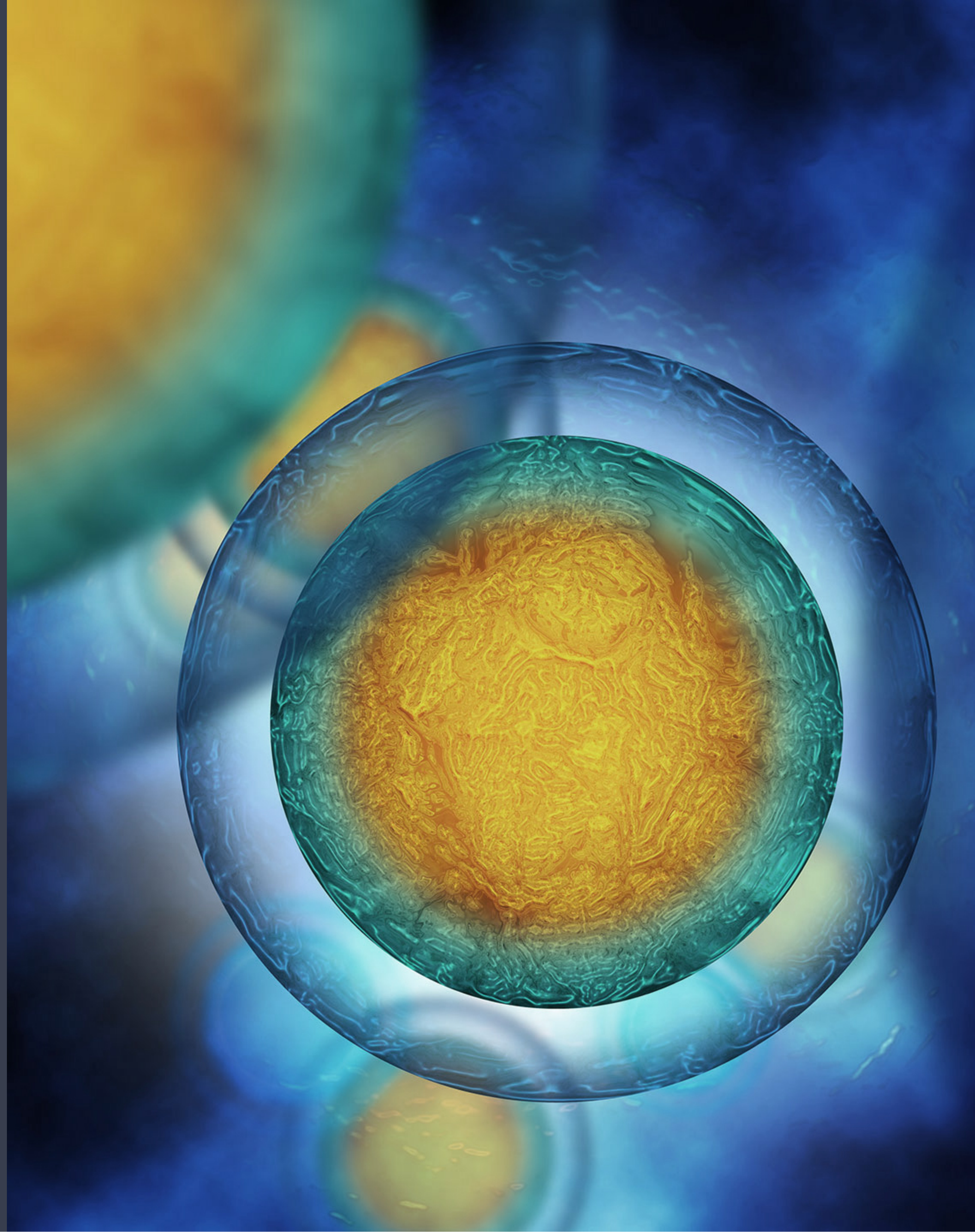
FE3CWS

# MATERIAL DIDACTIC PENTRU ŞCOALA DE IARNĂ THREE "CO"

Cod:
Output intelectual nr. 1 al proiectului Erasmus+ 2017-1-SK01-KA203-035402



# CUPRINS

**abreviat:**

- **9 teme, legate de dezvoltarea programelor complexe: compoziția sistemelor, inteligibilitatea codului, corectitudinea codurilor**
- **10 autori din 7 universități europene din Croația, Olanda, Portugalia, Slovacia și Ungaria**
- **Materialul didactic este disponibil în 7 limbi: bulgară, croată, engleză, maghiară, portugheză, română, slovacă**

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

Școala de iarnă ”The three CO” – Composability, Comprehensibility and Correctness – abreviat 3COWS – este primul program intensiv dedicat participanților din sistemul universitar. Programul extinde activitățile comunității în jurul programelor CEFP – Central European Functional Programming – axat pe aplicarea programării funcționale și a fost organizat în perioada 22-26 Ianuarie 2018. Școala este organizat prin proiectul ERASMUS+ cu codul 2017-1-SK01-KA203-035402, întitulat “Focusing Education on Composability, Comprehensibility and Correctness of Working Software”.

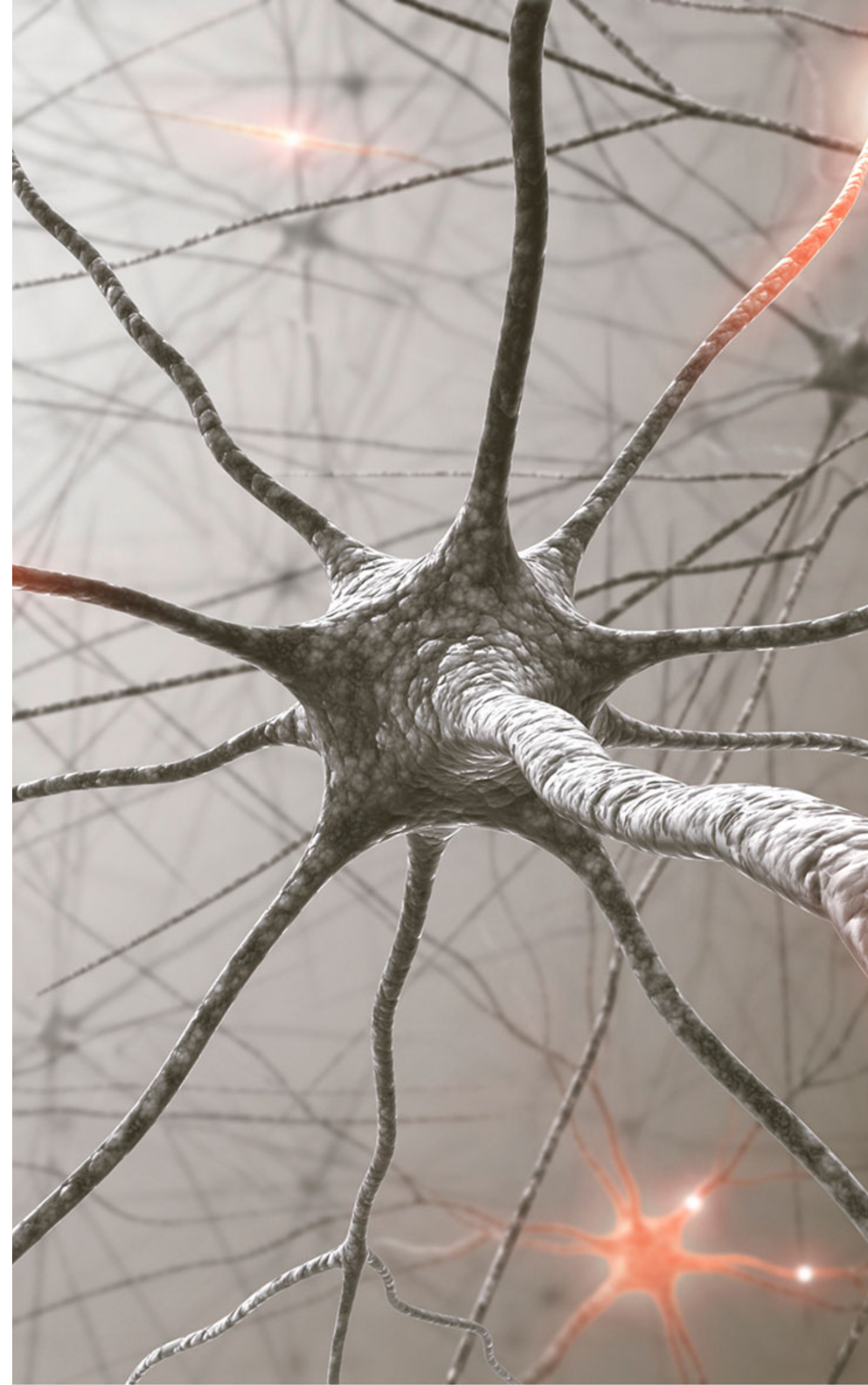
Materialul prezentat este produs în cadrul proiectului menționat mai sus. Publicația prezentă este formatul gata de tipar al output-ului intelectual ”O1”.

© European Union, 2017-2019

Conținutul prezentei publicații reflectă opiniile și ideile autorilor secțiunilor, acestea nu sunt sub nicio formă opiniile Uniunii Europene. Uniunea Europeană și persoanele menționate nu sunt responsabile pentru rezultatele folosirii informațiilor cuprinse în acest document.

# CUPRINS

1. Achiziţia algoritmică a limbajului natural
2. Analiza statică a codului cu ”CodeChecker”
3. Programare în sisteme de virtualizare a rețelelor
4. Programare funcțională și ”Cloud Computing” în educație
5. Gramatici de atribute în sistem ”type-safe”
6. Cât de economic – cât de verde – este procesul tău?
7. Programarea funcțională în IoT
8. Înțelegerea codului sursă cu CodeCompass
9. Şeme HPC în Programarea Funcțională



# ACHIZIȚIA ALGORITMICĂ A LIMBAJULUI NATURAL

Conceperea mașinilor capabile de a ”gândi” este o mare provocare. Într-o primă instanță relația dintre ”minte” – ca instrument al gândirii – și limbajul de exprimare este o problemă nu numai pentru cercetători informaticieni, dar și pentru o gamă mai largă de savanți. De exemplu,  Renfrew [1] caracterizează gândirea umană ca fiind una simbolică, cu urmare pentru el limbajul – natural – este rezultatul comunicării bidirecționale. În concepția lui Gardenfors [2] conceptele lingvistice sunt ordonate ierarhic. Limbajele se dezvoltă prin gramatică, deci și structural, deci teoriile sunt valabile pentru limbajele ”simple”. S-a verificat că string-urile genetice pentru concluzii semantice sunt metodele potrivite pentru descrierea evoluției limbajului natural. În aceeași ordine de idei am stabilit că procesul de evoluție lingvistică are loc concomitent pe mai multe nivele.

În continuare am evaluat ipoteza lui Chomsky relativă la existența unei gramatici universale [5]: am concluzionat că – în pofida faptului că axiomatizarea existenței unei gramatici universale aplicabilă pentru toate limbile, acest caracter universal nu s-ar înțelege ca o structură – gramatică în sensul clasic al cuvântului – dar ca un algoritm universal, adică o metodologie la un nivel meta-lingvistic astfel încât conceptele să reprezinte entități ”semantice”. Prin extensia ideii de limbaj universal, Edelman [6] presupune că ”algoritmul universal” al minții umane este una simbolică și că are extensie temporală. Concluzia este că mintea umană – prin perspectiva procesării informației – nu este numai o ”mașină” simbolică, dar și una dinamică temporală, care este capabilă pentru a exprima și a recunoaște atât limbajul cât și schimbările gramaticale ale acestuia. În a treia ipostază, în Shaumyans [7] este prezentat ideea unei gramatici aplicative universale, care iarăși este realizat printr-un proces dinamic, uneori chiar și aplicativ.

Teoria semiotică a limbajelor este avantajoasă prin faptul că sintaxa și semantica sunt în mod implicit legate unul de celălalt, ele nu sunt separabile. Această legătură – în mod implicit ridică necesitatea ca modificările în limbajul natural să fie reprezentate ca și categorii semantice.

Concluzia noastră este că teoriile semiotice a limbajelor naturale ne dă o oportunitate de a studia evoluția deterministă a sistemelor algoritmice de gândire – al agenților cognitivi. O consecință pozitivă a concluziei este posibilitatea unificării eforturilor pentru a realiza sisteme de comunicare bazate pe conversații între actori umani și calculatoare – Human-Computer Communication – respectiv între sisteme automatizate – Computer-Computer Communication. Ideile incipiente pentru soluționarea unificării celor două tipuri de interfețe este prezentat în [8].

Ca urmare a cercetărilor, credem că avem la dispoziție algoritmul de transformare a conceptelor lingvistice într-un limbaj propice agentului cognitiv. Avem totodată posibilitatea de a deduce concepte abstracte în limbajul agentului cognitiv. Credem că limbajul agentului este analogul limbajului ”intern” dezvoltat și folosit de mintea umană.

Nu cunoaștem nici regulile de conceptualizare, nici regulile aplicate pentru concepte în procesul de deducție a rezultatului. În schimb avem un sistem care ne dă o metodologie pentru a modela evoluția agentului ”mintal”. Acest mecanism este unul aplicativ și dinamic, care totodată prezintă un nivel înalt de calcul paralel. Presupunem că conceptele sunt reprezentate într-un mod neredundant.

Concomitent cu cele precizate mai sus, prin introducerea unui nivel mai mare de abstractizare se reduce numărul meta-operațiilor – primitivelor – și se va mări numărul de aplicații ale noțiunilor abstracte (aplicative binding – în termenii programării funcționale). Prin studiul procesului algoritmic al gândirii unui copil, am ajuns la o estimare a vitezei de transmisie a informației de 300 trilioane de semnale în fiecare secundă. Am introdus în procesul de prelucrare a informației gramatica limbii, respectiv generarea fonemelor limbii slovace, precum și translatarea cuvintelor scrise în sunete. Ilustrăm procesul de achiziție a elementelor lingvistice având diferite nivele de abstracție prin citirea unor fragmente scurte de text și folosirea dicționarelor ierarhice la mai multe nivele.

De asemenea, evaluăm procesul de achiziție a limbii la un corpus mai mare – o carte întreagă – și concludem că achiziția limbajului cu abstractizare ierarhică are ca rezultat un graf minimal – în sensul eliminării redundanțelor – cu legături (binding) paralele și seriale, fără necesitatea stocării datelor.

Am folosit contextul lingvistic gramatical pentru identificarea vizuală a obiectelor: mai întâi am dezvoltat un limbaj de descriere a obiectelor, după care am aplicat procesul de abstractizare la datele generate de descrierea obiectelor din limbaj. Ne-am axat pe descrierea obiectelor 3D prin gramatici descriptive: acest proces este numit ”simbolizare”.

Procesul de simbolizare asigură pentru fiecare obiect o descriere textuală, ca atare ne pune la dispoziție nivelul fundamental de nivel abstract de date. Prin definirea nivelului abstract de date și folosind programe funcționale, este posibilă abstractizarea și prelucrarea facilă a obiectelor.

Abordarea aplicativă se poate folosi chiar și în cazul gramaticilor independente de context. Am demonstrat – printr-un algoritm – că orice gramatică independentă de context poate fi transformată într-o formă folosind ”supercombinatori”. Rezultatul depinde însă de gramatica de intrare, deci problema originală se transformă în aceea de a găsi gramatica potrivită problemei actuale pe care vrem să-l rezolvăm.

Dezvoltăm diferite tipuri de ”supercombinatori”, care sunt rezultate ale diferitelor gramatici independente de context și le analizăm proprietățile. Estimăm și comparăm clasele de eficiență computațională pentru gramaticile în cauză și observăm că cel mai eficient algoritm apare pentru gramatici care nu folosesc cicluri. Prezentăm de-asemenea entitățile de ”supercombinatori” prin prisma metodelor de compresie a gramaticilor și a posibilității de refolosire a acestora. Este interesant aplicarea acestor metode la procesarea unui corpus textual mai mare.

## Referințe

[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 STATICĂ A CODULUI CU ”CODECHECKER”

## PREZENTARE GENERALĂ

Execuția simbolică [4] este o metodă populară de analiză statică a codului sursă și este folosit atât la verificarea programelor cât și la programe de detecție a erorilor (bug detection). Execuția simbolică este definit prin interpretarea codului sursă, introducerea unor valori pentru variabilele care nu sunt definite la compilare – de exemplu valorile câmpurilor introduse în timpul rulării programului – și efectuarea operațiilor descrise în program în mod simbolic. La aceste aplicații motorul de analiză (analysis engine) încearcă explorarea simultană a diferitelor posibilități de rulare în codul sursă – a tuturor ramificărilor, cu toate că exploatarea exhaustivă a tuturor drumurilor este imposibilă din cauza numărului mare de posibilități.

Există un număr mare de încercări de creare de programe pentru verificarea corectitudinii programelor, însă resursele îndreptate spre găsirea erorilor sunt puține și numai în articole generale [1]. În prezentul articol încercăm ca – pe lângă trecerea în revistă a metodelor existente – să discutăm interacțiunea între diferite decizii: cum ele interacționează și uneori acestea se re-asigură. Rezultatul combinării a mai multor metode este un sistem care este mai mult decât ”suma” sistemelor independente. Accentul principal este de a focusa pe o unealtă software numit ”Clang Static Analyzer” [2] – în continuare numele fiind de ”analizor” – și pe infrastructura soft care este construit bazat pe analizor, numit CodeChecker [3], sistemul fiind construit în așa fel încât să avem o scalabilitate maximă.

Sistemul CodeChecker include analiza în timpul de execuție a consumului de memorie, o prezentare grafică pentru utilizatori, un mecanism pentru ascunderea detecțiilor pozitive false, analiză incrementală, descoperirea erorilor tipice, folosirea sistemului în șirul uneltelor de integrare continuă.

În continuare enunțăm direcții de cercetare care se pot face cu combinarea acestor unelte.

Cu toate că în prezent analizorul este capabil de procesarea codurilor sursă din familia de limbaje C – C/C++/Objective-C – subliniem că tehnicile definite în articolul prezent pot fi ușor implementate și aplicate în alte unelte de analiză statică a codului.

# ANALIZORUL STATIC CLANG

În această secțiune facem o scurtă trecere în revistă a Analizorului CLang, a mecanismului de funcționare și al metodei de execuție simbolică a codului. Definim modul de reprezentare a memoriei [6], modul de reprezentare a conexiunii între valorile unor variabile, respectiv locația acestora în memorie. Este important corespondența dintre valori și locația în memorie pentru Analizor-ul în modului de testare (prin test înțelegem un modul al Analizor-ului implementat pentru un anumit tip de eroare). În articol introducem noțiunea de calcul simbolic și menționăm că alegerea modurilor de reprezentare joacă un rol important în realizarea unui sistem viabil, capabil de analiză de software la o scală largă.

Deoarece testare tuturor posibilelor drumurilor de execuție este imposibil de testat într-un timp rezonabil, suntem nevoiți să introducem conceptul de buget al procesului de analiză, care este o estimare a timpului de lucru pe care putem să analizăm o parte dintr-un cod sursă. Scopul nostru este de a găsi cât se poate de multe erori cu cât mai puține pozitive false. Pentru aceasta am construit Analyser-ul care alocă prioritate mare drumurile de execuție "mai interesante" în vederea analizei și în același timp elimină drumurile ne-fezabile. Procesul de găsire a drumurilor probabile respectiv de eliminare a secvențelor ne-fezabile este realizat prin procesul de "tiered constraint solving" [5].

Analyzer-ul folosește un număr de euristici designate eliminării acelor detectări care au posibilitatea de a rezulta în apeluri de tip pozitiv fals.

Când un bug – o eroare – a fost găsită, raportăm drumul de execuție și constrângerile actuale. Din punctul de vedere al utilizatorului este însă nefolositor să prezentăm în forma inițială acele condiții în care apare eroarea.

Facem tot posibilul ca Analizer-ul să prezinte informațiile într-un mod concis dar în același timp informativ astfel încât să înlesnească procesul de corectare al erorii..

# CODECHECKER

Definim scalabilitatea procesului de analiză statică nu numai ca funcție al folosirii eficiente a resurselor computaționale, dar și în termenii folosirii eficiente a resurselor umane, și amintim timpul de dezvoltare (developer time) ca cel mai important exemplu. Sistemul CodeChecker este o unealtă soft destinat ușurării procesului de integrare al Analizor-ului și a celorlalte unelte soft în sisteme integrate de dezvoltare (ca și sistemele de continuous integration). CodeChecker este totodată un sistem de sine-stătător de care memorează erorile găsite de aceste unelte soft.

Având la dispoziție un buget limitat – în speță timpul dezvoltatorilor de soft – este important să evaluăm rapoartele care sunt cele mai promițătoare din punctul de vedere a resurselor alocate.

Menționăm că sistemul CodeChecker suportă analiza diferențiată care previne situația când dezvoltatorii introduc noi erori înaintea să corecteze cele existente.

# SUMAR

În acest articol am făcut o trecere în revistă a experiențelor culese în cursul dezvoltării unei Analizor Clang de ultimă generație și al sistemului CodeChecker asociat. Sperăm ca aceste experiențe să fie utile ca și referință pentru viitorii cercetători care vor dezvolta programe de analiză statică.

## Referințe


[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.

# PROGRAMARE ÎN SISTEME DE VIRTUALIZARE A REȚELELOR

Virtualizare funcțiilor unei rețea – network functions virtualization NFV – este o paradigmă nouă de a concepe și construi rețele. Decuplarea nivelului de software de resursele fizice de rețea printr-un nivel de virtualizare a introdus nevoia de a fi dezvoltate seturi de unelte de management NFV respective de orchestrare a acestor resurse (MANO).

Noi ne focusăm pe metodele de coordinare a funcțiilor de management care sunt implementate în blocuri funcționale diferite. Scopul acestor metode este de a obține familii de funcții MANO ce operează în medii distribuite. Pentru a obține o imagine de ansamblu a sistemului funcțional, am aplicat funcțiile pe un exemplu practic de sistem concret din lumea telecomunicațiilor, implementat în Open Stack.

## ÎNTRODUCERE

Scopul acestor note de prelegere este de a introduce noile concepte, cum ar fi virtualizarea funcțiilor de rețea (NFV), care sunt implementate în prezent în sisteme și rețele software complexe. Al doilea scop este să explicăm noile provocări ivite datorită acestor virtualizări și să arătăm elevilor cum să facă față cerințelor noului sistem pentru coordonarea funcțiilor de management și orchestrare a resurselor de rețea virtualizate care operează în medii distribuite.

# SISTEME COMPLEXE

Tema acestei serii de lecturi se încadrează în teoria sistemelor complexe, cu accent pus pe sistemele software complexe și rețelele complexe. Prin urmare, prelegerile încep cu o introducere concisă a teoriei sistemelor complexe, prezentând cu mai multă profunzime problemele și provocările considerate în cadrul configurației actuale a rețelelor și a sistemelor software.

Virtualizarea este introdus ca o paradigmă frecvent utilizată în gestionarea sistemelor software complexe și care implică introducerea unui nou strat abstract, o ediție virtuală a stratului de sistem și a funcțiilor sale, care evită introducerea dependenței între straturile de sistem.

# FUNCȚII DE MANAGEMENT ȘI DE ORCHESTRAȚIE

În rețelele de telecomunicații am introdusă o nouă paradigmă, numită Network Functions Virtualization (NFV), care decuplează partea funcțională a rețelei de resursele fizice pentru rețea, această separare este realizată cu un nou strat de virtualizare [2]. Această virtualizare introduce necesitatea dezvoltării seturilor de funcții de gestionare și orchestrare a NFV (MANO).

În acest scop, în cadrul Institutului European de Standarde pentru Telecomunicații (ETSI) un grup de lucru a fost creat pentru soluționarea acestei probleme. Cadrul arhitectural și cea de gestionare a virtualizării funcțiilor de rețea și de orchestrare este definit în [1]. În prezentarea de față ne concentrăm pe funcțiile de management și orchestrare implementate în diferite blocuri funcționale, pentru a realiza o operație fiabilă pentru funcțiile de management și orchestrare care operează în medii distribuite

# EXEMPLE

În lectura de față am definit FVN cu scopul de a explica problemele și principiile, metodele și tehnicile utilizate pentru soluționarea virtualizării. Exemplele și exercițiile sunt accesibile pentru studenți ca material didactic. Scopul este de a învăța – cu unelte de programare funcțională – pentru a coordona eficient funcțiile de management și orchestrare în sisteme complexe distribuite folosind VF.

Problemele și provocările coordonării funcțiilor de management și orchestrare sunt abordate cu ajutorul platformei OpenStack [3]. Aceasta este un sistem de operare cloud open source care integrează o colecție de module software care sunt necesare pentru a oferi un model stratificat de cloud computing. O astfel de tehnologie este necesară pentru a face față problemelor care apar din paradigma virtualizării în rețelele actuale, iar studenții care înțeleg soluțiile din OpenStack vor putea transfera cunoștințele lor către alte tehnologii existente cu același scop sau similar.

Provocările care rezultă din noile paradigme de rețea, precum și soluțiile acestora, sunt ilustrate prin exemple practice folosind tehnologia virtuală OpenStack și inspirate de problemele din industria telecomunicațiilor. Toate exemplele și exercițiile sunt elaborate în tehnologia virtuală OpenStack.

# Referințe

[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

# PROGRAMARE FUNCȚIONALĂ ȘI ”CLOUD COMPUTING” ÎN EDUCAȚIE

Calculul în platforme „cloud” – paradigma calculului în sisteme distribuite pe internet – a devenit o tehnologie cheie și – ca urmare – ele constituie o componentă importantă a multor programe de informatică a universităților de prestigiu.

Parte constitutivă a universului cloud - în proiectarea aplicațiilor - microserviciile sunt un concept cheie. Acestea realizează o descompunere funcțională intrinsecă a unui sistem și prin descompunere sistemul nou construit poate fi instalat pe platforme fără server, cum ar fi sistemul AWS Lambda.

## MICROSERVICII

Adoptarea unei arhitecturi bazat pe microservicii este frecvent recomandat atunci când se proiectează aplicații cloud [1,2]. Definit în mod vag, stilul arhitectural al microserviciilor este reprezentat printr-un un set de caracteristici frecvent întâlnite: implementare automatizată, pipe-line-uri goale de date cu puncte extremități ”inteligente”, scenarii de control decentralizat a datelor [2].

Se observă eforturi pentru a realiza migrarea aplicațiilor tradiționale de ”cloud” către arhitecturi bazate pe microservicii. Automatizarea este anevoioasă din cauza complexității crescute a procesului atunci când numărul de microservicii este numeroasă; deși mici unul câte unul, compunerea acestora poate să ducă la probleme de latență în sistem [3]. Și în acest caz este important o clarificare a noțiunilor: În această prezentare ne

concentrăm pe operațiile de orchestrare care sunt definite formal ca o conglomerare a logicii de afaceri cu acelea a serviciilor "cloud" granulare. În funcție de necesitatea aplicației pentru funcționare asincronă, avem diferite tipuri de "deployment" – setare a sistemului în arhitectura cloud – care poate fi orchestrată (adică există o componentă centrală care guvernează execuția) sau coregrafiată (fiecare micro-serviciu acționează independent). Având în vedere faptul că agilitatea și flexibilitatea sunt atribute foarte dorite pentru sistemele funcționale, de obicei este de preferat o implementare coregrafiată [1]. Cu toate acestea, aspectele nefuncționale ale microserviciilor, cum ar fi performanța de rulare, joacă un rol important în implementarea aplicației.

Prin urmare, intenționăm să familiarizăm studenții cu profilarea performanței în cloud în contextul AWS Lambda. Mai mult, completăm paradigma funcțională a sistemului AWS Lambda prin utilizarea unui framework Haskell pentru controlul experimentelor.

# CLOUD CENTRAT PE UTILIZATOR

În domeniul cloud computing un aspect important este asistarea utilizatorilor vizavi de deciziile luate. Tipic avem următoarele întrebări:

A. Cum folosește aplicația resursele virtualizate – resursele sistemului virtual?

B. Ce cantitate de resurse virtuale și de ce tip, de la care furnizor de cloud trebuie achiziționat pentru implementarea aplicației?

C. Pentru cât timp se va face achiziția? Cât va costa?

Aceste întrebări sunt tipic modelate ca o problemă de programare, funcționând sub presupunerea că nu există cunoștințe a priori despre aplicație. Un set simplu de cerințe este acela că aplicația este implementată cu succes, iar costurile să fie reduse la minimum.

# Arhitectura sistemului de "scheduler" pentru sistemul cloud

Definim un planificator – "scheduler" – BaTS [4], care  a fost dezvoltat pentru a ajuta utilizatorii atunci când își dezvoltă aplicațiile în cloud.

În acele cazuri este nevoie de o planificare automată pentru a realiza acest lucru și verifică în mod regulat progresul desfășurării. Într-o primă etapă, BaTS colectează statistici din eșantionare cu înlocuire. Aici este nevoie doar de un eșantion mic (30-50 de sarcini) pentru a calcula media și abaterea standard a timpului de rulare a sarcinilor pe diverse platforme cloud.

Modulul de estimare a bugetului efectuează apoi regresia liniară pentru a optimiza această fază.

# Folosirea metodologiei BaTS pentru virtualizare "ușoară"

Prezentăm studenților problema asistării dezvoltatorilor de aplicații care caută să selecteze cele mai bune opțiuni în ceea ce privește virtualizarea ușoară atunci când desfășoară aplicația lor ca un set de microservicii.

# Sistemul "AWS Lambda"

Sistemul "AWS Lambda" este o resursă virtualizată de calcul extrem de ușoară – lightweight – și care este oferită de Amazon.

Granularitatea sistemului – fiind "lightweight", minimalist – este la nivel de funcție, în consecință timpul de tipic de execuție a funcției este cel mult în ordinea de secunde. De asemenea, acest sistem presupune o arhitectură tip asincron – care nu blochează execuția pe mai multe ramuri. Menționăm că există 4-6 de tipuri de mașini virtuale "AWS Lambda" care au prețuri exprimate prin transfer de date respectiv timp de rulare, cu unitate de măsură EUR / (GB * sec).

# Implementarea Haskell a API-ului AWS

Pe baza unei implementări în Haskell a API-ului AWS [5], ne propunem în continuare să replicăm metodologia de estimare BaTS pe resurse virtualizate "lightweight".

# Exerciții de folosire a sistemului

Îi instruim pe studenți să coreleze performanța funcției "factorial" folosind diferite funcții lambda; să facă o estimare a eficienței prețurilor. De exemplu, le cerem stabilirea variantei care are cel mai bun randament la un preț minim. De asemenea, cerem să identifice cea mai eficientă sau cea mai profitabilă combinație: cel mai mic cost pentru cea mai bună performanță.

# Validarea sistemului AWS Lambda

Funcțiile sistemului „Lambda AWS" sunt executate într-un mediu virtualizat asemănător unui container de tip docker. Sistemul AWS alocă o cantitate variabilă de resurse de calcul care este proporțională cu memoria DRAM solicitată de utilizator.

Pentru a determina modul în care funcțiile lambda pot procesa diferite sarcini de lucru, comparăm separat resursele folosite independent: CPU, memoria folosită, lățime de bandă I/O, respectiv latență. Aceste micro-valori de referință se calculează prin lansarea – în cadrul funcțiilor – diferitelor tipuri de teste pentru calculul cu memorie, calcul cu I/O, respectiv transfer pe rețea bine cunoscute. Exemplu de acest tip de sistem ar fi calculul numerelor prime, rularea benchmark-ului pentru I/O, sau citirea sau scrierea datelor în AWS S3.

Transferul metodologiilor consacrate este cheia educației. Ca activitate viitoare, am dori să sprijinim funcțiile Haskell AWS Lambda desfășurate prin implementarea în Haskell al API-ului AWS.

# Referințe


[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.

# GRAMATICI DE ATRIBUTE ÎN SISTEME ”TYPE-SAFE”

Gramaticile de atribute – attribute grammars – reprezintă un formalism puternic și declarativ pentru programele care, prin definiția lor pot fi implementate modular. Deși un sistem complet cu compilator de gramatică de atribute poate fi folosit în cazuri specifice, implementarea acestora este dificilă, iar mentenanța pe termen lung a acestora este greoaie – aproape imposibilă. În mod concret, efortul de mentenanță al unui sistem tradițional cu gramatici de atribute este atât de mare încât majoritatea sistemelor nu mai sunt active.

Abordarea noastră pentru implementarea gramaticilor de atribute este de a le implementa într-un limbaj de programare funcțională modernă. Îmbunătățim o variantă de compilator de gramatică de atribute anterioară bazată pe sisteme ”zipper”, folosind varianta neintruzivă (fără modificarea tipurilor de date definite de utilizator) și sigure de tip – type-safe. În plus, obținem o sintaxă mai clară folosind variante moderne în Haskell. Considerăm că metoda noastră poate fi folosită în practică ca soluții elegante, eficiente și modulare la provocările programării din viața reală.

## INTRODUCERE

Gramatici de Atribute -- attribute grammars (AG) în engleză – sunt un formalism declarativ care a fost propus de Knuth [7] la sfârșitul anilor 60 și care permite definirea și implementarea de sisteme într-un mod modular și convenabil. Un AG concret se bazează pe o gramatică fără context – context-free grammar – pentru a defini sintaxa unei limbi și pe atributele asociate cu regulile de producții gramaticale care definesc semantica limbii respective. AG-urile au fost utilizate în producție pentru a specifica limbaje reale de programare, cum ar fi Haskell [2], precum și algoritmi performanți de rescriere – pretty-printing – [16],

tehnici de de-forestare [4] și la implementarea a sistemelor puternice de tip [11]. La programarea cu AG, modularitatea se realizează datorită posibilității de a defini și de a folosi diferite aspecte ale calculelor ca atribute separate.

Considerăm atributele ca unități de calcul distincte, de obicei destul de simple și modulare, care pot fi combinate pentru a rezolva probleme complexe de programare. Un avantaj ce rezultă din definiția precedentă este că ele pot fi analizate și întreținute independent, acest lucru ușurează dezvoltarea și evoluția programului.

AG-urile s-au dovedit a fi deosebit de utile pentru a specifica calculele pe arbori: dat fiind un arbore, există în literatură mai multe sisteme AG, cum ar fi [14,3,8,17], care prelucrează specificațiile date, pentru care este necesar calculul de valori sau atribute și efectuate calculele respective. Eforturile de proiectare și codificare depuse în crearea, îmbunătățirea și întreținerea acestor sisteme AG sunt totuși extraordinare, ceea ce este adesea un obstacol în atingerea succesului.

O abordare alternativă din ce în ce mai populară a utilizării AG-urilor se bazează pe încorporarea lor ca ”cetățeni de primă clasă” – first class citizens – în limbajele de programare [12, 9, 13, 15, 18, 1]. Acesta evită sarcina implementării unui limbaj nou, respectiv al unui sistem asociat prin găzduirea acestuia în limbaje de programare de ultimă generație. În urma acestei abordări, se exploatează elementele și infrastructura limbajului modern de programare care sunt deja furnizate de aceste limbi și se concentrează pe particularitățile limbajului specific domeniului – domain specific language – care se implementează.

Unealta de bază în acest caz este zipper-ul funcțional – functional zipper – [6], care este o abstractizare puternică ce simplifică foarte mult implementarea algoritmilor de traversare a arborilor și care efectuează o multitudine de actualizări locale. Zipperele funcționale au fost aplicate cu succes pentru definirea unui AG în limbajul Haskell [9,10]. În ciuda eleganței sale, această soluție are dezavantajul major care împiedică utilizarea acesteia în aplicații reale: atributele nu au fost memorate în cache, ci erau recalculate în mod repetat, ceea ce a redus performanța sistemelor. Recent, acest defect a fost eliminat [5] și înlocuit cu unul diferit, o abordare case s-a dovedit a fi intruzivă, adică pentru a încorpora structurie de date definite de utilizator, acestea necesită ajustare, un lucru extrem de dificil.

În această lucrare prezentăm un mecanism alternativ la atributele de memorie cache bazate pe o grilă infinită de autoorganizare. Acest grafic este așezat în partea de sus a tipului de date algebraice definite de utilizator (ADT) și reflectă structura acestuia. Tipul de date definit în sine rămâne neatins. Încorporarea se bazează apoi pe două (mai degrabă decât una) fermoare coerente care parcurg structurile de date în paralel. Pe deasupra de a fi neintruzivi, soluția noastră este complet sigură de tip. Extensiile moderne Haskell, cum ar fi ConstraintKinds, ne permit să propagăm constrângerile din ADT, eliminând distribuțiile de tip runtime prezente în versiunile anterioare.

Un alt avantaj secundar al utilizării caracteristicilor moderne Haskell este o sintaxă mai curată, cu mai puțin cod generat prin intermediul șemelor "Template Haskell".

# Referințe


[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)

# CÂT DE ECONOMIC – CÂT DEVERDE – ESTE PROCESUL TĂU?

Similar cu produsele ”Bio”, lumea actuală evoluează înspre a deveni un ecosistem mai conștient de natură. Inițiativa ecologică definește două obiective principale: reducerea consumului de energie și utilizarea surselor naturale în producția de energie electrică.Una dintre provocările producătorilor de baterii este timpul de funcționare a bateriilor. O altă provocare este dimensiunea acestora care afectează foarte mult forma și greutatea dispozitivului în care sunt puse. Bateria trebuie să fie puțin mai ușoară comparat cu dispozitivul care are nevoie de acesta.

Provocarea de aici este modul în care dimensiunea acestuia este mai mică și mai ușoară și cu siguranță o eficiență ridicată în ceea ce privește timpul de funcționare al dispozitivului mobil fără a fi încărcat.

Adăugat provocărilor hardware, mai sunt și provocările software: software-ul în sine ar trebui să sprijine economiile de energie. Trebuie să consume deci puțin fără a limita experiența utilizatorului – este un obiectiv latent dar important al fiecărei sistem de software ce rulează de dispozitive portabile [1]. Acest obiectiv apare de obicei atunci când cerințele aferente sistemului se schimbă și funcționarea îndelungată fără întrerupere devine o cerință importantă. Reamintim faptul că consumul de energie al unui dispozitiv mobil este influențat de rularea aplicațiilor, de frecvența de utilizare a serviciilor, chiar și de dispoziția utilizatorului. În consecință dezvoltarea de software pentru astfel de dispozitive este deja o provocare [4]. S-ar putea spune, provocarea de dezvoltare software este întotdeauna aceeași, dar trebuie să subliniem „mobilitatea” – lunga disponibilitate fără reîncărcare – este proprietate cheie a sistemului. Starea de alimentare a bateriei determină, de asemenea, performanța sistemului datorită configurației nivelului sistemului de operare - bine cunoscut sub numele de „preferințe de economisire a energiei”.

Este și mai greu, dacă unul (în cazul nostru profesorul) trebuie să pregătească elevii pentru astfel de provocări [6]. Toate „cele mai bune practici” și „sfaturi de economisire a energiei” deja cunoscute trebuie prezentate într-un context, ușor de înțeles pentru elevi.

Această prezentare poate fi realizată prin poziționarea conceptelor într-un mediu cunoscut, cum ar fi testarea de software respectiv automatizarea testelor [2].

Acesta este ceea ce ne propunem cu acest tutorial, acesta este conținutul secțiunilor viitoare, începând cu propunerea urmată de exemple și închizând cu sfaturi suplimentare despre îmbunătățire.

Accentul principal este pus pe consumul de energie al software-ului de lucru și procesele sale de dezvoltare, unde fiecare etapă de dezvoltare joacă un rol semnificativ.

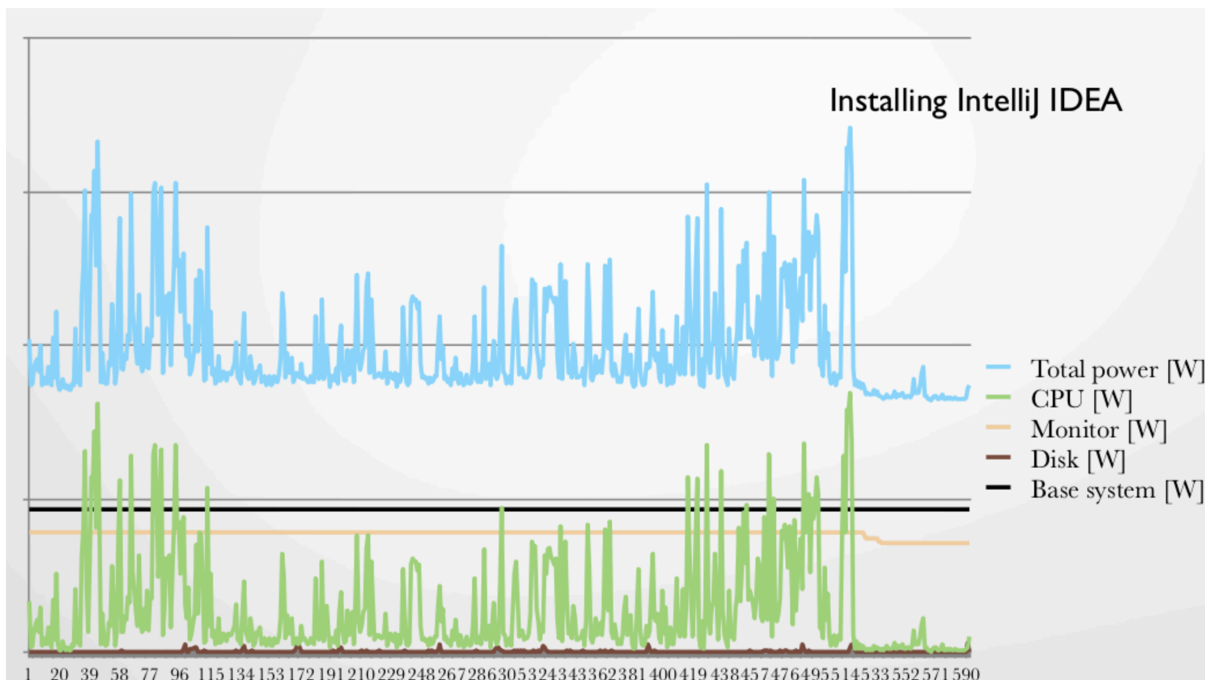
Având în vedere orice proces de dezvoltare a software-ului, energia este consumată în timp ce se analizează problemele, se construiește și se evaluează codul [3]. Instrumentele software sau hardware trebuie utilizate pentru a implementa monitorizarea consumului de energie pentru software-ul rulat în partea de sus a sistemelor de operare selectate și pentru evaluarea consumului de energie. Scenariile de utilizare obișnuite sunt de a monitoriza consumul de energie al software-ului selectat [5], dar vom analiza și posibilitatea de a utiliza aceste instrumente pentru a măsura cât de verde este procesul care produce versiunea finală a software-ului de lucru.

Exemplele acoperă o varietate de situații. Începând cu cazul profilării energetice a software-ului de lucru terță parte în scenarii specifice de utilizare, subliniem proprietățile cheie (avantaje și dezavantaje) ale instrumentelor existente. În timpul testării unității a codului de software în curs de dezvoltare, prezentăm utilizarea tipică a software-ului de profilare a energiei.

Abilitatea de a scala abordarea de măsurare de la profilarea unui fragment de cod sau a unei singure aplicații la analiza consumului de energie a lanțurilor de instrumente este ultimul exemplu prezentat. Acesta prezintă o abordare generică pentru profilarea energiei și are ca scop înlocuirea semnului de întrebare a titlului tutorialului cu o perioadă reprezentând evaluarea rezultatelor pentru fiecare caz specific acoperit de tutorial.

## 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




# Referințe

[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.

[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.

# PROGRAMAREA FUNCȚIONALĂ ÎN IOT

## NOTĂ

Această contribuție este o versiune extinsă a articolului din RWDSL18 [2]. În articolul prezent folosim același DSL – domain specific language – la care s-au operat numai cu modificări minore. În contribuția prezentă ne concentrăm pe DSL-ul mTask care poate fi folosit pentru programarea uneltelor IoT. Definim un simulator pentru uneltele mTask ca  un sistem iTask.

## ÎNTRODUCERE

În prezent, multe dispozitive sunt echipate cu un microprocesor simplu pentru a controla acestea. Exemple tipice sunt termostate, becuri, prize electrice, alarme de incendiu, deschizători de uși și așa mai departe. Când aceste dispozitive pot comunica între ele sau cu un computer la distanță, se spune că fac parte din Internet of Things, IoT. Microcomputerele de pe aceste dispozitive sunt foarte accesibile și devin omniprezente.

Dispozitivele scumpe, cum ar fi mașinile și aparatele cu o sarcină foarte complexă sunt echipate cu un computer integrat integrat și un software adecvat. Pentru majoritatea dispozitivelor IoT mici și relativ ieftine, un astfel de computer încorporat este prea scump sau consumă prea multă energie; un microprocesor simplu și ieftin este utilizat pentru a executa software-ul. Aceste sisteme au o putere și o memorie de calcul foarte limitate, de obicei memorie flash de 30 KB până la 4 MB pentru a stoca programul. Durata de viață a acestei memorii este limitată la 1000 de cicluri de scriere. Pentru a stoca variabile, heap-ul și stiva sistemele au 2 până la 40 KB de memorie RAM.

Limitările de viteză și de capacitatea de memorie ale procesorului exclud utilizarea unui sistem de operare. Aparatul execută doar programul care controlează dispozitivul. Chiar și programele de control de pe aceste dispozitive IoT constau din mai multe sarcini. De exemplu, pentru a verifica starea unui buton de zece ori pe secundă, pentru a actualiza un afișaj în fiecare secundă, pentru a măsura temperatura de două ori pe minut și pentru a schimba încălzirea după cel puțin cinci minute, dacă nu este apăsat butonul mai devreme. Datorită diferitelor perioade de timp și dependențelor acestor sarcini, programul de control tinde să devină destul de dezordonat, independent de limbajul de programare utilizat. Mai mult, dispozitivele IoT execută programe separate pentru restul aplicației din IoT și comunică folosind o multitudine de protocoale. Acest lucru face ca dezvoltarea și întreținerea aplicațiilor IoT să fie complexă și predispusă la erori.

Programarea orientată spre sarcini – Task-oriented programming, TOP – oferă task-uri (sau thread-uri) ușoare care se pot compune pentru sisteme mai complexe. Sarcinile sunt evaluate pas cu pas și aici se pot inspecta valoarea curentă a altor sarcini după un astfel de pas. TOP este implementat în sistemul iTask [4,5] încorporat în Clean [6]. În sistemul iTask, sarcinile primitive sunt colectarea informațiilor prin intermediul unui formular web generat automat sau prin colectarea datelor din alte programe și depozite de date. Un set puternic de combinatoare este utilizat pentru a compune sarcini la sarcini mai complexe. În această lucrare, arătăm că TOP este foarte potrivit pentru programarea dispozitivelor IoT. Sarcinile primitive furnizează valoarea curentă a intrărilor și senzorilor. Constructorii foarte similari cu sistemul iTask sunt folosiți pentru a combina sarcinile cu sarcini mai complexe.

Dispozitivele IoT au în mod obișnuit sarcini dependente care controlează senzorii, actuatoarele și comunicarea dispozitivelor. Programarea acestui lucru într-un stil TOP oferă programe concise. Executarea acestor sarcini în limitele microrcontrolerelor cu o putere de procesare foarte limitată și unele KB-uri de memorie RAM merită gândite. Datorită limitărilor severe ale microcontrolerelor utilizate, nu putem transporta sistemul iTask la dispozitivele IoT, deoarece un program tipic iTask necesită aproximativ 100 MB de spațiu de acumulare. Definim un limbaj specific de domeniu încorporat, eDSL, numit mTask pentru dispozitivele IoT. Acest eDSL este încorporat în sistemul iTask, deoarece intenționăm ca aceste limbi TOP să fie complet interoperabile.

Contribuțiile acestei lucrări sunt:

- Această lucrare introduce un limbaj de programare funcțional bazat pe sarcini pentru dispozitivele IoT. Comparativ cu limbajul nostru anterior pentru programarea prin microprocesor [3], controlul periferic imperativ este înlocuit de constructe transparente referențiale.
- Demonstrăm cum se face un DSL integrat extensibil, multi-vizualizare, sigur de tip, încorporat. Acesta este un eDSL fără etichetă [1].
- Codul generat rulează pe dispozitive mici și lente, precum și pe mașini mai mari și pe o mașină simulată.
- Datorită utilizării Arduino C ++ ca limbaj intermediar, acest eDSL funcțional rulează pe mai multe microcontrolere diferite.

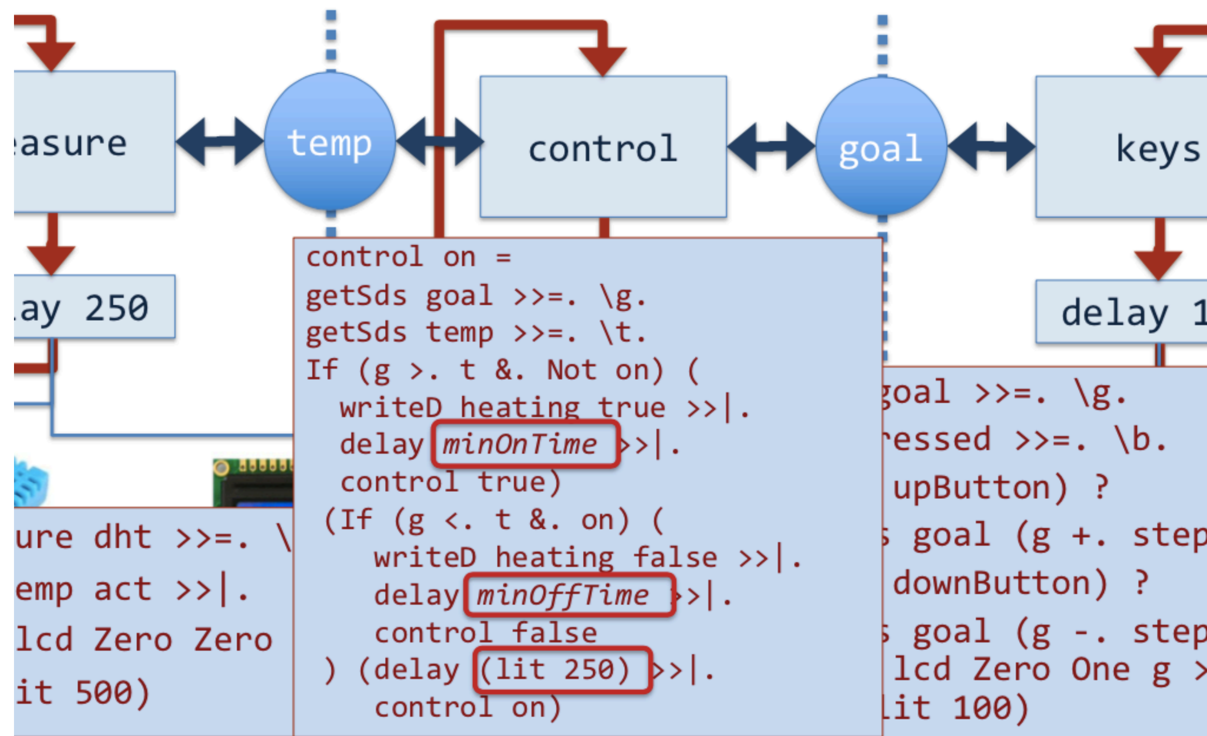
Simularea la nivel înalt a programelor mTask dintr-un program iTask oferă posibilitatea de a vizualiza efectul programului eDSL și de a manipula mediul simulat pentru a experimenta comportamentul specificat. Într-un astfel de simulator este mult mai ușor să manipulezi timpul și senzorii decât într-o configurație din viața reală, de exemplu, putem schimba temperatura semnalată de un senzor prin apăsarea unui buton în loc să expunem fizic dispozitivul IoT la aceste temperaturi.

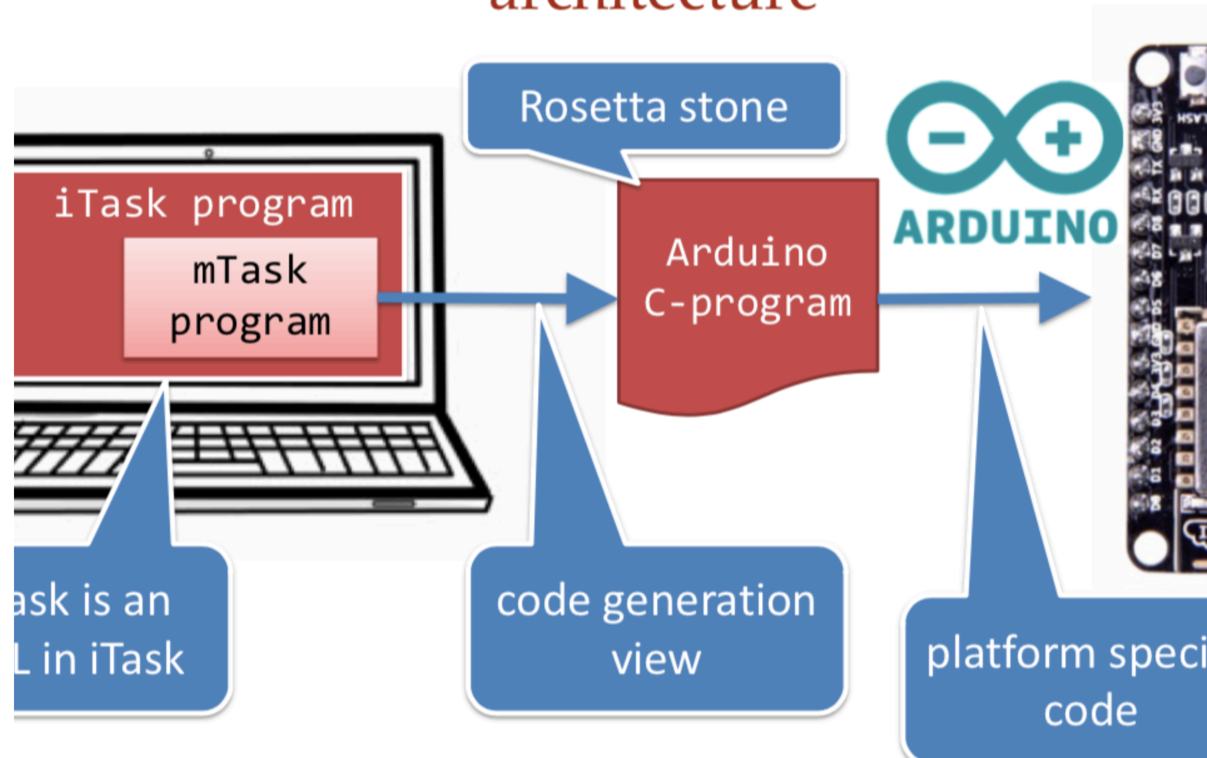
# Referințe


[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

# ÎNȚELEGEREA CODULUI SURSĂ CU CODECOMPASS

## SUMAR

Dezvoltarea unui sistem și întreținerea – la nivel de cod sursă – sunt etape separate în viața activă al unui sistem informatic.  Aceste etape au caracteristici diferite, deci și suportul – la nivel de ”unealtă software” trebuie să fi adecvat – diferit – pentru cele două etape. În timpul dezvoltării scriem – implementăm – în mare măsură coduri ”noi” care necesită suport pentru completarea codului, potrivirea parantezelor sau a blocurilor neterminate etc. În aceste sisteme doar câteva fișiere sunt implicate – mai mult sau mai.  În schimb, la întreținere numai consultăm sau navigăm în principiu într-o bază de cod existentă; acesta implică un număr mare de module și fișiere cu diferite niveluri de abstractizare [1]. În faza de dezvoltare soft avem intenții clare, spre deosebire de faza de înțelegere a codului, unde sarcina este de a recupera scopul inițial al anumitor fragmente de cod.

În mediul industrial [2] un proiect poate avea milioane de linii de cod. Pentru sistemele existente care au anvergură mai mare, unde baza de cod a fost dezvoltată și întreținută zeci de ani de către echipe cu fluctuații mari, și unde intențiile originale nu mai sunt cunoscute, unde documentația nu este de încredere sau lipsește, singura informație fiabilă este codul în sine. Înțelegerea unor astfel de sisteme software mari este o sarcină de bază, dar de obicei foarte provocatoare. Acest lucru implică faptul că este nevoie de sprijin cu unelte software [3].

Atunci când ne familiarizăm cu un cod sursă încă necunoscut, primul pas este de a găsi părțile relevante ale sistemului. Acest proces necesită localizarea rapidă a funcțiilor, pe baza unor entități a căror nume este achiziționată din mesaje LOG sau din alte resurse.

Pasul următor este extinderea cunoștințelor despre sistem prin diagrame, prin înșiruiri de apeluri de funcții, etc. În pasul final al analizei am dori să ne testăm cunoștințele colectate prin mesaje de control de versiuni, informații arhitecturale și referințe pe module conexe.

# SISTEMUL CODECOMPASS

Sistemul CodeCompass [4, 5] este un framework de înțelegere a codului sursă care este accesibilă oricui (este publicat cu licență OPEN-SOURCE). Sistemul oferă o arhitectură extensibilă – prin unealta de plugin – care ne permite adăugarea diferitelor instrumente de analiză a codului care produc vizualizări diferite, cu diferite sisteme de colectoare de informații, cu calculul a diferitelor metrici [6], etc. Cel mai important obiectiv de proiectare a fost o implementare a sistemului CodeCompass care permite folosirea sistemului pentru proiecte industriale la scară largă.

În prima etapă, produsul trebuie analizat: toate informațiile sunt colectate și stocate într-o bază de date care permite modulului de servicii să furnizeze vizualizările necesare.

Pentru o căutare rapidă CodeCompass folosește indexarea textului care duce la navigarea independentă de limbă în codul sursă. Întrucât principalul scop este să ofere informații exacte despre elementele limbajului, identificarea simbolurilor după numele lor nu este suficientă. Folosim infrastructura compilatorului LLVM pentru a identifica simbolurile cu precizie și pentru a rezolva entitățile numite folosind arborele sintaxei abstracte. CodeCompass este extins de către parserele de limbi. Cele mai acceptate limbi sunt C / C ++, dar Java și Python sunt gestionate parțial.

Pe lângă simboluri și nume, avem și alte informații suplimentare care sunt de asemenea stocate în baza de date, cum ar fi relațiile dintre nodurile AST (apeluri funcționale, moștenire) și fișiere (relația furnizorului de interfață, incluziune etc.). Acestea sunt utilizate pentru afișarea unei imagini la nivel arhitectural despre sistem pe baza utilizării simbolurilor [7].

Baza de cod nu este singura sursă de documentații. Mesajele de LOG ale unui sistem de control al versiunii conțin informații care sunt importante pentru a înțelege de ce s-au întâmplat anumite modificări la modulul dat. CodeCompass citește și depozitul Git, dacă este cazul.

CodeCompass este de asemenea echipat cu funcționalități avansate. Poate afișa funcțiile generate de compilator, care lipsesc din sursă. Analiza pointerului ajută la înțelegerea ce variabile se referă la același obiect. Putem inspecta relațiile de apeluri funcționale chiar dacă cele invocate prin intermediul unei funcții virtuale sau a unui indicator de funcții.

# SUMAR

Înțelegerea codului necesită suport specific pentru instrumente pentru înțelegerea software-ului la scară largă. Analizăm și clasificăm instrumentele de înțelegere a codurilor după arhitectură și funcționalități pentru a examina capabilitățile acestora.

## Referințe

[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.

# ŞEME HPC ÎN PROGRAMAREA FUNCȚIONALĂ

Cercetările privind extinderea aplicabilității scheletelor în mediul de înaltă performanță de calcul este punctul cheie în abordarea paralelă fp. Adaptarea cunoștințelor anterioare despre programarea scheletului a sistemelor multicore eterogene are ca rezultat viteze mai mari, unde măsurătorile și comparațiile evaluează noile procese de paralelizare.

Prototipurile scheletului sunt definite în funcționalități și coordonări. Studiile de caz ilustrează conexiunile cu alte tipuri de sisteme distribuite, care sunt importante datorită structurii multistratate a acestora. Proprietățile distribuite ale sistemului date în moduri executabile sunt testate de scheleturi funcționale și programare distribuite pe clustere și grile.

## Referințe


[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.