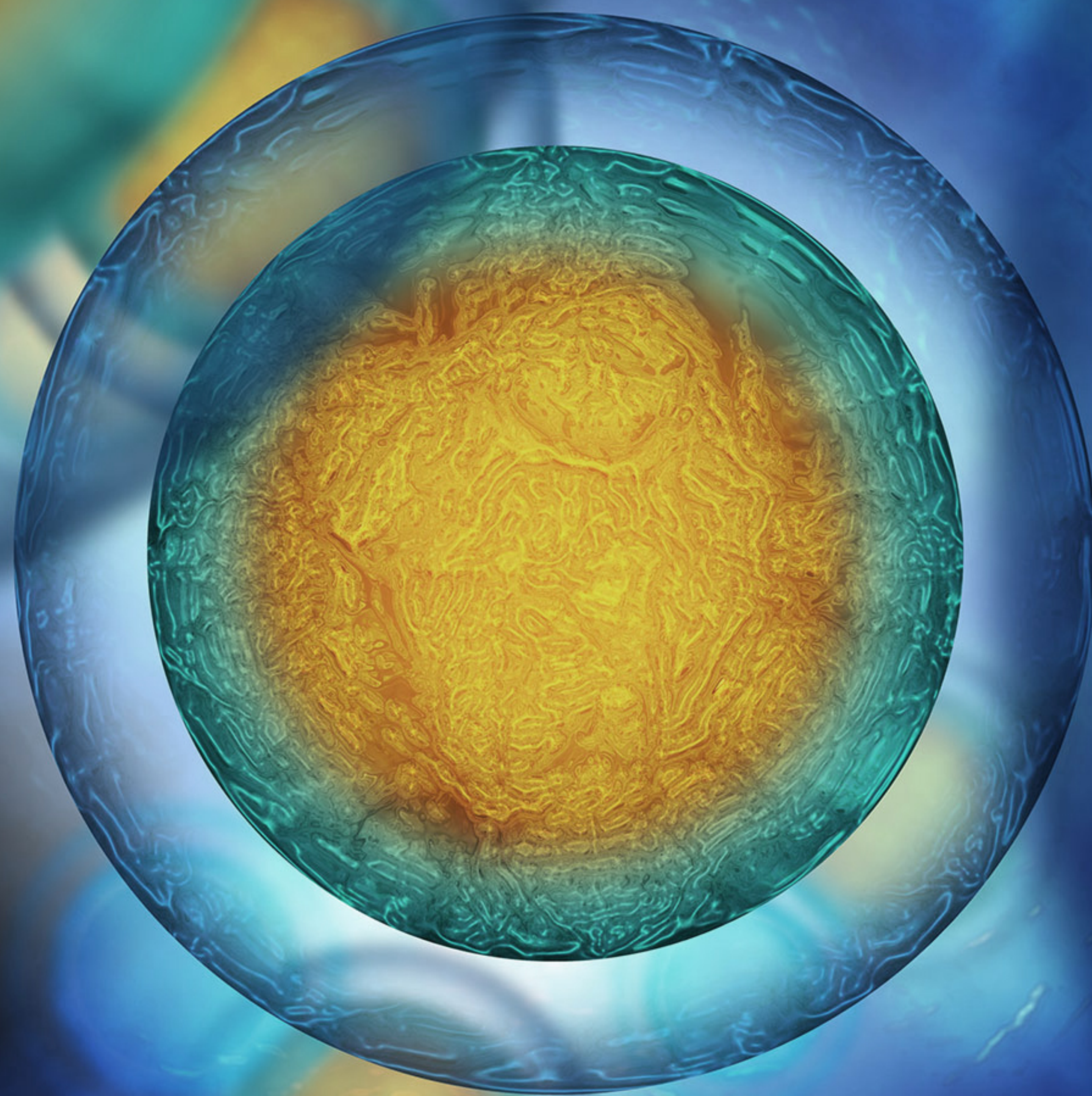
FE3CWS

УЧЕБНИ МАТЕРИАЛИ ЗА ЛЯТНО УЧИЛИЩЕ

Интелектуален резултат 1 на проекта ERASMUS + 2017-1-SK01- KA203-035402



Някои думи по **съдържанието**

- 9 теми, свързани със състава на софтуера, разбирането и коректността
- 10 автори от Южна Корея и 7 европейски университета от Хърватия, Унгария, Холандия, Португалия и Словакия
- Предлага се на 7 езика: английски, унгарски, словашки, хърватски, румънски, български и португалски

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

Трите зимни училища „CO“ (Компонируемост, разбираемост и коректност) (3COWS) е първата интензивна програма за обучаващи се и преподаватели с висше образование, разширяваща общността на Централното европейско функционално програмиране (CEFP) в рамките на проекта ERASMUS + № 2017-1-SK01-KA203-035402 „Фокусиране на образованието върху комбинируемостта, разбираемостта и коректността на работния софтуер“, което се проведе между 22 и 26 януари 2018 г.

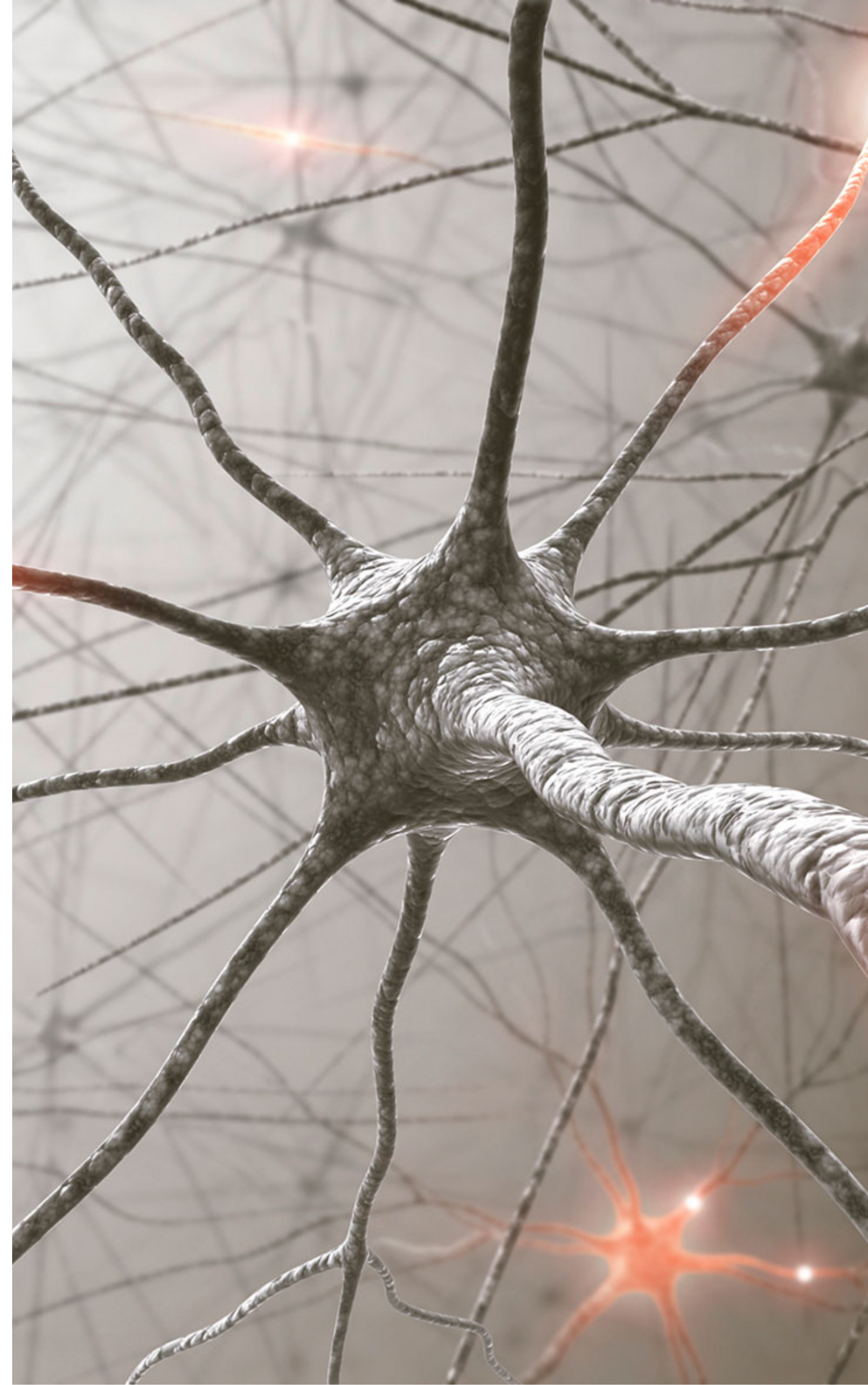
Включеният материал е създаден и представен в рамките на гореспоменатия проект. Тази публикация е печатната версия на интелектуалния продукт O1 на проекта.

© Европейски съюз, 2017-2019 г.

Информацията и възгледите, изложени в тази публикация, са тези на автора (ите) и не отразяват непременно официалното становище на Европейския съюз, институциите и органите на Европейския съюз, както и което и да е лице, действащо от тяхно име, не може да бъде отговорно за използването на съдържащата се тук информация.

Съдържание

1. Научаване на естествен език от машините
2. Анализ на статичния код с CodeChecker
3. Ролята на функционалното програмиране в управлението и оркестрирането на виртуализирани мрежови ресурси
4. Облачни изчисления и функционално програмиране в образованието
5. Модерно типово-безопасно вграждане на атрибутни граматики
6. Колко зелен е процесът Ви?
7. Функционално програмиране на устройства
8. Разбиране на кода с CodeCompass
9. Шаблони за функционално програмиране за HPC



Научаване на естествен език от машините

Изграждането на мислеща машина е голямо предизвикателство. Връзката между езика и ума е интересна за компютърните учени, тъй като се обсъжда в по-широк контекст и от историци, психолози, езиковеди и философи.

Ренфрю [31], например, характеризира човека като символично същество, което счита, че развитието на естествения език е резултат от обмен на символи в общуването. G¨ardenfors [14] твърди, че езиковите понятия имат йерархична структура. Граматичната еволюция [24] и генетичната еволюция на езиците [15] влияят на проблема със структурното разширяване, следователно те са приложими само за прости езици по детерминиран начин. Оказва се, че низове в генетичния алгоритъм представени като стойностите на семантичните заключения са по-обещаващ подход към езиковата еволюция, отколкото директното прилагане на генетичната структура на човешкия мозък чрез аналогия. Също така осъзнахме, че процесът на еволюция протича на различни езикови мета-нива.

Хипотезата на Чомски за съществуването на универсална граматика за всички естествени езици [7] е много интересна и мотивираща. Но може би универсалната граматика не трябва да се разбира като фиксирана и параметризирана граматическа структура, а по-скоро като универсален алгоритъм, т.е. детерминиран напредък на метаезиково ниво, параметриран от параметрите, които представляват смислени, т.е. семантични стойности.

Еделман [12] твърди, че машинният ум има езикова субстанция и е символичен, като в същото време е и изчислителен. От това следва, че умът е не само символичен, но и динамичен процес, който изразява непрекъсната промяна на езика. И накрая, в апликативната универсална граматика на Shaumyans [34] езикът се изразява и по отношение на динамични, дори приложни процеси.

Ползата от теорията на семиотичните езици е, че синтаксисът и семантиката са винаги взаимно обвързани и не могат да се разделят. Разбира се, това се позовава на необходимостта да се вземат предвид семантичните категории, за да се изрази промяната на даден език.

Според нас семиотичната теория на езиците предоставя възможност за крайна еволюция на машинния ум в случай за подмножества на естествените езици и формалните езици, използвани в комуникацията под различни форми. В този смисъл комуникацията между човек и компютър и компютър-компютър на базата на езици може да бъде унифицирана. Отправна точка на тази идея, ограничена до обикновените езици, представяме в [17]. Сега знаем алгоритъма за трансформация на езиковите понятия във вътрешния език на машинния ум и също така сме в състояние да извлечем понятия от този език. Той е аналогия с вътрешния език на човешкото мислене - той е спокоен език зад шумния естествен език. В момента не знаем нито правилата за концептуализация, нито правилата за разсъждения върху концепции, представени в машинния ум. Имаме, обаче, ясна методология на еволюцията на машинния ум, представена от апликативни динамични процеси с висока степен на паралелизъм, които представляват ненужно информацията за езиковата субстанция. Освен това, увеличаването на абстракцията на езиковите понятия намалява броя на метаоперациите и увеличава броя на приложните връзки.

Обсъждаме ефективния алгоритъм на детското мислене и оценяваме скоростта на информационния поток в човешкия мозък до 300 трилиона сигнала в секунда. След това въвеждаме словашка текстова и фонетична граматика, както и метода за превод на текст на глас. После илюстрираме процеса на усвояване на езикови елементи от различни абстракции при четене на кратка извадка от текст и използване на йерархични хеш таблици като модел.

Освен това, оценяваме процеса на придобиване на масивен текст на избрана книга и правим заключението, че придобиването на език с йерархична абстракция дава ненужен график със същия брой последователни и паралелни връзки, без да е необходимо физическо съхранение на данни.

След това използваме граматически подход за разпознаване на комуникационни визуални обекти. Първо трябва да измислим как да опишем обектите и след това можем да приложим метода на абстракция към тези данни. Основно се фокусираме върху 3D описание на обекти с помощта на граматики. В теорията на граматиката тази стъпка се нарича символизация. Символизацията осигурява описание на обекта и осигурява основния абстрактен слой от данни. Както можем да видим, извличането на данни с помощта на функционален език ни позволява да направим абстракция и лесно да обработваме обекти.

Приложимият подход може да бъде използван при езикова обработка, дори когато използваме граматики без контекст. Показваме алгоритъм, който е в състояние да трансформира всяка граматика без контекст във форма на суперкомбинатор. Получената форма зависи от формата на входна граматика, следователно възниква нов проблем: намиране на правилната граматика за въпросната задача.

Показваме набързо получените суперкомбинаторни форми от различни типове граматики и сравняваме техните свойства. Също така, сравняваме ефективността на алгоритъма в представените граматични случаи и показваме, че алгоритъмът ни може да бъде подобрен в случай, че използваме граматики без никакви цикли. Освен това, обсъждаме получените форми на суперкомбинатор по отношение на граматично компресиране и повторна използваемост на елементите, които са крайния резултат от обработката на текстове в по-голям мащаб като входни примерни данни.

Източници

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

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

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

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

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

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

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

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

Анализ на статичния код с CodeChecker

Преглед

Символичното изпълнение [4] е популярна техника за статичен анализ, използвана както в проверката на програмата, така и в инструментите за откриване на грешки. Той работи, като интерпретира кода, въвежда символ за всяка неизвестна по време на компилиране стойност (например, въведени от потребителя входове) и извършва изчисления символично. Двигателят за анализ се стреми да изследва няколко пътища за изпълнение едновременно, въпреки че проверката на всички пътища е неразрешим проблем поради огромния брой възможности.

Въпреки че съществува богата литература относно инструментите за проверка на програмите, инструментите за намиране на грешки обикновено трябва да се намерят за проучвания на отделни техники [1]. В тази статия не само обсъждаме отделните методи, но и как тези решения си взаимодействат и се подсилват взаимно, създавайки система, която е по-голяма от сбора на неговите части. Ние се съсредоточаваме върху инструмента за намиране на грешки, наречен Clang Static Analyzer [2] (наричан по-долу Анализатор), и изградената около него инфраструктура, наречена CodeChecker [3]. Акцентът е върху постигането на мащабируемост от край до край. Това включва времето за изпълнение и потреблението на паметта на анализа, представянето на грешки на потребителите, автоматично фалшиво положително потискане, инкрементален анализ, откриване на образи в резултатите и използване в непрекъснати интеграционни контури. Освен това, очертаваме бъдещи посоки и открити проблеми, свързани с тези инструменти.

Въпреки че анализаторът може да борави само с C / C + + / Objective-C код, въведените в този документ техники са независими от езика и са приложими за други подобни инструменти за статичен анализ.

Статичен анализатор на Кланг

Обобщаваме работния механизъм на символичното изпълнение и неговото прилагане в анализатора. Обсъждаме нейното представяне на паметта [6], обработката на връзките между стойности и местоположения в паметта и неговото представяне на състояния, специфични за проверка (където с проверка имаме предвид един модул на анализатора, написан, за да намерим един конкретен тип грешка). Въвеждаме и концепцията за символични изчисления. Изборът на представителства, използвани от анализатора, играе решаваща роля за превръщането на мащабен софтуер в анализ.

Тъй като проверката на всички възможни пътища за изпълнение не е възможна за разумен период от време, трябва да въведем концепцията на бюджета за анализ: прогноза за периода от време, който можем да си позволим да анализираме даден фрагмент от код. Целта е да се намерят възможно най-много грешки с ниска фалшиво положителна честота. Показваме как анализаторът дава приоритет на по-интересните пътища за анализ и как елиминира невъзможните пътища по ефективен начин, използвайки многостранно решаване на ограничения [5].

Анализаторът използва и редица евристики за автоматично потискане на отчетите, които е вероятно да са невярно положителни.

Когато се намери грешка, съответният път и набор от ограничения са полезни за разбиране на проблема. Въпреки това е непрактично да се представя цялата тази информация на потребителя. Показваме как анализаторът се стреми да представи на потребителя кратък, но проницателен доклад за грешка, който минимизира времето за коригиране на споменатата грешка.

CODECHECKER

Дефинираме мащабируемостта на статичния анализ не само по отношение на ефективното използване на изчислителните ресурси, но и по отношение на ефективното използване на човешките ресурси като времето на разработчиците. CodeChecker е инструмент, предназначен да улесни интегрирането на анализатора и други подобни инструменти за статичен анализ в изграждане на системи и непрекъснати интеграционни контури. Това също е пълноценна система за управление на грешки, която следи грешките, открити от тези инструменти.
Като се има предвид ограниченото време на софтуерните инженери и многобройните отчети около изработването голям софтуер, важно е първоначално да се оценят докладите с най-добра възвръщаемост на инвестицията.
CodeChecker също поддържа диференциален анализ, който не позволява на програмистите да въвеждат нови грешки, без да изискват предварително да коригират всички наследени отчети.

резюме

В тази статия обобщаваме опита, събран, докато допринасяме за най-модерните статични анализатори Clang и продуктите CodeChecker. Надяваме се, че той ще се окаже полезен ресурс за всеки, който реши да работи върху инструменти за статичен анализ.

Препратки

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

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

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

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

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

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

Ролята на функционалното програмиране в управлението и оркестрирането на виртуализирани мрежови ресурси

Network Functions Virtualization, NFV (виртуализацията на мрежовите функции) е нова парадигма за промяна на начина на изграждане и експлоатация на мрежите. Въвеждането на софтуер за отделяне (decoupling software) в мрежовите ресурси чрез виртуализиращ слой въвежда необходимост от разработване на набор от функции за управление и оркестрация на NFV (MANO). Фокусът пада върху координирането на управленческите функции, реализирани в различни функционални блокове за се осигури надеждна работа за MANO функции, работещи в разпределени среди. Предизвикателствата са илюстрирани чрез практически пример за виртуалната технология Open Stack и за проблемите, възникващи в от телекомуникационната индустрия.

Въведение

Целта на лекцията е да представи нови концепции като NFV, които в момента се реализират в сложни софтуерни системи и мрежи. Освен това се стреми да обясни новите появяващи се предизвикателства и да покаже на студентите как да се справят с тях, използвайки техники на програмиране за координация на функциите, както и за управление и оркестрация на виртуализирани мрежови ресурси, работещи в разпределени среди.

Сложни системи

Фокусът на тези лекции е върху теорията на сложните системи, по-специално на сложните софтуерни системи и сложните мрежи. Следователно лекциите започват с леко въведение в теорията, като внимателно се разглеждат проблеми и предизвикателства в рамките на развитието на мрежите и софтуерните системи.

Виртуализацията е парадигма, която често се използва за управление на сложни софтуерни системи. Това предполага въвеждане на нов абстрактен слой, виртуално изграждане на системен слой и неговите функции, което избягва въвеждането на зависимост между системните слоеве.

Функции за управление и оркестрация

В телекомуникационните мрежи се въвежда нова парадигма, наречена виртуализация на мрежовите функции (NFV), която отделя мрежовата функция от физическите мрежови ресурси чрез нов слой за виртуализация [2]. Това, обаче, въвежда необходимост от разработване на набор от функции за управление и оркестрация на NFV (MANO). За тази цел е определена специална работна група в рамките на Европейския институт за стандарти в областта на телекомуникацията (European Telecommunications Standards Institute, ETSI). Мрежовата функция за управление на виртуализацията и оркестрационната архитектурна рамка е дефинирана в [1]. В тези лекции се фокусираме върху функциите за управление и оркестрация, реализирани в различни функционални блокове, за да се постигне надеждна работа за функциите в разпределени среди.

Примери и упражнения

Тази част от лекциите предоставят въведение в темата, като целта е да се обяснят проблемите и принципите, методите и техниките, използвани за тяхното решаване. Работните примери и упражнения служат на студентите като учебени материали, от който могат да научат как

да използват функционалното програмиране за ефективно и ефикасно координиране на управленските и оркестрационните функции в разпределени сложни системи с NFV.

Методите и техниките, обяснени в лекциите и приложени към проблемите с управлението и организацията на виртуализацията на мрежата, вече съществуват. Поради тази причина не оспорваме оригиналността на идеята. Целта е лекциите по-скоро да послужат като учебен материал за улеснение на работата с тези методи.

Проблемите и предизвикателствата при координацията на функциите за управление и оркестрация се решават чрез платформата OpenStack [3]. Това е облачна операционна система с отворен код, която интегрира колекция от софтуерни модули, необходими за осигуряване на слоест модел в облачни компютри. Подобна технология е необходима при справяне с проблеми, произтичащи от парадигмата за виртуализация в настоящите мрежи, Студентите, които разбират от работа с OpenStack, ще могат да прехвърлят знанията си в други вече съществуващи технологии с идентична или подобна цел.

Предизвикателствата, произтичащи от новите мрежови парадигми, както и техните решения са илюстрирани чрез практически примери, използващи виртуалната технология на OpenStack и са вдъхновени от проблемите от телекомуникационната индустрия. Всички примери и упражнения са разработени с OpenStack.

Източници

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

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

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

Облачни изчисления и функционално програмиране в образованието

Облачните изчисления се превръщат в ключова технология днес и следователно стават част от много учебни програми по компютърни науки. Ключова концепция в дизайна на приложения се оказват микроуслугите (microservices). Функционалната декомпозиция, присъща за микроуслигите, би могло да се обслужва от безсървърни платформи, като AWS Lambda.

Микроуслуги

Все повече специалисти препоръчват при проектиране на облачни приложения да се използва архитектура на микроуслуга [1, 2]. Обобщено дефиниран, архитектурният стил на микроуслугата представлява общ набор от характеристики: автоматизирано пускане на приложението, smart endpoints dumb pipes и децентрализиран контрол на данните [2]. Неотдавнашните усилия за преминаването на традиционните облачни приложения към архитектура на микроуслуги напомнят за повишената сложност при управлението на много на брой, макар и малки, услуги за композиране [3]. Тук е важно да отбележим, че ще се съсредоточим върху оркестрацията като основа на бизнес логиката при прецизната композиция на облачните услуги. Дали управлението ще е оркестрирано (централен компонент, управляващ изпълнението) или хореографирано (всяка услуга действа независимо) се свежда до това дали желаното приложение се нуждае от синхронен контрол или може да функционира под асинхронен.

Въпреки това, като се има предвид, че гъвкавостта и скалируемостта са силно желани качества, за предпочитане се оказва хореографираното управление на приложения [1].

Въпреки това, нефункционалните аспекти на микроуслугите, като например времето за изпълнение, играят важна роля при внедряването на приложението на пазара. Поради тази причина планираме да запознаем студентите с концепцията за профилиране на резултатите в облака в контекста на AWS Lambda. Освен това допълваме функционалната парадигма на AWS Lambda, като използваме рамка на Haskell за контрол на експериментите.

Облачни изчисления, ориентирани към потребителя

Важен аспект в областта на облачните изчисления е подпомагането на потребителите при вземане на решения. Такива решения се отнасят до следните въпроси:

- Как се върви приложението на виртуализираните ресурси?

- Колко виртуални ресурси от какъв тип облачен доставчик трябва да бъдат придобити за внедряване на приложения?

- Колко време ще отнеме? Колко ще струва?

Тези въпроси обикновено се определят като проблеми с планирането, които работят с предположението, че няма предварителни познания за приложението. Основните изисквания са приложението да е успешно внедрено, а разходите да са сведени до минимум.

Архитектурата на планер (scheduler) за облачно приложение

Планерът на BaTS [4] е разработен, за да помогне на потребителите при внедряване на приложенията им в облака. За целите е необходим подход за самопланиране, който редовно проверява напредъка на внедряването. В първия етап, BaTS събира статистически данни от проби, които се заменят. Тук е необходима само малка извадка (30-50 задачи), за да се изчисли средната стойност и стандартното отклонение на изпълнението на задачите за различни облачни решения. След това модулът за оценка на бюджета изпълнява линейна регресия, за да оптимизира фазата.

Използване на методологията BaTS за лека виртуализация

Запознаваме студентите с проблема за подпомагането на собствениците на приложения, които искат да изберат най-добрия избор по отношение на леката виртуализация, когато внедряват приложението си като набор от микроуслуги.

AWS Lambda

AWS Lambda е изключително лек виртуализиран изчислителен ресурс, който се предлага от Amazon. Прецизността на детайлите му е на функционално ниво и препоръчителното време за изпълнение на задачата е не повече от няколко секунди. Освен това, ресурсът, приема неблокиращо поведение. Има 46 вида AWS Lambda с цени в евро на GB * в сек.

Реализация на AWS API Haskell

Стремим се, въз основа на обширно приложение на Haskell при реализация на AWS API [5], да копираме методологията за оценка BaTS върху леко виртуализирани ресурси.

Практическа работа

Инструктираме студентите да проверят връзката между ефективността на работа на фабриката при различни видове ламбда, като използват проби и линейна регресия, за да начертаят пропускателната способност спрямо кривите, отговарящи за на ефективността на цените. Обучаемите трябва да обмислят кой тип има най-добра производителност за, съответно, най-ниска цена. На следващо място, те трябва да определят как ефикасно да намерят най-печелившата комбинация: най-малък разход за най-добра работа.

Benchmarking AWS Lambda

Функциите AWS Lambda работят в виртуализирана среда, подобна на контейнер. Известно е, че количеството изчислени ресурси, разпределени за функциите, е пропорционално на заявената от потребителя DRAM памет. За да определим как функциите на Lambda могат да обработват различни натоварвания, можем да сравним всеки техен изчислителен ресурс независимо: CPU, честотна лента на паметта, честотна лента на I / O и честотна лента и латентност. Такива microbenchmarks могат да бъдат изпълнени чрез стартиране в рамките на функциите. Добре познати интензивни натоварвания могат да се получат при изчисленията, паметта, I / O и мрежата. Примери за това са изчисляване на първите N прости числа, изпълнение на benchmark на потока, показване на benchmark на показателя за йонна зона, както и четене или запис на данни в AWS S3.

Предаването на установени методологии е ключово за образованието. Като бъдеща работа бихме искали да подкрепим Haskell AWS Lambda, функции, внедрени чрез реализацията на Haskell AWS API.

Източници

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

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

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

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

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

Модерно типово-безопасно вграждане на атрибутни граматики

Атрибутните граматики са мощен декларативен формализъм за прилагане и взимане на решения в програми, които, по дизайн и за наше удобство, са модулни. Въпреки че един напълно оборудван компилатор на граматики може да бъде съобразен с конкретни нужди, прилагането му е специфично, а поддръжка му е основно предизвикателство. Всъщност, поддръжката на традиционна система за атрибутивни граматики изисква толкова голямо усилие, че повечето системи, предложени в миналото, вече не са активни. Нашият подход при прилагането на атрибутивните граматики е да ги пишем като first class citizens в модерен функционален език за програмиране. Ние подобряваме атрибутната граматика с вече вграден zipper, като я правим невлияеща (т.е. не се налагат промени в дефинираните от потребителя типове данни) и типово-безопасна. Освен това, генерираме по-ясен синтаксис, като използваме съвременни разширения в Haskell. Вярваме, че нашето вграждане може да бъде използвано на практика за прилагане на елегантни, ефективни и модулни решения на предизвикателствата в реалния живот.

Въведение

Атрибутните граматики (АГ) са декларативен формализъм, предложен от Кнут [7] в края на 60-те години и позволява да се разсъждава върху и да се прилагат програми по модулен и удобен начин. За да дефинира семантиката на даден език, АГ разчита на безконтекстна граматика да определи синтаксиса на езика и атрибутите, свързани с граматичната му структура. АГ се използват на практика за конкретизиране на реални езици за програмиране, като например Haskell [2], мощни печатащи алгоритми [16], deforestation техники [4] и мощни видове системи [11].

При програмиране с АГ се постига модулност поради възможността да се дефинират и използват различни аспекти на изчисленията като отделни атрибути. Атрибутите са отличаващи се изчислителни единици, обикновено доста прости и модулни, които могат да бъдат комбинирани в обмислени решения на сложни проблеми в програмирането. Те могат също да бъдат анализирани, грешките да бъдат отстранени и да се поддържат независимо, което облекчава софтуерното развитие.

АГ се оказват особено полезни за определяне на изчисления върху дървета: като се има предвид едно дърво и няколко AG системи като [14, 3, 8, 17] приемат спецификации за това, кои стойности или атрибути трябва да бъдат изчислени и да ги изчислят. Усилията вложени в проектирането и създаването на кода, подобряването и поддръжката на тези AG системи, обаче, са огромни, което често представлява пречка за постигане на успеха, който заслужават.

Все по-популярен алтернативен подход за използването на АГ зависи от вграждането им като first-class citizens в езици за програмиране с общо предназначение [12, 9, 13, 15, 18, 1]. По този начин, чрез хостването им в най-съвременните езици за програмиране се избягва проблема от имплементиране на изцяло нов език и свързана система. Така се използват съвременните конструкции и инфраструктура, които вече са осигурени от тези езици и се фокусира върху особеностите на специфичния за домейна език, който се разработва.

Functional zipper [6] е мощна абстракция, която значително опростява прилагането на алгоритми за преминаване (traversal algorithms), които изпълняват много локални актуализации. Functional zippers се прилагат успешно за вграждане на AG в Haskell [9, 10]. Въпреки елегантността си, това решение имаше основен недостатък, който попречи на реалните приложения: атрибутите не бяха кеширани, а по-скоро многократно преизчислени, което силно навреди на изпълнението. Този недостатък беше отстранен скоро [5] и заменен с друг: подходът стана натрапчив, т.е. за да се извлекат ползи от вградените потребителски структури от данни, те трябва да бъдат коригирани.

В този документ представяме алтернативен механизъм за кеширане на атрибути, базиран на самоорганизираща се безкрайна мрежа. Тази графика се намира над алгебрични типове данни (ADT), които потребителя определя и показва огледално структурата й. Самият дефиниран тип данни остава неизползван. След това, вграждането се базира на два (а не един) кохерентни zippers, които паралелно обикалят структурите на данните. Освен, че не е натрапчиво, нашето решение е напълно типово-безопасно. Съвременните разширения на Haskell, като ConstraintKinds, позволяват да разпространяваме ограниченията надолу в ADT, като напълно елиминираме проблеми от типа време за изпълнение, присъщи за предишните версии. Друго допълнително предимство при използването на съвременни функции на Haskell е по-чист синтаксис с по-малко код, генериран с помощта на Template Haskell.

Източници

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Колко зелен е процесът Ви?

Подобно на Био продуктите, светът се развива, за да се превърне в природосъобразна екосистема. Зелената инициатива определя две основни цели: намаляване на потреблението на енергия и използване на основни природни източници при производството на електрическа енергия.

Едно от предизвикателствата пред производителите на батерии е колко дълго може да работи, без да се зарежда. Има и много други предизвикателства, като, например, размерът, който значително влияе върху формата и теглото на устройството. Батерията се счита за малко по-лека в сравнение с устройството, което се нуждае от батерия, за да работи. Предизвикателството тук е как да се намалят размерите и теглото и същевременно да се генерира висока ефективност по отношение на времето на работа на мобилното устройство, без да се заплаща допълнително.

На върха на това хардуерно предизвикателство, съществува софтуерно предизвикателство: самата програма трябва да е икономична откъм разход на енергия. Постигането на това, без да се ограничава потребителското изживяване, в днешно време се счита за скрита, но важна цел при разработването на софтуер за преносими устройства [1]. Тази цел обикновено се появява, когато свързаните изисквания се променят от неизказан към важен въпрос.

Имайки предвид, че консумацията на енергия на всяко мобилно устройство се влияе от работещите приложения, честотата на използване на услугите и индивидуалните специфики на потребителя, разработването на софтуер за такива устройства се е превръща в предизвикателство [4].

Може да се каже, че предизвикателството за разработка на софтуер винаги е едно и също, но тук трябва да посочим „мобилността“ като ключова системна характеристика. Състоянието на батерията също определя колко добре работи системата, поради конфигурация на ниво операционна система - добре позната като „режим за запазване на батерията“.

Още по-трудно е, ако някой (в нашия случай учителят) трябва да подготви учениците за подобни предизвикателства [6]. Всички вече известни „най-добри практики“ и „съвети за пестене на енергия“ трябва да бъдат представени в контекст, който да е лесно разбираем за студентите. Това може да стане чрез позициониране на концепциите в известна среда, като тестване на софтуер и автоматизация на тестовете [2]. Това са целите на този урок и предстоящите раздели, като се започва с предложението, последвано от примери и се завършва с допълнителни съвети за подобряване.

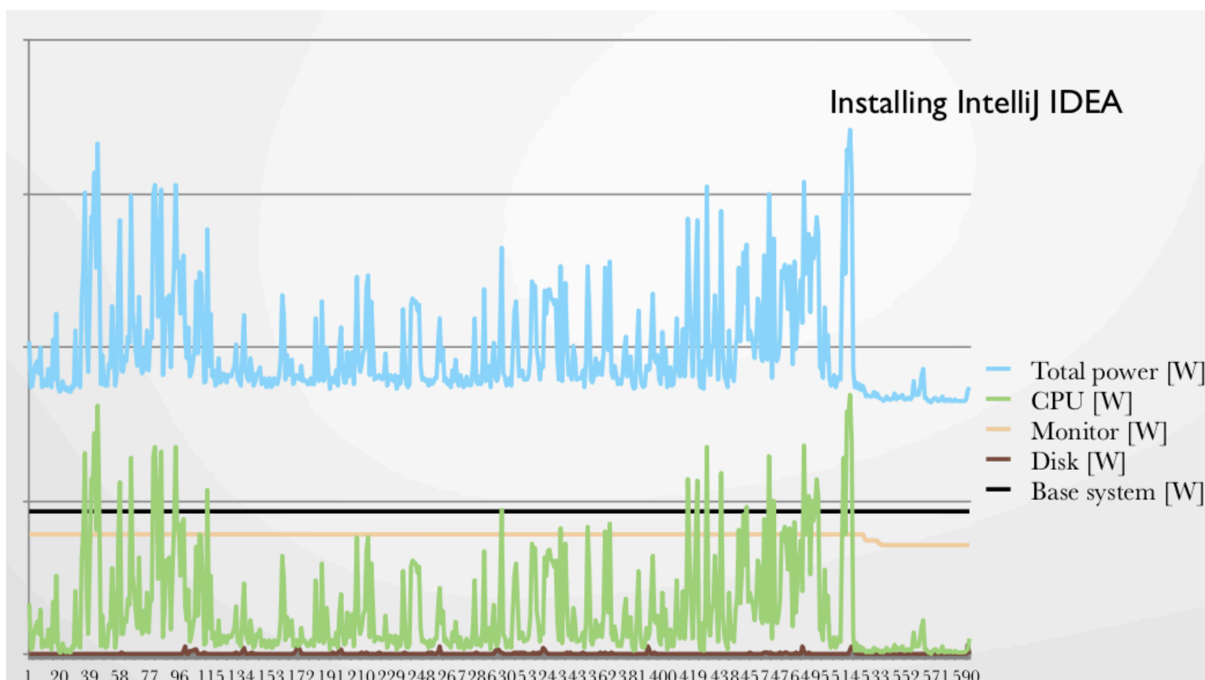
Основният акцент пада върху консумацията на енергия на работещия софтуер и процеса на развитие, при който всяка фаза има значителна роля.

При всяко разработване на софтуер се изразходва енергия за анализиране на проблеми, изграждане и оценка на кода [3]. Софтуерните или хардуерните инструменти трябва да се използват за осъществяване на мониторинг на потреблението на енергия за програми, работещи в основата на операционната система. Обичайният сценарий за използване е при наблюдение на потреблението на енергия на избрания софтуер [5], но също така ще разгледаме възможността да използваме тези инструменти, за да измерим колко зелен е процесът, който създава окончателната (ите) версия (и) на работещия софтуер.

Примерите обхващат различни ситуации. Започваме с енергийно профилирана на третичен софтуер за употреба в конкретни ситуации, посочваме основните характеристики (предимства и недостатъци) на съществуващите инструменти. По време на тестването, представяме типично използване на разработвания софтуер за профилиране на разхода на енергия.

The energy-measured development game

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



Последният пример, който представяме е способността за разширим приложението на измервателния подход от профилиране на фрагмент от код или цяло приложение, до анализ на потреблението на енергия на вериги от инструменти. Това представя общ подход за енергийно профилиране, което има за цел да замени въпросителния знак на заглавието на урока с точка, която категоризира оценката на резултата за всеки конкретен пример, разгледан в урока.

References

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

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

[3] M. Santos, J. Saraiva, Z. Porkolab, D. Krupp: Energy Consumption Measurement of C/C++ Programs Using Clang Tooling, in Proceedings of the SQAMIA 2017: 6th Workshop of Software Quality, Analysis, Monitoring, Improvement, and 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] J.Saraiva,M.Couto,Cs.Szabo,D.Novak:TowardsEnergy-AwareCodingPractices for Android, Acta Electrotechnica et Informatica, Vol. 18, No. 1, 2018, pp. 19–25. https://doi.org/10.15546/aeei-2018-0003

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

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

Функционално програмиране на устройства

Бележка

Тази статия представлява разширена версия на нашия RWDSL18 принос [2]. В настоящия труд ще използваме същия DSL, като са направени само някои минимални разширения и подобрения. Фокусът на статията пада върху това как mTask DSL може да се използва за програмиране на IoT и ще разгледа симулатор на високо ниво за mTask програми като iTask програма.

Въведение

Днес, много устройства са оборудвани с обикновен микропроцесор, който контролира работата им. Типични примери са термостати, електрически крушки, електрически контакти, противопожарни аларми, отварящи се врати и т.н. Когато тези устройства комуникират помежду си или чрез някакъв отдалечен компютър, се казва, че са част от Интернет на нещата (IoT). Микрокомпютрите в тези устройства са много достъпни и присъстват във всички сфери от живота. Скъпите устройства като автомобили и апарати с много сложни функции са оборудвани с вграден компютър и подходящ софтуер. За повечето малки и сравнително евтини IoT устройства такъв вграден компютър е твърде скъп или консумира твърде много енергия; за работата на софтуера се използва семпъл и евтин микропроцесор. Тези системи имат много ограничена изчислителна мощност и памет, обикновено от 30 KB до 4 MB флаш памет за съхранение на програмата. Животът на тази памет е ограничен до 1000 цикъла на запис. За да запазват променливи, heap и stack системите разполагат с 2 до 40 KB RAM.

Поради ограничената скорост на процесора и паметта се изключват използването на операционна система. Апаратът просто изпълнява програмата, контролираща устройството, дори програмите за управление на IoT устройства да се състоят от само няколко задачи.

Например, за да проверите състоянието на бутон е необходимо десет пъти в секунда, да актуализирате дисплея всяка секунда, да измервате температурата два пъти в минута и да включвате отоплението след поне пет минути, освен ако бутона не е натиснат по-рано. Поради различните времеви рамки и зависимостите на тези задачи, контролната програма има тенденция да стане доста объркана, независимо от това на кой език за програмиране е написана. Освен това, устройствата на IoT изпълняват отделни програми за останалата част от приложението в IoT и комуникацията, като използват множество протоколи. Това прави разработването и поддръжката на IoT приложения сложни и предразположени към грешки.

Програмиране ориентирано към задачата, TOP, предлага леки напътствия, които лесно могат да бъдат използвани за съставяне на по-сложни задачи. Задачите се оценяват стъпка по стъпка и могат да проверят текущата стойност на други задачи след такава стъпка. TOP първо се прилага в системата iTask [4, 5], вградена в Clean [6]. В системата iTask примитивните задачи събират информация чрез автоматично генерирана уеб форма или чрез събиране на данни от други програми и бази данни. Мощен набор от комбинатори се използва за съставяне на по-сложни задачи. В тази разработка показваме, че TOP е много подходящ за програмиране на IoT устройства. Примитивните задачи доставят текущата стойност на входовете и сензорите. Използват се конструктори за комбиниране на задачи с по-сложни задачи, което е много подобно на системата iTask.

IoT устройствата обикновено имат слабо зависими задачи, които контролират сензорите, изпълнителните механизми и комуникацията между устройствата. Програмирането с TOP предлага разработване на кратки програми. Изпълнението на тези задачи в ограниченията на малки микроконтролери с много ограничена процесорна мощност и малко килобайти RAM памет заслужава внимание. Поради сериозните ограничения на използваните микроконтролери, не можем да прехвърлим системата iTask към IoT устройствата, тъй като една типична iTask програма изисква около 100 MB място за heap. Дефинираме специфичен вграден домейн език, eDSL, наречен mTask за IoT устройствата. Този eDSL е вграден в iTask системата, тъй като планираме да направим тези TOP езици напълно оперативно съвместими.

Приносът на тази статия е:

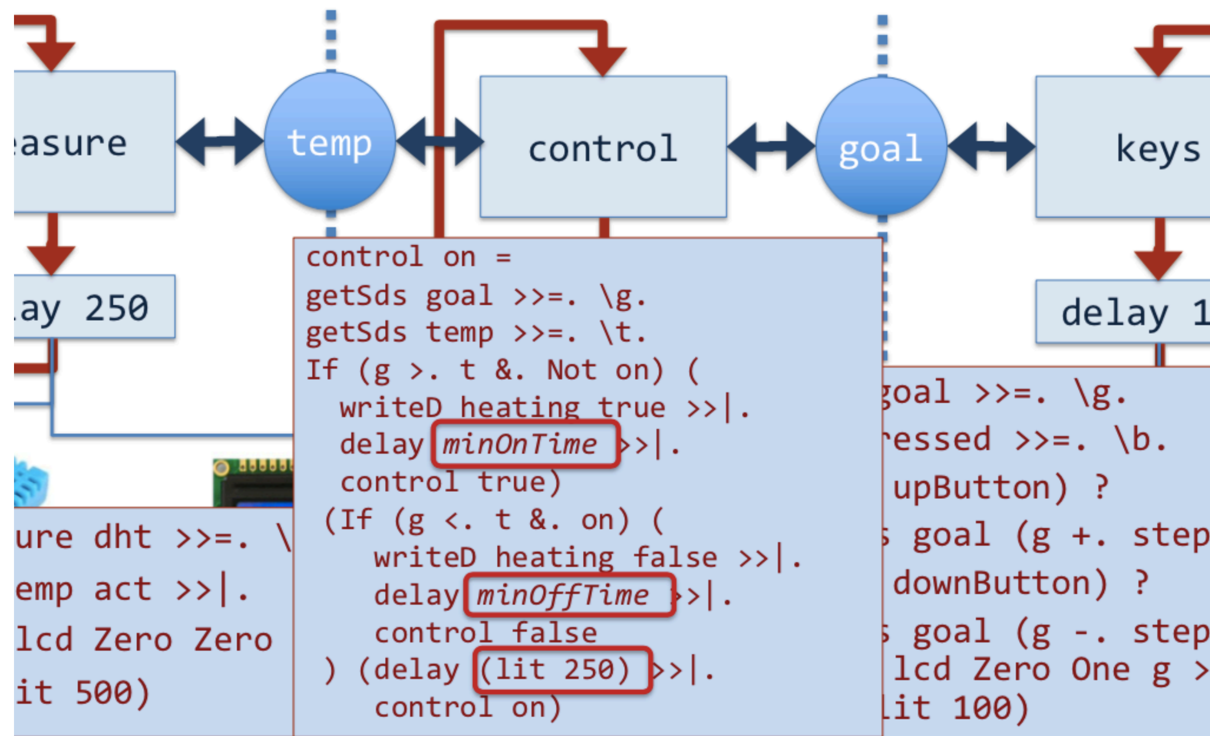
- Този документ представя функционален език за програмиране, базиран на задачи, за устройства с IoT. В сравнение с предишния ни език за микропроцесорно програмиране [3] императивният периферен контрол се заменя с референтни прозрачни конструкции.
- Демонстрираме как се разработва функционален, разширим, с опция за Multi-view, типово-безопасен и вграден DSL. Това е безтагов eDSL [1].
- Генерираният код работи на малки и бавни устройства, както и на по-големи машини и симулаторна машина.
- Понеже Arduino C ++ се използва като междинен език, функционалният eDSL работи на много различни микроконтролери.
- Високото ниво на симулацията на mTask програми в iTask предлага възможност за преглед на резултата от eDSL програмата и за управление на симулираната среда, за да експериментира със специфичната работа на софтуера. В ситуация от реалния живот при такъв тип симулатор е много по-лесно да се манипулират времето и сензорите, отколкото настройките. Можем, например, да променим температурата, отчетена от сензор с натискане на бутон, вместо физически да изложим IoT устройството на тези температури.

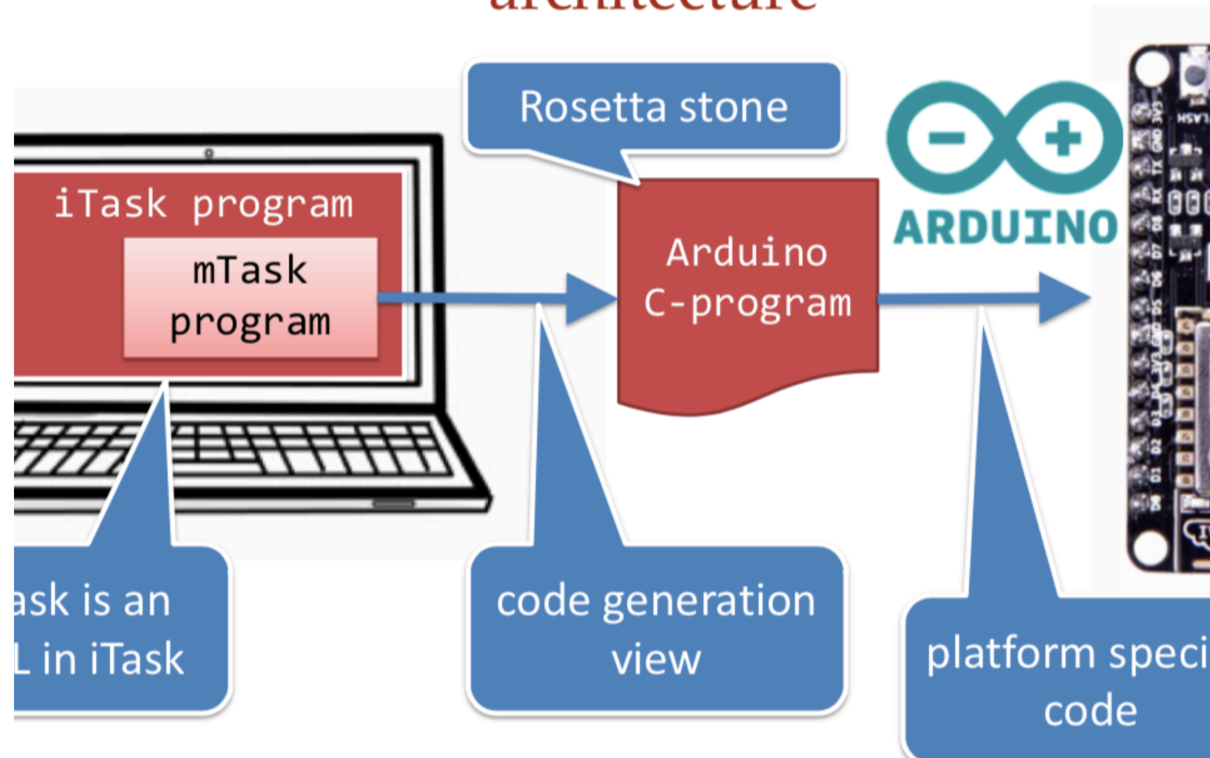
Източници

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

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

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





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

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

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

Разбиране на кода с CODECOMPASS

преглед

Разработването на програмен код и поддръжката му са два отделни етапа с различни характеристики, поради което в двата процеса на работа се изискват различни инструменти. По време на разработката основно пишем нов код, който изисква поддръжка на инструменти за завършване на кода, проверка на скоби и т.н. и обикновено само няколко файла участват повече или по-малко на същото ниво на абстракция. По време на поддръжката, основно четем и разглеждаме съществуващия код, който е съставен от голям брой модули и файлове на различни нива на абстракция [1]. В процеса на разработката целите са ясни, за разлика от разбирането на кода, където задачата е да се възстанови първоначалната цел на определени фрагменти от дадения код.

В индустриалната сфера [2] един проект може да съдържа милиони редове от код. За големи системи, съществуващи от много време, където кодовата база от десетилетия се разработва и поддържа от различни хора, първоначалните цели на продукта се загубват, документацията е недостоверна или дори липсва, а единствената достоверна информация е самият код. Разбирането на такива големи софтуерни системи е от съществено значение, но е и често много предизвикателна задача. Това означава, че има необходимост от поддръжка на софтуерни инструменти [3].

При запознаване с не позат програмен код, първата стъпка е да се намерят съответните части на системата. Този процес изисква бърза локализация на функциите, което се базира на позната информация за обекти, получена при размяна на съобщения или други ресурси. Следващата стъпка е да се разширят познанията за системата чрез диаграми, функционални вериги за

повикване и т.н. На края проверяваме придобитите знания чрез съобщения за контрол на версиите, архитектурна информация и препратки за свързани модули.

CODECOMPASS

CodeCompass [4, 5] е софтуерна рамка с отворен код, създадена за разбиране на програмен код. Тя осигурява гъвкава архитектура, която позволява добавянето на разнообразни инструменти за анализ, различни визуализации, колекционери на информация, показатели [6] и т.н. Най-важната цел относно дизайна на CodeCompass беше да се разработи, софтуера който да служи за големи по мащаб индустриални проекти.

Първата стъпка е продуктът да бъде анализиран: цялата информация се събира и съхранява в база данни, която след това позволява на обслужващия слой (service layer) да осигури необходимите визуализации. За бързо търсене CodeCompass използва индексиране на текстове, което в изходния код води до независима от програмния език навигация. Тъй като основната цел е да се даде точна информация за езиковите елементи, идентифицирането на символите по име не е достатъчно. Използваме компилаторната инфраструктура LLVM, за да идентифицираме прецизно символите и да разпознава познатите единици, използвайки абстрактното синтактично дърво. Анализаторите на езици разширява възможностите на CodeCompass. Най-поддържаните езици в CodeCompass са C / C ++, а Java и Python също се прилагат от части.

Освен посочените символи, в базата данни се съхранява и допълнителна информация, като например връзки между AST възли (извиквания на функции, наследяване) и файлове (връзка към доставчика на интерфейс, включване и т.н.). Те се използват за показване на системата на архитектурно ниво въз основа на използваните символи [7].

Кодовата база не е единственият източник на документации. Съобщенията за ангажиране на система за контрол на версиите също съдържат информация, която е важна, за да се разбере защо някои промени са възникнали в дадения модул. CodeCompass също чете базата на Git, ако съществува такава. CodeCompass е оборудван и с разширени функционалности. Той може да покаже генерираните от компилатора функции, които липсват от източника.

Pointer анализът помага да се разбере кои променливи се отнасят към един и същ обект. Можем да проверяваме функцията за връзките на повикванията, дори ако те са направени чрез виртуална функция или pointer функция.

Резюме

Разбирането на кода изисква специфична поддръжка на инструменти за интерпретация на мащабни софтуерни програми. В тази статия преглеждаме и категоризираме инструментите за разбиране на кода по характеристика, архитектура и функционалност, за да проучим техните възможности. Представяме CodeCompass, който предлага широк спектър от функции за визуализации, предоставяне на информация, контрол на версиите и събиране на документация, показатели и др.

Източници

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

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

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

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

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

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

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

Шаблони за функционално програмиране за HPC

Функционалните езици за програмиране предоставят инструменти и функции за проектиране и внедряване на разпределено приложение. Тъй като функционалните програми имат внедрени паралелни функции, те могат, чрез високо ниво на разпределение и координация, да се използват за получаване на надеждна паралелна обработка.
Съвременната паралелна разработка на софтуер екстензивно използва подходи и методи за постигане на висока скорост. Координирането, обаче, остава една от най-трудните области, особено при интегриране на функционални подходи за програмиране.
Основната цел е да се изследват паралелните изчислителни шаблони в нова среда, да се илюстрира целесъобразността и приложимостта на fp в новите разпределени изчислителни настройки. Тестван е набор от известни паралелни алгоритмични шаблони като HPC компоненти. Значителен брой примери потвърждават висока скорост. Размерът на паралелизма винаги зависи от много фактори като: прилагания модел на изчисление, усъвършенствана подробност, семантика на разпределени възли и поточно предаване на данни. Примерите проверяват както добре моделираната координация, така и семантичната здравина.

Приложни области за шаблони

Координация

Специфичната тема на разпределените и паралелни функционални изчисления изисква координационни езикови елементи. Изследваните въпроси, разгледани по-горе, засягат това как може да се постигне паралелно работа и комуникация на функционалните програми на по-високи нива. Въведените функционални елементи на езика за програмиране с по-високо ниво на абстракция се оказват приложими за паралелни изчисления с интензивни данни на високо ниво [2]. Следващите указания водят до увеличаване на силата на езиковите елементи за паралелизъм във функционалните програми. По-конкретно, тя проведе дизайна на езиково разширение DClean за разпределеното програмиране и координиране на функционалните програми Clean. Разширението се състои от езикови елементи на високо ниво, координиращи чисти функционални изчислителни възли в проектираната разпределена среда на клъстери. Конструкциите генерират изчислителни полета, свързани чрез буферирани комуникационни канали. Използването на програмисти от DClean показва как разпределеният изчислителен модел е организиран в генериран разпределен график и те контролират потоците от данни в процеса на мрежата по търсени канали. Езикът предлага предимството да се пишат разпределени и функционални приложения, без да се изисква предварително запознаване с подробности за многопластовата среда и техническите аспекти на услугите на междинен софтуер. Разпределението на работата се извършва по предварително зададена паралелна изчислителна схема, алгоритмичен шаблон, параметризиран по функции, типове и входни потоци.Основната цел на въвеждането на езика за координация е да се определят функционалните паралелни изчислителни шаблони. В голям брой примери е получена висока скорост на паралелизъм. Действителният обем на паралелизъм следва реда за създаване на канали, обема на работа, скоростта на извличане и съхраняване на данни, както и сложността на възлите [1].

Графичната визуализация на разпределените изчисления чрез поддръжката на изпълним инструмент за разбиране на кода на семантиката [4] е незаменима в реалните разпределени приложения. Това изобразява очаквания паралелизъм на кутии и канали, генерирани с помощта на добре дефинирани шаблони на високо ниво. Паралелизмът е насочен към моделиране и формулиране на свойства на оперативната семантика на DClean [3].

Кибер физически системи

Подходът за функционално моделиране, базиран на шаблона, използван от езиците за координация, се прилага и за съвременните прототипи на CPS системи. Изучаването на връзките между CPS и разпределените системи, или CPS и вградените, системи са важни за вземане на адекватни решения в стъпките на проектиране и моделиране на прототипа на сложните CPS системи.
Прилаганите казуси на системата на CPS [5] описват сътрудничещите изчислителни единици, контролиращи физическите образувания (сензори) и връзките с други сложни системи. Системата Cartthouse CPS установява нови аспекти, характеристики и подходи в общото прототипиране, използвайки шаблони. При такова проектиране на системата на CPS важните семантични въпроси са адресирани от вероятностни и поведенчески гледни точки, където оперативната съвместимост е основна характеристика за анализ и уточняване.

Шаблони за изчисления с висока производителност

Изследванията за разширяване на приложимостта на шаблони във високоефективна изчислителна среда е ключовият момент в паралелния fp подход.
Адаптирането на по-ранното ноу-хау за шаблонното програмиране на разнородни многоядрени системи увеличава скоростта, при които измерванията и сравненията показват новите процеси на паралелизация.

Шаблонните прототипи се определят по отношение на техните функционалности и координати. Казусите илюстрират връзките с друг тип разпределени системи, които са важни поради многопластовата им структура. Свойствата на разпределената система, дадени по изпълними начини, се тестват от шаблони на функционално и разпределено програмиране на клъстери и мрежи.

Източници

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

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

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

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

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