logiku finih zrnatih oblaka. Bez obzira je li njihova implementacija orkestrirana (središnja komponenta koja upravlja izvršenjem) ili koreografirana (svaka usluga djeluje neovisno), svodi se na to hoće li željena aplikacija trebati sinkronu kontrolu ili može tolerirati asinkronu kontrolu. Međutim, s obzirom da su agilnost i fleksibilnost visoko željeni atributi, koreografirana implementacija je poželjna [1].

Međutim, nefunkcionalni aspekti mikro usluga, poput performansi vremena izvođenja, igraju važnu ulogu u implementaciji aplikacije.

Stoga namjeravamo upoznati studente s konceptom profiliranja izvedbe u oblaku u kontekstu AWS Lambda. Nadalje nadopunjavamo funkcionalnu paradigmu AWS Lambda pomoću Haskellovog okvira za kontrolu eksperimenata.

RAČUNARSTVO U OBLAKU ORIJENTIRANO KORISNIKU

Unutar područja računarstva u oblaku važan je aspekt pomoći korisnicima u njihovim odlukama. Takve odluke govore o sljedećim pitanjima:

A. Kako se aplikacija ponaša na virtualiziranim resursima?

B. Koliko virtualnih resursa, koje vrste te od kojeg pružatelja oblak treba nabaviti za implementaciju aplikacija?

C. Koliko dugo? Koliko će to stajati?

Ta se pitanja obično modeliraju kao problem raspoređivanja, radeći pod pretpostavkom da ne postoji a priori znanje o aplikaciji. Jednostavan skup zahtjeva je da je aplikacija uspješno implementirana, a troškovi su minimizirani.

Arhitektura raspoređivanja za aplikaciju u oblaku

BaTS raspoređivanje [4] razvijen je kako bi pomogao korisnicima prilikom implementacije aplikacija u oblaku. Potrebno je samo-raspoređivanje kako bi se to postiglo i redovito provjerava napredak implementacije. U prvoj fazi, BaTS prikuplja statistiku iz uzorkovanja sa zamjenom. Ovdje je potreban samo mali uzorak (30 do 50 zadataka) za izračunavanje srednje vrijednosti i standardne devijacije radnog vremena zadataka na raznim oblicima oblaka. Modul procjene proračuna zatim vrši linearnu regresiju kako bi optimizirao ovu fazu.

Upotreba BaTS metodologije za laganu vizualizaciju

Upoznajemo studente s problemom pomaganja vlasnicima aplikacija koji žele odabrati najbolje izbore u smislu lagane virtualizacije prilikom implementacije svoje aplikacije kao skupa mikro usluga.

AWS Lambda

AWS Lambda je vrlo lagani virtualizirani računalni resurs koju nudi Amazon. Granularnost je na razini funkcija, a preporučeni radni učinak po funkciji poziva je najviše u redu veličine sekundi. Također pretpostavlja ne-blokirajuće ponašanje. Postoji 46 vrsta AWS Lambda s modelom cijena od eura po GB * sec.

AWS API Haskell implementation

Na temelju sveobuhvatne implementacije AWS API-ja [5] Haskell, nastojimo ponoviti metodologiju procjene BaTS-a na laganim virtualiziranim resursima.

Praktični rad

Poučavamo studente povezivanje performansi upravljanja faktorijela na različitim lambda tipovima pomoću uzorkovanja i linearne regresije na krivulje tokova u odnosu na krivulje cijena. Na primjer, trebali bi razmotriti koja vrsta ima najbolju propusnost za najjeftiniju cijenu. Zatim bi trebali identificirati kako učinkovito pronaći najprofitabilniju kombinaciju: najmanji trošak za najbolju izvedbu.

Benchmarking AWS Lambda

AWS Lambda funkcije funkcioniraju u virtualnom okruženju nalik spremniku. Poznato je da je količina računalnih resursa dodijeljenih funkcijama proporcionalna DRAM memoriji koju zahtijeva korisnik. Kako bismo utvrdili kako funkcije Lambda mogu obraditi različita opterećenja, svaki njihov računalni resurs može se usporediti: CPU, memorija, propusnost I / O, propusnost mreže i latencija. Takve mikro usporedbe mogu se izvesti pokretanjem, unutar funkcija, poznatim računalnim, memorijskim, I / O i mrežnim intenzivnim radnim opterećenjem, kao što je izračunavanje prvih N prime brojeva, prikazivanje referentnog toka benchmark usporedbe, vođenje iozone benchmark mjerenja, ili čitanje ili pisanje podataka u AWS S3.

Prijenos utvrđenih metodologija ključan je za obrazovanje. Kao budući rad, željeli bismo podržavati Haskell AWS Lambda funkcije implementirane kroz implementaciju API-ja Haskell AWS.

Literatura

[1] Newman, Sam. Building Microservices. O'Reilly Media, Inc., 2015.

[2] Fowler, M., Lewis, J.: Microservices. http://martinfowler.com/articles/microservices.html (March 2014), Last accessed: 15-08-2018

[3] Balalaie, Armin, Abbas Heydarnoori, and Pooyan Jamshidi. "Migrating to cloud- native architectures using microservices: An experience report." arXiv preprint arXiv:1507.08217 (2015).

[4] AM Oprescu. Stochastic Approaches to Self-Adaptive Application Execution on Clouds. PhD Thesis, Amsterdam, Vrije Universiteit, 2013.

[5] https://hackage.haskell.org/package/amazonka-lambda-1.5.0, Last accessed: 15-08-2018.

MODERNA UGRADNJA GRAMATIKE ATRIBUTA ZA OSIGURANJE TIPA

Gramatički atributi su snažni, deklarativni formalizam koji provodi i razmišlja o programima koji su po dizajnu povoljno modularni. Iako se gramatički sastavljač punog atributa može prilagoditi specifičnim potrebama, njegova implementacija je vrlo ne-trivijalna, a njegovo dugoročno održavanje glavni je zadatak. Zapravo, održavanje tradicionalnog atributa gramatičkog sustava je takav veliki napor da većina sustava koji su predloženi u prošlosti više nije aktivna. Naš pristup implementaciji atributnih gramatika je opisati ih kao prvoklasne građane suvremenog funkcionalnog programskog jezika. Poboljšavamo prethodno rješenje ugrađivanja gramatike atributa temeljenih na zatvaračima tako što ih činimo nenametljivim (tj. nije potrebna nikakva promjena u vrstama podataka definiranim od strane korisnika) i tip-sigurna. Povrh toga postižemo jasniju sintaksu korištenjem suvremenih Haskell proširenja. Vjerujemo da se naše ugrađivanje može primijeniti u praksi za implementaciju elegantnih, učinkovitih i modularnih rješenja za izazove programiranja u stvarnom životu.

UVOD

Atributne Gramatike (AGs) predstavljaju deklarativni formalizam koji je Knuth [7] predložio krajem šezdesetih i omogućava implementaciju i razumijevanje programa na modularan i prikladan način. Konkretni AG se oslanja na gramatiku bez konteksta za definiciju sintakse jezika i na atribute povezane s produkcijama gramatike kako bi definirala semantiku tog jezika. U praksi se koriste pravi programski jezici, kao što je primjerice Haskell [2], kao i

moćni prilični algoritmi ispisa [16], tehnike krčenja šuma [4] i moćni tipovi sustava [11]. Kod programiranja s AGs, modularnost se postiže zbog mogućnosti definiranja i korištenja različitih aspekata računanja kao zasebnih atributa.

Atributi su različite računalne jedinice, tipično vrlo jednostavne i modularne, koje se mogu kombinirati u razrađena rješenja za složene programske probleme. Također se mogu analizirati, ispravljati i održavati samostalno, što olakšava razvoj programa i evoluciju.

AGs su se pokazali osobito korisnima za određivanje računanja stabala: s obzirom na jedno stablo, nekoliko AG sustava kao što je [14,3,8,17] uzimaju specifikacije koje vrijednosti ili atribute treba izračunati i izvršiti za taj izračun. Napor dizajna i kodiranja potreban za izradu, unapređenje i održavanje tih AG sustava je međutim ogroman, što je često prepreka za postizanje uspjeha koji oni omogućuju.

Sve popularniji alternativni pristup uporabi AG-ova oslanja se na njihovo ugrađivanje u programske jezike opće namjene [12, 9, 13, 15, 18, 1]. Time se izbjegava opterećenje implementacije potpuno novog jezika i pridruženog sustava tako što će ga ugostiti u najsuvremenijim programskim jezicima. Slijedeći ovaj pristup, moguće je iskoristiti moderne konstrukcije i infrastrukturu koje su već pružene od strane tih jezika i usredotočiti se na posebnosti specifičnog jezika koji se razvija.

Funkcionalni zatvarač [6] moćna je apstrakcija koja uvelike pojednostavljuje implementaciju algoritama prebacivanja koji obavljaju puno lokalnih ažuriranja. Funkcionalni patentni zatvarači uspješno su primijenjeni za konstrukciju AG ugrađivanja u Haskell [9,10]. Unatoč svojoj eleganciji, ovo rješenje je imalo veliki nedostatak koji je spriječio njegovu upotrebu u stvarnom svijetu primjene: atributi nisu spremljeni, već više puta ponovljeni što ozbiljno narušuje performanse. Nedavno je ovaj nedostatak otklonjen [5] i zamijenjen s drugačijim: pristup je postao nametljiv, tj. da bi se koristilo korisničkim strukturama za ugradnju potrebno ih je prilagoditi.

U ovom radu predstavljamo alternativni mehanizam za predmemoriranje atributa koji se temelje na samoorganizirajućoj beskonačnoj rešetki. Ovaj grafikon položen je povrh korisnički definiranog algebarskog tipa podataka (ADT) i odražava njegovu strukturu. Samostalno definirana vrsta podataka ostaje netaknuta. Ugrađivanje se zatim temelji na dva (a ne jednom) koherentnom patentnom zatvaraču koji paralelno prelaze strukture

podataka. Povrh toga što je ne invazivno, naše je rješenja potpuno sigurno za tipove. Suvremene Haskell ekstenzije kao što su ConstraintKinds omogućuju nam da ograničenja propagiraju u ADT-u i potpuno uklanjaju pretvorbu tipova podataka za vrijeme izvođenja koji su prisutni u prethodnim verzijama.

Druga korist od korištenja suvremenih Haskellovih značajki je čišća sintaksa s manje koda generiranog pomoću predloška Haskell.

Literatura

[1] Balestrieri, F.: The Productivity of Polymorphic Stream Equations and The Com- position of Circular Traversals. Ph.D. thesis, University of Nottingham (2015)

[2] Dijkstra, A., Fokker, J., Swierstra, S.D.: The architecture of the Utrecht Haskell compiler. In: Haskell Symposium. pp. 93–104 (2009)

[3] Dijkstra, A., Swierstra, D.: Typing Haskell with an Attribute Grammar (Part I). Tech. Rep. UU-CS-2004-037, Institute of Information and Computing Sciences, Utrecht University (2004)

[4] Fernandes, J.P., Saraiva, J.: Tools and Libraries to Model and Manipulate Circular Programs. In: Symposium on Partial Evaluation and Program Manipulation. pp. 102–111. ACM (2007)

[5] Fernandes, J.P., Martins, P., Pardo, A., Saraiva, J., Viera, M.: Memoized zipper- based attribute grammars and their higher order extension. Science of Computer Programming (2018)

[6] Huet, G.: The zipper. Journal of functional programming 7(5), 549–554 (1997)

[7] Knuth, D.: Semantics of Context-free Languages. Mathematical Systems Theory 2(2) (June 1968), Correction: Mathematical Systems Theory 5 (1), March 1971.

[8] Kuiper, M., Saraiva, J.: Lrc - A Generator for Incremental Language-Oriented Tools. In: International Conference on Compiler Construction. pp. 298–301. Springer-Verlag (1998)

[9] Martins, P., Fernandes, J.P., Saraiva, J.: Zipper-based attribute grammars and their extensions. In: Programming Languages - 17th Brazilian Symposium, SBLP 2013, Brasılia, Brazil, October 3 - 4, 2013. Proceedings. pp. 135–149 (2013)

[10] Martins, P., Fernandes, J.P., Saraiva, J., Wyk, E.V., Sloane, A.: Embedding at- tribute grammars and their extensions using functional zippers. Science of Computer Programming 132, 2 – 28 (2016), selected and extended papers from SBLP 2013

[11] Middelkoop, A., Dijkstra, A., Swierstra, S.D.: Iterative type inference with attribute grammars. In: International Conference on Generative Programming. pp. 43–52. ACM (2010)

[12] de Moor, O., Backhouse, K., Swierstra, D.: First-Class Attribute Grammars. In: 3rd. Workshop on Attribute Grammars and their Applications. pp. 1–20. Ponte de Lima, Portugal (2000)

[13] Norell, U., Gerdes, A.: Attribute Grammars in Erlang. In: Workshop on Erlang. pp. 1–12. 2015, ACM (2015)

[14] Reps, T., Teitelbaum, T.: The synthesizer generator. SIGPLAN Not. 19(5), 42–48 (Apr 1984)

[15] Sloane, A.M., Kats, L.C.L., Visser, E.: A pure object-oriented embedding of attribute grammars. Electronic Notes in Theoretical Computer Science 253(7), 205–219 (2010)

[16] Swierstra,D.,Azero,P.,Saraiva,J.:Designing and Implementing Combinator Languages. In: Third Summer School on Advanced Functional Programming. LNCS Tutorial, vol. 1608, pp. 150–206. Springer Verlag (1999)

[17] Van Wyk, E., Bodin, D., Gao, J., Krishnan, L.: Silver: an Extensible Attribute Grammar System. Electronic Notes in Theoretical Computer Science 203(2), 103–116 (2008)

[18] Viera, M., Swierstra, D., Swierstra, W.: Attribute Grammars Fly First-class: how to do Aspect Oriented Programming in Haskell. In: International Conference on Functional Programming. pp. 245–256. ACM (2009)

KOLIKO JE ZELEN VAŠ PROCES?

Kao i kod Bio proizvoda, svijet se razvija kako bi postao ekosustav koji je svjestan prirode. Zelena inicijativa definira dva glavna cilja: smanjiti potrošnju energije i koristiti osnovne prirodne izvore u proizvodnji električne energije.

Jedan od izazova za proizvođače baterija je koliko dugo baterija može raditi bez kontinuiranog punjenja. Ima mnogih drugih izazova, kao što je veličina koja uvelike utječe na oblik i težinu uređaja. Baterija se smatra nešto lakšom u usporedbi s uređajem koji treba baterije za rad. Izazov je ovdje kako smanjiti veličinu i težinu, a postići visoku učinkovitost u smislu radnog vremena mobilnog uređaja bez naplaćivanja.

Povrh ovog hardverskog izazova, postoji softver izazov: sam softver trebao bi podržati uštedu energije. Učiniti to bez ograničavanja korisničkog doživljaja danas se smatra tihim, ali važnim ciljem svakog razvojnog softvera koji cilja na bilo koju vrstu prijenosnih uređaja [1]. Ovaj se cilj obično pojavljuje kada se vezani zahtjevi mijenjaju od neizrečenog do važnog problema.

Imajući na umu da potrošnja energije bilo kojeg mobilnog uređaja utječe na pokrenute aplikacije kroz učestalost korištenja usluga do stvarnog raspoloženja korisnika, razvoj softvera za takve uređaje već je izazov [4]. Moglo bi se reći da je izazov za razvoj softvera uvijek isti, ali ovdje moramo istaknuti "mobilnost" kao ključni sustav imovine. Status napajanja baterije također određuje performanse sustava zbog konfiguracije razine operacijskog sustava - dobro poznate kao "postavke za uštedu energije".

Još je teže, ako netko (u našem slučaju učitelj) mora pripremiti učenike za takve izazove [6]. Sve već poznate "najbolje prakse" i "savjeti za uštedu energije" moraju biti predstavljeni u kontekstu, što je lako razumljivo studentima.

To se može učiniti pozicioniranjem koncepata u poznato okruženje kao što je testiranje softvera i automatizacija testiranja [2]. To se namjerava predstaviti ovom pokaznom vježbom, sadržaj je idućih odjeljaka, počevši od prijedloga, preko primjera i završiti savjetima za daljnja poboljšanja.

Glavni fokus je na potrošnji energije radnog softvera i njegovih razvojnih procesa, gdje svaka razvojna faza igra značajnu ulogu.

Uzimajući u obzir bilo koji proces razvoja softvera, energija se troši dok se analiza problema, konstruira i ocjenjuje kod [3]. Softver ili hardverski alati moraju se koristiti za implementaciju monitoringa potrošnje energije za pokretanje softvera na vrhu odabranih operativnih sustava i za procjenu potrošnje energije. Uobičajeni scenariji korištenja su praćenje korištenja energije odabranog softvera [5], ali ćemo također razmotriti mogućnost korištenja tih alata za mjerenje zelenog procesa koji proizvodi konačnu inačicu rada softvera.

Primjeri pokrivaju različite situacije. Polazeći od energetskog profiliranja softvera treće strane u specifičnim scenarijima korištenja, ističemo ključna svojstva (prednosti i mane) postojećih alata. Tijekom testiranja jedinica koda od kojih se softver razvio, predstavljamo tipičnu uporabu softvera za profiliranje energije.

Sposobnost skaliranja mjernog pristupa od profiliranja isječka koda ili jedne aplikacije do analize potrošnje energije lanaca alata posljednji je primjer koji predstavljamo. Ovo predstavlja generički pristup za profiliranje energije i ima za cilj zamijeniti upitnik naslova udžbenika za razdoblje koje predstavlja procjenu rezultata za svaki pojedini slučaj obuhvaćen pokaznom vježbom.

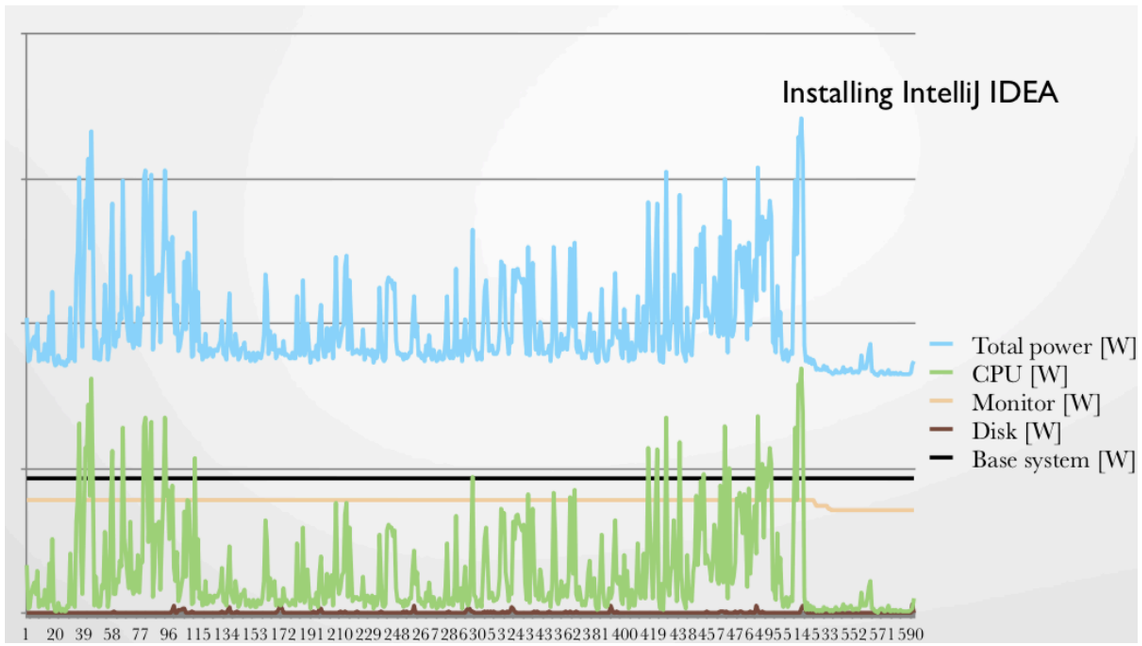
Literatura

[1] D. Li, W. G. J. Halfond, An investigation into energy-saving programming practices for android smartphone app development, in Proc. of the 3rd International Workshop on Green and Sustainable Software, GREENS 2014, ACM, 2014, pp. 46–53.

[2] D. Li, Y. Jin, C. Sahin, J. Clause, W. G. J. Halfond: Integrated energy-directed test suite optimization, in Proc. of the 2014 International Symposium on Software Testing and Analysis, ISSTA 2014, ACM, 2014, pp. 339–350.

The energy-measured development game

1. Setup the environment
2. Start the energy monitor
3. Develop (think, code, test, fix) for 15 minutes
4. Have a 5 minutes break (stop energy usage monitoring, set up the next one, get a coffee)
5. Finish (for this time) if there is no further idea
6. Repeat (jump to label 2)
7. Analyse collected data (energy efficiency of your development process) inside the team



[3] M. Santos, J. Saraiva, Z. Porkolab, D. Krupp: Energy Consumption Measurement of C/C++ Programs Using Clang Tooling, in Proceedings of the SQAMIA 2017: 6th Workshop of Software Quality, Analysis, Monitoring, Improvement, and Appli- cations, Z. Budimac, ed., Belgrade, Serbia, 11-13.9.2017, Paper No. 15, 8 pages, also published online by CEUR Workshop Proceedings No. 1938 ISSN 1613-0073.

[4] J.Saraiva,M.Couto,Cs.Szabo,D.Novak:TowardsEnergy-AwareCodingPractices for Android, Acta Electrotechnica et Informatica, Vol. 18, No. 1, 2018, pp. 19–25. https://doi.org/10.15546/aeei-2018-0003

[5] Cs. Szabo, E.M.M. Alzeyani: Measuring Energy Efficiency of Selected Working Software, Studia Universitatis Babe¸s-Bolyai Informatica, Vol. 63, No. 1, 2018, pp. 5–16. https://doi.org/10.24193/subbi.2018.1.01

[6] Cs. Szabo, J. Saraiva: Focusing software engineering education on green application development, in Conference of Information Technology and Development of Education – ITRO 2017, Novi Sad, Serbia, pp. 165–169, ISBN 978-86-7672-302-7.

FUNKCIONALNO PROGRAMIRANJE UREĐAJA

OBAVIJEST

Ovaj će rad biti proširena verzija našeg doprinosa RWDSL18 [2]. U trenutnom ćemo radu upotrijebiti isti DSL, a izvršena su samo neka manja proširenja i poboljšanja. Trenutni rad će se usredotočiti na to kako se mTask DSL može koristiti za programiranje interneta stvari (eng. Internet of Things, IoT) i raspravljati o simulatoru visoke razine za mTask programe kao program iTask.

UVOD

Mnogi uređaji danas su opremljeni jednostavnim mikroprocesorom koji kontrolira njihovo ponašanje. Tipični primjeri su termostati, žarulje, električne utičnice, vatrodojavni alarmi, otvori za vrata i tako dalje. Kada ti uređaji mogu komunicirati jedni s drugima, ili nekim udaljenim računalima, za njih se kaže da su dio Interneta stvari, IoT. Mikro računala u tim uređajima su vrlo pristupačna i postaju sveprisutna. Skupi uređaji kao što su automobili i uređaji s vrlo složenim zadatkom opremljeni su potpunim ugrađenim računalom i odgovarajućim softverom. Za većinu malih i relativno jeftinih I / O uređaja takvo ugrađeno računalo je preskupo ili troši previše energije; za izvršavanje softvera koristi se jednostavan i jeftin mikroprocesor. Takvi sustavi imaju vrlo ograničenu računalnu snagu i memoriju, obično 30 KB do 4 MB flash memorije za spremanje programa. Životni vijek takve memorije ograničen je na 1000 ciklusa pisanja. Za spremanje varijabli, hrpa i stogova sustavi imaju 2 do 40 KB RAM-a.

Brzina procesora i ograničenja memorije isključuju korištenje operativnog sustava. Uređaj samo izvršava program koji upravlja uređajem. Čak i kontrolni programi na ovim IoT uređajima sastoje se od nekoliko zadataka. Primjerice, provjera stanja gumba deset puta u sekundi, ažuriranje zaslona svake sekunde, mjerenje temperature dva puta u minuti i promjena temperature nakon najmanje pet minuta, osim ako prethodno nije pritisnut gumb. Zbog različitih vremenskih okvira i zavisnosti tih zadataka, upravljački program obično postaje prilično neuredan, neovisno o upotrijebljenom programskom jeziku. Štoviše, IoT uređaji izvršavaju zasebne programe za ostatak aplikacije u internetskoj postaji i komuniciraju koristeći mnoštvo protokola. To čini razvoj i održavanje IoT aplikacija složenim i podložnim neispravnostima.

Programiranje temeljeno na zadatcima (eng. Task Oriented Programming, TOP) nudi lagane thread-ove koje se lako mogu sastaviti prema složenijim zadacima. Zadaci se procjenjuju korak po korak i mogu se nakon takvog koraka pregledati trenutne vrijednosti drugih zadataka. TOP se prvo provodi u sustavu iTask [4,5] koji je ugrađen u Clean [6]. U sustavu iTask, primitivni zadatci prikupljaju podatke putem automatskog generiranog web-obrasca ili prikupljanjem podataka iz drugih programa i spremišta podataka. Snažan skup kombinatora koristi se za sastavljanje zadataka u složenije zadacima. U ovom radu pokazujemo da je TOP prikladan za programiranje IoT uređaja. Primitivni zadatci daju trenutnu vrijednost ulaza i senzora. Konstruktori vrlo slični sustavu iTask koriste se za kombiniranje zadataka složenijim zadacima.

IoT uređaji obično imaju labavo ovise zadatke koji kontroliraju senzore, aktuatore i komunikaciju uređaja. Programiranje u TOP stilu nudi koncizne programe. Izvođenje ovih zadataka unutar ograničenja malih mikrokontrolera s vrlo ograničenom snagom procesiranja i nekih KB-ova RAM memorije zaslužuje pozornost. Zbog teških ograničenja korištenih mikrokontrolera, ne možemo priključiti sustav iTask na IoT uređaje jer tipičan program iTask zahtijeva oko 100 MB prostora hrpe. Definiramo ugrađeni specifični jezik domene, eDSL, koji se zove mTask za IoT uređaje. Ovaj eDSL ugrađujemo u sustav iTask budući da namjeravamo napraviti ove TOP jezike potpuno interoperabilne.

Doprinosi ovoga rada bit će:

- Ovaj rad uvodi funkcionalni programski jezik temeljen na zadatcima za IoT uređaje. U usporedbi s našim prethodnim jezikom za programiranje mikroprocesora [3] imperativna periferna kontrola zamijenjena je referentnim transparentnim konstruktima.

- Pokazujemo kako napraviti funkcionalni proširivi, više-pregledni, sigurni za tipove, ugrađeni DSL. Ovo eDSL bez oznake [1].

- Generirani kod radi na malim i sporim uređajima, kao i na većim strojevima i simuliranom stroju.

- Zbog upotrebe Arduino C++ kao posredničkog jezika, ovaj funkcionalni eDSL radi na mnogim različitim mikrokontrolerima.

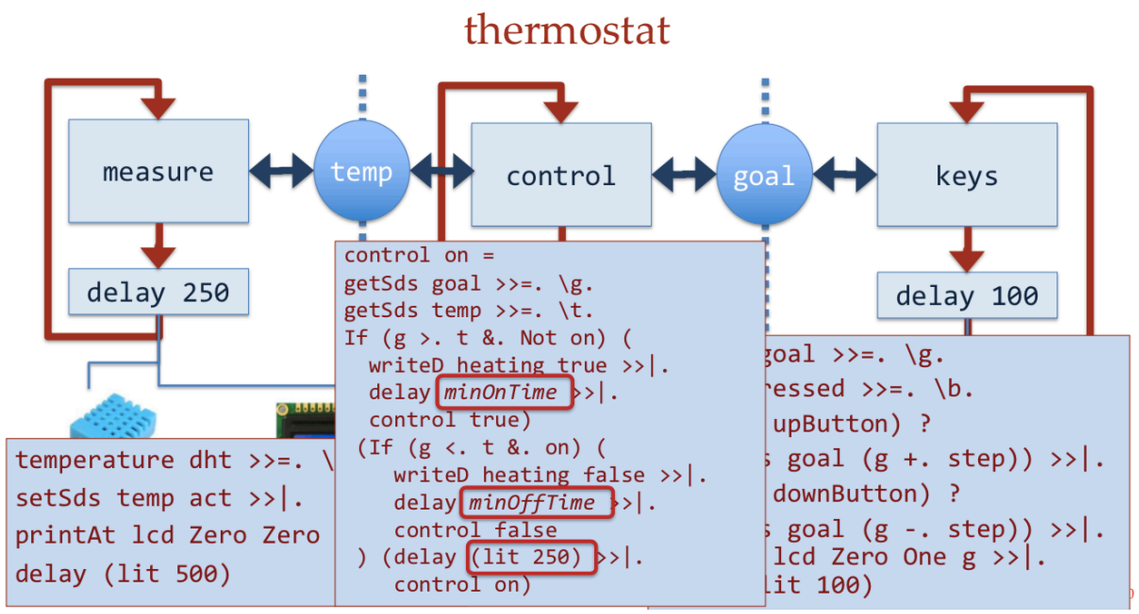
- Simulacija mTask programa na visokoj razini u programu iTask nudi mogućnost pregleda efekta eDSL programa i manipulacije simuliranim okruženjem za eksperimentiranje sa zadanim ponašanjem. U takvom simulatoru puno je lakše manipulirati vremenom i senzorima nego u stvarnom životnom okružju, npr. promijenimo temperaturu koju je senzor prenio pritiskom na gumb umjesto fizički izložiti IoT uređaju na te temperature.

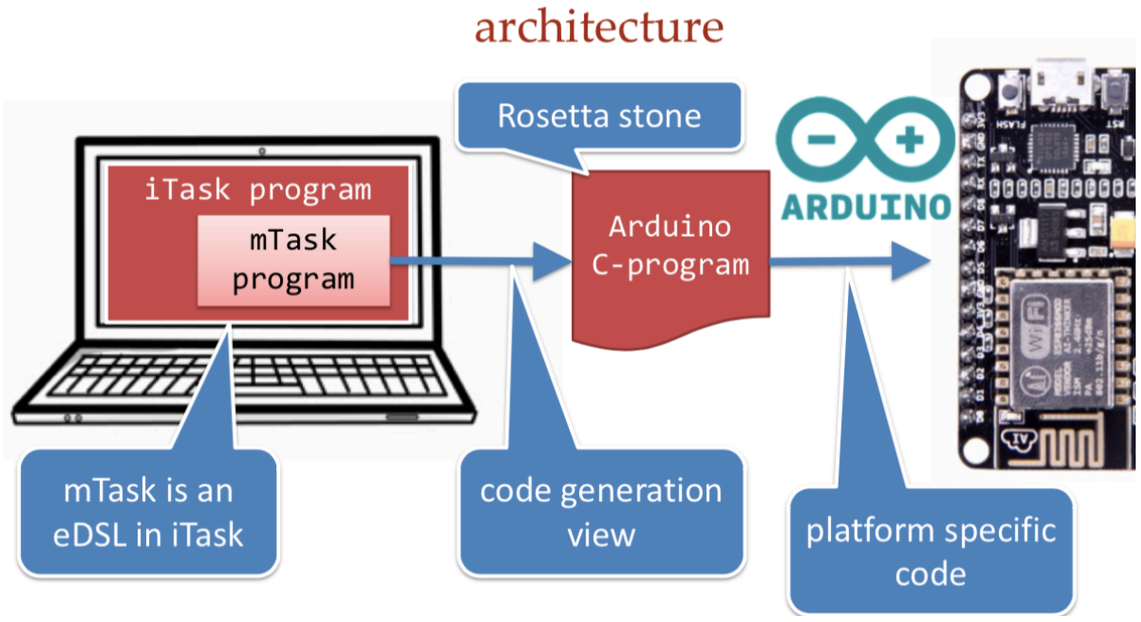
Literatura

[1] Carette, J., Kiselyov, O., Shan, C.c.: Finally tagless, partially evaluated: Tagless staged interpreters for simpler typed languages. J. Funct. Program. 19(5) (Sep 2009). https://doi.org/10.1017/S0956796809007205

[2] Koopman, P., Lubbers, M., Plasmeijer, R.: A task-based dsl for microcomputers. In: Proceedings of the Real World Domain Specific Languages Workshop 2018. pp. 4:1–4:11. RWDSL2018, ACM, New York, NY, USA (2018). https://doi.org/10.1145/3183895.3183902, http://doi.acm.org/10.1145/ 3183895.3183902

[3] Koopman, P., Plasmeijer, R.: A shallow embedded type safe extendable dsl for the arduino. In: Revised Selected Papers of the 16th International Symposium on Trends in Functional Programming - Volume 9547. pp. 104–123. TFP 2015, Springer-Verlag New York, Inc., New York, NY, USA (2016), http://dx.doi.org/ 10.1007/978-3-319-39110-6_6





[4] Plasmeijer, R., Achten, P., Koopman, P.: iTasks: executable specifications of inter- active work flow systems for the web. In: Hinze, R., Ramsey, N. (eds.) Proceedings of the ICFP'07. ACM, Freiburg, Germany (2007)

[5] Plasmeijer, R., Lijnse, B., Michels, S., Achten, P., Koopman, P.: Task-oriented programming in a pure functional language. In: Proceedings of the 14th Symposium on Principles and Practice of Declarative Programming. pp. 195–206. PPDP '12, ACM, New York, NY, USA (2012). https://doi.org/10.1145/2370776.2370801, http://doi.acm.org/10.1145/2370776.2370801

[6] Plasmeijer, R., van Eekelen, M., van Groningen, J.: Clean language report (version 2.2) (2011), http://clean.cs.ru.nl/Documentation

RAZUMIJEVANJE KODOVA S CODECOMPASS-OM

PREGLED

Razvoj i održavanje su dvije odvojene faze s različitim karakteristikama, tako da zahtijevaju i drugačiju podršku alatima. Tijekom razvoja uglavnom pišemo novi kôd koji zahtijeva podršku alata kao što su kompilacija koda, brace matching i sl. te obično samo nekoliko datoteka koje su više ili manje uključene na istu razinu apstrakcije. Tijekom održavanja uglavnom čitamo ili se krećemo kroz postojeću bazu koda velikog broja modula i datoteka na različitim razinama apstrakcije [1]. Prilikom razvoju namjere su jasne, za razliku od razumijevanja programskog koda gdje je zadatak da se pronađe izvorna svrha određenih fragmenata koda.

U industrijskom okruženju [2] projekt može sadržavati milijune redaka koda. Za dugo postojeće velike sustave, gdje je baza kodova razvijena i održavana već desetljećima od strane timova koji fluktuiraju, originalne namjere su izgubljene, dokumentacija nije pouzdana ili nedostaje, a jedini pouzdan podatak je sam kod. Razumijevanje takvih velikih softverskih sustava vrlo je važan, ali obično i vrlo zahtjevan zadatak. To podrazumijeva potrebu podrške odgovarajućih alata [3].

Prilikom upoznavanja nepoznatog izvornog koda, prvi korak je pronaći relevantne dijelove sustava. Ovaj proces zahtijeva brzu lokalizaciju značajki, temeljenu na nekim entitetima pribavljenim iz dnevnika ili drugih resursa. Sljedeći je korak proširiti naše znanje o sustavu kroz dijagrame, funkcijske pozivne lance itd. Na kraju želimo potvrditi prikupljeno znanje putem poruka sustava za upravljanje inačicama, arhitektonskih informacija i referenci na povezanim modulima.

CODECOMPASS

CodeCompass [4, 5] je okvir otvorenog koda za razumijevanje programskog koda. On pruža proširivu arhitekturu kako bi se omogućilo dodavanje raznih alata analizatora koji proizvode različite vizualizacije, prikupljanje informacija, mjerenja [6] itd. Najvažniji projektni cilj bio je u prilagoditi CodeCompass za industrijske projekte velikih razmjera.

U prvom koraku proizvod mora biti analiziran: svi se podaci prikupljaju i pohranjuju u bazu podataka koja zatim omogućuje da sloj usluga pruži potrebne vizualizacije. Za brzo pretraživanje CodeCompass koristi indeksiranje teksta koje rezultira navigacijom u izvornom kodu koja je neovisna o jeziku. Budući da je primarni cilj dati točne informacije o jezičnim elementima, identifikacija simbola po njihovom imenu nije dovoljna. Koristimo LLVM prevoditeljsku infrastrukturu kako bismo identificirali simbole precizno i riješili imenovane entitete koristeći abstract syntax tree. CodeCompass se proširuje putem parsera jezika. Najviše podržanih jezika su C / C ++, ali Java i Python su djelomično obrađeni. Osim imenovanih simbola, u bazu podataka se pohranjuju i neke dodatne informacije, kao što su odnosi između AST čvorova (funkcionalni pozivi, nasljedstva) i datoteka (odnos sučelja, uključivanje, itd.). Oni se koriste za prikaz slike arhitektonske razine o sustavu na temelju uporabe simbola [7].

Kodna baza nije jedini izvor dokumentacije. Poruke izmjena unutar sustava za praćenje inačica također sadrže informacije koje su važne za razumijevanje zašto su se određene promjene dogodile na danom modulu. CodeCompass također čita Git-ovo spremište, ako ih ima. CodeCompass je opremljen i još nekim naprednim funkcionalnostima. Može prikazati funkcije generirane kompajlerom, koje nedostaju iz izvora. Analiza pokazivača pomaže u razumijevanju koje se varijable odnose na isti objekt. Možemo provjeriti odnose funkcija poziva, čak i ako se oni pozivaju putem virtualne funkcije ili funkcijskog pokazivača.

SAŽETAK

Za razumijevanje kodova potrebna je posebna podrška alatima za razumijevanje velikog softvera. Pregled i kategoriziranje alata za razumijevanje kodova prema arhitekturi i funkcionalnostima kako bismo ispitali njihove mogućnosti.

Upoznajemo se s CodeCompass koji predstavlja širok spektar funkcionalnosti o vizualizaciji, informiranju, kontroli verzije i prikupljanju dokumentacije, mjernim podacima itd.

Literatura

[1] Jonathan Sillito, Gail C. Murphy, Kris De Volder. (2008). Asking and Answering Questions during a Programming Change Task. IEEE Transactions on Software Engineering, VOL. 34, NO. 4, July/August 2008.

[2] Porkolab, Zoltan & Brunner, Tibor & Krupp, Daniel & Csordas, Marton. (2018). Codecompass: an open software comprehension framework for industrial usage. 361- 369. 10.1145/3196321.3197546.

[3] Nathan Hawes, Stuart Marshall, Craig Anslow. (2015). CodeSurveyor: Mapping LargeScale Software to Aid in Code Comprehension. 2015 IEEE 3rd Working Conference on Software Visualization (VISSOFT) , 27-28 Sept. 2015.

[4] Porkolab,Zoltan & Brunner,Tibor (2018). The codecompass comprehension framework. 393-396. 10.1145/3196321.3196352

[5] CodeCompass, https://github.com/Ericsson/CodeCompass. Last accessed 5 Nov 2018.

[6] Brunner, Tibor & Porkolab, Zoltan. (2017). Two Dimensional Visualization of Soft- ware Metrics. Proceedings of the Sixth Workshop on Software Quality Analysis, Monitoring, Improvement, and Applications.

[7] B. De Alwis and G.C. Murphy. (1998). Using Visual Momentum to Explain Dis- orientation in the Eclipse IDE. Proc. IEEE Symp. Visual Languages and Human Centric Computing, pp. 51-54, 2006.

OBRASCI FUNKCIONALNOG PROGRAMIRANJE ZA RAČUNARSTVO VISOKIH PERFORMANSI

Jeizci funkcionalnog programiranja pružaju alate i značajke dizajniranja i primjene distribuiranih aplikacija. Budući da funkcionalni programi imaju svojstvenu paralelnost, ona se može iskoristiti za postizanje pouzdane konkurentne obrade s distribucijom i koordinacijom na visokoj razini.

Suvremeni razvoj paraleliziranog softvera opsežno primjenjuje različite metodologije i pristupe za postizanje velika ubrzanja. Međutim, konkurentnost ostaje jedna od najtežih domena, posebno u slučaju pristupa funkcionalnom programiranju.

Glavna svrha je istražiti obrasce paralelnog računanja u novom okruženju, kako bi ilustrirali primjerenost i primjenjivost FP-a u modernim distribuiranim računalnim okruženjima. Skup poznatih obrazaca paralelnih algoritama se testiraju kao HPC komponente. Značajan broj primjera omogućuje veće ubrzanje. Količina paralelizma uvijek ovisi o mnogim čimbenicima kao što su: primijenjeni obrazac izračuna, rafinirana zrnatost, semantika distribuiranih čvorova, prijenos podataka. Primjeri će pregledati i dobro modeliranu koordinaciju i semantičku ispravnost.

DOMENE PRIMJENE ZA OBRASCE

Koordinacija

Specifična tema distribuiranog i paralelnog funkcionalnog računanja zahtijeva elemente koordinacije jezika. Ranija istraživačka pitanja bavila su se načinom postizanja paralelnosti i komunikacije funkcionalnih programa na višim razinama. Uvedeni elementi funkcionalnih programskih jezika s većom razinom apstrakcije pokazali su se izvedivim za paralelne izračune visoke razine intenzivnih podataka [2].

Smjerovi koji su uslijedili rezultirali su pojačanim izražavanjem snaga jezičnih elemenata za paralelizam u funkcionalnim programima. Konkretnije, doveli su do dizajna jezičnog nastavka DClean za raspodijeljeno programiranje i koordinaciju funkcionalnih programa Clean. Proširenje se sastoji od jezičnih elemenata na visokoj razini koji koordiniraju čiste funkcionalne računske čvorove u dizajniranom raspodijeljenom okruženju na klasterima.

Konstrukcije generiraju računske okvire povezane putem puniranih komunikacijskih kanali. Korištenje DClean-a programeri pokazuju kako se obrasci raspodijeljenog računanja organiziraju u generirani raspodijeljeni graf, i kontroliraju tok podataka procesne mreže kroz upisane kanale. Jezik nudi prednost pisanja distribuiranih i funkcionalnih aplikacija bez da se programer mora upoznati s detaljima višeslojnog okruženja i tehničkim aspektima usluga middleware-a. Raspodjela rada vrši se prema unaprijed definiranoj shemi paralelnog računanja, algoritamskom obrascu, parametriziranom prema funkcijama, vrstama i ulaznim tokovima.

Glavna svrha uvođenja jezika za koordinaciju je definicija obrazaca za funkcionalno paralelno računanje. U velikom broju primjera ostvareno je veliku ubrzanje. Stvarni iznos paralelizma je podložan redoslijedu stvaranja kanala, količini posla na njima, brzini dohvaćanja i pohrane podataka te složenost čvorova [1].

Grafička vizualizacija raspodijeljenog računanja uz pomoć alata za razumijevanje izvršnog semantičkog koda [4] bio je nezamjenjiv u stvarnim distribuiranim aplikacija. Na taj način prikazana je očekivana paralelizacija računskih okvira i kanala generiranih pomoću dobro definirani obrazaca na

visokoj razini. Istovremeno, cilj je bio modeliranje i formuliranje svojstava operativne semantike DClean-a [3].

Cyber fizički sustavi

Pristup funkcionalnom modeliranju utemeljenom na obrascima koji koriste koordinacijske jezike u današnje vrijeme primjenjuju se u modernim prototipiva CPS sustava. Proučavanjem odnosa između CPS-a i distribuiranih sustava ili CPS-a i ugradbenih sustava važni su za poduzimanje odgovarajućih odluka u koracima dizajna i modeliranja prototipa sofisticiranog CPS sustava.

Studije slučaja primjene CPS sustava [5] opisuju suradnju računskih jedinica koje kontroliraju fizičke entitete (senzore) i njihove veze s drugim složenim sustavima. Sustav pametne kuće CPS uspostavlja nove aspekte, svojstva i pristupe u općoj izradi prototipa koristeći obrasce. U takvom dizajn CPS sustava važna pitanja semantike rješavaju se iz probabilističkog i ponašajnog gledišta, pri čemu je interoperabilnost glavna osobina za analizu i definiciju.

OBRASCI ZA RAČUNARSTVO VISOKIH PERFORMANSI

Istraživanje o proširenju primjenjivosti obrazaca u okruženje računarstva visokih performansi ključna je točka u paralelnom fp pristupu. Prilagođavanje ranijeg znanja o obrascima za programiranje heterogenih više-jezgrenih sustava rezultira većim ubrzanjima, pri čemu mjerenja i usporedbe ocjenjuju nove procese paralelizacije.

Prototipovi obrazaca definirani su u smislu njihovih funkcionalnosti i koordinacije. Studije slučaja ilustriraju veze s drugim vrstama distribuiranih sustava, koji su važni zbog višeslojnosti njihove strukture. Svojstva distribuiranog sustava koja su ponuđena u izvedbenoj formi testiraju se pomoću obrazaca funkcionalnog i distribuiranog programiranja klastera i mreža.

Literatura

[1] Zsok V.: D-Clean Semantics for Generating Distributed Computation Nodes, Work- shop on Generative Technologies, WGT 2010, Satellite workshop at ETAPS 2010, Paphos, Cyprus, March 27, 2010, pp. 77–84.

[2] Zsok V., Hernyak Z., and Horvath, Z.: Designing Distributed Computational Skeletons in D-Clean and D-Box. Central European Functional Programming School CEFP 2005, First Summer School, Budapest, Hungary, July 4-15, 2005, Revised Selected Lectures, LNCS Vol. 4164, Springer-Verlag, 2006, pp. 223–256.

[3] Zsok V., Koopman, P., Plasmeijer, R.: Generic Executable Semantics for D-Clean, Proceedings of the Third Workshop on Generative Technologies, WGT 2011, ETAPS 2011, Saarbrucken, Germany, March 27, 2011, ENTCS Vol. 279, Issue 3, Elsevier, December 2011, pp. 85–95.

[4] Zsok V., Porkolab Z.: Rapid Prototyping for Distributed D-Clean using C++ Tem- plates, Annales Universitatis Scientiarum Budapestinensis de Rolando Eotvos Nominatae, Sectio Computatorica, Eotvos Lorand University, Budapest, Hungary, 2012, Vol. 37, pp. 19–46.

[5] Zsok V. et al.: Modeling CPS Systems using Functional Programming, Proc. of IFL17, Uni. of Bristol, pp. 168–174.