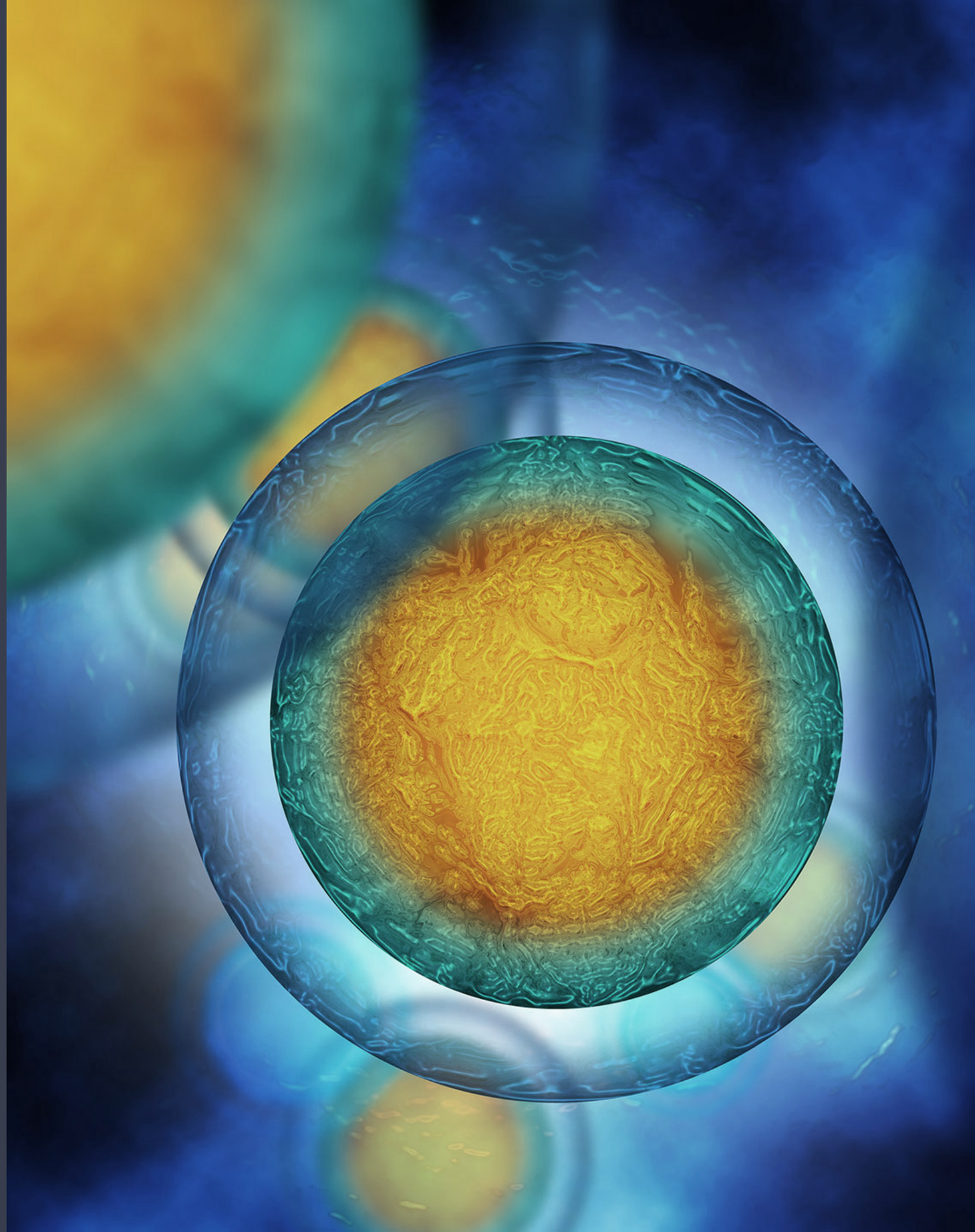
FE3CWS

MATERIÁL K ZIMNEJ ŠKOLE TRI “CO”

Intelektuálny výstup č.1
ERASMUS+ projektu číslo 2017-1-SK01-KA203-035402



Niekoľko slov

O OBSAHU

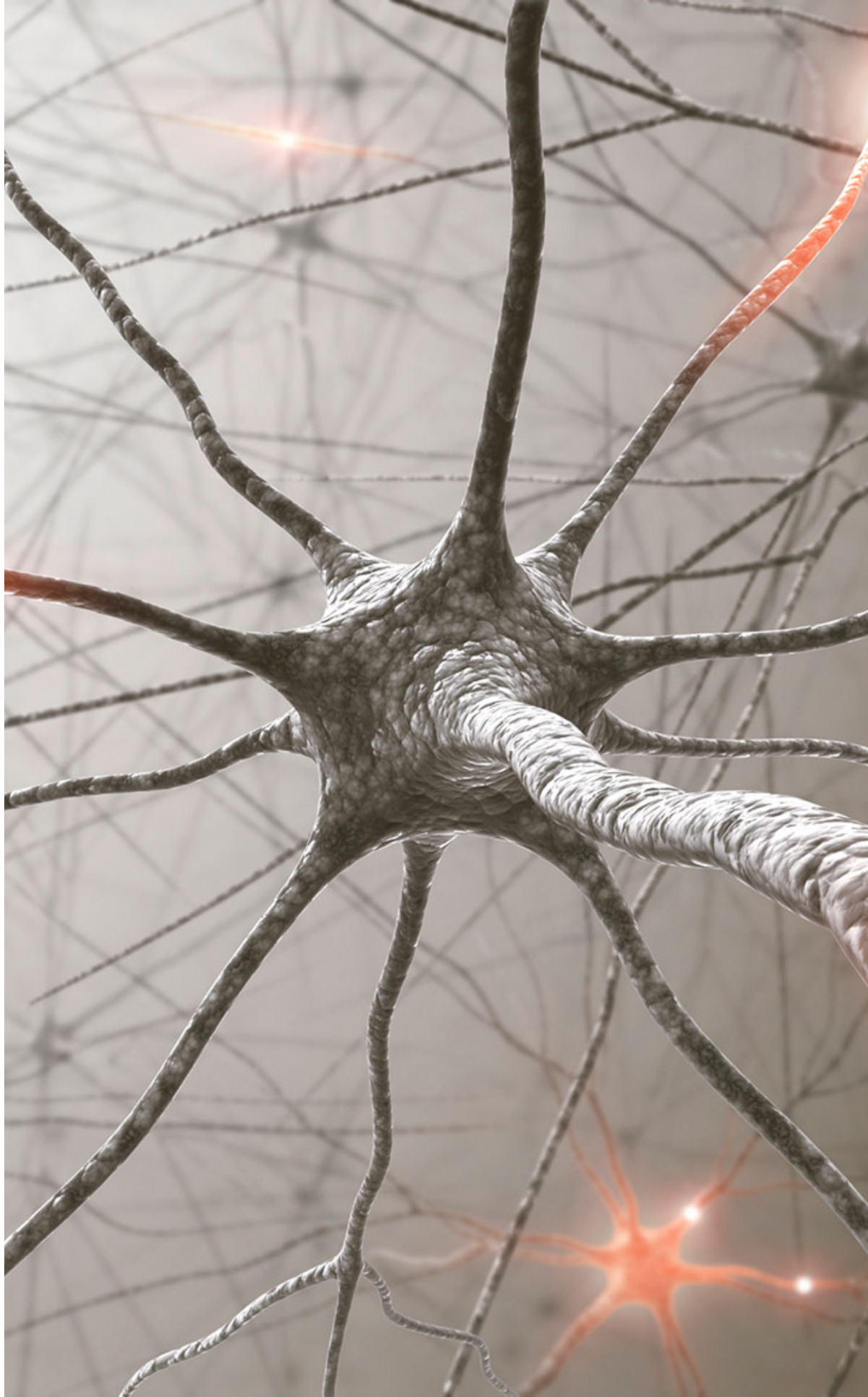
- 9 tém o kompozícii, pochopení a korektnosti softvéru
- 10 autorov zo 7 európskych univerzít z Holandska, Chorvátska, Maďarska, Portugalska a Slovenska
- Dostupný v 7 jazykoch: anglický, maďarský, slovenský, chorvátsky, rumunský, bulharský a portugalský

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

Zimná škola tri “CO” (Composability, Comprehensibility and Correctness Winter School, 3COWS) je prvým intenzívnym školením vysokoškolských študentov a pedagógov, ktorá rozširuje komunitu Central European Functional Programming (CEFP). Uskutočnila sa v rámci implementácie ERASMUS+ projektu číslo 2017-1-SK01-KA203-035402 s názvom “Focusing Education on Composability, Comprehensibility and Correctness of Working Software”, a to v období medzi 22. a 26. januárom 2018.

Materiály obsiahnuté v tejto publikácii boli vytvorené a prezentované v rámci vyššie uvedeného projektu. Táto brožúra je tlačenou obdobou intelektuálneho výstupu č. 1 tohoto projektu.

OBSAH



STROJOVÉ SPRACOVANIE PRÍRODNÉHO JAZYKA

Výstavba mysliaceho stroja je veľkou výzvou. Vzťah jazyka a mysle je zaujímavý pre počítačových vedcov a diskutuje sa v širšom kontexte aj s historikmi, psychológmi, lingvistami a filozofmi.

Renfrew napr. charakterizuje človeka ako symbolickú bytosť keď skonštatuje, že výmena symbolov počas komunikácie je príčinou vzniku a vývoja prírodného jazyka. Gardenfors [14] uvádza, že jazykové koncepty majú hierarchickú štruktúru. Evolúcia gramatík [24] či genetická evolúcia jazykov [15] riešia problém štrukturálnej expanzie a sú použiteľné iba v prípade jednoduchých jazykov, aj to iba v deterministickej forme. Genetické reťazce ako hodnoty sémantických konklúzií predstavujú perspektívnejší prístup k evolúcii jazyka ako priama aplikácia analógie s ľudským mozgom.

Zistili sme aj to, že proces evolúcie prebieha na viacerých metaúrovniach jazykov.

Chomského hypotéza existencie univerzálnej gramtiky pre všetky prírodné jazyky [7] je veľmi zaujímavá a motivujúca, ale táto univerzálna gramatika by sa nemala predstavovať ako pevná a parametrická gramatická štruktúra. Bude to skôr univerzálny algoritmus, teda deterministický proces na úrovni meta-jazyka, ktorý je parametrizovaný hodnotami reprezentujúcimi zmysluplné (sémantické) hodnoty.

Edelman [12] píše, že centrála myslenia stroja by mala disponovať jazykovým centrom, ktoré je súčasne symbolické aj výpočtové. Vyjadril tým to, že myslenie je súčasne symbolický aj dynamický proces akceptujúci možnú zmenu jazyka. V Shaumyanovej aplikatívnej univerzálnej gramatike [34] je jazyk taktiež vyjadrený formou dynamických aplikatívnych procesov.

Výhodou teórie sémiotických jazykov je, že syntax a sémantika sú navzájom viazané a nie sú oddeliteľné. Samozrejme, toto hovorí o potrebe zohľadniť sémantické kategórie na vyjadrenie zmeny jazyka.

Podľa nášho názoru poskytuje sémiotická teória jazykov príležitosť pre deterministický vývoj mysle strojov pre prípad podmnožín prírodných jazykov a formálnych jazykov a to v rôznych formách používaných v komunikácii. V tomto zmysle môže byť komunikácia založená na jazykoch zjednotená medzi ľuďmi a počítačmi a medzi počítačom a druhým počítačom. Východiskový bod tejto myšlienky obmedzený na bežné jazyky sme prezentovali v [17]. V súčasnej dobe poznáme algoritmus na transformáciu jazykových konceptov na vnútorný jazyk strojovej mysle a sme tiež schopní odvodiť pojmy z tohto vnútorného jazyka. Tento vnútorný jazyk je analógia s vnútorným jazykom ľudského myslenia - je to pokojný jazyk v pozadí hlasného prirodného jazykom. V súčasnosti nepoznáme pravidlá konceptualizácie ani pravidlá týkajúce sa úvah o konceptoch reprezentovaných v strojovej mysli. Máme však jasnú metodológiu vývoja strojovej mysle, ktorá je reprezentovaná aplikačnými dynamickými procesmi s vysokým stupňom paralelizmu, ktoré reprezentujú informácie o čistej jazykovej substancii. Navyše zvýšenie abstrakcie jazykových konceptov znižuje počet meta-operácií a zvyšuje počet aplikačných väzieb.

Diskutujeme o efektívnom algoritme detského myslenia a odhadujeme rýchlosť toku informácií v ľudskom mozgu na 300 biliónov signálov za sekundu. Potom tu predstavujeme slovenskú textovú a fonetickú gramatiku, ako aj metódu prekladu textu na reč. Ďalej ilustrujeme proces získavania jazykových prvkov rôznych abstrakcií čítaním krátkych vzoriek textu a pomocou modelových hierarchických tabuliek typu hash. Takisto vyhodnocujeme proces získavania masívnych blokov textov vybranej knihy a dospejeme k záveru, že získanie jazyka s hierarchickou abstrakciou prináša neredundantný graf s rovnakým počtom sekvenčných a paralelných väzieb bez potreby fyzického ukladania dát.

Ďalej používame gramatický prístup k rozpoznaniu vizuálnych objektov. Po prvé musíme zistiť, ako opísať objekty, a potom môžeme na tieto údaje aplikovať metódu abstrakcie. Zameriavame sa predovšetkým na popis 3D objektov pomocou gramatiky. V teórii gramatík sa tento krok nazýva symbolizácia. Symbolizácia zabezpečuje popis objektu a poskytuje základnú abstraktnú vrstvu údajov.

Ako môžeme vidieť, abstrakcia dát pomocou funkcionálneho jazyka nám umožňuje abstraktne a ľahko spracovávať objekty.

Aplikačný prístup môže byť použitý pri spracovaní jazyka aj v prípade keď používame kontextové gramatiky. Ukazujeme algoritmus, ktorý je schopný transformovať akúkoľvek kontextovú gramatiku na superkombinantnú formu. Výsledná forma závisí od formy vstupnej gramatiky, preto vzniká nový problém: nájdenie správnej gramatiky pre danú úlohu.

V krátkosti ukážeme výsledné formy superkombinátorov rôznych typov gramatík a porovnávame ich vlastnosti. Porovnávame aj efektívnosť algoritmov v prezentovaných príkladoch gramatík a ukážeme, že náš algoritmus možno zlepšiť v prípade, že použijeme gramatiky bez akýchkoľvek cyklov. Taktiež diskutujeme o výsledných superkombinantných formách z hľadiska kompresie gramatiky a opätovnej použiteľnosti prvkov, ktoré sú výsledkom spracovania pomerne veľkých vstupných vzoriek textov.

Literatúra

[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]6Edelman, 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)

STATICKÁ ANALÝZA KÓDU NÁSTROJOM CODECHECKER

PREHĽAD

Symbolické vykonávanie [4] je populárna metodika statickej analýzy používaná ako pri overovaní programu tak pri nástrojoch na detekciu chýb. Funguje tak, že interpretuje kód, zavedie symbol pre každú hodnotu neznámu v čase kompilácie (napr. vstupy zadané používateľom) a vykoná výpočty symbolicky. Analytický engine sa snaží súčasne preskúmať viacero ciest vykonania, hoci kontrola všetkých ciest je kvôli obrovskému počtu možností nepostrádateľným problémom.

Zatiaľ čo existuje bohatá literatúra o nástrojoch na overenie programu, nástroje na vyhľadávanie chýb sa zvyčajne musia vyrovnať s prieskumnými dokumentmi o jednotlivých technikách [1]. V tomto príspevku diskutujeme nielen o jednotlivých metódach, ale aj o tom, ako sa tieto rozhodnutia navzájom ovplyvňujú a posilňujú a vytvárajú systém, ktorý je významnejší ako súčet jeho častí. Zameriavame sa na nástroj na vyhľadávanie chýb s názvom Clang Static Analyzer [2] (ďalej len analyzátor) a na infraštruktúru postavenú pod názvom CodeChecker [3]. Dôraz sa kladie na dosiahnutie škálovateľnosti “od konca po koniec” (angl. end-to-end).

Spomenieme časové a pamäťové analýzy, prezentáciu chýb používateľom, automatické potlačenie falošne pozitívnych výsledkov, prírastková analýza, objavovanie vzoriek vo výsledkoch a použitie v systémoch kontinuálnej integrácie. Taktiež načrtneme budúce smerovanie a otvorené problémy týkajúce sa týchto nástrojov. Hoci analyzátor dokáže spracovať iba kód v C/C++/Objective-C, techniky uvedené v tomto dokumente sú jazykovo nezávislé a použiteľné aj pre iné podobné nástroje statickej analýzy.

CLANG STATIC ANALYZER

Zhrnieme pracovný mechanizmus symbolického vykonávania a jeho implementáciu v analyzátore. Diskutujeme o jeho reprezentácii v pamäti [6], o manipulácii s väzbami medzi hodnotami a pamäťovými umiestneniami a o zastúpení kontrolných špecifických stavov v ňom (kde kontrolou máme na mysli jeden modul napísaného analyzátora na nájdenie konkrétneho typu chyby). Zavádzame aj koncept symbolických výpočtov. Voľba reprezentácií používaných analyzátorom zohráva kľúčovú úlohu pri realizácii rozsiahlej softvérovej analýzy.

Vzhľadom na to, že v primeranom čase nie je možné vykonať kontrolu všetkých možných spôsobov vykonávania, musíme predstaviť koncept rozpočtu analýzy: odhad časového rozpätia, v ktorom si môžeme dovoliť analyzovať daný kus kódu. Cieľom je nájsť čo najviac chýb s nízkou mierou falošne pozitívnych zistení. Ukazujeme, ako analyzátor uprednostňuje zaujímavejšie cesty analýzy a ako efektívne eliminuje nerealizovateľné cesty pomocou stupňovitého riešenia [5].

Analyzátor tiež využíva množstvo heuristiky na automatické potlačenie správ, ktoré pravdepodobne budú falošne pozitívne.

Keď sa nájde chyba, príslušná cesta a súbor obmedzení sú užitočné na pochopenie problému. Je však nepraktické predložiť všetky tieto informácie používateľovi. Ukážeme, ako analyzátor usiluje o to, aby používateľovi poskytol stručné a napriek tomu nápadné hlásenie o chybe, ktoré minimalizuje čas na odstránenie uvedenej chyby.

CODECHECKER

Definujeme škálovateľnosť statickej analýzy nielen z hľadiska efektívneho využívania výpočtových zdrojov, ale aj z hľadiska efektívneho využívania ľudských zdrojov, napr. doba vývoja. CodeChecker je nástroj určený na uľahčenie integrácie analyzátora a iných podobných nástrojov statickej analýzy do konštrukčných systémov a kontinuálnych integračných slučiek. Je tiež plnohodnotným systémom na správu chýb.

Vzhľadom na konečný rozpočet vývojárov a tisíce správ o veľkom softvéri je dôležité najprv vyhodnotiť správy s najvyššou návratnosťou investícií.

CodeChecker tiež podporuje diferenciálnu analýzu, ktorá zabraňuje vývojárom zavádzať nové chyby do systému bez toho, aby museli predtým opraviť všetky staršie (dlhšie známe) chyby.

ZHRNUTIE

V tomto dokumente sumarizujeme naše skúsenosti zozbierané pri súčasnom prispievaní k najmodernejším produktom Clang Static Analyzer a CodeChecker. Našou nádejou je, že sa ukáže byť užitočným zdrojom pre každého, kto sa rozhodne pracovať na nástrojoch statickej analýzy.

Literatúra

[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.

PROGRAMOVANIE V OBLASTI SPRÁVY A ORCHESTRÁCIE VIRTUALIZOVA-NÝCH SIEŤOVÝCH ZDROJOV

Virtualizácia sieťových funkcií (NFV) je nová paradigma pre zmenu spôsobu budovania a prevádzky sietí. Oddelenie implementácie softvéru od sieťových zdrojov prostredníctvom virtualizačnej vrstvy predstavuje potrebu vývoja súborov funkcií riadenia a orchestrácie NFV (MANO). Zameriavame sa na koordináciu riadiacich funkcií implementovaných v rámci rôznych funkčných blokov, aby sme dosiahli spoľahlivú prevádzku funkcií MANO pracujúcich v distribuovaných prostrediach. Výzvy sú ilustrované praktickým príkladom virtuálnej technológie Open Stack a problémami inšpirovanými telekomunikačným priemyslom.

UVOD

Účelom týchto prednáškových poznámok je predstaviť nové koncepty, ako napríklad virtualizáciu sieťových funkcií (NFV), ktoré sú v súčasnosti implementované v rámci komplexných softvérových systémov a sietí, vysvetľovať nové výzvy a ukázať študentom, ako s nimi zaobchádzať pomocou programovacích techník na koordináciu riadiacich a orchestrálnych funkcií virtualizovaných sieťových zdrojov pracujúcich v distribuovaných prostrediach.

ZLOŽITÉ SYSTÉMY

Zameranie týchto prednášok je na oblasť teórie zložitých systémov, najmä komplexných softvérových systémov a komplexných sietí. Preto prednášky začínajú jemným úvodom do teórie, starostlivo sa kladú do úvahy uvažované problémy a výzvy v rámci súčasného vývoja sietí a softvérových systémov. Virtualizácia je paradigma často používaná pri riadení komplexných softvérových systémov. Predstavuje zavedenie novej abstraktnej vrstvy, virtuálnej verzie systémovej vrstvy a jej funkcií, ktorá zabraňuje zavedeniu závislosti medzi vrstvami systému.

FUNKCIE MANAŽMENTU A ORCHESTRÁCIE

V telekomunikačných sieťach sa zavádza nová paradigma nazývaná Virtualizácia sieťových funkcií (NFV), ktorá oddeľuje funkciu siete od fyzických sieťových zdrojov cez novú virtualizačnú vrstvu [2]. To však prináša potrebu vývoja súborov riadiacich a orchestračných funkcií NFV (MANO). Na tento účel je v rámci Európskeho inštitútu pre telekomunikačné normy (ETSI) definovaná špeciálna pracovná skupina. Riadenie virtualizácie sieťových funkcií a architektonický rámec orchestrácie sú definované v [1]. V týchto prednáškach sa zameriavame na funkcie riadenia a orchestrácie implementované v rôznych funkčných blokoch s cieľom dosiahnuť spoľahlivú prevádzku pre riadiace a orchestrálne funkcie pracujúce v distribuovanom prostredí.

PRÍKLADY

Tieto poznámky poskytujú úvod do témy s cieľom vysvetliť problémy a princípy, metódy a techniky používané pri ich riešení. Vypracované príklady a cvičenia slúžia študentom ako učebný materiál, z ktorého sa môžu naučiť používať funkčné programovanie na efektívnu a účinnú koordináciu riadiacich a orchestračných funkcií v distribuovaných komplexných systémoch pomocou NFV.

Metódy a techniky vysvetlené v týchto poznámkach prednášok a aplikované na problémy riadenia a orchestracie sieťovej virtualizácie už existujú a v tomto zmysle nepožadujeme originálnosť. Účelom týchto poznámok je slúžiť ako učebný materiál pre tieto metódy.

Problémy a výzvy súvisiace s koordináciou riadiacich a orchestrálnych funkcií sú riešené pomocou platformy OpenStack [3]. Je to open source cloud operačný systém, ktorý integruje kolekciu softvérových modulov, ktoré sú potrebné na poskytovanie vrstveného modelu cloud computingu. Takáto technológia je potrebná pri riešení problémov vyplývajúcich z virtualizačnej paradigmy v súčasných sieťach a študenti, ktorí chápu riešenia v OpenStack, budú môcť preniesť svoje poznatky do iných existujúcich technológií s rovnakým alebo podobným zameraním.

Problémy vyplývajúce z nových paradigiem siete, ako aj ich riešenia, sú ilustrované praktickými príkladmi využívajúcimi virtuálnu technológiu OpenStack a inšpirovanými problémami z telekomunikačného priemyslu. Všetky príklady a cvičenia sú spracované vo virtuálnej technológii OpenStack.

Literatúra

[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

CLOUD COMPUTING A FUNKCIONÁLNE PROGRAMOVANIE VO VZDELÁVANÍ

Cloud computing sa stala kľúčovou technológiou, a preto je súčasťou mnohých učebných osnov informatiky. V koncepcii cloudových aplikácií sú kľúčové tzv. mikroslužby. Funkčný rozklad typický mikroslužbám by mohol byť dobre obsluhovaný platformami bez servera, ako je AWS Lambda.

MIKROSLUŽBY

Stále viac a viac odborníkov odporúča pri navrhovaní aplikácií cloud [1,2] prijať architektúru s mikroslužbami. Neurčito definovaný architektonický štýl mikroslužby predstavuje spoločný súbor charakteristík: automatizované nasadenie, inteligentné koncové body, decentralizované riadenie údajov [2]. Nedávne úsilie o migráciu tradičných aplikácií cloud do architektúry mikrosllužieb varuje pred zvýšenou zložitosťou pri správe mnohých, hoci malých komponovacích služieb [3]. Tu je dôležité rozlišovať: zameriavame sa na orchestráciu ako obchodné logické zloženie jemnozrnných cloudových služieb. Či je ich nasadenie riadené (ústredná zložka, ktorá riadi vykonávanie) alebo choreografia (každá služba funguje nezávisle), určuje, či požadovaná aplikácia potrebuje synchrónne riadenie alebo toleruje asynchrónne riadenie. Avšak vzhľadom na to, že agilnosť a flexibilita sú vysoko požadované atribúty, je preferované choreografické rozmiestnenie [1].

Aj nefunkčné aspekty mikroslužieb, ako napríklad výkonnosť počas spustenia, zohrávajú dôležitú úlohu pri zavádzaní aplikácií.

Preto plánujeme študentov oboznámiť s konceptom profilovania výkonov v cloude v kontexte AWS Lambda. Okrem toho doplníme funkčnú paradigmu AWS Lambda pomocou rámca Haskell na kontrolu experimentov.

CLOUD COMPUTING ORIENTOVANÝ NA POUŽÍVATEĽA

V oblasti cloud computingu je dôležitým aspektom pomáhať používateľom pri ich rozhodovaní. Takéto rozhodnutia sa týkajú nasledujúcich otázok:

A. Ako sa aplikácia správa na virtualizovaných zdrojoch?

B. Koľko virtuálnych zdrojov a akého typu, od ktorého poskytovateľa cloud by sa malo získať na nasadenie aplikácie?

C. Ako dlho? Koľko to bude stáť?

Tieto otázky sú zvyčajne modelované ako problém plánovania, vychádzajúc z predpokladu, že žiadna vlastnosť systému vopred nie je známa. Jednoduchý súbor požiadaviek je poskladaný tak, aby bola aplikácia úspešne nasadená a náklady minimalizované.

Architektúra plánovača pre aplikáciu pre cloud

Plánovač BaTS [4] bol vyvinutý, aby pomohol používateľom pri nasadzovaní ich aplikácií do cloudu. Na dosiahnutie tohto cieľa je potrebné vlastné plánovanie a pravidelne kontrolovať pokrok v nasadení. V prvej fáze BaTS zhromažďuje štatistiky odberu vzorky s náhradou. Tu je potrebná iba malá vzorka (30-50 úloh) na výpočet priemernej a štandardnej odchýlky úloh pri rôznych ponukách cloudu. Modul rozpočtového odhadu potom vykoná lineárnu regresiu na optimalizáciu tejto fázy.

Použitie metodiky BaTS pre ľahkú virtualizáciu

Predstavujeme študentom problém pomoci majiteľom aplikácií, ktorí chcú vybrať tie najlepšie možnosti z hľadiska ľahkej virtualizácie pri nasadzovaní ich aplikácie ako množiny mikroslužieb.

AWS Lambda

AWS Lambda je veľmi ľahký virtualizovaný počítačový zdroj ponúkaný spoločnosťou Amazon. Granularita funkčnosti a odporúčaná doba spustenia na vyvolanie funkcie je nanajvýš záležitosťou niekoľkých sekúnd. Predpokladá sa aj neblokujúce správanie. Existuje 46 typov AWS Lambda s cenovým modelom eura na GB * s.

AWS API: Haskell implementácia

Na základe komplexnej implementácie aplikácie AWS API v jazyku Haskell [5] sa snažíme replikovať metodiku odhadu BaTS na ľahké virtualizované zdroje.

Praktická práca

Učíme študentov, aby sprostredkovali výkonnosť faktoriálneho chodu na rôznych typoch Lambdy pomocou odberu vzoriek a lineárnej regresie na výpočet priepustnosti a krivky efektívnosti cien. Napríklad by mali zvážiť, ktorý typ má najlepšiu priepustnosť za najlacnejšiu cenu. Ďalej by mali určiť, ako efektívne nájsť najvýhodnejšiu kombináciu: najnižšie náklady na dosiahnutie najlepšej výkonnosti.

Porovnávanie AWS Lambda

Funkcie AWS Lambda pracujú v kontajnerovom virtualizovanom prostredí. Je známe, že množstvo výpočtových prostriedkov priradených k funkciám je úmerné užívateľskej pamäti DRAM. Ak chcete zistiť, ako môžu funkcie Lambda spracovávať rôzne pracovné zaťaženia, môžeme nezávisle porovnávať každý z ich výpočtových zdrojov: CPU, šírku pásma pamäte, šírku pásma V/V a šírku pásma siete a latenciu. Také mikrobenchmarky porovnávania môžu byť vykonávané spustením funkcií známych výpočtov, pamäťových, V/V- a sieťovo náročných pracovných zaťažení a operácií, ako je výpočet prvých N prvočísel, spúšťanie benchmarku streamu, spúšťanie benchmarku iozone, alebo čítanie alebo zapisovanie údajov do AWS S3.

Prevod zavedených metodík je kľúčom k vzdelávaniu. Ako budúcu prácu by sme chceli podporiť funkcie Haskell AWS Lambda nasadené prostredníctvom Haskell implementácie API AWS.

Literatúra

[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.

MODERNÉ TYPOVO BEZPEČNÉ VLOŽENIE ATRIBÚTOVÝCH GRAMATÍK

Atribútové gramatiky sú silný, deklaratívny formalizmus na implementáciu a overovanie programov, ktoré sú modulárne skonštruované. Napriek tomu, že kompilátor celej atribútovej gramatiky môže byť prispôsobený konkrétnym potrebám, jeho implementácia je netriviálna a jeho dlhodobá údržba je vážny problém. V skutočnosti si vyžaduje údržba tradičného atribútovo gramatického systému také veľké úsilie, že väčšina systémov, ktoré boli navrhnuté v minulosti, už nie je aktívna. Náš prístup k implementácii atribútových gramatík je taký, že ich navrhneme ako prvotriedne funkcie moderného funkionálneho programovacieho jazyka. Vylepšíme predchádzajúcu verziu vkladania atribútových gramatík založenú na zipsovaní, čím sa stáva neinterrujúcim (tj nie sú potrebné žiadne zmeny v používateľom definovaných údajových typoch) a typovo bezpečným. Okrem toho dosahujeme jasnejšiu syntax pomocou moderných rozšírení jazyka Haskell. Veríme, že naše zavádzanie môže byť v praxi využívané na realizáciu elegantných, efektívnych a modulárnych riešení reálnych problémov programovania.

ÚVOD

Atribútové gramatiky (AG) sú deklaratívnym formalizmom, ktorý navrhol Knuth [7] koncom 60. rokov a umožňuje modulárne a pohodlné implementovanie a odôvodnenie vlastností programov.

Konkrétna AG sa opiera o bezkontextovú gramatiku, ktorá definuje syntax jazyka a atribúty spojené s produkciou gramatiky, aby definovala sémantiku daného jazyka. AG sa v praxi používali na špecifikovanie reálnych programovacích jazykov, ako napríklad Haskell [2], ako aj efektívne algoritmy formátovanej tlače [16], techniky deforestácie [4] a výkonné typové systémy [11]. Pri programovaní s AG je modularita dosiahnutá vďaka možnosti definovať a používať rôzne aspekty výpočtov formou samostatných atribútov.

Atribúty sú jedinečné výpočtové jednotky, zvyčajne pomerne jednoduché a modulárne, ktoré je možné kombinovať do prepracovaných riešení komplexných programovacích problémov. Môžu sa tiež analyzovať, ladiť a udržiavať nezávisle, čo uľahčuje vývoj a evolúciu programu.

AG sa osvedčili ako obzvlášť užitočné na špecifikovanie výpočtov nad údajovou štruktúrou strom: pri jednom strome je špecifikovaných niekoľko systémov AG, ako napr. [14,3,8,17], tie si samé určujú, ktoré hodnoty alebo atribúty je potrebné vypočítať a vykonať počas výpočtov. Úsilie vynaložené počas návrhu, tvorby, zdokonaľovaní a údržbe týchto AG systémov je však obrovské, čo je často prekážkou pri dosahovaní úspechu aký si zaslúžia.

Stále populárnejší alternatívny prístup k využívaniu AG spočíva v tom, že sa vkladajú ako prvotriedne funkcie do univerzálnych programovacích jazykov [12, 9, 13, 15, 18, 1]. Tým sa zabráni záťaži pri zavádzaní úplne nového jazyka a pridruženého systému tým, že ho budete hosťovať v najnovších programovacích jazykoch. Na základe tohto prístupu sa potom využívajú moderné konštrukcie a infraštruktúra, ktoré už tieto jazyky poskytujú, a sústreďuje sa na osobitosti daného doménovo špecifického jazyka.

Funkcionálne zipsovanie (FZ) [6] je výkonná abstrakcia, ktorá značne zjednodušuje implementáciu traverzálnych algoritmov, ktoré vykonávajú množstvo miestnych aktualizácií. FZ boli úspešne použité na zostrojenie AG vložky do Haskell [9,10]. Napriek svojej elegancii malo toto riešenie veľkú nevýhodu, ktorá zabraňovala jeho použitiu v reálnych aplikáciách: atribúty neboli ukladané vo vyrovnávacej pamäti, ale skôr opakovane prepočítané, čo vážne znížilo výkon. Nedávno bola táto chyba odstránená [5] ale zároveň nahradená inou: prístup sa dostal do rušivého stavu, t. j. na to, aby bolo prospešné vkladanie, musia byť doplnené užívateľom definované dátové štruktúry.

V tomto príspevku prezentujeme alternatívny mechanizmus na ukladanie atribútov do vyrovnávacej pamäte na základe samoorganizujúcej nekonečnej siete. Tento graf je umiestnený na vrchole užívateľsky definovaného algebraického dátového typu (ADT) a zrkadlí jeho štruktúru. Použitý údajový typ zostáva nedotknutý. Vloženie je založené na dvoch (skôr ako jednom) koherentných zipsoch, ktoré prechádzajú paralelne dátovými štruktúrami. Navyše je naše riešenie úplne bezpečné. Moderné rozšírenia Haskell, ako je napr. ConstraintKinds, umožňujú šíriť obmedzenia v ADT, ktoré úplne eliminujú zaťaženie typu v čase behu ako to bolo v predchádzajúcich verziách.

Ďalšou vedľajšou výhodou používania moderných funkcií Haskell je čistejšia syntax s menším počtom riadkov kódu generovaných pomocou šablón v Haskelli.

Literatúra

[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)

AKO ZELENÝ JE VÁŠ PROCES?

Podobne ako v prípade Bio produktov sa svet rozvíja aby sa stal ekosystémom, ktorý je viac povedomý prírodou. Zelená iniciatíva definuje dva hlavné ciele: zníženie spotreby energie a využívanie základných prírodných zdrojov pri výrobe elektrickej energie.

Jednou z výziev pre výrobcov batérií je ako dlho môže batéria fungovať bez toho, aby bola znovu nabitá. Existuje mnoho ďalších problémov, ako napríklad veľkosť, ktorá výrazne ovplyvňuje tvar a hmotnosť zariadenia. Batéria je považovaná za trochu ľahšiu v porovnaní so zariadením, ktoré potrebuje batériu na prevádzku. Výzvou je ako urobiť jeho veľkosť menšiu a ľahšiu a určite vysokú efektívnosť, pokiaľ ide o prevádzkovú dobu mobilného zariadenia bez toho, aby bola nabitá.

Na vrchole tejto hardvérovej výzvy existuje softvérová výzva: samotný softvér by mal podporovať úspory energie. Robiť to bez obmedzenia užívateľského zážitku sa v dnešnej dobe považuje za tichý resp. zamlčaný, ale dôležitý cieľ každého vývoja softvéru, ktorý je zameraný na akýkoľvek druh prenosných zariadení [1]. Tento cieľ sa zvyčajne objavuje, keď sa súvisiace požiadavky zmenia z nezvyčajného na dôležitý problém.

Pri zohľadnení, že spotreba energie každého mobilného zariadenia je ovplyvnená bežiacimi aplikáciami prostredníctvom frekvencie používania služieb až po skutočnú náladu používateľa, bude vývoj softvéru pre takéto zariadenia už poriadnou výzvou [4]. Dalo by sa povedať, že výzva vývoja softvéru je vždy rovnaká, ale musíme zdôrazniť "mobilitu" ako kľúčovú vlastnosť systému. Stav napájania/nabitia batérie tiež určuje výkon systému kvôli nastaveniam úrovne úspory z operačného systému - dobre známe ako "predvoľby úspory energie" na mobilných zariadeniach.

Je ešte ťažšie, ak jeden (v našom prípade učiteľ) musí pripraviť študentov na takéto výzvy [6]. Všetky už známe "najlepšie postupy" a "tipy na úsporu energie" musia byť prezentované v kontexte, ktorý je pre študentov ľahko zrozumiteľný.

To sa dá dosiahnuť umiestnením konceptov do známeho prostredia, ako je testovanie softvéru a automatizácia testov [2]. To je práve to, na čo sa zameriavame s týmto tutoriálom. Odzrkadluje to obsah nadchádzajúcich sekcií, počínajúc návrhom, po ktorom nasledujú príklady a uzatváraním ďalších tipov na zlepšenie.

Hlavné zameranie je na spotrebu energie aplikačného softvéru a jeho vývojových procesov, kde každá fáza vývoja zohráva významnú úlohu.

Vzhľadom na akýkoľvek proces vývoja softvéru sa energia spotrebováva pri analýze problému, pri vytváraní a vyhodnocovaní kódu [3]. Softvérové alebo hardvérové nástroje sa musia použiť na monitorovanie spotreby energie pre softvér spustený na vrchole vybraných operačných systémov a na vyhodnotenie spotreby energie. Zvyčajné scenáre používania sú na monitorovanie spotreby energie vybraného softvéru [5], ale tiež sa pozrieme na možnosť použitia týchto nástrojov na meranie toho, ako zelený je proces, ktorý produkuje konečnú verziu vytváraného softvéru.

Príklady pokrývajú rôzne situácie. Počnúc prípadom energetického profilovania pracovného softvéru tretích strán v konkrétnych scenároch použitia poukazujeme na kľúčové vlastnosti (výhody a nevýhody) existujúcich nástrojov. Počas testovania kódu vyvinutého softvéru prezentujeme typické využitie softvéru na tvorbu profilov spotreby energie.

V poslednom príklade, ktorý prezentujeme, je uvedený prístup merania od profilovania útržku kódu alebo jedinej aplikácie po analýzu spotreby energie nástrojových reťazcov. Tento článok predstavuje všeobecný prístup k profilovaniu energie a jeho cieľom je nahradiť otáznik v názve tutoriálu bodkou predstavujúcou hodnotenie výsledkov pre každý špecifický prípad tutoriálu.

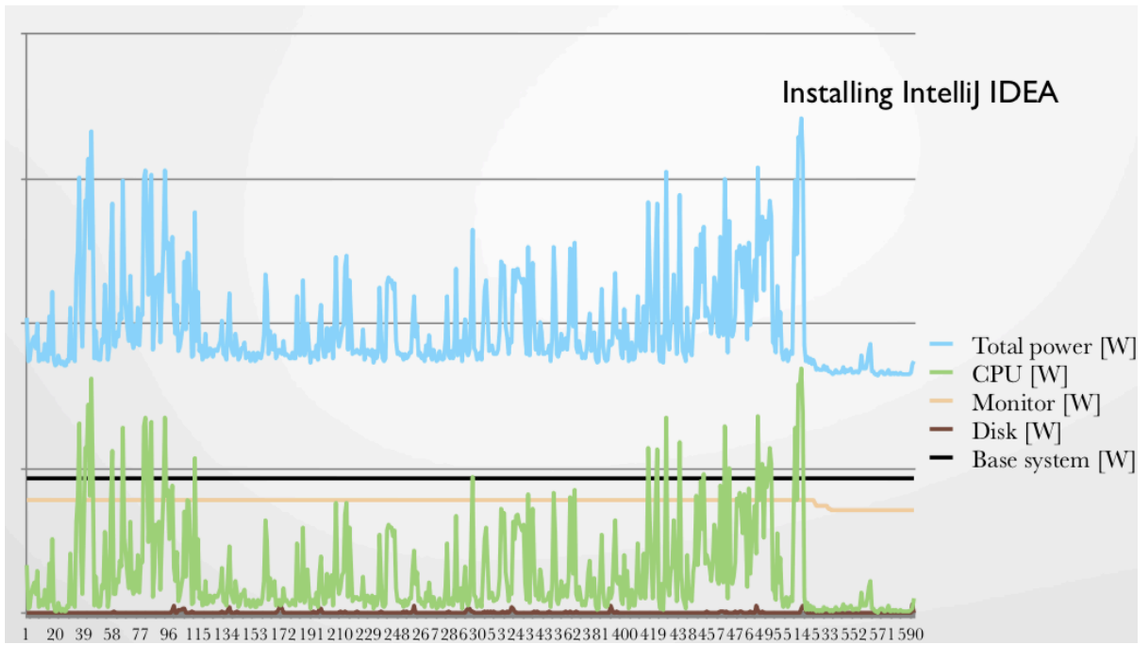
Literatúra

[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.

FUNKCIONÁLNE PROGRAMOVANIE ZARIADENÍ

POZNÁMKA

Tento príspevok je rozšírenou verziou nášho príspevok RWDSL18 [2]. V tomto článku budeme používať ten istý DSL, s niektorými drobnými rozšíreniami a vylepšeniami. Aktuálne sa zameriavame na to ako môže byť program mTask DSL použitý na programovanie zariadení internetu vecí a diskutuje o simulátore pre programy mTask ako programov iTask na vysokej úrovni abstrakcie.

ÚVOD

Mnoho zariadení je dnes vybavených jednoduchým mikroprocesorom na ovládanie ich správania. Typickými príkladmi sú termostaty, žiarovky, elektrické zásuvky, požiarne poplachy, otvárače dverí atď. Keď tieto zariadenia môžu komunikovať medzi sebou alebo s niektorým vzdialeným počítačom, hovoríme, že sú súčasťou internetu vecí, IoT. Mikropočítače v týchto zariadeniach sú cenovo veľmi dostupné a stávajú sa všadeprítomnými zariadeniami. Drahé zariadenia, ako sú autá a prístroje s veľmi zložitou úlohou, sú vybavené plnohodnotným vstavaným počítačom a vhodným softvérom. Pre väčšinu malých a relatívne lacných zariadení IoT je takýto vstavaný počítač príliš drahý alebo spotrebováva príliš veľa energie; v ich prípade sa na spúšťanie softvéru používa jednoduchý a lacný mikroprocesor. Tieto systémy majú veľmi obmedzený výpočtový výkon a limitovanú pamäť, zvyčajne 30 KB až 4 MB flash pamäte na ukladanie programu. Životnosť tejto pamäte je obmedzená na 1000 cyklov zápisu. Na ukladanie premenných, haldy a programového zásobníka majú tieto systémy pamäte RAM veľkosti 2 až 40 KB.

Obmedzenia rýchlosti procesora a pamäte vylučujú použitie operačného systému. Prístroj priamo vykonáva program riadiaci zariadenie. Dokonca aj kontrolné programy na týchto zariadeniach IoT pozostávajú z niekoľkých úloh. Napríklad na kontrolu stavu tlačidla desaťkrát za sekundu, aktualizácia displeja každú sekundu, meranie teploty dvakrát za minútu a zapnutie vykurovania po najmenej piatich minútach, pokiaľ nie je tlačidlo stlačené skôr. V dôsledku rozdielnych časových rámcov a závislostí týchto úloh má kontrolný program tendenciu byť trochu chaotický, a to nezávisle od použitého programovacieho jazyka. Zariadenia IoT navyše vykonávajú oddelené programy pre zvyšok aplikácie v rámci internetu a komunikujú pomocou množstva protokolov. To robí vývoj a údržbu aplikácií internetu vecí komplexnou a náchylnou k chybám.

Programovanie orientované na úlohy (TOP) ponúka malé podprocesy, ktoré sa dajú jednoducho skladať na vytváranie a riešenie zložitejšie úlohy. Úlohy sa hodnotia krok za krokom a po každom kroku si môžu vypýtať aktuálnu hodnotu ďalších úloh. TOP sa prvýkrát implementuje v systéme iTask [4,5], ktorý je vložený do programu Clean [6]. Primárne úlohy v systéme iTask zhromažďujú vstupy prostredníctvom automaticky generovaného webového formulára alebo zhromažďovaním údajov z iných programov a dátových skladov. Výkonná sada kombinátorov sa používa na zostavovanie úloh na zložitejšie úlohy. V tomto článku uvádzame, že TOP je veľmi vhodný pre programovanie IoT zariadení. Prvotné úlohy poskytujú aktuálnu hodnotu vstupov a snímačov. Konštruktory (veľmi podobné systému iTask) sa používajú na kombinovanie úloh na zložitejšie úlohy, napr. spracovanie hodnoty snímača.

Zariadenia IoT obvykle vykonávajú voľne závislé úlohy, ktoré riadia senzory, akčné členy a komunikáciu zariadení. Programovanie v štýle TOP ponúka stručné programy. Vykonávanie týchto úloh v obmedzeniach malých mikrokontrolérov s veľmi obmedzeným výkonom a nikoľkými KB RAM pamäte si zaslúži nejakú myšlienku. Z dôvodu vážnych obmedzení použitých mikrokontrolérov nemôžeme systém iTask prenášať do zariadení IoT, pretože typický program iTask vyžaduje približne 100 MB haldy pri vykonávaní. Definujeme preto vstavaný doménový špecifický jazyk (eDSL) nazývaný mTask pre IoT zariadenia. Táto eDSL je zakotvená v systéme iTask, pretože plánujeme, aby tieto TOP jazyky boli plne interoperabilné.

Prínosy tohoto príspevku sú:

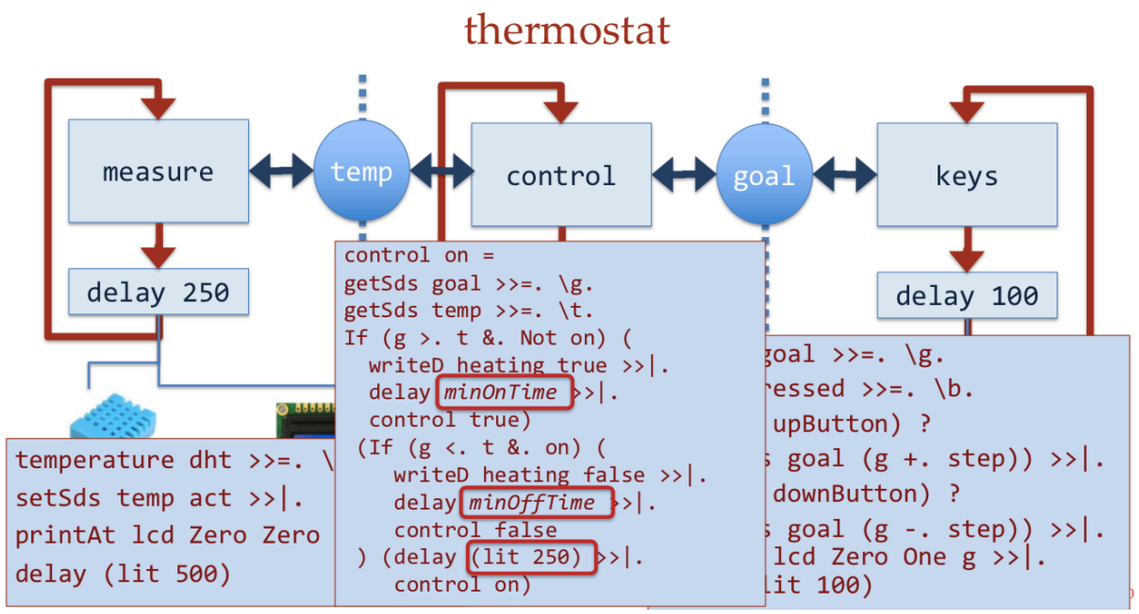
- Predstavenie funkionálneho programovacieho jazyka založeného na úlohách pre zariadenia IoT. V porovnaní s predchádzajúcimi jazykmi pre mikroprocesorové programovanie [3] je imperatívna periférna kontrola nahradená referenčnými transparentnými konštrukciami.
- Demonštrácia posture vytvorenia rozširiteľného, funkcionálneho, viacpohľadového a typo bezpečného vnoreného DSL. Je to eDSL [1].
- Generovaný kid rovnako pobeží na malých a pomalých zariadeniach ako na väčších počítačoch a v simulátore.
- Použitie Arduino C++ v roli medzijazyka prekladu je kód písaný v tomto eDSL spustiteľný na veľkom počte mikrokontrolérov.
- Simulácia mTask programu v iTask programe ponúka možnosť sledovania aj interných záležitostí mTask programu a manipulácie simulačného prostredia pre účely experimentovania so správaním eDSL programu. V takýchto simulátoroch je jednoduché manipulovať s časom a senzormi, čo by neplatilo pre reálne nasadenia, napr. by sme mohli nastaviť teplotu hlásenú senzorom namiesto reálnej potreby chladenia resp. ohrievania zariadenia IoT.

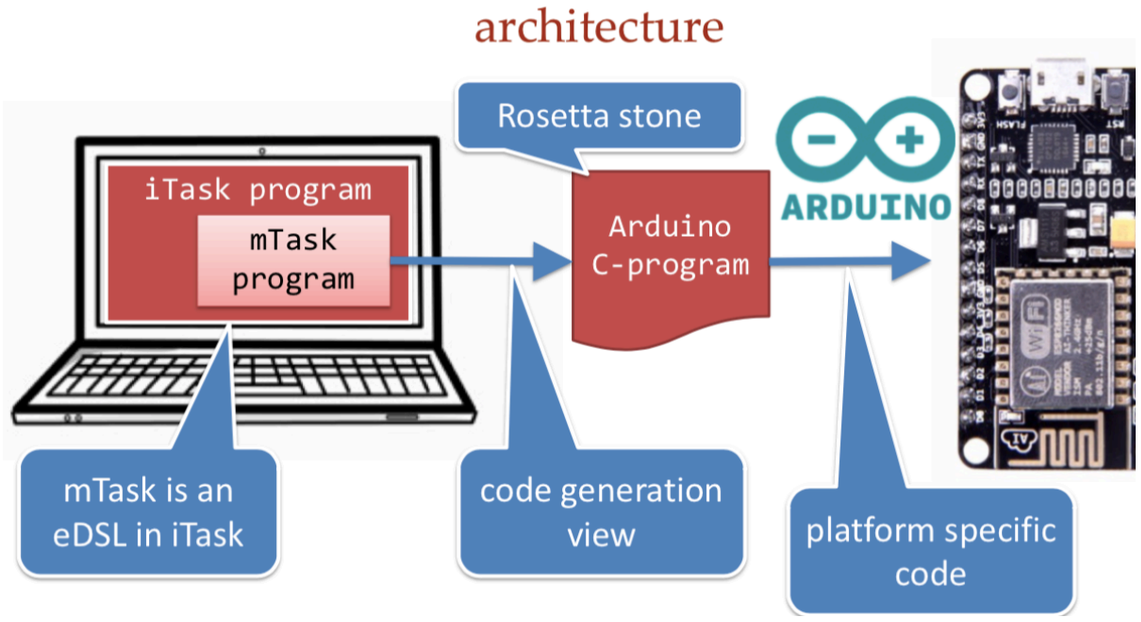
Literatúra

[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

POCHOPENIE KÓDU NÁSTROJOM CODECOMPASS

PREHĽAD

Vývoj a údržba sú dve samostatné etapy s rôznymi charakteristikami, preto si vyžadujú aj odlišnú podporu nástrojov. Počas vývoja sme predovšetkým písali nový kód, ktorý vyžaduje podporu nástrojov ako je dokončenie kódu, zosúladenie atď. A zvyčajne spracujeme len niekoľko súborov, ktoré sa viac alebo menej podieľajú na rovnakej úrovni abstrakcie. Počas údržby čítame a prechádzame existujúcou kódovou základňou medzi veľkým počtom modulov a súborov na rôznych úrovniach abstrakcie [1]. Vo vývoji sú zámery jasné, na rozdiel od porozumenia kódu, kde je úlohou obnoviť pôvodný účel niektorých fragmentov kódu.

V priemyselnom prostredí [2] môže projekt pozostávať z miliónov riadkov kódu. V prípade dlho existujúcich veľkých systémov, kde bola základňa kódov vyvinutá a udržiavaná už desiatky rokov kolísajúcimi tímami, sa pôvodné zámery stratia, dokumentácia je nedôveryhodná alebo chýba. Preto je jedinou spoľahlivou informáciou samotný kód. Chápanie takýchto veľkých softvérových systémov je nevyhnutnou, ale zvyčajne veľmi náročnou úlohou. To znamená, že je potrebná podpora nástrojov [3].

Keď sa oboznamujeme s neznámym zdrojovým kódom, prvým krokom je nájsť príslušné časti systému. Tento proces si vyžaduje rýchlu lokalizáciu funkcií založenú na niektorých menovaných entitách získaných zo záznamových správ (logov) alebo iných zdrojov. Ďalším krokom je rozšírenie vedomostí o systéme prostredníctvom diagramov, funkčných reťazcov atď. A nakoniec by sme chceli overiť získané poznatky prostredníctvom správ zo systémov správy verzií, architektonických informácií a odkazov na súvisiace moduly.

CODECOMPASS

CodeCompass [4, 5] je rámec pre pochopenie otvoreného zdrojového kódu. Poskytuje doplnkovú architektúru, ktorá umožňuje pridanie rôznych nástrojov analyzátora, ktoré produkujú rôzne vizualizácie, zberače informácií, metriky [6] atď. Najdôležitejším cieľom projektu bolo rozširovať CodeCompass pre rozsiahle priemyselné projekty.

V prvom kroku musí byť produkt analyzovaný: všetky informácie sú zhromaždené a uložené do databázy, ktorá potom umožňuje vrstve služby poskytovať požadované vizualizácie. Pre rýchle vyhľadávanie CodeCompass používa indexovanie textu, ktoré vedie navigáciu nezávisle od jazyka zdrojového kódu. Keďže hlavným cieľom je poskytnúť presné informácie o jazykových prvkoch, identifikácia symbolov podľa ich názvu nie je dostatočná. Využívame infraštruktúru kompilátorov LLVM na presnú identifikáciu symbolov a na vyriešenie pomenovaných entít pomocou stromu abstraktnej syntaxe.

CodeCompass je rozšírený jazykovými analyzátormi. Najčastejšie podporované jazyky sú C / C ++, ale aj Java a Python sú čiastočne spracované. Okrem uvedených symbolov sa v databáze ukladajú aj ďalšie informácie, ako sú vzťahy medzi uzlami AST (funkcie, dedičstvom) a súbory (vzťah poskytovateľa rozhrania, zaradenie atď.). Používajú sa na zobrazenie architektonického pohľadu o systéme na základe používania symbolov [7].

Základňa kódov nie je jediným zdrojom dokumentácie. Spätné hlásenia systému riadenia verzií obsahujú aj informácie, ktoré sú dôležité pre pochopenie toho, prečo sa na danom module uskutočnili určité zmeny. CodeCompass tiež číta úložisko Git, ak nejaké existuje. CodeCompass je vybavený aj pokročilými funkciami. Môže zobrazovať funkcie generované kompilátorom, ktoré chýbajú zo zdroja. Analýza smerníkov pomáha pochopiť, ktoré premenné sa týkajú toho istého objektu. Môžeme kontrolovať funkčné relačné vzťahy aj vtedy, ak sú vyvolané prostredníctvom virtuálnej funkcie alebo smerníka na funkciu.

ZHRNUTIE

Porozumenie kódu vyžaduje špecifickú podporu nástroja na pochopenie rozsiahleho softvéru. Prehľadáme a kategorizujeme nástroje na pochopenie kódu podľa architektúry a funkcií s cieľom preskúmať ich schopnosti.

Zavádzame CodeCompass, ktorý predstavuje širokú škálu funkcií týkajúcich sa vizualizácií, poskytovania informácií, kontroly verzií a zhromažďovania dokumentácie, metrík atď.

Literatúra

[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.

SKELETY VO FUNKCIONÁLNOM PROGRAMOVANÍ PRE VYSOKOVÝKONNÉ VÝPOČTY

Funkcionálne programovacie jazyky poskytujú nástroje a funkcie na navrhovanie a implementáciu distribuovaných aplikácií. Keďže funkcionálne programy majú prirodzené paralelné vlastnosti, je možné ich využiť na získanie spoľahlivého súbežného spracovania prostredníctvom sinej distribúcie a koordinácie na vysokej úrovni.

Najmodernejší vývoj paralelného softvéru umožňuje rozsiahle využitie rôznych metodológií a prístupov na dosiahnutie želanej vysokej rýchlosti. Súbežnosť však zostáva jednou z najťažších oblastí, najmä v prípade prístupov funkcionálneho programovania.

Hlavným cieľom je preskúmať paralelné výpočtové skelety v novom prostredí, ilustrovať vhodnosť a použiteľnosť FP v nových distribuovaných výpočtových nastaveniach. Súbor známych paralelných algoritmických skeletov sa testuje v komponentoch pre vysokovýkonné výpočty (high performance computing, HPC). Značný počet príkladov poskytuje vysoké zrýchlenie. Množstvo rovnobežných výpočtov vždy závisí od mnohých faktorov, napríklad: použitý vzorec výpočtu, vylepšená granularita, sémantika distribuovaných uzlov, či forma zdielania údajov. Príkladmi skúmame dobre modelovanú koordináciu a sémantickú spoľahlivosť.

APLIKAČNÉ DOMÉNY SKELETOV

Koordinácia

Špecifické oblasti distribuovaného a paralelného funkcionálneho výpočtu vyžadujú prvky koordinačného jazyka. Výskumné otázky, ktoré sa predtým riešili, sa týkali toho, ako možno dosiahnuť paralelné správanie a komunikáciu funkcionálnych programov na vyšších úrovniach. Zavedené funkčné prvky programovacieho jazyka s vyššou úrovňou abstrakcie sa ukázali ako uskutočniteľné pre paralelné, vysokoúrovňové a dátovo náročné výpočty. [2].

Uvedené zámery viedli k zvýšenej vyjadrovacej sile jazykových prvkov pre paralelizmus vo funkcionálnych programoch. Presnejšie povedané, bolo navrhnuté jazykové rozšírenie DClean pre distribuované programovanie a koordináciu funkcionálnych programov Clean. Rozšírenie pozostáva z jazykových prvkov vysokej úrovne koordinujúcich čisto funkcionálne výpočtové uzly v navrhnutom distribuovanom prostredí a klastroch.

Konštrukty generujú výpočtové boxy spojené prostredníctvom vyrovnávacích komunikačných kanálov. Pri použití DClean programátor iba naznačuje ako je distribuovaný výpočtový model usporiadaný do vygenerovaného distribuovaného grafu a ako sa riadia toky údajov procesnej siete pomocou zadaných kanálov. Jazyk ponúka výhodu písania distribuovaných a funkcionálnych aplikácií bez toho, aby bolo nutné sa oboznámiťä s podrobnosťami o technických aspektoch viacvrstvového prostredia a middlewarových služieb. Distribúcia práce sa vykonáva podľa preddefinovanej paralelnej výpočtovej schémy, algoritmického skeletu parametrizovanej funkciami, typmi a vstupnými tokmi.

Hlavným účelom zavedenia koordinačného jazyka bolo definovať funkcionálne paralelné výpočtové kostry (skelety). Vo veľkom počte príkladov sa dosiahlo vysoké zrýchlenie paralelizmu. Skutočné množstvo paralelizmu záviselo od poradia vytvorenia kanálov, množstva práce na nich, rýchlosti načítania a uchovávania údajov a zložitosti uzlov [1].

V reálnych distribuovaných aplikáciách bola nevyhnutná grafická vizualizácia distribuovaného výpočtu pomocou spustiteľného nástroja na porozumenie sémantiky kódu [4]. Toto znázornilo očakávaný paralelizmus na blokoch a na tokoch vytvorených pomocou dobre definovaných skeletov vysokej úrovne. Všetko zamerané na modelovanie a formulovanie vlastností operačnej sémantiky jazyka DClean [4].

Počítačovo fyzikálne systémy

Prístup funkcionálneho modelovania založený na skeletoch, ktorý používa koordinačné jazyky, sa uplatňuje aj na súčasné prototypy moderných systémov CPS. Štúdium vzťahov medzi CPS a distribuovanými systémami alebo CPS a zabudovanými systémami je dôležité pre prijímanie primeraných rozhodnutí v krokoch návrhu a modelovania sofistikovaných prototypov systémov CPS.

Realizované prípadové štúdie systému CPS [5] opisujú spolupracujúce výpočtové jednotky riadiace fyzické entity (senzory) a vzťahy s inými komplexnými systémami. Systém CPS smarthouse zavádza nové aspekty, vlastnosti a prístupy vo všeobecnom prototypovaní pomocou skeletov. V takomto návrhu systému CPS sa riešia dôležité sémantické otázky z hľadiska pravdepodobnosti a správania, kde interoperabilita je špecifická hlavná črta, ktorú treba analyzovať a špecifikovať.

SKELETY VO FUNKCIONÁLNOM PROGRAMOVANÍ PRE VYSOKOVÝKONNÉ VÝPOČTY

Výskum rozširovania použiteľnosti skeletov vo vysokovýkonnom výpočtovom prostredí je kľúčovým bodom v paralelnom prístupe FP. Prispôsobenie skoršieho know-how o programovaní kostry heterogénnych viacjadrových systémov vedie k vyšším zrýchleniam, kde merania a porovnania hodnotia nové procesy paralelizácie.

Prototypy skeletov sú definované z hľadiska ich funkcií a koordinácie. Prípadové štúdie ilustrujú súvislosti s inými typmi distribuovaných systémov, ktoré sú dôležité z dôvodu ich viacvrstvovej štruktúry. Vlastnosti distribuovaného systému poskytované spôsobom opisov očakávaného správania sú testované funkcionálnymi skeletmi distribuovaného programovania zhlukov a sietí.

Literatúra

[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.