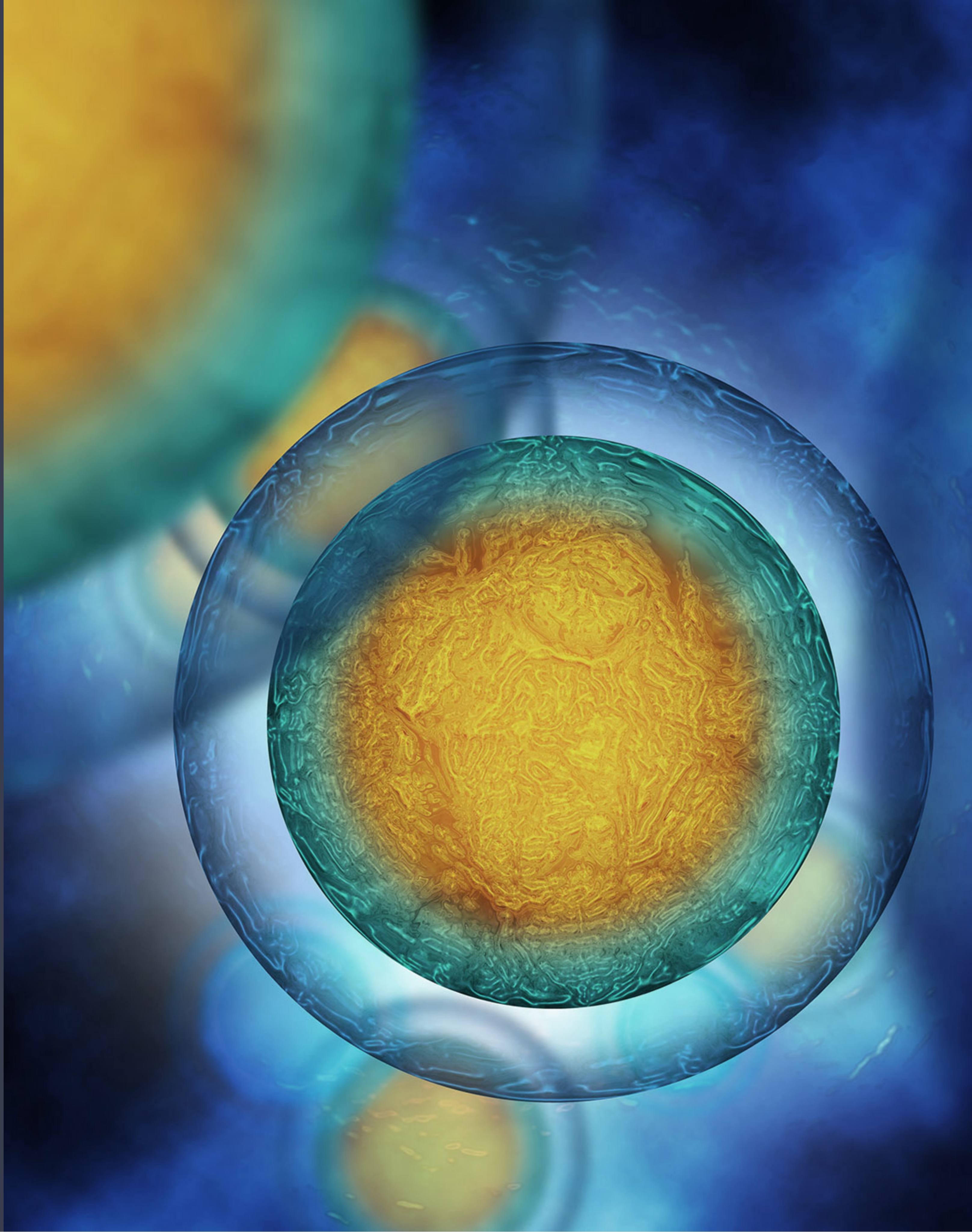
FE3CWS

THE THREE "CO" WINTER SCHOOL MATERIAL

Intellectual output 1 of the ERASMUS+ project 2017-1-SK01-KA203-035402



Some words about the

CONTENTS

- 9 topics related to software composition, comprehension and correctness
- 10 authors from 7 European universities from Croatia, Hungary, Netherlands, Portugal and Slovakia
- Available in 7 languages: English, Hungarian, Slovak, Croatian, Romanian, Bulgarian and Portuguese

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

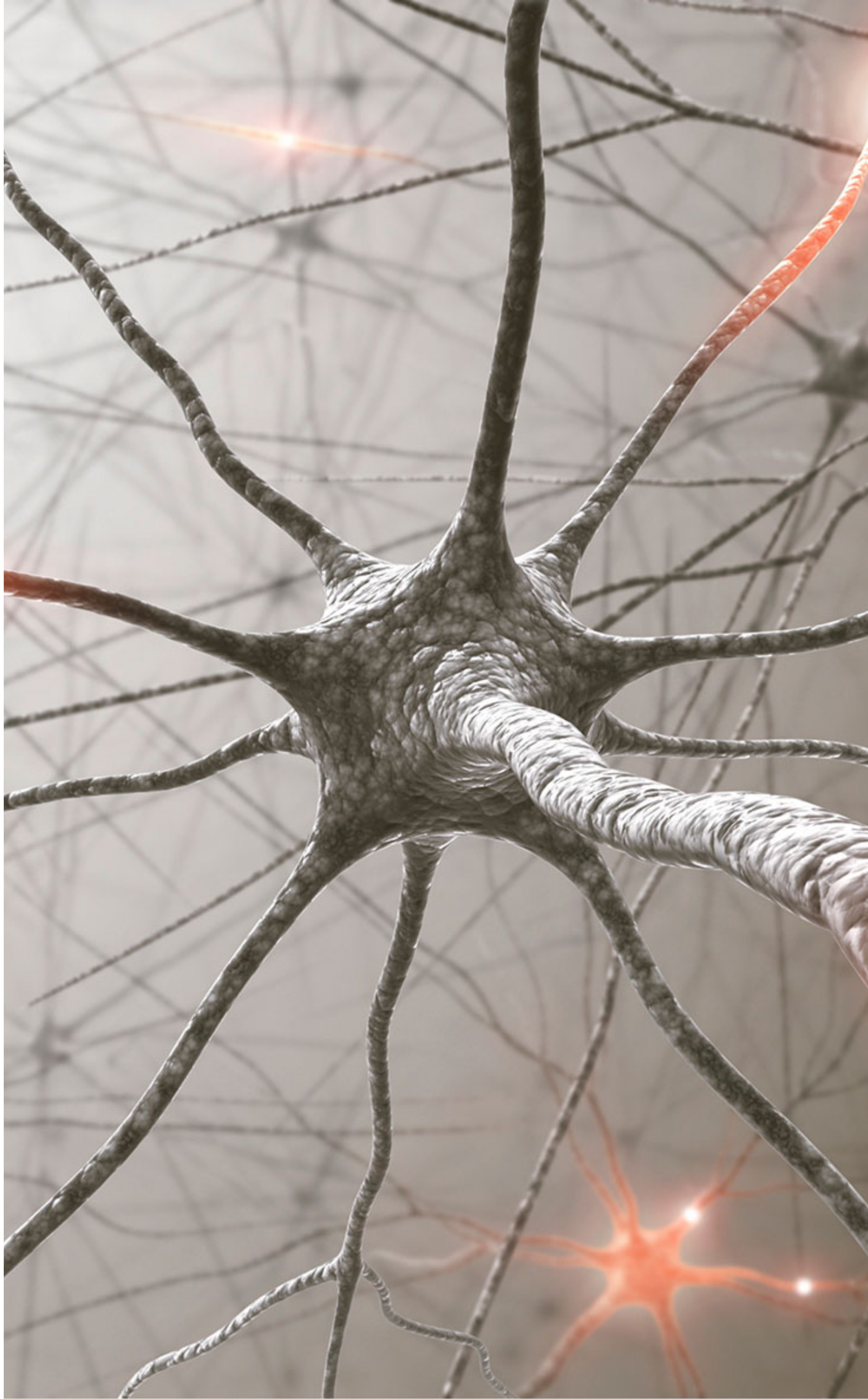
The three "CO" (Composability, Comprehensibility and Correctness) Winter School (3COWS) is the first intensive programme for higher education learners and teaching staff extending the community of the Central European Functional Programming (CEFP) summer school in the frame of the ERASMUS+ project No. 2017-1-SK01-KA203-035402 "Focusing Education on Composability, Comprehensibility and Correctness of Working Software" that was held between 22 and 26 January 2018.

The included material was created and presented in the frame of the above mentioned project. This publication is the print-formatted version of the intellectual output O1 of the project.

The information and views set out in this publication are those of the author(s) and do not necessarily reflect the official opinion of the European Union. Neither the European Union institutions and bodies nor any person acting on their behalf may be held responsible for the use which may be made of the information contained therein.

TABLE OF CONTENTS

1. Acquisition of Natural Language by Machines
2. Static Code Analysis with CodeChecker
3. Programming in Management and Orchestration of Virtualized Network Resources
4. Cloud Computing and Functional Programming in Education
5. Modern type-safe embedding of attribute grammars
6. How Green Is Your Process?
7. Functional Programming of Devices
8. Code Comprehension with CodeCompass
9. Functional Programming Skeletons for High-Performance Computing



ACQUISITION OF NATURAL LANGUAGE BY MACHINES

Construction of thinking machine is a great challenge. The relationship of language and mind is interesting for computer scientists a it is discussed in wider context also by historians, psychologists, linguists and philosophers.

For example, Renfrew [31] characterizes human as a symbolic being considering that the natural language development is the result of symbols exchange in communication. Gardenfors [14] states that the language concepts have a hierarchical structure. Grammatical evolution [24] and genetic evolution of languages [15] impact the problem of structural expansion, hence, they are applicable just for simple languages in a deterministic manner. It turns out that genetic strings in the role of the values of semantic conclusions is more promising approach to the language evolution than the straightforward application of the genetic structure of the human brain by an analogy. We also recognized that the process of evolution takes place at different language meta-levels.

Chomskys hypothesis on the existence of an universal grammar for all natural languages [7] is very interesting and motivating. But maybe the universal grammar should not be understood like a fixed and parametrized grammatical structure, but rather as an universal algorithm, i.e. deterministic progress at a meta-language level parametrized by the parameters that represent meaningful, i.e. semantic values. Edelman [12] states that the machine mind has language substance and is symbolic and at the same time computational. It follows that the mind is not just symbolic but also a dynamic process, which expresses ceaseless change of language. And finally, in Shaumyans applicative universal grammar [34], language is also expressed in terms of dynamic, even applicative processes.

The benefit of semiotic language theory is that syntax and semantics are mutually bound in each time, and they are not separable. Of course, this invokes the need for taking into account semantical categories to express the change of a language.

In our opinion, semiotic theory of languages provides an opportunity for deterministic evolution of the machine mind for the case of subsets of natural languages and formal languages in different forms used in communication. In this sense Human-Computer and Computer-Computer communication based on languages can be unified. Some starting point of this idea, restricted to regular languages we present in [17]. Currently we know the algorithm for the transformation of language concepts to the internal language of the machine mind and we are also able to derive concepts from this internal language. This internal language is an analogy to internal language of human thinking – it is a calm language behind the loud natural language. At present, we do know neither conceptualization rules nor the rules on reasoning on concepts represented in the machine mind. However, we have clear methodology of the evolution of the machine mind, represented by applicative dynamic processes with high degree of parallelism that represent information of language substance non-redundantly. Moreover, increasing the abstraction of language concepts decreases the number of meta-operations and increases the number of applicative bindings.

We discuss the efficient algorithm of child thinking and we estimate information flow speed in human brain to 300 trillions signals per second. Then we introduce Slovak textual and fonetic grammar as well as the method to translation of text to voice. Further, we illustrate the process of acquisition of language elements of different abstraction reading short sample of text, and using hierarchical hash tables as a model. We also evaluate acquisition process for massive text of selected book and conclude that language acquisition with hierarchical abstraction yields non-redundant graph with the same number of sequential and parallel bindings without the need for physical store of data.

Next, we use grammatical approach to communicated visual objects recognition. First, we need to figure out how to describe the objects and then we can apply the method of abstraction to these data. We primarily focus on 3D object description using grammars. In the grammar theory, this step is called symbolization.

The symbolization ensures an object description and provides the fundamental data abstract layer. As we can see, data abstraction using functional language allows us to abstract and to process objects easily.

Applicative approach can be used in language processing even when we use context-free grammars. We show an algorithm that is able to transform any context free grammar into supercombinator form. The resulting form depends on the form of an input grammar, therefore a new problem arises: finding the proper grammar for the task at hand.

We briefly show the resulting supercombinator forms of various grammar types and compare their properties. We also compare the algorithm efficiency in presented grammar cases and show that our algorithm can be improved in case we use grammars without any cycles. We also discuss the resulting supercombinator forms in term of grammar compression and re-usability of elements that are the end result of processing larger scale texts as input samples.

References

[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)

STATIC CODE ANALYSIS WITH CODECHECKER

OVERVIEW

Symbolic execution [4] is a popular static analysis technique used both in program verification and in bug detection tools. It works by interpreting the code, introducing a symbol for each value unknown at compile time (e.g. user-given inputs), and carrying out calculations symbolically. The analysis engine strives to explore multiple execution paths simultaneously, although checking all paths is an intractable problem, due to the vast number of possibilities.

While a rich literature exists on program verification tools, error finding tools normally need to settle for survey papers on individual techniques [1]. In this paper, we not only discuss individual methods, but also how these decisions interact and reinforce each other, creating a system that is greater than the sum of its parts. We focus on an error finding tool called the Clang Static Analyzer [2] (hereafter referred to as the Analyzer), and an infrastructure built around it named CodeChecker [3]. The emphasis is on achieving end-to-end scalability.

This includes the run time and memory consumption of the analysis, bug presentation to the users, automatic false positive suppression, incremental analysis, pattern discovery in the results, and usage in continuous integration loops. We also outline future directions and open problems concerning these tools.

Although the Analyzer can only handle C/C++/Objective-C code, the techniques introduced in this paper are language-independent and applicable to other similar static analysis tools.

CLANG STATIC ANALYZER

We summarize the working mechanism of symbolic execution and its implementation in the Analyzer. We discuss its memory representation [6], its handling of the bindings between values and memory locations, and its representation of check-specific states (where by check we mean one module of the Analyzer written to find one specific type of bug). We also introduce the concept of symbolic calculations. The choices of representations used by the Analyzer play a crucial role in making large-scale software analysis viable.

Since checking all possible execution paths is not possible in a reasonable amount of time, we need to introduce the concept of the analysis budget: an estimate of the time-span we can afford to analyze a given chunk of code. The goal is to find as many bugs as possible with a low false positive rate. We show how the Analyzer prioritizes more interesting paths for analysis, and how it eliminates infeasible paths in an efficient way using tiered constraint solving [5].

The Analyzer also employs a number of heuristics to automatically suppress reports that are likely to be false positives.

When a bug is found, the corresponding path and set of constraints are useful to understand the problem. It is, however, impractical to present all this information to the user. We show how the analyzer strives to present the user with a concise yet insightful error report that minimizes the time to fix said error.

CODECHECKER

We define the scalability of static analysis not only in terms of efficient use of computing resources, but also in terms of efficient use of human resources like developer time. CodeChecker is a tool designed to ease the integration of the Analyzer and other similar static analysis tools into build systems and continuous integration loops. It is also a full-fledged bug management system that keeps track of errors found by these tools.

Given a finite budget of developer time and thousands of reports on large software, it is important to evaluate reports with the best return on investment at first.

CodeChecker also supports differential analysis that prevents developers from introducing new bugs without requiring them to fix all legacy reports beforehand.

SUMMARY

In this paper we summarize our experiences collected while contributing to the state-of-the-art Clang Static Analyzer and CodeChecker products. Our hope is that it will prove to be a useful resource for anyone who decides to work on static analysis tools.

References

[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.

PROGRAMMING IN MANAGEMENT AND ORCHESTRATION OF VIRTUALIZED NETWORK RESOURCES

Network Functions Virtualization (NFV) is a new paradigm for changing the way networks are built and operated. Decoupling soft- ware implementation from network resources through a virtualization layer introduces a need for developing sets of NFV management and orchestration (MANO) functions. We focus on coordinating management functions implemented within different functional blocks to accomplish reliable operation for MANO functions operating in distributed environments. The challenges are illustrated on a practical example on Open Stack virtual technology and on the problems inspired by telecommunication industry.

INTRODUCTION

The purpose of these lecture notes is to introduce the new concepts, such as Network Functions Virtualization (NFV), that are currently implemented within complex software systems and networks, explain the new challenges arising and show students how to deal with them using the programming techniques for coordination of management and orchestration functions of virtualized network resources operating in distributed environments.

COMPLEX SYSTEMS

The setting of these lectures is within the theory of complex systems, in particular, the complex software systems and complex networks. Hence, the lectures start with a gentle introduction to the theory, carefully positioning the considered problems and challenges within the current evolution of networks and software systems. Virtualization is a paradigm frequently used in management of complex software systems. It implies introduction of a new abstract layer, a virtual edition of system layer and its functions, which avoids introducing dependency between system layers.

MANAGEMENT AND ORCHESTRATION FUNCTIONS

In telecommunication networks a new paradigm is introduced, called Network Functions Virtualization (NFV), that decouples network function from physical network resources through a new virtualization layer [2]. However, this introduces a need for developing sets of NFV management and orchestration functions (MANO). For this purpose a special working group is defined within the European Telecommunications Standards Institute (ETSI). The network function virtualisation management and orchestration architectural framework is defined in [1]. In these lecture notes, we focus on the management and orchestration functions implemented in different functional blocks, in order to accomplish reliable operation for management and orchestration functions operating in distributed environments.

EXAMPLES

These notes provide an introduction to the subject, with the goal of explaining the problems and the principles, methods and techniques used for their solution. The worked examples and exercises serve students as the teaching material, from which they can learn how to use functional programming to effectively and efficiently coordinate management and orchestration functions in distributed complex systems using NFV.

The methods and techniques explained in these lecture notes, and applied to the problems of management and orchestration of network virtualization, are already existing and we claim no originality in that sense. The purpose of these notes is to serve as a teaching material for these methods.

The problems and challenges of coordination of management and orchestration functions are addressed using the OpenStack platform [3]. It is an open source cloud operating system which integrates a collection of software modules that are necessary to provide cloud computing layered model. Such technology is necessary in dealing with problems arising from the virtualization paradigm in current networks, and the students understanding solutions in OpenStack will be able to transfer their knowledge to other existing technologies with the same or similar purpose.

The challenges arising from the new network paradigms, as well as their solutions, are illustrated through practical examples using OpenStack virtual technology and inspired by the problems from the telecommunication industry. All examples and exercises are worked out in OpenStack virtual technology.

References

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

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

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

CLOUD COMPUTING AND FUNCTIONAL PROGRAMMING IN EDUCATION

Cloud computing has become a key technology and therefore part of many computer science curricula. In application design, mi- croservices are a key concept. The functional decomposition intrinsic to microservices could be well served by serverless platforms, such as AWS Lambda.

MICROSERVICES

More and more practitioners recommend adopting a microservice architecture when designing cloud applications [1,2]. Vaguely defined, the microservice architectural style stands a common set of characteristics: automated deployment, smart endpoints dumb pipes, decentralized control of data [2]. Recent efforts to migrate traditional cloud applications to a microservice architecture warns of the increased complexity when managing the many, albeit small, composing services [3]. Here, it is important to disambiguate: we will focus on orchestration as the business logic composition of fine-grained cloud services. Whether their deployment is orchestrated (central component governing the execution) or choreographed (each service acts independently) boils down to whether the desired application needs synchronous control or may tolerate asynchronous control. However, given that agility and flexibility are highly desired attributes, a choreographed deployment is preferable [1].

However, non-functional aspects of microservices, such as runtime performance, play an important role in the application deployment.

We therefore plan to familiarize students with the concept of performance profiling in the cloud in the context of AWS Lambda. Furthermore, we complement the functional paradigm of AWS Lambda by using a Haskell framework to control the experiments.

USER-CENTRIC CLOUD COMPUTING

Within the area of cloud computing an important aspect is assisting users in their decisions. Such decisions speak to the following questions:

A. How does the application behave on the virtualized resources?

B. How many virtual resources of what type from which cloud provider should be acquired for application deployment?

C. For how long? How much will it cost?

These questions are typically modeled as a scheduling problem, working under the assumption that there is no a-priori knowledge about the application. A simple set of requirements is that the application is successfully deployed, and the costs are minimized.

The architecture of a scheduler for a cloud application

The BaTS scheduler [4] has been developed to assist users when deploying their applications to the cloud. It takes a self-scheduling approach to achieve this and it regularly checks the deployment progress. In a first stage, BaTS collects statistics from sampling with replacement. Here, only a small sample is needed (30-50 tasks) to compute the mean and the standard deviation of the tasks' runtime on various cloud offerings. The budget estimation module then performs linear regression to optimize this phase.

Using the BaTS methodology on lightweight virtualization

We introduce students to the problem of assisting application owners looking to select the best choices in terms of lightweight virtualization when deploying their application as a set of microservices.

AWS Lambda

The AWS Lambda is a highly light-weight virtualized compute resource offered by Amazon. The granularity is function level and the recommended runtime per function invocation is at most in the order of seconds. It also assumes a non- blocking behaviour. There are 46 types of AWS Lambda with a pricing model of euro per GB*sec.

AWS API Haskell implementation

Based on a comprehensive Haskell implementation of the AWS API [5], we aim to replicate the BaTS estimation methodology on lightweight virtualized resources.

Practical work

We instruct the students to relate the performance of running the factorial on various lambda types by using sampling and linear regression to plot throughput versus price efficiency curves. For instance, they should consider which type has the best throughput for the cheapest price. Next, they should identify how to efficiently find the most profitable combination: smallest cost for best performance.

Benchmarking AWS Lambda

The AWS Lambda functions run in a container-like virtualized environment. The amount of compute resources allocated to the functions is known to be proportional with the user-requested DRAM memory. To determine how the Lambda functions can process various workloads, we can benchmark each of their computational resources independently: CPU, memory bandwidth, I/O bandwidth, and network bandwidth and latency. Such microbenchmarks can be performed by launching, within the functions, well known compute-, memory-, I/O-, and network- intensive workloads, such as computing the first N prime numbers, running the stream benchmark, running the iozone benchmark, or reading or writing data to AWS S3.

Transfer of established methodologies is key to education. As future work, we would like to support Haskell AWS Lambda functions deployed through the Haskell AWS API implementation.

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

MODERN TYPE-SAFE EMBEDDING OF ATTRIBUTE GRAMMARS

Attribute grammars are a powerful, declarative formalism to implement and reason about programs which, by design, are conveniently modular. Although a full attribute grammar compiler can be tailored to specific needs, its implementation is highly non-trivial, and its long- term maintenance is a major endeavor. In fact, maintaining a traditional attribute grammar system is such a big effort that most systems that were proposed in the past are no longer active. Our approach to implementing attribute grammars is to write them as first-class citizens of a modern functional programming language. We improve a previous zipper-based attribute grammar embedding making it non-intrusive (i.e. no changes to the user-defined data types are required) and type-safe. On top of that, we achieve clearer syntax by using modern Haskell extensions. We believe that our embedding can be employed in practice to implement elegant, efficient, and modular solutions to real-life programming challenges.

INTRODUCTION

Attribute Grammars (AGs) are a declarative formalism which was proposed by Knuth [7] in the late 60s and allows one to implement and reason about programs in a modular and convenient way. A concrete AG relies on a context-free grammar to define the syntax of a language, and on attributes associated with the productions of the grammar to define the semantics of that language. AGs have been used in practice to specify real programming languages, like for example Haskell [2], as well as powerful pretty printing algorithms [16], deforestation techniques [4], and powerful type systems [11]. When programming with AGs, modularity is achieved due to the possibility of defining and using different aspects of computations as separate attributes.

Attributes are distinct computation units, typically quite simple and modular, that can be combined into elaborated solutions to complex programming problems. They can also be analyzed, debugged, and maintained independently which eases program development and evolution.

AGs have proven to be particularly useful to specify computations over trees: given one tree, several AG systems such as [14,3,8,17] take specifications of which values or attributes need to be computed and perform these computations. The design and coding efforts put into the creation, improvement, and maintenance of these AG systems, however, is tremendous which is often an obstacle to achieving the success they deserve.

An increasingly popular alternative approach to the use of AGs relies on embedding them as first-class citizens of general purpose programming languages [12, 9, 13, 15, 18, 1]. This avoids the burden of implementing a totally new language and associated system by hosting it in state-of-the-art programming languages. Following this approach one then exploits the modern constructions and infrastructure that are already provided by those languages and focuses on the particularities of the domain specific language being developed.

Functional zipper [6] is a powerful abstraction which greatly simplifies the implementation of traversal algorithms performing a lot of local updates. Functional zippers have successfully been applied to construct an AG embedding in Haskell [9,10]. Despite its elegance, this solution had a major drawback which prevented its use in real-world applications: attributes were not cached, but rather repeatedly recomputed which severely hurt performance. Recently, this flaw has been eliminated [5] and replaced with a different one: the approach be- came intrusive, i.e. to benefit from the embedding user-defined data structures have to be adjusted.

In this paper we present an alternative mechanism to cache attributes based on a self-organising infinite grid. This graph is laid on top of the user-defined algebraic data type (ADT) and mirrors its structure. The used-defined data type itself remains untouched. The embedding is then based on two (rather than one) coherent zippers traversing the data structures in parallel. On top of being non- intrusive our solution is completely type-safe. Modern Haskell extensions such as ConstraintKinds allow us to propagate constraints down in the ADT completely eliminating runtime type casts present in the previous versions.

Another side benefit of using modern Haskell features is a cleaner syntax with less code being generated by means of Template Haskell.

References

[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)

HOW GREEN IS YOUR PROCESS?

Like with Bio products, the world is developing to become a more nature-aware ecosystem. The green initiative defines two main goals: reduce energy consumption and use basic natural sources in electrical energy production.

One of the challenges for battery manufacturers is how long the battery can operate without being continuously charged. There are many other challenges as well, such as the size that greatly affects the shape and weight of the device. The battery is considered to be somewhat lighter compared to the device that needs a battery to operate. The challenge here is how to make its size smaller and lighter weight and certainly a high efficiency in terms of operating time of the mobile device without being charged.

On the top of this hardware challenge, a software challenge exists: the soft- ware itself should support energy savings. Doing that without limiting the user experience is considered nowadays as a silent but important goal of each software development that is targeting any kind of portable devices [1]. This goal usually emerges when the related requirements change from untold to important issue.

Remembering that the energy consumption of any mobile device is influenced by the running applications through the usage frequency of services up to the actual mood of the user, to develop software for such devices is already a challenge [4]. One could say, the software development challenge is always the same, but we have to point out "mobility" as key system property here. Battery power status also determines the system performance due to operating system level configuration – well known as "energy saving preferences".

It is even harder, if one (in our case the teacher) has to prepare students for such challenges [6]. All the already known "best practices" and "energy saving tips" have to be presented in a context, which is easily comprehensible to the students.

This can be done by positioning the concepts into a known environment such as software testing and test automation [2]. This is what we aim with this tutorial, this is the content of the upcoming sections, starting with the proposal followed by examples and closing with further tips on improvement.

The main focus is on energy consumption of working software and its development processes, where each development phase plays a significant role.

Considering any software development process, the energy is being consumed while problem analysis, constructing and evaluating the code as well [3]. Software or hardware tools have to be used to implement energy consumption monitoring for software run at the top of selected operating systems and for evaluation of the energy consumption. Usual usage scenarios are to monitor energy usage of selected software [5], but we will also take a look at the possibility to use these tools to measure how green is the process that produces the final version(s) of working software.

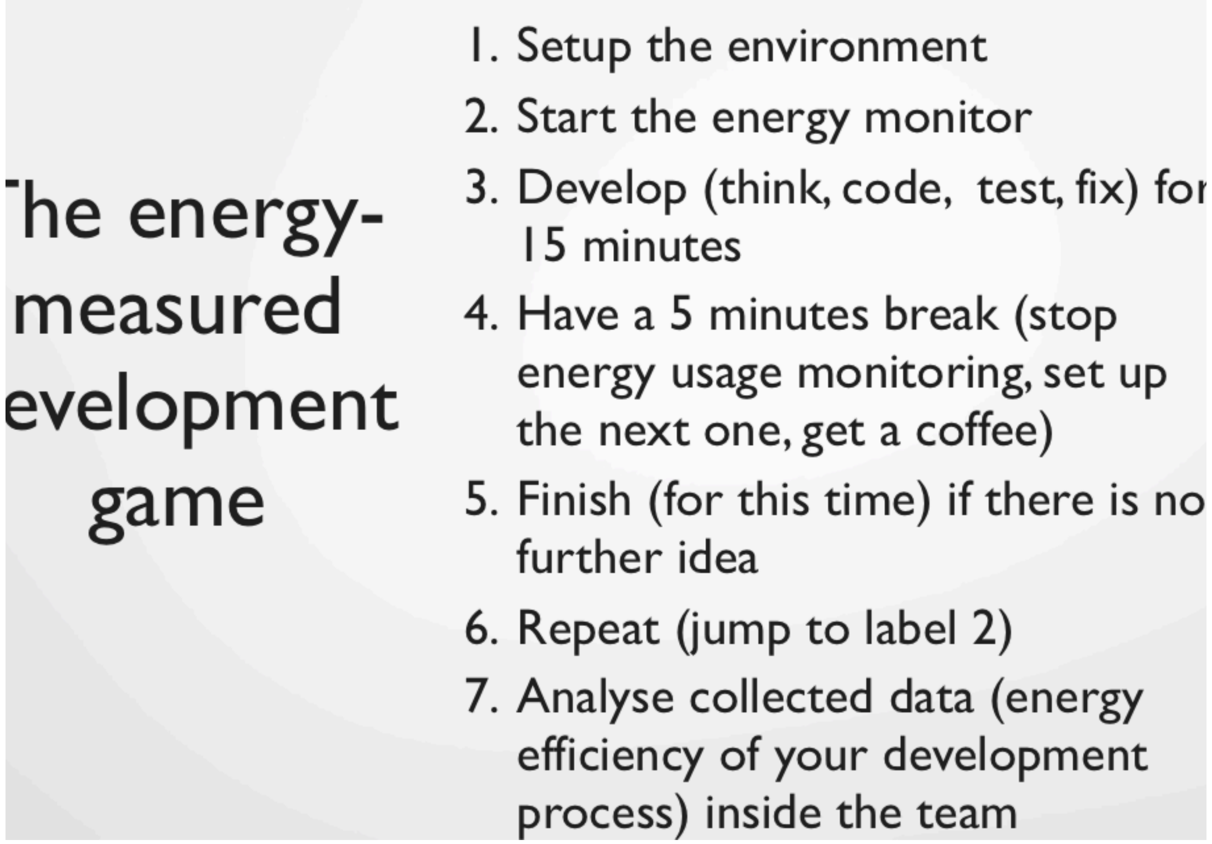
The examples cover a variety of situations. Starting with the case of energy profiling of third-party working software in specific usage scenarios, we point out key properties (advantages and disadvantages) of existing tools. During unit testing of code of software being developed, we present typical usage of energy profiling software.

The ability to scale the measurement approach from profiling a code snippet or single application to energy consumption analysis of tool-chains is the last example we present. This one is presenting a generic approach for energy profiling and is aimed to replace the question mark of the tutorial title by a period representing the result evaluation for each specific case covered by the tutorial.

References

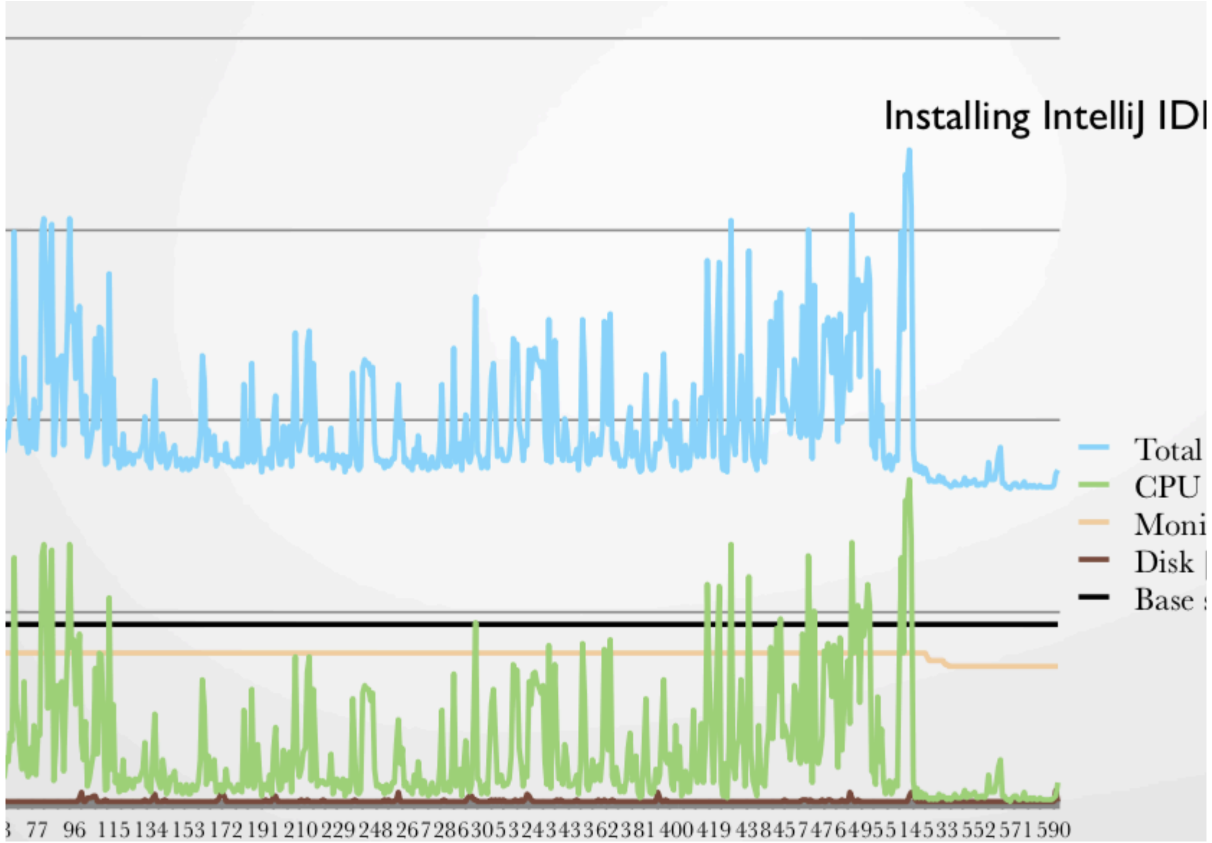
[1] D. Li, W. G. J. Halfond, An investigation into energy-saving programming practices for android smartphone app development, in Proc. of the 3rd International Workshop on Green and Sustainable Software, GREENS 2014, ACM, 2014, pp. 46–53.

[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.



he energy-measured evelopment game

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



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

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

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

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

FUNCTIONAL PROGRAMMING OF DEVICES

NOTICE

This paper will be an extended version our RWDSL18 contribution [2]. In the current paper we will use the same DSL, only some minor extensions and improvements are done. The current paper will focus on how the mTask DSL can be use to program the IoT and will discuss a high-level simulator for mTask programs as an iTask program. on to simulate .

INTRODUCTION

Many devices are nowadays equipped with a simple microprocessor to control their behaviour. Typical examples are thermostats, light bulbs, electric sockets, fire alarms, door openers and so on. When these devices can communicate with each other, or some remote computer, they are said to be part of the Internet of Things, IoT. The microcomputers in these devices are very affordable and becoming omnipresent. Expensive devices like cars and apparatus with a very complex task are equipped with a full-fledged embedded computer and appropriate software. For most small and relatively cheap IoT devices such an embedded computer is too expensive, or consumes too much energy; a simple and cheap microprocessor is used to execute the software. These systems have very limited computing power and memory, typically 30 KB to 4 MB flash memory to store the program. The life of this memory is restricted to 1000 write cycles. To store variables, the heap and the stack the systems have 2 to 40 KB of RAM.

The processor speed and memory limitations exclude the use of an operating system. The apparatus just executes the program controlling the device. Even the control programs on these IoT devices consist of several tasks. For instance, to check the state of a button ten times a second, to update a display every second, to measure the temperature two times a minute, and to switch the heating after at least five minutes unless the button is pressed earlier. Due to the different time frames and the dependencies of these tasks, the control program tends to become rather messy, independent of the programming language used. Moreover, the IoT devices execute separate programs for the rest of the application in the IoT and the communicate using a plethora of protocols. This makes the development and maintenance of IoT applications complex and error prone.

Task Oriented Programming, TOP, offers lightweight threads that can easily be composed to more complex tasks. Tasks are evaluated step-by-step and can inspect the current value of other tasks after such a step. TOP is first implemented in the iTask system [4,5] embedded in Clean [6]. In the iTask system, primitive tasks are gathering input via automatically generated web-form or by collecting data from other programs and data stores. A powerful set of combinators is used to compose tasks to more complex tasks. In this paper, we show that TOP is very suited for programming IoT devices. Primitive tasks deliver the current value of inputs and sensors. Constructors very similar to the iTask system are used to combine tasks to more complex tasks.

IoT devices typically have loosely dependent tasks that control the sensors, actuators and communication of the devices. Programming this in a TOP style offers concise programs. Executing these tasks within the constraints of small microcontrollers with very limited processing power and some KBs of RAM memory deserves some thought. Due to the severe limitations of the microcontrollers used we cannot port the iTask system to the IoT devices since a typical iTask program requires about 100 MB of heap space. We define an embedded Domain Specific Language, eDSL, called mTask for the IoT devices. This eDSL is embedded in the iTask system since we plan to make these TOP languages fully interoperable.

The contributions of this paper will be:

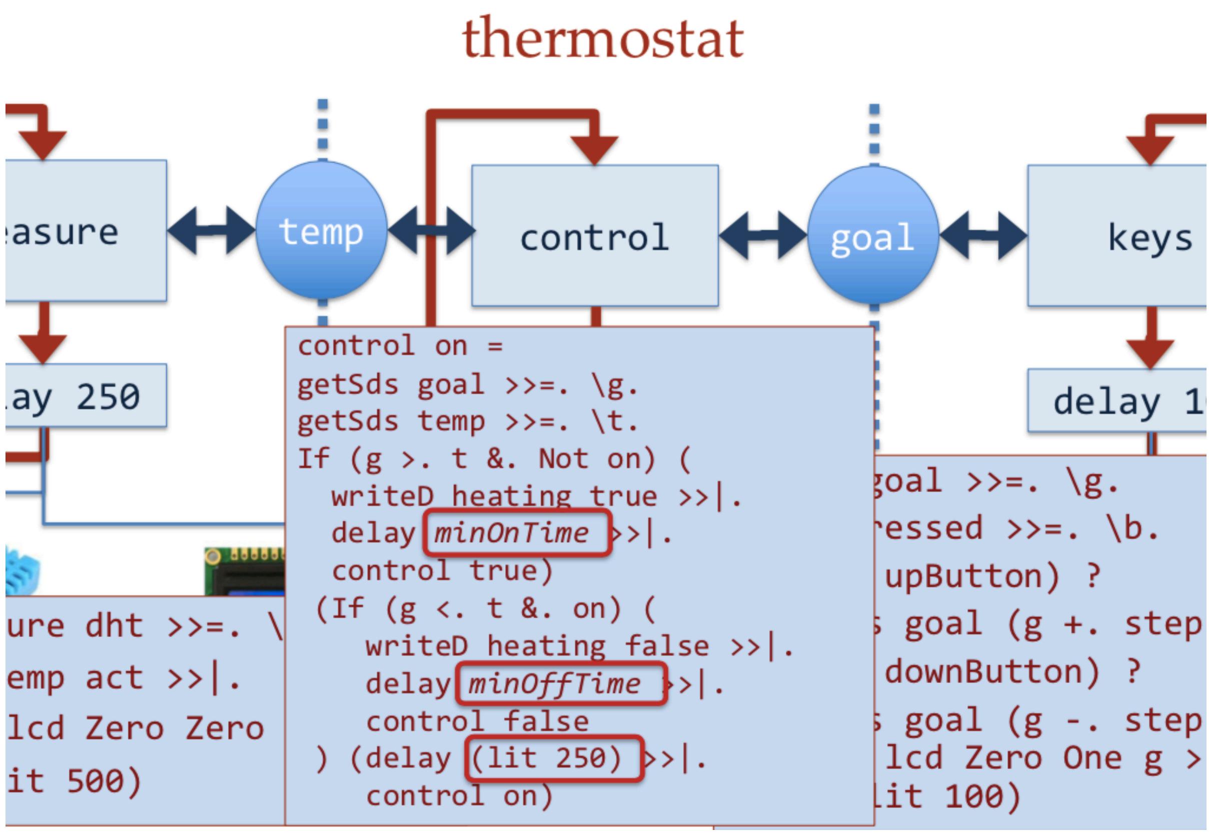
- This paper introduces a task-based functional programming language for IoT devices. Compared with our previous language for microprocessor programming [3] the imperative peripheral control is replaced by referential transparent constructs.
- We demonstrate how make a functional extendable, multi-view, type-safe, embedded DSL. This is a tag-less eDSL [1].
- The generated code runs on small and slow devices as well as on bigger machines and a simulated machine.
- Due to the use of Arduino C++ as the intermediate language, this functional eDSL runs on many different microcontrollers.
- The high-level simulation of mTask programs in an iTask program offers the possibility to view the effect of the eDSL program and to manipulate the simulated environment to experiment with the specified behavior. In such a simulator it is much easier to manipulate the time and the sensors than in a real-life setup, e.g., we can change the temperature reported by a sensor by the push of a button instead of physically expose the IoT device to those temperatures.

References

[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

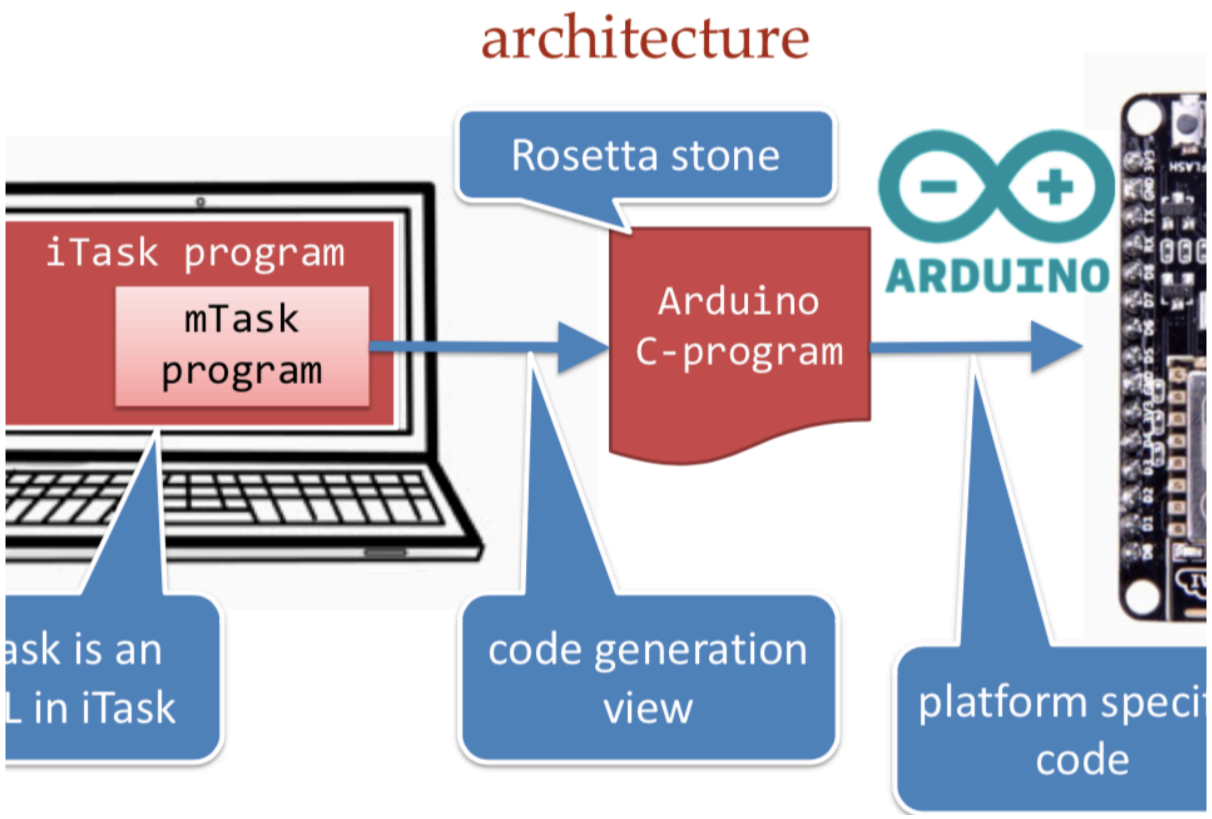
[3] Koopman, P., Plasmeijer, R.: A shallow embedded type safe extendable dsl for the arduino. In: Revised Selected Papers of the 16th International Symposium on Trends in Functional Programming - Volume 9547. pp. 104–123. TFP 2015, Springer-Verlag New York, Inc., New York, NY, USA (2016), http://dx.doi.org/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


thermostat

```
control on =
getSds goal >>=. \g.
getSds temp >>=. \t.
If (g >. t &. Not on) (
  writeD heating true >>|.
  delay minOnTime >>|.
  control true)
 (If (g <. t &. on) (
    writeD heating false >>|.
    delay minOffTime >>|.
    control false
 ) (delay (lit 250) >>|.
    control on)
```

```
ure dht >>=. \
emp act >>|.
lcd Zero Zero
it 500)
```

```
goal >>=. \g.
essed >>=. \b.
 upButton) ?
 goal (g +. step
 downButton) ?
 goal (g -. step
 lcd Zero One g >
lit 100)
```


architecture

[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

CODE COMPREHENSION WITH CODECOMPASS

OVERVIEW

Development and maintenance are two separate stages with differ- ent characteristics, thus they require different tooling support as well. During development we are mainly writing new code which requires tooling support like code completion, brace matching, etc. and usually only a few files involved more or less on the same abstraction level. During maintenance we are mainly reading and navigating through an existing code base among the large number of modules and files on different abstraction levels [1]. In development the intentions are clear, as opposed to code comprehension where the task is to recover the original purpose of certain code fragments.

In industrial environment [2] a project may consist millions lines of code. For long existing large systems, where the code base has been developed and maintained for decades by fluctuating teams, original intentions are lost, the documentation is untrustworthy or missing, the only reliable information is the code itself. Comprehension of such large software systems is an essential, but usually very challenging task. This implies that tooling support is needed [3].

When getting familiar with an unknown source code, the first step is to find the relevant parts of the system. This process requires fast feature localization, based on some named entities acquired from log messages or other resources. The next step is to extend our knowledge about the system through diagrams, function call chains, etc. And in the end we would like to verify the gathered knowledge through version control messages, architectural information and references on related modules.

CODECOMPASS

CodeCompass [4, 5] is an open source code comprehension frame- work. It provides a pluginable architecture in order to enable the addition of various analyzer tools which produce different visualizations, information collectors, metrics [6], etc. The most important design goal was to scale CodeCompass for large scale industrial projects.

In the first step the product has to be parsed: all the information is collected and stored to a database which then enables the service layer to provide the required visualizations. For fast searching CodeCompass uses text indexing that results language-independent navigation in the source code. As the primary tar- get is to give accurate information about language elements, identifying symbols by their name is not sufficient. We use the LLVM compiler infrastructure to identify symbols precisely, and to resolve named entities using the abstract syntax tree. CodeCompass is extended by parsers of languages. The most supported languages are C/C++, but Java and Python are partially handled as well.

Besides the named symbols some additional information is also stored in the database, such as relations between AST nodes (function calls, inheritance) and files (interface provider relation, inclusion, etc.). These are used for displaying an architectural level picture about the system based on the symbols' usage [7].

The code base is not the only source of documentations. The commit mes- sages of a version control system also contain information which is important to understand why certain changes happened at the given module. CodeCompass reads the Git repository too, if any. CodeCompass is also equipped with advanced functionalities. It can display the compiler generated functions, that are missing from the source. Pointer analysis helps to understand which variables are referring the same object. We can inspect the function call relations even if those invoked via a virtual function or a function pointer.

SUMMARY

Code comprehension requires specific tool support for understanding large-scale software. We overview and categorise code comprehension tools by architecture and functionalities in order to examine their capabilities.

We introduce CodeCompass which presents a wide range of functionalities about visualizations, information providing, version control and documentation collection, metrics, etc.

References

[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.

FUNCTIONAL PROGRAMMING SKELETONS FOR HIGH-PERFORMANCE COMPUTING

Functional programming languages provide tools and features for designing and implementing distributed application. As functional programs have inherent parallel features, it can be exploited to obtain reliable concurrent processing by high level distribution and coordination.

The state-of-the-art parallel software development made extensive usage of various methodologies and approaches to obtain high speed up. However, concurrency remains one of the most difficult domains especially in the case of functional programming approaches.

The main purpose is to explore parallel computation skeletons in a new environment, to illustrate the appropriateness and applicability of FP in novel distributed computation setups. A set of known parallel algorithmic skeletons are tested as HPC components. Significant number of examples provide high speed-up. The amount of parallelism always depends on many factors such as: the computation pattern applied, refined granularity, semantics of distributed nodes, data streaming. Examples inspect both the well modelled coordination and the semantical soundness.

APPLICATION DOMAINS FOR SKELETONS

Coordination

The specific topic of the distributed and parallel functional computation requires coordination language elements. The research questions addressed earlier were concerned about how the parallel behavior and the communication of functional programs can be obtained at higher levels. The introduced functional program- ming language elements with higher abstraction level proved to be feasible for parallel, high level, data intensive computations [2].

The followed directions resulted in increased expressing power of language elements for parallelism in functional programs. More specifically, it conducted to the design of a language extension DClean for the distributed programming and coordination of the Clean functional programs. The extension consists of high-level language elements coordinating pure functional computational nodes in the designed distributed environment on clusters. The constructs generate computation boxes connected via buffered communications channels. Using DClean programmers indicate how the distributed computation pattern is organized into a generated distributed graph, and they control data-flows of the process-network by typed channels. The language offers the advantage of writing distributed and functional applications without being acquainted with details of the multi-layered environment and middleware services' technical aspects. Work distribution is made according to a predefined parallel computational scheme, algorithmic skeleton, parameterized by functions, types and input streams.

The main purpose of introducing the coordination language was to define functional parallel computation skeletons. In a large number of examples high speed-up for parallelism was obtained. The actual amount of parallelism was subject to channel creation order, the amount of work on them, the speed of retrieving and storing data, and the complexity of the nodes [1].

The graphical visualization of the distributed computation by the support of executable semantics code comprehension tool [4] was indispensable in real distributed applications.

This depicted the expected parallelism on boxes and on channels generated using well-defined high-level skeletons. It aimed at modeling and formulating properties of operational semantics of the DClean [3].

Cyber physical systems

The skeleton based functional modeling approach used by the coordination languages is applied to the nowadays fashionable CPS systems' prototypes as well. Studying relationships between CPS and distributed systems, or CPS and embedded systems are important for taking adequate decisions in the design and modeling steps of the sophisticated CPS systems' prototype.

The CPS system case studies implemented [5] describe the collaborating computational units controlling physical entities (sensors) and the relationships with other complex systems. The smarthouse CPS system establishes novel aspects, features and approaches in a general prototyping using skeletons. In such CPS system design important semantics questions are addressed from probabilistic and behavioral viewpoints, where the interoperability is the specific, main feature to analyze and specify.

SKELETONS FOR HIGH PERFORMANCE COMPUTATIONS

Research on extending the applicability of skeletons in high-performance computing environment is the key point in the parallel fp approach. Adapting the earlier know-how about skeleton programming of heterogeneous multicore systems results in higher speed-ups, where the measurements and the comparisons evaluate the novel parallelisation processes.

The skeleton prototypes are defined in terms of their functionalities and coordinations. The case studies illustrate the connections with other type of distributed systems, which are important due to the multi-layered structure of them. The distributed system properties given in executable ways are tested by skeletons of functional and distributed programming of clusters and grids.

References

[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.