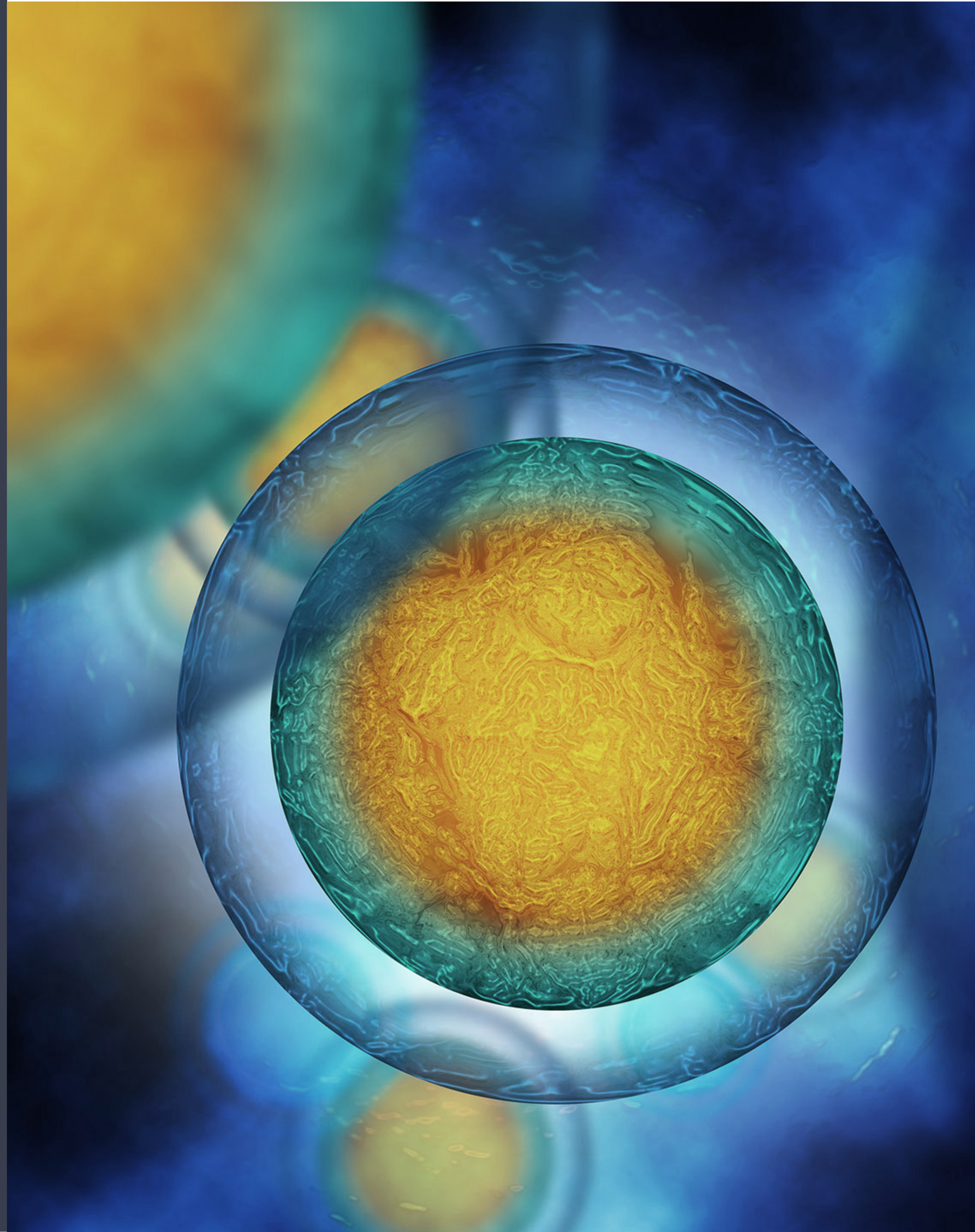
FE3CWS

OKTATÓI TRÉNING AMSTER-DAMBAN

A 2017-1-SK01-KA203-035402 számú ERASMUS+ projekt O2 megjelölésű intelektuális kimenete



BEVEZETÉS

gyanánt

- 6 téma szoftverfejlesztésről, -megértésről és helyességről
- Elérhető 7 nyelven: angolul, magyarul, szlovákul, horvátul, románul, bolgárul és portugálul

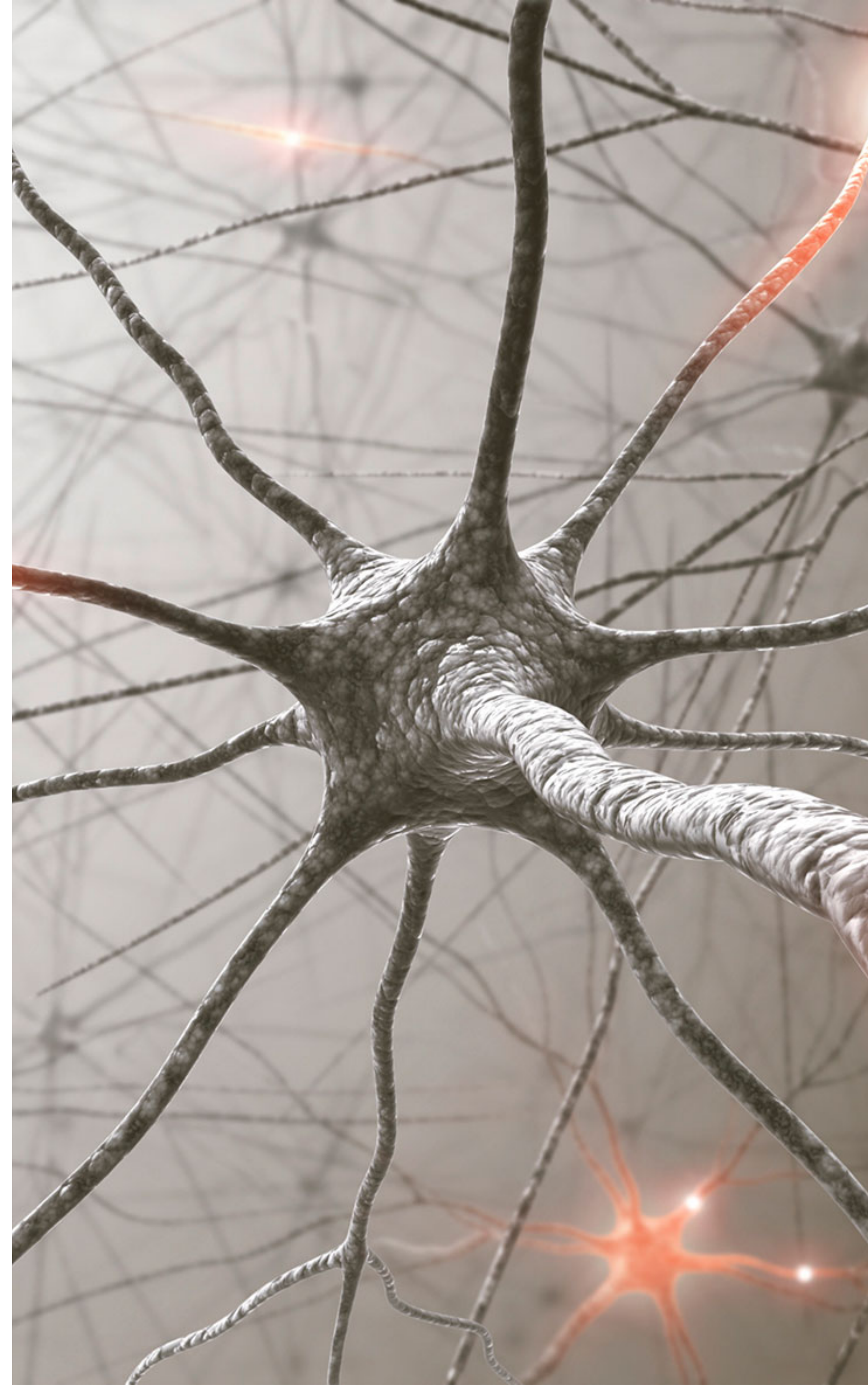
Co-funded by the
Erasmus+ Programme
of the European Union

© European Union, 2017-2019

A kiadványban szereplő információk és nézetek a szerző(k) álláspontjának felelnek meg, és nem feltétlenül tükrözik az Európai Unió hivatalos véleményét. Sem az Európai Unió intézményei és szervei, sem a nevükben eljáró személyek nem tehetők felelőssé az abban foglalt információk felhasználásáért.

TARTALOMJEGYZÉK

1. Felhasználóközpontú felhőalapú számítástechnika az oktatásban
2. Az energiahatékonyság szoftvertesztelés során történő mérése az oktatásban
3. A környezetbarát szoftvermérnöki megközelítés
4. A feladatorientált programozás oktatása
5. Színezett Petri Hálók Oktatásának Interaktív Megközelítése
6. CodeCompass: egy kiterjeszthető kódmegértési keretrendszer



Felhasználóközpontú felhőalapú számítástechnika az oktatásban

Ana Oprescu

Amszterdami Egyetem,
Amsterdam, Hollandia
a.m.oprescu@uva.nl

Abstract. A felhőalapú számítástechnika mostanra kulcsfontosságú technológiává vált, így sok informatikai tantervnek részét képezi. A felhasználóközpontú kutatás a futásidők és telepítési költségek becslési stratégiáira összpontosít valós alkalmazásoknál.

1 Felhasználóközpontú felhőalapú számítástechnika

A felhőalapú számítástechnika területén belül fontos szempont a felhasználói döntések támogatása, amely döntések az alábbi kérdésekhez köthetők:

- Hogy viselkedik az alkalmazás a virtualizált erőforrásokon?
- Az alkalmazás telepítéséhez mennyi és milyen típusú virtuális erőforrás beszerzése szükséges az egyes felhőszolgáltatóktól?
- Mennyi időre? Mennyibe fog kerülni?

Ezek a kérdések jellemzően ütemezési problémaként modellezhetők úgy, hogy közben az alkalmazásról nem feltételezünk eleve ismereteket. Tipikus követelmény szokott lenni, hogy az alkalmazás sikeresen kiszállításra került a költségek minimalizálása mellett.

1.1 Egy felhőalapú alkalmazás ütemezőjének architektúrája

A BaTS ütemezőt [4] azért fejlesztették, hogy segítse a felhasználókat a felhőalapú alkalmazásaik szállításában. Ennek elérése érdekében önütemező megközelítést alkalmaz, a szállítás folyamatát pedig rendszeres időközönként ellenőrzi.

Az 1 ábra a BaTS architektúráját illusztrálja. A *mintavételezési fázisban* a BaTS statisztikát gyűjt az alkalmazás futásidejű folyamatairól, mintevételezést használva pótlással. Itt csak néhány mintavételre van szükség (30-50 folyamat) a futásidő átlagának és a szórásának kiszámításához különféle felhőalapú megoldásokon. Lineáris regresszió felhasználásával a számítási kapacitás és teljes futásidő megbecsülhető.

A *végrehajtási fázisban* a konfiguráció szabályos időközönként kiértékelődik azt ellenőrizve, hogy az ütemezés még mindig kielégíthető-e. A kvóta elfogyasztása esetén nyereségesebb (jobb költség/teljesítmény aránnyal rendelkező) gépek beszerzése szükséges. Ha pedig a teljes futásidő a szűk keresztmetszet, akkor gyorsabb gépek szükségesek.

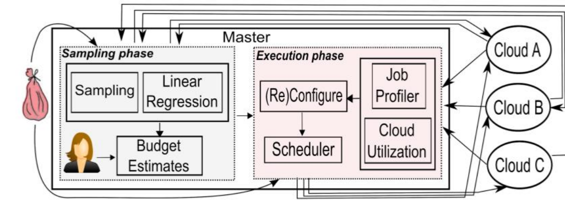


Fig. 1. BaTS architektúra.

2 A BaTS módszertan használata AWS erőforrásokon

Ebben a fejezetben bemutatjuk, hogyan segíthetjük az alkalmazások tulajdonosait a virtualizált erőforrások kiválasztásában, Amazon EC2 (AWS) [7] platformra való szállításkor.

2.1 Az optimalizált mintavételezési fázis

A fő elképzelésünk, hogy az átlagos futásidőt használjuk minden virtualizált erőforrástípusra a számítási kapacitás és teljes futásidő becsléséhez. Azonban ezen statisztikák előállítása jelentős költségvonzattal járhat az AWS EC2 lehetőségek sokszínűsége miatt (jelenleg 123 [8] típus). A 2. ábrán néhány válatlenszerűen kiválasztott alkalmazásfolyamat látható, amelyek valamilyen eloszlást követnek. Ezen folyamatok futásidejét használjuk egy felhőalapú platform statisztikáinak meghatározására. Az összes felhő platform statisztikáinak kiszámítása után elvégezhetjük a számítási kapacitás és teljes futásidő becslését. Abban az esetben, ha az összes jelenleg elérhető AWS EC2 platformot ki szeretnénk értékelni, 30 folyamat futtása lenne szükséges mind a 123 géptípuson. Ha csak egyszerűen futtatnánk a véletlenszerűen kiválasztott folyamatokat (összesen 3690-et), az túl hosszú és költséges mintavételezési fázist eredményezne. Ez két okból is irrelevánssá tehetné a felhasználói döntéseket: a) túl kevés folyamat maradna futtatásra, és b) a felhasználó kvóta korán elfogyhat.

A nagy mennyiségű folyamat számának csökkentésére lineáris regressziót használtunk, amelyek futtatása szükséges a statisztikák előkészítésére. Ugyanazt a 7 véletlenszerűen kiválasztott folyamatot futtatjuk mindegyik géptípuson, majd a futásidőket feljegyezzük [9]. Ezután futtatunk 23 véletlenszerűen kiválasztott folyamatot azokon a gépeken, amelyek elsőként elérhetővé válnak. A 3. ábra szemlélteti ezt a megközelítést. A 7 ismételt folyamat futásidejét felhasználva lineáris kapcsolatot létesítünk az egyes géptípusokon keresztül futtatott folyamatok futásidejével. Ezen kapcsolatok segítségével pedig a 23 futásidőt leképezzük az összes többi géptípusra. A leképezések elkészítése után 30 futásidő áll rendelkezésre minden géptípusra, minössze 884 folyamat futtatásával a teljes 3690 futtatás helyett.

2.2 Az árazás nagyságrendjei

Ha megvannak a futásidő becslései, akkor a kvóta és teljes futásidő becsléseket elvégezhetjük a módosított Bounded Knapsack algoritmussal [11]. Azonban ha az AWS EC2 erőforrásokat vesszük különböző árazási modellekkel, akkor ez a megközelítés nem igazán skálázódik a különböző nagyságrendekre.

A probléma kezelésére a Bounded Knapsack determinizmusát egy általános algoritmus skálázhatóságával vegyítettük [12]. Egy általános algoritmust felhasználva közelíteni tudtuk az alkalmazás megvalósítható ütemezésének Pareto-hatékonyságát adott géptípus halmazra. A 4. ábra a valós Pareto-hatékonyságot és két becslést mutat egy alkalmazáshoz, a folyamatok futásidejének kombinált eloszlásával. A mi megoldásunk pontos Pareto-hatékonyságot generál 1 másodpercen belül.

A kísérlet fő tanulsága, hogy a valós Pareto-határ jó lefedettségéhez a célfüggvénynek jutalmaznia kell a gyors/olcsó futásidőt.

2.3 A számítás végső fázisa

A számítás utolsó fázisában azzal a feltételezéssel élünk, hogy a számítási idő "fokozatos" [10]. Ezen a ponton az alkalmazás definíció szerint túl kevés folyamatot tartalmaz, hogy a feltétel igaz legyen. Több megközelítést elemeztünk a probléma megoldására, szem előtt tartva az alkalmazás végső számításának pontosságát, hogy a folyamatoknak foglalt erőforrások minél optimálisabbak legyenek. A megközelítéseink meglehetősen változatos eredményt mutattak a fennmaradó futásidő tökéletes ismeretétől a véletlenszerű futásidő becslésig. Mivel az eredmények kevéssé voltak jelentősek, azért a problémakört más fázisra, nevezetesen a kvóta és teljes futásidő becslésének fázisára kellett áthelyezni.

Ennek érdekében a BaTS számontartja a felhasználatlan időegységek becsült arányát. A BaTS kezeli azokat a tartalékokat, amiket a végső kiosztható időegységekből

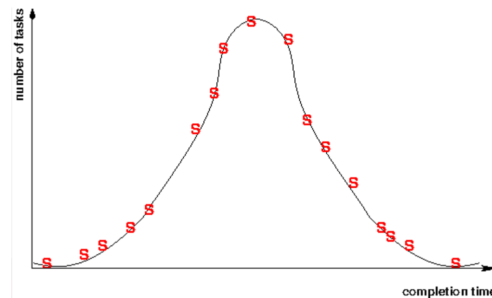


Fig. 2. Alapértelmezett mintavételezési fázis.

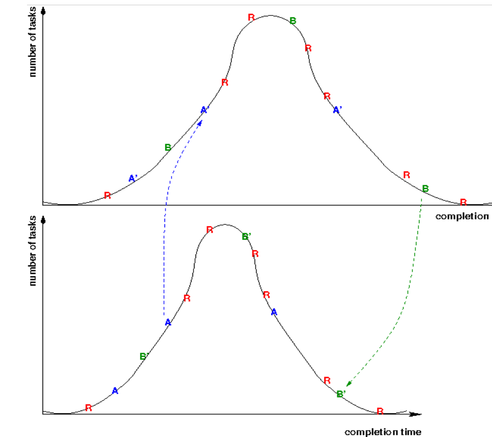


Fig. 3. Optimalizált ütemezési fázis

kilógó folyamatoknak tart fenn, és az ütemezésükhöz virtualizált erőforrásokat rendel.

3 A BaTS technológia használata az oktatásban

Az Amszterdami Egyetem számítástudományi mesterképzésében a "Webes szolgáltatások és felhő alapú rendszerek" évek óta a tanterv részét képezi. A kurzus fontos célja megismertetni a hallgatókkal a felhő alapú rendszerek kihívásait. A kurzus gyakorlati részeként egy kezdetleges ütemezőt kell fejleszteni avirtualizált erőforrásokhoz, és elemezni kell azok viselkedését egy "házon belül" és egy kereskedelmi felhő rendszerben is. Előbbi beállítások tartalmanak egy OpenNebula [5] telepítést DAS-on [6], a holland nemzeti számító klaszteren. Az OpenNebula egy nyílt forráskódú megoldás az infrastruktúra szolgáltatásra: a fizikai erőforrásokat virtuálisként kezeli és osztja ki. Itt a hallgatóknak felső korlát van a virtuális gépek számára, amelyeket párhuzamosan lehet használni.

A kereskedelmi felhő alapú rendszerként az Amazon EC2 [7] áll rendelkezésre. Itt a hallgatóknak bármilyen laborral kapcsolatos erőforrásra van kvótájuk. A labormunka értékelésénél figyelembe vesszük a kvóta túllépését.

Az ütemezőnek egy nagymértékben párhuzamosított alkalmazás: Twitter üzenetek szövegének feldolgozását kell elvégezni hangulatelemzés céljából.

A labormunka kimenetele általában az volt, hogy a hallgatók képesek voltak megérteni a *best-effort* és kereskedelmi virtuális erőforráskezelés közti különbségeket.

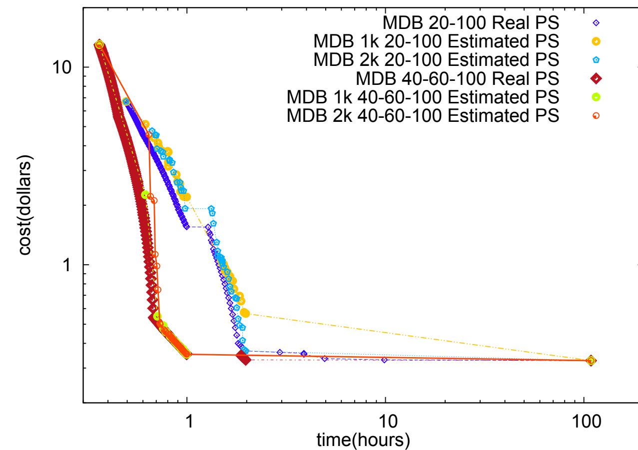


Fig. 4. Egy alkalmazás Pareto-határa folyamatok futásidejének és két különböző erőforrás kombinált eloszlásával.

Általában jövedelmező ütemezőket fejlesztettek, míg a legjobban teljesítő hallgatók kifinomultabb ütemezőket hoztak létre, ahol a felhasználók több irányelv alapján választhattak: gyorsaság, költséghatékonyság és jövedelmezőség.

4 Eredmények és továbbfejlesztési lehetőségek

A felhasználóközpontú felhő alapú ütemezési megközelítések nagyon ígéretesnek bizonyultak. Az oktatásba való minél hamarabbi bevezetésük rendkívül fontos, ha stabil állapotba jut.

Továbbfejlesztési lehetőségként szeretnénk támogatni a Haskell AWS lambdafüggvények szállítását a Haskell AWS API implementáción keresztül.

Köszönetnyilvánítás

Ez a cikk az Erasmus+ Kulcsfontosságú Akciók 2 projektcsalád `2017-1-SK01-KA203-035402` számú projektjének – Stratégiai Partnerség a Felsőoktatásért, ,,*A működő szoftverek összeépíthetőségének, megérthetőségének és helyességének oktatását elősegítését célzó” 3COWS* (*“Focusing Education on **C**omposability, **C**omprehensibility and **C**orrectness of Working Software” (3COWS)*) projekt eredményeként valósult meg.

Felelősség kizárása

E kiadványban közölt tudás, információk és állítások a szerző(k) véleménye(i), meggyőződése(i) és világnézete(i) alapján kerültek megfogalmazásra, eképp nem feltétlenül tükrözik az Európai Unió hivatalos álláspontját. Sem az Európai Unió intézményei és testületei, sem bármely Uniós intézmények és testületek megbízatásából eljáró személy nem tehető felelőssé a kiadványban szereplő megállapítások felhasználásáért, vagy a felhasználásból eredő bármilyen következményért.

References

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://opennebula.org/`, Last accessed: 15-11-2018.
6. `https://www.cs.vu.nl/das5/`, Last accessed: 15-11-2018.
7. `https://console.aws.amazon.com/ec2/v2/home`, Last accessed: 15-11-2018.
8. `https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/instance-types.html`, Last accessed: 15-11-2018.
9. A.-M. Oprescu; T. Kielmann; H. Leahu, *Budget estimation and control for bag-of-tasks scheduling in clouds*, 2011, Parallel Processing Letters, vol. 21.
10. A.-M. Oprescu; T. Kielmann; H. Leahu, *Stochastic tail-phase optimization for bag-of-tasks execution in clouds*, 2012, Proceedings of the 2012 IEEE/ACM Fifth International Conference on Utility and Cloud Computing.
11. A.-M. Oprescu; T. Kielmann; *Bag-of-tasks scheduling under budget constraints*,2010, IEEE Second International Conference on Cloud Computing Technology and Science (CloudCom).
12. A. Vintila; A.-M. Oprescu; T. Kielmann; *Fast (re-) configuration of mixed on-demand and spot instance pools for high-throughput computing*, 2013, Proceedings of the first ACM workshop on Optimization techniques for resources management in clouds.

Az energiahatékonyság szoftvertesztelés során történő mérése az oktatásban

Csaba Szabó[0000−0001−5147−2452]

Department of Computers and Informatics
Faculty of Electrical Engineering and Informatics, Technical University of Košice,
Letná 9, 04200 Košice, Slovakia
csaba.szabo@tuke.sk

Abstract. A szoftverfejlesztés oktatók célja felkészíteni a jövő szoftverfejlesztőit, hogy azok a szoftverfejlesztés teljes életciklusában felmerülő valamennyi problémát kezelni tudják. A megrendelők igényeinek megértésével és a megfelelő szoftver felépítésével kapcsolatos képességek mellett jelentős szerepet játszik az eredmények helyességének ellenőrzésében való jártasság is. Ebben a cikkben az utóbbiakra koncentrálunk. A tesztelést vizsgáljuk, ahol az automatizáció szintje kevésbé fontos. Ezzel szemben kihangsúlyozzuk a tesztelés méréssel kapcsolatos természetét, pontosabban a szoftver energiafogyasztásának mérését, amit a tesztelés különböző szintjein lehet végezni. Mindezeket a szempontokat az oktató szempontjából mutatjuk be, leírva, hogy miként lehet egy olyan szoftvertechnológiai labor kurzust felépíteni, ahol a szoftvertesztelés energiahatékonyságának mérése kerül előtérbe a szerzők megjegyzéseivel.

Keywords: Energy consumption · Measurement · Software engineering education · Testing.

1 A probléma meghatározása

Az akkumulátor gyártók körében az egyik fő kihívás olyan akkumulátorokat gyártani, amelyek egy feltöltéssel minél hosszabban üzemelnek. Persze több más kihívással is szembe kell nézni, mint a méretezés – ami nagyban befolyásolja az eszköz formáját –, vagy az akkumulátorok másik fontos tulajdonsága, a súly csökkentése. Az akkumulátornak valamelyest könnyebbnek kell lennie, mint az eszköz, amit üzemeltet. A kihívást itt az jelenti, hogy miként lehet az akkumulátor méretének és súlyának csökkentése mellett nagyobb teljesítményt és hosszabb üzemidőt elérni a mobil eszköz töltése nélkül.

A hardveres kihívásokon felül szoftveres szempontok is léteznek, nevezetesen a szoftvernek is támogatnia kell az energiatakarékosság kritériumait. Ennek elérése a felhasználói élmény korlátozása nélkül manapság egy hallgatólagos, de fontos célja minden hordozható eszközre történő szoftverfejlesztésnek.

Tekintve, hogy a mobil eszközök energiafogyasztására hatással van sok minden a futó alkalmazásoktól kezdve az alapszolgáltatások hozzáférési szintjein keresztül a felhasználó hangulatáig, az ilyen eszközökre történő fejlesztés már önmagában kihívást jelent [1]. Azt mondhatnánk, hogy a szoftverfejlesztés kihívása mindig ugyanaz, de a "mobilitást" mint kulcsfontosságú rendszertulajdonságot ki kell emelnünk. Az akkumulátor töltöttségi szintje a rendszer teljesítményét is meghatározza az "energiatakarékossági beállítások" néven ismert operációs rendszer konfigurációjának függvényében.

További nehézséget jelent, ha egy oktatónak ezekre a kihívásokra is fel kell készítenie a hallgatókat. A közismert "ökölszabályokat" és "energiatakarékossági tippeket" a hallgatók számára könnyen érthető formában kell prezentálni. Ezt úgy lehet legegyszerűbben megtenni, ha a fogalmakat ismerős környezetben, esetünkben a szoftvertesztelésbe és teszt automatizálásban tárgyaljuk [2]. A cikk további szakaszaiban ezt tűzzök ki célul, kezdve az ajánlással, majd annak kiértékelésével, végül továbbfejlesztési lehetőségek ismertetésével.

2 Megoldási javaslat

Ahogy a fentiekben állítottuk, meg kell találnunk a legmegfelelőbb környezetet az energiafogyasztás mérésére [3] és az energiahatékonyság kiértékelésére. Ez a környezet lehet a szoftverfejlesztés kezdeti vagy a későbbi szakasza, minthogy a szoftver általános életciklusának mindkét fejlesztési fázisa lehetőséget biztosít a készülő termék mérésére [4].

A szoftverfejlesztés kezdeti fázisának választása azzal az előnnyel jár, hogy az energiatakarékossági döntésekben már az elején minden tevékenység kellő figyelmet kaphat, míg a szoftverfejlődés későbbi szakaszaiban lehetőség adódik a termék implementációjában elvégzett javítások kiértékelésére. Másfelől a későbbi fejlesztési szakaszban történő mérésekhez működő szoftverre van szükség, a kezdeti fázisban pedig még csak a szoftverre vonatkozó első követelmények teljesülnek.

A két lehetséges megközelítést oktatási környezetben tekintve az a legjobb, ha mindkettőt alkalmazzuk: az első szemeszterben a szoftver kezdeti megvalóstása, egy másikban pedig ugyanazon szoftver továbbfejlesztése kerül sorra.

Egy oktatónak általában nem áll rendelkezésre két félév az előző szakaszban ismeretett kurzusok végigvitelére. Éppen ezért választani kell, hogy melyik fejlesztési módszert alkalmazza a jelölt gyakorlatok bemutatására. A fejlesztési folyamatok felépítésének szempontjából és a kezdeti fejlesztés esetleges evolúciós jellege miatt az utóbbit választjuk, mivel az tartalmaz több, a fejlesztés kezdeti fázisára jellemző tevékenységet is (leszámíta a korai követelménygyűjtést és elemzést), illetve kihangsúlyozza a tesztelés és kiértékelés fontosságát.

Ez az opció lehetővé teszi az oktató számára,

- hogy a hallgatók visszatekinthessenek korábbi fejlesztéseikre, hogy kritikusak lehessenek magukkal,
- hogy kódmetrikák és energiafogyasztási mérésekkel (vagy becslésekkel) kiértékeljék az eredményeiket,
- illetve hogy a fenti tevékenységeket beépíthessék az alapvető verifikációs és validációs folyamatokban a szoftver fejlődése során.

Mivel a fejlesztés programozási nyelve nem fontos, ezért a hallgató tetszőleges korábbi projektjét kiválaszthatja, vagy akár az összeset is, ha versengeni szeretnének a projektek vagy a felhasznált programozási nyelvek számában. Azonban a programozási nyelv meghatározhatja vagy korlátozhatja a fejlesztői környezet és egyéb eszközök használatát. Ezen eszközök és bővítményeik kiválasztása segítséget nyújthat a kódmetrikák kiértékelésében.

Az energiafogyasztás és energiahatékonysági mérések általában különböző eszközöket köivetelnek, mivel manapság csak kevés energiafogyasztást mérő és becslő fejlesztői környezet van.

Ha a tesztelést vesszük a mérések alapjául, akkor meg kell jegyeznünk, hogy a statikus kódanalízis szintén a szoftvertesztelés része. Mindemellett az alkalmazás egyes részeit fehér doboz tesztelés részeként (főként egységtesztekben) futtatva, azt némi energiafogyasztásra vonatkozó méréssel kiegészítve a fogyasztás prezentálható. Ez a tesztelt kód energiafogyasztásának egy indirekt nézete. A fekete doboz tesztelés során a teljes alkalmazást teszteljük teszt forgatókönyvek felhasználásával. Ezek leginkább a szoftver mindennapos használati eseteit fedik le, de egyesek a szélsőséges eseteket is tesztelik, beleértve a felhasználó rossz hangulatában történő használatot. A fekete doboz tesztek energiafogyasztásának mérése tehát a fogyasztás egy hozzávetőleges, de direkt nézetét adja.

A kezdeti szoftverfejlesztéssel szemben a későbbi szoftverfejlődés vizsgálatának fő előnye, hogy az oktató előkészítheti a kezdeti verziót a későbbi evolúciós lépésekre, beleértve az ismert hibák listáját és a tesztbázist. Az újratesztelés és regressziós tesztek megléte itt rendkívül fontos, hiszen ez nagyban segíti a továbbfejlesztés hatékonyságát.

A fentiek teljesülése esetén az energiafogyasztás mérése a felhasználó szemszögéből az alábbi lépésekben történik:

1. Ki kell választani egy terméket a tárolóból, amely a múltbeli projekt lesz.
2. Ezt ki kell értékelni statikus kódanalízissel, teszteléssel, energiamérésekkel, használhatósági felmérésekkel, stb.
3. Tovább kell fejleszteni (többféle változás lehetséges, mint egy új funkcionalitás hozzáadása, módosítása, javítása vagy adaptálása).
4. Újra kell tesztelni, hogy megbizonyosodjunk a hiányosságok vagy hibák megszüntetéséről.
5. Regressziós tesztelés (beleértve az újraértékelést)
6. Konklúzió levonása.

3 Eredmények

Az energiafogyasztás mérése viszonylag újszerű más technikákhoz képest, mint a statikus kódanalízis, fekete/fehér doboz tesztelés és hibakeresés. De ha kombináljuk ezekkel a régebbi technikákkal, akkor a hallgatók összesített pontszámában a kritikus tulajdonság sokkal kisebb.

Az értékelési során többféle "szabadságfok" elképzelhető, amelyek az osztályzat kialakításánál együtt és külön is számbavehetők:

1. különböző projektek száma,
2. felahasznált programozási nyelvek száma,
3. a végleges termék kódminősége,
4. a végleges termék energiahatékonysága,
5. a minőségjavulás mértéke a fejlesztés során,
6. energiafogyasztás hatékonyságának javítása a fejlesztés során.

A fenti szempontok figyelembevételével a nagyobb versenyszellemmel rendelkező hallgatók számára többféle versenyt lehet felállítani, ahol a teljesítők győzelmi pontokkal gyarapodnak az egyes projektekben. A maximalisták feladata optimalizálni a kód metrikákat és minimalizálni az energiafogyasztást. Az átlagos hallgatók számára az energiafogyasztás javítása és az olvasható kód készítése lehet kitűzött cél.

A versenyeket tovább lehet támogatni azzal, ha a hallgatóknak nem a saját korábbi tárolóikról, hanem egy kijelölt tárolóból kell projektet választaniuk. A számítógépes játékokhoz hasonlóan így minden játékszint mindenki számára egységesen elérhető. Az egyenlőség megteremtéséhez közös diák és tanár fórum indítása kezdeményezhető.

Továbbfejlesztési lehetőség ezen a területen egy szállítható integrált környezet készítése a tesztelésre és az energiafogyasztás becslésére. Ezt a környezetet a szoftver evolóciót vagy kezdeti fejlesztést végző tárgyak keretében lehet használni a tesztek energiafogyasztási mérésének támogatására az oktatásban. A hallgatók kreativitását korlátozhatja ha egy félig zárt környezetet biztosítunk, hiszen a hardver fontos szerepet játszik az energiafogyasztási mérésekben. Vannak kutatások, amelyek ezen korlátozások feloldására irányulnak.

Köszönetnyilvánítás

Ez a cikk az Erasmus+ Kulcsfontosságú Akciók 2 projektcsalád `2017-1-SK01-KA203-035402` számú projektjének – Stratégiai Partnerség a Felsőoktatásért, *„A működő szoftverek összeépíthetőségének, megérthetőségének és helyességének oktatását elősegítését célzó" 3COWS* (*"Focusing Education on **C**omposability, **C**omprehensibility and **C**orrectness of Working Software" (3COWS)*) projekt eredményeként valósult meg.

Felelősség kizárása

E kiadványban közölt tudás, információk és állítások a szerző(k) véleménye(i), meggyőződése(i) és világnézete(i) alapján kerültek megfogalmazásra, ekképp nem feltétlenül tükrözik az Európai Unió hivatalos álláspontját. Sem az Európai Unió intézményei és testületei, sem bármely Uniós intézmények és testületek megbízatásából eljáró személy nem tehető felelőssé a kiadványban szereplő megállapítások felhasználásáért, vagy a felhasználásból eredő bármilyen következményért.

References

1. J. Saraiva, M. Couto, Cs. Szabó, D. Novák: Towards Energy-Aware Coding Practices for Android, Acta Electrotechnica et Informatica, Vol. 18, No. 1, 2018, pp. 19–25. https://doi.org/10.15546/aeei-2018-0003
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. Porkoláb, 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 Applications, 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. Cs. Szabó, 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.

A környezetbarát szoftvermérnöki megközelítés *

João Paulo Fernandes[1] and João Saraiva[2]

[1] CISUC & Universidade de Coimbra, Portugal
[2] HASLab/INESC TEC & Depart. de Informática, Univ. do Minho, Portugal
jpf@dei.uc.pt , saraiva@di.uminho.pt

Abstract. Ez a technikai riport a Coimbra és Minho Egyetemek Zöld Szoftver Labor kutatásait írja le, amelyek az első tanár képző találkozón lettek bemutatva az Erasmus+ projekt keretében, "Működő Szoftverek Alakíthatóságának, Érthetőségének és Helyességének Vizsgálata az Oktatásban" címmel. A riport szemlélteti mind a programozási nyelvek és adatszerkezetek *zöld besorolását*, mind pedig a szoftver rendszerek rendellenes energiafelhasználását.

Keywords: Green Computing, Energy-aware Software, Source Code Analysis

1 Motiváció

A vezeték nélküli, hatékony eszközök, mint az okostelefonok, laptopok, stb. jelenleg elterjedt használata mind a számítógépgyártók, mind pedig a szoftverfejlesztők munkájára hatással van. A számítógép/szoftver futásideje, ami elsődleges szempont volt a múlt században, már nem az egyetlen szempont. Az energiafogyasztás egyre inkább szűk keresztmetszet a hardver- és szoftverrendszereknél. Következésképpen a zöld szoftver releváns és aktív kutatási terület.

Ebben a riportban röviden leírjuk a zöld szoftverrel kapcsolatos kutatásokat a *Green Software Laboratory* (GSL) keretein belül. A GSL több portugál kutatócsoportot magába foglal, beleértve a "Működő Szoftverek Alakíthatóságának, Érthetőségének és Helyességének Vizsgálata az Oktatásban" projekt két telephelyét. A GSL egy kezdeményezés, ami olyan technikák és eszközök kifejlesztését tűzte ki célul, melyek különféle számítási rendszerek (mobil, program, adatbázis, stb.) energiafogyasztásának csökkentését teszik lehetővé. A GSL kifejezetten a szoftver részre fókuszál, ahol forráskódanalízist és transzformációs technikákat alkalmaz energiafogyasztási anomáliák felderítésére, illetve optimalizációkat határoz meg ezek csökkentésére.

* Ez a munka a ERDF – European Regional Development Fund through the Operational Programme for Competitiveness and Internationalisation - COMPETE 2020 Programme within project POCI-01-0145-FEDER-006961, és a National Funds through the Portuguese funding agency, FCT - Fundação para a Ciencia e a Tecnologia támogatásával készült a POCI-01-0145-FEDER-016718 and UID/EEA/50014/2013. projekt kereteiben

A múlt században egy szoftverrendszer hatékonyságát főként a futásidő és memória felhasználás hatékonyságára értették. Manapság a fejlesztők gyakran felteszik a kérdést, hogy egy gyorsabb program egyben zöldebb-e? Egy szoftverrendszer energiafogyasztásának hatékonyságát több tényező befolyásolja: a programozási nyelv és annak végrehajtási modellje (bináris kódra vagy virtuális gépre való fordítás, interpretált kód, lusta vs. mohó kiértékelés, futásidejű parciális kiértékelés, stb.). A memóriamodell hatékonysága és a nyelvi könyvtárak szintén befolyásolhatják a hatékonyságot. Az algoritmus összetettségével megoldja a számítógépes problémát, de a teljesítményre is hatással van: ha az algoritmusnak több munkát kell végeznie, mint amennyi feltétlenül szükséges, akkor több CPU-t és energiát használ.

Ebben a cikkben röviden beszámolunk a GSL-beli kutatás eredményeiről, nevezetesen a programozási nyelvek energiafelhasználásának elemzéséről (2. fejezet), az adatszerkezet könyvtárakról (3. fejezet) és a szoftver forráskódjáról (4. fejezet).

2 Zöld programozási nyelvek

A programozási nyelvek kapcsán felmerül egy érdekes kérdés az energiafelhasználással kapcsolatban: vajon egy gyorsabb nyelv egyben energiatakarékosabb-e vagy sem? A programozási nyelvek összehasonlítása e tekintetben egy rendkívül összetett feladat, hiszen a nyelv teljesítményére hatással van a fordítóprogram minősége, a virtuális gép, a szemétgyűjtő, stb.

A GSL-ben vizsgáltuk, kiértékeltük és összehasonlítottuk a 27 legelterjedtebb programozási nyelv teljesítményét. Ehhez két különböző feladatbankot használtunk: *Computer Language Benchmark Game* (CLBG)[3] és *Rosetta Code*[4] [1–3]. Mindkét feladatbank tartalmaz megoldott feladatokat számos programozási nyelven. Míg a CLBG-t programozási nyelvek futásidejű hatékonyságának elemzésére alkották, a *Rosetta Code*-ot inkább kódmegértési céllal készítették.

Ezeket a programokat lefordítottuk és lefuttattuk a legkorszerűbb fordítókkal, virtuális gépekkel, interpreterekkel és könyvtárcsomagokkal minden nyelven. Ezután monitoroztuk a futásidőt, a kiugró és teljes memóriafelhasználást és a CPU/-DRAM/GPU energiafelhasználását. Előállítottunk egy energia rangsort a 27 programozási nyelvből, majd elemeztük az eredményeket a végrehajtás típusa (fordított, virtuális gépen futó, interpretált) és a nyelv paradigmája (imperatív, funkcionális, objektumorientált, szkriptnyelv) szerint. Minden végrehajtási típusra és programozási paradigmára külön lefuttattunk egy nyelvi rangsorolást a vizsgált szempontok (pl. futásidő vagy energiafogyasztás) alapján. Az első kísérleteink a várt eredményeket adták: a C nyelv gyorsabb és zöldebb volt, de a lassabb nyelvek között is találtunk energiatakarékosabbat a többinél [2, 3].

[3] http://benchmarksgame.alioth.debian.org/
[4] http://www.rosetta.org

3 Zöld adatszerkezetek

A programozási nyelv/paradigma és a hatékony fordítóprogram optimalizációk nem az egyetlen szempont, ami egy szoftverrendszer energiafogyasztására hatással van. Egy program valójában hatékonyabbá válhat "csupán" a könyvtárcsomagjaik optimalizálásával [4, 5]. A legtöbb nyelv hatékony könyvtárcsomagot biztosít az adatszerkezetek kezelésére. A GSL-ben két fejlett adatszerkezet energiafogyasztását tanulmányoztuk, amelyeket széleskörben alkalmaznak a Java és Haskell programozási nyelvekben.

Java-ban részletes tanulmányt készítettünk a *Java Collections Framework* (JCF) könyvtár energiafogyasztásával kapcsolatban [5]. Az adatszerkezetek három hagyományos csoportját, nevezetesen a halmazokat, listákat és leképezéseket tekintettük, és mindegyikre megvizsgáltuk a különböző implementációik energiafogyasztását [4]. Ez az energiatudatosság a JCF használatában nemcsak a fejlesztők ösztönzésére, hanem örökölt Java kódok optimalizálására is alkalmas. Készítettünk egy Java adatszerkezet refaktoráló eszközt *JStanley* néven, ami Java forráskódokat refaktorál, ha egy zöldebb tároló is elérhető [6]. Végrehajtottunk egy kezdeti kiértékelést 7 publikusan elérhető Java projektre, ahol az energiafogyasztásban 2% és 17% közötti javulást sikerült elérni.

Haskellben az *Edison*[6] energiafogyasztását vizsgáltuk, ami egy fejlett és jól dokumentált könyvtárcsomat tisztán funkcionális adatszerkezetek kezelésére [7]. Az Edison különböző funkcionális adatszerkezeteket biztosít a háromféle absztrakcióra: *sorozatok* (listák, sorok és vermek), *tárolók* (halmazok és kupacok) valamint asszociatív tárolók (leképezések és véges relációk). 16 ilyen adatszerkezet implementáció energia és futásidő metrikáit elemeztünk [5]. Továbbá megvizsgáltuk a különféle fordítási optimalizációk hatását. Azt az eredményt kaptuk, hogy az energiafogyasztás pontosan arányos a futásidővel, és hogy a DRAM energiafogyasztása a teljes fogyasztás 15 és 31 közötti százalékát adja. Végül megfigyeltük azt is, hogy az optimalizációk pozitív és negatív hatással is lehettek az energiafogyasztásra.

4 Zöld forráskód

Nemcsak a nyelvek és az adatszerkezeteket tároló könyvtárcsomagok befolyásolják az energiafogyasztást, hanem az algoritmusok és programozási gyakorlat is kulcsszerepet játszik a programok hatékonyságában. A GSL-ben ismert hibakereső technikákat adaptáltunk az "energiaszivárgás" statikus lokalizálására [8–11]. A SPELL (SPectrum-based Energy Leak Localization) bevezetésével meghatároztuk a szoftver piros területeit. Az első kísérleti tanulmány azt mutatta, hogy a tapasztalt programozók a SPELL találatai alapján 15 és 74 százalék közti mértékben tudták tovább optimalizálni a programok energiafogyasztását, szemben az olyanokkal, akiknek semmilyen információjuk nem volt, vagy csak hagyományos, futásidejű profilozókat használtak. Ezen túl C/C++ programok viselkedését is vizsgáltuk energiafogyasztás terén [12].

A vezeték nélküli és IoT eszközök megjelenésével változik a programozók szoftverfejlesztése is. A szoftvereknek ugyanis különféle mobil eszközökön kell működniük, és ez a szoftverfejlesztésben egy fő szempont. Szoftver termékcsaládok (SPL) mint mérnöki tudományágak jelentek meg, hogy lehetővé tegyék közös funkcionalitások megosztását. A GSL-ben feltételes fordításon alapuló statikus analítis technikákat határoztunk meg az SPL-ek energiafogyasztásának megfigyelésére [13].

Az Android egy széleskörben elterjedt környezet vezeték nélküli eszközökhöz, ahol a szoftver energiaelemzés és optimalizáció egy aktívan kutatott terület. A GSL csapata több technikát és eszközt fejlesztett ki [14, 15] Androidos alkalmazások forráskódjainak elemzésére és energiafogyasztásának optimalizálására [16, 17].

Manapság a legtöbb mobil eszközön tárolt adat (fájlok, fotók, videók) az eszköz operációs rendszerének ökoszisztémájában biztosított felhőben is tárolódik. Ezek a felhőalapú rendszerek olyan adatközpontok, amelyek naponta futtatnak nagymértékű adatlekérdező folyamatokat olyan kifinomult adatbáziskezelő rendszerekkel monitorozva és irányítva, amik a lekérdezéseket támogató folyamatokért felelősek. Az adatbáziskezelő rendszerek általában a válaszidőt optimalizáló tervezeteken alapszanak. A munkánk során létrehoztunk és kifejlesztettünk egy alternatív megoldást az adatbázis lekérdezés energiafogyasztási tervezetének meghatározására [18, 19]. Az első tapasztalati eredményeink azt mutatják, hogy az optimalizációs heurisztikák használata jelentős nyereséget jelent a lekérdezések energiafogyasztásában és futásidejében.

5 Eredmények

Ebben a cikkben bemutattuk a GSL-ben folytatott kutatásokat, nevezetesen a programozási nyelvek és adatszerkezetek zöld rangsorolását, a szoftverrendszerek energiaveszteségét detektáló technikákat a forráskódban, illetve egy energiatudatos végrehajtási terved adatbázisrendszerekben.

Köszönetnyilvánítás

Ez a cikk az Erasmus+ Kulcsfontosságú Akciók 2 projektcsalád `2017-1-SK01-KA203-035402` számú projektjének – Stratégiai Partnerség a Felsőoktatásért, *„A működő szoftverek összeépíthetőségének, megérthetőségének és helyességének oktatását elősegítését célzó" 3COWS* (*"Focusing Education on* ***C****omposability,* ***C****omprehensibility and* ***C****orrectness of Working Software" (3COWS)*) projekt eredményeként valósult meg.

[5] `docs.oracle.com/javase/7/docs/technotes/guides/collections/index.html`
[6] `hackage.haskell.org/package/EdisonAPI-1.3/docs/Data-Edison.html`

Felelősség kizárása

E kiadványban közölt tudás, információk és állítások a szerző(k) véleménye(i), meggyőződése(i) és világnézete(i) alapján kerültek megfogalmazásra, eképp nem feltétlenül tükrözik az Európai Unió hivatalos álláspontját. Sem az Európai Unió intézményei és testületei, sem bármely Uniós intézmények és testületek megbízatásából eljáró személy nem tehető felelőssé a kiadványban szereplő megállapítások felhasználásáért, vagy a felhasználásból eredő bármilyen következményért.

References

1. Couto, M., Pereira, R., Ribeiro, F., Rua, R., Saraiva, J.: Towards a green ranking for programming languages. In: Proceedings of the 21st Brazilian Symposium on Programming Languages. SBLP (2017) 7:1–7:8 (best paper award).
2. Pereira, R., Couto, M., Ribeiro, F., Rua, R., Cunha, J., Fernandes, J.P., Saraiva, J.: Energy efficiency across programming languages: How do energy, time, and memory relate? In: Proc. of the 10th ACM SIGPLAN Int. Conference on Software Language Engineering. SLE 2017, New York, NY, USA, ACM (2017) 256–267
3. Pereira, R., Couto, M., Ribeiro, F., Rua, R., Cunha, J., Fernandes, J.P., Saraiva, J.: Ranking programming languages by energy efficiency. Science of Computer Programming (2018) Submitted.
4. Pereira, R., Couto, M., Saraiva, J., Cunha, J., Fernandes, J.P.: The Influence of the Java Collection Framework on Overall Energy Consumption. In: 5th Int. Workshop on Green and Sustainable Software. GREENS '16, ACM (2016) 15–21
5. Melfe, G., Fonseca, A., Fernandes, J.P.: Helping developers write energy efficient haskell through a data-structure evaluation. In: Proceedings of the 6th International Workshop on Green and Sustainable Software. GREENS '18, New York, NY, USA, ACM (2018) 9–15
6. Pereira, R., Simão, P., Cunha, J., Saraiva, J.: jStanley: Placing a Green Thumb on Java Collections. In: 33rd ACM/IEEE International Conference on Automated Software Engineering. ASE 2018, New York, NY, USA, ACM (2018) 856–859
7. Lima, L.G., Melfe, G., Soares-Neto, F., Lieuthier, P., Fernandes, J.P., Castor, F.: Haskell in Green Land: Analyzing the Energy Behavior of a Purely Functional Language. In: Proc. of the 23rd IEEE Int. Conf. on Software Analysis, Evolution, and Reengineering (SANER'2016), IEEE (2016) 517–528
8. Pereira, R., Carção, T., Couto, M., Cunha, J., Fernandes, J.P., Saraiva, J.: Helping programmers improve the energy efficiency of source code. In: Proc. of the 39th Int. Conf. on Soft. Eng. Companion, ACM (2017)
9. Pereira, R.: Locating energy hotspots in source code. In: Proceedings of the 39th International Conference on Software Engineering Companion. ICSE-C '17, Piscataway, NJ, USA, IEEE Press (2017) 88–90 (ACM SRC silver award).
10. Pereira, R.: Energyware Engineering: Techniques and Tools for Green Software Development. PhD thesis, Depart. de Informática, Universidade do Minho (2018)
11. Pereira, R., Carção, T., Couto, M., Cunha, J., Fernandes, J.P., Saraiva, J.: Spelling out energy leaks: Aiding developers locate energy inefficient code. (2018) (submitted).
12. Santos, M., Saraiva, J., Porkoláb, Z., Krupp, D.: Energy consumption measurement of c/c++ programs using clang tooling. SQAMIA'17 - CEUR Workshop Proceedings **1938** (2017)
13. Couto, M., Borba, P., Cunha, J., Fernandes, J.P., Pereira, R., Saraiva, J.: Products go green: Worst-case energy consumption in software product lines. In: Proceedings of the 21st International Systems and Software Product Line Conference - Volume A. SPLC '17, ACM (2017) 84–93
14. Couto, M., Carção, T., Cunha, J., Fernandes, J.P., Saraiva, J.: Detecting anomalous energy consumption in android applications. In Quintão Pereira, F.M., ed.: Programming Languages: 18th Brazilian Symposium, SBLP 2014, Maceio, Brazil, October 2-3, 2014. Proceedings. (2014) 77–91
15. Cruz, L., Abreu, R.: Performance-based guidelines for energy efficient mobile applications. In: 4th International Conference on Mobile Software Engineering and Systems. MOBILESoft '17, Piscataway, NJ, USA, IEEE Press (2017) 46–57
16. Couto, M., Cunha, J., Fernandes, J.P., Pereira, R., Saraiva, J.: Greendroid: A tool for analysing power consumption in the android ecosystem. In: 2015 IEEE 13th International Scientific Conference on Informatics. (Nov 2015) 73–78
17. Cruz, L., Abreu, R., Rouvignac, J.N.: Leafactor: Improving energy efficiency of android apps via automatic refactoring. In: IEEE/ACM International Conference on Mobile Software Engineering and Systems, MobileSoft 2017. (2017)
18. Gonçalves, R., Saraiva, J., Belo, O.: Defining energy consumption plans for data querying processes. In: 2014 IEEE International Conference on Big Data and Cloud Computing (BdCloud)(BDCLOUD). Volume 00. (Dec. 2015) 641–647
19. Belo, O., Gonçalves, R., Saraiva, J.: Establishing energy consumption plans for green star-queries in data warehousing systems. In: 2015 IEEE International Conference on Data Science and Data Intensive Systems. (Dec 2015) 226–231

A feladatorientált programozás oktatása

Pieter Koopman and Rinus Plasmeijer

Institute for Computing and Information Sciences
Radboud University Nijmegen, The Netherlands
{pieter,rinus}@cs.ru.nl

Kivonat A Composability, Comprehensibility, Correctness téli iskolában két továbbképzést tartunk a feladatorientált programozásról és az erre a paradigmára épülő konkrét rendszerekről. Az iTask rendszer egy webalapú interfészt kínál emberei felhasználásra, hogy láthassák és megoszthassák feladataikat és előrehaladásukat. Az mTask rendszer ugyanazokat a koncepciókat alkalmazza a mikroprocesszorok által elvégzett feladatokra. Ebben az útmutatóban igazoljuk azokat a döntéseket, amelyeket az amszterdami oktatóképzésen hoztunk arról, hogy milyen témák és hogyan lesznek prezentálva a téli iskolában. A korlátozott mennyiségű időre és a résztvevők változatos hátterére való tekintettel a paradigma gyakorlati alkalmazásaira fogunk koncentrálni. Nagyon kevés idő van ezen rendszerek felépítésének kihívásait és szépségeit ismertetni.

Keywords: Task Oriented Programming · Domain Specific Language · Microprocessor · Code Generation · Simulation · Generic Programming · Monad

1 Bevezetés

A feldatorientált programozás (Task Oriented Programming, TOP) egy programozási mód, amely az emberek és gépek által elvégzett feladatok körül összpontosul. E feladatokat a funkcionális programozási nyelvekben egyszerű függvények specifikálják. Minden példát Clean [6] nyelven demonstráljuk. A feladatok szemantikája meglehetősen különbözik az egyszerű függvény-kiértékeléstől. Egy feladatot újra és újra kiértékelünk, amíg nem állít elő egy stabil értéket, vagy amíg már nincs szükség az eredményére. A közbenső feladatok eredményei megfigyelhetőek más feladatok számára. A feladatokat feladatkombinátorok segítségével állíthatjuk elő.

A kosicei Composability, Comprehensibility, Correctness téli iskolában két kurzust fogunk tartani a TOP-ról, ezek címe *Why "Task Oriented Programming" matters* (Miért fontos a feladatorientált programozás) és *Functional Programming of Devices* (Eszközök funkcionális programozása). Mindekét kurzus egy előadásból és egy gyakorlatból fog állni. A cikkben kifejtjük a kurzusok tartalmáról és szervezéséről az oktatóképzésen hozott döntéseket.

2 Résztvevők

A Composability, Comprehensibility, Correctness téli iskolát BSc, MSc, illetve PhD hallgatók, valamint oktatóik számára szervezzük. Az oktatóképzés megbeszélései rávilágítottak, hogy a résztvevők változó tapasztalatokkal rendelkeznek a funkcionális programozás terén, mind a tapasztalat mennyiségét, mind a nyelvismeretet illetően. A használt nyelvek a tisztán funkcionális nyelvektől, mint a Haskell és a Clean, a lusta nyelvekig terjednek, mint a Erlang, a Scheme és a Scala.

Ebből következik, hogy nem építhetünk komolyabb közös funkcionális programozással kapcsolatos tapasztalatokra. A résztvevőknek csak egy részének lesznek ismerősök olyan koncepcik, mint az erős típusosság, a magasabbrendű függvények, a típuskonstruktor osztályok, a monádok és a generikusok. Bár ezek a témák a TOP építőkövei, nem feltételezhetjük, hogy az összes résztvevő által ismertek lesznek.

3 Task Oriented Programming

A feladatorientált programozás kisszámú domainspecifikus, egyszerű feladatra épül, amelyek jellemzően a környezetükkel állnak interakcióban, pl. a feladatrészeket végrehajtó emberekkel vagy a fizikai világgal kommunikáló hardverekkel. Feladatokat feladatkombinátorok segítségével hozhatunk létre kisebb feladatokból.

A feladatok az eredményeiken, valamint megosztott adatforrásokon (Shared Data Sources, SDSs) keresztül képesek kommunikálni. Egy SDS típusozott adatokat tartamaz, amelyek olyan eljárásokon keresztül hozzáférhetőek, mint a `get` és a `set`. Ezek az egyszerű eljárások a feladat állapotának megfelelően viselkednek, hogy biztosítsák a hivatkozási helyfüggetlenséget.

Ahhoz, hogy egy létező nyelv adattípusai és műveletei újrafelhasználhatóak legyenek, egy TOP rendszer gyakran domainspecifikus nyelvként (Domain Specific Language, DSL) kerül megvalósításra úgy, hogy egy létező (funkcionális) nyelvbe beágyazzuk.

4 Az iTask rendszer

A TOP [5] első implementációja az iTask rendszer volt, amely egy, a Clean funkcionális nyelvbe ágyazott DSL. Az iTask rendszer megkönnyíti a fejlesztővel való interakciót a weblapok típusvezérelt generálásán keresztül. Ezen oldalak létező böngészőkben kerülnek megjelenítésre. Az oldalak az aktuális feladatokról biztosítanak információkat a felhasználó számára. A felhasználó az iTask rendszerrel űrlapok kitöltésén és gombok nyomásán keresztül tudnak kommunikálni.

4.1 Felhasznált technológiák

Az iTask rendszer egy State monádhoz nagyon hasonló állapotot ad körbe. A `return`, bind ($\gg=$), és sequence (`>>|`) műveleteknek megfelelő operátorok nagyon

hasonlóak a monadikus megfelelőikhez [4, 7]. Ehhez magasabbrendű függvényekre és a felhasználó által definiált infix operátorokra van szükség. Az operátorszimbólum különböző monádoknál történő újrafelhasználása érdekében ezeket típuskonstruktor osztályokként definiáljuk.

A feladatkombinátorok mind magasabbrendű függvények, tipikusan felhasználó által definiált infix operátorok, amelyek a feladatok eredményeit és a globális feladatállapotot manipulálják. A TOP-hoz specifikus, hogy a feladatok, amelyek megfigyelhető közbenső eredményeket állítanak elő, amíg a feladatokat újra és újra megismételjük, amíg egy stabil eredményt nem adnak, vagy az eredményekre már nincs szükség. Ehhez magasabbrendű függvényekre, lustaságra és automatikus szemétgyűjtésre van szükség.

Az iTask rendszer egy webszervert generál, amelyet a fejlesztők használhatnak, hogy megtalálják a feladatukat. Mint minden webszerver, ennek is szüksége van az állapot szerializációjára és deszerializációjára, hogy eltárolja és visszanyerje az aktuális állapotot. A tetszőleges algebrai adattípusokra vonatkozó webes űrlapok kitöltésével a felhasználók jelzik előrehaladásukat a feladattal. Minden funkciót generikus programozással implementáljuk.

Ahhoz, hogy implementáljuk egy SDS értékét monitorozó feladatokat, létezik egy rejtett publish-subscribe rendszer minden SDS számára, amely aktiválja az SDS-t használó feladatokat, amikor az értékük frissül.

Ezektől a tulajdonságoktól eltekintve az iTask rendszer sok további technológiát alkalmaz, például egy böngészőt futtató értelmezőben a feladatrészek végrehajtása a rendszertől kapott gyors válasz biztosítása érdekében a nagyon interaktív feladatok, mint a rajzolás esetében.

4.2 Oktatása a Téli Iskolában

Bármilyen programozásoktatás esetén az elsajátításhoz szükség van gyakorlati programozási tapasztalatra az oktatott technológiákkal. Ez ugyanúgy a TOP-ra is vonatkozik. Ennek eredményeképp a négyórás *Why "Task Oriented Programming" matters* kurzust két, közel egyenlő hosszúságú részre bontjuk.

Az első részben átvesszük a TOP koncepcióját az iTask rendszer használatával. A hallgatók többségének hátterére való tekintettel majdnem minden részletet ki kell hagynunk a rendszer implementációjáról, és helyette a könyvtár használatára kell fókuszálnunk. Ez a könyvtár valójában egy sekélyen beágyazott DSL a TOP-ból. Az előadáson ezen eljárások halmazát használjuk anélkül, hogy túl sok időt töltenénk az architektúra és az implementáció megmagyarázásával.

A gyakorlaton felosztjuk a létező alappéldát több kisebb független TOP projektre. A beadand feladatok ezeknek a projekteknek kisebb variációit fogják tartalmazni, hogy elősegítsék a tapasztalatszerzést a feladatorientált programozás terén.

A tapasztaltabb funkcionális programozók kihagyhatják az alapvető gyakorlati példákat és a magasabb szintű feladatokra ugorhatnak. Ezáltal képesek leszünk alkalmazkodni minden résztvevő egyéni képességeihez.

5 Az mTask rendszer

A mikroprocesszorok számítógépes rendszerek, amelyek nagyon korlátozott számítási kapacitásokkal rendelkeznek. Általában meglehetősen alacsony órajellel és erős memóriát érintő megszorításokkal rendelkeznek, pl. egy futó program néhány KB memórián tárolhatja az adatait. Ezek az olcsó processzorok a dolgok internete (Internet of Things, IoT) számos elemének vezérlőerejét alkotják. Ezekben a mikroprocesszoros rendszerekben egy tipikusan számos input portot ellenőriz, valamint néhány outputot, a megfigyelések alapján. A hardveres megszorítások miatt jellemzően nincs hozzájuk támogatást nyújtó operációs rendszer.

A TOP paradigma erőforrás-kímélő szálakat biztosít, amelyek jól alkalmazhatóak a jól definiált, egyszerű feladatok monitorozására és koordinálására. Ezek a feladatok a saját tempójukban futhatnak, amíg kombinátorokat és megosztott adatforrásokat használunk a koordinálásukhoz. Az iTask rendszer futtatása az IoT eszközökön lehetővé teszi olyan programok konstruálását, amelyeket részben a webszerveren, részben az IoT eszközökön hajtunk végre. A mikroprocesszorok korlátai ellehetetlenítik a teljesértékű iTask programok futtatását IoT eszközökön.

Az ideális megoldás megbecsüléséhez fejlesztettük ki az mTask rendszert [3, 2]. Ez egy többszörös nézet, amely sekélyen DSL-be van ágyazva, amely az iTask rendszer részeként használható. Támogatja a TOP paradigmát, többek között ugyanazokat a feladateredményeket, mint az iTask rendszer, a feladatkombinátorokat és a megosztott adatforrásokat. Konstrukció szerint ez a DSL nem rendelkezik magasabb rendű függvényekkel, de nincsenek rekurzív adattípusai. Az ebben a DSL-ben állított megszorítások miatt az mTask programokat mikroprocesszorok által futtatható kóddá lehet fordítani. E megszorítások ellenére a DSL teljesen alkalmas az IoT eszközökön végrehajtandó feladatok könnyű és pontos specifikálására.

5.1 Használt technológiák

Az mTask rendszer egy többszörös nézet, amely sekélyen DSL-be van ágyazva típuskonstruktor osztályok alapján [1]. Ezen osztályokból minden példány egy *view* nevű interpretációt definiál egy egyszerű eljárásokból konstruált programból. Tipikus view-k pedáns megjelenítést, mikroprocesszorok számára kódgenerálást, és mTask programok iTask szimulációját implementálják.

A DSL-t konstrukció szerint kiterjeszthetjük, hogy újra felhasználjunk létező könyvtárakat olyan perifériák számára, mint hőmérsékletszenzorok, képernyők és szervo motorok. Egy ehhez hasonló könyvtárat egyszerű hozzáadni az mTask rendszerhez egy nyelvi eljárásként úgy, hogy egy új típuskonstruktort és a fontos példányokat létrehozzuk.

Ahhoz, hogy az mTask implementációt hordozhatóvál tegyük több különböző mikroprocesszor számára hogy megkönnyítsük a létező C++ könyvtárak újrafelhasználását, a kódgeneráló view C++ kódot állít elő az Arduino platformra

valamilyen specifikus processzor által futtathat natív gépi kód helyett. Az Arduino platformon belül lévő avr-gcc fordító a generált C++ kódot és a sok különböző mikroprocesszor számára készített natív kódot le tudja fordítani.

5.2 Oktatása a Téli Iskolában

Jó ötletnek tűnik magasszintű TOP programokat végrehajtani perifériákkal kommunikáló kis mikroprocesszorokon a téli iskolai kurzus keretében. Azonban mTask programokat végrehajtani egy tényleges mikroprocesszorokon túl nagy elvárás lenne a hallgatók felé; össze kell állítaniuk egy mTask programot az iTask rendszeren belül, végre kell hajtaniuk az iTask programot a C++ kód előállítására, ezt át kell adniuk az Arduino IDE-nek, össze kell kötniük az Arduino IDE-t a mikroprocesszorral és ki kell választaniuk a megfelelő opciókat, be kell tölteniük a lefordított programot a mikroprocesszorba, végül futtatniuk kell. Ezen lépések mindegyike egyszerű, de a teljes folyamat csak akkor ad eredményt, ha minden lépést helyesen hajtunk végre. Mivel a generált program operációs rendszer nélkül és erősen korlátozott input/output perifériákkal fog futni egy mikroprocesszoron, egy ilyen program debugolása nagy kihívás. Hosszú megbeszélés után e lépések sorozatát túl bonyolultnak nyilvánítottuk az adott időhöz és a résztvevőkhöz képest.

Szerencsére az mTask rendszer szimulátor view-ja alternatívát kínál, amelyet sokkal egyszerűbb használni. Ez a view átalakítja az mTask programot egy egyszerű iTask programmá. A szimulátor egy lépésről lépésre történő végrehajtást kínál az mTask programhoz. Megjelenít egy nyomot az utoljára végrehajtott feladat utolsó lépéséről és az összes perifériáról és megosztott adatforrásról. Az óra, az SDS-ek értéke, valamint a perifériák állapota interaktívan megváltoztatható annak érdekében, hogy felügyelet alatt tartsuk a végrehajtást és felderítsünk többféle történést. Ez az útmutató a gyakorlati munkát az előző iTask útmutató közvetlen rákövetkezőjévé teszi.

Eldöntöttük, hogy ezeket a kurzusokat egy napon tartjuk az iTask előadással és a hozzátartozó reggeli gyakorlattal úgy, hogy az mTask kurzust délután tartjuk. Ezáltal az mTask kurzus közvetlenül építhet arra a tudásra, amelyet a résztvevők az iTask kurzuson szereztek. A TOP megértése, amelyet reggel megalapozunk, a délutáni kurzus során tovább mélyül.

6 Konklúzió

A TOP-ot érintő mindkét kurzus és más érdekes témák is szerepelni fognak a téli iskola ideje alatt. A kurzusok közben a TOP paradigma megértésére és használatára fogunk fókuszálni viszonylag egyszerű példákon keresztül. Haladóbb példákkal fogjuk illusztrálni e megközelítés kapacitásait. A gyakorlatokon szemléletes példákra fogunk koncentrálni, amelyek nagyrészt a kurzusokon bemutatott példák variációi. A haladó résztvevők számára is lesz néhány kihívást jelentő beadandó feladat, és lehetőség adódik a bemutatott rendszerek aspektusainak megvitatására.

Köszönetnyilvánítás

Ez a cikk az Erasmus+ Kulcsfontosságú Akciók 2 projektcsalád `2017-1-SK01-KA203-035402` számú projektjének – Stratégiai Partnerség a Felsoktatásért, *„A mköd szoftverek összeépíthetségének, megérthetségének és helyességének oktatását elsegítését célzó”* *3COWS* (*“Focusing Education on **C**omposability, **C**omprehensibility and **C**orrectness of Working Software” (3COWS)*) projekt eredményeként valósult meg.

Felelősség kizárása

E kiadványban közölt tudás, információk és állítások a szerz(k) véleménye(i), meggyzdése(i) és világnézete(i) alapján kerültek megfogalmazásra, eképp nem feltétlenül tükrözik az Európai Unió hivatalos álláspontját. Sem az Európai Unió intézményei és testületei, sem bármely Uniós intézmények és testületek megbízatásából eljáró személy nem tehet felelssé a kiadványban szerepl megállapítások felhasználásáért, vagy a felhasználásból ered bármilyen következményért.

Hivatkozások

1. Carette, J., Kiselyov, O., Shan, C.c.: Finally tagless, partially evaluated: Tagless staged interpreters for simpler typed languages. J. Funct. Program. **19**(5), 509–543 (Sep 2009). https://doi.org/10.1017/S0956796809007205, `http://dx.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). https://doi.org/10.1007/978-3-319-39110-6_6, `http://dx.doi.org/10.1007/978-3-319-39110-6_6`
4. Peyton Jones, S.L., Wadler, P.: Imperative functional programming. In: Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 71–84. POPL '93, ACM, New York, NY, USA (1993). https://doi.org/10.1145/158511.158524, `http://doi.acm.org/10.1145/158511.158524`
5. Plasmeijer, R., Achten, P., Koopman, P.: iTasks: executable specifications of interactive work flow systems for the web. In: Hinze, R., Ramsey, N. (eds.) Proceedings of the ICFP'07. pp. 141–152. ACM, Freiburg, Germany (2007)
6. Plasmeijer, R., van Eekelen, M., van Groningen, J.: Clean language report (version 2.2) (2011), `http://clean.cs.ru.nl/Documentation`

7. Wadler, P.: Comprehending monads. In: Proceedings of the 1990 ACM Conference on LISP and Functional Programming. pp. 61–78. LFP '90, ACM, New York, NY, USA (1990). https://doi.org/10.1145/91556.91592, http://doi.acm.org/10.1145/91556.91592

Színezett Petri Hálók Oktatásának Interaktív Megközelítése

Štefan Korečko

Department of Computers and Informatics,
Faculty of Electrical Engineering and Informatics,
Technical University of Košice, Letná 9, 041 20 Košice, Slovakia
`stefan.korecko@tuke.sk`

Abstract. A formális módszerek olyan technikák közé tartoznak, melyek – megfelelően használva – nagyban hozzájárulhatnak a fejlesztett szoftveres és hardveres rendszer helyességéhez. A konkurrens vagy nemdeterminisztikus tulajdonságokkal bíró rendszerek egyik leírási formája a színezett petri hálók modellező nyelve. Jelen cikk célja, hogy bemutatassa miként lehet oktatni ennek a modellező nyelvnek alapvető elveit, illetve kapcsolatát a funkcionális programozási nyelvekkel. A két és fél órát felölelő foglalkozás során modellalkotásra került sor interaktív módon, a résztvevők aktív közreműködésével.

1 Bevezetés

Figyelembe véve a modern társadalom növekvő függőségét a számítógépes rendszerektől, azok helyessége rendkívül fontos. Az egyik megközelítés, amely jelentősen hozzájárulhat a helyességhez, a formális módszerek alkalmazása a szoftver- és hardverfejlesztés során. A formális módszerek egy matematikai alapokon nyugvó technikaák, melyek formális nyelvet biztosítanak egyértelműen definiált szintaxissal és szemantikával, valamint egy olyan eszközzel, amely lehetővé teszi a hitelesítési, fejlesztési és szimulációs feladatok elvégzését a rendszer specifikációival. A formális módszerek családjának egyik jelentős tagja a színezett Petri háló (CPN) modellező nyelv. A CPN [4, 3] ötvözi a Petri hálók formalizmusát [1] egy funkcionális nyelvvel az adatkezelési és döntési eljárások kezelésére. A funkcionális nyelvet CPN ML-nek hívják, és ez a Standard ML [2, 5] kissé módosított változata. A CPN nyelvét, valamint a megfelelő specifikációs, ellenőrzési és szimulációs feladatokat a CPN Tools [6] szoftver támogatja.

A CPN több mint egy évtizede a formális módszerekkel, modellezéssel és szimulációval kapcsolatos egyetemi kurzusok része. Az egyik módszer, amelyet a szerző a CPN fogalmainak magyarázata során alkalmazott, egy interaktív megközelítés, a hallgatóság aktív részvételével. Itt a közönség kiválasztja azt a domént és folyamatot, amelyre a CPN modellt megtervezik, és segít kiválasztott részeinek létrehozásában. Ennek a megközelítésnek az egyetemi tanárok képzési tevékenysége során történő megvalósításából származó tapasztalatokat a cikk többi része ismerteti.

2 Képzési tevékenység az interaktív CPN modell létrehozásával

A képzést kb. 10 résztvevő számára szervezték meg, akik funkcionális nyelvi háttérrel rendelkező egyetemi oktatók voltak. A résztvevőknek nem voltak korábbi ismereteik a CPN-ről. A tevékenység teljes időtartama körülbelül 2,5 óra volt, a szünetek kivételével, és három szakaszra osztották.

Az első szakasz körülbelül 30 percet vett igénybe, és elmagyarázta a CPN alapelveit. Nevezetesen, hogy a CPN grafikusan ábrázolva egy páros gráf, amely kétféle csúcsot tartalmaz: ellipszisként rajzolt helyek és téglalapokként rajzolt átmenetek. A helyek tartalmaznak tokeneket, amelyek a hálózat állapotát képviselik, és az átmenetek eseményekként értelmezhetők, amelyek megváltoztatják az állapotot a meglévő tokenek felhasználásával és újak létrehozásával.

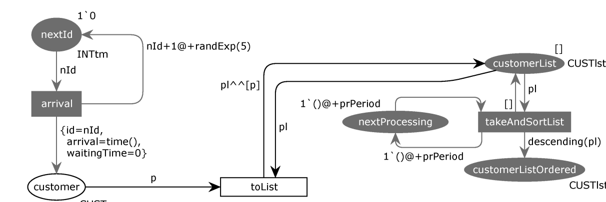


Fig. 1. A kezdő CPN-modell, amelyet a tevékenység résztvevői kapnak

A második és a harmadik fázist a CPN modell létrehozására fordították. Mivel a tevékenység egyik célja az volt, hogy megmutassa, hogy néhány fejlettebb Standard ML fogalom, nevezetesen a struktúrák és funktorok, használhatóak a CPN modellekben, a résztvevők kaptak egy kezdő CPN modellt, amely már alkalmazta a fogalmakat kezdésként a második fázis előtt. Az indítómodell a 1 ábrán látható. Az a rész – amely a `nextId`, `arrival` és `customer` csomópontokból áll – az ügyfelek érkezését modellezi, akik egyenként érkeznek kiszolgálásra. Maga az adagolás nem szerepel a kezdő modellben. Ehelyett van egy `toList` átmenet, amely egy tokent vesz a `customer` és hozzáadja értékét egy listához, amelyet a `customerList` helyen tartanak. A `takeAndSortList` átmenet rendszeres időközönként kerül felhasználásra, amelyet a `prPeriod` érték határoz meg. A `takeAndSortList` minden aktiválása törli a listát a `customerList`-ben, rendezi annak tartalmát és a megrendelt változatot a `customerListOrdered`-ben tárolja. A hely `nextProcessing` kiegészítő szerepet játszik és biztosítja, hogy `takeAndSortList` csak rendszeres időközönként tüzel. A rendezést egy `descending` elnevezésû függvény biztosítja, amely a Quicksort algoritmust valósítja meg. A funkció a szabványos ML struktúrákat és funktorokat használja.

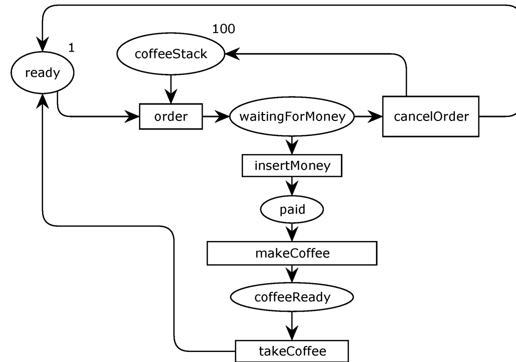


Fig. 2. A kiszolgáló rész CPN modellje, interaktív módon létrehozva a második szakaszban

A tevékenység résztvevői úgy döntöttek, hogy modelleznek egy kávé automatát. A második szakaszban részt vettek egy CPN modell létrehozásában, amely rögzíti a gép alapvető működését. A modellt a 2 ábra mutatja. A gép kezdeti állapotában készen áll az ügyfelek kiszolgálására (egy zseton a `ready` helyen), és 100 kávé adaggal van feltöltve (100 zseton `coffeeStack`). A kiszolgálás azzal kezdődik, hogy egy ügyfél megrendel egy kávét az `order` átmenet aktiválásával. Ezután a gép megvárja az ügyfél következő lépését (egy token a `waitingForMoney`-ban). Az ügyfél pénzt helyezhet be (`insertMoney` aktiválása) vagy törölheti a megrendelést (a `cancelOrder` aktiválása). A visszavonás visszaállítja a készüléket a “ready” állapotba. Ha a pénzt behelyezik, a gép elkészíti a kávét (`makeCoffee` aktiválásával). Végül, az `takeCoffee` aktiválásával az ügyfél elkészíti az elkészített kávét, és a gép visszatér “ready” állapotba.

A második szakasz után kb. 70 percnyi szünet volt. A szünet alatt az előadó a 2. fázistól a modellt összekapcsolta az indító modell részeivel, és csúcsokat és íveket adott hozzá az ügyfél viselkedését leírva. Kijavította a modell néhány következetlenségét is, amelyet az egyik résztvevő rámutatott. A kapott végső CPN modell a 3 ábrán látható. Az indító modellből vett csúcsok (1 ábra) változtatás nélkül szürke színben jelennek meg. A `customer` hely helyébe a `customerQueue` lép, amelyben az értéklistával rendelkező token található, amely a gépen várakozó ügyfelek sorát képviseli. Az átmenet `toList` helyett van egy kiszolgáló rész, amelyet a második fázis eredményeként hoztak létre (2 ábra). A végső modell kiszolgáló része három fő szempontból különbözik a 2 ábrától:

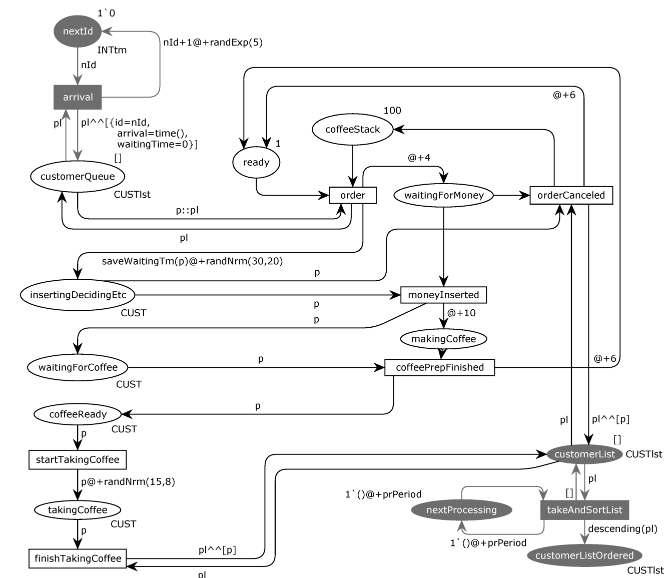


Fig. 3. A kávéfőző CPN modelljének végső formája

– A tokenek információkat tartalmaznak a kiszolgált ügyfélről, és az ív kifejezések határozzák meg a megfelelő műveletek időtartamát.
– A helyek és az átmenetek szerepével kapcsolatos következetlenségeket kijavítják. Most az összes átmenet pillanatnyi eseményeket képvisel. Például a `makeCoffee` átmenet (2 ábra) helyébe a `makingCoffe` hely, valamit a `takeCoffee` helyébe a `startTakingCoffee`, `takingCoffe` és `finishTakingCoffe` csúcsok kerültek.
– Az ügyfelek és a gép műveleteit és állapotát külön-külön modellezik. A `customerQueue`, `insertingDecidingEtc`, `waitingForCoffee`, `coffeeReady`, `startTakingCoffe`, `takingCoffee` és `finishTakingCoffe` csúcsok az ügyfelekhez tartoznak, míg a kiszolgáló rész fennmaradó része a gépet vagy mindkét felet képviseli.

A képzési tevékenység harmadik szakaszát a végső modell kiértékelésére és az ilyen modelleknek a helyes számítógépes rendszerek fejlesztésében való szerepének megvitatására fordították. Kb. 30 percig tartott.

3 Összefoglalás

Az itt bemutatott interaktív képzési tevékenység rövid, intenzív kurzusokra alkalmas, amelyek gyakran nyári iskolákban vagy más hasonló oktatási rendezvények során zajlanak. A tevékenység leírt tesztfuttatása során kiderült, hogy az eredeti 2 órás keret nem volt elegendő. Ezért szükség van a harmadik szakaszra, ahol az előadó bemutatta a végső modellt. Figyelembe véve azt az időt, amely alatt az előadó elkészíti a végső modellt, további legalább két órára lesz szükség ahhoz, hogy az egész modell létrehozásának folyamatát interaktív módon, a hallgatósággal elvégezzük. Az itt bemutatott vagy említett CPN-modellek a szerzőtől kérésére beszerezhetők.

Köszönetnyilvánítás

Ez a cikk az Erasmus+ Kulcsfontosságú Akciók 2 projektcsalád `2017-1-SK01-KA203-035402` számú projektjének – Stratégiai Partnerség a Felsőoktatásért, *„A működő szoftverek összeépíthetőségének, megérthetőségének és helyességének oktatását elősegítését célzó” **3COWS*** (*“Focusing Education on **C**omposability, **C**omprehensibility and **C**orrectness of Working Software” (3COWS)*) projekt eredményeként valósult meg.

Felelősség kizárása

E kiadványban közölt tudás, információk és állítások a szerző(k) véleménye(i), meggyőződése(i) és világnézete(i) alapján kerültek megfogalmazásra, eképp nem feltétlenül tükrözik az Európai Unió hivatalos álláspontját. Sem az Európai Unió intézményei és testületei, sem bármely Uniós intézmények és testületek megbízásából eljáró személy nem tehető felelőssé a kiadványban szereplő megállapítások felhasználásáért, vagy a felhasználásból eredő bármilyen következményért.

References

1. Desel, J., Reisig, W.: Place/transition petri nets. In: Reisig, W., Rozenberg, G. (eds.) Lectures on Petri Nets I: Basic Models, Lecture Notes in Computer Science, vol. 1491, pp. 122–173. Springer Berlin Heidelberg. DOI: 10.1007/3-540-65306-6 (1998)
2. Harper, R.: Programming in Standard ML. Carnegie Mellon University (2011), `http://www.cs.cmu.edu/~rwh/smlbook/book.pdf`
3. Jensen, K.: An introduction to the theoretical aspects of coloured petri nets. In: A Decade of Concurrency, Reflections and Perspectives, REX School/Symposium. pp. 230–272. Springer-Verlag, London, UK. DOI: https://doi.org/10.1007/3-540-58043-3_21 (1994)
4. Jensen, K., Kristensen, L.M.: Coloured Petri Nets: Modelling and Validation of Concurrent Systems. Springer. DOI: 10.1007/b95112 (2009)
5. Milner, R., Tofte, M., Macqueen, D.: The Definition of Standard ML. MIT Press, Cambridge, MA, USA (1997), `http://sml-family.org/sml97-defn.pdf`
6. Cpn tools homepage (2018), `http://cpntools.org/`

CodeCompass: egy kiterjeszthető kódmegértési keretrendszer

Brunner Tibor

Informatikai Kar, Eötvös Loránd Tudományegyetem,
Budapest, Magyarország
bruntib@caesar.elte.hu

Kivonat A *CodeCompass* egy nyílt forráskodú kódmegértés-támogató eszköz, melynek elsődleges célja nagyméretű, hosszú életciklusú projektek megértésének támogatása. A CodeCompass pontos információt ad a C/C++ nyelvi elemek legkomplexebb előfordulásaira, melyet az LLVM-Clang fordítói keretrendszerre épített elemző segítségével teszi. A vizualizációs eszközök széles spektruma magában foglalja az osztály és függvényhívás diagramokat, architekturális, komponens és interfész diagramok készítését, „*points-to*" analízis elvégézést és még sok mást. Az eszköz ezen felül felhasználja a vizsgált szoftverprojekt fordítását vezérlő rendszer információit a projekt architekturájának még pontosabb modellezésére, amennyiben ez megfelelően rendelkezésre áll. A Clang Statikus Elemző (*Clang SA*) és más Clang-alapú elemző eszközök eredményeit a projekt forráskódjával együtt tekintheti meg a felhasználó. Ámbár a CodeCompass elsődleges fókuszcsoportja C és C++ projektek voltak, lehetpség van Java és Python projektek navigálására is. Egy webalkalmazás alapú, kiegészíthető csomagokkal bővíthető architektúrájú jellege miatt a CodeCompass a kódmegértés-támogatáson felül a statikus analízis és a metrikák elemzésének platformja lehet.

1 Bevezetés

A szoftverprojekteken végzett munka során mind a hibajavítás, mind az új fejlesztések elvégzéséhez elengedhetetlen egy biztos tudás a projekt részleteiről és arról, hogy milyen hatása lehet a tervezett módosításoknak. A kódmegértés-támogató eszközök célja a programból (nem kizárólag csak a forráskódból, hanem más elérhető információ felhasználásával karöltve) modellek építésével az eredeti szándékok és az implementációs részletek közös megjelenítése. Bár számos ilyen eszköz elérhető szabadalmaztatott vagy ingyenesen felhasználható szoftverként, a képességeik korlátosak.

A CodeCompass fejlesztésének célja ezen korlátok feloldása volt. A projekt az Ericsson Magyarország Kft., és az Eötvös Loránd Tudományegyetem közös erőfeszítése, melynek célja nagy szoftverrendszerek megértésének támogatása. A megértéshez szükséges az olyan magas komplexitású nyelvi elemek támogatása, amelyeket pusztán a kód olvasásával nem lehetne pontosan felfedezni: ilyenek például a túlterhelés, az akár többszörös öröklődési hierarchiák, függvénymutatók és *virtuális* függvények használata. A meglévő eszközök ezeket a lehetőségeket csak korlátozott mértékben támogatják, ezért a CodeCompass alapját egy valós, éles fordítóprogram, az *LLVM/Clang Compiler Infrastructure Project* képezi. Ezzel átléphetőek a gyengeségei más „pehelysúlyú" kódmegértési eszközöknek, mint például az *OpenGrok*.

Azonban a CodeCompass nem csak a forráskód elemzését végzi el. A projekt fordítását végző eszközből kinyerésre kerülnek információk amelyek a szoftver architekturális összefüggéseit segítik felfedezni. Ha elérhető valamilyen verziókövetési rendszer (VCS) a projekthez, akkor ennek információi is felhasználásra kerülnek: például a kód böngészése közben felfedezhetőek forrásállományok közötti összefüggések abból kiindulva, hogy „véletlenül" pont fájlok egy csoportját módosították a fejlesztők mindig együtt egy-egy *commitban*. A gyors és pontos megértés támogatása érdekében a CodeCompass szöveges és grafikus felületeket és reprezentációkat épít a számára megadott szoftverprojektből. Számos (interaktív) diagram és vizualizáció érhető el, kezdve a megszokott függvényhívási-gráfoktól egészes az architekturális megjelenítésig. Az eszköz elérésének támogatása érdekében az egy webalkalmazás-alapú architektúrán nyugszik. A kliens egy tetszőleges (modern) webböngésző, kódszerkesztő program bővítménye vagy bármely harmadik fél által elkészített szoftver. A szerverrel való kommunikáció webes csatornán, REST API segítségével történik, és jól skálázódik a párhuzamosan dolgozó kliensek számával.

Ebben a cikkben bemutatom a CodeCompass képességeit és összehasonlítom a megvalósított eszközt más, elérhető, hasonló célt szolgáló szoftverekkel. Erre az összehasonlításra a 2. fejezetben kerül sor, ahol áttekintjük a kódmegértést-támogató eszközök alaptípusait. A CodeCompass kiterjeszthető architektúráját a 3. fejezetben tárgyalom. A legfontosabb képességei a CodeCompassnak a 4. fejezetben kerülnek bemutatásrsa. A cikk az 5. fejezetben összefoglalóval zárul.

2 Kapcsolódó munkák

A szoftverpiacon számos olyan eszköz érhető el, amelyek valamilyen szinten támogatják a forráskódok megértését. Néhányuk statikus analízist használ, mások a feldolgozott szoftver dinamikus, futás közbeni viselkedését is vizsgálják. Az eszközök feloszthatók őstípusokba az architektúrájuk és a főbb elveik szerint.

A *szerver-kliens architektúrájú eszközök* elkészítik a számukra megadott projekt modelljét egy elemzési lépésben, az adatokat pedig valamilyen adatbázisba integrálják. A felhasználókat a klienszoftver szolgálja ki, amely általában webalkalmazás alapú. Az ilyen eszközök elemzési lépései könnyedén beilleszthetőek a fejlesztői munkafolyamatba a *continuous integration (CI)* részeként. A munkafolyamatba integráltság kihasználásával a fejlesztők mindig képesek a hosszú életciklusú projektek teljes egészét navigálni.

Léteznek azonban kliensek, melyek a projekt csak egy kisebb egységét elemzik. Ilyen típusú viselkedésre van szükség például integrált fejlesztői keretrendszerek (IDE) szerkesztőiben, ahol a forrásfájlok gyakori és kis méretű módosításai a tudásbázis gyors frissítésének lehetőségét követelik meg.

Ebben a fejezetben bemutatunk néhány eszközt mindkét kategóriából, melyeket ipari környezetben használnak.

Woboq [3]: egy webalkalmazás-alapú kódnavigációs eszköz C és C++ nyelven készített projektekhez számos olyan képességgel, amelyek specifikusan a gyors kódböngészésre lettek tervezve. A fájlok és programentitások gyorsan kereshetőek a kódkiegészítést is biztosító keresőmező használatával. A kódnavigáció során az elemezési lépésben előre elkészített statikus HTML weblapokat tekinti meg a felhasználó. Az előre elkészítés előne, hogy a kiszolgálón nincs szükség semmilyen további, működés közbeni számítás elvégzésére.

Az egérrel egy adott függvényre, típusra, változóra, makróra, egyéb nyelvi elemre pozícionálva megjeleníthető az adott programegység részletes tulajdonságai. Egy ilyen például függvény kijelölése esetén annak szignatúrája, definíciójának és felhasználásának (hívásának) helyei. Rekordokra, típusokra megtekinthető a rekord elemeinek mérete, belső szerkezete, valamint a típusrendszerben, az öröklési hierarchiában elfoglalt helye. Változók esetén a felhasználó látja a változó (statikus) típusát, valamint a változó írási és olvasási felhasználásait.

A makrók a C és C++ nyelv kereti között egy beágyazott nyelvet alkotnak, amelyek a fordítás során, annak legelső lépéseként oldódnak fel. A makrók kiértékelése szövegrészletek lecserélését jelenti a forráskódban, azaz a fordítóprogram más kódszöveget fog fordítani, mint amit a forrásfájlt olvasva a programozó látna. A makrók teljesen kifejtett állapota megjeleníthető a Woboq-ban.

Az eszköz egy hasznos tulajdonsága a szemantikus információt felhasználó kódszínezés, melynek segítségével a különböző nyelvi elemek ránézésre elkülöníthetőek az eltérő formázás miatt.

A Woboq azért képes ennyire széleskörű információmegjelenítési képességeket felmutatni, mert az elemzési fázisban egy valódi fordítóprogramot használnak, az eszköz LLVM/Clang infrastruktúrára épül, amely rendelkezésre bocsát egy absztrakt szintaxisfa reprezentációt, melyen keresztül ugyanaz a szemantikus információhalmaz érhető el, mint amit a projekt fordításakor a fordítóprogram a binárisok előállításához használna. Ez azonban a Woboq egyik legnagyobb hátrányának az okozója is egyben, mivel csak a C nyelvcsaládba tartozó projektekre használható.

OpenGrok [4]: egy gyors forráskódkereső és kereszthivatkozásokat biztosító eszköz. A Woboq-kal ellentétben nem végez mély elemzést a projekten, így nem képes részletes szemantikus információk felhasználásával dolgozni. A programegységek jellegének meghatározásához a *CTags* [5] eszközt használja, amely szövegesen dolgozza fel a forráskód fájlokat, és egyszerű szintaktikus elemzést végez. A felfedezett programegységek közötti keresés gyors és optimalizált, így nagyméretű szoftverek esetén is megfelelő felhasználói élményt biztosít. A kódban való keresést összetett kifejezések segítségével végezhetjük el (például `defs:target`), amelyek helyettesítő karaktereket tartalmazhatnak, valamint az eredményeket a projekt egy almappájára szűrhetjük. A szöveges keresésen kívül az OpenGrok lehetőséget biztosít szimbólumok és definíciók felderítésére is. Mivel a *CTags* nem használ szemantikus elemzést, képes nagy számú (a cikk írásakor 41) programozási nyelvet támogatni. Egy hozzáadott előnye ennek a megvalósításnak az inkrementális adatbázis-frissítések támogatottsága a projekt fejlődésével összehangoltan. Ezenfelül lehetőség van verziókövetési rendszerekből, például Mercurial, SVN, Git, való információk felhasználására is.

Understand [6]: nem csak egy kódnavigáló eszköz, hanem egy teljesértékű integrált fejlesztőkörnyezet (IDE). Az *Understand* előnye abból származik, hogy a forráskód helyben módosítható, és a módosítás eredményei azonnal látszanak.

A korábbi eszközöknél említett kódnavigálási képességeken túl az Understand számos metrika számítási és jelentéskészítési lehetőséget is támogat. Ezek között találhatóak: kódosorok száma (*LoC*, összesen, átlagosan, osztályonként), összefüggő/bázis/leszármazott osztályok száma, kohéziós metrikák [2], *McCabe*-komplexitás [1], és sok más. A metrikák a megszokott *Tree-map* reprezentációban kerülnek a felhasználó számára közvetítésre, amely egy egymásba ágyazott téglalapokból álló nézet, ahol az egymásbaágyazottság a hierarchikus alá-fölé rendeltséget, a téglalapok egymáshoz viszonyított színe és mérete pedig a kiválasztott metrika szerinti értéket mutatja.

Nagyméretű szoftverprojektek esetén a termék architektúrájának az áttekintése fontos lépése a tervezési és fejlesztési folyamatnak. Az Understand képes kirajzolni függőségi gráfokat a „függőség" számos értelmezésének felhasználásával: függvényhívások, öröklődési hierarchia, projektfájlok közötti kapcsolatok. Az eszköz egy programozási felületet (*API*) biztosít felhasználói diagramtípusok építésére.

A programozás központi fogalmai a legtöbb nyelvben azonosan léteznek, de majdnem minden nyelv hordoz valamilyen sajátosságot. Az eszköz előnye, hogy ~ 15 nyelvet kezel és ezekhez képes az adott nyelv sajátosságait figyelembe vevő megjelenítéseket adni, például a C, C++ nyelvhez a függvénymutatók jó kezelése, vagy Ada esetén a csomagok hierarchiájának megértése.

Az eszköz működési alapja az elemzésre átadott projektről adattárház építése, melyből egy API-n keresztül kinyerhető az összes információ: ennek segítségével a felhasználó olyat is lekérdezhet, amelyet a kulcsrakész felhasználói felületen nem tudna kiszámítani.

CodeSurfer [7]: hasonló eszköz, mint az *Understand*, hiszen ez is vastag klienst és statikus analízist használ. Az eszköz fő célterülete a C/C++ nyelven, vagy `x86` architektúrára gépi kóddal írt projektek. Mély analízist végez a nyelv szabályainak ismerete mellett, melynek segítségével részletes tudás állapítható meg a projektről, és az elkészített szoftver viselkedéséről. Például támogat *pointer-analízist*, melynek segítségével kiszámítható, hogy egy konkrét változóhoz milyen mutatókon keresztül lehet hozzáférni. Ezen felül *slicing* alkalmazásával megtekinthető, hogy egy adott utasítás végrehajtásától mely más utasítások függnek. Adatfolyam-analízis segítségével pontosabban meghatározható az a pont, ahol a változó az adott értéket felvette.

3 A *CodeCompass* architektúrája

Az előző fejezetben megmutattam néhány nézőpontját a kódmegértés-támogató eszközök céljainak és architektúrájának. Ebben a fejezetben megmutatom, hogy

a CodeCompass hogyan helyezhető el az előbb felsorolt eszközökből álló széles skálán.

A CodeCompass kliens-szerver architektúrával rendelkezik, mely során a navigálás előtt végrehajtott elemzési lépés által kigyűjtött tudást adja a felhasználó rendelkezésére. Ennek oka a CodeCompass által megvalósítandó célban keresendő: a *kódszerkesztő* eszközökkel ellentétben a CodeCompass egy *kófmegértés-támogató* eszköznek lett szánva. A két használati eset és hozzáállás között alapvető különbségek vannak: kód írása közben a programozó egy jól behatárolt néhány soros részét kezeli csak a forráskódnak; szemben a kódmegértéssel, amikor forrásfájlokon, csomagokon, a projekt logikai egységein átívelő gondolkodást kell végrehajtani. A fejlesztői környezetekben és kódszerkesztőkben az egyik leghasznosabb funkció a kódkiegészítés – a programozónak nem kell megjegyezenie az összes elérhető változót, mezőt, függvényt, mert a szerkesztő ezt kilistázza neki; kódmegértés során azonban számos vizualizációra van szükség, hogy a fejlesztő átlássa az programegységek közötti összefüggéseket.

A CodeCompass felhasználói felülete webalkalmazás alapú. A korábban említett vizualizációk mindegyike lekérdezhető egy nyilvános API-n keresztül, amelyet a szerverfolyamat biztosít. A webes felület a felhasználó böngészőjében biztosítja a gyors és kényelmes kódnavigációt, lefúrási és megértési lépéseket. Azonban a CodeCompass több, mint egy navigációs eszköz – a CodeCompass egy keretrendszer: egy kiegészíthető gyűjtő és megjelenítő szoftvere a statikus analízis eredményeinek. A központi szerveren végzett eredménytárolás elősegíti azt, hogy ne kelljen a felhasználói gépeken masszív adatmennyiséget kezelni, amelyet egyes felhasználók nem biztos, hogy használnak, míg másoknak szüksége lehet rájuk. Példaként, amennyiben egy konkrét pillanatban szükséges, elkészíthető egy szkript, amely az egymást hívó függvények lezártját számítja ki egy adott függvényből indítva, így a teljes projekt egy szeletét adja a felhasználó kezébe.

A CodeCompass tervezése és fejlesztése során a másik fő szempont a nagyméretű projekteken végzett gyors – maximum másodpercekben mérhető felhasználói várakozás – kódmegértés támogatása volt. Ennek érdekében az adatbázisba csak a lehető legkisebb szükséges információt tároljuk el az elemzés során, melyből a kérések eredménye gyorsan kiszámítható. Kezdetben a pontos lekérdezések támogatása miatt a teljes szintaxisfa-reprezentációt az adatbázisba íruk, amely eredménye egy 1 : 1000 (forráskód:adatbázis) méretarány lett. Ezt követően megállapítottuk, hogy felhasználók majdnem minden esetben csak névvel ellátott programegységekre kíváncsiak (függvények, változók, osztályok, makrók, stb.) így felesleges a „maradékhoz" tartozó szintaxisfa-csúcsok (vezérlési szerkezetek, típuskonverziók, egyéb a C++ nyelv intrikus elemei) tárolása. Természetesen vannak olyan feladatok, amelyekhez szükséges a tárolt információknál többet ismerni, ilyen lehet egy *slicing* algoritmus, mely segítségével a felhasználó láthatja, hogy egy változó értékének megváltozása milyen egyéb állapotokat változtató kifejezésekre van hatással. Az ilyen algoritmusok végrehajtásához helyszíni újraelemzést kell végezni, amely megtehető, mivel a CodeCompass számára rendelkezésre áll az eredeti forrásszöveg és annak fordítási információi.

4 A *CodeCompass* képességei

Ebben a fejezetben egy összefoglaló bemutatást adok a CodeCompass azon képességeiről, amelyek a szerver által biztosított felhasználói felületen elérhetőek. A diszkusszió során az egyes nyelv-specifikus elemekre való hivatkozáskor feltételezzük, hogy az elemzett szoftverprojekt C/C++ nyelven íródott – jelenleg ennek a nyelvcsaládnak a legnagyobb a támogatottsága az eszközben, de hasonló lehetőségek érhetőek el Java és Python projektekhez is.

4.1 Keresés

A kódmegértési eszközök alapvető használati esete az, amikor a felhasználó a program egy részét keresi. A keresés célja lehet egy fájl vagy forráskód(részlet) megtalálása. A forráskód elemeinek kereséséhez három lehetőség áll rendelkezésre:

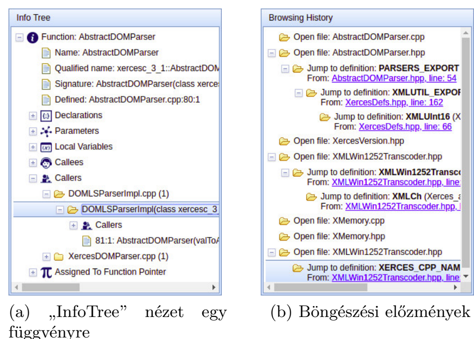
A **teljes, szöveges keresés** esetén a keresési kifejezés szavakból áll, például „`returns an astnode*`". A keresési kifejezés szövegrészletekre illik akkor, ha a kifejezés szavai egymás után mind, és helyes sorrendben szerepelnek. Helyettesítő karakterek (`?`, `*`) használhatóak arra, hogy egy vagy több bármilyen karakterre illeszkedjenek. A keresőkifejezések között logikai operátorokat (`AND`, `OR`, `NOT`) helyezhetünk el, amellyel a részkifejezések összekapcsolhatóak közös keresés érdekében.

Egy magasabb absztrakciós szintet biztosít a **definíciós keresés**, amellyel a forráskód egyes megnevezhető szimbólumaira keresünk. Ennek érdekében *CTags*-szel egy indexet képezünk a forráskódból, amelyen keresztül megkereshetőek változók, függvények, osztályok, makrók, stb. Ez a nyelvi elem keresés azonban felületesebb és független a mély nyelvi elemzéstől és a programegységek keresésétől.

Egy program hibakeresése során előfordulhat, hogy csak valamilyen naplófájlból elérhető információból kell kiindulni. Egy ilyen lehet például a naplóban megjelenő alábbi sor: „`DEBUG INFO: TSTHan: sys_offset=-0.019821, drift_comp=-90.4996, sys_poll=5`". Vegyük észre, hogy a kimenetben futásidőben kiszámított értékek – például időbélyeg – található, tehát egy teljes szöveges keresés nem vezetne eredményre, mivel nagy valószínűséggel a naplóba írás helyén a helyettesítések változókból olvassák a megjelenítésre szánt értéket. A CodeCompass képes egy bolyhos keresést végezni a szöveges kimenetek formázásán a **napló keresés** opció használatával.

4.2 Programegységek adatainak lekérdezése

Amikor a felhasználó megtalálta a keresett programegységet, a következő lépés annak részletes vizsgálata, arról információk kigyűjtése. Ennek megvalósítása érdekében a felhasználó lekérheti az *„Info Tree"* nézetet minden névvel ellátott programegységre. Ebben a fastruktúrában az adott nyelvhez tartozó mély elemző (C/C++ esetén az LLVM/Clang fordítóprogram) által kigyűjtött tudás jelenik meg.



(a) „InfoTree" nézet egy függvényre (b) Böngészési előzmények

1. ábra: Információgyűjtő panelek

Függvények esetén megtekinthetőek azok paraméterei, lokális változói, hívói és hívottjai. A fastruktúra egy érdekessége, hogy a hívók listájának bejárása rekurzívan bővíthető, azaz az éppen vizsgált függvény hívóiból tovább „nyitva" a fa ágait a hívók hívói is láthatóak, elméletben a `main()` függvényig visszamenve. Azonban a gyakorlatban a függvényhívások nem egyszerűek, történhetnek például függvénymutatókon keresztül is. A CodeCompass képes megjeleníteni az összes pontot ahol egy adott függvényre mutatót állítottunk és az ezeken keresztüli eseteket is hívási helyként kezelni, függetlenül attól, hogy a mutatók értéke egy alapvetően futásidejű tulajdonsága a programnak.

Típusok, osztályok esetén az összegyűjtött tudásba tartozik a típusszinonímák (`typedef`) felsorolása, az öröklési hierarchia mutatása, *barát*-osztályok, közvetlen vagy örökölt mezők és metódusok, valamint a típus felhasználási helyei.

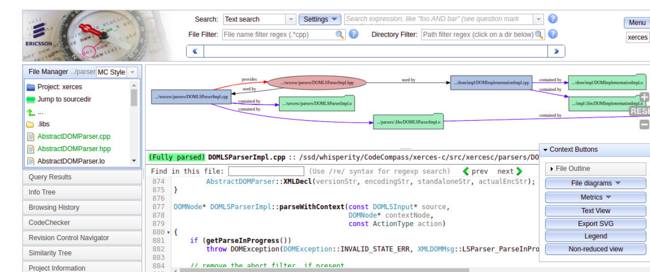
Változók esetén a változók olvasási és írási helyei lehetnek érdekesek. Felsorolási típusokhoz megmutatjuk a felsorolás konstansait, azok numerikus értékével.

4.3 Diagramok

Az emberi felhasználók számára a különböző vizualizációk a legjobb módja egy komplex rendszer megértésének. A CodeCompassban számos szimbólum és fájl finomságú diagram érhető el. Ezek a diagramok gráfokon alapulnak, azaz valamilyen entitás és azok kapcsolatait mutatják be. A megjelenített diagramok interaktívak, a diagramban megjelenő entitásra pozícionálva a kurzort megjelenik a programegység forráskódja, kattintással pedig a diagram újszámítható a kijelölt entitásból indulva.

A *függvényhívás* diagram megmutatja a függvény összes hívóját és hívottját egy gráfban. Az *UML osztály öröklődés* diagram megmutatja egy osztályhoz tartozó teljes öröklési hierarchiát a legfelső bázis és a legalsó leszármazott osztályáig. Elkészítettünk egy *pointer-analízis* diagramot, amely megmutatja a futás közben a memóriába kerülő objektumokat és az azokra potenciálisan ráállított mutatókat. Természetesen az ilyen futásidejű állapottól függni képes információ csak korlátozott mértékben állítható elő statikus analízis segítségével.

Az *interfész diagramot* C/C++ forrásfájlokra futtatva megmutatja, hogy mely fejlécállományok „felhasználtak" vagy „implementáltak" a forrásfájl által. A *„felhasználás"* azt jelenti, hogy az egyik fájl által deklarált szimbólumot a másik fájlban felhasználják. Az *„implementálja"* reláció azt fejezi ki, hogy az egyik fájlban deklarált – és így interfészt biztosító – szimbólumot a másik fájlban definiálják. Ezek a relációk nem csak forrásfájlok, hanem azok csoportjai (mappák) között is értelmezhetőek. Gépi kódra fordított nyelvek esetén a fordítás kimenetei (C++ esetében tárgykódok és binárisok) is elérhetőek. Mivel ismerjük a fordítórendszer információt az elemzés során, ezért ki tudjuk rajzolni, hogy mely forrásfájl mely binárisokba került fordításra.



2. ábra: Interfész diagram

A *CodeBites* nézet a megtekintett forrásfájl egy más szemszögből való vizsgálatát teszi lehetővé. Ebben a nézetben a gráf csúcsai elnevezett szimbólumokat definiáló forráskód-szeletek. A nézet terve mögötti gondolatom az volt, hogy a programozó szeretné megérteni az entitás viselkedését az arra tett fókusz elvesztése nélkül. A nézetben a forráskód elemeire kattintva nem a kattintott elem definíciójára ugrik a fókusz, hanem egy új csúcsként annak kódja megjelenik a gráfban, de a korábban kiválasztott elemek is megmaradnak.

4.4 Verziókövetési rendszerek vizualizációja

A verziókövetési információk vizualizációja egy hasznos segítség a szoftverprojekt fejlődésének megértése szempontjából. A Git *blame* nézete megmutatja a fájlokon sorról sorra az adott sort legutóbb megváltoztató módosítást (*commit*). Az újabb módosítások zöldesebb, a régebbi módosítások pirosasabb színt kapnak. Ez a

nézet jó eszköz annak áttekintésére, hogy egy adott sor miért került a kódba hozzáadásra.

A CodeCompass ezen felül mutatja a Git commitokat egy szűrhető listában azok időbélyege alapján. A szűrés segítségével kiválaszthatóak egy adott személy vagy egy adott kifejezést a leírásukban tartalmazó commitok.

4.5 Metrikák

A CodeCompass megjeleníti a McCabe Ciklomatikus komplexitás metrikát [1], a sorok számosságát, a fájlban talált *Clang* statikus analízis találatok számát minden fájlra, és ezeket összegezve mappákra. A metrikákat *treemap* vizualizációban jelenítem meg, amelyben a téglalapok a mappákat reprezentálják, a méretek és színek pedig a metrikák dimenziójában az egymáshoz való viszonyítást teszik lehetővé.

4.6 Böngészési előzmények

De Alwis és Murphy kutatásukban azt vizsgálták, hogy a programozók miért éreznek tájékozódási zavart az *Eclipse* Java integrált fejlesztői környezet (IDE) használata során [8]. A „vizuális lendület" (*visual momentum*) fogalmát [10] felhasználva megállapítottak három alapvető faktort, amely a tájékozódási zavarhoz vezetett: i) hiányoznak az összekötő kapcsok a navigáció során amikor a programot felfedezik; ii) a fejlesztők nagyon gyakran ugrálnak össze-vissza különböző képernyők és nézetek között, hogy átlássák a szükséges kódrészleteket; és iii) gyakran kajtatnak egyszerre több, egymástól független részfeladat után.

Az első faktor értelmében a programozók egy programhiba vizsgálata során számos forrásfájlt tekintenek meg a hívási lánc vagy egy változó használatainak bejárásakor. Egy ilyen hosszú felfedezőút végén nehéz megjegyezni, hogy miért abban a fájlban ért véget, ahol. A tájékozódási zavar második indoka az Eclipse-ben elszenvedett gyakori nézetek közötti váltás. A harmadik mögöttes indok abból fakad, hogy a program módosítását végző fejlesztők a munkájuk során számos hipotézist vizsgálnak meg, amelyek önmagukban is teljes értékű kódmegértési részfeladatok. A programozók gyakran csak „pihentetik" ezeket a részfeladatokat azok befejezése helyett és átváltanak egy másikra. Egy ilyen eset például amikor megvizsgáljuk, hogy egy függvény visszatérési értéke miként kerül felhasználásra, de közben átváltunk az adott függvény belső viselkedésének megértésére. Megfigyelték, hogy nehéz az embereknek magukat emlékeztetni a félbehagyott részfeladatokra [11].

A CodeCompass biztosít egy *böngészési előzmények* nézetet amelyben feljegyzésre kerülnek – fastruktúra formában ábrázolva a hierarchiát – a forráskódban való navigálás lépései. Egy elágazás a fában egy új részfeladatot reprezentál, a fa csúcsai a fájlokbani ugrásokat reprezentálják, megannotálva annak okával (például „az `init` függvény definíciójára ugrás"). Tehát a korábban említett i) és ii) problémákat ezzel a címkézéssel orvosoltuk. A iii) probléma megoldását a fa szerteágazása teszi lehetővé.

4.7 *CodeChecker* – C/C++ Hibajelentések

A *Clang Static Analyzerben* megtalálható egy fejlett szimbolikus végrehajtással dolgozó analízismotor amely képes programozói hibákat jelezni a kódban. A CodeCompass képes az ilyen találatokat a forráskóddal együtt jelezni, amennyiben a szervert indításkor összekapcsolják egy CodeChecker [12] szerverrel. A megjelenített információk a hibajelzés pozíciója, és a hibát előidézni képes szimbolikus végrehajtási út részletei.

4.8 Névtér- és típuskatalógus

A CodeCompass elemző feldolgozza a *Doxygennel* készített, kódba ágyazott dokumentációs kommenteket, és ezeket is letárolja az adott szimbólum definíciójához. A típus katalógus nézetben megtalálhatóak a projektben definiált típusok a névtereik hierarchikus listájában.

5 Összefoglalás

Bemutattam a CodeCompass kódmegértést támogató eszközt, melynek célja nagyméretű szoftverprojektek megértésének segítése. A CodeCompass tervezése során fő szempont volt a korábbi, hasonló célt szolgáló eszközök gyengeségeinek elkerülése, mivel ezek az eszközök vagy pehelysúlyúak és bár könnyen használhatóak, nem rendelkeznek egy rendes fordítóprogram tudásával, vagy nehézsúlyúak és ezért nehezen kezelhetőek, skálázhatóak, a kliensgépekre települnek. A webalkalmazás-alapú, bővíthető és kiterjeszthető architektúrájának köszönhetően a CodeCompass keretrendszere megfelelő táptalaja lehet a kódmegértésnek, a statikus analízisnek és a szoftvermetrikák platformjának. Az eszközünk hasznosságát a fejlesztési folyamatokban mutatja, hogy számos kezdeti felhasználói visszajelzés érkezett. Az eszköz használati statiszikái azt mutatják, hogy a felhasználók a CodeCompass a klasszikus integrált fejlesztői környezetek és kereszthivatkozást biztosító eszközökkel együttesen használják.

Köszönetnyilvánítás

Ez a cikk az Erasmus+ Kulcsfontosságú Akciók 2 projektcsalád `2017-1-SK01-KA203-035402` számú projektjének – Stratégiai Partnerség a Felsőoktatásért, *„A működő szoftverek összeépíthetőségének, megérthetőségének és helyességének oktatását elősegítését célzó"* `3COWS` (*"Focusing Education on **C**omposability, **C**omprehensibility and **C**orrectness of Working Software"* (`3COWS`)) projekt eredményeként valósult meg.

Felelősség kizárása

E kiadványban közölt tudás, információk és állítások a szerző(k) véleménye(i), meggyőződése(i) és világnézete(i) alapján kerültek megfogalmazásra, eképp nem

feltétlenül tükrözik az Európai Unió hivatalos álláspontját. Sem az Európai Unió intézményei és testületei, sem bármely Uniós intézmények és testületek megbízatásából eljáró személy nem tehető felelőssé a kiadványban szereplő megállapítások felhasználásáért, vagy a felhasználásból eredő bármilyen következményért.

Hivatkozások

1. Thomas J. McCabe, *A Complexity Measure*, IEEE Transactions on Software Engineering: 308–320, December 1976
2. Henderson-Sellers, *Object-Oriented Metrics: Measures of Complexity* Prentice-Hall, 1996, Upper Saddle River, NJ, ISBN-13: 978-0132398725
3. Woboq, `http://woboq.com/codebrowser.html`, 18. 03. 2018
4. OpenGrok, `http://opengrok.github.io/OpenGrok`, 18. 03. 2018
5. CTAGS, `http://ctags.sourceforge.net`, 18. 03. 2018
6. Understand, `http://scitools.com`, 18. 03. 2018
7. CodeSurfer, `http://www.grammatech.com/products/codesurfer`, 18. 03. 2018
8. B. De Alwis and G.C. Murphy, *Using Visual Momentum to Explain Disorientation in the Eclipse IDE*, Proc. IEEE Symp. Visual Languages and Human Centric Computing, pp. 51-54, 2006.
9. E. Baniassad and G. Murphy, "Conceptual Module Querying for Software Engineering," Proc. Int'l Conf. Software Eng., pp. 64-73, 1998.
10. D. D. Woods., *Visual momentum*, A concept to improve the cognitive coupling of person and computer. Int. J. Man-Mach. St., 21:229–244, 1984.
11. D. Herrmann, B. Brubaker, C. Yoder, V. Sheets, and A. Tio. *Devices that remind*, In F. T. Durso et al., editors, Handbook of Applied Cognition, pages 377–407. Wiley, 1999.
12. Dániel Krupp, György Orbán, Gábor Horváth and Bence Babati, *Industrial Experiences with the Clang Static Analysis Toolset*, EuroLLVM 2015 Conference, April 2015